— Vx Tools Support Package

S. Kent
R. Rechenmacher
D. Slimmer
M. Vittone
G. Zioulas
Online Support

Fermilab

September 16, 1994

Abstract

The vx_tools support package is a conglomeration of generic and target specific routines disseminated into appropriate target directories.

Keywords: VME VxWorks vx_tools

This documentation was prepared with LaTeX.

Contents

Chapter 1

Vx Tools

1.1 Introduction

The vx_tools package is a conglomeration of useful code to supplement the VxWorks package. This package provide generic and target specific code that resides in a common user package. vx_tools depends on VxWorks product: two versions of VxWorks are currently supported , v5_0_2 and v5_1_1; to manage the co-existence of both versions two separate databases have been defined. To setup the desired version of vx_tools product, the user needs first to do the following:

```
% setup vxprods                  to setup the version built against
                                         VxWorks v5_0_2

or

% setup -n vxprods               to setup the version built against
                                         VxWorks v5_1_1
```

There are two flavors of this product. The M68k flavor provides a supplement package for the Motorola MC68000 series of processors. The R3k flavor provides a supplemental package for the LR3000 series processors.

```
% setup -f M68k vx_tools

or

% setup -f R3k vx_tools
```

Setting up the product, the environmental variable VX_TOOLS_DIR is defined, pointing to the product root directory, to help the user moving around in the directory tree.

The vx_tools directory structure is divided among a generic "all" subdirectory and various target specific directories. The "all" subdirectory contains generic source files, headers files, documentation files, and makefile scripts. This product contains a generic header file "vx_tools.h" that includes global definitions for NVRAM partitioning, and conditional include on one of the various hardware definition header files as defined in the compilation options of makefile.

The vx_tools directory structure is as follows:

```
        $VX_TOOLS_DIR/

                        all/                Generic
                                bin/
                                            IRIX/
                                            SunOS/

                                doc/        (eg. pn440.tex, pn440.ps)
                                include/    (eg. vx_tools.h)
                                lib/
                                src/        (eg. makefile, premake)

                        target/             Specific
                                include/
                                lib/
                                src/

                        ups/
```

The target subdirectory tree exists for example for the following targets:

- mv147
- mv167
- pc4a (for vx_tools versions greater then v1_4)
- pc4 (for vx_tools versions up to v1_4)

The $VX_TOOLS_DIR/all/include/vx_tools.h header file contains global definitions for the NVRAM partitions. The NVRAM is part of the battery backup calendar clock device that is common among the Targets. These devices include the DS1386 (Dallas Semiconductors), MK48T02, and the MK48T08 (SGS-Thomson). These devices provide from 2K X 8 to 8K X 8 bytes of NVRAM.

1.2 How to build the software

The user can either separately build different components of the vx_tools package or the entire package itself. The building is done basically through the use of three files: build, premake and makefile. Different parameters can be passed to allow for different platforms, CPU types and also to reflect which VxWorks version should be used to build against. The three files reside under the $VX_TOOLS_DIR/all/src area.

Some examples on how to invoke the build command follow:

The general syntax is:

```
                % build target vx_version

  where :
         target can be "all" to build the entire product, "tod" for time
                of the day etc.
         vx_version refers to the VxWorks version to use:
                v5_0_2b, v5_1_1 etc.

 example:
                % build sup v5_1_1
```

In this case build will invoke premake to build the supplement library for the MOTOROLA MVME167 running under version v5_1_1 of VxWorks. Then premake will in turn invoke make.

Since the large number of targets to build, the user should refer to the build,premake and makefile files for a complete and detailed description.

1.3 How to distribute the software

The vx_tools product is maintained on the OLS unix cluster and when new versions are released they are put in the fnsg01 kits database. When retrieving a new version the user/distributer should notice that in the kits database, the flavor is declared with the following conventions:

`VxWorksM68k_5.1.1`	`for the version built under`
	`VxWorks v5_1_1`

`VxWorksM68k`	`for the version built under`
	`VxWorks v5_0_2b`

`VxWorksR3k`	`for the version built under`
	`VxWorks R3000 for the IRIX.`

Part I

Generic Libraries and Routines

Chapter 2

Calendar Clock

The calendar library provides time and date functions. The tm structure and externals are provided in the $VX_TOOLS_DIR/all/include/timevw.h file. The time routines provide access to the battery backup calendar device.

2.1 Time Structure and External Variables

- typedef time_t long;
- struct tm {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
  };
- extern long timezone;
- extern long altzone;
- extern char *tzname[];
- extern int daylight;

The tm structure includes the following fields:

```
int tm_sec;     /* seconds: range 0..59 */
int tm_min;     /* minutes: range 0..59 */
int tm_hour;    /* hours since midnight: range 0..23 */
int tm_mday;    /* day of the month: range 1..31 */
int tm_mon;     /* month: range Jan==0..11 */
int tm_year;    /* year: with 0==1900 */
int tm_wday;    /* dayOfWeek: range Sun==0..6 */
int tm_yday;    /* dayOfYear: 0..365 */
int tm_isdst;   /* !zero implies daylight savings */
```

The timezone, altzone, and tzname[] variables are initialized with the local timezone data. A call to localtime() invokes an internal routine tzset() that determines the value of the daylight variable.

2.2 Calendar Time Routines

- time_t time(time_t *);
- char *ctime(const time_t *);
- char *asctime(const struct tm *);
- struct tm *gmtime(const time_t *);
- struct tm *localtime(const time_t *);
- int stime(time_t *);
- time_t mktime(struct tm *);
- int date(void);

The calendar time routines provide date and time information from the on board time of day device available on many target modules.
The time() function returns the number of seconds since Jan 1, 1970 (Day 0, Unix Epoch).
The asctime function returns the time as ASCII string.
The gmtime function converts the time_t time into the tm structure.
The localtime function is like gmtime but it corrects for time zone.
The stime function sets the time.
The mktime function reconstructs a local time value.
The date() function displays the current date and time represented as both GMT, and local time.

2.3 Initialization Routines

- int tod_initial(void);
- int tod_setlocal(INT32 yymmdd, INT32 hhmmss);
- int tod_setgmt(INT32 yymmdd, INT32 hhmmss);
- int tod_show_tzrules(void);
- int tod_get_tzrules(void);
- int tod_set_tzrules(
  INT32 tz_hhmm,
  INT32 atz_hhmm,
  INT32 atzBegin_mmddhhmmd,
  INT32 atzEnd_mmddhhmmd,
  char *tz_name,
  char *atz_name);

The initialization routines allow the setting of date, time, and read/write access to the timezone data in NVRAM.

The tod_initial() routine reads the timezone data from NVRAM and sets the timezone, altzone, tzname[], and tzrules variables. This routine should be invoked after a reboot.

The tod_setlocal() and tod_setgmt() routines allow the setting of calendar date and time using either local time or GMT time. The argument yymmdd implies a decimal format using 2 digits fields to represent the year, month, and day as an integer. The argument hhmmss implies a decimal format using 2 digit fields to repersent the hours, minutes, and seconds as an integer. (Do not modify the date and time between daylight savings transition. The results may be unfavorable.)

```
Usage: (decimal)
                tod_setlocal (yymmdd, hhmmss);
                tod_setgmt   (yymmdd, hhmmss);

                yy = year
                mm = month  (01..12)
                dd = day    (01..31)
                hh = hour   (00..23)
                mm = minute (00..59)
                ss = second (00..59)
Example:
Set date and time to April 1, 1993, 2:30 P.M. local time.

    tod_setlocal(930401,143000)
```

The tod_show_tzrules() and tod_get_tzrules() routines display usage and verification of the tzrules and data.

The tod_set_tzrules() routine writes timezone data into NVRAM. The rules allow an exact date or a conditional day of the week transition for daylight savings. This routine need only be executed once when a new module has it Ethernet address installed. Other reasons to execute this command are a physically move of the module into a new timezone or corrupted NVRAM.

```
Zone Format:  +/- hhmm (+East, -West GMT)
              hh = hours      (0..23)
              mm = minutes    (00..59)

Conditional Format:  altzoneBegin / altzoneEnd mmddhhmmd
              mm = Month      (1..12)
              dd = Day        (01..31) / (00,01)first/last
              hh = Hour       (00..23)
              mm = Minute     (00..59)
               d = DayOfWeek  0=ExactDate / (1..7)conditional
                              Sunday=1, Saturday=7

The 'dd' and 'd' fields provide a conditional set of rules to
cause daylight transitions to occur on the first or last (week
of the month) specific day of the week, or an exact date.

Rules:
 mmddhhmm0  daylight transition on exact day (dd)
 mm00hhmm1  daylight transition first Sunday of the month
 mm01hhmm1  daylight transition last Sunday of the month
 mm00hhmm0  GMT time only, rules ignored

Example:
Set the timezone for central USA with daylight savings
transition starting on the first Sunday in April, ending
on the last Sunday in October (timezone CST = GMT-6Hrs0Min;
altzone CDT = GMT-5Hrs0Min)

    tod_set_tzrules(-600,-500,40002001,100102001,"CST","CDT")

Default:

    tod_set_tzrules(0,0,0,0,"GMT","  ")
```

2.4 Calendar Device Driver Routines

- void tod_device_id(void);
- void tod_device_on(void);
- void tod_device_off(void);
- time_t tod_device_read(struct tm *);
- time_t tod_read(void);
- void tod_device_write(const struct tm *);
- void tod_write(time_t timer);

These routines access the hardware directly. The calendar devices can be switched off to extend battery life when the module is in storage. The read routines convert the 8 bit BCD time and date representation of the device into decimal. The write routines converts a decimal representation of time into the device 8 bit BCD representation.

2.5 Usage

- libtod_"targetboard" "devicedriver".a
- tod_"targetboard" "devicedriver".o

A library file and downloadable object file is created for the specified target board and device driver. Currently, the target board and device driver is appended to the library filename. The object file may be downloaded directly to the target board to provide access to these routines.

```
The FSCC time of day object may be downloaded to any PC4 FSCC.
the file is $VX_WORKS_DIR/pc4/lib/tod_fsccDS1386.o

At the VxWorks prompt "->"
-> cd "/usr/products/SunOS/vw/vx_tools/devel/pc4/lib"
value = 0 = 0x0
-> ld < tod_fsccDS1386.o
value = 0 = 0x0
-> tod_initial()
value = 0 = 0x0
-> date()
Thu Apr 1 20:30:00 1993 GMT
Thu Apr 1 14:30:00 1993 CST
value = 0 = 0x0
```

Chapter 3

Tools

The vx_tools object file contains a variety of useful routines. The stack checking routines provide an automatic overflow test of the VxWorks tasking facility. The socket cleanup routine shuts down network connections prior to calling reboot (which kills the network and transfer control to boot ROMs). The cleanup routine has limited use.

3.1 Auto Stack Checking

- STATUS autoStackCheck(int Mode);
- void autoStackShow(void);

The autoStackCheck routine is used to add or remove a global stack check hook on tasks. The stack check routine tests the end of the interrupt stack for overflow. It then continues to check the local stack for overflow. Any error messages are sent to the log message queue.

3.1.1 Auto Stack Checking Modes

- AUTOSTK_NORMAL (0)
- AUTOSTK_SWITCH (1)
- AUTOSTK_DELETE (2)
- AUTOSTK_CREATE (4)

The AUTOSTK_NORMAL mode removes any stack check hooks associated with tasks. This is the default setting.

The AUTOSTK_SWITCH mode adds the hook routine AutoStackSwiCheck() that checks the stacks when tasks are switched.

The AUTOSTK_DELETE mode adds the hook routine AutoStackDelCheck() that checks the stacks when a task is deleted.

The AUTOSTK_CREATE mode adds the hook routine AutoStackCreCheck() that checks the stacks when a task is created.

3.2 Socket Cleanup

- STATUS cleanup(int Mode);

The socket cleanup is of limited use. This routine does not address the closing of sockets on a power cycle, a power glitch, a front panel reset, a hung module, invoking reboot() or a Ctrl X.

3.2.1 Cleanup Reboot Modes

- BOOT_NORMAL (0)
- BOOT_NO_AUTOBOOT (1)
- BOOT_CLEAR (2)
- BOOT_QUICK_AUTOBOOT (4)

The reboot definitions and explanations are taken out of the VxWorks Reference Manual (see rebootLib for more detail).

3.3 TRACE facility

3.3.1 Introduction

TRACE is a debugging aid to trace execution of code through printed information in a similar fashion as using printf's statements. The programmer should include TRACE calls at points in the code that represent execution of significant functionality or at critical timing points. TRACE is designed to have minimal impact on the real-time aspects of the software and therefore TRACE is not meant to be inserted for debugging and removed for production. TRACE should be included in development, added during debugging and kept (for the most part) for production. The print out from TRACE will contain timing information.

3.3.2 Using Trace

TRACE is currently implemented for Vxworks on the MVME167.

As mentioned in the introduction, trace is similar to adding printf's to the code except that trace can be "controlled." TRACE can either print messages to the your terminal or to your terminal and the console (if not the same) or to a circular queue in memory. TRACE can be enabled (for printing) on a per task basis.

TRACE uses taskCreateHookAdd(), taskDeleteHookAdd(), and taskVarAdd() vxworks functions to facilitate per task functionality. In addition to task TRACE-ing, functions that are common to multiple tasks can be given their own pseudo task id to enable TRACE-ing of a particular function.

A very significant aspect of TRACE is that the user can have two MVME 167 boards in the same crate, and use one to trace what happended in the other in the case of a crash for which the second board is hung.

Within each task, individual "levels" can be enabled/disabled to allow the debugger to TRACE only what requested.

An example of how TRACE looks in code is:

```
event_worker_task()
{
    while (1)
    {
        msgQReceive( evnt_q, &evnt_ptr, sizeof(evnt_ptr), WAIT_FOREVER );
        TRACE( 1, "Received event 0x%x", evnt_ptr, 0 );

        check_event();

        if (log_event)  log_event();
    }
}

check_event()
{
    if ( event_looks_like_this )
    {
      TRACE( 2, "checking event that looks like ---", 0, 0 );
    }
    else
    {   TRACE( 2, "checking event that looks like +++", 0, 0 );
    }
}
```

```
log_event()
{
    TRACE( 2, "logging event", 0, 0 );
}
```

The first parameter to the TRACE macro is the level. The TRACE printout has indentation corresponding to the level. Although the value of level is arbitraty, it is recommended that the levels be determined by thinking of the stack level at which the call is most likely to occur at. For recursive functions, level can be a variable that is shifted. Level is actually a bit mask such that only 32 levels are valid. ((lvl&0x80000000)?lvl:lvl<<1) could be used in a recursive function.

The second parameter is a printf style string without an ending newline. The string can contain formatting for up to two variables. The remaining two parameters are to fill the variable formatting in the second parameter. If they are not needed, they should be 0's.

3.3.3 Controlling Trace

The first step in using TRACE is to call traceInit(): it creates a traceTask that displays trace messages to the terminal for the task issueing the TRACE call when the trace mode is set to 4 (see below).

Only tasks created after traceInit has been called are ready for TRACE control. Functions are made ready for TRACE-ing by including the macro TRACEPROC(name) as the first line after the declarations if the function (as this macro includes a static declaration).

traceInit(int circ_que_entries, int max_tasks, unsigned int traceclk);

Arguments:

circ_que_entries	
max_tasks	
traceclk	The address of (or pointer to) the register timer. In the case of the MVME167 and if the proper vxworks includes are used, the values for traceclk can be PCC2_TIMER2_CNT, PCC2_TIMER1_CNT, VMECHIP2_TTCOUNT2, VMECHIP2_TTCOUNT1, or 0 which will use VMECHIP2_TTCOUNT1.

After tasks and functions are ready for TRACE-ing, the following
commands can be used:
traceInfo()

Arguments:
==

traceInfo lists tasks and functions ready for TRACE-ing. Also lists the current trace level mask. The mask is changed by the following commands.

traceOn(unsigned int tid, unsigned int l1, unsigned int l2);

Arguments:

tid	Task ID.
l1	level to be set.
l2	level to be set.

traceOn uses the task Id tid listed in traceInfo() or as returned by the vxworks command i(). l1 and l2 are us

- If $l1 < l2$ a range is set.
- If $l1 > l2$ just l1 and l2 are set as separate levels.
- If l1 = l2 just l1 is set.

The valid range for l1 and l2 is 0 to 31.

traceOff(unsigned int tid, unsigned int l1, unsigned int l2);

same as traceOn() except it resets the specified level(s).

traceGlobalOn (unsigned int l1, unsigned int l2);
traceGlobalOff (unsigned int l1,unsigned int l2);

Arguments:

l1	level to be set/reset.
l2	level to be set/reset.

They affect tracing of all tasks and functions.

traceMode(int mode);
It selects different modes for tracing.
Arguments:

mode	trace mode setting: mode=0 ==> TRACE-ing disabled mode=1 ==> print to terminal using printf mode=2 ==> use logMsg for printing mode=3 ==> print to circular buffer mode=4 ==> send msg to traceTask which will use printf

traceShow(int delta_or_abs_time);
It displays TRACE messages printed to the circular queue.
This is only applicable for traceMode set to 3.

Arguments:

delta_or_abs_time	If = 0 ==> displays absolute time
	If = 1 ==> displays delta time

traceRemoteShow(int delta_or_abs_time, unsigned int slave_base_addr, 0);

Same as traceShow, except will search over VME starting at slave_base_addr and continuing for 32 meg for a

For example, if the 167 VME address over the bus is 0x90000000:

```
vx_prompt> traceRemoteShow 0,0x90000000,0

the output will be:

set traceRemoteInfoBlockAddress to 91f4b854
traceRemoteInfoBlockAddress is 91f4b854


    0       0                       dscSCD_ISPScroll: enter
    7       0                       dscSCD_ISPScroll: enter
16635       0                       dscSCD_ISPScroll: enter
16642       0                       dscSCD_ISPScroll: enter
  ...

vx_prompt>

next call would be:
```

```
vx_prompt> traceRemoteShow 0,0x90000000,0x1f4b854
```

`in which the offset is given as an internal offset from the beginning of the memory (offset 0 == 0x90000000) and not as an absolute address over the VME bus.`

traceReset(); this function clears the circular que;

traceInit(circ_que_entries, max_tasks, traceclk);

3.3.4 Special Functionality

For the case where tasks are created and deleted in real time such that traceOn cannot be executed in a timely manor, the trace levels 15-31 default to on. This fuctionality can be taken advantage of when debugging a procedure interactively. If trace levels 15-31 are use and a procedure is activated from the shell using sp(), TRACE-ing will automatically commense if the traceMode is not zero. If trace levels 15-31 would not normally be used, one of several tricks that come to mind could be used to temporarily shift the trace level up into that range.

EXAMPLE PRINT OUT FROM traceShow

```
vx_prompt> traceShow
         0       0       (null)          dscSCD_ISPScroll: enter
         7       0       (null)          dscSCD_ISPScroll: enter
     16635       0       (null)          dscSCD_ISPScroll: enter
     16642       0       (null)          dscSCD_ISPScroll: enter
     33268       0       (null)          dscSCD_ISPScroll: enter
     33276       0       (null)          dscSCD_ISPScroll: enter
     49900       0       (null)          dscSCD_ISPScroll: enter
     49910       0       (null)          dscSCD_ISPScroll: enter
     66538       0       (null)          dscSCD_ISPScroll: enter
     66545       0       (null)          dscSCD_ISPScroll: enter
     83172       0       (null)          dscSCD_ISPScroll: enter
     83180       0       (null)          dscSCD_ISPScroll: enter
     99807       0       (null)          dscSCD_ISPScroll: enter
     99814       0       (null)          dscSCD_ISPScroll: enter
    116441       0       (null)          dscSCD_ISPScroll: enter
    116449       0       (null)          dscSCD_ISPScroll: enter

(16) press q to quit, any other key to continue...
```

The first column is the value read from the timer. The second column is the task id (it may be 0 if ISP). The third column is task name or name given in TRACEPROC(name) macro. The forth column is the message string given in the TRACE macro indented by the level.

A common way to look at trace circular queue information is to redirect the output to a file onto a disk that is cross-mount on a host computer where the file can then be browsed or printed out. Use:

$traceShow > somefile$

making sure that you have permission to create the file. Note that the output will only pause every sixteen lines if NOT redirected.

Chapter 4

DVX Dart VxWorks eXtensions

4.1 Introduction to DVX task variables

Dart **V**xWorks e**X**tensions provides the user with an alternative way of creating and handling task variables.

The user can create two types of task variables: scalar and pointers. Upon variable creation the user is returned a handle for it that can be subsequently use to identify the variable itself. The task private data are stored in an array of structures that are task specific. The handles are globally known in the system and the bookeeping is maintained through an array whose elements have the handles value as indexes and the handles pointers as values.

Only one VxWorks task variable is used in the system , for each task, and it is the pointer to the task private data array of structures.

- scalars : the user can create, set a value and retrieve the task variable.
- pointers: the user can create and retrieve a pointer to a wanted malloced space: he simply needs to specify the size for it and the malloc operation is internally done. Upon task deletion the malloced space is freed automatically.

The user is also provided by an exit handlers mechanism that supports the atexit function implemented for VxWorks with the same functionality as on a UNIX system. The exit handler implementation exists as a layer above the DVX task variables and is implemented using the DVX task variables; however, it is also available using the VxWorks task variables. The implementation is completely trasparent to the user.

Besides the "UNIX" style atexit, the user is also provided with another exit handler mechanism that allows to pass an integer variable to the user exit handler; this turned out to be a need under VxWorks to be able to access task variables from within the exit handler. This is because, differently from UNIX, the exit handler is not executed in the context of the task that registered the exit handler itself, so if the user wants to access his task variables, the only safe way is to pass the task ID to the exit handler and the task ID would be the integer variable to pass to the handler when registering it with the dvx_atexit call.

As a reminder, the standard atexit call does not allow any argument to be passed .

4.2 Loading the dvx package

The manager for the system is responsible to load the dvx object library that contains the proper DVX object modules. Two object files are available according to the user's need whether to write an application that uses the atexit call and strictly dvx task variable or atexit and VxWorks task variables:

$VX_TOOLS_DIR/mv167/lib/dvx_var_dvx.o
$VX_TOOLS_DIR/mv167/lib/dvx_var_vxw.o

The load and the initialization should be automatically done through a startup script at startup time (or boot time) of the VxWorks system ; the DVX task facility is created and initialized simply invoking:

dvx_setup

At this stage, using the VxWorks task hook library, the hooks executed at each task creation and deletion are created. The delete hook code is the one that will invoke the functions registered by the user via the atexit and/or the dvx_atexit call.

4.3 Task variable functions: Internals

The function prototypes and structures are defined in the file
$VX_TOOLS_DIR/mv167/include/dvx.h In the same files the "arbitrary" maximun number of dvx task variables allowed for the system is set to 100; the maximum number of exit handlers registered through the atexit call is set to 32 just to conform with the UNIX value.

STATUS dvx_setup()

This function is called only once at startup time:
it is called by the system.
It initializes the task variable and hook facilities;
it registers hooks for task creation and deletion.
It creates and initializes the tag array of global

indexes that will be used to register the number of
dvx task variables in the whole system.

4.3.1 Task creation and deletion

When a task is created the hook dvx_task_hook_create is executed; at task deletion dvx_task_hook_delete will be invoked.

static void dvx_task_hook_create (WIND_TCB *taskPtr)

This function will be called by the system at each task creation.
It allocates the space for the task private data
array of structures and the pointer to it is stored
as a VxWorks task variable .
It allocates the space for the exit handlers using either
code based on DVX task variables or code based on
VxWorks task variables.

Arguments:
WIND_TCB *taskPtr
Task id, the pointer to the Task Control Block.

static void dvx_task_hook_delete (WIND_TCB *taskPtr)

This function will be called by the system at each task deletion.
It invokes the task specific exit handlers previously
registered with the atexit call.
It also frees space that was malloced for internal
use and for user's specific purposes.

Arguments:
WIND_TCB *taskPtr
Task id, the pointer to the Task Control Block.

STATUS dvx_validate_handle (int handle, int handle_flag, int tId)

This function is used internally to validate the handle
assigned to a DVX task variable.

Arguments:

int handle	Task variable handle
int handle_flag	To distinguish whether the handle is being created or being used to set/retrieve a task variable.
int tId	Task ID as returned , for example by the call to

4.4 Task variable functions: User Interface

The user who is going to create a task using the DVX task variable facility should first check that the DVX package has been initialized in the system. This can be done either from the VxWorks shell or within the application code simply calling the function dvx_init_verify. An error message is returned in case DVX was not started up.

STATUS dvx_init_verify() This function returns either 0 (OK) or an error message.

As previously mentioned, the user is provided with functions to create and manipulate DVX task variables:

- STATUS dvx_create_scalar (int tId, int *usrHandle);
- STATUS dvx_set_scalar (int tId, int usrHandle, int varVal);
- STATUS dvx_get_scalar (int tId, int usrHandle, int rdVar);
- STATUS dvx_create_ptr (int tId, int *usrHandle, int size, DVX_PTR **usrPtr);
- STATUS dvx_get_ptr (int tId, int usrHandle, DVX_PTR **usrPtr);

STATUS dvx_create_scalar (int tId, int *dvxHandle);

To create a scalar type of task variable.

Arguments:

int tId	Task ID as returned , for example by the call to taskIdSelf();
int usrHandle	Task variable handle: the user supplies a location for it and the functions stores in it an available one; If the handle address already contains a valid handle

its value is returned.

STATUS dvx_set_scalar (int tId, int usrHandle, int varVal);

To set a value to the task variable.
The user passes the previously created handle to access the task variable and to set a value; it returns OK if the handle is valid or an error message if not.

Arguments:

int tId	Task ID as returned , for example by the call to taskIdSelf();
int usrHandle	Task variable handle.
int varVal	Task variable value to be set.

STATUS dvx_get_scalar (int tId, int usrHandle, int *rdVar);

To retrieve the value assigned to the task variable.
The user passes the previously created handle to access the task variable and its value is stored in rdVar;
it returns OK or an error message
if the handle is not valid .

Arguments:

int tId	Task ID as returned , for example by the call to taskIdSelf();
int usrHandle	Task variable handle.
int *rdVar	Pointer to store returned vavariable value

STATUS dvx_create_ptr (int tId, int *usrHandle, int size, DVX_PTR **usrPtr);

To create a ptr task variable.
The user supplies a location for the handle
and the functions stores in it an available one;
The user also passes the size in bytes of the desired space
to be malloced and the pointer to it is stored in usrPtr.

Arguments:

int tId	Task ID as returned , for example by the call to taskIdSelf();
int *usrHandle	Task variable handle: the user supplies a location

for it and the functions stores in it an available one;
If the handle address already contains a valid handle
its value is returned.

int size	Size in bytes to be malloced.
DVX_PTR **usrPtr	Pointer to the malloced space. DVX_PTR is a typeded defined in the include file dvx.h as a pointer to void.

STATUS dvx_get_ptr (int tId, int usrHandle, DVX **usrptr);

To retrieve the value assigned to the task variable.
The user passes the previously created handle to access
the task variable and its value is stored in usrPtr;
it returns OK or an error message
if the handle is not valid .

Arguments:

int tId	Task ID as returned , for example by the call to taskIdSelf();
int usrHandle	Task variable handle.
DVX_PTR **usrPtr	Pointer to store the retrieved value of the requested pointer.

STATUS atexit(void (*regFunc)(void));

To register the function regFunc as a task exit handler.
The function will be invoked by the system at task deletion time.

STATUS dvx_atexit(int (*regFunc)(int), int);

To register the function regFunc as a task exit handler.
The function will be invoked by the system at task deletion time.
The second argument is the argument to pass to the exit handler;
A typical example is to pass the task ID to be able to access
task variables during task exit .

4.5 Code example

The following is a code example of usage of the DVX task variables; at first the user checks if the DVX package has been initialized. Two exit handlers are registered; then three DVX task variables are created and for the scalar variable a value is set and then retrieved.

Please note: due to the use of task variable, the user should ALWAYS spawn the application and NEVER execute it interactively.

```
/* Test to handle task variables using dvx task variable package */

#include "dvx.h"

int testVar(void);
void myexsub1 (void);
void myexsub2 (void);
int  myexsub2 (int);

int  scalarHandle;

/*============================================================== */

void myexsub1()
{
printf ("\n myexsub1: Hello from exiting \n");
return;
}

/* ============================================================= */
void myexsub2()
{
printf ("\n myexsub2: Hello from exiting \n");
return;
}

/* ============================================================= */
/* This exit handler accepts one argument and it is registered using
   dvx_atexit, an extension of the standard atexit that allows to pass
   one argument.                                                 */

int myexsub3 (int tvar)
{
  STATUS ret_status;
  int    rd_var;

        printf ("\n myexsub3: Hello from exiting task Id = 0x%x\n", tvar);

/* Accessing a task variable */

        ret_status = dvx_get_scalar (tvar, scalarHandle,&rd_var);
        printf ("\n myexsub3: tid = 0x%x, data read = 0x%x\n",tvar, rd_var);

        return (OK);
}

/* ==================================================================== */
int testVar()
{

   STATUS ret_status;
   int dat;

   int  rdvar;
   char *malptr;
   int  malsiz = 16; /* In bytes */

   int  *malintptr;
   int  *rdptr;
   int  tId;
   int  i;

   int ret_atex;

   int  ptrHandle1;
   int  ptrHandle2;
/* ---------------------------------------------------- */

/* Checking if DVX task var facility has been initialized */
   if ( dvx_init_verify() != OK) return (ERROR);

/* Registering two exit handlers */

   ret_atex = atexit (myexsub1);
   if ( ret_atex != 0)
printf ("\n Unable to register function, table is full\n");

   ret_atex = atexit (myexsub2);
   if ( ret_atex != 0)
printf ("\n Unable to register function, table is full\n");

   tId = taskIdSelf();


   ret_status = dvx_create_ptr (tId, &ptrHandle1, malsiz, (DVX_PTR **)&malptr);
   if (ret_status != OK)
      {printf ("\n main: Error in creating pointer malptr\n"); }
   else
   {
      printf ("\n main: handle ptr = 0x%x ,malptr = 0x%x \n", ptrHandle1,(int)malptr);
      for (i=0;i<malsiz;i++)     malptr[i]=(char)i;
   }
   ret_status = dvx_create_ptr (tId, &ptrHandle2, malsiz, (DVX_PTR **)&malintptr);
   if (ret_status != OK)
      {printf ("\n main: Error in creating pointer malintptr\n");}
   else
      {printf ("\n  main: handle ptr = 0x%x, malintptr = 0x%x \n", ptrHandle2, (int)malintp

   printf ("\n main: trying to use same handle for malinptr \n");

   ret_status = dvx_create_ptr (tId, &ptrHandle2, malsiz, (DVX_PTR **)&malintptr);

   if (ret_status != OK)
        {printf ("\n main: Error in creating pointer malintptr\n");}
   else
        {printf ("\n  main: handle ptr = 0x%x, malintptr = 0x%x \n", ptrHandle2, (int)malin

   return_status = dvx_get_ptr (tId, ptrHandle2, (DVX_PTR **)&rdptr);
   printf ("\n main: retrieving variable ptr2 := 0x%x\n",(int)rdptr);

   dat = 0x5000;

   ret_status = dvx_create_scalar (tId, &scalarHandle);

   printf ("\n main: scalarHandle = 0x%x, storing data = 0x%x\n", scalarHandle,dat);

   ret_status = dvx_set_scalar (tId, scalarHandle, dat);

   ret_status = dvx_get_scalar (tId, scalarHandle, rdPtr);

   printf ("\n main: rdvar = 0x%x\n",rdvar);
   printf ("\n main: Exiting test_addvar ");

   return (OK);
}
```

Part II

Target Specific Libraries and Routines

Chapter 5

MVME167 Supplement

5.1 Interrupt Routines

VxWorks uses the two timers in the PCCchip2; one for sysClk and one for sysAuxClk. The VMEchip2 has two additional timers - 1 and 2. The interrupt routines follow the exact same format as the sysAuxClkxxxx() routines, except they do not have the limitation of minimum 3 ticks/sec and max 5000 ticks/sec. See VxWorks documentation for additional information.

5.1.1 routines

- clk1Connect(interrupt_handler,param);
- clk1Disable();
- clk1Enable();
- clk1LevelSet(level); /* default is level 5 */
- clk1RateGet();
- clk1RateSet();
- clk2Connect(interrupt_handler,param);
- clk2Disable();
- clk2Enable();
- clk2LevelSet(level); /* default is level 5 */
- clk2RateGet();
- clk2RateSet();

5.1.2 routines description

clk1Connect(FUNCPTR interrupt_handler, int param);
This function specifies the interrupt handler to be called at each vmechip2 timer 1 clock interrupt. param is an argument to be passed to the interrupt handler. This function does not enable the interrupts.

clk1Disable();
To turn off the timer interupts disabling the vmechip2 timer1 interrupts.

clk1Enable();
To turn on system clock interrupts.

clk1LevelSet(int level);
To set interupt level, default is level5. Level will take effect at time of next clkxEnable call.

clk1RateGet();
To return the interrupt rate of the system clock as number of ticks per second.

clk1RateSet(int ticksPerSecond);
This routine sets the interrupt rate of the clock. It does not enable clock interrupts. It returns OK or ERROR if the tick rate is invalid or the timer cannot be set.

The clk2xxxx functions perform in the same way as the clk1xxxx for the vmechip2 timer2.

5.1.3 example

The following illustrates interrupt handler passing message to interrupt task.

```
#include "msgQLib.h"


void initialization(void)
{
    MSG_Q_ID    msg_q;
    void    clock1_interrupt_handler(),clock1_interrupt_task();

    msg_q = msgQCreate(100,4,MSG_Q_PRIORITY );
    clk1Disable(); /* just for good measure */
    clk1Connect(clock1_interrupt_handler,msg_q);
    clk1RateSet(1); /* one tick per second */

    taskSpawn("tClk1",200,0,2000,clock1_interrupt_task,msg_q);

    clk1Enable();
}


void clock1_interrupt_handler( MSG_Q_ID msg_q )
{
    msgQSend(msg_q,"1234",4,NO_WAIT,MSG_PRI_NORMAL);
}

void clock1_interrupt_task( MSG_Q_ID msg_q )
{
    char    msg_buf[4];

    while(1)
    {
        msgQReceive(msg_q,msg_buf,4,WAIT_FOREVER);

        /* more task code could go here */

        logMsg("receive interrupt");
    }
}
```

5.2 Simple Timing

The followinf routines are used to start ans stop timing measurements in the microsecond accuracy.

5.2.1 routines

- clk1Stop()
- clk1Start()
- clk2Stop()
- clk2Start()

5.2.2 macros

These macros are defined in mv167sup.h.

- t1=clk1Read /* read hardware timer and save in t1 */
- clk1Diff(t1) /* read hardware timer and return delta from t1 value */
- t1=clk2Read
- clk2Diff(t1)

(where t1 is an unsigned int)

5.2.3 example

```
#include "mv167sup.h"

example()
{
    unsigned int t1;

    clk2Stop();
    clk2Start();

    t1=clk2Read;

    /* place code to be timed here between clk2Read and clk2Diff(t1) */

    logMsg("time in usec for stuff is %u\n",clk2Diff(t1));
}
```

5.3 VME Mapping Routines

The default for the 4, 8, 16, or 32 Meg boards is to enable A32 USR,SUP,D64,BLK,PGM and DAT transfers (NOTE: no A24) starting at the following VME addresses:

- 4MB 0x00400000
- 8MB 0x00800000
- 16MB 0x01000000
- 32MB 0x02000000

Addresses given as parameters should have low 16 bits = 0. Blk xfers are only available through DMA controller routines. Starting addresses must be on boundaries equal to the size of the address range being mapped.

5.3.1 routines

support

- vmeSlaveMapShow()
- vmeMasterMapShow()

routines description

vmeSlaveMapShow();
It displays a map of memory sections assigned to a VME slave.

vmeMasterMapShow();
It displays a map of memory sections assigned to a VME master.

basic

This section deals with routines that allow to map the VMEbus to local bus using the Slave Map Decoders (Slave Map functions) and routines that allow to program information residing on the local bus to VMEbus map decoders (Maste Map functions).
For a complete description of the VME decoders for the VMEchip2 please refer to the MVME167/MVME187 Programmer's Reference Guide starting at page 2-25.

- vmeSlaveMap1(am,vmestart,vmeend,lbstart) /* am =0 disables; DEFAULTS */
- vmeSlaveMap2(am,vmestart,vmeend,lbstart)
- vmeMasterMap1(am,vmestart,vmeend,width) /* width: 0=D32,other=D16 */
- vmeMasterMap2(am,vmestart,vmeend,width)
- vmeMasterMap3(am,vmestart,vmeend,width)
- vmeMasterMap4(am,vmestart,vmeend,lbstart,width) /* only Map4 has translation */
- vmeMasterIoMap1(am,width)
- vmeMasterIoMap2(am_mode)

routines description

vmeSlaveMapN(am,vmestart,vmeend,lbstart) N = 1,2

It allows to map two different VMEbus segments to local bus using the two slave map decoders.
Arguments:

unsigned int	am	VME address modifier
unsigned int	vmestart	VME starting address
unsigned int	vmeend	VME last address
unsigned int	lbstart	local address

vmeMasterMapN(am,vmestart,vmeend,width) N = 1-3

It allows direct mapping between local address space and real VME address space.
Arguments:

unsigned int	am	VME address modifier
unsigned int	vmestart	VME starting address
unsigned int	vmeend	VME last address
unsigned int	width	Data width: 0 = D32, other = D16

vmeMasterMap4(am,vmestart,vmeend,lbstart,width)

Same as the Master Map 1-3 plus it allows for translation of VME address. example, map to a VME address
Arguments:

unsigned int	am	VME address modifier
unsigned int	vmestart	VME starting address
unsigned int	vmeend	VME last address
unsigned int	lbstart	local address
unsigned int	width	Data width: 0 = D32, other = D16

vmeMasterIoMap1(am,width)

Arguments:

unsigned int	am	VME address modifier :
		User = 0x29
		Supervisory= 0x2D
		Other disabled

unsigned int	width	VME address modifier
unsigned int	width	Data width: 0 = D32, other = D16

vmeMasterIoMap2(am_mode)

Arguments:

unsigned int	am_mode	address modifier mode:
		User = 0 or 0x29
		Supervisory= 1 or 0x2D

next level

This section deals with routines that have to do with cache coherence. The MV167 local bus master is capable of driving the snoop control signals during bus transactions. With these controls, the MC68040 internal cache can be maintained cache coherent, allowing software applications to run without flushing the data cache. Please note that the bus overhead required to maintain the internal 68040 data cache coherence may counteract the benefits of not having to control cache coherence through software cache management.

- vmeSlaveMap1Am(am) /* add additional ams; am=0 disables mapping register */

- vmeSlaveMap1SnWp(snoop_write_post) /* 0=NO snoop,NO wp; 1=NO snoop, WP */
  /* 2=wr snk,rd dirty, NO wp */
  /* 3=wr snk,rd dirty, WP */
  /* 4=wr invalidate,rd invalidate, NO wp */
  /* 4=wr invalidate,rd invalidate, WP */

- vmeSlaveMap2Am(am)
- vmeSlaveMap2SnWp(snoop_write_post)
- vmeMasterMap1Wp(enable) /* 0=disable; other=enable */
- vmeMasterMap2Wp(enable)
- vmeMasterMap3Wp(enable)
- vmeMasterMap4Wp(enable)
- vmeMasterIoMap1Wp(enable)
- vmeMasterIoMap2Wp(enable)
- vmeBR(lvl) /* Its a constraint of the hardware on the board */
  /* that the first Bus Request level is 3 */

5.4 DMA Routines

5.4.1 routines

basic

- dmaConnect(interrupt_handler, param)
- dmaDisable()
- dmaEnable()
- dmaStart(vmeadd,localadd,bytecnt,direction,dtb_mode,non_blk_am)
- dmaDone()

routines description

dmaConnect(FUNCPTR interrupt_handler, int param);
It allows to associate an interrupt handler routine to the arrival of a dma interrupt; param is an argument passed to the interrupt handler code.

dmaDisable();
It disables dma operations disabling dma interrupt generation.

dmaEnable();
It enables dma operations enabling dma interrupt generation.

dmaStart(vmeadd,localadd,bytecnt,direction,dtb_mode,non_blk_am)
It starts dma transfers.

Arguments:

caddr_t	vmeadd	VME starting address for the transfer.
caddr_t	localadd	User's program-defined starting address to write data to or to read data from depending on the transfer direction.
unsigned int	bytecnt	Number of bytes to transfer.
int	direction	Transfer direction: 0 = from VME , 1 = to VME.
int	dtb_mode	Data size transfer mode: 0 = D16 1 = D32 2 = D32BLK 3 = D64BLK
int	non_blk_am	Non block transfer address modifier.

Note: When dtb_mode's 2 or 3 are chosen, the dma controller will automatically modify the am. The controller will use the un-modified non_blk_am if xfers at the beginning and end of the dma that are not D32/D64 aligned. For example, if your parameters were (0x82000003,&buff,10,dir,2,9), (09=extended,usr,data) the DMA controller would automatically do a single byte xfer using am=09 followed by a 2 D32 BLOCK xfer with am=0x0b (0B=extended,usr,block) and finish again with a single byte xfer with am=09. Symbols D16, D32, D32BLK, D64BLK are defined in mv167sup.h */

dmaDone();
This function is used to poll for completion of DMA transfer.
It returns: 1 = DMA done; 0 = DMA NOT done; any other status means an error. See $VX_TOOLS_DIR/mv167/include/mv167sup.h for "DMA_DONE" defines and macros.

example

```
#include "mv167sup.h"
#include "semLib.h"

ttt()
{
    extern int    dma_interrupt_handler();
    int i;
    SEM_ID sem;

    sem = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
    dmaConnect(dma_interrupt_handler,sem);
    dmaEnable();
    dmaStart(0x82000000,0x100000,1000,0,0,9);
    for (i=0;i<3;i++)
    {
        semTake(sem,WAIT_FOREVER);
        dmaStart(0x82000000,0x100000,1000,0,0,9);
    }
}

dma_interrupt_handler(sem)
SEM_ID sem;
{
    logMsg("hello");
    semGive(sem);
}
```

next level

- dmaStatus() To test dma status: 0 in progress, 1 done, otherwise error.
- dmaLevelSet(level) To set the interrupt level; default is 5.
- dmaStartChain(chain_struct_addr_ptr) This is used to chain different vme address ranges and automatically transfer data to/from them . The user needs to supply a link list of structures containing the same information as he would pass through dmaStart call.
- dmaVmeBR(lvl) To control the VME bus request levels : 0-3 are the allowed level values.
- dmaSn(snoop_mode) To control the snoop mode = 0-3 as on p. 2-53 of 167 programmer's ref */
- dmaThrottle(release_mode,time_on,time_off) To control the relesae mode of the DMA controlle: see page 2-51 and 2-60 for the time values in the mv167 programmer's reference guide.

  release mode = 0-7
  0 = release on timer expire AND BRx active
  1 = release on timer expire
  2 = release when a BRx is active
  3 = release on timer expire OR BRx active
  4-7 = repeat above and add "fair mode" (DMAC waits until it's BR level is INactive)
  time_on and time_off times increase as number goes up */

- †chain_struct_addr_ptr = dmaChainHead(chain_ptr,vmeadd,localadd,
  bytecnt,dir,dtb_mode,non_blk_am)
  /* If chain_ptr is NULL, malloc */
  /* will be called. */
- †chain_struct_addr_ptr = dmaChainAdd(chptr,vmeadd,localadd,bytecnt,
  dir,dtb_mode,width,non_blk_am)
  /* If chain_ptr is NULL, malloc */
  /* will be called. */
- †dmaChainSn(mode) /* mode = 0-3 as on p. 2-49 of 167 programmer's ref */

†not complete

5.5 Readout routines

5.5.1 routines for reading EBIs

- readEbiOne(addrebi,chkebi,nwords,data)

addrebi -	Pointer to read address 0xXXXX0000	
chkebi -	Pointer to status address 0xXXXX8004	
nwords -	Total number of 32-bit words read	
data -	Array of words read	
	First word:	Word count for buffer
	Second ... last word:	Data from buffer

routine to read one EBI with address XXXX. It waits until buffer has data

- readEbi(nebis,addrebi,chkebi,nwords,data)

nebis -	Number of EBIS to read	
addrebi -	Array of pointers to read addresses 0xXXXX0000	
chkebi -	Array of pointers to status addresses 0xXXXX8004	
nwords -	Total number of 32-bit words read	
data -	Array of words read	
	First word:	Word count for first buffer (=N1)
	Second ... N1+1:	Data from buffer one
	N1+2:	Word count for second buffer (=N2)
	N1+3 ... N1+n2+2:	Data from buffer two etc.

routine to read up to 10 buffers. It checks if buffers are synchronized and returns 0 if they are OK or 1 if they are out of Synch. It waits until all buffers have data before it reads them out.

5.5.2 example

```
/* routine to read EBIs from MVME167 and return the average readout time*/
#include "vxWorks.h"
#include "stdioLib.h"
#include "math.h"
#include "mv167sup.h"
#include "vme167.h"
#include "taskLib.h"
#include "wdLib.h"

int rdebi(nevt)
int nevt;
{
int n2,j,i;
int ReadEbi();
unsigned int istat,nwords,data[2000];
int *addrebi[10], *chkebi[10];
unsigned int t1,tim,t2,t2in;
float mean,rms;

vmeMasterMap3Am(SUP_DAT);

clk2Stop();
clk2Start();

printf("Number of events to be read %d\n",nevt);
n2=0;
t1=clk2Read;
t2in=clk2Diff(t1);
printf("time %u\n",t2in);
\* Addresses of EBIs */
chkebi[0] = (int *)0xc5008004;
addrebi[0] = (int *)0xc5000000;
chkebi[1] = (int *)0xc4008004;
addrebi[1] = (int *)0xc4000000;

tim=0;
do {            /* read nevt events with dma */
   t1=clk2Read;
   istat=ReadEbi(2,addrebi,chkebi,&nwords,&data);
   if(istat != 0) goto lp2;
   ++n2;
   tim += clk2Diff(t1);
} while (n2 < nevt);
lp2:
printf("end***       Number of events read: %d\n", n2);
mean = (float) tim;
mean /= n2;
mean -= t2in;
printf("mean: %f usec\n",mean);
return 0;
}
```

5.6 VME Interrupts

5.6.1 handler

VME interrupters generate the VECTOR_NUMBER. From the applications notes - VxWorks uses the following interrupts:

Mailbox	71
System Clock	113
Auxillary Clock	72
Exelan	193
CMC (enp)	192
Serial Tx	74
Serial Rv	76

- ††intConnect(INUM_TO_IVEC(VECTOR_NUMBER), routine, parameter); To connect a specified C routine to a specified interrupt vector.Routine is called with the parameter when the interrupt occurs.
- †sysIntDisable(vmebusirqlevel) To disable a specified VMEbus interrupt level.
- †sysIntEnable(vmebusirqlevel) To enable a specified VMEbus interrupt level.
- †vmeIntLevelSet(vmebusirqlevel,cpulevel) To set an interrupt level, default level is that of irq.
- †sysBusIntAck(vmebusirqlevel) To acknowledge a specified VMEbus interrupt level.

††implemented by VxWorks as part of the intArchLib †implemented by VxWorks as part of sysLib as specified in the MVME 167 VxWorks board support document.

5.6.2 generator

The big thing here is the interrupt that occurs when the interrupt you generate is acknowledged. But it would not be unlikely that you would do nothing when the interrupt is acknowledged; so you would not use vmeIacked*.

- sysBusIntGen(vmebusirqlevel,vector) To generate a VMEbus interrupt for a specified interrupt level.
- vmeIackedConnect(routine,arg)
- vmeIackedDisable() To disable system clock interrupts.
- vmeIackedEnable() To enable system clock interrupts.
- vmeIackedLevelSet(level) To set an interupt level, default is level5. Level will take effect at time of next clkxEnable call.

5.7 General Interrupts

- interruptLevelShow() /* Shows interrupt level and vector number */
  /* information first for VMEchip2 interrupts, */
  /* then for PCCchip2 interrupts. */

5.8 Address Translation / Cache Mode Control

5.8.1 routines

- addressTranslationShow(force) /* 0=show page table if enabled; 1=show table */
- addressTranslationRoot(page_size) /* 1=8K pages; 0=4k pages */
- addressTranslationEnable(upper_mem) /* 0x80000000,0xc0000000,0xe0000000,...
  ...0xfc000000,0xfe000000 0=0xff000000 */
- addressTranslationDisable()
- addressTranslationClear(log_start,log_end)
- addressTranslation(log_start,log_end,phys_start,cache_mode,write_prot,expert)

  write_prot=1 → protect pages
  cache_mode=0 → writethrough
  cache_mode=0 → copyback
  cache_mode=0 → cache inhibit, serialized

cache_mode=0 → cache inhibit, nonserialized
expert → entire page descriptor (don't use)

- addressTranslationdtt0(base,mask,enable,cm,wp,expert)

These routines effect *paged* address translation. *Transparent* translation (which vxworks uses) will still be in effect for all direct physical memory accesses and accesses to the upper 16Meg address range.

The indended use for these routines is to change the *data space* (as opposed to instruction/execution space) caching mode for a block of physical local memory space BY ACCESSING THE BLOCK VIA LOGICAL ADDRESSES.

Any address block that does not overlap with the *physical memory* address range or the upper 16M address space (ffxxxxxx) may be used for a logical address block.

logical addresses can also be transalated/mapped to (physical) addresses above the *physical memory* address range. i.e. log_start same as phys_start in addressTranslation.

paged address translation can be used for VME mapping, or *transparent* can be used when vme mapping is placed above "upper_mem" (addressTranslationEnable(upper_mem)). A logical address

Example: To change the cache mode, from the vxworks default of copyback,
for a 2k address block at physical address ff0c10-ff140f, two 4k
block at (ff0000 and ff1000) will need to be mapped).
You can *pick* a logical address, say 0x07ff0000. If you use
0x07ff0c10 (physical address with upper bits changed to move it into
"logical address space") the program will automatically allocate
the correct pages.

- addressTranslationRoot(0); /*setup 4k page paged address translation table*/
- addressTranslation(0x07ff0c10,0x07ff140f,0xff0c10,0,3); /* setup cache */
  /* inhibited block(s) */

- addressTranslationShow(1); /* show pages defined */
- addressTranslationEnable();

Note: remember, because of the AddressTranslationCache, you may have to re-issue "addressTranslationEnable()" (which flushes the ATC) in order for changes made, by an addressTranslation(...) call while paged address translations are enabled, to take effect. An example of this is re-issuing the same addressTranslation(...),except with a different cache mode or write protect mode.

Note: you can have different caching mode within a 4k physical block by accessing them *through* different 4k logical address blocks.

5.9 Routine Summary

```
The support object module mv167sup.o is located in the vx_tools
directory tree.

$VX_TOOLS_DIR/(version dir)/mv167/lib/mv167sup.o /* object module to */
                                                 /*  download        */

$VX_TOOLS_DIR/(version dir)/mv167/inc/mv167sup.h /* "Simple Timing"     */
                                                 /*  macros and defines */

There are defines for some of these arguments in vm167sup.h;
Some of the args I don't have defines for (i.e. ENABLE) because
they are so general that they would probably conflict with someone
eventually.

The routines in mv167sup are

 gpioConnect(routine, arg)      /* for interrupt through J3pin19 and PCCChip2 */
 gpioDisable ()
 gpioEnable (mode)  /*0=high lvl; 1=rising edge; 2=low lvl; 3=falling edg*/
 gpioLevelSet( level )
 gpioStatus () /* status of PCCChip2 gpio (interrupt pin) */
 gpioMode(0=input,1=output) /* when not using J3Pin19 for interrupt */
 gpioSet( 0 or 1 ) /* returns -1 if not in output mode */
 gpioV{,3,1}Status       /* level of VMEChip2 gpio{3,1} ==> J3pin{18,16} */
 gpioV{3,1}Mode    /* set mode of VMEChip2 gpio{3,1} ==> J3pin{18,16} */
 gpioV{,3,1}Set /* for gpioVSet (returns 0-3), 3pin18 = most sig. bit */
 clkxConnect(routine, arg)      /* x can be 1 or 2 */
 clkxDisable ()
 clkxEnable ()
 clkxLevelSet(level)            /* level can be 0 thru 7 */
 clkxRateGet ()
 clkxRateSet (ticksPerSecond)
 clkxStart()
 clkxStop()
 LMxConnect(routine, arg)       /* x can be 0 or 1 */
 LMxDisable()
 LMxEnable()
 LMxLevelSet(level)
 LMCheck(LM)                    /* LM can be 0, 1, 2, or 3 */
 LMAddressSet(16bitaddr)        /* only upper byte is valid, lower */
                                /*  byte are preset - LM0:F1, 1:F3,*/
                                /*                      2:F5, 3:F7 */
/*NOTE: routines write same reg*/
 LMSIGAddressShow()
 SIGConnect(sig, routine, arg)  /* sig can be 0, 1, 2, or 3 */
 SIGEnable(sig)
 SIGDisable(sig)
 SIGLevelSet(level)
 SIGAddressSet(16bitaddr)       /* only upper 3 nibbles are valid, -  */
                                /*      low nibble is preset to 2, -  */
                                /* value xxFx disables the map decoder*/
/*NOTE: routines write same reg*/

 mx(addr)                       /* x can be b, w, or l */
 vmeSlaveMapShow()
 vmeMasterMapShow()
 interruptLevelShow()
 vmeSlaveMapx(am,vmestart,vmeend,lbstart)       /* x can be 1 or 2 */
                        NOTE: the window size (vmeend-vmestart)
                        determines the boundaries that vmestart
                        and lbstart must begin on.
 vmeSlaveMapxAm(am)                             /* x can be 1 or 2 */
 vmeSlaveMapxSnWp(am)                           /* x can be 1 or 2 */
 vmeMasterMapx(am,vmestart,vmeend,width)        /* x can be 1, 2, or 3 */
 vmeMasterMap4(am,vmestart,vmeend,lbstart,width)
                        NOTE: the window size (vmeend-vmestart)
                        determines the boundaries that vmestart
                        and lbstart must begin on.
 vmeMasterIoMap2(am_mode)
 vmeMasterIoMap1(am,width)
 vmeMasterMapxAm(am)            /* x can be 1, 2, 3, or 4 */
 vmeMasterMapxWp(enable)
 vmeBR()
 dmaShow()
 dmaConnect (routine, arg)
 dmaDisable()
 dmaEnable()
 dmaLevelSet(level)
 dmaStart(vmeadd,localadd,bytecnt,dir,dtb_mode,non_blk_am)
 dmaStatus()
 dmaDone()
 dmaThrottle()
 dmaSn()
 dmaChainSn()
 dmaVmeBR(level)
 dmaStartChain(addr)
 cacheControlShow()
 vmeIackedConnect(routine,arg)
 vmeIackedDisable()
 vmeIackedEnable()
 vmeIackedLevelSet(level)
 addressTranslationShow(force)  /* 0=show page tables if enabled; */
                                /* 1=show tables                  */
 addressTranslationRoot(page_size)  /* 1=8K pages; 0=4k pages */
 addressTranslationEnable(upper_mem)  /* 0x80000000,0xc0000000... */
                                      /* ...0xfc000000,0xfe000000 */
                                      /* 0=0xff000000             */
 addressTranslationDisable()
 addressTranslationClear(log_start,log_end)
 addressTranslation(log_start,log_end,phys_start,cache_mode,
                                                     write_prot,expert)
                             cache_mode=0 => writethrough
                             cache_mode=0 => copyback
                             cache_mode=0 => cache inhibit, serialized
                             cache_mode=0 => cache inhibit, nonserialized
                             write_prot=1 => protect pages
                             expert => entire page descriptor (don't use)
addressTranslationdtt0(base,mask,enable,cm,wp,expert)
```

Appendix A

devel Release Notes

This is a development version.

A.1 bug reports

The code used to create the ”support” object module includes the file $VXWORKS_DIR/config/mv167/mv167.h. I found a bug in that object module and have fixed it. (See note in VxWorks notes conference).

Appendix B

v1_3 Release Notes

B.1 flavor M68k

Sources were updated to reflect the changes from VxWorks v5_0_2b to VxWorks v5_1. These changes are meant to be backward compatible with VxWorks v5_0_2b. Version v1_3 of VX_TOOLS was built against dependent products under the umbrella product vxprods v5_0.

B.2 flavor R3k

Sources were modified to fix problems that generated errors or warnings from the native IRIX compilier with the fullwarn switch. Some warnings could not be addressed because of minor problems with the VxWorks v5_0_5 include files.

B.3 New Features

- A new script has been added to the $VX_TOOLS_DIR/all/bin area named vxsetod. This script can be used to set the time and date on a VxWorks node from a Unix node using vxRsh.
- VxWorks v5_1 includes a new library ansiTime that contains routines that collide in name space with routines in the VX_TOOLS time of day librarys. The VX_TOOLS time of day librarys have the advantage of including routines to set the time and date. The VX_TOOLS time of day routines are not affected if they are loaded over the VxWorks v5_1 ansiTime library.
- The VME interrupt routine interface has changed for several routines - see VME interrupt section for details.

Appendix C

v1_3_1 Release Notes

C.1 flavor M68k

This version was built with VxWorks version v5_1_1. Product dependencies were resolved with vxprods v5_1. There were no changes made in the product other than those made in the build and premake files to recognize the new VxWorks version.

Appendix D

v1_4 Release Notes

D.1 flavor M68k

This version is built under VxWorks version v5_1_1.
New features are the following:

- DVX Dart Vxworks eXtensions facility that includes the atexit function implementation and an alternative task variable package to the one provided by VxWorks.
- A TRACE facility, a debugging aid to trace execution of code through printed information in a similar fashion as using printf's statements. A very significant aspect of TRACE is that the user can have two MVME 167 boards in the same crate, and use one to trace what happended in the other in the case of a crash for which the second board is hung. The current TRACE implementation is supported only for the MVME167.
- A new enlarged version of the document.

Appendix E

v1_5 Release Notes

E.1 flavor M68k

This version is built under VxWorks version v5_1_1.
New features are the following:

- The fscc support software is implemented for the pc4a; no pc4 is available from this version or greater.
- DVX Dart Vxworks eXtensions facility includes now an alternative way to register exit handlers besides the use of the atexit function. The dvx_atexit function allows the user to pass an argument to the exit handler; this is not forseen under the standards for the "UNIX style" atexit call.

Appendix F

v1_6 Release Notes

F.1 flavor M68k

This version is built under VxWorks version v5_1_1.
This version has no new features, just bug fixes in the dvx exit handlers.