Contents

5. I/O MODULES

V6M6 Reference Manual

License Agreement

Warranty

Trademarks

1. Introduction

The V6M6 is a 6U VME carrier for up to six plug-in modules. The V6M6 and its modules together occupy a single VME slot.

A variety of module types are available to perform general purpose processing, digital signal processing, and I/O interface functions. Any mix of module types may be installed on the V6M6, subject to certain physical limitations such as I/O connector accessibility.

The six module sites on the V6M6 are interconnected by two buses. The first is a PCI bus, electrically compatible with the standard PCI bus used in PC applications. The second bus is a proprietary Time Division Multiplexed (TDM) bus typically used to carry isochronous (constant bit rate) data such as digitized audio.

In addition to the six module sites interconnected by PCI and TDM buses, the V6M6 includes:

- DRAM controller and 0, 32MB, or 64MB of DRAM attached to the PCI bus
- VME bridge attached to the PCI bus
- Configuration microcontroller to handle V6M6 initialization
- TDM bus controller to manage TDM bus activity
- Programmable clock synthesizers to generate PCI and TDM clocks
- 7 LEDs to display VMEbus activity and module-specific status
- Board reset and VME enable switches
- TDM bus front-panel expansion connector
- Connector to provide VME backplane P2 connector access to one of the six module sites

In addition, the "S" option of the V6M6 (i.e. V6M6/S) includes an interface between the TDM bus and the standard VITA 6-1994 SCSA bus on the VME P2 connector.

The host (VME) interface provides PIO ports to access the module sites and global memory. Each PIO port is implemented as a set of three registers: a Control / Status Register, an Address Register, and a Data Register. Access to these registers is controlled via the Unix device interface by a special device driver. A library of C–callable functions, the V6M6 Host Application Library, is provided for application programmers to access and utilize the interface for data transfer and module control operations.

2. V6M6 Installation

This chapter describes the hardware and software installation for the V6M6. It is organized into the following sections:

2.1 Overview of the V6M6 Hardware and Software

The CAC V6M6 embedded processing boards provide module sites which can carry various types of processing modules. The V6M6 can hold up to 6 modules.

In addition to the module sites, the V6M6 baseboards include:

- one or more banks of global DRAM memory.
- a TDM subsystem controller.
- a VME interface controller.
- a micro controller and Flash ROM for storing and loading FPGA configurations.

The module sites, global DRAM, and VME interface are connected via a local PCI bus which allows the host system and processor modules to communicate. The TDM subsystem and micro controller communicate separately with the host. The PCI local bus gives rise to the terminology used in naming the files and devices. The device driver prefixes and many of the host programs use the string, "pci".

Figure 1 is a general outline of the V6M6 showing the location of the module sites, and VME address switches.

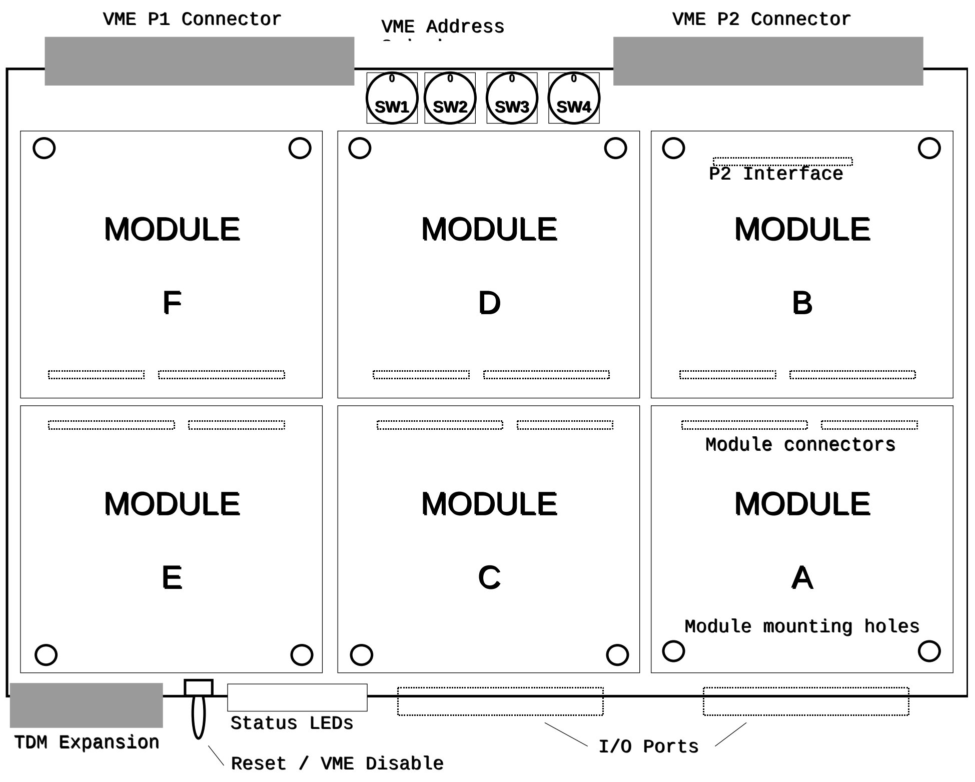


Figure 1: V6M6 Outline

2.1.1 Resources and VME Address Space

The V6M6 occupy 128KB of VME address space. This space is divided into 10 individual addressable resources:

- eight PIO resources
- a resource for the TDM subsystem and micro controller
- a resource for the VME interface.

Six of the eight PIO resources correspond to physical module sites and global memory. The remaining two may be used to access global memory.

The TDM resource is divided into three sections:

- the TDM map RAM
- the TDM controller
- the configuration micro controller.

The VME interface resource consists of various control and status registers.

2.2 Hardware Configuration and Installation

Various host platforms and operating systems will have differing methods of configuring and installing hardware. Much of the configuration information is specified as part of the device driver installation. This section describes how to configure the boards and VME backplane.

CAUTION: The V6M6 contains static sensitive components. Take precautions against static discharge whenever handling the boards.

2.2.1 VMEbus Address

The V6M6 may be configured to reside in the A32 or A24 VME address space. The installer must determine which address space is suitable for the host system to use and what addresses are available. The V6M6 occupy 128 Kbytes (0x00020000) of VME address space. Each board must be configured to occupy its own 128KB area.

The address switches (SW1–SW4 in figure 1) must be set to match the upper four hexadecimal digits of the selected address. This is the address at which the board will respond to accesses on the VMEbus.

The exception is bit 16 of the address (LSB of SW4). When this bit is 0, the board will respond to addresses in the A32 space. When it is 1, the board will respond in the A24 space.

On most SPARC architectures, A32 addresses in the range of 0x10000000 through 0x14F20000 are suitable. Check the documentation for the host system or VME adapter you are using to determine suitable address ranges for your system. Also determine the address space in use by other boards installed in the system.

Once the address space for the board is determined, set switches SW1–SW4 to select the address. For example, if the base address of a board is to be 0x10200000, the switches should be set as follows:

SW1=1   SW2=0   SW3=2   SW4=0

When installing multiple boards, be sure that each board's address is 0x00020000 from each other. In other words, the settings of SW4 should increment by two.

2.2.2 Interrupts and VMEbus MASTER

Parameters associated with VME interrupts and VMEbus master operation are configured by the device driver. The values for these parameters are chosen as part of the device driver installation.

2.2.3 VME Backplane Configuration

There are five jumpers associated with each slot on the VME backplane which are of concern to hardware installers. They are clearly labeled (IACK, BG0, BG1, BG2 and BG3) on some VME backplanes. Consult the documentation for your host system to determine the location of these jumpers.

First is the Interrupt Acknowledge daisy chain jumper. This jumper must be left open for slots that have a V6M6 installed. It must be closed for any unoccupied slots.

The other four jumpers are for the Bus Grant daisy chains—one for each bus request level. The V6M6 use one selected bus grant level at a time. The other bus grant signals are passed through. Therefore all four of the bus grant jumpers should be open for slots that have a V6M6 installed. All four should be closed for any unoccupied slots.

2.2.4 Board Installation

The board may be inserted in the VME backplane once the board's address and the backplane's jumpers have been configured. When applying force to insert the V6M6 into the VME chassis, press on the front panel only near the ejection tabs or the mounting screw near the middle of the panel.

Align the board with the slot guides. Slide the board in until its connectors mate with backplane. Press the board in until the front panel seats against the front of the VME chassis.

2.2.5 Powering Up

When the VME backplane is powered on, or when the backplane or board is reset, the V6M6 begins a configuration sequence. During this sequence, the base board FPGAs and module FPGAs are configured. For most installed modules, the status LED for each installed module will be red until the module is configured. Once configured, the LED will either turn green or go off, depending on the module type.

On boards with microprocessor program versions 12 or below, the FPGAs on the modules are automatically configured after the base board FPGAs. This has been found to cause problems configuring certain FPGA types due to timing of the VME Reset signal generated by some VME host controllers. This problem may cause the configuration of one or more modules to fail. This is usually corrected by resetting the V6M6 board(s).

Starting with microprocessor program version 13, released November 21, 1995, the FPGAs on the modules are not configured until commanded by either the device driver or the pciinit program.

2.3 Software Installation

Note: The software for CAC's Unix-based products is organized into the directory hierarchy in Figure X below. Only the major branches, relating to the V6M6, are illustrated.

```
$CAC       - top level directory for all Unix-based products.
    |          The environment variable, CAC,specifies the name
    |           of the top level. For example, /usr/cac.
    |
    |- /bin       - support programs used by all products such
    |               as dspinit, pciinit, sbdspinit, tdmc, etc.
    |               and the host servers and ushell for the
    |               MIPS kernel.
    |
    |- /include   - header files for host application programs
    |               written for all products (e.g. pciutil.h)
    |
    |- /lib      - library files for all products.  These
    |               include the host application libraries
    |               such as libpci.a as well as special MIPS
    |               code, FPGA config files, VME
    |               memory map for V6M6 MIPS Kernel, etc.
    |
    |- /pci       - source files and diags for the V6M6 and V9M12
    |    |          products excluding the MIPS Kernel, libraries
    |    |          and server.
    |    |
    |    |-- /libsrc  - Source for host library.
    |    |
    |    |-- /binsrc  - Source for host support programs.
    |    |
    |    |-- /pci*dev - Various directories for host-dependent
    |    |              device driver source and other files.
    |    |
    |    |-- /diag    - V6M6/V9M12 diagnostic programs
    |    |
    |    |-- /flash   - FPGA configuration and microcontroller
    |    |              code files that are loaded into flash.
    |    |
    |    |-- /mipsgcc - Gnu libraries and header files and source
    |                     for R3081 and R4600 programs.
    |
    |- /mips      - the MIPS kernel and server source for
    |               the VME6U6/R3081 mezzanine combination
    |               and the V6M6 MIPs processor modules.
    |               The directory also contains the MIPS
    |               application libraries.
    |
    |- /man       - on-line manual pages
    |
    |- /misc      - a place for user-created files such as macros
    |                for the dsptest and pmtest diagnostics, or
    |                special test programs.
    |
    |- /dsp       - source files, diags and demos for the
    |                VME6U6 and VME9U12 products.
    |
    |- /sbdsp     - source files, diags and demos for the
                     SB32C2 products.
```

The V6M6 Host Software is distributed as a tar file named **cacpci_*version*.tar** (where *version* is a version number. For example, cacpci_1.3.2.tar). This includes the device drivers, host application library, host support programs, diagnostics, and demonstration programs.

The MIPS Kernel software is distributed as a tar file named **cacmips_*version*.tar** and includes the WM Kernel with the WM Application Library for MIPS applications and the Host MIPS Kernel Server.

Software distribution archives are distributed on QIC150 tape or CDROM. The host applications and libraries are not compiled in these archives. CDROM distributions also contain the software extracted hierarchically. You may copy individual files, if necessary from these directories on the CDROM, itself.

Software releases and updates are also available on the Internet. On the World Wide Web at:

http://www.cacdsp.com or
http://www.wmi.com/soft_updates

For FTP access:

ftp.wmi.com/pub/pci
ftp.wmi.com/pub/mips

All distribution tar files are to be extracted onto your system from the top level CAC directory named by the environment variable, **$CAC**.

The software should be installed by the Superuser, **root**. It is assumed, for the following instructions, that the installation is performed using the C-shell. Certain commands differ depending on whether the installation is performed under SunOS 4.1.x or Solaris 2.x. Where necessary, alternate commands are shown. The example commands also assume that the top level directory is named **/usr/cac**.

The software installation requires approximately 16 Mbytes of disk space once compiled. Some of this space may be freed by removing the distribution tar files after installation is complete.

2.3.1 Installation Procedure

Below is the procedure for installing the V6M6 host software and MIPS Kernel software.

1. Define the environment variable **CAC** to be the name of the top level directory:

   In the C-shell:

   ```
   # setenv  CAC  /usr/cac
   ```

   In the Bourne shell:

   ```
   # CAC=/usr/cac
   # export  CAC
   ```

2. Create directory named in step 1 and make it accessible to users:

   ```
   # mkdir  $CAC
   # chmod  a+rx  $CAC
   ```

3. Extract the V6M6 host software archive from the distribution media.

   a. For Solaris 2.x from QIC150 tape:

   ```
   # tar  xvf  /dev/rst/0  cacpci_version.tar
   # tar  xvf  cacpci_version.tar
   ```

   b. For SunOS 4.1.x from QIC150 tape:

   ```
   # tar  xvf  /dev/rst0  cacpci_version.tar
   # tar  xvf  cacpci_version.tar
   ```

   c. For SunOS 4.1.x or Solaris 2.x from CDROM:

   Note: It may be necessary to mount the ***/cdrom*** file system prior to accessing the CDROM.

   ```
   # mount /cdrom
   # tar  xvf  /cdrom/cacpci_version.tar
   ```

4. Extract the MIPS Kernel Software tar file archive from the distribution media. Then extract the files from the archive:

   a. For Solaris 2.x from QIC150 tape:

   ```
   # tar  xvf  /dev/rst/0  cacmips_version.tar
   # tar  xvf  cacmips_version.tar
   ```

b. For SunOS 4.1.x from QIC150 tape:

```
# tar  xvf  /dev/rst0  cacmips_version.tar
# tar  xvf  cacmips_version.tar
```

c. For SunOS 4.1.x or Solaris 2.x from CDROM:

```
# tar  xvf  /cdrom/cacmips_version.tar
```

5. Compiling the V6M6 core software: If you have not received a separate binary distribution or should it become necessary to recompile the core software, use the **make** command in the core software directory, **$CAC/pci**. The following commands will force a complete recompile of the core software. In the commands shown, replace *compiler* with the name of your compiler (e.g. gcc).

```
# cd  $CAC/pci
```

a. For Solaris 2.x

```
# make  OS=SOLARIS  CC=compiler  install
```

b. For SunOS 4.1.x

```
# make  OS=SUNOS  CC=compiler  install
```

6. Compiling the MIPS server and other host utilities: If you have not received a separate binary distribution or should it become necessary to recompile the MIPS host applications, use the **make** command in the mips software directory, **$CAC/mips**. In the commands shown, replace *compiler* with the name of your compiler (e.g. gcc) and replace *os* with SUNOS for SunOS 4.1.3 or SOLARIS for Solaris 2.x.

```
# cd  $CAC/mips
# make  OS=os  CC=compiler  install
```

This will compile either or both of the host servers depending on whether the V6M6 and VME6U6 software has been previously installed.

7. Continue with the V6M6 device driver installation in section 4.

2.3.2 Programmer and User Setup

Every user of the software should have the environment variable named **CAC** set to the name of the top-level software directory.

Programmers and other users of the V6M6 boards should include the program and, optionally, the diagnostic directories on their program search paths. The following should be added to the shell's **path** variable:

```
            $CAC/bin and $CAC/pci/diag
```

To access the online manuals using the UNIX **man** command, the **$CAC/man** directory should be added to the MANPATH environment variable.

For example:

```
     setenv MANPATH
     /usr/man:/usr/local/man:/usr/openwin/man:$CAC/man
```

Users of the MIPS Kernel and Host Server should set the following environment variable:

```
            setenv  S3MIPSPATH  $CAC/mips/bin
```

2.4 Device Driver Installation

The procedures for installing device drivers under Solaris 2.x and SunOS 4.1.x are described in this section. The files for Solaris 2.x driver are in the directory, **$CAC/pci/pcisoldev**. The files for the SunOS 4.1.x driver are in the **$CAC/pci/pcisundev** directory.

The following parameters are involved in configuring the device driver into your host system. Some are specified in system configuration files and others are specified in a header file compiled with the V6M6 device driver.

The following parameters are defined in system configuration files.

Device name

Each board is named `pciN`, where *N* is an integer starting at zero. For Solaris 2.x installation, the generic device name, `CAC,pci`, is also specified.

VME Address

This is the physical base address of the board as determined and configured during the hardware installation.

Interrupt Vector

This is a hexadecimal value between 0x00 and 0xFF. Most systems reserve a range of values for user-installed hardware. Each V6M6 board must have a unique interrupt vector. Consult the manuals for your host system to determine the range of valid values. Also review the parameters of other installed devices to determine any values that may already be in use.

Interrupt Level

This parameter determines which VME interrupt line is used by the board and the priority level at which the operating system services interrupts from the board.

The interrupt priority level can dramatically affect system performance. Lower interrupt level values correspond to higher priorities. The priority for V6M6 boards should not be set so high as to interfere with the operation of other devices such as disk drives or network interfaces. Consult the relevant sections of your system's manual to determine recommended settings.

The following parameters are defined in the file, **pcidrv_options.h**. They are specified as C preprocessor macros and their values become hard coded in the device driver module.

VMEBREQ_SELECT

This macro specifies which of the four sets of VMEbus request and grant lines is to be used by the V6M6 boards. Consult the manual for your host system to determine which bus request lines are recommended. Bus request 3 is commonly used, especially in systems that do not implement prioritized bus arbitration The bus request parameter is specified in the file named **pcidrv_options.h**.

MASTER_ADDR_MOD

This macro specifies the address modifier used by the V6M6 boards when they are bus master. The default setting of 0x0D accesses the VME A32 address space.

PCIVME_DEFAULT_XLATE

This macro specifies the default address translation used by the V6M6 boards to convert a PCI Bus address to a VMEbus address. The value is a 24-bit word containing the translation MASK in bits 0:11 and the translation BASE offset in bits 12:23. A PCI address is translated to the VMEbus as:

```
VME_Addr[31:20]  =  ( PCI_Addr[31:20] & MASK ) | BASE
VME_Addr[19:2]   =  PCI_Addr[19:2]
VME_Addr[1:0]    =  0
```

The address translation may be modified on a per-board basis using the PCI_VMEXLATE ioctl request.

PCIVME_DEFAULT_MODES

This macro specifies the default settings for various modes of the PCI / VME interface. Currently, four bits are significant:

```
Bit 0     enable PCI data parity checking when set
Bit 1     enable PCI address parity checking when set
Bit 2     enable the VME interface to accept PCI memory
          transactions as accesses to the VMEbus
Bit 3     0 = VME Master Release-On-Request mode
          1 = VME Master Release-When-Done  mode
```

The PCI / VME interface modes may be modified on a per-board basis using the PCI_VMECONFIG ioctl request.

Other Macros

There are other, less important, macro parameter settings described in the **pcidrv_options.h** header file.

2.4.1 Solaris 2.x Driver Installation

The device driver and configuration files for Solaris 2.x are distributed in the **$CAC/pci/pcisoldev** directory. The files mentioned herein are in this directory. The installation commands should be run from within this directory.

2.4.2 Configuration Information

The distribution includes two configuration files which may need to be modified to describe the installation on your system. The **pcidrv_vme.conf** file should be used for systems in which the CPU is attached directory on the VMEbus. For Sbus systems using an Sbus-to-VME adapter from Solflower, use the **pcidrv_sfvme.conf** file. These files specify information about the V6M6 PCI boards to be installed in your system and contain the following entries for each board:

pcidrv_vme.conf

```
name="CAC,pci"
class="vme"
reg=0x4d,0x10200000,0x20000
interrupts=3,0xD0
```

pcidrv_sfvme.conf

```
name="CAC,pci"
parent="sfvme"
reg=0x4d,0x10200000,0x20000
interrupts=3,0xD0
```

The items shown in boldface type above are the values that you will probably need to change to match your system's configuration. All the parameters are specified on a single line for each board in the file. The order in which the lines appear determine how the device entries are numbered for each board - pci0, pci1, etc.

The second value of the “reg” field specifies the **VME address** for each board. This value must:

- be modulo 0x20000
- match its upper four digits with the address switch settings on the board
- lie within the addressable A32 VME space of the host system.

The first value in the "interrupts" field specifies the **interrupt level** for interrupts generated from the board. The second value in the "interrupts" field specifies the board's **interrupt vector**.

Note: If you are updating an existing V6M6 installation, the configuration information from **/kernel/drv/pcidrv.conf** should be copied into the appropriate configuration file before running the driver installation.

2.4.3 Installing the Driver

The device driver module, **pcidrv**, may be already compiled in the distribution using the default option settings in the file, **pcidrv_options.h**. However, there is no guarantee that the code was compiled for the proper platform architecture or operating system version on your system. To be sure the driver is initially rebuilt for your system begin the installation by removing any existing compiled driver
in the **$CAC/pci/pcisoldev** directory:

```
#  make  clean
```

Once the appropriate configuration file has been modified, the driver can be installed with the command:

```
#  make  install-vme
```

or

```
#  make  install-sfvme
```

depending on the system type (direct VME access or a Solflower Sbus-to-VME adapter). This will recompile the device driver module, if any changes have been made to the **dspdrv_options.h** file. It may be necessary to specify the C compiler on your system. If so, use the command:

# make CC=*compiler* install-vme

or

# make CC=*compiler* install-sfvme

Where *compiler* is the name of the compiler program and any required arguments. For example:

# make CC=*compiler* install-vme

The compiled driver module, **pcidrv**, and appropriate configuration file will be copied to the **/kernel/drv** directory. The configuration file will be named **pcidrv.conf** in the **/kernel/drv** directory.

The **make** command will update the file, **/etc/devlinks.tab** and then attempt to add the driver to the system. This final action may fail if a previous V6M6 driver is already installed. If the installation was performed only to update the driver module, the errors can be ignored. However the new driver will not be installed until the system is rebooted.

Once the driver is installed and operational (possibly requiring system reboot) test the system by running the command:

```
#  pciinit   -rZ  pci0
```

The names of additional boards may be appended to the command line to test other V6M6 boards in the system. You may access the **pcinit** information using the UNIX man program if you have added $CAC/man to the MANPATH as in the setup procedures in *Section Programmer and User Setup*. Copies of these online manuals are also available in Appendix B.

The **pciinit** program must be run whenever the system is rebooted. Once the system has been tested it can be configured to automatically initialize the boards as part of the system boot process. Edit the file, **pciinit.sh**, replacing the directory name, "/usr/cac" with the name of the installation directory you have chosen. This name appears in three places near the end of the file. Then run the following command to install the file in the system’s start-up directory.

```
#  make  install-rc
```

2.4.4 SunOS 4.1.x Driver Installation

With SunOS 4.1.x (Solaris 1.x) the V6M6 device drivers must be compiled into the kernel. The basic instructions described here should be sufficient for veteran kernel builders. If you have never rebuilt the kernel before please refer to the Sun Microsystems *System Administration Manual* for information on building new kernels. If a SPARCstation 2 and Solflower Sbus to VME adapter is used, consult the Solflower documentation for additional kernel configuration information.

Note: Be sure to save a copy of the existing kernel, **/vmunix**, in case a problem arises with the new kernel.

The device driver installation procedure is listed below. If you are installing the V6M6 device driver for the first time, begin with step 1. When updating the device driver or adding new boards to the configuration, start at step 5.

1. Create a symbolic link to the pcisundev directory in the Kernel system directory:

   ```
   #  cd  /usr/kvm/sys
   #  ln  -s  $CAC/pci/pcisundev  pcidev
   ```

   This creates a symbolic link to the PCI driver source code in the location searched by kernel build procedure. Future driver code updates will automatically be picked up by this link.

2. Edit the system device table C source file, **conf.c**, to include entry points for the driver. Use whatever editor you prefer.

   ```
   #  vi  /usr/kvm/sys/sun/conf.c
   ```

   The file, **$CAC/pci/pcisundev/pci_conf.c**, contains the two sections of code to be added to **conf.c**. The first section is declarations and macros starting with #include <pci.h>. These 13 lines should be added just before the declaration of the **cdevsw** array. The second section is five lines to be added to the end of the **cdevsw** array. Take note of the index associated with the new element—this is the major device number associated with the V6M6 boards. Modify the comment in the code, replacing "Maj Dev Num" with the index number for reference.

It is only necessary to add one entry to the "cdevsw" array. All of the boards (up to 14 per system) use the same major device number.

3. Change to the machine specific configuration directory.

    # cd ../*<arch>*/conf

   where *<arch>* is the directory specific to machine kernel architecture (and can be determined using the "arch" command).

4. Edit the file named **files** adding the following line at the end of the file to specify location of the device driver source software:

```
pcidev/pcidrv.c  optional  pci  device-driver
```

5. Also, in the machine specific configuration directory,

    # cd ../*<arch>*/conf

   edit the site-specific configuration file for your system. If you do not already have a kernel configuration file specific to your system:

    # cp GENERIC *<sys_name>*

   where *<sys_name>* is your system's kernel name.
   Edit the new configuration file:

    # vi *<sys_name>*

   For each V6M6 board installed in the system, add lines at the end of the configuration file in the following format:

```
device pci0 at vme32d32 ? csr 0x10200000 priority 3 vector pciintr
    0xD0
```

   The items shown in boldface type are the values that you may need to change to match your system's configuration.

   The number of the pci device will increment for each device line added. The value following "csr" specifies the **VME address** for each board. This value must a) be modulo 0x20000, b) match its upper four digits with the address switch settings on the board and c) lie within the addressable A32 VME space of the host system.

The value following "priority" specifies the **interrupt level** for interrupts generated from the board. The value following "vector pciintr" specifies the board's **interrupt vector**.

Example configuration entries can be found in the file:

```
$CAC/pci/pcisundev/pci_config_file
```

6. Run the system configuration program for the new or modified configuration file. For native VME systems:

   ```
   #  /etc/config  <sys_name>
   ```

   For Solflower Sbus to VME adapter systems:

   ```
   #  sfconfig  <sys_name>
   ```

7. Build a new kernel using the new configuration information.

   ```
   #  cd  ../<sys_name>
   #  make
   ```

   This will compile all the necessary system files, including the V6M6 device driver code (**pcidrv.c**) and create a new **vmunix** file in the current directory.

8. Install the new kernel.

   ```
   #  mv  /vmunix  /vmunix.old        (to backup old kernel)
   #  mv  vmunix  /vmunix             (to copy new kernel)
   ```

9. Create device node entries for the V6M6 boards.

   ```
   #  cd  $CAC/pci/pcisundev
   #  makepcidev  <major num>  <num boards>
   ```

   Replace <*major num*> with the major device number determined in step 2. If unsure, view the file, **/usr/kvm/sys/sun/conf.c**, and determine the index of the pci entry point structure in the "devsw" array. Replace <*num boards*> with the number of boards installed.

   The **makepcidev** program will remove any existing pci entries in the **/dev** directory and recreate them using the major device number and number of boards specified.

10. Reboot the system to bring up the new kernel:

```
# /etc/fastboot
```

11. Test the system by running the command:

```
# pciinit  -rZ  pci0
```

The names of additional boards may be appended to the command line to test other boards in the system. **pciinit** is described in Appendix B and in the online manuals. See **Programmer and User Setup** to access the online manuals.

12. Edit the system start-up script:

The **pciinit** program must be run whenever the system is rebooted. Once the system has been tested, it can be configured to automatically initialize the boards as part of the system boot process. Edit the file, **/etc/rc.local**, and add the following line:

```
/usr/cac/bin/pciinit  -rZ  -d /usr/cac  all
```

Environment variables are not active in the start-up script. The **-d** option tells **pciinit** where to look for the memory and processor initializer code that is run on the processor modules. Replace **/usr/cac** with the absolute pathname associated with the environment variable CAC defined during the installation.

2.5 Adding and Updating FPGA Configurations

The FPGAs on the V6M6 baseboard and modules are configured by the micro-processor from data stored in flash memory. The microprocessor executes code that also resides in the flash memory.

The flash memory is divided into 8 erasable sectors and 16 writable and readable segments. Most of the objects fit in a single flash segment. Below is a list of the current objects. The object names are consistent with the names of the files where the data is distributed and stored on disk, which are appended with ".mcs".

	Flash		
Object Name	Sector	/ Segment	Description
pm3081*xx*	0	0	PM3081 module configuration
mm32*xx*	1	2	MM32 memory module configuration
pm4600*xx*	2	4	PM4600 module configuration
memctl*xx*	6	13	Baseboard memory controllers configuration
progtdm*xx*	7	14	μ-processor prog and TDM / μ-proc interface configurations
vmepcib*xx*	7	15	VME interface configuration
For version 1 V6M6			
vmepciaxx*xx*	7	15	VME interface configuration

The characters, "*xx*", in the object names represent the version number of the object. The object files are kept in the directory, **$CAC/pci/flash**. The program, **pciflashup**, is used to update current flash objects. The program is run with arguments specifying the board or boards to be updated. For example:

```
    %  pciflashup  all
```

Or for a single board:

```
    %  pciflashup  pci0
```

Note: The directory, **$CAC/bin**, must be on the program search path.

The amount of user interaction required depends on the currently installed version of the micro-processor program. The *boardname* argument may be replaced with the word *all* to have **pciflashup** run for all the V6M6 boards found in the system. Normally **pciflashup** only rewrites flash objects that have newer versions on disk in the flash directory. The **-f** option forces it to rewrite all the objects with the current versions on disk.

The **pciflashup** program may be run with the **-s** option to show the versions of the objects in the flash and in files without actually updating the flash.

2.6 Installing and Removing Modules

Modules should never be added or removed from a V6M6 board while the board is installed in a system. Remove the board from the system and lay it on a flat, sturdy, and anti-static surface.

Modules are shipped with mounting spacers attached to the two-module mounting holes. To add a module to a V6M6 board, align the module so its two thin connectors and spacers are in line with the connectors and mounting holes on the baseboard. Press the module firmly until the connectors snap together. Use the two screws supplied with the module to secure the module from the bottom side of the baseboard.

Note: It is recommended that the baseboard's flash objects be updated before adding a new type of module that was not previously installed on the baseboard.

When removing a module, first remove the two mounting screws from the bottom side of the base board. Keep these screws with the module. Gently pry the module at the connector edge until it releases from the baseboard connectors. It is not necessary to modify the flash memory when modules are removed.

3. V6M6 Baseboard

This chapter describes the baseboard for the V6M6. It is organized into the following sections:

3.1 Overview

The V6M6 is a 6U VME carrier for up to six plug-in modules. The V6M6 and its modules together occupy a single VME slot.

A variety of module types are available to perform general purpose processing, digital signal processing, and I/O interface functions. Any mix of module types may be installed on the V6M6, subject to certain physical limitations such as I/O connector accessibility.

The six module sites on the V6M6 are interconnected by two buses. The first is a PCI bus, electrically compatible with the standard PCI bus used in PC applications. The second bus is a proprietary Time Division Multiplexed (TDM) bus typically used to carry isochronous (constant bit rate) data such as digitized audio.

In addition to the six module sites interconnected by PCI and TDM buses, the V6M6 includes:

- DRAM controller and 0, 32MB, or 64MB of DRAM attached to the PCI bus
- VME bridge attached to the PCI bus
- Configuration microcontroller to handle V6M6 initialization
- TDM bus controller to manage TDM bus activity
- Programmable clock synthesizers to generate PCI and TDM clocks
- 7 LEDs to display VMEbus activity and module-specific status
- Board reset and VME enable switches
- TDM bus front-panel expansion connector
- Connector to provide VME backplane P2 connector access to one of the six module sites

In addition, the "S" option of the V6M6 (i.e. V6M6/S) includes an interface between the TDM bus and the standard VITA 6-1994 SCSA bus on the VME P2 connector.

The host (VME) interface provides PIO ports to access the module sites and global memory. Each PIO port is implemented as a set of three registers: a Control / Status Register, an Address Register, and a Data Register. Access to these registers is controlled via the Unix device interface by a special device driver. A library of C–callable functions is provided for application programmers to access and utilize the interface for data transfer and module control operations.

Figure 1, below, is a general outline of the V6M6 showing the location of the module sites, and VME address switches.

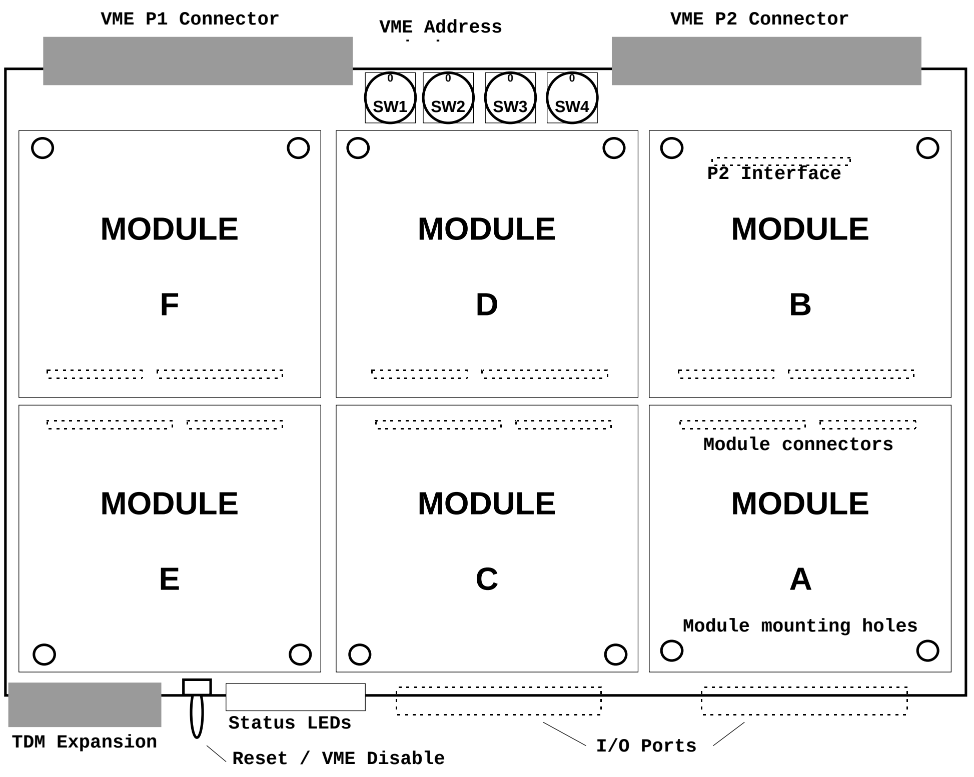


Figure 1: V6M6 Outline

3.2 Primary Components

MODULE A-F	The PCI Module Sites. These are the locations of the modules on the baseboard.
VME Connectors	Connectors P1 & P2 plug into the VME backplane.
SW1 - SW4	These select the VME address for the board. Their settings correspond to the upper 4 hex digits of the address (SW1 is the most significant).
TDM Expansion	This port connects via 20-pin ribbon cable to the TDM expansion ports of other V6M6 or VME6U6 DSP boards to combine TDM subsystems.
Reset / VME	This dual function switch is used to reset and reconfigure the V6M6 and to disable its VME interface. Toggling the switch towards the TDM connector (momentary position) will initiate a reset cycle. Toggling the switch towards the LEDs (stationary position) disables the VME interface.
Status LEDs	One tri-color LED for each module and one for the VME interface. These LEDs may take on three color states, RED, GREEN or AMBER, as well as OFF. The use of the module status LEDs is dependent on the module type installed. The VME status flashes green to indicate slave access and red when the V6M6 is VME master.
I/O Ports	Module sites A and C may have an I/O port accessed from the front panel. For example, RJ45 connectors for T1 or E1, special multi-channel audio connectors or serial ports for MIPS processors. Various front panel inserts are available for the different I/O connectors.

3.3 Configuration Microcontroller (CM)

The logic on the V6M6 baseboard, and on most of its plug-in modules, is implemented in field programmable gate arrays (FPGAs). The FPGAs used are primarily Xilinx and Lucent Technologies devices based on SRAM technology. These devices must be initialized with a configuration bit stream before they become functional.

The primary responsibility for configuring the FPGAs on the baseboard and on installed modules rests with a microcontroller subsystem on the V6M6 baseboard called the Configuration Microcontroller (CM). The CM is built around an 8051-compatible microcontroller from Dallas Semiconductor, the DS80C310. It does not rely on any FPGA logic itself, and so becomes functional immediately after power-up or board reset.

The CM's program is stored in two separate nonvolatile memories. The first memory is a 4 Mbit Flash EPROM, and the second is a standard 27C256 256 Kbit EPROM. These are described in later subsections. In addition, the CM owns a 1024-bit serial EEROM used to hold information such as desired PCI and TDM clock frequencies, baseboard serial number, and burn-in history.

Modules installed on the V6M6 identify themselves to the CM via a 1024-bit serial EEROM on each module. The EEROM contains a module-type indicator and other information. The CM queries the EEROM on each module to determine which FPGA configuration bit stream, if any, is required by the module. The serial EEROM is described in more detail in Section 3.3.3 Serial EEROMs.

The CM cannot be used as a general purpose processing resource on the V6M6. The only non-configuration functions performed by the CM are the generation of a 62.4 KHz timing signal distributed to the six module sites and control of the PCI and TDM clock synthesizer chips described later.

3.3.1 Microcontroller Flash Memory

The microcontroller's main memory is a 4 Mbit Flash EPROM that holds the current version of the CM's program as well as the current version of the configuration bit stream for each FPGA that the CM might need to initialize.

The V6M6 software distribution includes utility programs which allow the VME host computer to update the contents of the Flash memory. This could occur as a result of:

- Adding support for a new module type. If the new module includes an SRAM-based FPGA, a new configuration bit stream must be added to the Flash memory.
- Upgrading an existing FPGA configuration bit stream. This might be to correct a problem in a previous version or to add new functionality.
- Upgrading the CM program itself. This is rare, but is supported in case some new module design requires an enhancement to the configuring software.

3.3.2 Microcontroller EPROM Memory

The 27C256 EPROM is permanently mounted to the V6M6. It is preprogrammed when the V6M6 is manufactured with a basic CM program and with basic operating configurations for the FPGAs on the V6M6 baseboard.

The purpose of the EPROM is to have a way of getting the V6M6 operational enough to allow the Flash memory to be downloaded during initial board test. In addition, the EPROM allows recovery from a failed update of the CM program in Flash memory.

The EPROM includes the CM's reset vector, so the CM always begins operating from the program stored in the EPROM. The EPROM program first checks for a signature in the Flash memory to determine if the Flash has been initialized. If not, then the CM continues executing from EPROM and loads a minimum operating configuration into the V6M6 baseboard FPGAs. Once this configuration is complete, the VME host can communicate with the V6M6 to load current software and configuration bit streams into the Flash memory.

If the CM program in Flash has somehow been corrupted, a shunt can be installed on a header on the V6M6 baseboard to force the CM to execute only from the EPROM.

3.3.3 Serial EEROMs

Each V6M6 module must include a 1024-bit serial EEROM (93C46 style) to hold information about the module. The CM queries the 93C46 on each module to determine the module's need for configuration. It then finds the required configuration data in Flash and sends it to the module serially through the module's JTAG port.

The information held in the 93C46 varies depending on the type of module, but typically includes:

- Module type identifier
- PCI bus address type and size
- Hardware revision number
- Serial number
- Date serialized
- Burn-in hours
- Module options (e.g. memory size, clock speed)

This information is available to VME host software through functions in the V6M6 Host Application Library.

The CM itself has an additional 93C46 which holds the desired PCI and TDM clock frequencies, as well as information similar to that in the modules.

3.3.4 Host-Driven FPGA Configuration

For some possible module types, it is not feasible to store the FPGA configuration entirely in the Flash memory. For example, a module for "Configurable Computing" based on a large FPGA would, by its very nature, be subject to frequent changes to its functionality. It would not be reasonable to require new configurations to be downloaded to the Flash and then have the CM load them to the module.

Instead, the CM supports a download protocol which allows the VME host to send new configuration information through the CM and into the destination module FPGA. This transfer bypasses the Flash and can be repeated without resetting the V6M6.

3.4 PCI Bus

The major bus that ties together the elements of the V6M6 board is the PCI bus. It is electrically identical to the PCI bus version 2.1 standardized by the PCI Special Interest Group, commonly used in PCs. The mechanical provisioning—connectors and form factor—of the V6M6 PCI plug-in module is specific to CAC.

The V6M6 supports up to six plug-in PCI modules plus PCI-connected resources on the baseboard, including:

- Global DRAM controller(s). Rev 2 or lower baseboards may have zero, one or two DRAM controllers attached to the PCI bus, each of which can control 32 MB of DRAM. Rev 3 or higher baseboards have at most one DRAM controller which can control up to 64 MB of DRAM.

- VME bridge. This is the path between PCI-connected resources and the VMEbus.

- TDM controller and SCSA interface (Rev 3 and higher). The TDM controller on Rev 2 or lower is not PCI-connected, but instead is controlled solely through the VMEbus. Rev 3 and higher also supports the "S" option to connect the TDM bus to the SCSA bus on the VME P2 connector. The SCSA bus interface circuit is controlled via the same PCI interface as the TDM controller.

3.4.1 PCI Bus Clocking

The PCI specification places a limit of 10 loads on the PCI bus for 33 MHz operation. Traditionally, PCI electrical loading constraints count each plug-in module as two loads. By this counting method, the V6M6 PCI bus can have as many as 15 loads. The V6M6 limits PCI bus speed to 25 MHz max in order to overcome the excess loading and to ease the design task for FPGA-based PCI interfaces.

The PCI bus clock is generated by an ICS AV9110-01, a high resolution frequency synthesizer, controlled by the configuration microcontroller. The CM stores the desired frequency in its serial EEROM memory. The output of the synthesizer is buffered with a separate low-skew driver for each plug-in module and for each PCI interface on the baseboard. The individual clock lines are balanced for equal delay to each destination to minimize clock skew between all of the modules and on-board resources.

Note that the PCI clock line is NOT bused, as it is in typical PCI systems. A separate clock line is driven to each module. Each clock signal is terminated at its end-point with a series combination of a resistor (between 62 and 75 Ω) and a capacitor (between 100 pF and 220 pF) to ground. Module clock signals are terminated on the module, not on the baseboard.

The point-to-point distribution of PCI clock eliminates the requirement from the PCI specification of having only one load on the PCI clock signal on a plug-in module. Multiple loads are permissible as long as they are clustered within 1" of the clock signal terminator.

Because of the way that the clocks are distributed, there is very little skew between module clock signals at the module connectors. Therefore it is not necessary to maintain the precise 2.5" ± .1" clock signal length on the plug-in module, as called out in the PCI specification. The clock signal length on the module should be in the range of 1" to 2".

The PCI clock generator will never stop or slow to very low rates. Furthermore, its frequency cannot be changed without a hard reset of the V6M6 board. the V6M6 board. Therefore, it is permissible for plug-in modules to use phase-locked loops to regenerate or multiply the module PCI clock signal. Most CAC modules do, in fact, multiply up the PCI clock to create clocks on the module that are locked to the PCI clock. This greatly simplifies synchronization circuitry in the PCI interface FPGAs on the modules.

3.4.2 PCI IDSEL Signals

The IDSEL signals are produced separately for each module and for each baseboard PCI resource. During PCI configuration cycles, an active high IDSEL to a PCI resource causes that resource to respond to the configuration cycle.

The V6M6 produces IDSEL signals through resistive connections to specific PCI Address/Data lines. The following table shows the correspondence between PCI AD line and IDSEL for each resource.

PCI Resource	PCI AD Line
Module F	19
Module E	20
Module D	21
Module C	22
Module B	23
Module A	24
Mem2/TDM *	25
MemCtl 1	26
VME Bridge	27

* MemCtl 2 on Rev 1 or 2, TDM and SCSA Interface on Rev 3 and up

3.4.3 PCI Arbiter

The V6M6 includes an arbitration circuit that grants control of the PCI bus to PCI resources that request control. This arbiter accepts seven "Request" inputs: one from each module position and one from the VME bridge. It generates seven "Grant" signals back to the same resources.

The arbiter uses a rotating priority method of granting control to requesters. This assures that all PCI requesters are treated equally, and that each will be given control of the PCI bus within a predictable maximum time. The rotating priority scheme views the seven potential requesters in a fixed order:

VME	Mod A	Mod B	Mod C	Mod D	Mod E	Mod F

At any instant, one of these is considered the lowest priority potential requester and the others are considered incrementally higher priority in a circular fashion. The resource most recently in control of the PCI bus is considered the lowest priority. For example, if Module B most recently controlled the bus, then the priority assigned to each resource would be (with 0 lowest priority and 6 highest):

VME	Mod A	Mod B	Mod C	Mod D	Mod E	Mod F
5	6	0	1	2	3	4

The highest priority requester is given a Grant signal and will take control of the PCI bus immediately if no PCI transaction is in progress, or at the end of the current transaction. If no requests are present, the arbiter leaves the bus ownership "parked" on the last resource that controlled the bus.

3.4.4 PCI Interrupts

Each V6M6 module site can source two discrete interrupt request lines. These correspond to INTA/ and INTB/ interrupt signals in the PCI specification. Note that INTC/ and INTD/ are not supported on the V6M6.

The INTA/ and INTB/ signals are not common or bused as in many PC applications. Each is separately wired to an interrupt routing circuit. This circuit can selectively enable the twelve interrupt requests to generate a VMEbus interrupt or to generate interrupt requests back to the on-board PCI modules.

The VMEbus interrupt circuit has a mask register, described in Section 3.6 VME/PCI Bridge, to define which of the twelve PCI interrupt request signals will propagate through to the VMEbus. This interrupt path is supported in all revisions of the V6M6.

Revision 2 V6M6 boards contain another six bit mask register which allows selected PCI INTA signals from each module to propagate onto a single, global interrupt signal, "MODINT/", which is supplied in common to all six module sites. This signal is separate from the normal PCI bus signals and can be used on a module to generate an interrupt to a processor or DSP on that module. This is intended to allow I/O modules to generate interrupts directly to processor or DSP modules on the same V6M6.

Revision 3 V6M6 boards extend this on-board interrupt router by separating the MODINT/ signals to each module and by allowing both INTA and INTB from each module to participate in the routing. Additional control registers are included to allow selected interrupt requests to propagate to specific module MODINT/ signals. This is provided to better support interruptibility of multiple processor or DSP modules on a V6M6.

3.5 TDM Subsystem

The TDM bus on the V6M6 is a communication channel completely separate from the PCI bus.

The TDM bus is intended to carry low latency isochronous (constant bit rate) data efficiently, without burdening the PCI bus. Typical usage for the TDM bus is to carry digitized audio from one module to another or between off-board audio sinks/sources and modules on the V6M6.

Data passing between resources on a V6M6 assembly is routed through four serial time-division multiplexed (TDM) buses. They are called buses A, B, C, and D. These TDM buses run synchronously, each with the same bit clock, time slot definition, frame duration and frame sync pulse.

A connection established through the TDM system appears to be a dedicated link between the resources participating in the connection. Connections may be point-to-point or point-to-multipoint. That is, a single resource sources the data in a connection and any desired number of resources can receive the data.

The TDM buses interconnect the six module sites, a front-panel TDM expansion connector labeled "J4", and, on V6M6 Rev 3 and higher, a baseboard interface to an SCSA bus on the VME P2 connector.

The SCSA bus interface is used to connect TDM data streams between VME boards via an SCSA backplane on the P2 connector. This bus is a standardized method of exchanging telephony data between boards from various vendors. It does not require front-panel cables, and so is appropriate for uses where straightforward board replacement is important. Up to twenty one boards in a VME backplane may be connected together through this bus. For Rev 2 V6M6 boards, the SCSA bus is accessible through a plug-in module, the IMSCSA, which may be installed in Module position B.

J4 is a 20-pin right angle header intended to mate with a multi-connector ribbon cable assembly. It may be used to directly connect the TDM buses of a group of CAC V6M6 and/or VME6U6 boards. The number of boards that can be tied together through a J4 ribbon depends on the TDM clock rate. Assemblies of two V6M6 and/or VME6U6s tied together via the J4 connector will operate at up to 16 MHz, three or four boards at up to 8 MHz and five or more boards at up to 4 MHz. Note that some module designs cannot support data rates higher than 8 MHz.

3.5.1 TDM Bus Logical Structure

Consecutive TDM bit clocks are grouped together into a timeslot. All timeslots on all four TDM buses must be the same length. This length is programmably selectable via the TDM controller as 8 bits, 16 bits or 32 bits. Normally, a timeslot carries a single data item across each TDM data bus. However, the TDM system allows the use of multiple timeslots on a TDM bus to carry larger data items. Four 8-bit timeslots, for instance, can carry a 32-bit word between resources. CAC refers to the technique of using multiple small timeslots to carry a larger data item as "slot continuation".

Timeslots are organized into frames. A frame consists of a programmably selectable, or externally determined, number of consecutive timeslots. The maximum frame size on CAC VME boards is 128 slots, and so the maximum number of actual data timeslots is 128 per bus times 4 buses, or 512 slots. The connection pattern used on the TDM buses is recycled with each frame; that is, each connection established over a TDM bus recurs at the frame rate.

A frame sync signal is included in the TDM subsystem to indicate the boundary between consecutive frames. This sync pulse can be generated by the TDM controller on a V6M6 board, by a plug-in module, by external circuits connected to the J4 expansion cable, or by the SCSA interface.

The V6M6 J4 expansion connector directly links together the TDM subsystems of multiple VME boards. When boards are linked, they share the same TDM clock, timeslot definition and framing. One board must be programmed as clock master to generate the TDM clock; the remaining boards are programmed as clock slaves. The selection of which board generates frame sync is independent of the clock master selection. That is, TDM clock and TDM sync can come from the same board or different boards, depending on the application.

3.5.2 TDM Data Validity

The V6M6 TDM system includes a "Validity" signal associated with each TDM data bus. That is, TDM bus "A" has an associated "A Valid" signal, bus "B" has a "B Valid", etc. These lines indicate whether the associated TDM data bus lines carry valid data during each timeslot. This information is used to implement conditional TDM transfers on some modules.

Connections established through the TDM system provide the opportunity to transmit a data item from source to destination(s) each time its assigned timeslot occurs. If the source, for example, is assigned to drive

TDM bus A and has data ready to transmit when its timeslot comes by, it drives the TDM A Valid line high at the beginning of the time slot. This indicates to the destination(s) that bus A should be read. If the source has no data to transmit, it drives A Valid low, indicating that the destination(s) for the connection should ignore the data on bus A.

The SCSA bus does not include the concept of timeslot validity. Therefore, any TDM connections involving the SCSA bus are assumed to always have valid data.

In addition, some module types (the DM2C31, for example) are not capable of generating conditional transfers. In such modules, the data source (a TI TMS320C31 DSP in this case) does not indicate whether its serial port is ready to transmit. Thus the TDM interface of the data source must always assume that it is ready, and always drives data on its assigned timeslot(s).

3.5.3 TDM Clocking

The number of bits in a timeslot and the duration of a frame are programmable in the TDM controller. The TDM clock itself may be derived from:

- A plug-in module, such as a telecom network interface device.
- The J4 TDM expansion connector
- The SCSA bus, when the V6M6 operates as an SCSA timing slave.
- A programmable division of a frequency synthesizer on the V6M6 baseboard. This synthesizer is factory set to 16.384 MHz; a utility program allows this frequency to be set according to the needs of the application.

Typical telephony applications set the TDM clock to 8.192 MHz in order to provide the maximum number of 8-bit timeslots (128 per TDM bus, times 4 buses, yielding 512 slots). This clock is typically derived from a telecom network interface such as a T1 or E1 interface module or from another board via the J4 or SCSA interfaces.

In most telephony applications, the frame length is fixed at 125 µS, corresponding to an 8 KHz audio sampling rate. The number of timeslots per frame thus depends on the TDM clock rate chosen. For 2.048 MHz, the frame is 32 slots (128 timeslots total), for 4.096 MHz it is 64 slots (256 timeslots total), and for 8.192 MHz it is 128 slots (512 timeslots total).

The V6M6 generates a separately buffered TDM clock signal to each module and to baseboard resources concerned with the TDM system. These signals are supplied by low-skew drivers, with delay-balanced circuit board traces to each destination. Thus, the TDM clock signals at the six module sites have very little skew. Each clock must be terminated at its end-point with a series combination of a 62 ohm to 75 ohm resistor and a 100 pF to 220 pF capacitor to ground. The TDM clock trace on a module must be three inches or less in length, and the terminator must be within one inch of the end of the trace.

If the V6M6 is connected to the SCSA bus via an IMSCSA module or via the "S" option on Rev 3 and higher, a phase-locked loop associated with the SCSA interface forces the V6M6 TDMCLK signals to be properly phase aligned with the Local Bus Clock signal at the SCSA bus interface circuit (a VLSI SC4000 chip). This assures that setup and hold times associated with the SC4000 are satisfied.

The SC4000 may be programmed to become SCSA bus clock master. In this case, the TDM clock signal on the V6M6 board is derived from the frequency synthesizer on the baseboard or from a module with an external timing source, such as a T1 or an E1 interface. The SC4000 uses the output from its associated phase locked loop to produce the SCSA Bus Clock and a Local Bus Clock properly phase aligned to the TDM clock.

3.5.4 TDM Connection Control - The TDM Map RAM

Connectivity of the TDM buses on the V6M6 is directed by the content of a dual-port memory, called the TDMRAM. One port of the TDMRAM is loaded via the VME interface and, on V6M6 Rev 3 or higher, the PCI bus.

The TDM subsystem accesses the TDM RAM from the other port. It is addressed by a counter which resets to zero at the beginning of each new frame and increments with each new bit clock. An 8-bit control word is fetched from the TDMRAM each bit time. These words are not used on the V6M6 baseboard, but are broadcast to all six module sites. The modules, in turn, interpret these control words to determine their required TDM actions during the next TDM timeslot. Note that some modules (such as the IMSCSA and the DM4C51) promote their own TDM connection control, which is programmed separately.

Each timeslot has a sequence of 8 8-bit TDMRAM words (64 bits total) available to control the TDM subsystem. This 64-bit control word is staged in a holding register in each module and becomes active at the beginning of the next timeslot. That is, the control words fetched from

TDMRAM during timeslot "x" are used to control the TDM buses during timeslot "x+1" (modulo the number of timeslots in a frame).

Only 8 control words are required from the TDMRAM for each timeslot. If a slot size longer than 8 bits is selected, the TDMRAM addressing counter stops incrementing after 8 clocks in each timeslot, and resumes incrementing at the beginning of the next timeslot. Thus, exactly 8 locations in TDMRAM are used to control each timeslot regardless of the specified slot size.

The TDMRAM is divided into two banks. One bank actively controls the TDM subsystem, while the other is available for updating by the host processor through the VME interface (or through the PCI bus on Rev 3 and higher). Typically, each bank would contain a copy of the desired TDM interconnection map. When changes are made to the "Update" bank, they do not take effect until a "Bank Switch" command is issued to the TDMRAM controller. This approach is required in order to ensure that changes to the TDM interconnection map made in the TDMRAM take place in a consistent way, without the possibility of executing partially updated interconnection commands.

3.5.5 TDM Control Word

The table below shows the interpretation of the 64 TDM control bits associated with a TDM system time slot. It is divided into three main portions.

- The first portion (words 0 and 1) defines which module resources drive data on each TDM bus during the following time slot.
- The second portion (words 5, 6 and 7) define TDM destination control, such as from which TDM bus the module is to receive data
- The third portion (words 2, 3, and 4) are used to define some control action related to TDM source or destination activity.

Together these portions describe up to four connections or control actions to be made during the following time slot. Note that a connection can have only one source, but may have multiple destinations.

Word	Bits 7:4	Bits 3:0
0	Bus A Source	Bus B Source
1	Bus C Source	Bus D Source
2	Mod E Control	Mod F Control
3	Mod C Control	Mod D Control
4	Mod A Control	Mod B Control
5	Mod E Dest Control	Mod F Dest Control
6	Mod C Dest Control	Mod D Dest Control
7	Mod A Dest Control	Mod B Dest Control

Table x.a - TDM Control Words

3.5.6 TDM Source Control

Each of the six module sites interprets all four TDM bus source nibbles and two module-specific control nibbles, labeled "Control" and "Dest Control". Some module types only need the Dest Control nibble, while others use both.

The TDM bus source nibbles are shown below. Note that each module may interpret two different source codes, primary and secondary.

4-bit Code	Source Connection
0x0 - 0x5	Mod *N (Pri)* Drives TDM
0x6 - 0xB	Mod *N-6 (Sec)* Drives TDM
0xC	Src *CONTINUATION*
0xD - 0xE	Reserved
0xF	No Src Specified

Table x.b - TDM Bus Source Codes

When a module is assigned as a source on a TDM bus time slot, its TDM interface logic takes data serially from whatever source is appropriate for the module and drives it onto the designated TDM bus. Definitions of the TDM source functions for specific module types are provided in reference manual sections describing those modules.

As an example, the DM2C31 module contains two TI TMS320C31 DSP chips, numbered 0 and 1. The TDM primary source code refers to DSP 0 and the secondary source code refers to DSP 1. When a DSP is directed to be the source on a TDM time slot, interface logic on the DM2C31 generates an *FSX* pulse to that DSP's serial port, together with a burst of clock pulses to the *CLKX* pin. This causes the DSP to send serial output data onto its *DX* pin, which the TDM interface logic then drives onto the designated TDM bus.

The "Continuation" code causes the previously selected TDM source to continue transmitting data onto the designated TDM bus without generating a new "Start" command in the associated resource. This allows for data words larger than the basic TDM slot size to be transferred. Most module types do not use the Continuation code. A

DM2C31 module, for instance, has separate logic in its TDM interface to allow multiple, possibly noncontiguous, timeslots to be aggregated into larger word sizes, and does not need the Continuation code. The PM3081 and PM4600 modules do implement Continuation codes, as described in their respective reference manual sections.

3.5.7 TDM Destination Control

The encoding of TDM destinations is highly dependent on the specific module design. Generally, a V6M6 module has two 4-bit nibbles available to it to specify TDM destinations and, for some module types, additional TDM-related controls. These are labeled as *Mod n Dest Control* and *Mod n Control* in Table x.a above. Available TDM destination codes and control actions are defined in the reference manual sections describing each module.

As an example, the PM3081 and PM4600 modules implement the following *Mod n Dest Control* codes. They do not use the *Mod n Control* nibble:

Dest Ctl Code	Dest Connection	New Item
0 - 3	Receives TDM *N*	Yes
4 - 7	*CONTINUATION* Recv TDM *N-4*	No
8 - 11	*CONDITIONAL* Recv TDM *N-8*	if TDM VALID
0xF	No Dest Connection	No

Table x.c - PM3081/PM4600 Destination Codes

3.5.8 TDM Bus Timing

Overall timing of the TDM subsystem is controlled by a clock (TDMCLK) and two synchronizing signals (TDMSLOT and TDMFRAME). TDMSLOT and TDMFRAME are synchronous with TDMCLK, are sampled on the rising edge of TDMCLK, are active high, and last for exactly one TDMCLK period. TDMSLOT indicates the beginning of a new timeslot. TDMFRAME indicates the beginning of a new frame of timeslots. An additional synchronizing signal (TDMSF) is available to the TDM resources to indicate "superframe" alignment.

TDMFRAME and TDMSF are directly available on the J4 expansion connector. TDMSLOT is locally generated by the TDM controller on each V6M6 board. TDMCLK is a distinct signal on each V6M6 board. For V6M6 boards that are programmed to participate on the J4 expansion ribbon, TDMCLK is a buffered version of the J4 signal EXPANSION_CLK. The delay from EXPANSION_CLK to TDMCLK is a maximum of 5 nsec

The TDM subsystem relies heavily on pipelining. This means that the TDMSLOT and TDMFRAME signals do not line up directly with the actual data bits carried on the TDM data buses, but rather appear somewhat earlier in time. Specifically, TDMSLOT appears three clocks before the first bit of a new timeslot on the TDM data buses. TDMFRAME appears one full timeslot plus 5 clocks before the first bit of a new frame on the TDM data bus.

The effect of the pipelining for external devices which generate TDMFRAME on the J4 expansion cable is that TDMFRAME must be asserted 13 clocks before the first bit of frame data if the slot length is 8 bits, 21 clocks for 16-bit slots and 37 clocks for 32-bit slots.

On an IMSCSA module or on V6M6 "S" option boards, logic adapts the SC4000 chip's SCSA and Local Bus timing to proper TDM bus timing. The SCSA bus has no signals corresponding to TDMSLOT or TDMSF, and its SYNC signal is aligned differently from the V6M6's.

3.5.9 Superframes and TDMSF

Superframes are commonly encountered in T1 applications, where two standards are used. The first, and older, is "D4" or simply "Superframe" or SF. In SF, a sequence of 12 T1 frames, lasting 125µS each, is considered a superframe. Superframe alignment identifies "robbed bit" signaling frames (every 6th and 12th frame); the actual alignment is determined by the specific pattern of the framing bits in consecutive T1 frames. A newer standard, called Extended Superframe or ESF, groups 24 consecutive T1 frames per superframe and allows every other framing bit to carry a maintenance channel called the Facility Data Link (FDL).

E1 signals may also have superframes of two types, called CAS and CRC4. These are both 16-frame superframes, and either or both may be active at the same time, not necessarily aligned with each other. For more discussion of E1 frames and superframes, see the IM2E1 Mini-PCI Module section.

A signal called TDMSF in the TDM subsystem identifies the superframe repetition rate of the TDM data. TDMSF is a pulse which lasts for one full frame. It may be selected as an interrupt source to all processor and DSP modules on the V6M6. On DSP modules, it is also connected to a directly testable input pin on each DSP.

TDMSF cannot, in general, identify which TDM frame is the boundary of a superframe, because multiple T1 or E1 signals may be present on the TDM buses. The T1 or E1 signals are forced into alignment with TDMFRAME by the network interface module or by the SCSA bus connection, but forced superframe alignment on received signals is not possible without unacceptable delays in those input signals.

The actual superframe alignment of a specific T1 or E1 signal is not normally important to V6M6 modules. Indeed, it is generally not possible even to identify the alignment of T1/E1 signals received over the SCSA bus.

TDMSF may be generated by the TDM controller on a V6M6, by a module board or by external logic through the J4 interface. If a TDM controller is programmed to source TDMSF, it generates it by a division of the TDM frame rate. The divisor is specified as an integer between 2 and 31, so superframe lengths of 2 to 31 frames can be programmably specified. When sourced by the TDM controller, the bit-level timing of TDMSF's transitions are positioned in the middle of the first time slot of the TDM frame.

TDMSF is an especially important signal for DSP applications that send or receive more than one time slot of data per frame in any given DSP. In those situations, the DSP program needs somehow to align itself with the TDM framing so that it knows which time slot is associated with each inbound or outbound data item. TDMSF provides a mechanism to achieve this alignment. Its transition from low to high can be used to establish a reference point in the inbound and/or outbound TDM data streams within each DSP. The DSP programs are responsible for maintaining alignment between TDMSF transitions. In general, the alignment need only be determined once, prior to beginning data transfers.

Processor modules, such as the PM3081 and PM4600, do not need to use this mechanism. In these modules, each item in the TDM data stream is explicitly tagged with the timeslot number corresponding to that item. See the PM3081 and PM4600 Mini-PCI Module Reference Manuals for more detail.

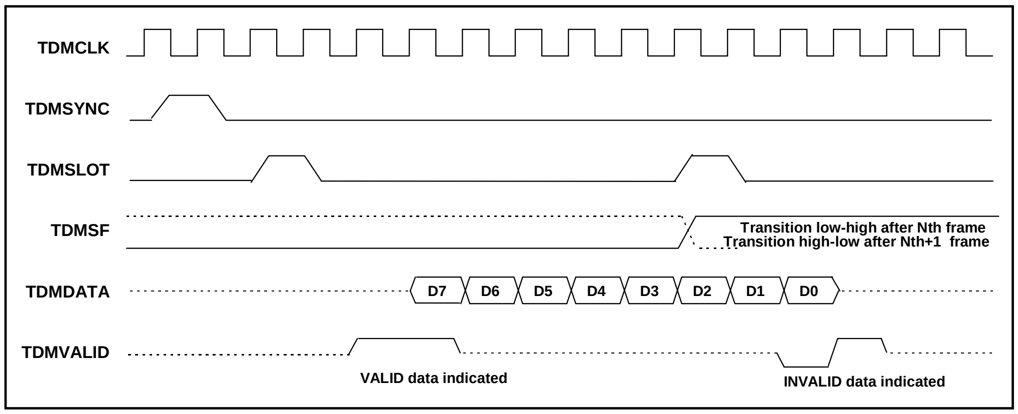


Figure 3.4 - TDM Subsystem Timing

3.5.10 TDM Subsystem Controller

(documentation not yet available)

3.6 VME/PCI Bridge

(documentation not yet available)

3.7 Software Support

The software for the V6M6 consists of the following categories:

- Unix Device Driver
- Host Application Library
- Host Support Programs and Diagnostics
- MIPS Kernel and Application Library
- MIPS Kernel Host Server
- pSOS Support

The software currently supports use on SunOS 4.1.x (sometimes referred to as Solaris 1.x) and Solaris 2.x (sometimes referred to as SunOS 5.x).

Except for the device drivers, the software is platform independent at the source code level. The device drivers and, in some instances, special "stub" modules in the host library handle platform and operating system differences.

3.7.1 Unix Device Driver

Versions of the device driver currently exist for SunOS 4.1.x and Solaris 2.x. The device driver provides access to the resources for host applications, configures the VME interface and handles interrupts from the board (passing software signals to the appropriate host processes).

Access to the board's resources is provided via entries in the /dev directory named `pci`*`NX`*, where N is an integer board number specifying a particular board and X is a lower case letter specifying the resources on the board as follows:

pci0a-f	refer to individual module resources.
pci0g,h	provide additional host access to global memory.
pci0q	is the VME Interface resource, which is used for board initialization, and special control operations.
pci0t	is the TDM subsystem resource used to control the TDM serial I/O configuration and operation. This resource is also used to access the microcontroller.

The primary data transfer method is through memory-mapped I/O. Some operations are handled by the device driver using the **ioctl** interface. Although MIPS processor modules can access other boards on the VMEbus, the V6M6 does not support DMA transfer with the host system.

3.7.2 V6M6 Host Application Library

The V6M6 Host Application C Library provides most of the functionality required to access the module resources and perform I/O such as loading and running programs, uploading and downloading data, and controlling the TDM interface.

The following sections discuss general purpose and baseboard-specific functions. Functions to control operations for specific modules are discussed in chapters 4 through 8 which describe the Mini-PCI Modules.

3.7.2.1 Host Device Access and Data Transfer Functions

(documentation not yet available)

3.7.2.2 TDM Subsystem Control Functions

(documentation not yet available)

3.7.3 Host Support Programs

The V6M6 software package includes several utilities run on the host These perform important operations such as board initialization and updating flash memory. These utilities include:

- pciinit - V6M6 initialization
- pciinfo - Obtains information about V6M6 boards
- pciflashup - Updates contents of flash memory.
- pcidmexec - Loads and executes programs on DSP modules
- pcitdmcfg- Configures TDM subsystem and connection map

To access the utilities and their online manuals, see Section 2.3.2 Programmer and User Setup p 2-16.

4. DSP Modules

4.1 DM2C31

The DM2C31 provides two Texas Instruments TMS320C31 DSPs for the V6M6 and PCI module carrier boards. The module includes either 512 Kbytes or 2 Mbytes of local SRAM for each of the two DSPs, a DMA interface to the PCI bus, an interface to the TDM subsystem, control / status / interrupt interface and a frequency synthesizer to generate DSP clocks of up to 60 MHz.

Information in this chapter is divided into the following sections:

4.1.1 DM2C31 Quick Tour

Figure 4.1-1 is a block diagram of the DM2C31 showing the major components and data paths.

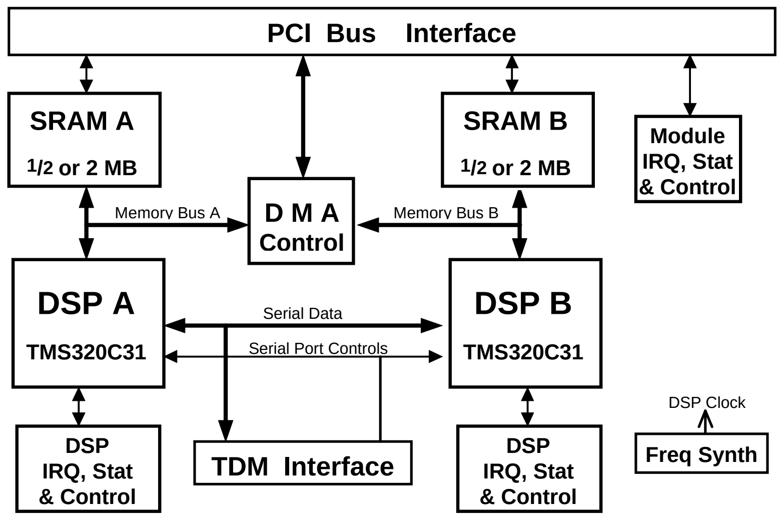


Fig. 4.1-1: DM2C31 Module Block Diagram

- **Local and Global Memory Access**

  Each of the DSPs has either 1/2 or 2 MB of local SRAM. The two memory banks are not interconnected. Each memory bank may be accessed by its DSP, as a PCI target, or by the DMA controller. The DMA controller is shared by the two DSPs but may not transfer data between the two memory banks.

  Global memory on the V6M6 baseboard is accessed from the DSPs indirectly by copying to or from local memory. Other resources, including other VME boards, are also accessed via the DMA controller.

- **Serial Data Access / TDM Interface**

  The serial port of each DSP interfaces to the TDM subsystem of the V6M6 or baseboard. Loading and unloading of a DSP's serial buffers is determined by the control words loaded into the TDM subsystem's connection map and the TDM clock. The size of the serial transfers is programmed in a module control register, accessed by the host.

  TDM frame or multiframe events may be used as interrupt and or timer sources for the DSPs.

- **DSP and Host Interrupts**

  Each DSP may select any of several possible interrupt sources, including TDM events, DMA completion, mailbox status, XF0 from the other DSP, Global PCI interrupt, and an interrupt from the host system. Any of these sources may be selected to generate INT0, INT1 or XF1, independently.

  The DSPs may generate PCI interrupts (INT-A from DSP 0 and INT-B from DSP 1) via their control registers. These in turn may be used to interrupt the host, as controlled from the host through the VME/PCI interface on the baseboard. PCI INT-A may be used to generate a Global PCI interrupt on version 2 V6M6 boards. Either PCI INT may generate a global interrupt on version 3 or higher V6M6 boards.

- **DSP Clock Generation**

  The clock for the DSPs is generated by a frequency synthesizer. The clock rate is a multiple of the PCI clock rate, programmed by the manufacturer. The maximum rate is 2.4 times the PCI frequency. The PCI clock rate is fixed at 20 MHz on version 1 V6M6 boards and is programmable (by the manufacturer) up to 25 MHz on version 2 V6M6 boards. The PCI clock and DSP clock multiplier generally depends on the other module types to be installed on the baseboard.

4.1.2 DM2C31 Detailed Hardware Description

The DM2C31 module uses FPGA configuration data stored in the Flash ROM on the V6M6 and V9M12. The flash object name is **dm2c31**. The FPGA is configured (along with those on other modules) by the micro-processor on the baseboard, under control of the device driver or the **pciinit** program.

4.1.2.1 Module Status and Control Registers

Registers for the module's PCI interface status and control and some basic DSP status and control are accessed in PCI configuration space. The table below shows the configuration space base addresses for the different module locations. The configuration space base addresses are set by physical hardware connections. The address is also stored in the PCI_MOD structure used by the V6M6 Host Application Library.

PCI Configuration Space Base Addresses		
PCI Module	V6M6 Confide Space	V9M12 Config Space
A	0x01000000	TBD
B	0x00800000	TBD
C	0x00400000	TBD
D	0x00200000	TBD
E	0x00100000	TBD
F	0x00080000	TBD

The following table lists the DM2C31 configuration space registers. The addresses shown are the offsets from the module's configuration space.

DM2C31 Module Status and Control Registers			
Register Name	Address	Acc	Function / Description
PCI CSR	0x04	rw	Standard PCI Control/Status
Mod Base	0x10	wo	PCI base addr of DM2C31 memory
DSP Status & Control	0x40	rw	DSP FP0 stat and RESET control
Module Interrupts	0x44	rw	PCI->DSP and DSP->PCI interrupts
Serial Port Control	0x48	rw	Serial port transfer size (also contains Module ID)

4.1.2.1.1 PCI Stat / Cmd Register (0x04)

This register is used for configuring and obtaining status of the module's interface to the PCI bus. The control bits are all either hardwired or programmed by the **pciinit** program. The bits set by **pciinit** are:

```
Bit 1 = 1: Memory Space Enabled
Bit 2 = 1: Bus Master Enabled
Bit 6 = 1: PCI Parity Error Response Enabled
```

The status bits, listed below, are available for PCI error checking. These are cleared by writing a 1 into them.

```
Bit 28 - Module received a Target Abort
Bit 29 - Module Master Abort
Bit 30 - Module signaled a System Error
Bit 31 - Module detected a PCI Parity Error
```

4.1.2.1.2 Mod Base Register (0x10)

This register controls the PCI address space in which the module will respond as a target. It is programmed by the **pciinit** program to respond in memory space. **pciinit** also determines the base address according to the other modules installed on the baseboard. A copy of its contents is stored in the host’s device driver and made available to host applications via the V6M6 Host Application Library.

4.1.2.1.3 DSP Status and Control Register (0x40)

This register allows other PCI modules and the host to examine the XF0 outputs of the DSPs and to reset the DSPs. The DSP resets may be controlled individually using PCI byte enables.

```
Byte 3:
    Bit   31    - DSP 1 XFO (read only)
    Bits 30:24  - (reserved)
Byte 2:
    Bit   23    - DSP 0 XFO (read only)
    Bits 22:16  - (reserved)
Byte 1:
    Bits 15:9   - (reserved)
    Bit   8     - DSP 1 Reset (low active)
Byte 0:
    Bits  7:1   - (reserved)
    Bit   0     - DSP 0 Reset (low active)
```

4.1.2.1.4 Module Interrupts Register (0x44)

This register provides control and status of the interrupts between the PCI bus and the DSPs on the DM2C31 module. The four interrupts may be accessed and modified individually using PCI byte enables.

```
Byte 3:
    Bit   31    - PCI Interrupt to DSP 1
    Bits 30:24  - (reserved)
Byte 2:
    Bit   23    - PCI Interrupt to DSP 0
    Bits 22:16  - (reserved)
Byte 1:
    Bits 15:9   - (reserved)
    Bit   8     - DSP 1 interrupt to PCI INT B
Byte 0:
    Bits  7:1   - (reserved)
    Bit   0     - DSP 0 interrupt to PCI INT A
```

The interrupts to the DSPs (bits 32 and 31) are set when written with a one. They are cleared by the DSP writing to the HOSTINT bit in its ISTAT register (see section 4.1.2.2.2 ISTAT Register (0x80A001)). When read from the PCI bus, these bits indicate the current state of the interrupts.

The interrupts from the DSPs to the PCI bus (bits 0 and 8) are set by the DSPs in their DSPCSR registers (see section 4.1.2.2.1 DSP Control / Status Register (0x80A000)). They are cleared when read via the Module Interrupts Register.

4.1.2.1.5 Serial Port Control Register (0x48)

The upper byte of this register controls the size of data transfers for the DSP serial ports.

```
Bits   31:30   - DSP 0 serial output size
Bits   29:28   - DSP 0 serial input size
Bits   27:26   - DSP 1 serial output size
Bits   25:24   - DSP 1 serial input size
```

The size is specified in terms of the number of TDM time slots. The sizes for each DSP's serial input and output transfers may be set to different values. The register uses two bits for each transfer size with the following values:

```
0   =   1 TDM time slot  per transfer
1   =   2 TDM time slots per transfer
2   =   3 TDM time slots per transfer
3   =   4 TDM time slots per transfer
```

The low byte of the Serial Byte Control Register contains the module ID (a value from 0 to 11) in bits 0:3 indicating the module's location on the baseboard. Bit 4 indicates the amount of local SRAM for each DSP; 0 for w Mbytes, 1 for 512K bytes. These values are loaded by the **pciinit** program.

4.1.2.2 DSP Status and Control Registers

The following table lists the status and control registers accessed by the DSPs on the DM2C31. Each DSP has its own set of these registers. These are all 8-bit registers. The addresses shown are absolute addresses on the DSP's memory bus.

DM2C31 DSP Status and Control Registers			
Register Name	Address	Acc	Function / Description
DSPCSR	0x80A000	rw	DSP Control and Status
ISTAT	0x80A001	rc	DSP Interrupt Status
IMASK0	0x80A002	rw	Enable INT0 low on ISTAT
IMASK1	0x80A003	rw	Enable INT1 low on ISTAT
XF1ENB	0x80A004	rw	Enable XF1 low on ISTAT
DMASTAT	0x80A005	rc	DMA Controller Status
MAILBOX	0x80A006	rw	MBOXIN and MBOXOUT registers
DMAPARM	0x80A008	w	DMA controller parameters

Note: `rc` means register is readable and cleared by writing.

4.1.2.2.1 DSP Control / Status Register (0x80A000)

This register contains miscellaneous control and status functions.

```
Bit   7  - DSPID  0 = DSP 0,  1 = DSP 1  (hardwired)
Bits 6:4 - MODID (read-only copy of value in PCI cfg reg)
Bit   2  - TDMISEL: 0 = frame, 1 = superframe
Bit   1  - LED:     0 = on, 1 = off
Bit   0  - PCIINT (write 1 to set)
```

The TDMISEL bit selects which TDM event is used as a possible DSP interrupt source. This may be either TDM frame events or TDM superframe events.

The LED bit controls illumination of the module's panel LED. DSP 0 controls the green element and DSP 1 controls the red element. The LED will be red while the module's FPGA is unconfigured.

The PCIINT bit generates the PCI interrupt (INT A for DSP 0 and INT B for DSP 1).

4.1.2.2.2 ISTAT Register (0x80A001)

This register indicates the status of all possible interrupt sources. Each condition is cleared by writing a one to the corresponding bit.

```
Bit  7 - DMA Done
Bit  6 - (reserved)
Bit  5 - XF0 from other DSP
Bit  4 - TDM Event
Bit  3 - Interrupt from PCI (Module Interrupts Register)
Bit  2 - PCI Global Interrupt occurred
Bit  1 - MBOXIN Full
Bit  0 - MBOXOUT Empty
```

4.1.2.2.3 Interrupt Mask 0 Register (0x80A002)

This register controls which of the condition sources in ISTAT are enabled to cause an interrupt on the DSP’s INT0. A one in any bit enables the corresponding condition to cause an interrupt.

4.1.2.2.4 Interrupt Mask 1 Register (0x80A003)

This register controls which of the condition sources in ISTAT are enabled to cause an interrupt on the DSP’s INT1. A one in any bit enables the corresponding condition to cause an interrupt.

4.1.2.2.5 XF1 Mask Register (0x80A004)

This register controls which of the condition sources in ISTAT are enabled to appear on the DSPs XF1 input.. A one in any bit enables the corresponding condition to make XF1 active.

4.1.2.2.6 DMA Status Register (0x80A005)

Status from the DMA controller is available in this register. This register should be read when servicing a DMA interrupt to determine whether errors occurred. All the bits are cleared by any write to the register. Status should be cleared before initiating a DMA transfer.

```
Bit  7 - DMA Success
Bit  6 - DMA Error:  PCI master disabled
Bit  5 - DMA Error:  PCI master access timeout
Bit  4 - DMA Error:  PCI master abort
Bit  3 - DMA Error:  PCI target abort received
Bit  2 - DMA Error:  PCI parity error detected
Bit  1 - DMA Error:  PCI parity error reported
Bit  0 - (reserved)
```

4.1.2.2.7 MailBox Register (0x80A006)

This is a bi-directional mailbox register: The MAILBOX IN and MAILBOX OUT registers are at the same address. Writing to this address sets the MBOXIN Full flag in ISTAT. Reading from this address sets the MBOXOUT Empty flag in ISTAT.

4.1.2.2.8 DMA Parameter Register (0x80A008)

This register is actually an auto-sequenced bank of registers used to store parameters for and initiate a DMA transfer. The DMA parameters must be written in the sequence show below. The DMA transfer is initiated when the last parameter is written.

```
PCI Addr 0  - PCI Address bits  9:2
PCI Addr 1  - PCI Address bits 17:10
PCI Addr 2  - PCI Address bits 25:18
PCI Addr 3  - PCI Address bits 31:26
DSP Addr 0  - DSP Address bits  7:0
DSP Addr 1  - DSP Address bits 15:8
DSP Addr 2  - DSP Address bits 18:16
Word Cnt 0  - Word count LST (bits  7:0)
Word Cnt 1  - Word count MST (bits 11:8)
PCI Control - Bits 3:0 contain PCI Command
              Bit 5 set requests PCI Lock
```

4.1.2.3 Local Memory and DMA Interface

The DM2C31 module is available with 2 MB or 512 KB of SRAM for each DSP. Each DSP's local memory is accessible from the DSP itself, from other PCI modules or VME boards via the PCI bus, and from the DMA controller on the DM2C31.

Memory chips with access times of 17ns are required to run the DSP clock at 60 MHz. Chips with 17ns access times for the 2 MB option were not available at the time this manual was written. Contact CAC for current memory option availability.

The local SRAM is addressed starting at 0 on the DSP's memory bus. The local SRAM is mapped into PCI memory space by the **pciinit** program and the base address is stored in the MOD BASE register in the module's configuration space. The base address and maximum address are also maintained by the host device driver and made available to host application programs.

The DSPs cannot access each other's local memory directly and the DMA controller cannot be used to transfer data between the two memory banks. Data to be shared between the two DSPs on a module must be located in global memory on the V6M6 or global memory module.

The DSPs access global memory and any other PCI module or VME device through the DMA controller on the module. The DMA controller is shared between the two DSPs on the module. Each DSP has its own DMA parameter and status registers allowing independent transfer requests to be queued. The DMA control processes the requests one at a time in the order received.

The parameters include the PCI and local memory addresses, the number of 32-bit words to be transferred, and a PCI control word. The PCI control words consists of the standard 4-bit PCI bus command encoding and a bit to select use of PCI resource locking.

4.1.2.4 DSP and PCI Interrupts

The INT0, INT1 and XF1 inputs of the DSPs are driven from a maskable set of conditions including:

Mailbox In Full and Mailbox Out Empty
XF0 from the other DSP on the Module
DMA Request Completed
PCI Global Interrupt
PCI Local Interrupt
TDM Frame or TDM Superframe

Each DSP has its own independent set of latched conditions. The status of these signals is always available, unmasked, in the DSPs ISTAT register. The condition bits are cleared by writing a one to the appropriate bit in the ISTAT register. Each DSP also has its set of 3 mask registers, IMASK0, IMASK1 and XF1ENB, for controlling which conditions are to generate the INT0, INT1 and XF1 signals.

The PCI local interrupts are generated from the PCI bus (by another PCI module or another VME board) using bits 23 and 31 of the Module Interrupts Register. The TDM event condition, Frame or Superframe is selected in the TDMISEL bit of the DSPCSR register.

4.1.2.4.1 Interrupts Generated from the DSPs

The DSPs can activate the module’s PCI INT signals by writing to bit 0 of their DSPCSR registers. DSP 0 generates PCI INT A and DSP 1 generates PCI INT B. These signals are cleared from the PCI bus when the appropriate byte of the Module Interrupts Register is read. These interrupts may be used to interrupt the host under control of the host in the interrupt mask of the VME Interface. Version 2 V6M6 boards may use the INTA signal to cause a PCI Global Interrupt. Version 3 V6M6 boards may use INT A and INT B to cause a PCI Global Interrupt.

4.1.2.5 TDM Subsystem Interface

The serial ports of the TMS320C31 DSPs on the DM2C31 are connected to the V6M6 TDM subsystem through logic in the FPGA. The TDM interface logic on the DM2C31 generates the serial clock and FSR or FSX pulses to the DSPs.

The TDM interface logic also decodes the TDM control words generated from the TDM Map RAM on the baseboard. There are 8 control words for each time slot which determine the connections between the DSPs and the TDM subsystem busses. The TDM map is programmed using C functions provided in the V6M6 Host Application Library (see 4.1.3.1 V6M6 Host Application Library). The words stored in the TDM Map RAM are described below:

DMC32 TDM MAP Control Words

Control Word	Bits 7:4	Bits 3:0
0	TDMA-SRC	TDMB-SRC
1	TDMC-SRC	TDMD-SRC
2	MOD4-CTL	MOD5-CTL
3	MOD2-CTL	MOD3-CTL
4	MOD0-CTL	MOD1-CTL
5	MOD4-DST	MOD5-DST
6	MOD2-DST	MOD3-DST
7	MOD0-DST	MOD1-DST

<u>TDM SRC</u> The 4-bit code in each of the TDM Source selectors determines the device to source that TDM Bus.

```
0x0   DSP 0 on Module A sources the TDM Bus
0x1   DSP 0 on Module B sources the TDM Bus
0x2   DSP 0 on Module C sources the TDM Bus
0x3   DSP 0 on Module D sources the TDM Bus
0x4   DSP 0 on Module E sources the TDM Bus
0x5   DSP 0 on Module F sources the TDM Bus
0x6   DSP 1 on Module A sources the TDM Bus
0x7   DSP 1 on Module B sources the TDM Bus
0x8   DSP 1 on Module C sources the TDM Bus
0x9   DSP 1 on Module D sources the TDM Bus
0xA   DSP 1 on Module E sources the TDM Bus
0xB   DSP 1 on Module F sources the TDM Bus
0xF   No source for the TDM Bus
```

MOD CTL The 2-bit codes in these words control reading and writing of the DSP serial I/O buffers for DM2C31 on the module.

Bits 1:0	Serial input control
11	No write
10	write DSP 0
01	write DSP 1
00	write DSPs 0 and 1

Bits 3:2	Serial output control
11	No read
10	read DSP 0
01	read DSP 1
00	read DSPs 0 and 1

MOD DST The 2-bit codes in these words select which TDM Bus is used for getting data to the DSPs on the module.

Bits 1:0	Bus for Data to DSP0
00	TDM Bus A
01	TDM Bus B
10	TDM Bus C
11	TDM Bus D

Bits 3:2	Bus for Data to DSP0
00	TDM Bus A
01	TDM Bus B
10	TDM Bus C
11	TDM Bus D

The size of each TDM slot is programmed in the TDM subsystem controller. It may be set to 8, 16 or 32 bits. Some applications may require a mix of data sizes to be transferred on the TDM subsystem. This may be accomplished by setting multiple TDM time slots for each DSP serial port transfer in the module's Serial Port Control Register, described in Section 4.1.2.1 Module Status and Control Registers. This register provides independent control of the transfer size for each of the DSP's serial input and output ports. For example, when a T1 or E1 interface is used the basic TDM time slot will generally be set to 8 bits, but it may be desirable to have the DSPs transfer 32-bit words over the TDM subsystem by setting its serial port to use 4 TDM time slots per transfer.

When multiple TDM slots per transfer are used, the first TDM/DSP connection for a transfer causes a FSR or FSX pulse to be generated. The FSR or FSX pulse is suppressed for the remaining TDM/DSP connections of the transfer. It is not necessary for the TDM/DSP connections of a transfer to be made in adjacent TDM slots. However, it is required that the number of TDM/DSP time slots in the TDM Map be a multiple of the number of time slots per DSP transfer. It is also necessary for the program running in the DSP to initialize the DSP's SIO register to the appropriate number of bits per transfer.

To transmit data onto the TDM busses, the DSP application writes the data to the C31 Serial Data-Transmit Register (DXR). For each time slot for which the DSP is mapped to drive data into a TDM Bus, the FSX pulse is generated and the serial shift clock is enabled to the DSP. The data presently in the C31 Serial Transmit Shift Register (XSR) is shifted onto the appropriate TDM Busses and then the data in the DXR is loaded into the empty XSR. Because the data in the DXR is loaded into the XSR after the start of a serial transfer, data written to the DXR is delayed by one serial transfer before being transferred onto the TDM Busses. Due to the one serial pipe line delay, data to be transmitted on a specific TDM time slot must be written to the DXR one serial transfer prior to the specified time slot. **Error! Reference source not found.** illustrates an example of the TDM serial data pipe line.

The following are some implications of the pipe line delay:

- When synchronizing the transmit data stream to the TDM frame, the first word written to the DXR after the TDMINT (bit 4 of the ISTAT register) becomes active (indicating the start of TDM frame) should be the data for the second transfer of the TDM frame.

- It is not possible to transmit valid data in the very first time slot of the TDM frame after synchronizing to the TDM frame.

- The first transfer of the current frame must have been written to the DXR before the occurrence of the last transfer of the prior frame.

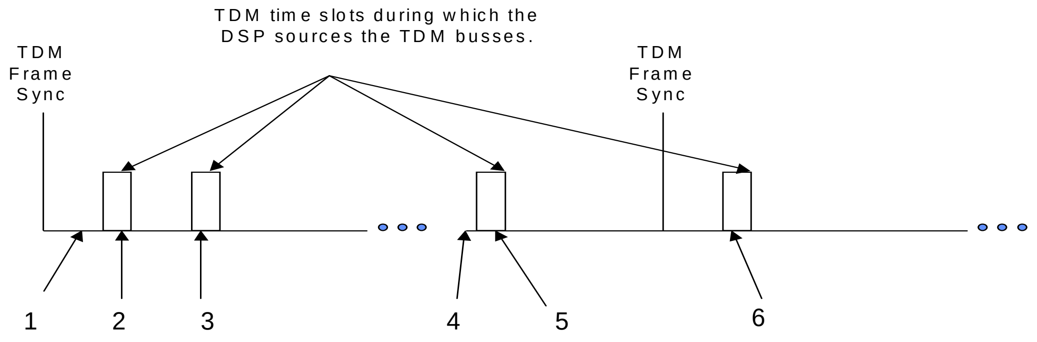


Figure 4.1.2-1 DM2C31 TDM Subsystem Transfer Timing

1 - The DSP application writes data to the DXR. For example: 0x12345678.

2 - The XSR is shifted onto the TDM Busses and 0x12345678 from the DXR is loaded into the XSR. After the DXR empty flag is set, the DSP application can write the next word into the DXR (for example: 0x55aa55aa).

3 - 0x12345678 is shifted out of the XSR onto the TDM Busses, then the XSR is loaded with 0x55aa55aa from the DXR.

4 - The TDM Data for the first time slot of the next TDM frame must be loaded into the DXR before this time.

5 - 0x55aa55aa is shifted out of the XSR onto the TDM Busses and the first transfer of the next frame is loaded into the XSR.

6 - The XSR is shifted onto the TDM Busses.

4.1.2.6 Specifications

The table below lists some of the significant performance characteristics of the DM2C31 module and its network interface components. For further details consult the reference manual for the TMS320C31 DSP.

Parameter	Max	Units
TMS320C31 Clock [1] with 2MB per DSP with 512KB per DSP	 55 60	 MHz
PCI Clock [2]	25	MHz
TDM Clock	8.192	MHz
Power Requirements with 2MB per DSP with 512KB per DSP	 1.05 TBD	 Amps @5V
MTBF with 2MB per DSP with 512KB per DSP	 TBD TBD	 1000 hrs

Notes:

1. Future DM2C31/2MB modules will be able to run at 60 MHz when SRAM chips with 17 ns access times are available.
2. The max PCI clock may be limited by other modules installed.
3. Characteristics apply under ambient temperature of 0 to +55 °C.

4.1.3 DM2C31 Software Support

4.1.3.1 V6M6 Host Application Library

Several functions are included in the V6M6 Host Application Library for access and control of the DM2C31 module and the TDM subsystem.

4.1.3.1.1 DM2C31 Functions Quick Reference

These are the C-callable functions of the V6M6 Host Application Library used for programming the DM2C31 module. Full descriptions begin in DM2C31 Module Access, Control and Interrupts on page 4.1-23 .

DM2C31 Module Access, Control and Interrupts

`PCI_MOD *` **`dm2c31_open`**`(`*`devname`*`)`	Opens the PCI module and initializes a PCI_MOD structure to reference the specified DSP on the module.
`int` **`pciclose`**`(`*`pci`*`)`	Relinquishes access to the module referenced by the PCI_MOD pointer, *pci*.
`int` **`dm2c31_init`**`(`*`pci`*`)`	Asserts the RESET signal to the DSP referenced by the *pci* argument.
`int` **`dm2c31_run`**`(`*`pci`*`)`	Initiates program execution on a DSP by de-asserting its RESET signal.
`int` **`dm2c31_halt`**`(`*`pci`*`)`	Stops program execution on a DSP by asserting its RESET signal.
`int` **`dm2c31_hostimask`**`(`*`pci, enb`*`)`	Enables VME interrupts from a DSP to the PCI bus interrupts.
`int` **`dm2c31_hoststat`**`(`*`pci`*`)`	Reads the DSP-to-PCI interrupt status for the DSP referenced by *pci.*
`int` **`dm2c31_dspiset`**`(`*`pci`*`)`	Sets the PCI-to-DSP interrupt for the DSP referenced by *pci.*

DM2C31 Module Access, Control and Interrupts (continued)

int **dm2c31_dspistat**(*pci*)	Reads the current state of the PCI-to-DSP interrupt for the DSP referenced by *pci.*
int **dm2c31_ser_cfg**(*pci*, *slot_x*, *slot_r*)	Sets the number of TDM slots per serial transfer for the DSP referenced by *pci.*
int **dm2c31_tdm_slot**(*pci*, *slot_xa, slot_ra, slot_xb, slot_rb*)	Sets the number of TDM slots per serial transfer for both DSPs on the module referenced by *pci* Similar to **dm2c31_ser_cfg**.

Data Transfer Functions

int **pci_dl_a32b**(*pci, pciadr, nwords, buf*)	Transfers an array **from the host to PCI memory**.
int **pci_up_a32b**(*pci, pciadr, nwords, buf*)	Transfers an array of 32-bit values **from PCI memory to the host**.
int **pci_dl_i32b**(*pci, pciadr, value*)	Transfers a single 32-bit value **from the host to PCI memory**.
u_long **pci_up_i32b**(*pci, pciadr*)	Transfers a single 32-bit value **from PCI memory to the host**.
int **pci_dl_float**(*pci, pciadr, value*) int **pci_dl_double**(*pci, pciadr, value*)	Both of these functions transfer a single or double-precision floating point *value* from the host to PCI memory.
double **pci_up_float**(*pci, pciadr*) double **pci_up_double**(*pci, pciadr*)	Both of these functions transfer a single or double-precision floating point value from PCI memory.
long **ieee_to_c3x**(*ieee_value*)	Converts an IEEE floating point value to TMS320C31 format.
double **c3x_to_ieee**(*c3x_value*)	Converts a TMS320C31 floating point value to IEEE format.

C31 Executable Code Functions

int **pci_load_code**(*pci, codename, codetype*)	Reads a TMS320C31 executable file and downloads the code to the DSP’s memory.
int **pci_read_code**(*pci, codename, codetype*)	Reads a TMS320C31 executable file named *codename* in format *codetype*. Used to initialize symbolic information for a progam that is already running on a DSP.
long **pci_find_label_name**(*pci, name*)	Searches through the symbol list in the PCI_MOD structure (*pci*) for a label whose name matches *name*.
char ***pci_find_label_address**(*pci, address*)	Searches through the symbol list in the PCI_MOD structure (*pci*) for a label at the address specified by the *address* argument.
void **pci_free_labels**(*pci*)	Deletes the symbol table list from the PCI_MOD structure and frees up the memory allocated to store the list.

TDM Subsystem Control Functions

int **pci_tdm_init**(*tdm, clk, bits, slots, frames*)	Initializes the parameters of the TDM subsystem controller.
int **pci_tdm_run**(*tdm*)	Starts the TDM subsystem timing logic.

TDM Connection MAP functions

int **pci_tdm_src_add**(*tdm, slot, bus, devtyp, devnum*)	Adds a device as a TDM data source for the specified time *slot* and TDM *bus*.
int **pci_tdm_src_del**(*tdm, slot, bus*)	Deletes a device as a TDM data source for the specified time *slot* and TDM *bus*.
int **pci_tdm_dst_add**(*tdm, slot, bus, devtyp, devnum*)	Adds a device as a TDM data destination for the specified time *slot* and TDM *bus*.
int **pci_tdm_dst_del**(*tdm, slot, devtyp, devnum*)	Deletes a device as a TDM data destination for the specified time *slot* and TDM *bus*.

4.1.3.1.2 DM2C31 Module Access, Control and Interrupts

The following functions are used to access and control the DM2C31 module through the VME interface and the Unix host device interface. These functions require a DM2C31 handle,a **PCI_MOD** structure pointer allocated and set up to access a DSP on a DM2C31 module. The function, **dm2c31_open**, must be used to allocate and initialize a **PCI_MOD** structure for use with the DM2C31 functions. The PCI_MOD structure is defined in <**pciutil.h**>.

function:	`PCI_MOD * ` **`dm2c31_open`** `(`*`devname`*`)`	
args:	`char *`*`devname`*	
return:	`PCI_MOD *`	`pointer to PCI_MOD structure.`
	`NULL`	`a NULL pointer if an error indicated.`
errors:	`EBADF`	*`devname`* `is invalid`
	`EINVAL`	`module referenced by` *`devname`* `is not a DM2C31.`
	`EBUSY`	`the module referenced by` *`devname`* `is in use by another program.`
	`ENODEV`	`device does not exist.`
	`ENXIO`	`device does not exist.`
	`ENOMEM`	`no memory for PCI_MOD structure.`

The **dm2c31_open** function opens the PCI module and initializes a PCI_MOD structure to reference the specified DSP on the module. The module and DSP are specified in the *devname* argument, which is a string in the format: **pci*NRD***

In this string the characters, **pci**, are required and may be preceded by the characters, **/dev**. The variable characters specify the baseboard, module and DSP as follows:

N	is the baseboard number in the system (an integer)
R	is the PCI module resource (a letter from a to p)
D	is the DSP device number (0 or 1)

This function calls **pciopen**(the generic module access function) to open the PCI module containing the specified DSP device. It then customizes information in that PCI_MOD structure to reference the specified DSP device.

It is possible for a program to open both DSPs on a DM2C31 module. This is accomplished by declaring two PCI_MOD structures and initializing them with separate calls to **dm2c31_open**. For example:

```
PCI_MOD *dm1, *dm2;

dm1 = dm2c31_open (“pci0a0”);
dm2 = dm2c31_open (“pci0a1”);
```

However, because access to a module cannot be shared between two different programs, it is not possible for two programs to each access different DSPs on the same module.

function:	`int` **`pciclose`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`PCI module device closed.`
	`0`	`error indicated.`
errors:	`EBADF`	*`pci`* `is a NULL pointer.`
	`EINTR`	`an interrupts was received before the file descriptor was closed.`

The **pciclose** function is used to relinquish access to the module referenced by the PCI_MOD pointer, *pci*. This function also frees the memory allocated for the PCI_MOD structure and unmaps the VME I/O space for the module.

function:	`int` **`dm2c31_init`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`DM2C31.DSP initialized.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM2C31.`
	`EIO`	`error in communicating with module.`

The **dm2c31_init** function asserts the RESET signal to the DSP referenced by the *pci* argument. Asserting RESET clears the DSP and associated interrupt and status registers. This function also loads the module ID in the module's Serial Port Control Register.

function:	`int` **`dm2c31_run`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`DM2C31.DSP execution started.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM2C31.`
	`EIO`	`error in communicating with module.`

The **dm2c31_run** function initiates program execution on a DSP by de-asserting its RESET signal. The DSP to be run is specified by the information stored in the PCI_MOD structure that is the *pci* argument. The **pci_load_code** function is used to download programs to the DSP's memory (see Section 4.1.3.1.3 Data Transfer Functions).

function:	`int` **`dm2c31_halt`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`DM2C31.DSP execution halted.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM2C31.`
	`EIO`	`error in communicating with module.`

The **dm2c31_halt** function stops program execution on a DSP by asserting its RESET signal. The DSP to be halted is specified by the information stored in the PCI_MOD structure that is the *pci* argument. Asserting the RESET signal also clears the DSP's interrupt and status registers.

function:	`int` **`dm2c31_hostimask`** `(`*`pci`*`, enb)`

args:	`PCI_MOD *`*`pci`*
	`int enb`

return:	`1 host interrupt mask written.`
	`0 error indicated.`

errors:	`EINVAL` *`pci`* `does not reference a DM2C31.`

The **dm2c31_hostimask** function is used to enable VME interrupts from a DSP to the host system. It uses the V6M6 device driver’s ioctl request to control the interrupt mask bit in the VME-PCI bridge for the DSP referenced by *pci*. The VME interrupt is enabled if *enb* is 1 and disabled if *enb* is 0.

function:	`int` **`dm2c31_hostistat`** `(`*`pci`*`)`
args:	`PCI_MOD *`*`pci`*

return:	` 1 DSP->PCI interrupt is active`
	` 0 DSP->PCI interrupt is not active`
	`-1 error indicated.`

errors:	`EINVAL` *`pci`* `does not reference a DM2C31 module.`
	`EIO error in communicating with module.`

The **dm2c31_hostistat** function reads the DSP-to-PCI interrupt status for the DSP referenced by *pci*. Note: calling this function reads the appropriate bit of the Module Interrupts Register (see 4.1.2.1.4 Module Interrupts Register (0x44) and clears the selected status.

function:	`int` **`dm2c31_dspiset`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`PCI->DSP interrupt asserted`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM2C31 module.`
	`EIO`	`error in communicating with module.`

The **dm2c31_dspiset** function sets the PCI-to-DSP interrupt for the DSP referenced by *pci*. It sets the appropriate bit in the Module Interrupts Register and will appear in the DSP's interrupt status register (see 4.1.2.1.4 Module Interrupts Register (0x44)).

function:	`int` **`dm2c31_dspistat`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`PCI->DSP interrupt is active`
	`0`	`PCI->DSP interrupt is not active`
	`-1`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM2C31 module.`
	`EIO`	`error in communicating with module.`

The **dm2c31_dspistat** function reads the current state of the PCI-to-DSP interrupt for the DSP referenced by *pci*. It reads the appropriate bit in the Module Interrupts Register (see 4.1.2.1.4 Module Interrupts Register (0x44)). There are no side effects.

function:	`int` **`dm2c31_ser_cfg`** `(`*`pci`*`, slot_x, slot_r)`	
args:	`PCI_MOD`	`*`*`pci`*
	`int`	`slot_x, slot_r`
return:	`1`	`Serial Port Control register written`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM2C31 module or invalid` *`slot_x`* `or` *`slot_r`* `values.`
	`EIO`	`error in communicating with module.`

The **dm2c31_ser_cfg** function sets the number of TDM slots per serial transfer for the DSP referenced by *pci*. The *slot_x* argument selects the number of TDM slots per transfer transmitted from the DSPs serial output. The *slot_r* argument selects the number of TDM slots per transfers received by the DSPs serial input. The function writes the values to the appropriate DSP's Serial Port Control Register (see 4.1.2.1.5 Serial Port Control Register (0x48)).

Valid values for slot_x and slot_r are:

```
0  =  1 TDM time slot per transfer
1  =  2 TDM time slots per transfer
2  =  3 TDM time slots per transfer
3  =  4 TDM time slots per transfer
```

These settings, in conjunction with the setting for the number of bits per TDM time slot via the **pci_tdm_init** function (section 4.1.3.1.5 TDM Subsystem Control Functions), determine the number of bits per DSP serial port transfer. The DSP's program must set its own Serial I/O configuration accordingly.

function:	`int` **`dm2c31_tdm_slot`** `(`*`pci`*`, slot_xa, slot_ra, slot_xb, slot_rb)`	
args:	`PCI_MOD`	`*`*`pci`*
	`int`	`slot_xa, slot_ra, slot_xb, slot_rb`
return:	`1`	`Serial Port Control register written`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM2C31 module or invalid` *`slot`* `arguments.`
	`EIO`	`error in communicating with module.`

The **dm2c31_tdm_slot** function is similar to the the **dm2c31_ser_cfg** function described above. This function sets the number of TDM slots per serial transfer for both DSPs on the module referenced by *pci*. The *slot_xa* and *slot_ra* arguments select the transfer size for DSP0’s serial transmit and receive buffers. The *slot_xb* and *slot_rb* arguments select the transfer size for DSP1’s serial transmit and receive buffers.

4.1.3.1.3 Data Transfer Functions

Data transfer between the host and the DSP’s memory is performed using memory mapped I/O. Transfers are limited to those involving 32 bit values. The V6M6 Host Application Library provides the following functions for data transfers.

function:	int **pci_dl_a32b** (*pci*, *pciadr*, *nwords*, *buf*) int **pci_up_a32b** (*pci*, *pciadr*, *nwords*, *buf*)	
args:	PCI_MOD u_long	**pci* *pciadr*, *nwords*, *buf*
return:	1 0	transfer was successful. error indicated.
errors:	EINVAL EIO	*pci* is NULL, *pciadr* out of bounds, *nwords* is 0, or *buf* is NULL error in communicating with module.

The **pci_dl_a32b** function transfers an array from the memory of the host host program to PCI memory on the module or baseboard referenced by *pci*. The **pci_up_a32b** function transfers an array from PCI memory to the host program’s memory.

The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP referenced by *pci* or global memory on the baseboard. The *nwords* argument specifies how many 32-bit words to transfer. The *buf* argument specifies the host program’s data buffer.

function:	`int` **`pci_dl_i32b`** `(`*`pci, pciadr, value`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`u_long`	*`pciadr, value`*
return:	`1`	`transfer was successful.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or` *`pciadr`* `out of bounds`
	`EIO`	`error in communicating with module.`

The **pci_dl_i32b** function transfers a single 32-bit value from the host to PCI memory on the module or baseboard referenced by *pci*. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* must be modulo-4 aligned and must reside within the DSP's local memory space or the baseboard's global memory space.

function:	`u_long` **`pci_up_i32b`** (*`pci, pciadr`*)

args:	`PCI_MOD` *`*pci`*
	 `u_long` *`pciadr`*

return:	*`value`* `data uploaded from PCI memory.`
	 `0` `error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or` *`pciadr`* `out of bounds`
	 `EIO` `error in communicating with module.`

The **pci_up_i32b** function transfers a single 32-bit value from PCI memory on the module or baseboard referenced by *pci* to the host. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* must be modulo-4 aligned and must reside within the DSP's local memory space or the baseboard's global memory space.

The function returns the 32-bit value uploaded from PCI memory. Note that an error condition is indistinguishable from a successful value of 0. To avoid this ambiguity it is recommended that the global variable **errno** be cleared prior to, and tested after the function call, for example:

```
u_long  pciadr, data;
extern int errno;

errno = 0;
data = pci_up_i32b (pci, pciadr);
if  (errno != 0)
    perror(“pci_up_i32b”);
```

function:	int **pci_dl_float** (*pci, pciadr, value*) int **pci_dl_double** (*pci, pciadr, value*)
args:	PCI_MOD **pci* u_long *pciadr* double *value*
return:	1 transfer was successful. 0 error indicated.
errors:	EINVAL *pci* is NULL or *pciadr* out of bounds EIO error in communicating with module.

The **pci_dl_float** and **pci_dl_double** functions transfer a single-precision or double precision floating point *value* from the host to PCI memory on the module or baseboard referenced by *pci*. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard.

Note that *pciadr* must reside within the DSP's local memory space or the baseboard's global memory space. It must also be modulo-4 aligned for **pci_dl_float** and modulo-8 aligned for **pci_dl_double**. The *value* argument for both functions is of type, double.

When used with a DM2C31 module, these functions convert the floating point value from IEEE format to the TMS320C31's floating point format and transfer the converted value. There is no functional difference between **pci_dl_double** and **pci_dl_float**.

function:	double **pci_up_float** (*pci, pciadr*)	
	double **pci_up_double** (*pci, pciadr*)	
args:	PCI_MOD	**pci*
	u_long	*pciadr*
return:	*value*	data uploaded from PCI memory.
	0.0	error indicated.
errors:	EINVAL	*pci* is NULL or *pciadr* out of bounds
	EIO	error in communicating with module.

The **pci_up_float** and **pci_up_double** functions transfer a single-precision or double-precision floating point value from PCI memory on the module or baseboard referenced by *pci* to the host. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard.

Note that *pciadr* must reside within the DSP's local memory space or the baseboard's global memory space. It must also be modulo-4 aligned for **pci_up_float** and modulo-8 aligned for **pci_up_double**.

Note that an error condition is indistinguishable from a successful value of 0.0. To avoid this ambiguity it is recommended that the global variable **errno** be cleared prior to, and tested after the function call, for example:

```
u_long  pciadr;
double data;
extern int errno;

errno = 0;
data = pci_up_float (pci, pciadr);
if  (errno != 0)
    perror("pci_up_i32b");
```

Transferring arrays of floating point values between a DM2C31 and the host involves using the **pci_dl_a32b** and **pci_up_a32b** functions to transfer the raw 32-bit data. TMS320C31 function libraries are available which include functions to convert to and from single-precision IEEE format on the DSP. The following two functions in the CAC V6M6 library may be used to convert the data on the host.

function:	`long` **`ieee_to_c3x`** `(`*`ieee_value`*`)`
args:	`double` *`ieee_value`*
return:	*`c3x_value`* `TMS320C3X floating-point value.`
errors:	`none`

function:	`double` **`c3x_to_ieee`** `(`*`c3x_value`*`)`
args:	`long` *`c3xe_value`*
return:	*`ieee_value`* `IEEE floating-point value.`
errors:	`none`

4.1.3.1.4 C31 Executable Code Functions

The following functions provide for downloading TMS320C31 executable code in extended COFF format to the DSPs on a DM2C31 module and finding the PCI addresses of global variables in the program.

function:	`int` **`pci_load_code`** `(`*`pci, codename, codetype`*`)`

args:	`PCI_MOD  *`*`pci`*
	`char     *`*`codename`*
	`int       `*`codetype`*

return:	`1  code file read and downloaded successfully.`
	`0  error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or` *`codetype`* `is unknown.`
	`ENOEXEC  the file contains invalid COFF data.`
	`EIO  error in communicating with module.`
	`others  errors from the open() system call.`

The **pci_load_code** function reads a TMS320C31 executable file and downloads the code to the DSP's memory. The file is specified in the *codename* argument. The *codetype* argument specifies the file format. For the TMS320C31, the format type should be the macro, **MM_COFF**, defined in <**pciutil.h**>.

In addition to downloading the code, this function builds a list of the symbols and their addresses found in the COFF file. The list is stored in the PCI_MOD structure for later access using the **pci_find_label_name** and **pci_find_label_address** functions.

function:	`int` **`pci_read_code`** `(`*`pci, codename, codetype`*`)`

args:	`PCI_MOD  *`*`pci`*
	`char     *`*`codename`*
	`int       `*`codetype`*

return:	`1         code file read successfully.`
	`0         error indicated.`

errors:	`EINVAL    `*`pci`*` is NULL or `*`codetype`*` is unknown.`
	`ENOEXEC   the file contains invalid COFF data.`
	`EIO       error in communicating with module.`
	`others    errors from the open() system call.`

The **pci_read_code** function reads a TMS320C31 executable file named *codename* in format *codetype*. The format type should be **MM_COFF**, a macro defined in <**pciutil.h**>.

pci_read_code does not download the code to DSP memory. It only reads the COFF file and builds a list of the symbols and addresses for the **pci_find_label_name** and **pci_find_label_address** functions. This is useful for host applications required to access a DSP that is already running the program.

function:	`long` **`pci_find_label_name`** `(`*`pci, name`*`)`

args:	`PCI_MOD  *`*`pci`*
	`char     *`*`name`*

return:	*`address`*`   memory address of named label.`
	`-1        label not found or error indicated.`

errors:	`EINVAL    `*`pci`*` is NULL`

The **pci_find_label_name** function searches through the symbol list in the PCI_MOD structure (*pci*) for a label whose name matches *name*. If found the PCI memory address of the label is returned. The address returned may be used as the *pciadr* argument to the data transfer functions.

function:	`char * `**`pci_find_label_address`**` (`*`pci, address`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`u_long`	*`address`*
return:	*`name`*	`pointer to label name string.`
	`NULL`	`label not found or error indicated.`
errors:	`EINVAL`	*`pci`*` is NULL`

The **pci_find_label_address** function searches through the symbol list in the PCI_MOD structure (*pci*) for a label at the address specified by the *address* argument. If found, a pointer to the label's name is returned.

For this function, values for address are memory addresses with respect to the DSP address rather than the PCI memory space.

function:	`void  `**`pci_free_labels`**` (`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`none`	
errors:	`EINVAL`	*`pci`*` is NULL`

The **pci_free_labels** function deletes the symbol table list from the PCI_MOD structure and frees up the memory allocated to store the list. This operation is required in situations where a completely new DSP program is to be loaded using the same PCI_MOD structure that has been used to download a previous program. Freeing the previous symbol table will ensure that the PCI_MOD structure contains only the symbols for the new program.

4.1.3.1.5 TDM Subsystem Control Functions

The following library functions are used to control the operational parameters of the TDM subsystem.

function:	`int` **`pci_tdm_init`** `(`*`tdm, clk, bits, slots, frames`*`)`

args:	`PCI_TDM *`*`tdm`*
	`int` *`clk, bits, slots, frames`*

return:	`1    TDM controller initialized.`
	`0    error indicated.`

errors:	`EINVAL` *`tdm`* `is NULL or other arguments are invalid.`

The **pci_tdm_init** function initializes the parameters of the TDM subsystem controller.

For the DM2C31, use the following values for the arguments:

clk	When any module or the TDM expansion port is configured to generate the TDM subsystem clock, this argument must be 0. To generate TDM clocks from the TDM subsystem controller, a value of 2, 4, or 8 should be used.
bits	Bits per time slot, 8, 16 or 32.
slots	This value determines the number of TDM time slots per TDM frame and may range from 2 to 128. If another board connected to the TDM expansion port is generating TDM frame and superframe pulses, this value should be 0.
frames	This value determines the number of TDM frames per TDM superframe and may range from 0 to 31. If set to 0, the TDM controller will not generate superframe pulses.

clk	When any module or the TDM expansion port is configured to generate the TDM subsystem clock, this argument must be 0. To generate TDM clocks from the TDM subsystem controller, a value of 2, 4, or 8 should be used.
bits	Bits per time slot, 8, 16 or 32.
slots	This value determines the number of TDM time slots per TDM frame and may range from 2 to 128. If another board connected to the TDM expansion port is generating TDM frame and superframe pulses, this value should be 0.
frames	This value determines the number of TDM frames per TDM superframe and may range from 0 to 31. If set to 0, the TDM controller will not generate superframe pulses.

function: `int` **`pci_tdm_run`** `(`*`tdm`*`)`

args: `PCI_TDM` `*`*`tdm`*

return:	`1`	`TDM subsystem started.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`tdm`* `is NULL or other arguments are invalid.`
	`EIO`	`failed to update the TDM MAP RAM, usually due to the absence of TDM clocks.`

The **pci_tdm_run** function starts the TDM subsystem timing logic. The TDM map is updated, if necessary, and the TDM controller will begin generating the sync pulses as specified by the **pci_tdm_init** function.

4.1.3.1.6 TDM Connection MAP functions

The following library functions provide the means for setting up TDM data transfer connections between DM2C31 modules and other modules.

function:	int **pci_tdm_src_add** (*tdm, slot, bus, devtyp, devnum*) int **pci_tdm_src_del**(*tdm, slot, bus*)
args:	PCI_TDM **tdm* int *slot* int *bus* int *devtyp* int *devnum*
return:	1 Buffered TDM Map modified. 0 error indicated.
errors:	EINVAL *tdm* is NULL or does not reference the TDM subsystem of a V6M6 or V9M12, or other arguments are invalid. ENODEV the specified *devtyp* (an DM2C31) is not installed at the module location specified in *devnum*.

These two functions add or delete a device as a TDM data source for the specified time *slot* and TDM *bus*. Only one device on a board may be the source for a TDM bus during a time slot. Therefore, it is not necessary to specify the device type or unit number for the delete function.

The following arguments values are used for the DM2C31 module:

slot	Values between 1 and the number of slots configured for the TDM subsystem frame (see **pci_tdm_init**).
bus	One of the macros defined in <**pciutil.h**>, **TDM_BUSA**, **TDM_BUSB**, **TDM_BUSC** and **TDM_BUSD**, corresponding to the four TDM busses.
devtyp	The device type for the DM2C31 the device type is **TDM_DEVDM2C31**. This is a macro defined in <**pciutil.h**>.
devnum	The device unit number encodes the module location and which DSP on the module as shown in the table below.

DM2C31 TDM device unit number

Bit 4 = DSP Select	Bits 3:0 = Module
0 = DSP 0 1 = DSP 1	0 = A 1 = B 2 = C 3 = D 4 = E 5 = F

function:	int **pci_tdm_dst_add** (*tdm, slot, bus, devtyp, devnum)*		
	int **pci_tdm_dst_del** (*tdm, slot, devtyp, devnum*)		
args:	PCI_TDM	**tdm*	
	int	*slot*	
	int	*bus*	
	int	*devtyp*	
	int	*devnum*	
return:	1	Buffered TDM Map modified.	
	0	error indicated.	
errors:	EINVAL	*tdm* is NULL or does not reference the TDM subsystem of a V6M6 or V9M12, or other arguments are invalid.	
	ENODEV	the specified *devtyp* (an DM2C31) is not installed at the module location specified in *devnum*.	

These two functions add or delete a device as a TDM data destination for the specified time *slot* and TDM *bus*. A device may be a destination for only one TDM bus during a time slot so it is not necessary to specify the TDM bus for the delete function. The use of the arguments is the same as the **pci_tdm_src_add** function.

4.1.3.2 C31 DSP Programming

Writing and executing programs in the Texas Instrument TMS320C31 dsp requires considerations of the hardware environment. The CAC PCI software distribution includes several files to aid in the generation and execution of DSP applications. These files are located in the directory **$CAC/pci/modsupport/dm2c31**. Files other than the ones described here are the source code for the DSP portions of the DM2C31 diagnostic programs. They are intended to be used as programming examples.

4.1.3.2.1 Linker Command File

Linker command files control the building of a DSP executable program. The linker file, for the DM2C31 is named **dm2c31.cmd**. It performs the following tasks:

- allocates 256 words for stack space in the end of internal memory bank 1 starting at 0x809f00.
- reserves 64 words starting at zero for the interrupt vector table
- allocates 32k words for heap space
- puts the start up code right after the interrupt vector table at address 0x40
- locates the text and data sections in external SRAM
- includes the CAC C startup up code, c31int00.obj
- includes the standard C library

4.1.3.2.2 DM2C31 Start-Up Code

The start-up code for the DM2C31 is in file **c31int00.asm**. The code performs the following tasks:

- initializes the stack pointer
- reserves space for the C31 interrupt vector table
- configures the Primary-Bus Control Register (see below)
- processes the runtime initialization table and initializes the global variables
- loads the data page register(DP) to the start of the .bss section
- calls the function **main**

The start-up code initializes the Primary Bus Control Register as follows:

Field Name	Value	Function
SWW	2	RDY_{int} is the logical-OR of RDY and RDY_{wtcnt}
WTCNT	3	internal wait states for software wait-state mode
BNKCMP	1	Bank Size is 2^{23} - 32-bit words.

All other fields of the Primary Bus Control Register are set to zero.

4.1.3.2.3 DM2C31 Header File

The file, **dm2c31.h**, contains macros for the addresses and special bits of many of the DSP internal and memory mapped registers and registers on the DM2C31 module. The file also includes some functional macros for operations such as controlling the LED and determining the DSP's module and ID.

4.1.3.2.4 Serial Port Global Configuration Register

To properly transfer serial data to and from the TDM subsystem, the Serial-Port Global Control register fields XVAREN and RVAREN must be set. In addition, the transmit and receive word sizes must be set to match the serial transfer size selected on the DM2C31. See the description of the Serial Port Control Register(see Section 4.1.2.1.5 Serial Port Control Register (0x48)) and the **dm2c31_ser_cfg** host library function (Section 4.1.3.1.2 DM2C31 Module Access, Control and Interrupts). Below are some examples of values for the serial port configuration.

Serial Port Configuration	Value
8-bit transmit and receive word	0x0e000300
16-bit transmit and receive word	0x0e940300
32-bit transmit and receive word	0x0ebc0300

4.1.3.3 Utility Programs

The following programs, in **$CAC/bin**, are provided for the V6M6 PCI carrier boards. They are useful for initializing and maintaining the DM2C31 module. Detailed descriptions of these programs are in **Chapter 3. V6M6 Baseboard.**

pciinit	Instructs the on-board microprocessor to configure FPGAs on the module. It then initializes the baseboard and modules and sets up PCI address mappings.
pciinfo	Examines the EEROMs on the baseboard and modules and the flash memory. It then displays all pertinent version and serial number information.
pciflashup	Compares flash object (FPGA configurations and microprocessor program) versions in the flash memory with those residing in host files. Depending on the command line options it will either display the version numbers or update the flash memory with any new versions on disk.

The following two programs are used to set the clock frequencies of the PCI bus, TDM bus and the TMS320C31 DSPs. They are not installed as part of the normal software installation because the clock frequencies are set to specifications prior to delivery of the boards. However, the source code and a Makefile are distributed in **$CAC/pci/pciutils** and they may be compiled and run in case the clocks need to be modified in the field.

pcifreq Adjusts the clock synthesizers for the PCI bus and TDM bus clock frequencies on version 2 and above V6M6 boards. The usage of the program is:

```
pcifreq pci_board [pci_freq [tdm_freq]]
```

Where pci_board is the name of the V6M6 board (e.g. pci0), pci_freq is desired clock frequency for the PCI bus in Mhz (default is 20) and tdm_freq is the desired maximum TDM clock frequency in MHz (default is 16.384). Note that in order to specify the TDM clock, the PCI clock must be specified first.

The program adjusts the settings, asks the user to reset the board using the reset switch on the front panel, and then runs the pciinit program to reinitialize the board with the new settings.

dmfreq

Adjusts the clock for the DSPs on DM2C31 modules by setting the frequency multiplier value. It uses the current setting of the PCI clock frequency to determine the proper multiplier value to achieve the desired DSP clock frequency. Usage of the program is:

```
dmfreq -c31 pci_board [dm_freq]
```
or
```
dmfreq -c31 dm_module [dm_freq]
```

The first form is used to adjust the DSP clock for all DM modules found on the specified V6M6 board, pci_board (e.g. pci0). The second form adjusts the clock for the DSPs on the specific dm_module named as a V6M6 board and module letter (e.g. pci0a). In both cases dm_freq specifies the DSP clock frequency in MHz. The default frequency is 2.4 times the current PCI bus clock frequency, which is the maximum multiplier value.

The program adjusts the setting, asks the user to reset the board using the reset switch on the front panel, and then runs the pciinit program to reinitialize the board with the new settings.

Note that the **-c31** option instructs **dmfreq** to adjust only DM2C31 modules. Without this option, the program will also operate on DM4C51 modules.

4.1.3.4 Diagnostic programs

The following diagnostic programs are provided for the DM2C31 module in the directory, **$CAC/pci/diag**.

pcipiomem	Tests communication and data transfer between the host and memory on the baseboard and modules.
pcimemory	Runs a memory diagnostic from the DSPs.
pcimemslice	Runs a memory diagnostic with the host and all processors found on the baseboard such that each processor is testing a slice of global memory and local memory on all the other modules.
pcichip	Test various functions of the DSP processors and interfaces.
pciburn	Runs the set of diagnostics on all V6M6 boards installed in the system or a specified list of boards. It logs the tests that are run and any errors reported by the diagnostics.

4.2 DM4C51

The DM4C51 provides four Texas Instruments TMS320C51 DSPs for the V6M6 and PCI module carrier boards. The module includes 256 Kbytes of local SRAM for each of the four DSPs, a DMA interface to the PCI bus, an interface to the TDM subsystem, control / status / interrupt interface and a frequency synthesizer to generate DSP clocks of up to 60 MHz.

Information in this chapter is divided into the following sections:

4.2.1 DM4C51 Quick Tour

Figure 4.2-1 is a block diagram of the DM4C51 showing the major components and data paths.

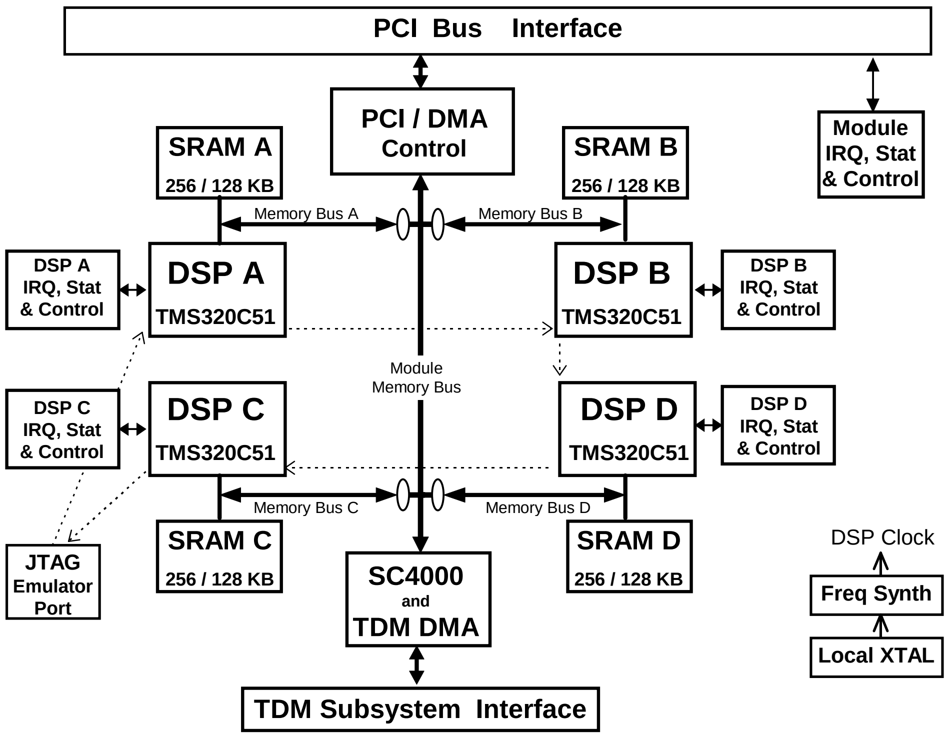


Fig. 4.2-1: DM4C51 Module Block Diagram

- **Module Sub-Types**

  The DM4C51 is available in four optional types:

  PCI_DM4C51
  This is the standard DM4C51 module with 128K words of external memory partitioned as 64K for program and 64K for data.

  PCI_DM4C51_JTAG
  A DMC451 module with a JTAG header installed for program debugging.

  PCI_DM4C51_64K
  A DM4C51 module with 64K words of external unified memory (program and data reside in a single memory segment).

  PCI_DM4C41_64K_JTAG
  A DMC451 module with 64K words of unified memory and JTAG header installed for program debugging.

  Note that for the 64K module types, memory address bit 16 is pulled up so when linking your code, both program and data must be placed in SRAM1 bank. This does not effect the C51 memory address since it can only address 64K external, but is used by the host library to determine which bank of memory to load and resolve address labels properly.

  Module types DM4C51_JTAG and DM4C51_JTAG_64K include a special debug port to connect with a Texas Instruments JTAG emulator.

- **Local and Global Memory Access**

  Each of the TMS320C51 DSPs has 256 Kbytes of local SRAM. The memory is arranged in 16-bit words. For 65K module types only half of this memory is accessible due to the unified memory mapping.

  The four memory banks are not interconnected. Each memory bank may be accessed by its DSP, as a PCI target, or by the DMA controller. The DMA controller is shared by the four DSPs but may not transfer data between the four memory banks.

  Global memory on the V6M6 baseboard is accessed from the DSPs indirectly by copying to or from local memory via the DMA controller. Other resources, including other VME boards, are also accessed via the DMA controller.

- **TDM Interface**

  The TDM interface of DM4C51 module consists of a SC4000 device, a SC4000 to local data space DMA controller (referred as the TDM DMA controller) and four local memory data buffers. The 128 local time slots of the SC4000 are partitioned between the four DSPs.

  Each DSP can access up to 32 8-bit TDM time slots for receiving data and, Independently, 32 8-bit TDM time slots for transmitting data. There is a delay of one TDM frame between the TDM subsystem and the local data memory buffers. TDM frame or multiframe events may be used as DSP interrupt sources.

- **DSP and Host Interrupts**

  Each DSP may select any of several possible interrupt sources, including TDM events, DMA completion, Global PCI Interrupt, Local PCI Interrupt and memory accesses.

  The DSPs may generate PCI interrupts (INT-A and INT-B) via their control registers. These in turn may be used to interrupt the host, as controlled from the host through the VME/PCI interface on the baseboard. PCI INT-A may be used to generate a Global PCI interrupt on version 2 V6M6 boards. Either PCI INT may generate a global interrupt on version 3 or higher V6M6 boards.

- **DSP Clock Generation**

  The clock for the DSPs is generated by a frequency synthesizer whose time-base is a crystal oscillator on the module. The DSP clock is programmable up to 55MHz. However, the access time of the SRAM on the module limits the frequency to 41.5 MHz.

  The clock may be programmed up to the 50 MHz rating of the DSPs if the DSP startup code is modified to use 1 wait state for external RAM access.

4.2.2 DM4C51 Detailed Hardware Description

The DM4C51 module uses FPGA configuration data stored in the Flash ROM on the V6M6. The flash object name is **dm4c51**. The FPGA is configured (along with those on other modules) by the micro-processor on the baseboard, under control of the device driver or the **pciinit** program.

4.2.2.1 Module Status and Control Registers

Registers for the DM4C51 PCI interface status and control and some basic DSP status and control are accessed in PCI configuration space. The table below shows the configuration space base addresses for the different module locations. The configuration space base addresses are set by physical hardware connections. The address is also stored in the PCI_MOD structure used by the V6M6 Host Application Library.

PCI Configuration Space Base Addresses

PCI Module	V6M6 Configuration Space
A	0x01000000
B	0x00800000
C	0x00400000
D	0x00200000
E	0x00100000
F	0x00080000

The following table lists the DM4C51 configuration space registers. The addresses shown are the offsets from the module's configuration space.

DM4C51 Module Status and Control Registers			
Register Name	Address	Acc	Function / Description
PCI CSR	0x04	rw	Standard PCI Control/Status
Mod Base	0x10	wo	PCI base addr of DM4C51 memory
DSP Control	0x40	rw	DSP LED Enable, SC4000 Control and DSP RESET Control
Module Interrupt Status	0x44	rw	PCI->DSP and DSP->PCI interrupts status
TDM DMA Control and PCI Interrupt Mask	0x48	rw	SC4000 Reset, TDM DMA Enable, Count and PCI Interrupt Mask

PCI Stat / Cmd Register (0x04):

This register is used for configuration and obtaining status of the module's interface to the PCI bus. The control bits are all either hardwired or programmed by the **pciinit** program. The bits set by **pciinit** are:

```
Bit 1  =  1: Memory Space Enabled
Bit 2  =  1: Bus Master Enabled.
Bit 6  =  1: PCI Parity Error Response Enabled
```

The status bits, listed below, are available for PCI error checking. These are cleared by writing a 1 into them.

```
Bit 28 - Module received a Target Abort
Bit 29 - Module Master Abort
Bit 30 - Module signaled a System Error
Bit 31 - Module detected a PCI Parity Error
```

Mod Base Register (0x10):

This register controls the PCI address space in which the module will respond as a target. It is programmed by the **pciinit** program to respond in memory space. **pciinit** also determines the base address according to the other modules installed on the baseboard. A copy of its contents is stored in the host's device driver and made available to host applications via the V6M6 Host Application Library.

DSP Control Register (0x40):

This register allows other PCI modules and the host to control which DSPs drive the LEDs, enable TDM dma transfers and to reset the DSPs. The DSPs may be controlled individually using PCI byte enables.

```
Byte 3:
    Bit   31    - DSP 3 - LED Control Enable
    Bits 30:27  - (reserved)
    Bit   26    - DSP 3 - TDM to DSP data transfer
                  Enable
    Bit   25    - DSP 3 - DSP to TDM data transfer
                  Enable
    Bit   24    - DSP 3 - Reset (low active)
Byte 2:
    Bit   23    - DSP 2 - LED Control Enable
    Bits 22:19  - (reserved)
    Bit   18    - DSP 2 - TDM to DSP data transfer
                  Enable
    Bit   17    - DSP 2 - DSP to TDM data transfer
                  Enable
    Bit   16    - DSP 2 - Reset (low active)
Byte 1:
    Bit   15    - DSP 1 - LED Control Enable
    Bits 14:11  - (reserved)
    Bit   10    - DSP 1 - TDM to DSP data transfer
                  Enable
    Bit   9     - DSP 1 - DSP to TDM data transfer
                  Enable
    Bit   8     - DSP 1 - Reset (low active)
Byte 0:
    Bit   7     - DSP 0 - LED Control Enable
    Bits  6:3   - (reserved)
    Bit   2     - DSP 0 - TDM to DSP data transfer
                  Enable
    Bit   1     - DSP 0 - DSP to TDM data transfer
                  Enable
    Bit   0     - DSP 0 - Reset (low active)
```

Module Interrupts Register (0x44):

This register provides control and status of the interrupts between the PCI bus and the DSPs on the DM4C51 module. The four interrupts may be accessed and modified individually using PCI byte enables.

```
Byte 3:
    Bit   31    - PCI Interrupt to DSP 3
    Bits 30:26  - (reserved)
    Bit   25    - DSP 3 interrupt to PCI INT B
    Bit   24    - DSP 3 interrupt to PCI INT A
Byte 2:
    Bit   23    - PCI Interrupt to DSP 2
    Bits 22:18  - (reserved)
    Bit   17    - DSP 2 interrupt to PCI INT B
    Bit   16    - DSP 2 interrupt to PCI INT A
Byte 1:
    Bit   15    - PCI Interrupt to DSP 1
    Bits 14:10  - (reserved)
    Bit   9     - DSP 1 interrupt to PCI INT B
    Bit   8     - DSP 1 interrupt to PCI INT A
Byte 0:
    Bit   7     - PCI Interrupt to DSP 0
    Bits 6:2    - (reserved)
    Bit   1     - DSP 0 interrupt to PCI INT B
    Bit   0     - DSP 0 interrupt to PCI INT A
```

The interrupts to the DSPs (bits 7, 15, 23 and 31) are set when written with a one. They are cleared by the DSP writing to the DSPIRQ bit in its ICLR register (see section 4.2.2.2 DSP Status and Control Registers). When read from the PCI bus, these bits indicate the current state of the interrupts.

The interrupts from the DSPs to the PCI bus (bits 0, 1, 8, 9. 16, 17, 24 and 25) are set by the DSPs in their PCIIRQ registers (see section 4.2.2.2 DSP Status and Control Registers). They are cleared when read via the Module Interrupts Register.

TDM DMA Control and PCI Interrupt Mask Register (0x48):

```
Bit   31    - TDM SC4000 and fifo reset(active
              low)
Bit   30    - TDM DMA Controller Start
Bit   29    - (reserved)
Bits 28:24  - TDM DMA Count (number of TDM
              transfers - 2)
Bit   23    - DSP 3 INTB Mask
Bit   22    - DSP 2 INTB Mask
Bit   21    - DSP 1 INTB Mask
Bit   20    - DSP 0 INTB Mask
Bit   19    - DSP 3 INTA Mask
Bit   18    - DSP 2 INTA Mask
Bit   17    - DSP 1 INTA Mask
Bit   16    - DSP 0 INTA Mask
Bits 15:0   - (reserved)
```

The host controls the TDM Subsystem Interface through Bits 24 - 31. The SC4000 device is held in reset until Bit 31 is set high. The TDM DMA controller is enabled by setting bit 30 high. The DMA transfer count is controlled using bits 24-28. The number written to the register is 2 less than the maximum number of TDM time slots to be transferred in a TDM frame. Bits 16-23 mask the DSPs to PCI INT A and PCI INT B from generating a host interrupt.

4.2.2.2 DSP Status and Control Registers

The following table lists the status and control registers accessed by the DSPs on the DM4C51. Each DSP has its own set of these registers. These are all 4-bit registers. The registers are in the I/O address space of the DSPs.

DM4C51 DSP Status and Control Registers

Register Name	Address	Acc	Function / Description
DSPCSR	0x00	rw	DSP Control
ICLR	0x01	w	DSP Interrupt Clear
DSPSTAT	0x02	r	DSP Status
DMASTAT	0x03	r	Global DMA Status
IE	0x04	rw	Interrupt Enable
PCIIRQ	0x07	w	PCI Interrupts A and B

DSP Control Register (0x00):

This register contains miscellaneous control functions.

```
Bit  3  - TDMIRQ_SEL: 0 = frame, 1 = superframe
Bit  2  - RED LED:    0 = on, 1 = off
Bit  1  - GREEN LED:  0 = on, 1 = off
Bit  0  - TDMBUFPTR: TDM Buffer Pointer 0 = Buffer 2 valid
                                        1 = Buffer 1 valid
```

The TDMIRQ_SEL bit selects which TDM event is used as a possible DSP interrupt source. This may be either TDM frame events or TDM superframe events.

The LED bits control illumination of the module's LED on the front panel. The four DSPs control the green and red element of the LED. The LED will be red while the module's FPGA is unconfigured.

The DSPs use the TDMBUFPTR bit to determine the TDM DMA buffers to read from and write to. When the bit is set, the DSPs should read from TDM_IN_1 and write to TDM_OUT_1. The TDMBUFPTR bit is updated on a frame boundary.

Interrupt Clear Register (0x01):

This register clears the status of the Memory Write Interrupt, TDM Interrupt, host to dsp Interrupt and the Global Interrupt. Each condition is cleared by writing a one to the corresponding bit.

```
Bit  3 - Memory Write Interrupt
Bit  2 - TDM Event
Bit  1 - Interrupt from PCI (Module Interrupts Register)
Bit  0 - PCI Global Interrupt
```

DSP Interrupt Status Register (0x02):

This register holds the status of the interrupt sources in ICLR. The register shows the status independent of the mask register (IE) setting. A one in any bit indicates the corresponding interrupt event is active.

Global DMA Status Register (0x03):

This register contains the Global DMA Status. The status register is cleared when the C51 XF pin transitions from a zero to a one.

```
Bit  3 - Master Mode Disabled
Bit  2 - Global Time-out
Bit  1 - Target Abort
Bit  0 - Global Bus Parity Error
```

Interrupt Enable Register (0x04):

This register controls which of the condition sources in DSPSTAT are enabled to source C51 external interrupt 1. A one in any bit enables the corresponding condition to make INT1 active.

PCI Interrupt Request Register (0x07):

The register controls the PCI Interrupt A and B. Writing a one to any bit actives the corresponding PCI Interrupt Signal.

```
Bit  1 - PCI Interrupt B
Bit  0 - PCI Interrupt A
```

4.2.2.3 Local Memory and DMA Interface

The DM4C51 module has 256 Kbytes of SRAM for each DSP. Each DSP's local memory is accessible from the DSP itself, from other PCI modules or VME boards via the PCI bus, and from the DMA controller on the DM4C51.

The memory partitioning depends on the module sub-type. For DM4C51 and DM4C51_JTAG types the local SRAM is partitioned into two 64 K half-word pages, program space and data space. For DM4C51_64K and DM4C51_JTAG_64K types only 64 Kbytes of local SRAM are accessible and it is all in one, unified, partition containing both program and data.

The local SRAM is mapped into PCI memory space by the **pciinit** program and the base address is stored in the MOD BASE register in the module's configuration space. The program space starts at the base address and the data space starts at the base address plus 0x20000. The base address and maximum address are also maintained by the host device driver and made available to host application programs.

The DSPs cannot access each other's local memory directly and the DMA controller cannot be used to transfer data between the four memory banks. Data to be shared between the four DSPs on a module must be located in global memory on the V6M6 baseboard or a global memory (MM32) module.

The DSPs access global memory and any other PCI module or VME device through the DMA controller on the module. The DMA controller is shared between the four DSPs on the module. Each DSP has its own set of DMA parameter and status registers allowing independent transfer requests to be queued. The DMA controller processes the requests one at a time in the order received.

The parameters include the PCI and local memory addresses, the number of 32-bit words to be transferred, and a PCI control word. The PCI control words consists of the standard 4-bit PCI bus command encoding and a bit to select use of PCI resource locking. The local memory address must be a modulo2 address.

The control registers of the DMA in the data memory space of the DSPs are shown below:

Name	Address	Description
DMAPCILOADDR	0x1ff60	Bits 2 - 17 of PCI Address
DMAPCIUPADDR	0x1ff61	Bits 18 - 31 of PCI Address
DMADSPADDR	0x1ff62	Local Data Space Address, address must be modulo-2 aligned
DMASIZE	0x1ff63	Number of 32-bit words to transfer
DMACTRL	0x1ff64	DMA control parameters local -> global 0x07 global->local 0x06 global bus lock: add 0x20

The Global DMA is requested when the DSP clears the xf_pin. The DMA complete interrupt is the external interrupt 2. The following DSP code is a function for initializing the DMA parameters.

```
void
dma_start(laddr, gaddr, size, mode)
ulong laddr, gaddr, size, mode;
{
   intStatus = 2;
   /* clear external interrupt flags */
   asm(" lmmr IFR, #_intStatus");

   /* pci address load */
   *DMAPCILOADDR = gaddr >> 2;    /* pci addr bits [17:2] */
   *DMAPCIUPADDR = (gaddr >> 18); /* pci addr bits [31:18] */

   /* local address load */
   *DMADSPADDR = laddr;   /* local address bits */

   /* word size */
   *DMASIZE = size;       /* word count bits */

   /* mode */
   *DMACTRL = mode;

    asm(" clrc XF");      /* start DMA */
}
```

The DMA status function should set XF to clear the DMA status register.

4.2.2.4 DSP and PCI Interrupts

Interrupts to the DSPs

The INT0 input of the DSPs are driven from a maskable set of conditions including:

External Memory Access Interrupt
PCI Global Interrupt
PCI Local Interrupt
TDM Frame or TDM Superframe

Each DSP has its own independent set of latched conditions. The status of these signals is always available, unmasked, in the DSPs DSPSTAT register. The condition bits are cleared by writing a one to the appropriate bit in the Interrupt Enable register.

The PCI local interrupts are generated from the PCI bus (by another PCI module or another VME board) using bits 7, 15, 23 and 31 of the Module Interrupts Register. The TDM event condition, Frame or Superframe, is selected in the TDMIRQ_SEL bit of the DSPCSR register. The External Memory Access Interrupt is generated by writing to the DSP's memory with address bit 22 set.

Interrupts Generated from the DSPs

The DSPs can activate the module's PCI INT A signal by writing to bit 0 of their PCIIRQ registers. The DSPs can activate the module's PCI INT B signal by writing to bit 1 of their PCIIRQ registers. These signals are cleared from the PCI bus when the appropriate byte of the Module Interrupts Register is read. These interrupts may be used to interrupt the host under control of the host in the interrupt mask of the VME Interface. Version 2 V6M6 boards may use the INTA signal to cause a PCI Global Interrupt. Version 3 V6M6 boards may use INT A and INT B to cause a PCI Global Interrupt.

4.2.2.5 TDM Subsystem Interface

The four DSPs communicate with the TDM subsystem through an SC4000 SCSA interface. Each DSP is allocated 32 time slots on the local side of the SC4000. The 32 time slots may be mapped to any TDM time slot and TDM bus on the TDM subsystem side of the SC4000.

The DSPs read and write TDM data in their local memory using double buffers. Data is transferred between the TDM data buffer memory and the SC4000 by a DMA controller on the DM4C51 module which is dedicated to handling TDM data for the DSPs. The maximum size of the TDM data buffers are 32 bytes and are at fixed locations in the DSP's memory as indicated below:

TDM Data Buffers	
DSP Memory Addr	Buffer Use
0x1ff80	TDM Out Buffer 1 (to TDM)
0x1ffa0	TDM Out Buffer 2 (to TDM)
0x1ffc0	TDM In Buffer 1 (from TDM)
0x1ffe0	TDM In Buffer 2 (from TDM)

The TDMBUFPTR in each DSPCSR register indicates which TDM buffer the DSP is to access. When the flag is set the DSPs should write out going TDM data to TDM Out Buffer 1 and read incoming TDM data from TDM In Buffer 1. TDM Out Buffer 2 and TDM In Buffer 2 are used when the flag is clear. TDMBUFPTR is updated on a TDM frame boundary. The flag can be polled to check for a state change or the TDM frame interrupt can be used to determine when the data is available.

The standard TDM mapping functions, **pci_tdm_src_add**, **pci_tdm_dst_add**, etc. are not used for the DM4C51. Instead, the functions described in section 4.2.3.1.6, “DM4C51 TDM Connection MAP Functions”, are used to establish TDM connectivity for the DM4C51. Note that these functions assume the TDM subsystem is configured to use 8-bit TDM slots.

4.2.2.6 Specifications

The table below lists some of the significant performance characteristics of the DM4C51 module and its network interface components. For further details consult the reference manual for the TMS320C51 DSP.

Parameter	Typ	Max	Units
TMS320C51 Clock [1]	41.5	55	MHz
PCI Clock [2]	20	24	MHz
TDM Clock	-	8.192	MHz
Power Requirements		TBD	Amps @5V
MTBF		TBD	1000 hrs

Notes:

1. The TMS320C51 DSPs are rated for a 50 MHz clock. However, access times on the local SRAM parts limit the clock to 41.5 MHz when 0 wait-states are used.
2. The maximum PCI clock may be limited by other mini-PCI modules installed on the V6M6.
3. Characteristics apply under ambient temperature of 0 to +55 °C.

4.2.3 DM4C51 Software Support

4.2.3.1 V6M6 Host Application Library

Several functions are included in the V6M6 Host Application Library for access and control of the DM4C51 module and the TDM subsystem.

4.2.3.1.1 DM4C51 Functions Quick Reference

These are the C-callable functions of the V6M6 Host Application Library used for programming the DM4C51 module. Full descriptions begin on page 4.2-21.

DM4C51 Module Access, Control and Interrupts

`PCI_MOD *` **`dm4c51_open`**`(`*`devname`*`)`	Opens the PCI module and initializes a PCI_MOD structure to reference the specified DSP on the module.
`int` **`pciclose`**`(`*`pci`*`)`	Relinquishes access to the module referenced by the PCI_MOD pointer, *pci*.
`int` **`dm4c51_init`**`(`*`pci`*`)`	Asserts the RESET signal to the DSP referenced by the *pci* argument.
`int` **`dm4c51_run`**`(`*`pci`*`)`	Initiates program execution on a DSP by de-asserting its RESET signal.
`int` **`dm4c51_halt`**`(`*`pci`*`)`	Stops program execution on a DSP by asserting its RESET signal.
`int` **`dm4c51_hostimask`**`(`*`pci, enb`*`)`	Enables VME interrupts from a DSP to the PCI bus interrupts.
`int` **`dm4c51_hoststat`**`(`*`pci`*`)`	Reads the DSP-to-PCI interrupt status for the DSP referenced by *pci.*
`int` **`dm4c51_dspiset`**`(`*`pci`*`)`	Sets the PCI-to-DSP interrupt for the DSP referenced by *pci.*

DM4C51 Module Access, Control and Interrupts (continued)

`int` **`dm4c51_dspistat`**`(`*`pci`*`)`	Reads the current state of the PCI-to-DSP interrupt for the DSP referenced by *pci.*
`int` **`dm4c51_memirq`**`(pci, pciadr, value)`	Downloads a 16-bit value to the DSP memory and generates a Memory Access Interrupt

Data Transfer Functions

`int` **`dm4c51_dl_a32b`**`(`*`pci, pciadr, nwords, buf`*`)`	Transfers an array of 32-bit values **from the host to PCI memory**.
`int` **`dm4c51_up_a32b`**`(`*`pci, pciadr, nwords, buf`*`)`	Transfers an array of 32-bit values **from PCI memory to the host**.
`int` **`dm4c51_dl_i32b`**`(`*`pci, pciadr, value`*`)`	Transfers a single 32-bit value **from the host to PCI memory**.
`u_long` **`dm4c51_up_i32b`**`(`*`pci, pciadr`*`)`	Transfers a single 32-bit value **from PCI memory to the host**.
`int` **`pci_dl_a16b`**`(`*`pci, pciadr, nwords, buf`*`)`	Transfers an array of 16-bit values **from the host to PCI memory**.
`int` **`pci_up_a16b`**`(`*`pci, pciadr, nwords, buf`*`)`	Transfers an array of 16-bit values **from PCI memory to the host**.
`int` **`pci_dl_i16b`**`(`*`pci, pciadr, value`*`)`	Transfers a single 16-bit value **from the host to PCI memory**.
`u_long` **`pci_up_i16b`**`(`*`pci, pciadr`*`)`	Transfers a single 16-bit value **from PCI memory to the host**.

C51 Executable Code Functions

`int` **`pci_load_code`**`(`*`pci, codename, codetype`*`)`	Reads a TMS320C51 executable file and downloads the code to the DSP's memory.
`int` **`pci_read_code`**`(`*`pci, codename, codetype`*`)`	Reads a TMS320C51 executable file to initialize symbolic information for a program already running on a DSP.
`long` **`pci_find_label_name`**`(`*`pci, name`*`)`	Searches through the symbol list in the PCI_MOD structure (*pci*) for a label with a specified *name*.
`char *`**`pci_find_label_address`**`(`*`pci, address`*`)`	Searches through the symbol list in the PCI_MOD structure (*pci*) for a label at a specified *address*.
`void` **`pci_free_labels`**`(`*`pci`*`)`	Deletes the symbol table list from the PCI_MOD structure and frees up the memory allocated to store the list.

TDM Subsystem Control Functions

`int` **`pci_tdm_init`**`(`*`TDM, clk, bits, slots, frames`*`)`	Initializes the parameters of the TDM subsystem controller.
`int` **`pci_tdm_run`**`(`*`TDM`*`)`	Starts the TDM subsystem timing logic.

DM4C51 TDM Connection MAP functions

`int` **`dm4c51_tdm_src_add`**`(`*`TDM, slot, bus`*`)`	Adds a device as a TDM data source for the specified time *slot* and TDM *bus*.
`int` **`dm4c51_tdm_src_del`**`(`*`TDM, slot, bus`*`)`	Deletes a device as a TDM data source for the specified time *slot* and TDM *bus*.
`int` **`dm4c51_tdm_dst_add`**`(`*`TDM, slot, bus`*`)`	Adds a device as a TDM data destination for the specified time *slot* and TDM *bus*.
`int` **`dm4c51_tdm_dst_del`**`(`*`TDM, slot, bus`*`)`	Deletes a device as a TDM data destination for the specified time *slot* and TDM *bus*.

SC4000 Control functions

`int` **`dm4c51_sc4000_init`**`(pci, tdmframe)`	Resets the SC4000, disables TDM DMA controller and initializes the SC4000.
`int` **`dm4c51_sc4000_read`**`(pci, sc4000reg)`	Reads an SC4000 Register through the SC4000 parallel port.
`int` **`dm4c51_sc4000_write`**`(pci, sc4000reg, data)`	Writes an SC4000 Register through the SC4000 parallel port.
`int` **`dm4c51_cfg_read`**`(pci, sc4000reg)`	Reads an SC4000 Configuration Register using the SC4000 indirect registers.
`int` **`dm4c51_cfg_write`**`(pci, sc4000reg, data)`	Writes an SC4000 Configuration Register using the SC4000 indirect registers.
`int` **`dm4c51_route_read`**`(pci, LCLslot, dir)`	Reads an SC4000 Routing Register.
`int` **`dm4c51_route_write`**`(pci, LCLslot, TDMslot, TDMbus, dir, oe)`	Writes an SC4000 Routing Register.
`int` **`dm4c51_sc4000_mode`**`(pci, dma, reset, count)`	Initializes the SC4000 /TDM DMA control register
`int` **`dm4c51_read_mode`**`(pci)`	Reads the SC4000 /TDM DMA control register

4.2.3.1.2 DM4C51 Module Access, Control and Interrupts

The following functions are used to access and control the DM4C51 module through the VME interface and the Unix host device interface. These functions require a DM4C51 handle, a **PCI_MOD** structure pointer. The function, **dm4c51_open**, must be used to allocate and initialize the **PCI_MOD** structure for use with the DM4C51 functions. The PCI_MOD structure is defined in <**pciutil.h**>.

```
function:  PCI_MOD * dm4c51_open (devname)

args:      char   *devname

return:    PCI_MOD *   pointer to PCI_MOD structure.
           NULL        a NULL pointer if an error indicated.

errors:    EBADF       devname is invalid
           EINVAL      module referenced by devname is not a
                       DM4C51.
           EBUSY       the module referenced by devname is in
                       use by another program.
           ENODEV      device does not exist.
           ENXIO       device does not exist.
           ENOMEM      no memory for PCI_MOD structure.
```

```
function:  int pciclose (pci)

args:      PCI_MOD   *pci

return:    1           PCI module device closed.
           0           error indicated.

errors:    EBADF       pci is a NULL pointer.
           EINTR       an interrupts was received before the
                       file descriptor was closed.
```

The **dm4c51_open** function opens the PCI module and initializes a PCI_MOD structure to reference the specified DSP on the module. The module and DSP are specified in the *devname* argument, which is a string in the format: **pci*NRD***

In this string the characters, **pci**, are required and may be preceded by the characters, **/dev**. The variable characters specify the baseboard, module and DSP as follows:

N	is the baseboard number in the system (an integer)
R	is the PCI module resource (a letter from a to f)
D	is the DSP device number (0, 1, 2 or 3)

This function calls **pciopen** (the generic module access function) to open the PCI module containing the specified DSP device. It then customizes information in that PCI_MOD structure to reference the specified DSP device.

It is possible for a program to open all DSPs on a DM4C51 module. This is accomplished by declaring four PCI_MOD structures and initializing them with separate calls to **dm4c51_open**. For example:

```
PCI_MOD *dm1, *dm2, *dm3 *dm4;

dm1 = dm4c51_open (“pci0a0”);
dm2 = dm4c51_open (“pci0a1”);
dm3 = dm4c51_open (“pci0a2”);
dm4 = dm4c51_open (“pci0a3”);
```

However, because access to a module cannot be shared between two different programs, it is not possible for different host programs to access DSPs on the same module.

The **pciclose** function is used to relinquish access to the module referenced by the PCI_MOD pointer, *pci*. This function also frees the memory allocated for the PCI_MOD structure and unmaps the VME I/O space for the module.

function:	`int` **`dm4c51_init`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`DM4C51.DSP initialized.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM4C51.`
	`EIO`	`error in communicating with module.`

The **dm4c51_init** function asserts the RESET signal to the DSP referenced by the *pci* argument. Asserting RESET clears the DSP and associated interrupt and status registers.

This operation is also performed by the **dm4c51_halt** function. **dm4c51_init** is primarily intended for use by support and diagnostic programs written at CAC.

function:	`int` **`dm4c51_run`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`DM4C51.DSP execution started.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM4C51.`
	`EIO`	`error in communicating with module.`

function:	`int` **`dm4c51_halt`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`DM4C51.DSP execution halted.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM4C51.`
	`EIO`	`error in communicating with module.`

The **dm4c51_run** function runs a DSP program by initializing the DSP ID and Module ID data memory locations and initiating program execution on the DSP by de-asserting its RESET signal. The DSP to be run is specified by the information stored in the PCI_MOD structure that is the *pci* argument. The **pci_load_code** function is used to download programs to the DSP's memory (see Section 4.2.3.1.3).

The **dm4c51_halt** function stops program execution on a DSP by asserting its RESET signal. The DSP to be halted is specified by the information stored in the PCI_MOD structure that is the *pci* argument. Asserting the RESET signal also clears the DSP's interrupt and status registers.

function:	`int` **`dm4c51_hostimask`** `(`*`pci`*`, enb)`

args:	`PCI_MOD  *`*`pci`*
	`int       enb`

return:	`1          host interrupt mask written.`
	`0          error indicated.`

errors:	`EINVAL     `*`pci`*` does not reference a DM4C51.`

The **dm4c51_hostimask** function is used to enable VME interrupts from a DSP to the host system. It uses the V6M6 device driver's ioctl request to control the interrupt mask bit in the VME-PCI bridge for the DSP referenced by *pci*. The VME interrupt is enabled if *enb* is 1 and disabled if *enb* is 0.

function:	`int` **`dm4c51_hostistat`** `(`*`pci`*`)`
args:	`PCI_MOD  *`*`pci`*

return:	` 1         DSP->PCI interrupt is active`
	` 0         DSP->PCI interrupt is not active`
	`-1         error indicated.`

errors:	`EINVAL     `*`pci`*` does not reference a DM4C51 module.`
	`EIO        error in communicating with module.`

The **dm4c51_hostistat** function reads the DSP-to-PCI interrupt status for the DSP referenced by *pci*. Note: calling this function reads the appropriate bit of the Module Interrupts Register (see Section 4.2.2.1 Module Status and Control Registers) which clears the selected status.

function:	`int` **`dm4c51_dspiset`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`PCI->DSP interrupt asserted`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM4C51 module.`
	`EIO`	`error in communicating with module.`

The **dm4c51_dspiset** function sets the PCI-to-DSP interrupt for the DSP referenced by *pci*. It sets the appropriate bit in the Module Interrupts Register (see Section 4.2.2.1 Module Status and Control Registers) and will appear in the DSP's interrupt status register.

function:	`int` **`dm4c51_dspistat`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	`PCI->DSP interrupt is active`
	`0`	`PCI->DSP interrupt is not active`
	`-1`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM4C51 module.`
	`EIO`	`error in communicating with module.`

The **dm4c51_dspistat** function reads the current state of the PCI-to-DSP interrupt for the DSP referenced by *pci*. It reads the appropriate bit in the Module Interrupts Register (see Section 4.2.2.1 Module Status and Control Registers). There are no side effects.

function:	`int` **`dm4c51_memirq`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`u_long`	`pciadr, value`
return:	`1`	`Memory location written`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `does not reference a DM4C51 module or invalid` *`pciadr`* `address.`
	`EIO`	`error in communicating with module.`

The **dm4c51_memirq** function adds bit 22 to the specified *pciadr* to form a new address and then downloads the 16-bit *value*. Setting bit 22 causes the DSP Memory Interrupt to be generated. The 16 bit memory location pointed to by *pciadr* is updated as if the **pci_dl_i16b** function was used.

4.2.3.1.3 Data Transfer Functions

Data transfer between the host and the DSP’s memory is performed using memory mapped I/O. Transfers are limited to those involving 16 bit or 32 bit values. The V6M6 Host Application Library provides the following functions for data transfers.

To transfer single precision floating point values between the host and a DM4C51 module, use the 32-bit data transfer functions with type casts.

```
function:  int dm4c51_dl_a32b (pci, pciadr, nwords, buf)
           int dm4c51_up_a32b (pci, pciadr, nwords, buf)

args:      PCI_MOD  *pci
           u_long    pciadr, nwords, buf

return:    1          transfer was successful.
           0          error indicated.

errors:    EINVAL     pci is NULL, pciadr out of bounds,
                      nwords is 0, or buf is NULL
           EIO        error in communicating with module.
```

The **dm4c51_dl_a32b** function transfers an array from the memory of the host program to PCI memory on the module or baseboard referenced by *pci*. The **dm4c51_up_a32b** function transfers an array from PCI memory to the host program’s memory.

The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP referenced by *pci* or global memory on the baseboard. The *nwords* argument specifies how many 32-bit words to transfer. The *buf* argument specifies the host program’s data buffer.

Note: These special 32-bit array transfer functions must be used to properly transfer data or from the DM4C51 module for two reasons:

1. The format of a 32-bit value created by the Texas Instrument C compiler for the C51.

2. A 32-bit value does not have to start on a PCI address long-word byte boundary for the DM4C51.

function:	`int` **`dm4c51_dl_i32b`** `(`*`pci, pciadr, value`*`)`

args:	`PCI_MOD  *`*`pci`*
	`u_long   `*`pciadr, value`*

return:	`1        transfer was successful.`
	`0        error indicated.`

errors:	`EINVAL   `*`pci`*` is NULL or `*`pciadr`*` out of bounds`
	`EIO      error in communicating with module.`

The **dm4c51_dl_i32b** function transfers a single 32-bit value from the host to PCI memory on the module or baseboard referenced by *pci*. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* must be modulo-2 aligned and must reside within the DSP's local memory space or the baseboard's global memory space.

Note: This special 32-bit transfer function must be used to properly download the data to the DM4C51 module for two reasons:

1. The format of a 32-bit value created by the Texas Instrument C compiler for the C51.

2. A 32-bit value does not have to start on a PCI address long-word byte boundary for the DM4C51.

function:	u_long **dm4c51_up_i32b** (*pci, pciadr*)	
args:	PCI_MOD **pci*	
	u_long *pciadr*	
return:	*value*	data uploaded from PCI memory.
	0	error indicated.
errors:	EINVAL	*pci* is NULL or *pciadr* out of bounds
	EIO	error in communicating with module.

The **dm4c51_up_i32b** function transfers a single 32-bit value from PCI memory on the module or baseboard referenced by *pci* to the host. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* must be modulo-2 aligned and must reside within the DSP's local memory space or the baseboard's global memory space.

The function returns the 32-bit value uploaded from PCI memory. Note that an error condition is indistinguishable from a successful value of 0. To avoid this ambiguity, it is recommended that the global variable **errno** be cleared prior to, and tested after the function call, for example:

```
u_long  pciadr, data;
extern int errno;

errno = 0;
data = dm4c51_up_i32b (pci, pciadr);
if  (errno != 0)
    perror("pci_up_i32b");
```

Note: This special 32-bit transfer function must be used to properly upload the data from the DM4C51 module for two reasons:

1. The format of a 32-bit value created by the Texas Instrument C compiler for the C51.

2. A 32-bit value does not have to start on a PCI address long-word byte boundary for the DM4C51.

function:	`int` **`pci_dl_a16b`** `(`*`pci`*`, pciadr, nwords, buf)`
	`int` **`pci_up_a16b`** `(pci, pciadr, nwords, buf)`

args:	`PCI_MOD  *`*`pci`*
	`u_long    pciadr, nwords, buf`

return:	`1          transfer was successful.`
	`0          error indicated.`

errors:	`EINVAL     `*`pci`*` is NULL, `*`pciadr`*` out of bounds, `*`nwords`*` is 0, or `*`buf`*` is NULL`
	`EIO        error in communicating with module.`

The **pci_dl_a16b** function transfers an array of 16-bit values from the memory of the host program to PCI memory on the module or baseboard referenced by *pci*. The **pci_up_a16b** function transfers an array of 16-bit values from PCI memory to the host program's memory.

The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP referenced by *pci* or global memory on the baseboard. The *nwords* argument specifies how many 16-bit words to transfer. The *buf* argument specifies the host program's data buffer.

function:	`int` **`pci_dl_i16b`** `(`*`pci, pciadr, value`*`)`

args:	`PCI_MOD` *`*pci`*
	`u_long` *`pciadr, value`*

return:	`1` `transfer was successful.`
	`0` `error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or` *`pciadr`* `out of bounds`
	`EIO` `error in communicating with module.`

The **pci_dl_i16b** function transfers a single 16-bit value from the host to PCI memory on the module or baseboard referenced by *pci*. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* must be modulo-2 aligned and must reside within the DSP's local memory space or the baseboard's global memory space.

function:	`u_long` **`pci_up_i16b`** (*`pci, pciadr`*)	
args:	`PCI_MOD`	`*`*`pci`*
	`u_long`	*`pciadr`*
return:	*`value`*	`data uploaded from PCI memory.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or` *`pciadr`* `out of bounds`
	`EIO`	`error in communicating with module.`

The **pci_up_i16b** function transfers a single 16-bit value from PCI memory on the module or baseboard referenced by *pci* to the host. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* must be modulo-2 aligned and must reside within the DSP's local memory space or the baseboard's global memory space.

The function returns the 16-bit value uploaded from PCI memory. Note that an error condition is indistinguishable from a successful value of 0. To avoid this ambiguity, it is recommended that the global variable **errno** be cleared prior to, and tested after the function call, for example:

```
u_long  pciadr;
u_short data;
extern int errno;

errno = 0;
data = pci_up_i16b (pci, pciadr);
if  (errno != 0)
    perror("pci_up_i16b");
```

4.2.3.1.4 C51 Executable Code Functions

The following functions provide for downloading TMS320C51 executable code in extended COFF format to the DSPs on a DM4C51 module and finding the PCI addresses of global variables in the program.

function: `int` **`pci_load_code`** `(`*`pci, codename, codetype`*`)`

args:
```
PCI_MOD   *pci
char      *codename
int        codetype
```

return:
```
1           code file read and downloaded
            successfully.
0           error indicated.
```

errors:
```
EINVAL      pci is NULL or codetype is unknown.
ENOEXEC     the file contains invalid COFF data.
EIO         error in communicating with module.
others      errors from the open() system call.
```

The **pci_load_code** function reads a TMS320C51 executable file and downloads the code to the DSP's memory. The file is specified in the *codename* argument. The *codetype* argument specifies the file format. For the TMS320C51, the format type should be the macro, **MM_COFF**, defined in <**pciutil.h**>.

In addition to downloading the code, this function builds a list of the symbols and their addresses found in the COFF file. The list is stored in the PCI_MOD structure for later access using the **pci_find_label_name** and **pci_find_label_address** functions.

function:	`int` **`pci_read_code`** `(`*`pci, codename, codetype`*`)`

args:	`PCI_MOD *`*`pci`*
	`char *`*`codename`*
	`int` *`codetype`*

return:	`1 code file read successfully.`
	`0 error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or` *`codetype`* `is unknown.`
	`ENOEXEC the file contains invalid COFF data.`
	`EIO error in communicating with module.`
	`others errors from the open() system call.`

The **pci_read_code** function reads a TMS320C51 executable file named *codename* in format *codetype*. The format type should be **MM_COFF**, a macro defined in <**pciutil.h**>.

pci_read_code does not download the code to DSP memory. It only reads the COFF file and builds a list of the symbols and addresses for the **pci_find_label_name** and **pci_find_label_address** functions. This is useful for host applications required to access a DSP that is already running the program.

function:	long **pci_find_label_name** (*pci, name*)	
args:	PCI_MOD	**pci*
	char	**name*
return:	*address*	memory address of named label.
	-1	label not found or error indicated.
errors:	EINVAL	*pci* is NULL

function:	char * **pci_find_label_address** (*pci, address*)	
args:	PCI_MOD	**pci*
	u_long	*address*
return:	*name*	pointer to label name string.
	NULL	label not found or error indicated.
errors:	EINVAL	*pci* is NULL

The **pci_find_label_name** function searches through the symbol list in the PCI_MOD structure (*pci*) for a label whose name matches *name*. If found, the PCI memory address of the label is returned. The address returned may be used as the *pciadr* argument to the data transfer functions.

The **pci_find_label_address** function searches through the symbol list in the PCI_MOD structure (*pci*) for a label at the address specified by the *address* argument. If found, a pointer to the label's name is returned.

For this function, values for address are memory addresses with respect to the DSP address rather than the PCI memory space.

function:	`void` **`pci_free_labels`** `(`*`pci`*`)`
args:	`PCI_MOD` `*`*`pci`*
return:	`none`
errors:	`EINVAL` *`pci`* `is NULL`

The **pci_free_labels** function deletes the symbol table list from the PCI_MOD structure and frees up the memory allocated to store the list. This operation is required in situations where a completely new DSP program is to be loaded using the same PCI_MOD structure that has been used to download a previous program. Freeing the previous symbol table will ensure that the PCI_MOD structure contains only the symbols for the new program.

4.2.3.1.5 TDM Subsystem Control Functions

The following library functions are used to control the operational parameters of the TDM subsystem.

function:	`int` **`pci_tdm_init`** `(`*`tdm, clk, bits, slots, frames`*`)`	
args:	`PCI_TDM`	`*`*`tdm`*
	`int`	*`clk, bits, slots, frames`*
return:	`1`	`TDM controller initialized.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`tdm`* `is NULL or other arguments are invalid.`

The **pci_tdm_init** function initializes the parameters of the TDM subsystem controller.

For the DM4C51, use the following values for the arguments:

clk	When any module or the TDM expansion port is configured to generate the TDM subsystem clock, this argument must be 0. To generate TDM clocks from the TDM subsystem controller, a value of 2, 4, or 8 should be used.
bits	Bits per time slot, 8, 16 or 32.
slots	This value determines the number of TDM time slots per TDM frame and may range from 2 to 128. If another board connected to the TDM expansion port is generating TDM frame and superframe pulses, this value should be 0.
frames	This value determines the number of TDM frames per TDM superframe and may range from 0 to 31. If set to 0, the TDM controller will not generate superframe pulses.

function:	`int` **`pci_tdm_run`** `(`*`tdm`*`)`	
args:	`PCI_TDM`	`*`*`tdm`*
return:	`1`	`TDM subsystem started.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`tdm`* `is NULL or other arguments are invalid.`
	`EIO`	`failed to update the TDM MAP RAM, usually due to the absence of TDM clocks.`

The **pci_tdm_run** function starts the TDM subsystem timing logic. The TDM map is updated, if necessary, and the TDM controller will begin generating the sync pulses as specified by the **pci_tdm_init** function.

4.2.3.1.6 DM4C51 TDM Connection MAP Functions

The following library functions provide the means for setting up TDM data transfer connections between DM4C51 modules and other modules.

function:

```
int dm4c51_tdm_src_add (pci, slot, bus)
int dm4c51_tdm_src_del(pci, slot, bus)
```

args:

```
PCI_MOD   *pci
int        slot
int        bus
```

return:

```
1          SC4000 modified.
0          error indicated.
```

errors:

```
EINVAL      pci is NULL or does not reference a
            DM4C51 DSP, or other arguments are
            invalid.
EIO         error in communicating with module.
```

These two functions add or delete a device as a TDM data source for the specified time *slot* and TDM *bus*. Only one device on a board may be the source for a TDM bus during a time slot.

The following arguments values are used for the DM4C51 module:

slot	Values between 1 and the number of slots configured for the TDM subsystem frame (see **pci_tdm_init**).
bus	One of the macros defined in <**pciutil.h**>, **TDM_BUSA**, **TDM_BUSB**, **TDM_BUSC** and **TDM_BUSD**, corresponding to the four TDM busses.

function:	`int` **`dm4c51_tdm_dst_add`** `(`*`pci, slot, bus)`*
	`int` **`dm4c51_tdm_dst_del`** `(`*`pci, slot, bus`*`)`

args:	`PCI_MOD *`*`pci`*
	`int` *`slot`*
	`int` *`bus`*

return:	`1` — `SC4000 modified.`
	`0` — `error indicated.`

errors:	`EINVAL` — *`pci`* `is NULL or does not reference a DM4C51 dev, or other arguments are invalid.`
	`EIO` — `error in communicating with module.`

These two functions add or delete a device as a TDM data destination for the specified time *slot* and TDM *bus*. A device may be a destination for only one TDM bus during a time slot. The use of the arguments is the same as the **dm4c51_tdm_src_add** function.

4.2.3.1.7 SC4000 Control functions

function:	`int` **`dm4c51_sc4000_init`** `(`*`pci, tdmframe`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`int`	*`tdmframe`*
return:	`1`	`SC4000 modified.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference a DM4C51 dev, or tdm frame size is not 32, 64 or 128 invalid.`

The function is used to reset and configure the SC4000. The *tdmframe* specifies the number of slots in a TDM frame. The function also disables the TDM DMA controller.

The following SC4000 control functions are typically not called directly by the user application program. These functions are called by the DM4C51 TDM connection mapping functions for configuring the SC4000 and the TDM DMA controller.

```
function:  int dm4c51_sc4000_read (pci, sc4000reg)

args:      PCI_MOD   *pci
           int        sc4000reg

return:    value       SC4000 register.
           -1          error indicated.

errors:    EINVAL      pci is NULL or does not reference a
                       DM4C51 dev, or sc4000reg >= 0x200
           EIO         error in communicating with module.
```

The function is used read the parallel access registers of the SC4000. See the SC4000 data sheet for details.

```
function:  int dm4c51_sc4000_write (pci, sc4000reg, data)

args:      PCI_MOD   *pci
           int        sc4000reg
           int        data

return:    1           SC4000 modified.
           0           error indicated.

errors:    EINVAL      pci is NULL or does not reference a
                       DM4C51 dev, or sc4000reg >= 0x200
           EIO         error in communicating with module.
```

The function is used to write the parallel access registers of the SC4000. See the SC4000 data sheet for details.

function:	`int` **`dm4c51_cfg_read`** `(`*`pci, sc4000reg`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`int`	*`sc4000reg`*
return:	`value`	`SC4000 register value.`
	`-1`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference a DM4C51 dev, or sc4000reg > 12.`
	`EIO`	`error in communicating with module.`

The function reads the indirect configuration registers of the SC4000. See the SC4000 data sheet for details.

function:	`int` **`dm4c51_cfg_write`** `(`*`pci, sc4000reg, data`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`int`	*`sc4000reg`*
	`int`	*`data`*
return:	`1`	`SC4000 modified.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference a DM4C51 dev, or sc4000reg > 10`
	`EIO`	`error in communicating with module.`

The function write an indirect SC4000 configuration register. See the SC4000 data sheet for details.

function:	`int` **`dm4c51_route_read`** `(`*`pci, LCLSlot, dir)`*

args:	`PCI_MOD *`*`pci`*
	`int LCL`*`slot`*
	`int` *`dir`*

return:	`value SC4000 modified.`
	`-1 error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or does not reference a DM4C51 dev, or` *`LCLslot`* `> 32`
	`EIO error in communicating with module.`

The function returns the value of the SC4000 routing register specified by the *LCLslot*. *LCLslot* is the offset into the SC4000 routing registers specific for the C51 pointed by pci.

function:	`int` **`dm4c51_route_write`** `(`*`pci, LCLslot, TDMslot, TDMbus, dir, oe)`*

args:	`PCI_MOD *`*`pci`*
	`int` *`LCLslot`*
	`int` *`TDMslot`*
	`int` *`TDMbus`*
	`int` *`TDMdir`*
	`int` *`oe`*

return:	`1 SC4000 modified.`
	`0 error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or does not reference a DM4C51 dev, or another parameter is invalid.`
	`EIO error in communicating with module.`

The function write the SC4000 routing register specified by *LCLslot*. *LCLslot* is the offset into the SC4000 routing registers specific for the C51 pointed by pci. The *TDMslot* and *TDMbus* specifies which TDM slot of the TDM frame and which TDM bus the C51 should be sourcing data. The *oe* is the output enable control bit. A non-zero value indicates the C51 should be driving the specified TDM slot and bus.

function:	`int` **`dm4c51_sc4000_mode`** `(`*`pci, dma, reset, count`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`int`	*`dma`*
	`int`	*`reset`*
	`int`	`count`
return:	`1`	`DMA Control Register modified.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference a DM4C51 dev.`
	`EIO`	`error in communicating with module.`

The function configures the TDM DMA controller and the SC4000 reset. Setting *dma* to one enables the DMA controller, setting *reset* to a one releases the SC4000 reset signal and *count* is the number of TDM subsystem to local memory buffer transfers.

function:	`int` **`dm4c51_read_mode`** `(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`value`	`TDM DMA and SC4000 reset configuration.`
	`-1`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference a DM4C51 dev.`
	`EIO`	`error in communicating with module.`

The function returns the configuration of the TDM DMA controller and the SC4000 reset signal. The format of the return data is shown below:

bit 0 - 4	- dma count - 2
bit 6	- dma enable
bit 7	- SC4000 reset, active low

4.2.3.2 C51 DSP Programming

Writing and executing programs in the Texas Instrument TMS320C51 dsp requires considerations of the hardware environment. The CAC PCI software distribution includes several files to aid in the generation and execution of DSP applications. These files are located in the directory **$CAC/pci/modsupport/dm4c51**. Files other than the ones described here are the source code for the DSP portions of the DM4C51 diagnostic programs. They are intended to be used as programming examples.

4.2.3.2.1 Linker Command Files

Linker command files control the building of a DSP executable program. The linker file, for the DM4C51 is named **dm4c51.cmd**. It performs the following tasks:

- allocates 256 words for stack space in the end of internal memory bank 1 starting at 0x1fe60.
- reserves 48 half-words starting at zero for the interrupt vector table
- allocates 8194 half-words for heap space
- puts the start up code right after the interrupt vector table at address 0x30
- locates the text and data sections in external SRAM
- includes the CAC startup up code, C51int00.obj
- includes the standard C library

The linker command file for the 256 Kbytes dm4c51 has the program memory address at 0x0000 and the data memory start at 0x10800. Even though the c51 is only a 16-bit address dsp, the data memory must start above 0x10000. The C compiler for the C51 will only use the lower 16 bits of the address. Bit 16 is used by the host application to distinguish between program space and data. When bit 16 is set, the host accesses data space.

Below is the contents of the 128K linker file for the C51 on DM4C51 modules:

```
/*****************************************************************/
/* TI C51 linker command file for DSP Module 4 C51 Board (128K) */
/*****************************************************************/
-c                              /* Link using C Conventions      */
-v0                             /* Generate Version 0 COFF format */
-w                              /* Generate Warnings             */
-stack 0x100                    /* 256 Stack                     */
-heap  0x2002                   /* 8194 Heap - Size to allocate  */
                                /* one 4095 dma buffer           */
-lc51int00.obj                  /* must be before run-time lib.  */
-lrts50.lib                     /* Run-time Support              */

MEMORY
{
  PAGE 0: VECT:   o = 0x0000,  l = 0x30    /* Interrupt Vectors */
          SRAM0:  o = 0x0030,  l = 0xffd0  /* 64K words         */

  PAGE 1: SRAM1:  o = 0x10800, l = 0xf660  /* 64K words         */
          STACK:  o = 0x1fe60, l = 0x0100  /* Stack Space       */
}

SECTIONS
{
  vectors: {c51int00.obj(.vect)} > VECT PAGE 0  /* Vector Table      */
  .boot:  {c51int00.obj(.text)} > SRAM0 PAGE 0  /* Start code in mem */
  .cinit:  > SRAM0 PAGE 0                    /* Initialization Tables */
  .text:   > SRAM0 PAGE 0
  .switch: > SRAM0 PAGE 0                  /* Switch Statement Tables */
  .data:   > SRAM0 PAGE 0                  /* Assembly Language Const */
  .const:  > SRAM1 PAGE 1                  /* Constants               */
  .bss:    > SRAM1 PAGE 1, block = 0x80, fill = 0
  .sysmem: > SRAM1 PAGE 1                  /* Dynamic Memory          */
  .stack:  > STACK PAGE 1                  /* System Stack            */
}
```

Below is the contents of the linker file for the C51 on DM4C51_64K modules:

```
/*****************************************************************/
/* TI C51 linker command file for DSP Module 4 C51 Board (64K)  */
/* configured with a 64 Kword unified Program/Data Memory Space */
/*****************************************************************/
-c                              /* Link using C Conventions       */
-v0                             /* Generate Version 0 COFF format */
-w                              /* Generate Warnings              */
-stack 0x100                    /* 256 Stack                      */
-heap  0x2002                   /* 8194 Heap - Size to allocate   */
                                /* one 4095 dma buffer            */
-lc51int00.obj                  /* must be before run-time lib.   */
-lrts50.lib                     /* Run-time Support               */

MEMORY
{
  PAGE 0: VECT:   o = 0x10000, l = 0x30    /* Interrupt Vectors */
          SRAM1:  o = 0x10800, l = 0xf660  /* 64K words         */
          STACK:  o = 0x1fe60, l = 0x0100  /* Stack Space       */
}

SECTIONS
{
  vectors: {c51int00.obj(.vect)} > VECT PAGE 0  /* Vector Table      */
  .boot:   {c51int00.obj(.text)} > SRAM1 PAGE 0 /* Start code in mem */
  .cinit:  > SRAM1 PAGE 0               /* Initialization Tables   */
  .text:   > SRAM1 PAGE 0
  .switch: > SRAM1 PAGE 0               /* Switch Statement Tables */
  .data:   > SRAM1 PAGE 0               /* Assembly Language Const */
  .const:  > SRAM1 PAGE 0               /* Constants               */
  .bss:    > SRAM1 PAGE 0, block = 0x80, fill = 0
  .sysmem: > SRAM1 PAGE 0               /* Dynamic Memory          */
  .stack:  > STACK PAGE 0               /* System Stack            */
}
```

4.2.3.2.2 DM4C51 Start-Up Code

The start-up code for the DM4C51 is in file **C51int00.src**. The code performs the following tasks:

- initializes the stack pointer
- reserves space for the C51 interrupt vector table
- configures the Program / Data space wait states to 0
- configures the I/O space wait states to 7
- processes the runtime initialization table and initializes the global variables
- calls the function **main**

4.2.3.2.3 DM4C51 Header File

The file, **dm4c51.h**, contains macros for the addresses and special bits of many of the DSP internal and memory mapped registers and registers on the DM4C51 module. The file also includes some functional macros for operations such as controlling the LED and determining the DSP's module and ID.

4.2.3.2.4 DSP ID and Module ID

The DSP ID and Module ID are available to the C51 application program in two locations of data space memory. The two locations are initialized by the **dm5c41_run** function. The table below shows their addresses.

Name	Address	Description
DSPIDptr	0xff68	The DSP ID (0 - 3)
MODIDptr	0xff69	The module ID (0 - 5) corresponding to module site 'A' through 'F"

4.2.3.3 Utility Programs

The following programs, in **$CAC/bin**, are provided for the V6M6 PCI carrier boards. They are useful for initializing and maintaining the DM4C51 module. Detailed descriptions of these programs are in **Chapter 3. V6M6 Baseboard.**

pciinit	Instructs the on-board microprocessor to configure FPGAs on the module. It then initializes the baseboard and modules and sets up PCI address mappings.
pciinfo	Examines the EEROMs on the baseboard and modules and the flash memory. It then displays all pertinent version and serial number information.
pciflashup	Compares flash object (FPGA configurations and microprocessor program) versions in the flash memory with those residing in host files. Depending on the command line options it will either display the version numbers or update the flash memory with any new versions on disk.

The following two programs are used to set the clock frequencies of the PCI bus, TDM bus and the TMS320C51 DSPs. They are not installed as part of the normal software installation because the clock frequencies are set to specifications prior to delivery of the boards. However, the source code and a Makefile are distributed in **$CAC/pci/pciutils** and they may be compiled and run in case the clocks need to be modified in the field.

pcifreq

Adjusts the clock synthesizers for the PCI bus and TDM bus clock frequencies on version 2 and above V6M6 boards. The usage of the program is:

```
pcifreq pci_board [pci_freq [tdm_freq]]
```

Where pci_board is the name of the V6M6 board (e.g. pci0), pci_freq is desired clock frequency for the PCI bus in MHz (default is 20) and tdm_freq is the desired maximum TDM clock frequency in MHz (default is 16.384). Note that in order to specify the TDM clock, the PCI clock must be specified first.

The program adjusts the settings, asks the user to reset the board using the reset switch on the front panel, and then runs the pciinit program to reinitialize the board with the new settings.

dmfreq

Adjusts the clock for the DSPs on DM4C51 modules by setting the frequency multiplier value. Usage of the program is:

```
    dmfreq  -c51  pci_board  [dm_freq]
```
or
```
    dmfreq  -c51  dm_module  [dm_freq]
```

The first form is used to adjust the DSP clock for all DM modules found on the specified V6M6 board, pci_board (e.g. pci0). The second form adjusts the clock for the DSPs on the specific dm_module named as a V6M6 board and module letter (e.g. pci0a). In both cases dm_freq specifies the DSP clock frequency in MHz. The default frequency is 41.5 MHz

The program adjusts the setting, asks the user to reset the board using the reset switch on the front panel, and then runs the pciinit program to reinitialize the board with the new settings.

Note that the **-c51** option instructs dmfreq to adjust only DM4C51 modules. Without this option the program will also operate on DM2C31 modules.

4.2.3.4 Diagnostic programs

The following diagnostic programs are provided for the DM4C51 module in the directory, **$CAC/pci/diag**.

pcipiomem	Tests communication and data transfer between the host and memory on the baseboard and modules.
pcimemory	Runs a memory diagnostic from the DSPs.
pcimemslice	Runs a memory diagnostic with the host and all processors found on the baseboard such that each processor is testing a slice of global memory and local memory on all the other modules.
pcichip	Test various functions of the DSP processors and interfaces.
pciburn	Runs the set of diagnostics on all V6M6 boards installed in the system or a specified list of boards. It logs the tests that are run and any errors reported by the diagnostics.

5. I/O Modules

5.1 IM10BT

(documentation not yet available)

5.2 IM2E1

The IM2E1 provides two CEPT/E1 interfaces for the V6M6 PCI module carrier boards. Electrical interfaces for HDB3 networks are provided via RJ45 jacks, one for each E1 line. E1 frame encoding and decoding is performed on each E1 line by a Dallas Semiconductor DS2153Q CEPT Transceiver. Elastic frame buffers (DS2175) interface the outgoing E1 streams to the TDM subsystem. Incoming E1 data is passed through the internal elastic store of the DS2153Q framer chip.

Information in this manual is divided into the following sections:

5.2.1 IM2E1 Quick Tour

Figure 4-1 is a block diagram of the IM2E1 showing the major components and signal paths. The primary components are the two Dallas Semiconductor DS2153Q E1/CEPT Transceivers. Please refer to the DS2153Q data sheet (provided in Appendix A: "Component Data Sheets") for details on their operation.

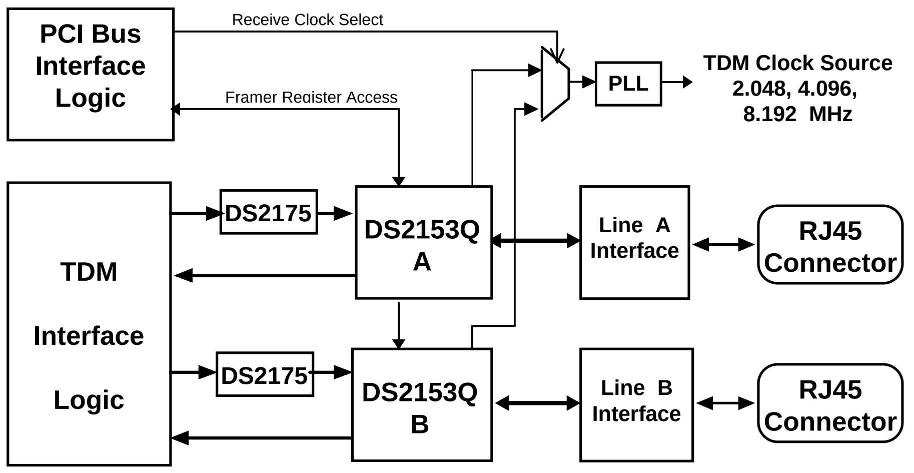


Fig. 4-1: IM2E1 Module Block Diagram

The two E1 lines, designated A and B, connect to the board through RJ45 connectors. The E1 signals pass through the line interface components to and from the DS2153Q framers. The input impedance for each line is selectable to be 75Ω or 120Ω (see Section 5.2.2.3 Electrical Line Interface). The output impedance is controlled by the line build-out select in the DS2153Q's LICR registers (see table 12-2 in the Dallas DS2153Q data sheet). The output pulse transformer ratios are 1:1.15.

The DS2153Q E1 components provide clock and data recovery for the incoming E1 lines and E1 framing and pulse generation for the outgoing E1 lines. Access to the DS2153Q control, status and data registers is provided through the PCI interface.

Incoming and outgoing E1 data is accessed by other PCI modules via the TDM subsystem of the V6M6 board. Incoming multiframe and loss of signal status is also available via the TDM subsystem. The TDM interface is designed to be operated with 8-bit TDM time slots and TDM frames of 32 slots / 2.048 MBPS, 64 slots / 4.096 MPBS or 128 slots (8.192 MPBS). A phase-locked loop may be used to synchronize the TDM subsystem clock with one of the incoming receive clocks.

FIFO storage between incoming E1 data and the TDM subsystem is provided by the DS2153Q framers. The two Dallas Semiconductor 2175 elastic store buffers provide a FIFO interface for outgoing E1 data.

IM2E1 modules may be installed on V6M6 boards at module locations A and C for front panel access to the RJ45 connectors. A future option will allow IM2E1 modules to be installed at module locations B, D or F with its I/O connections made via the VME P2 connector.

5.2.2 IM2E1 Hardware Description

The IM2E1 and IM2T1 use the same FPGA configuration data stored in the Flash ROM on the V6M6. The flash object name is **imtelco**. The PCI interface logic uses the same generic, configurable design as other IM modules for the V6M6. Configuration of this logic is performed by the host system using the V6M6 board initialization program, **pciinit**.

5.2.2.1 Module Control and E1 Framer Registers

The registers for control of the module and the E1 framers are accessed in PCI bus configuration space and i/o space. The table below shows the base addresses for the module locations that may be used to carry IM2E1 modules. The configuration space base addresses are determined by physical hardware connections. I/O space base addresses are determined by convention and are hard coded in various software modules. The values are loaded into the IM2E1 registers by the **pciinit** program.

PCI Base Addresses for IM2E1 Modules

PCI Module	Config Space Base Address	I/O Space Base Address
A	0x01000000	0x30000000
B	0x00800000	0x31000000
C	0x00400000	0x32000000
D	0x00200000	0x33000000
F	0x00080000	0x35000000

The following two tables list the IM2E1 registers. The address or address range shown are the offsets from the base address of the module.

IM2E1 Configuration Space Registers			
Register Name	Address	Acc	Function / Description
PCI Stat/Cmd	0x04	rw	See PCI Spec for details
Mod Base	0x10	wo	PCI base address of module
Dev Cmd	0x40	rw	Generic I/O module controls and settings
Interrupt Stat/Mask	0x44	rw	I/O Interrupt stat and mask
Interrupt Ctl	0x48	rw	Controls interrupt polarity

IM2E1 I/O Space Registers			
Register Name	Address Range	Acc	Function / Description
E1 Framer 0	0x000 to 0x3FC	rw	Line A DS2153Q Registers (see Dallas Semi. data sheet)
E1 Framer 1	0x400 to 0x7FC	rw	Line B DS2153Q Registers (see Dallas Semi. data sheet)
TDM ID	0x800	rw	Stores module location ID
Clock Ctl	0x804	rw	Controls clock sources for TDM Subsystem and E1 Transmit
RSYNC Ctl	0x808	rw	Controls TDM SYNC drive to RSYNC to DS2153Q Framer
Slip Ctl	0x80C	rw	Controls slipped frame action and clears Fast RCLK Status

PCI Stat / Cmd Register:

This register is used for configuring and obtaining status of the module's interface to the PCI bus. The control bits are all either hardwired or programmed by the **pciinit** program. The bits set by **pciinit** are:

```
Bit 0 = 1: I/O Space Enabled
Bit 6 = 1: PCI Parity Error Response Enabled
```

Two status bits, listed below, are available for PCI error checking. These are cleared by writing a 1.

```
Bit 30 - Module signaled a System Error
Bit 31 - Module detected a PCI Parity Error
```

Mod Base Register:

This register stores the module's I/O space base address on the PCI bus. It is programmed by the **pciinit** program and a copy of its contents is stored in the host's device driver and made available to host applications.

Dev Cmd Register:

This register is used to control low-level module functions.

```
Bits 31:28 - Timing for the module's internal bus
Bit  23    - Module Reset (low active)
```

The timing bits are programmed by the **pciinit** program to meet timing requirements of the E1 Framer devices on the module. Module Reset is only used to center the DS2175 Elastic Storage FIFOs.

Interrupt Mask / Stat Register:

This register controls the interrupt enable and reads interrupt status for the module.   The IM2E1 supports the 2 interrupt sources from both of the DS2153Q Framers and a special interrupt indicating that one of the E1 line’s RCLK is running at a faster rate than TDMCLK.

The interrupt mask bits allow any of the interrupt sources to be the source of PCI interrupt A or B. For each bit, a 1 passes the interrupt and a 0 masks it.

```
Bit 0 - Framer 0 Int 1 to PCI interrupt A
Bit 1 - Framer 0 Int 2 to PCI interrupt A
Bit 2 - Framer 1 Int 1 to PCI interrupt A
Bit 3 - Framer 1 Int 2 to PCI interrupt A
Bit 4 - Framer 0 Int 1 to PCI interrupt B
Bit 5 - Framer 0 Int 2 to PCI interrupt B
Bit 6 - Framer 1 Int 1 to PCI interrupt B
Bit 7 - Framer 1 Int 2 to PCI interrupt B
Bit 8 - RCLK > TDMCLK to PCI interrupt A
Bit 9 - RCLK > TDMCLK to PCI interrupt B
```

The interrupt status bits are valid regardless of the interrupt mask setting. A 1 indicates that the interrupt source is active.

```
Bit 24 - Framer 0 Int 1
Bit 25 - Framer 0 Int 2
Bit 26 - Framer 1 Int 1
Bit 27 - Framer 1 Int 2
Bit 28 - RCLK > TDMCLK
```

Interrupt Ctl Register:

This register is written by the **pciinit** program to match the polarity of the E1 Framer interrupts to the polarity expected on the PCI interrupt lines.

E1 Framer Registers:

The various registers of the DS2153Q Framers are accessed via PCI I/O space. Details of the registers' functions and contents are available in the Dallas Semiconductor DS2153Q data sheet.

Framer 0 is accessed starting at offset 0x000 from the module's I/O base. Framer 1 is accessed starting at offset 0x400 from the module's base. Each of the 8-bit registers are accessed on 32-bit boundaries with the register data in the LSB of the word. They may be accessed as 32-bit words (with only the lower 8-bits containing useful data) or as 8-bit words. In either case the DS2153Q register offset, as specified in the data sheet must be shifted left by two bits then added to the module base and framer base addresses. For byte access, the address must be further offset by 3.

32-bit access:

```
PCIaddr = Mod_IObase + Framer_base + (Reg_offset << 2)
```

8-bit access:

```
PCIaddr = Mod_IObase + Framer_base + (Reg_offset << 2) + 3
```

32-bit example: LICR register of Framer 1 on Module C

```
PCIaddr = 0x32000000 + 0x400 + (0x18 << 2) = 0x32000460
```

Certain control register bits must be set to the following values in order for the IM2E1 to operate properly and maintain synchronization with the TDM subsystem. These are shown in the table below. Other framer controls and parameters are application dependent.

Register	Offset	Bit	Val	Comment
RCR1	0x10	5	0	RSYNC pin is an output
RCR1	0x10	6	1	RSYNC pin in multiframe mode
RCR2	0x11	1	1	Rcv Elastic Store Enabled
RCR2	0x11	2	1	SCLK is 2.048 MHz
TCR1	0x12	0	0	TSYNC pin is an input
TCR2	0x13	0	0	RLOS/LOTC pin indicates RLOS
CCR3	0x1B	7	0	Xmt Elastic Store Disabled
TAF	0x20	6:0	0x1b	Frame alignment bits
TNAF	0x21	6	1	Frame alignment bit

TDM ID Register:

This register stores the module ID (location) of the module and is used for identifying the module for TDM data transfers. Bits 2:0 contain the ID value as programmed by the **pciinit** program.

Clock Ctl Register:

This register is used to control the relationship between the TDM subsystem clock and the E1 receive and transmit clocks.

The clock divider bits determine the relationship between the E1 clock rate and the TDM clock rate.

```
Bits 1:0 - Clock Divider
            00 : TDM clock = E1 Clock * 1 (2.048 MHz)
            01 : TDM clock = E1 Clock * 2 (4.096 MHz)
            10 : TDM clock = E1 Clock * 4 (8.192 MHz)
```

The TDM clock may be derived from either of the two Framer's RCLK or not from the module at all. The IM2E1 (configured with version 13 or higher of the **imtelco** FPGA configuration) includes logic to detect and select the faster of the two received clocks. This is useful in applications where the incoming E1 lines may have different time bases. The value of bits 1:0 multiply the selected RCLK to derive TDM clock.

```
Bits 3:2 - TDM Clock select bits
            00 : TDM clock not sourced by the module
            01 : TDM clock derived from RCLK A
            10 : TDM clock derived from RCLK B
            11 : TDM clock derived from the faster of RCLK A
                 and RCLK B
```

E1 transmit clock sources may be derived from either Framer's RCLK or from the TDM subsystem clock. When TDM-derived TCLK is selected, the value of bits 1:0 divide TDM clock to derive TCLK.

```
Bits 5:4 - TLCK B Select
            00 : TCLK B = RCLK B
            01 : TCLK B = RCLK A
            10 : TCLK B derived from TDM Clock

Bits 7:6 - TLCK A Select
            00 : TCLK A = RCLK A
            01 : TCLK A = RCLK B
            10 : TCLK A derived from TDM Clock
```

RSYNC Ctl Register:

This register controls whether or not the TDM SYNC signal is driven to the E1 framer’s RSYN pin. It is used by the **im2e1_init_rsync** function to initialize synchronization between the E1 framer and the TDM subsystem.

SLIP Ctl Register:

This register enables or disables the IM2E1 logic from invalidating TDM bus data for slipped frames. The **im2e1_slipinval** library function accesses this register to control this mode. Bits 0 and 1 of the register determine whether this mode is enabled for each E1 line:

```
Bit 0 -
      0 : Slip invalidation mode disabled for line 0
      1 : Slip invalidation mode enabled for line 0
Bit 1 -
      0 : Slip invalidation mode disabled for line 1
      1 : Slip invalidation mode enabled for line 1
```

5.2.2.2 TDM Interface

The primary method for transferring incoming and outgoing E1 data is via the TDM subsystem. The serial data input and output of the E1 framers are connected to the TDM data busses through elastic storage buffers, allowing the TDM clock rate to be multiple of the E1 data rate.

Data is transferred between a TDM data bus and an elastic storage buffer on a slot-by-slot basis based on commands stored in the TDM subsystem MAP RAM. The MAP is built and loaded from the host system using the PCI library TDM functions described in Section **Error! Reference source not found. Error! Reference source not found.**.

The TDM interface logic also decodes the TDM control words generated from the TDM Map RAM on the base board. There are 8 control words for each time slot which determine the connections between the DSPs and the TDM subsystem busses. The TDM map is programmed using C functions provided in the V6M6 Host Application Library (see Section 5.2.3.1 V6M6 Host Application Library for IM2E1).

5.2.2.2.1 TDM Map Control Words

The TDM control words, as stored in the TDM Map RAM, are described below:

IM2E1 TDM MAP Control Words

Control Word	Bits 7:4	Bits 3:0
0	TDMA-SRC	TDMB-SRC
1	TDMC-SRC	TDMD-SRC
2	MOD4-CTL	MOD5-CTL
3	MOD2-CTL	MOD3-CTL
4	MOD0-CTL	MOD1-CTL
5	MOD4-DST	MOD5-DST
6	MOD2-DST	MOD3-DST
7	MOD0-DST	MOD1-DST

<u>TDM SRC</u> The 4-bit code in each of the TDM Source selectors determines the device to source the TDM Bus.

0x0	Line A on Module A sources the TDM Bus
0x1	Line A on Module B sources the TDM Bus
0x2	Line A on Module C sources the TDM Bus
0x3	Line A on Module D sources the TDM Bus
0x4	Line A on Module E sources the TDM Bus
0x5	Line A on Module F sources the TDM Bus
0x6	Line B on Module A sources the TDM Bus
0x7	Line B on Module B sources the TDM Bus
0x8	Line B on Module C sources the TDM Bus
0x9	Line B on Module D sources the TDM Bus
0xA	Line B on Module E sources the TDM Bus
0xB	Line B on Module F sources the TDM Bus
0xF	No source for the TDM Bus

<u>MOD CTL</u> The 2-bit codes in these words control loading and unloading of the E1 data elastic buffers on the module.

Bits 1:0	Loading of outgoing buffers
11	No load
10	Load Line A buffer
01	Load Line B buffer
00	Load Line A and Line B buffers

Bits 3:2	Unloading of incoming buffers
11	No unload
10	Unload Line A buffer
01	Unload Line B buffer
00	Unload Line A and Line B buffers

<u>MOD DST</u> The 2-bit codes in these words select which TDM Bus is used for getting data to the outgoing E1 data elastic buffers on the module.

Bits 1:0	Bus for Data to Line A
00	TDM Bus A
01	TDM Bus B
10	TDM Bus C
11	TDM Bus D

Bits 3:2	Bus for Data to Line B
00	TDM Bus A
01	TDM Bus B
10	TDM Bus C
11	TDM Bus D

5.2.2.2.2 TDM Framing

Exactly 32 bytes (time slots) per TDM frame must be transferred to and from each of the E1 line's elastic buffers. It is possible to read and write the buffers for both lines during the same time slot. It is also possible to unload a byte from an incoming buffer without driving the data onto a TDM bus. This allows unused data to be purged without using up TDM bandwidth.

TDM frames must always be set for 125 microseconds. They may be configured as 32 slots at 2.048 MHz, 64 slots at 4.096 MHz or 128 slots at 8.192 MHz. TDM framing parameters are configured using host application functions described in section 4.3.1.2. When frames of 64 or 128 slots are used, it is recommended that the E1 slots be spread evenly across the available TDM time slots.

The E1 data frames, incoming and outgoing, are aligned with the TDM frames. Outgoing multiframes are synchronized with the multiframe sync generated by the TDM subsystem. Incoming multiframes are synchronized with the receive multiframe sync for each line.

The first incoming slot contains framer status information and some of the bits of the E1 alignment slot (slot 0), as described below. The other 31 E1 data slots are available as received from the RSER pin of the framers.

Contents of first slot from each E1 line:

Bit 7	MSB First						Bit0
Si	TDM MF	Alarm	Sn0	RLOS B	RLOS A	RSYNC B	RSYNC A

Si	International spare bit from slot 0 of selected line.
TDM MF	State of the MultiFrame Sync generated by the TDM Subsystem - indicates 1st frame of a transmit multiframe.
Alarm	Remote alarm bit from slot 0 of selected line.
Sn0	1st national spare bit from incoming slot 0 of selected line.
RLOS A/B	Sampled output of framers' RLOS pin - indicates loss of receive sync.
RSYNC A/B	Sampled output of framers' RSYNC pin - indicates 1st frame of a receive multiframe.

5.2.2.2.3 TDM Clock Rates and Frame Slips

Most applications have the TDM subsystem clock, TDMCLK, derived from the clock received from an incoming E1 line. The E1 transmit clock, in turn, is usually derived from TDMCLK. The library functions, **im2e1_rcvclk**, **im2e1_xmtclk** and **im2e1_tdmclkdiv**, control the TDM and E1 clock timing. These functions are described in 5.2.3 IM2E1 Software Support.

For the majority of systems, multiple incoming E1 lines will be running at the same clock rate, being derived from a common timing source. In a few situations, however, the timing of multiple incoming E1 lines may be derived from different time bases; their clocks may be running at slightly different frequencies. Depending on which receive clock is faster, data from one of the incoming lines could be either lost or repeated as the elastic storage buffer in the framer chip under-flows or over-flows. This data loss or repetition occurs at a frame boundary; an entire frame is either lost or repeated at a time.

The IM2E1 has logic to compare the frequencies of its two receive clocks, to determine which is faster and to use the faster of the two clocks to derive the TDM subsystem clock. The **im2e1_rcvclk** function is used by host applications to select this mode of operation. With TDMCLK derived from the faster E1 receive clock, any frame slips occurring on the other E1 line will be due to elastic store under-flows and data will be repeated rather than lost. Should the frequencies of the two incoming E1 lines drift, the IM2E1 will automatically switch which receive clock is used to derive TDMCLK. Furthermore, if one of the framers is not in sync, the IM2E1 will automatically select the other framer's receive clock to derive TDMCLK.

Additional logic, available in Revision 2 of the IM2E1, provides a mode by which TDM data from a slipped frame is invalidated. This is accomplished using the TDM VALID signals that accompany each TDM data bus. For each TDM slot sourced by an E1 line during a slipped frame, the corresponding TDM VALID is pulsed low. Modules capable of interpreting the TDM Valid signals will be able to filter out data repeated due to slips. The slipped frame invalidation mode is enabled using the **im2e1_slipinval** function described in the IM2E1 Software Support section of this manual.

Slips are indicated to the IM2E1 logic by an interrupt from the DS2153 framer. The program controlling the IM2E1 (running either on the host or a processor on the V6M6) must enable and service this interrupt. To enable the interrupt, bit 4 of the framer's **Interrupt Mask Register 1** must be turned on. Servicing the interrupt requires the following standard sequence of accessing status from the framer:

Write a 1 to bit 4 of **Status Register 1**
Read **Status Register 1**
Write 1 to bit 4 of **Status Register 1**

Important notes

1. The slip interrupt must be serviced in time for the next frame slip to be recognized. The time between slips will vary depend on the frequency offset between the two incoming RCLKs. For example, with a frequency offset of 0.01%, a slip can be expected every 1.25 seconds.

2. Status Register 1 and the corresponding Interrupt Mask Register 1 contain other important status bits that an application is likely to be interested in and that may be used for other interrupt conditions.

Yet another situation for some applications is the use of more than one IM2E1 module. If the timing for all of the incoming E1 lines is from the same time base, all of the incoming E1 data will remain in sync. However, if the E1 lines are coming from different timing sources the possibility of slippage and lost data becomes an issue.

To aid in this situation, the IM2E1 compares its receive clock frequencies with the frequency of TDMCLK. When one of its two receive clocks is faster than TDMCLK, it sets a status flag which may be enabled to assert a PCI interrupt. The interrupt is enabled using the **im2e1_intenb** function. The **im2e1_intstat** function is used to read the status of this detection is read using the. The status condition is latched and remains active until cleared by calling the **im2e1_clrfastclk** function.   When this interrupt is serviced, the application monitoring E1 performance may decide to switch which IM2E1 module is used to source TDMCLK.

5.2.2.3 Electrical Line Interface

The IM2E1 interfaces to external E1 networks via two RJ45 connectors, one for each line. The electrical interface includes voltage clamping and current limiting devices to protect the module and carrier board from differential and common mode spikes and noise on the E1 signal wires.

The IM2E1 has selectable input impedance matching for the HDB3 interface. The options are 75Ω, 120Ω or Hi Z. The impedance is selected individually for each of the four E1 lines via jumper settings. Figure 4-2 shows the location of the impedance selection jumpers and the RJ45 connectors on the module.

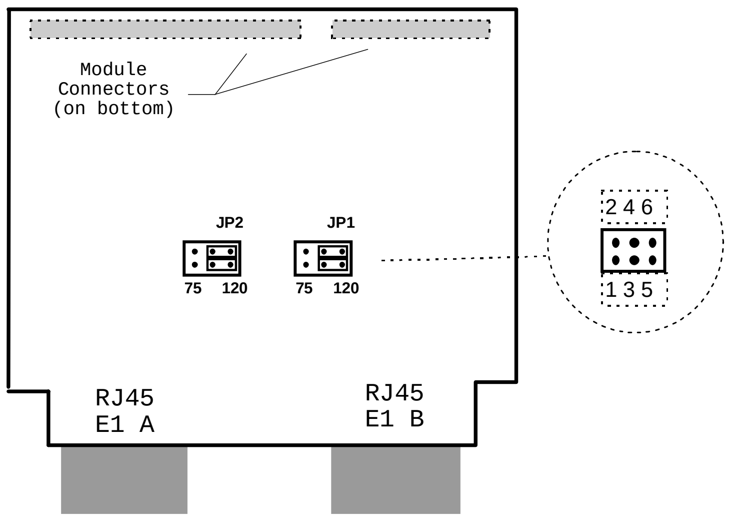
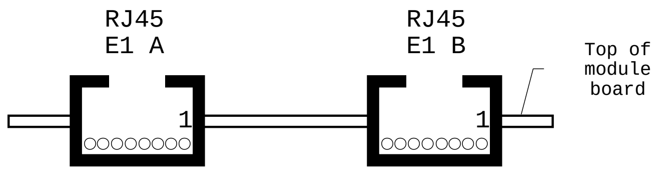


Figure 4-2: IM2E1 Line Impedance Selection

The HDB3 line input impedance is selected by positioning jumper pairs on JP1 and JP2 to connect the middle pins with the left (3-5 and 4-6) pins or right (3-1 and 4-2) pins. This will select the impedance, as indicated. Both jumpers in a pair should be positioned in the same way. The figure above shows the jumpers in their default (as shipped) positions, selecting 120Ω impedance. High input impedance (for using the IM2E1 to monitor an E1 line without additional loading) may be selected by positioning the jumpers to connect pins 5-6 and pins 1-2.

Figure 4-3 shows the signal connections for the RJ45 connectors. In this view, pin 1 is at the left.



Pin	Signal
1	Receive Ring
2	Receive Tip
4	Transmit Ring
5	Transmit Tip

Figure 4-3: IM2E1 RJ45 Pinout

5.2.2.4 LED Indicator

The IM2E1 uses the LED for its module location to indicate the following status:

RED	Module FPGAs are not configured
GREEN	Receive synchronization on one or both E1 lines

5.2.2.5 Specifications

The table below lists some of the significant performance characteristics of the IM2E1 module and its network interface components. For further details, consult the data sheets for the following components, some of which are available in Appendix A.

E1 Line Interface and Decoding and Framing:

Dallas Semiconductor	DS2153Q	CEPT Primary Rate Transceiver

E1 to TDM Interface:

Dallas Semiconductor	DS2175	T1/CEPT Elastic Store

Parameter	Min	Typ	Max	Units
HDB3 Interface				
Line Impedance (selectable) Coax Cable Twisted Pair		 75 120		Ω
Output transformer ratio		1:1.15		
Line Length			1500	meters
Output Pulse Amplitudes 75Ω coax 120Ω twisted		 2.37 3.00		V peak
Input Surge Protection (breakover)	27		36	V
Power Requirements		TBD		Amps @ 5V
MTBF		TBD		1000 hrs

Notes:

1. Characteristics apply under ambient temperature of 0 to +55 °C.

5.2.3 IM2E1 Software Support

5.2.3.1 V6M6 Host Application Library for IM2E1

The V6M6 Host Application Library includes several functions for control of the IM2E1 module, the DS2153Q framers and the TDM subsystem.

Functions operating on an IM2E1 module require a PCI_MOD structure pointer obtained by calling the **pciopen** function. The module type of the open module must be **PCI_IM2E1** (a macro defined in **<pciutil.h>**. Functions operating on the TDM subsystem require a PCI_TDM structure pointer obtained by calling the **pci_tdmopen** function.

Example:

```
#include <pciutil.h>

PCI_MOD *pci;
PCI_TDM *tdm;

pci = pciopen ("pci0a");
if  (pci == NULL) {
    perror ("pci0a");
    exit (1);
}

if  (pci->module_type != PCI_IM2E1) {
    printf ("IM2E1 not installed on pci0a\n");
    exit (1);
}

tdm = pci_tdmopen ("pci0");
if  (tdm == NULL) {
    perror ("pci0t");
    exit (1);
}
```

5.2.3.1.1 IM2E1 Functions Quick Reference

These are the C-callable functions of the V6M6 Host Application Library used for programming the IM2E1 module. Full descriptions begin on page 5-22.

IM2E1 and Framer Control

`int` **`im2e1_init`**`(`*`pci`*`)`	Initializes the IM2E1 module.
`int` **`im2e1_intenb`**`(`*`pci`*`,` *`intmask`*`)`	Loads the IM2E1 interrupt mask register to specify the routing of interrupts from the IM2E1.
`int` **`im2e1_intstat`**`(`*`pci`*`)`	Reads the IM2E1 interrupt status register.
`int` **`im2e1_clrfastclk`**`(`*`pci`*`)`	Clears the status of the comparison of the receive clock frequencies and TDMCLK.
`int` **`im2e1_outparams`**`(`*`pci`*`,` *`line`*`,` *`params`*`)`	Loads control registers in a E1 framer from an array of values.
`int` **`im2e1_lirst`**`(`*`pci`*`,` *`line`*`)`	Momentarily activates the LIRST of the selected E1 framer.
`u_long` **`im2e1_status`**`(`*`pci`*`,` *`line`*`,` *`statmask`*`)`	Reads the four status registers of the selected E1 framer.
`int` **`im2e1_xmtsig`**`(`*`pci`*`,` *`line`*`,` *`data`*`)`	Writes data to the signaling registers in a E1 framer chip.
`int` **`im2e1_rcvsig`**`(`*`pci`*`,` *`line`*`,` *`data`*`)`	Reads data from the signaling registers in a E1 framer chip.
`u_char` **`im2e1_inreg`**`(`*`pci`*`,` *`line`*`,` *`reg`*`)`	Reads a single register from a E1 framer chip.
`int` **`im2e1_outreg`**`(`*`pci`*`,` *`line`*`,` *`reg`*`,` *`data`*`)`	Writes an 8-bit value to a single register in a E1 framer.

TDM Clock, Framing and Synchronization

int **im2t1_xmtclk**(*pci*, *select*)	Selects the source of the transmit clocks for the T1 lines on an IM2T1 module.
int **im2t1_rcvclk**(*pci*, *select*)	Selects whether or not one of the T1 receive clocks on the IM2T1 is used to derive the TDM subsystem clock.
int **im2t1_tdmclkdiv**(*pci*, *clkdiv*)	Sets the ratio between T1 clocks and TDM subsystem clocks.
int **im2e1_slipinval**(p*ci*, *select*)	Enables or disables the slipped frame data invalidation mode.
int **pci_tdm_clksrc**(*tdm*, *clksrc*)	Selects the clock source for the TDM subsystem control logic.
int **pci_tdm_init**(*tdm*, *clk*, *bits*, *slots*, *frames*)	Initializes the parameters of the TDM subsystem controller.
int **pci_tdm_run**(*tdm*)	Starts the TDM subsystem timing logic.
int **im2e1_init_rsync**(*pci*, *line*)	Initializes synchronization of the incoming E1 frames to the TDM subsystem frames.

TDM Connection MAP Functions

int **pci_tdm_src_add**(*tdm*, *slot*, *bus*, *devtyp*, *devnum*)	Adds a device as a TDM data source for the specified time *slot* and TDM *bus*.
int **pci_tdm_src_del**(*tdm*, *slot*, *bus*)	Deletes a device as a TDM data source for the specified time *slot* and TDM *bus*.
int **pci_tdm_dst_add**(*tdm*, *slot*, *bus*, *devtyp*, *devnum*)	Adds a device as a TDM data destination for the specified time *slot* and TDM *bus*.
int **pci_tdm_dst_del**(*tdm*, *slot*, *devtyp*, *devnum*)	Deletes a device as a TDM data destination for the specified time *slot* and TDM *bus*.
int **pci_tdm_special_add**(*tdm*, *slot*, *devtyp*, *devnum*)	Adds reads of incoming E1 elastic store buffers which do not drive data onto a TDM bus.
int **pci_tdm_special_del**(*tdm*, *slot*, *devtyp*, *devnum*)	Deletes reads of incoming E1 elastic store buffers which do not drive data onto a TDM bus.

5.2.3.1.2 IM2E1 and Framer Control

The following library functions are used to control the IM2E1 module and its DS2153Q E1 framers.

function:	`int` **`im2e1_init`**`(`*`pci`*`)`
args:	`PCI_MOD *`*`pci`*

return:	`1    IM2E1 module initialized.`
	`0    error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or doesn't reference an IM2E1.`
	`EIO    error in communicating with the module or the framer chips.`

The **im2e1_init** function initializes the IM2E1 module. This includes setting up the module's PCI interface and clearing the DS2153Q registers. It is called from the **pciinit** program. This function should not be used by an application that expects the IM2E1 to be previously configured and currently in operation.

function: `int` **`im2e1_intenb`**`(`*`pci`*`, `*`intmask`*`)`

args:
```
PCI_MOD   *pci
int        intmask
```

return:
```
1    interrupts enabled.
0    error indicated.
```

errors:
```
EINVAL   pci is NULL or doesn't reference an IM2E1.
EIO      error in communicating with the module.
```

The **im2e1_intenb** function loads the IM2E1 interrupt mask register bits to specify the routing of interrupts (from the DS2153Q framers and the comparison of receive clocks with TDMCLK) to the PCI bus interrupts. Bits turned on in *`intmask`* enable interrupts as follows:

im2e1_intstat mask bits

intmask Bit	IM2E1 Interrupt	to	PCI Interrupt
0	Line 0 Int 1		A
1	Line 0 Int 2		A
2	Line 1 Int 1		A
3	Line 1 Int 2		A
4	Line 0 Int 1		B
5	Line 0 Int 2		B
6	Line 1 Int 1		B
7	Line 1 Int 2		B
8	RCLK > TDMCLK		A
9	RCLK > TDMCLK		B

function: `int` **`im2e1_intstat`**`(`*`pci`*`)`

args: `PCI_MOD *`*`pci`*

return: *`status`*
`-1   error indicated.`

errors:
```
EINVAL   pci is NULL or doesn’t reference an IM2E1.
EIO      error in communicating with the module.
```

The **im2e1_intstat** function reads the IM2E1 interrupt status register bits which indicate the state of the interrupts from the DS2153Q framers and the comparison of receive clocks with TDMCLK. Bits turned on in the returned *intstat* indicate the interrupts as follows:

im2e1_intstat returned status

status Bit	IM2E1 Interrupt
0	Line 0 Int 1
1	Line 0 Int 2
2	Line 1 Int 1
3	Line 1 Int 2
4	RCLK > TDMCLK

function: `int` **`im2e1_clrfastclk`**`(`*`pci`*`)`

args: `PCI_MOD *`*`pci`*

return: 1 fast RCLK detection status cleared
`0    error indicated.`

errors:
```
EINVAL   pci is NULL or doesn’t reference an IM2E1.
EIO      error in communicating with the module.
```

The **im2e1_clrfastclk** function clears the status of the comparison of the frequencies of the 2 receive clocks and TDMCLK. See Section 5.2.2.2.3 TDM Clock Rates and Frame Slips for further information.

```
function:  int im2e1_outparams(pci, line, params)

args:      PCI_MOD  *pci
           int       line
           u_char   *params

return:    1    Framer registers loaded.
           0    error indicated.

errors:    EINVAL   pci is NULL or doesn’t reference an IM2E1,
                    line is invalid, params is NULL.
           EIO      error in communicating with the module.
```

Most applications are likely to use a fixed set of operational settings for the DS2153Q chips. The **im2e1_outparams** function provides a way to load most of control registers in the DS2153Q with an array of values.

The *line* argument (0 or 1) specifies which DS2153Q is to be loaded. The *params* argument points to an array of 48 unsigned characters containing the values to be loaded into registers 0x00 through 0x2F of the selected DS2153Q with the following exceptions and conditions:

- Values that correspond to read-only registers are ignored.
- The two Test registers are always loaded with zeros.
- Values that correspond to the Status and Receive Information registers should contain 1’s for any status bits to be cleared.
- The bit corresponding to LIRST in CCR3 is ignored and replaced with a zero (use the **im2e1_lirst** function for line interface reset).

function: `int` **`im2e1_lirst`**`(`*`pci`*`, `*`line`*`)`

args:
```
PCI_MOD  *pci
int       line
```

return:
```
1    LIRST pulsed.
0    error indicated.
```

errors:
```
EINVAL   pci is NULL or doesn’t reference an IM2E1,
         line is invalid.
EIO      error in communicating with the module.
```

The **im2e1_lirst** function momentarily activates the LIRST bit in Common Control Register 3 of the selected DS2153Q. This may be required during operation if a problem with the incoming signal is detected. The *line* argument (0 or 1) specifies which DS2153Q is to be reset.

function: `u_long` **`im2e1_status`**`(`*`pci`*`, `*`line`*`, `*`statmask`*`)`

args:

```
PCI_MOD   *pci
int        line
u_long     statmask
```

return:

```
status      framer status (see table below).
0xffffffff  error indicated.
```

errors:

```
EINVAL   pci is NULL or doesn't reference an IM2E1,
         line is invalid.
EIO      error in communicating with the module.
```

The DS2153Q has four registers which provide status information to the host. These are Status Register 1, Status Register 2, the Receive Information register and the Synchronizer Status register. The first 3 registers require subsequent write accesses in order to clear the status information after it is read.

The **im2e1_status** function handles reading of all four status registers and clearing selectable active status bits. The *line* argument (0 or 1) specifies which DS2153Q is to be accessed.

The *statmask* argument specifies which bits of Status Registers 1 and 2 and the Receive Information register are to be read and cleared as shown in the table below. The *statmask* value does not affect the information from the Synchronizer Status register.

im2e1_status mask and returned bits

DS2153Q Register	*statmask* bits	*status* bits
Status Register 1	0:7	0:7
Status Register 2	8:15	8:15
Receive Information	23:16	23:16
Synchronizer Status	n/a	31:24

function: `int` **`im2e1_xmtsig`**`(`*`pci`*`, `*`line`*`, `*`data`*`)`
`int` **`im2e1_rcvsig`**`(`*`pci`*`, `*`line`*`, `*`data`*`)`

args:	`PCI_MOD`	`*`*`pci`*
	`int`	*`line`*
	`u_char`	`*`*`data`*

return:	`1`	*`data`* `transferred to / from DS2153Q.`
	`0`	`error indicated.`

errors:	`EINVAL`	*`pci`* `is NULL or doesn't reference an IM2E1,` *`line`* `is invalid or` *`data`* `is NULL.`
	`EIO`	`error in communicating with the module.`

The DS2153Q framers automatically extract and insert Channel Associated Signaling bits to and from registers. This operation is described in section 7 of the DS2153Q data sheet. The IM2E1 function library contains functions for reading and writing the signaling registers.

The Transmit Signaling registers (0x40 through 0x4F) are loaded using the **im2e1_xmtsig** function. The Receive Signaling registers (0x30 through 0x3F) are read using the **im2e1_rcvsig** function.

For both functions the *line* argument specifies which DS2153Q is to be accessed. The *data* argument is a pointer to an array of 16 unsigned characters that contains the data to be written to or read from the framer.

function: `u_char` **`im2e1_inreg`**`(`*`pci`*`, `*`line`*`, `*`reg`*`)`

```
args:     PCI_MOD  *pci
          int       line
          int       reg

return:   data   read from DS2153Q register.
          0      error indicated.

errors:   EINVAL    pci is NULL or doesn't reference an IM2E1,
                    or line or reg invalid.
          EIO       error in communicating with the module.
```

The **im2e1_inreg** function may be used to read a single register from the DS2153Q specified by the *line* argument.

function: `int` **`im2e1_outreg`**`(`*`pci`*`, `*`line`*`, `*`reg`*`, `*`data`*`)`

```
args:     PCI_MOD  *pci
          int       line
          int       reg
          u_char    data

return:   1    data transferred to DS2153Q register.
          0    error indicated.

errors:   EINVAL    pci is NULL or doesn't reference an IM2E1,
                    or line or reg invalid.
          EIO       error in communicating with the module.
```

The **im2e1_outreg** function may be used to write an 8-bit value, *data*, to a single register in the DS2153Q specified by the *line* argument.

For both the **im2e1_inreg** and **im2e1_outreg** functions the *reg* argument may be the offset of the register in the DS2153Q chip or one of the following macros, defined in <**pciutil.h**>:

IM2E1REG_BCVC1	BVP or Code Violation Count 1
IM2E1REG_BCVC2	BVP or Code Violation Count 2
IM2E1REG_CRCE1	CRC4 Error Count 1 , FAS Error Count 1
IM2E1REG_CRCE2	CRC4 Error Count 2
IM2E1REG_EBIT1	E-Bit Count 1 , FAS Error Count 2
IM2E1REG_EBIT2	E-Bit Count 2
IM2E1REG_STAT1	Status Register 1
IM2E1REG_STAT2	Status Register 2
IM2E1REG_RECVI	Receive Information Register
IM2E1REG_RCTL1	Receive Control Register 1
IM2E1REG_RCTL2	Receive Control Register 2
IM2E1REG_TCTL1	Transmit Control Register 1
IM2E1REG_TCTL2	Transmit Control Register 2
IM2E1REG_CCTL1	Common Control Register 1
IM2E1REG_CCTL2	Common Control Register 2
IM2E1REG_CCTL3	Common Control Register 3
IM2E1REG_MASK1	Interrupt Mask 1
IM2E1REG_MASK2	Interrupt Mask 2
IM2E1REG_LICTL	Line Interface Control Register
IM2E1REG_SSTAT	Synchronizer Status Register
IM2E1REG_RCA F	Receive Align Frame
IM2E1REG_RCNAF	Receive Non-Align Frame
IM2E1REG_XTAF	Transmit Align Frame
IM2E1REG_XTNAF	Transmit Non-Align Frame
IM2E1REG_XBLK1	Transmit Channel Blocking 1
IM2E1REG_XBLK2	Transmit Channel Blocking 2
IM2E1REG_XBLK3	Transmit Channel Blocking 3
IM2E1REG_XBLK4	Transmit Channel Blocking 4
IM2E1REG_XIDL1	Transmit Idle Blocking 1
IM2E1REG_XIDL2	Transmit Idle Blocking 2
IM2E1REG_XIDL3	Transmit Idle Blocking 3
IM2E1REG_XIDL4	Transmit Idle Blocking 4
IM2E1REG_XIDLE	Transmit Idle Definition
IM2E1REG_RBLK1	Receive Channel Blocking 1
IM2E1REG_RBLK2	Receive Channel Blocking 2
IM2E1REG_RBLK3	Receive Channel Blocking 3
IM2E1REG_RBLK4	Receive Channel Blocking 4
IM2E1REG_RSIG	1st of 16 Receive Signaling Registers
IM2E1REG_TSIG	1st of 16 Transmit Signaling Registers

5.2.3.1.3 TDM Clock, Framing and Synchronization

The functions listed in this section control E1 clocks on an IM2E1 and set up the timing of the interface between IM2E1 modules and the TDM subsystem.

function: `int` **`im2e1_xmtclk`**`(`*`pci`*`, `*`select`*`)`

args:

```
PCI_MOD   *pci
int        select
```

return:

```
1    E1 transmit clocks selected.
0    error indicated.
```

errors:

```
EINVAL   pci is NULL or doesn’t reference an IM2E1.
EIO      error in communicating with the module.
```

The **im2e1_xmtclk** function selects the E1 transmit clock for both E1 lines on an IM2E1 module. The *select* argument encodes the selection as shown below. When deriving E1 transmit clock from TDMCLK, the IM2E1 generates a 2.048 MHz clock, dividing the TDMCLK rate as specified with the **im2e1_tdmclkdiv** function.

im2e1_xmtclk select encoding

Bits	controls	code	selection
3:2	TCLK A	00	RCLK A
		01	RCLK B
		10	TDM Derived
1:0	TCLK B	00	RCLK-B
		01	RCLK A
		10	TDM-Derived

function:	`int` **`im2e1_rcvclk`**`(`*`pci`*`, `*`select`*`)`

args:	`PCI_MOD *`*`pci`*
	 `int` *`select`*

return:	`1    RCLK to TDM clock selected.`
	 `0    error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or doesn’t reference an IM2E1.`
	 `EIO    error in communicating with the module.`

The **im2e1_rcvclk** function selects whether or not one of the E1 receive clocks on the IM2E1 is used to derive the TDM subsystem clock. The actual TDM clock rate can be a multiple of the selected E1 RCLK as set with the **im2e1_tdmclkdiv** function. In addition, the **pci_tdm_clksrc** function must be used to select the clock source for the TDM system control, itself.

When receiving E1 data, it usually desirable to derive the TDM clock from the received E1 clock to ensure consistent capture of the incoming data. However, if more than one IM2E1 module is used or some other clock source must be used as the TDM master clock the module may be set to not drive the TDM clock.

When the two incoming E1 lines may be at differenct clock frequencies it is desirable to have the IM2E1 automatically select the faster of the two receive clocks as the source for deriving TDMCLK. See Section 5.2.2.2.3 TDM Clock Rates and Frame Slips for more information.

im2e1_rcvclk settings	
select	RCLK used for TDM Clk
0	none
1	RCLK A
2	RCLK B
3	Faster of RCLK A and B

function: `int` **`im2e1_tdmclkdiv`**`(`*`pci`*`, `*`clkdiv`*`)`

args:

```
PCI_MOD   *pci
int        clkdiv
```

return:

```
1    Clock divider set.
0    error indicated.
```

errors:

```
EINVAL   pci is NULL or doesn't reference an IM2E1,
         or clkdiv is invalid.
EIO      error in communicating with the module.
```

The **im2e1_tdmclkdiv** function sets the ratio between E1 clocks and TDM subsystem clocks. For E1 transmit clocks derived from the TDM clock, it specifies the ratio between the TDM clock rate and the E1 TCLK rate. For TDM clock derived from the E1 receive clock, it determines the ratio between the selected E1 RCLK and the TDM clock.

The clock ratio is always based on an E1 clock of 2.048 MHz. Therefore the value of clkdiv actually specifies the TDM clock rate as:

im2e1_tdmclkdiv settings

clkdiv	TDM Clock Rate
0	2.048 MHz
1	4.096 MHz
2	8.192 MHz

```
function:   int im2e1_slipinval(pci, select)

args:       PCI_MOD   *pci
            int        select

return:     1    Slip invalidation mode set.
            0    error indicated.

errors:     EINVAL   pci is NULL or doesn’t reference an IM2E1,
                     or the module is prior to Revision 2
            EIO      error in communicating with the module.
```

The **im2e1_slipinval** function enables or disables the slipped frame data invalidation mode described in section 4.2.2.3. The two low bits of the *select* argument determine whether this mode is enabled or disabled for each of the two E1 lines. A 1 enables and a 0 disables the slipped frame data invalidation mode.

There is no practical reason why one line would have this mode enabled and the other would not. Therefore, normal values for *select* are 0 for no slip invalidation mode or 3 for enabling slip invalidation mode. The E1 framer chips must also be programmed to enable framer interrupt 1 on elastic store slip events as described in section 4.2.2.3.

This feature is supported on IM2E1 modules of Revision 2 or higher.

function: `int` **`pci_tdm_clksrc`**`(`*`tdm`*`, `*`clksrc`*`)`

args:
```
PCI_TDM  *tdm
int       clksrc
```

return:
```
1    Clock divider set.
0    error indicated.
```

errors: `EINVAL` *`tdm`* `is NULL or` *`clksrc`* `is invalid.`

The **pci_tdm_clksrc** function selects the clock source for the TDM subsystem control logic. The values for *clksrc* defined in <pciutil.h> are:

TDM_BASECLK	selects the clock generated within the controller or from the TDM expansion port.
TDM_MEZZCLK	selects a clock generated from a module on the board.

The application must be careful not to enable more than one clock driver at a time. When TDM_BASECLK is used, none of the modules should be set to drive TDM clock. When TDM_MEZZCLK is used, one and only one module must generate the TDM clock.

function: `int` **`pci_tdm_init`**`(`*`tdm, clk, bits, slots, frames`*`)`

args:

```
PCI_TDM   *tdm
int        clk, bits, slots, frames
```

return:

```
1    TDM controller initialized.
0    error indicated.
```

errors: `EINVAL` *`tdm`* `is NULL or other arguments are invalid.`

The **pci_tdm_init** function initializes the parameters of the TDM subsystem controller.

With the IM2E1, use the following values for the arguments:

clk	When an IM2E1 (or any module or TDM expansion port) is configured to generate the TDM subsystem clock this argument must be 0. To generate TDM clocks from the TDM subsystem controller a value of 2, 4, or 8 should be used. See the table below.
bits	Bits per time slot should always be set to 8.
slots	This value may be set to 32, 64 or 128, depending on the clock rate being used. The combination of clock rate and frame size must result in a 125 microsecond frame.
frames	Frames per multiframe should be set to 16.

TDM clock and slots values for IM2E1

TDM clock rate	TDM Slots per frame
2.048 MHz	32
4.096 MHz	64
8.192 MHz	128

function: `int` **`pci_tdm_run`**`(`*`tdm`*`)`

args: `PCI_TDM  *`*`tdm`*

return:
```
1    TDM subsystem started.
0    error indicated.
```

errors:
```
EINVAL  tdm is NULL or other arguments are invalid.
EIO     failed to update the TDM MAP RAM, usually
        due to the absence of TDM clocks.
```

The **pci_tdm_run** function starts the TDM subsystem timing logic. The TDM map is updated, if necessary, and the TDM controller will begin generating the sync pulses as specified by the **pci_tdm_init** function.

function: `int` **`im2e1_init_rsync`**`(`*`pci`*`, `*`line`*`)`

args:
```
PCI_MOD  *pci
int       line
```

return:
```
1    E1 elastic store synchronized.
0    error indicated.
```

errors:
```
EINVAL   pci is NULL or doesn’t reference an IM2E1,
         or line is invalid.
EIO      error in communicating with the module.
```

The **im2e1_init_rsync** function initializes synchronization of the incoming E1 frames to the TDM subsystem frames. It must be called after the source of TDM clocks is established and the TDM subsystem is running.

It is necessary to call **im2e1_init_rsync**, for each E1 line to be used for incoming E1 data, whenever the TDM subsystem is started or restarted.

5.2.3.1.4 TDM Connection MAP Functions

The following library functions provide the means for setting up TDM data transfer connections between IM2E1 modules and other modules.

Reading an incoming E1 data slot or status involves making one of the IM2E1 lines a TDM source during a TDM time slot. Sending outgoing E1 data involves making one of the IM2E1 lines a TDM destination during a TDM time slot.

All 32 data slots of each E1 line must be read and written during a TDM frame. One or both lines of an IM2E1 may be read or written during the same TDM time slot.

function:

```
int pci_tdm_src_add(tdm, slot, bus, devtyp, devnum)
int pci_tdm_src_del(tdm, slot, bus)
```

args:

```
PCI_TDM   *tdm
int        slot
int        bus
int        devtyp
int        devnum
```

return:

```
1    Buffered TDM Map modified.
0    error indicated.
```

errors:

```
EINVAL  tdm is NULL or does not reference the TDM
        subsystem of a V6M6, or other
        arguments are invalid.
ENODEV  the specified devtyp (an IM2E1) is not
        installed at the module location specified
        in devnum.
```

These two functions add or delete a device as a TDM data source for the specified time *slot* and TDM *bus*. Only one device on a board may be the source for a TDM bus during a time slot so it is not necessary to specify the device type or unit number for the delete function.

The following arguments values are used for the IM2E1 module:

slot	Values between 1 and the number of slots configured for the TDM subsystem frame (see **pci_tdm_init**).
bus	One of the macros defined in <**pciutil.h**>, **TDM_BUSA**, **TDM_BUSB**, **TDM_BUSC** and **TDM_BUSD**, corresponding to the four TDM busses.
devtyp	The device type for the IM2E1 the device type is **TDM_DEVIM2E1**. This is a macro defined in <**pciutil.h**>.
devnum	The device unit number encodes the module location and which E1 line for the module as shown in the table below.

IM2E1 TDM device unit number

Bit 4 = E1 Line	Bits 3:0 = Module
0 = Line A 1 = Line B	0 = A 1 = B 2 = C 3 = D 5 = F

function: `int` **`pci_tdm_dst_add`**`(`*`tdm, slot, bus, devtyp, devnum`*`)`
`int` **`pci_tdm_dst_del`**`(`*`tdm, slot, devtyp, devnum`*`)`

args:

```
PCI_TDM   *tdm
int        slot
int        bus
int        devtyp
int        devnum
```

return:

```
1    Buffered TDM Map modified.
0    error indicated.
```

errors:

```
EINVAL  tdm is NULL or does not reference the TDM
        subsystem of a V6M6, or other
        arguments are invalid.
ENODEV  the specified devtyp (an IM2E1) is not
        installed at the module location specified
        in devnum.
```

These two functions add or delete a device as a TDM data destination for the specified time *slot* and TDM *bus*. A device may be a destination for only one TDM bus during a time slot so it is not necessary to specify the TDM bus for the delete function. The use of the arguments is the same as the **pci_tdm_src_add** function.

function: int **pci_tdm_special_add**(*tdm, slot, devtyp, devnum*)
int **pci_tdm_special_del**(*tdm, slot, devtyp, devnum*)

args:

PCI_TDM	**tdm*
int	*slot*
int	*devtyp*
int	*devnum*

return:

1	Buffered TDM Map modified.
0	error indicated.

errors:

EINVAL	*tdm* is NULL or does not reference the TDM subsystem of a V6M6, or other arguments are invalid.
ENODEV	the specified *devtyp* (an IM2E1) is not installed at the module location specified in *devnum*.

The TDM **special** functions are used to add or delete reads of incoming E1 elastic store buffers which do not drive data onto a TDM bus. These may be necessary when not all of the incoming data is being used and the TDM bandwidth is required for other data transfers. It is still necessary to specify unloading of the elastic stores to maintain slot and frame alignment. The use of the arguments is the same as the **pci_tdm_src_add** function.

5.2.3.2 MIPS Application Library

A library of functions provides access to the DS2153Q E1 framer registers and the IM2E1 interrupt mask and status by MIPS processor module applications run under the WMI MIPS Kernel.

The functions are ported from the host application library and have the same names, arguments and return values. The IM2E1 functions implemented for MIPS applications are:

```
im2e1_intenb
im2e1_intstat
im2e1_outparams
im2e1_lirst
im2e1_status
im2e1_xmtsig
im2e1_rcvsig
im2e1_inreg
im2e1_outreg
```

5.2.3.3 Utility Programs

The following programs, in **$CAC/bin**, are provided for the V6M6 PCI carrier boards. They are useful for initializing and maintaining the IM2T1 module. Detailed descriptions of these programs are available in Appendix B or online in UNIX man file format provided that you followed the setup procedures in Section 2.3.2 Programmer and User Setup.

pciinit	initializes the base board and modules, instructs the on-board microprocessor to configure FPGAs on the module. It also sets up PCI address mappings.
pciinfo	examines the EEROMs on the base board and modules and the flash memory. It then displays all pertinent version and serial number information.
pciflashup	compares flash object (FPGA configurations and microprocessor program) versions in the flash memory with those residing in host files. Depending on the command line options it will either display the version numbers or update the flash memory with any new versions on disk.

5.2.3.4 Diagnostic programs

The following diagnostic programs are being developed for the IM2E1 module. These programs will be distributed in the directory, **$CAC/pci/diag.**

pcie1loop	Runs a loop-back test with a single IM2E1 sending data to itself. The data is generated from, received and tested by a processor module on the same baseboard. The E1 signal may be looped back externally or internal to the DS2153Q framer.
pcie1xmt	Transmits a known E1 data pattern generated from a processor module on the same baseboard.
pcie1rcv	Receives E1 data and compares with the known pattern used by **pcie1xmt**.

The **pcie1loop** program is one of the components of the test and burn-in program, **pciburn**.

5.3 IM2T1

The IM2T1 provides two T1 interfaces for the V6M6 PCI module carrier boards. Electrical interfaces for AMI/B8ZS networks are provided via RJ45 jacks, one for each T1 line. T1 frame encoding and decoding is performed on each T1 line by a Dallas Semiconductor DS2151Q T1 Transceiver. Elastic frame buffers (DS2175) interface the outgoing T1 streams to the TDM subsystem. Incoming T1 data is passed through the internal elastic store of the DS2151Q framer chip.

Information in this manual is divided into the following sections:

5.3.1 IM2T1 Quick Tour

Figure 5-1 is a block diagram of the IM2T1 showing the major components and signal paths. The primary components are the two Dallas Semiconductor DS2151Q T1/B8ZS Transceivers. Please refer to the DS2151Q data sheet (provided in Appendix A: "Component Data Sheets") for details on their operation.

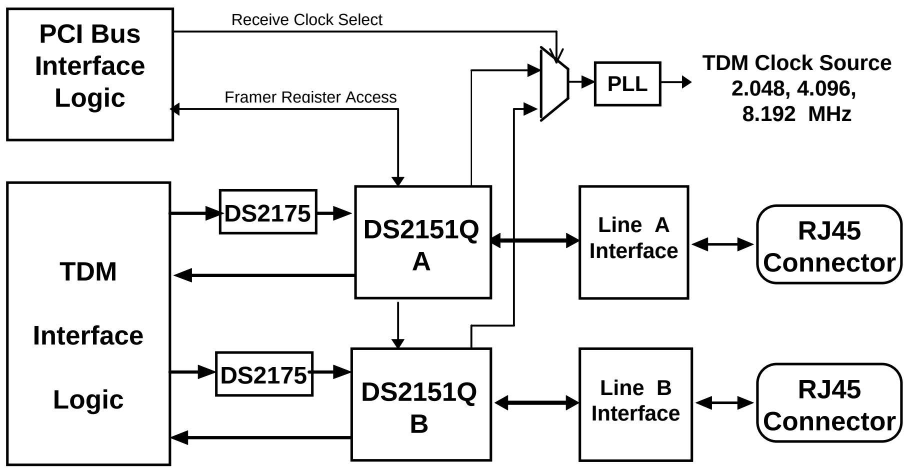


Fig. 5-1: IM2T1 Module Block Diagram

The two T1 lines, designated A and B, connect to the board through RJ45 connectors. The T1 signals pass through the line interface components to and from the DS2151Q framers. The input impedance for each line is selectable to be 100Ω or high impedance (see Section 5.3.2.3 Electrical Line Interface). The output impedance is controlled by the line build-out select in the DS2151Q's LICR registers (see table 12-2 in the Dallas DS2151Q data sheet). The output pulse transformer ratios are 1:1.15.

The DS2151Q T1 components provide clock and data recovery for the incoming T1 lines and T1 framing and pulse generation for the outgoing T1 lines. Access to the DS2151Q control, status and data registers is provided through the PCI interface.

Incoming and outgoing T1 data is accessed by other PCI modules via the TDM subsystem of the V6M6 board. Incoming multiframe and loss of signal status is also available via the TDM subsystem. The TDM interface is designed to be operated with 8-bit TDM time slots and TDM frames of 32 slots / 2.048 MBPS, 64 slots / 4.096 MPBS or 128 slots (8.192 MPBS).

A phase-locked loop may be used to synchronize the TDM subsystem clock with one of the incoming receive clocks.

FIFO storage between incoming T1 data and the TDM subsystem is provided by the DS2151Q framers. The two Dallas Semiconductor 2175 elastic store buffers provide a FIFO interface for outgoing T1 data.

IM2T1 modules may be installed on V6M6 boards at module locations A and C for front panel access to the RJ45 connectors. A future option will allow IM2T1 modules to be installed at module locations B, D or F with its I/O connections made via the VME P2 connector.

5.3.2 IM2T1 Hardware Description

The IM2T1 and IM2E1 use the same FPGA configuration data stored in the Flash ROM on the V6M6. The flash object name is **imtelco**. The PCI interface logic uses the same generic, configurable design as other IM modules for the V6M6. Configuration of this logic is performed by the host system using the V6M6 board initialization program, **pciinit**.

5.3.2.1 Module Control and T1 Framer Registers

The registers for control of the module and the T1 framers are accessed in PCI bus configuration space and i/o space. The table below shows the base addresses for the module locations that may be used to carry IM2T1 modules. The configuration space base addresses are determined by physical hardware connections. I/O space base addresses are determined by convention and are hard coded in various software modules. The values are loaded into the IM2T1 registers by the **pciinit** program.

PCI Base Addresses for IM2T1 Modules

PCI Module	Config Space Base Address	I/O Space Base Address
A	0x01000000	0x30000000
B	0x00800000	0x31000000
C	0x00400000	0x32000000
D	0x00200000	0x33000000
F	0x00080000	0x35000000

The following two tables list the IM2T1 registers. The address or address range shown are the offsets from the base address of the module.

IM2T1 Configuration Space Registers

Register Name	Address	Acc	Function / Description
PCI Stat/Cmd	0x04	rw	See PCI Spec for details
Mod Base	0x10	wo	PCI base address of module
Dev Cmd	0x40	rw	Generic I/O module controls and settings
Interrupt Stat/Mask	0x44	rw	I/O Interrupt stat and mask
Interrupt Ctl	0x48	rw	Controls interrupt polarity

IM2T1 I/O Space Registers

Register Name	Address Range	Acc	Function / Description
T1 Framer 0	0x000 to 0x3FC	rw	Line A DS2151Q Registers (see Dallas Semi. data sheet)
T1 Framer 1	0x400 to 0x7FC	rw	Line B DS2151Q Registers (see Dallas Semi. data sheet)
TDM ID	0x800	rw	Stores module location ID
Clock Ctl	0x804	rw	Controls clock sources for TDM Subsystem and T1 Transmit
RSYNC Ctl	0x808	rw	Controls TDM SYNC drive to RSYNC to DS2153Q Framer
Slip Ctl	0x80C	rw	Controls slipped frame action and clears Fast RCLK Status

PCI Stat / Cmd Register:

This register is used for configuring and obtaining status of the module’s interface to the PCI bus. The control bits are all either hardwired or programmed by the **pciinit** program.   The bits set by **pciinit** are:

```
Bit 0 = 1: I/O Space Enabled
Bit 6 = 1: PCI Parity Error Response Enabled
```

Two status bits, listed below, are available for PCI error checking. These are cleared by writing a 1.

```
Bit 30 - Module signaled a System Error
Bit 31 - Module detected a PCI Parity Error
```

Mod Base Register:

This register stores the module’s I/O space base address on the PCI bus. It is programmed by the **pciinit** program and a copy of its contents is stored in the host’s device driver and made available to host applications.

Dev Cmd Register:

This register is used to control low-level module functions.

```
Bits 31:28 - Timing for the module’s internal bus
Bit  23    - Module Reset (low active)
```

The timing bits are programmed by the **pciinit** program to meet timing requirements of the T1 Framer devices on the module. Module Reset is only used to center the DS2175 Elastic Storage FIFOs.

Interrupt Mask / Stat Register:

This register controls the interrupt enable and reads interrupt status for the module.   The IM2T1 supports the 2 interrupt sources from both of the DS2151Q Framers and a special interrupt indicating that one of the E1 line's RCLK is running at a faster rate than TDMCLK.

The interrupt mask bits allow any of the interrupt sources to be the source of PCI interrupt A or B. For each bit, a 1 passes the interrupt and a 0 masks it.

```
Bit 0 - Framer 0 Int 1 to PCI interrupt A
Bit 1 - Framer 0 Int 2 to PCI interrupt A
Bit 2 - Framer 1 Int 1 to PCI interrupt A
Bit 3 - Framer 1 Int 2 to PCI interrupt A
Bit 4 - Framer 0 Int 1 to PCI interrupt B
Bit 5 - Framer 0 Int 2 to PCI interrupt B
Bit 6 - Framer 1 Int 1 to PCI interrupt B
Bit 7 - Framer 1 Int 2 to PCI interrupt B
Bit 8 - RCLK > TDMCLK to PCI interrupt A
Bit 9 - RCLK > TDMCLK to PCI interrupt B
```

The interrupt status bits are valid regardless of the interrupt mask setting. A 1 indicates that the interrupt source is active.

```
Bit 24 - Framer 0 Int 1
Bit 25 - Framer 0 Int 2
Bit 26 - Framer 1 Int 1
Bit 27 - Framer 1 Int 2
Bit 28 - RCLK > TDMCLK
```

Interrupt Ctl Register:

This register is written by the **pciinit** program to match the polarity of the T1 Framer interrupts to the polarity expected on the PCI interrupt lines.

T1 Framer Registers:

The various registers of the DS2151Q Framers are accessed via PCI I/O space. Details of the registers' functions and contents are available in the Dallas Semiconductor DS2151Q data sheet.

Framer 0 is accessed starting at offset 0x000 from the module's I/O base and Framer 1 is accessed starting at offset 0x400 from the module's base. Each of the 8-bit registers are accessed on 32-bit boundaries with the register data in the LSB of the word. They may be accessed as 32-bit words (with only the lower 8-bits containing useful data) or as 8-bit words. In either case, the DS2151Q register offset, as specified in the data sheet, must be shifted left by two bits then added to the module base and framer base addresses. For byte access, the address must be further offset by 3.

32-bit access:

```
PCIaddr = Mod_IObase + Framer_base + (Reg_offset << 2)
```

8-bit access:

```
PCIaddr = Mod_IObase + Framer_base + (Reg_offset << 2) + 3
```

32-bit example: LICR register of Framer 1 on Module C

```
PCIaddr = 0x32000000 + 0x400 + (0x18 << 2) = 0x32000460
```

Certain control register bits must be set to the following values in order for the IM2T1 to operate properly and maintain synchronization with the TDM subsystem. These are shown in the table below. Other framer controls and parameters are application dependent.

Register	Offset	Bit	Val	Comment
RCR2	0x2C	3	0	RSYNC pin is an output
RCR2	0x2C	4	1	RSYNC pin in multiframe mode
CCR1	0x37	3	0	SCLK is 1.544 MHz
CCR1	0x37	2	1	Rcv Elastic Store Enabled
TCR2	0x36	2	0	TSYNC pin is an input
CCR3	0x30	5	0	RLOS/LOTC pin indicates RLOS
CCR1	0x37	7	0	Xmt Elastic Store Bypassed

TDM ID Register:

This register stores the module ID (location) of the module and is used to identify the module for TDM data transfers. Bits 2:0 contain the ID value as programmed by the **pciinit** program.

Clock Ctl Register:

This register is used to control the relationship between the TDM subsystem clock and the T1 receive and transmit clocks.

The clock divider bits determine the relationship between the T1 clock rate and the TDM clock rate.

```
Bits 1:0 - Clock Divider
            00 : TDM clock = T1 Clock * 1 (2.048 MHz)
            01 : TDM clock = T1 Clock * 2 (4.096 MHz)
            10 : TDM clock = T1 Clock * 4 (8.192 MHz)
```

The TDM clock may be derived from either of the two Framer's RCLK or not from the module at all. The IM2T1 (configured with version 13 or higher of the **imtelco** FPGA configuration) includes logic to detect and select the faster of the two received clocks. This is useful in applications where the incoming T1 lines may have different time bases. The value of bits 1:0 multiply the selected RCLK to derive TDM clock.

```
Bits 3:2 - TDM Clock select bits
            00 : TDM clock not sourced by the module
            01 : TDM clock derived from RCLK A
            10 : TDM clock derived from RCLK B
            11 : TDM clock derived from the faster of RCLK A
                 and RCLK B
```

T1 transmit clock sources may be derived from either Framer's RCLK or from the TDM subsystem clock. When TDM-derived TCLK is selected, the value of bits 1:0 divide TDM clock to derive TCLK.

```
Bits 5:4 - TLCK B Select
            00 : TCLK B = RCLK B
            01 : TCLK B = RCLK A
            10 : TCLK B derived from TDM Clock

Bits 7:6 - TLCK A Select
            00 : TCLK A = RCLK A
            01 : TCLK A = RCLK B
            10 : TCLK A derived from TDM Clock
```

RSYNC Ctl Register:

This register controls whether or not the TDM SYNC signal is driven to the T1 framer's RSYN pin. It is used by the **im2t1_init_rsync** function to initialize synchronization between the T1 framer and the TDM subsystem.

SLIP Ctl Register:

This register enables or disables the IM2T1 logic from invalidating TDM bus data for slipped frames. The **im2t1_slipinval** library function accesses this register to control this mode. Bits 0 and 1 of the register determine whether this mode is enabled for each T1 line:

```
Bit 0 -
      0 : Slip invalidation mode disabled for line 0
      1 : Slip invalidation mode enabled for line 0
Bit 1 -
      0 : Slip invalidation mode disabled for line 1
      1 : Slip invalidation mode enabled for line 1
```

5.3.2.2 TDM Interface

The primary method for transferring incoming and outgoing T1 data is via the TDM subsystem. The serial data input and output of the T1 framers are connected to the TDM data busses through elastic storage buffers, allowing the TDM clock rate to be multiple of the T1 data rate.

Data is transferred between a TDM data bus and an elastic storage buffer on a slot by slot basis based on commands stored in the TDM subsystem MAP RAM. The MAP is built and loaded from the host system using the PCI library TDM functions described in Section 5.3.3.1.4. The control words, as stored in the TDM Map RAM, are described below.

The TDM interface logic also decodes the TDM control words generated from the TDM Map RAM on the base board. There are 8 control words for each time slot which determine the connections between the DSPs and the TDM subsystem busses. The TDM map is programmed using C functions provided in the V6M6 Host Application Library (see Section 5.3.3.1 V6M6 Host Application Library).

5.3.2.2.1 TDM Map Control Words

The TDM control words, as stored in the TDM Map RAM, are described below:

IM2T1 TDM MAP Control Words		
Control Word	Bits 7:4	Bits 3:0
0	TDMA-SRC	TDMB-SRC
1	TDMC-SRC	TDMD-SRC
2	MOD4-CTL	MOD5-CTL
3	MOD2-CTL	MOD3-CTL
4	MOD0-CTL	MOD1-CTL
5	MOD4-DST	MOD5-DST
6	MOD2-DST	MOD3-DST
7	MOD0-DST	MOD1-DST

<u>TDM SRC</u> The 4-bit code in each of the TDM Source selectors determines the device to source the TDM Bus.

```
0x0   Line A on Module A sources the TDM Bus
0x1   Line A on Module B sources the TDM Bus
0x2   Line A on Module C sources the TDM Bus
0x3   Line A on Module D sources the TDM Bus
0x4   Line A on Module E sources the TDM Bus
0x5   Line A on Module F sources the TDM Bus
0x6   Line B on Module A sources the TDM Bus
0x7   Line B on Module B sources the TDM Bus
0x8   Line B on Module C sources the TDM Bus
0x9   Line B on Module D sources the TDM Bus
0xA   Line B on Module E sources the TDM Bus
0xB   Line B on Module F sources the TDM Bus
0xF   No source for the TDM Bus
```

MOD CTL The 2-bit codes in these words control loading and unloading of the T1 data elastic buffers on the module.

Bits 1:0	Loading of outgoing buffers
11	No load
10	Load Line A buffer
01	Load Line B buffer
00	Load Line A and Line B buffers

Bits 3:2	Unloading of incoming buffers
11	No unload
10	Unload Line A buffer
01	Unload Line B buffer
00	Unload Line A and Line B buffers

MOD DST The 2-bit codes in these words select which TDM Bus is used for getting data to the outgoing T1 data elastic buffers on the module.

Bits 1:0	Bus for Data to Line A
00	TDM Bus A
01	TDM Bus B
10	TDM Bus C
11	TDM Bus D

Bits 3:2	Bus for Data to Line B
00	TDM Bus A
01	TDM Bus B
10	TDM Bus C
11	TDM Bus D

5.3.2.2.2 TDM Framing

Exactly 25 time slots per TDM frame must be transferred to and from each of the T1 line's elastic buffers. The first TDM slot will contain FDL and status data. The remaining 24 time slots contain data for the 24 T1 channels. It is possible to read and write the buffers for both lines during the same time slot. It is also possible to unload a byte from an incoming buffer without driving the data onto a TDM bus. This allows unused data to be purged without using up TDM bandwidth.

TDM frames must always be set for 125 microseconds. They may be configured as 32 slots at 2.048 MHz, 64 slots at 4.096 MHz or 128 slots at 8.192 MHz. TDM framing parameters are configured using host application functions described in section 5.3.1.2. When frames of 64 or 128 slots are used, it is recommended that the T1 slots be spread evenly across the available TDM time slots.

The T1 data frames, incoming and outgoing, are aligned with the TDM frames. Outgoing multiframes are synchronized with the multiframe sync generated by the TDM subsystem. Incoming multiframes are synchronized with the receive multiframe sync for each line.

The first incoming slot of each frame contains framer status information and the FDL bit, as described below. The first outgoing slot of each frame is used to send an FDL data bit. The remaining bits for the first outgoing slot should be zero.

Contents of first incoming slot from each T1 line:

Bit 7	MSB First						Bit0
FDL	TDM MF	-	-	RLOS B	RLOS A	RSYNC B	RSYNC A

FDL	incoming and outgoing FDL bit.
TDM MF	State of the MultiFrame Sync generated by the TDM Subsystem - indicates 1st frame of a transmit multiframe.
RLOS A/B	Sampled output of framers' RLOS pin - indicates loss of receive sync.
RSYNC A/B	Sampled output of framers' RSYNC pin - indicates 1st frame of a receive multiframe.

5.3.2.2.3 TDM Clock Rates and Frame Slips

Most applications have the TDM subsystem clock, TDMCLK, derived from the clock received from an incoming T1 line. The T1 transmit clock, in turn, is usually derived from TDMCLK. The library functions, **im2t1_rcvclk**, **im2t1_xmtclk** and **im2t1_tdmclkdiv**, control the TDM and T1 clock timing. These functions are described in 5.3.3 IM2T1 Software Support.

For the majority of systems, multiple incoming T1 lines will be running at the same clock rate, being derived from a common timing source. In a few situations, however, the timing of multiple incoming T1 lines may be derived from different time bases; their clocks may be running at slightly different frequencies. Depending on which receive clock is faster, data from one of the incoming lines could be either lost or repeated as the elastic storage buffer in the framer chip under-flows or over-flows. This data loss or repetition occurs at a frame boundary; an entire frame is either lost or repeated at a time.

The IM2T1 has logic to compare the frequencies of its two receive clocks, to determine which is faster and to use the faster of the two clocks to derive the TDM subsystem clock. The **im2t1_rcvclk** function is used by host applications to select this mode of operation. With TDMCLK derived from the faster T1 receive clock, any frame slips occurring on the other T1 line will be due to elastic store under-flows and data will be repeated rather than lost. Should the frequencies of the two incoming T1 lines drift, the IM2T1 will automatically switch which receive clock is used to derive TDMCLK. Furthermore, if one of the framers is not in sync, the IM2T1 will automatically select the other framer's receive clock to derive TDMCLK.

Additional logic, available in Revision 2 of the IM2T1, provides a mode by which TDM data from a slipped frame is invalidated. This is accomplished using the TDM VALID signals that accompany each TDM data bus. For each TDM slot sourced by an T1 line during a slipped frame, the corresponding TDM VALID is pulsed low. Modules capable of interpreting the TDM Valid signals will be able to filter out data repeated due to slips. The slipped frame invalidation mode is enabled using the **im2t1_slipinval** function described in the IM2T1 Software Support section of this manual.

Slips are indicated to the IM2T1 logic by an interrupt from the T1 framer. The program controlling the IM2T1 (running either on the host or a processor on the V6M6) must enable and service this interrupt. To enable the interrupt, bit 4 of the framer's **Interrupt Mask Register 1** must be turned on. Servicing the interrupt requires the following standard sequence of accessing status from the framer:

Write a 1 to bit 4 of **Status Register 1**
Read **Status Register 1**
Write a 1 to bit 4 of **Status Register 1**

Important notes

1. The slip interrupt must be serviced in time for the next frame slip to be recognized. The time between slips will vary depend on the frequency offset between the two incoming RCLKs. For example, with a frequency offset of 0.01%, a slip can be expected every 1.25 seconds.

2. Status Register 1 and the corresponding Interrupt Mask Register 1 contain other important status bits that an application is likely to be interested in and that may be used for other interrupt conditions.

Yet another situation for some applications is the use of more than one IM2T1 module. If the timing for all of the incoming T1 lines is from the same time base, all of the incoming T1 data will remain in sync. However, if the T1 lines are coming from different timing sources the possibility of slippage and lost data becomes an issue.

To aid in this situation, the IM2T1 compares its receive clock frequencies with the frequency of TDMCLK. When one of its two receive clocks is faster than TDMCLK, it sets a status flag which may be enabled to assert a PCI interrupt. The interrupt is enabled using the **im2t1_intenb** function. The **im2t1_intstat** function is used to read the status of this detection. The status condition is latched and remains active until cleared by calling the **im2t1_clrfastclk** function. When this interrupt is serviced, the application monitoring T1 performance may decide to switch which IM2T1 module is used to source TDMCLK.

5.3.2.3 Electrical Line Interface

The IM2T1 interfaces to external T1 networks via two RJ45 connectors, one for each line. The electrical interface includes voltage clamping and current limiting devices to protect the module and carrier board from differential and common mode spikes and noise on the T1 signal wires.

The IM2T1 has selectable input impedance matching for the AMI interface. The options are 100Ω or Hi-Z. The impedance is selected individually for each of the four T1 lines via jumper settings. Figure 5-2 shows the location of the impedance selection jumpers and the RJ45 connectors on the module.

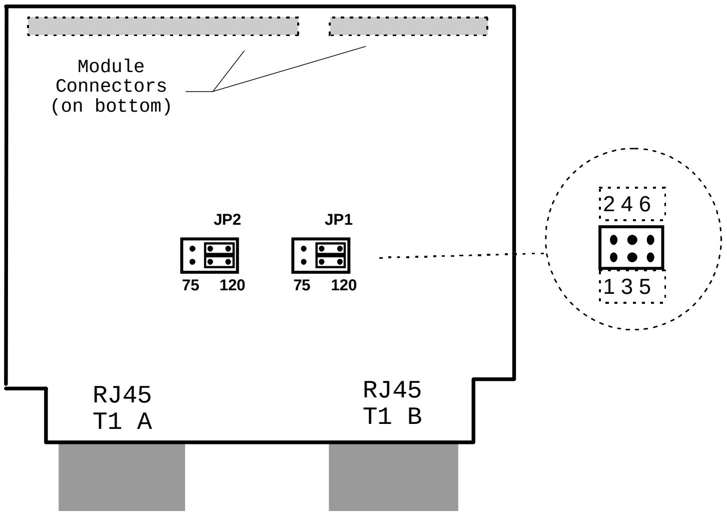
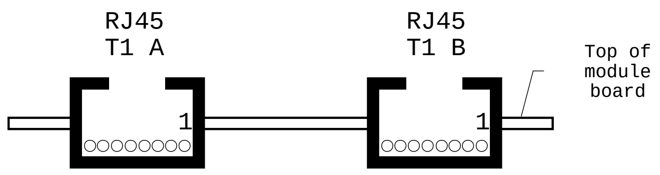


Figure 5-2: IM2T1 Line Impedance Selection

The AMI line input impedance is selected by positioning jumper pairs on JP1 and JP2 to connect the middle pins with the pins on the left (3-5 and 4-6) pins, as shown in the figure. Both pairs of jumpers on JP1 and JP2 must be positioned this way for proper electrical operation. High input impedance (for using the IM2T1 to monitor a T1 line without additional loading) may be selected by positioning the jumpers to connect pins 5-6 and pins 1-2.

Figure 5-3 shows the signal connections for the RJ45 connectors. In this view pin 1 is at the left.



Pin	Signal
1	Receive Ring
2	Receive Tip
4	Transmit Ring
5	Transmit Tip

Figure 5-3: IM2T1 RJ45 Pinout

5.3.2.4 LED Indicator

The IM2T1 uses the LED for its module location to indicate the following status:

RED	Module FPGAs are not configured
GREEN	Receive synchronization on one or both T1 lines

5.3.2.5 Specifications

The table below lists some of the significant performance characteristics of the IM2T1 module and its network interface components. For further details consult the data sheets for the following components, some of which are available in the appendices section of the reference manual.

T1 Line Interface and Decoding and Framing:

Dallas Semiconductor	DS2151Q	T1 / B8ZS Transceiver and Framer

T1 to TDM Interface:

Dallas Semiconductor	DS2175	T1 Elastic Store

Parameter	Min	Typ	Max	Units
HDB3 Interface				
Line Impedance (selectable) Coax Cable Twisted Pair		75 120		Ω
Output transformer ratio		1:1.15		
Line Length			1500	meters
Output Pulse Amplitudes 100Ω twisted pair		2.37 3.00		V peak
Input Surge Protection (breakover)	27		36	V
Power Requirements		TBD		Amps @ 5V
MTBF		TBD		1000 hrs

Notes:

1. Characteristics apply under ambient temperature of 0 to +55 °C.

5.3.3 IM2T1 Software Support

5.3.3.1 V6M6 Host Application Library for IM2T1

The V6M6 Host Application Library includes several functions which control the IM2T1 module, the DS2151Q framers and the TDM subsystem. These are C-callable functions that may be used in application programs.

Functions operating on an IM2T1 module require a PCI_MOD structure pointer obtained by calling the **pciopen** function. The module type of the open module must be **PCI_IM2T1** (a macro defined in <**pciutil.h**>. Functions operating on the TDM subsystem require a PCI_TDM structure pointer obtained by calling the **pci_tdmopen** function.

Example:

```
#include <pciutil.h>

PCI_MOD *pci;
PCI_TDM *tdm;

pci = pciopen ("pci0c");
if  (pci == NULL) {
    perror ("pci0c");
    exit (1);
}

if  (pci->module_type != PCI_IM2T1) {
    printf ("IM2T1 not installed on pci0c\n");
    exit (1);
}

tdm = pci_tdmopen ("pci0");
if  (tdm == NULL) {
    perror ("pci0t");
    exit (1);
}
```

5.3.3.1.1 IM2T1 Functions Quick Reference

These are the C-callable functions of the V6M6 Host Application Library used for programming the IM2T1 module. Full descriptions begin on page 65.

IM2T1 and Framer Control

`int` **`im2t1_init`**`(`*`pci`*`)`	Initializes the IM2T1 module.
`int` **`im2t1_intenb`**`(`*`pci, intmask`*`)`	Loads the IM2T1 interrupt mask register to specify the routing of interrupts from the IM2T1.
`int` **`im2t1_intstat`**`(`*`pci`*`)`	Reads the IM2T1 interrupt status register.
`int` **`im2t1_clrfastclk`**`(`*`pci`*`)`	Clears the status of the comparison of the receive clock frequencies and TDMCLK.
`int` **`im2t1_outparams`**`(`*`pci, line, params`*`)`	Loads control registers in a T1 framer from an array of values.
`int` **`im2t1_lirst`**`(`*`pci, line`*`)`	Momentarily activates the LIRST of the selected T1 framer.
`u_long` **`im2t1_status`**`(`*`pci, line, statmask`*`)`	Reads the four status registers of the selected T1 framer.
`int` **`im2t1_xmtsig`**`(`*`pci, line, data`*`)`	Writes data to the signaling registers in a T1 framer chip.
`int` **`im2t1_rcvsig`**`(`*`pci, line, data`*`)`	Reads data from the signaling registers in a T1 framer chip.
`u_char` **`im2t1_inreg`**`(`*`pci, line, reg`*`)`	Reads a single register from a T1 framer chip.
`int` **`im2t1_outreg`**`(`*`pci, line, reg, data`*`)`	Writes an 8-bit value to a single register in a T1 framer.

TDM Clock, Framing and Synchronization

int **im2t1_xmtclk**(*pci*, *select*)	Selects the source of the transmit clocks for the T1 lines on an IM2T1 module.
int **im2t1_rcvclk**(*pci*, *select*)	Selects whether or not one of the T1 receive clocks on the IM2T1 is used to derive the TDM subsystem clock.
int **im2t1_tdmclkdiv**(*pci*, *clkdiv*)	Sets the ratio between T1 clocks and TDM subsystem clocks.
int **im2t1_slipinval**(p*ci*, *select*)	Enables or disables the slipped frame data invalidation mode.
int **pci_tdm_clksrc**(*tdm*, *clksrc*)	Selects the clock source for the TDM subsystem control logic.
int **pci_tdm_init**(*tdm, clk, bits, slots, frames*)	Initializes the parameters of the TDM subsystem controller.
int **pci_tdm_run**(*tdm*)	Starts the TDM subsystem timing logic.
int **im2t1_init_rsync**(*pci*, *line*)	Initializes synchronization of the incoming T1 frames to the TDM subsystem frames.

TDM Connection MAP Functions

int **pci_tdm_src_add**(*tdm, slot, bus, devtyp, devnum*)	Adds a device as a TDM data source for the specified time *slot* and TDM *bus*.
int **pci_tdm_src_del**(*tdm, slot, bus*)	Deletes a device as a TDM data source for the specified time *slot* and TDM *bus*.
int **pci_tdm_dst_add**(*tdm, slot, bus, devtyp, devnum*)	Adds a device as a TDM data destination for the specified time *slot* and TDM *bus*.
int **pci_tdm_dst_del**(*tdm, slot, devtyp, devnum*)	Deletes a device as a TDM data destination for the specified time *slot* and TDM *bus*.
int **pci_tdm_special_add**(*tdm, slot, devtyp, devnum*)	Adds reads of incoming T1 elastic store buffers which do not drive data onto a TDM bus.
int **pci_tdm_special_del**(*tdm, slot, devtyp, devnum*)	Deletes reads of incoming T1 elastic store buffers which do not drive data onto a TDM bus.

5.3.3.1.2 IM2T1 and Framer Control

The following library functions are used to control the IM2T1 module and its DS2151Q T1 framers.

function: `int` **`im2t1_init`**`(`*`pci`*`)`

args: `PCI_MOD *`*`pci`*

return:
```
1    IM2T1 module initialized.
0    error indicated.
```

errors:
```
EINVAL   pci is NULL or doesn’t reference an IM2T1.
EIO      error in communicating with the module or
         the framer chips.
```

The **im2t1_init** function initializes the IM2T1 module. This includes setting up the module’s PCI interface and clearing the DS2151Q registers. It is called from the **pciinit** program. This function should not be used by an application that expects the IM2T1 to be previously configured and currently in operation.

function: `int` **`im2t1_intenb`**`(`*`pci`*`, `*`intmask`*`)`

args:

```
PCI_MOD  *pci
int       intmask
```

return:

```
1    interrupts enabled.
0    error indicated.
```

errors:

```
EINVAL   pci is NULL or doesn’t reference an IM2T1.
EIO      error in communicating with the module.
```

The **im2t1_intenb** function loads the IM2T1 interrupt mask register bits to specify the routing of interrupts (from the DS2151Q framers and the comparison of receive clocks with TDMCLK) to the PCI bus interrupts. Bits turned on in *intmask* enable interrupts as follows:

im2t1_intstat mask bits

intmask Bit	IM2T1 Interrupt	to	PCI Interrupt
0	Line 0 Int 1		A
1	Line 0 Int 2		A
2	Line 1 Int 1		A
3	Line 1 Int 2		A
4	Line 0 Int 1		B
5	Line 0 Int 2		B
6	Line 1 Int 1		B
7	Line 1 Int 2		B
8	RCLK > TDMCLK		A
9	RCLK > TDMCLK		B

function:	`int` **`im2t1_intstat`**`(`*`pci`*`)`
args:	`PCI_MOD   *`*`pci`*

return:	*`status`*
	`-1   error indicated.`

errors:	`EINVAL   `*`pci`*` is NULL or doesn't reference an IM2T1.`
	`EIO      error in communicating with the module.`

The **im2t1_intstat** function reads the IM2T1 interrupt status register bits which indicate the state of the interrupts from the DS2151Q framers and the comparison of receive clocks with TDMCLK. Bits turned on in the returned *intstat* indicate the interrupts as follows:

im2t1_intstat returned status	
status Bit	IM2T1 Interrupt
0	Line 0 Int 1
1	Line 0 Int 2
2	Line 1 Int 1
3	Line 1 Int 2
4	RCLK > TDMCLK

function:	`int` **`im2t1_clrfastclk`**`(`*`pci`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
return:	`1`	fast RCLK detection status cleared
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or doesn’t reference an IM2T1.`
	`EIO`	`error in communicating with the module.`

The **im2t1_clrfastclk** function clears the status of the comparison of the frequencies of the 2 receive clocks and TDMCLK. See Section 5.3.2.2.3 TDM Clock Rates and Frame Slips for further information.

function:	`int` **`im2t1_outparams`**`(`*`pci`*`,` *`line`*`,` *`params`*`)`	
args:	`PCI_MOD`	`*`*`pci`*
	`int`	*`line`*
	`u_char`	`*`*`params`*
return:	`1`	`Framer registers loaded.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or doesn’t reference an IM2T1,` *`line`* `is invalid,` *`params`* `is NULL.`
	`EIO`	`error in communicating with the module.`

Most applications are likely to use a fixed set of operational settings for the DS2151Q chips. The **im2t1_outparams** function provides a way to load most of control registers in the DS2151Q with an array of values.

The *line* argument (0 or 1) specifies which DS2151Q is to be loaded. The *params* argument points to an array of 48 unsigned characters. It contains the values to be loaded into registers 0x00 through 0x2F of the selected DS2151Q with the following exceptions and conditions:

- Values that correspond to read-only registers are ignored.
- The Test Register is always loaded with zero.
- Values that correspond to the Status and Receive Information registers should contain 1’s for any status bits to be cleared.
- The bit corresponding to LIRST in CCR3 is ignored and replaced with a zero (use the **im2t1_lirst** function for line interface reset).

```
function:   int im2t1_lirst(pci, line)

args:       PCI_MOD   *pci
            int        line

return:     1    LIRST pulsed.
            0    error indicated.

errors:     EINVAL   pci is NULL or doesn’t reference an IM2T1,
                     line is invalid.
            EIO      error in communicating with the module.
```

The **im2t1_lirst** function momentarily activates the LIRST bit in Common Control Register 3 of the selected DS2151Q. This may be required during operation if a problem with the incoming signal is detected. The *line* argument (0 or 1) specifies which DS2151Q is to be reset.

function:	`u_long` **`im2t1_status`**`(`*`pci`*`,` *`line`*`,` *`statmask`*`)`

args:	`PCI_MOD` `*`*`pci`*
	`int` *`line`*
	`u_long` *`statmask`*

return:	*`status`* `framer status (see table below).`
	`0xffffffff` `error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or doesn't reference an IM2T1,` *`line`* `is invalid.`
	`EIO` `error in communicating with the module.`

The DS2151Q has four registers which provide status information to the host. These are Status Register 1, Status Register 2, Receive Information Register 1 and Receive Information Register 2. These registers require subsequent write accesses in order to clear the status information after it is read.

The **im2t1_status** function handles reading of all four status registers and clearing selectable active status bits. The *line* argument (0 or 1) specifies which DS2151Q is to be accessed.

The *statmask* argument specifies which bits of Status Registers 1 and 2 and the Receive Information Registers 1 and 2 are to be read and cleared as shown in the table, below.

im2t1_status mask and returned bits

DS2151Q Register	*statmask* bits	*status* bits
Status Register 1	0:7	0:7
Status Register 2	8:15	8:15
Rcv Information Reg 1	23:16	23:16
Rcv Information Reg 2	31:24	31:24

function: `int` **`im2t1_xmtsig`**`(`*`pci`*`, `*`line`*`, `*`data`*`)`
`int` **`im2t1_rcvsig`**`(`*`pci`*`, `*`line`*`, `*`data`*`)`

args:

```
PCI_MOD  *pci
int       line
u_char   *data
```

return:

```
1    data transferred to / from DS2151Q.
0    error indicated.
```

errors:

```
EINVAL   pci is NULL or doesn't reference an IM2T1,
         line is invalid or data is NULL.
EIO      error in communicating with the module.
```

The DS2151Q framers automatically extract and insert Channel Associated Signaling bits to and from registers. This operation is described in section 7 of the DS2151Q data sheet. The V6M6 Host Support Library contains functions for reading and writing the signaling registers in the IM2T1 framer chips.

The Transmit Signaling registers (0x70 through 0x7B) are loaded using the **im2t1_xmtsig** function. The Receive Signaling registers (0x60 through 0x6B) are read using the **im2t1_rcvsig** function.

For both functions the *line* argument specifies which DS2151Q is to be accessed. The *data* argument is a pointer to an array of 12 unsigned characters that contains the data to be written to or read from the framer.

function: `u_char` **`im2t1_inreg`**`(`*`pci`*`, `*`line`*`, `*`reg`*`)`

```
args:     PCI_MOD   *pci
          int        line
          int        reg

return:   data   read from DS2151Q register.
          0       error indicated.

errors:   EINVAL   pci is NULL or doesn’t reference an IM2T1,
                   or line or reg invalid.
          EIO      error in communicating with the module.
```

The **im2t1_inreg** function may be used to read a single register from the DS2151Q specified by the *line* argument.

function: `int` **`im2t1_outreg`**`(`*`pci`*`, `*`line`*`, `*`reg`*`, `*`data`*`)`

```
args:     PCI_MOD   *pci
          int        line
          int        reg
          u_char     data

return:   1    data transferred to DS2151Q register.
          0    error indicated.

errors:   EINVAL   pci is NULL or doesn’t reference an IM2T1,
                   or line or reg invalid.
          EIO      error in communicating with the module.
```

The **im2t1_outreg** function may be used to write an 8-bit value, *data*, to a single register in the DS2151Q specified by the *line* argument.

For both the **im2t1_inreg** and **im2t1_outreg** functions the *reg* argument may be the offset of the register in the DS2151Q chip or one of the following macros, defined in <**pciutil.h**>:

IM2T1REG_STAT1	Status Register 1
IM2T1REG_STAT2	Status Register 2
IM2T1REG_RECV1	Receive Information Register 1
IM2T1REG_RECV2	Receive Information Register 2
IM2T1REG_MASK1	Interrupt Mask Register 1
IM2T1REG_MASK2	Interrupt Mask Register 2
IM2T1REG_RCTL1	Receive Control Register 1
IM2T1REG_RCTL2	Receive Control Register 2
IM2T1REG_TCTL1	Transmit Control Register 1
IM2T1REG_TCTL2	Transmit Control Register 2
IM2T1REG_CCTL1	Common Control Register 1
IM2T1REG_CCTL2	Common Control Register 2
IM2T1REG_CCTL3	Common Control Register 3
IM2T1REG_LICTL	*Line* Interface Control Register
IM2T1REG_LCVC1	*Line* Code Violation Count 1
IM2T1REG_LCVC2	*Line* Code Violation Count 2
IM2T1REG_PCVC1	Path Code Violation Count 1
IM2T1REG_PCVC2	Path Code Violation Count 2
IM2T1REG_MOSC1	Multiframe Out of Sync Count 1
IM2T1REG_RFDL	Receive FDL Register
IM2T1REG_FDLM1	Receive FDL Match Register 1
IM2T1REG_FDLM2	Receive FDL Match Register 2
IM2T1REG_RMRK1	Receive Mark Register 1
IM2T1REG_RMRK2	Receive Mark Register 2
IM2T1REG_RMRK3	Receive Mark Register 3
IM2T1REG_RBLK1	Receive Channel Blocking Register 1
IM2T1REG_RBLK2	Receive Channel Blocking Register 2
IM2T1REG_RBLK3	Receive Channel Blocking Register 3
IM2T1REG_XFDL	Transmit FDL Register
IM2T1REG_XBLK1	Transmit Channel Blocking Register 1
IM2T1REG_XBLK2	Transmit Channel Blocking Register 2
IM2T1REG_XBLK3	Transmit Channel Blocking Register 3
IM2T1REG_XIDL1	Transmit Idle Register 1
IM2T1REG_XIDL2	Transmit Idle Register 2
IM2T1REG_XIDL3	Transmit Idle Register 3
IM2T1REG_XIDLE	Transmit Idle Definition Register
IM2T1REG_XTRNP1	Transmit Transparency Register 1
IM2T1REG_XTRNP2	Transmit Transparency Register 2
IM2T1REG_XTRNP3	Transmit Transparency Register 3
IM2T1REG_RSIG	1st of 12 Receive Signaling Registers
IM2T1REG_TSIG	1st of 12 Transmit Signaling Registers

5.3.3.1.3 TDM Clock, Framing and Synchronization

The functions listed in this section control T1 clocks on an IM2T1 and set up the timing of the interface between IM2T1 modules and the TDM subsystem.

function:	`int` **`im2t1_xmtclk`**`(`*`pci`*`, `*`select`*`)`

args:	`PCI_MOD` `*`*`pci`*
	`int` *`select`*

return:	`1` `T1 transmit clocks selected.`
	`0` `error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or doesn't reference an IM2T1.`
	`EIO` `error in communicating with the module.`

The **im2t1_xmtclk** function selects the T1 transmit clock for both T1 *line*s on an IM2T1 module. The *select* argument encodes the selection as shown below. When deriving T1 transmit clock from TDMCLK, the IM2T1 generates a 1.544 MHz clock, dividing the TDMCLK rate as specified with the **im2t1_tdmclkdiv** function.

im2t1_xmtclk select encoding

Bits	controls	code	selection
3:2	TCLK A	00	RCLK A
		01	RCLK B
		10	TDM-Derived
1:0	TCLK B	00	RCLK B
		01	RCLK A
		10	TDM-Derived

Note that if the IM2T1 module is sourcing TDMCLK from either of its T1 receive clocks, it cannot use the actual TDMCLK for a T1 transmit clock. However, logic on the module detects this situation and will use the RCLK selected to derive TDMCLK and the T1 transmit clock instead of trying to derive T1 transmit clock from TDMCLK. This results in an equivalent T1 transmit clock because the ultimate source of TDMCLK and T1 transmit clock is the same—the selected T1 receive clock.

function:	`int` **`im2t1_rcvclk`**`(`*`pci`*`, `*`select`*`)`

args:	`PCI_MOD *`*`pci`*
	`int` *`select`*

return:	`1    RCLK to TDM clock selected.`
	`0    error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or doesn't reference an IM2T1.`
	`EIO    error in communicating with the module.`

The **im2t1_rcvclk** function selects whether or not one of the T1 receive clocks on the IM2T1 is used to derive the TDM subsystem clock. The actual TDM clock rate can be a multiple of the selected T1 RCLK as set with the **im2t1_tdmclkdiv** function. In addition, the **pci_tdm_clksrc** function must be used to select the clock source for the TDM system control, itself.

When receiving T1 data, it usually desirable to derive the TDM clock from the received T1 clock to ensure consistent capture of the incoming data. However, if more than one IM2T1 module is used or some other clock source must be used as the TDM master clock the module may be set to not drive the TDM clock.

When the two incoming T1 lines may be at differenct clock frequencies it is desirable to have the IM2T1 automatically select the faster of the two receive clocks as the source for deriving TDMCLK. See section 5.2.2.3 for more information.

im2t1_rcvclk settings	
select	RCLK used for TDM Clk
0	none
1	RCLK A
2	RCLK B
3	Faster of RCLK A and B

function: `int` **`im2t1_tdmclkdiv`**`(`*`pci`*`, `*`clkdiv`*`)`

args:

```
PCI_MOD   *pci
int        clkdiv
```

return:

```
1    Clock divider set.
0    error indicated.
```

errors:

```
EINVAL   pci is NULL or doesn't reference an IM2T1,
         or clkdiv is invalid.
EIO      error in communicating with the module.
```

The **im2t1_tdmclkdiv** function sets the ratio between T1 clocks and TDM subsystem clocks. For T1 transmit clocks derived from the TDM clock it specifies the ratio between the TDM clock rate and the T1 TCLK rate. For TDM clock derived from T1 receive clock, it determines the ratio between the selected T1 RCLK and the TDM clock.

The clock ratio is always based on an T1 clock of 1.544 MHz. Therefore the value of *clkdiv* actually specifies the TDM clock rate as:

im2t1_tdmclkdiv settings

clkdiv	TDM Clock Rate
0	2.048 MHz
1	4.096 MHz
2	8.192 MHz

```
function:  int im2t1_slipinval(pci, select)

args:      PCI_MOD   *pci
           int        select

return:    1    Slip invalidation mode set.
           0    error indicated.

errors:    EINVAL   pci is NULL or doesn't reference an IM2T1,
                    or the module is prior to Revision 2
           EIO      error in communicating with the module.
```

The **im2t1_slipinval** function enables or disables the slipped frame data invalidation mode described in section 5.2.2.3. The two low bits of the *select* argument determine whether this mode is enabled or disabled for each of the two T1 lines. A 1 enables and a 0 disables the slipped frame data invalidation mode.

There is no practical reason why one line would have this mode enabled and the other would not. Therefore, normal values for *select* are 0 for no slip invalidation mode or 3 for enabling slip invalidation mode. The T1 framer chips must also be programmed to enable framer interrupt 1 on elastic store slip events as described in section 5.2.2.3.

This feature is supported on IM2T1 modules of Revision 2 or higher.

function:	`int` **`pci_tdm_clksrc`**`(`*`tdm`*`,` *`clksrc`*`)`

args:	`PCI_TDM *`*`tdm`*
	`int` *`clksrc`*

return:	`1    Clock divider set.`
	`0    error indicated.`

errors:	`EINVAL` *`tdm`* `is NULL or` *`clksrc`* `is invalid.`

The **pci_tdm_clksrc** function selects the clock source for the TDM subsystem control logic. The values for *clksrc* defined in <pciutil.h> are:

TDM_BASECLK	selects the clock generated within the controller or from the TDM expansion port.
TDM_MEZZCLK	selects a clock generated from a module on the board.

The application must be careful not to enable more than one clock driver at a time. When TDM_BASECLK is used, none of the modules should be set to drive TDM clock. When TDM_MEZZCLK is used, one and only one module must generate the TDM clock.

function:	`int` **`pci_tdm_init`**`(`*`tdm, clk, bits, slots, frames`*`)`

args:	`PCI_TDM` `*`*`tdm`*
	`int` *`clk, bits, slots, frames`*

return:	`1` `TDM controller initialized.`
	`0` `error indicated.`

errors:	`EINVAL` *`tdm`* `is NULL or other arguments are invalid.`

The **pci_tdm_init** function initializes the parameters of the TDM subsystem controller.

For the IM2T1 use the following values for the arguments:

clk	When an IM2T1 (or any module or TDM expansion port) is configured to generate the TDM subsystem clock this argument must be 0. To generate TDM clocks from the TDM subsystem controller, a value of 2, 4, or 8 should be used. See the table below.
bits	Bits per time slot should always be set to 8.
slots	This value may be set to 32, 64 or 128, depending on the clock rate being used. The combination of clock rate and frame size must result in a 125 microsecond frame.
frames	Frames per multiframe should be set to 16.

TDM clock and slots values for IM2T1

TDM clock rate	TDM Slots per frame
2.048 MHz	32
4.096 MHz	64
8.192 MHz	128

```
function:  int pci_tdm_run(tdm)

args:      PCI_TDM   *tdm

return:    1    TDM subsystem started.
           0    error indicated.

errors:    EINVAL  tdm is NULL or other arguments are invalid.
           EIO     failed to update the TDM MAP RAM, usually
                   due to the absence of TDM clocks.
```

The **pci_tdm_run** function starts the TDM subsystem timing logic. The TDM map is updated, if necessary, and the TDM controller will begin generating the sync pulses as specified by the **pci_tdm_init** function.

```
function:  int im2t1_init_rsync(pci, line)

args:      PCI_MOD   *pci
           int        line

return:    1    T1 elastic store synchronized.
           0    error indicated.

errors:    EINVAL   pci is NULL or doesn't reference an IM2T1,
                    or line is invalid.
           EIO      error in communicating with the module.
```

The **im2t1_init_rsync** function initializes synchronization of the incoming T1 frames to the TDM subsystem frames. It must be called after the source of TDM clocks is established and the TDM subsystem is running.

It is necessary to call **im2t1_init_rsync**, for each T1 line to be used for incoming T1 data, whenever the TDM subsystem is started or restarted.

5.3.3.1.4 TDM Connection MAP Functions

The following library functions provide the means for setting up TDM data transfer connections between IM2T1 modules and other modules.

Reading an incoming T1 data slot or status involves making one of the IM2T1 lines a TDM source during a TDM time slot. Sending outgoing T1 data involves making one of the IM2T1 lines a TDM destination during a TDM time slot.

All 32 data slots of each T1 line must be read and written during a TDM frame. One or both lines of an IM2T1 may be read or written during the same TDM time slot.

function:

```
int pci_tdm_src_add(tdm, slot, bus, devtyp, devnum)
int pci_tdm_src_del(tdm, slot, bus)
```

args:

```
PCI_TDM   *tdm
int        slot
int        bus
int        devtyp
int        devnum
```

return:

```
1    Buffered TDM Map modified.
0    error indicated.
```

errors:

```
EINVAL  tdm is NULL or does not reference the TDM
        subsystem of a V6M6, or other
        arguments are invalid.
ENODEV  the specified devtyp (an IM2T1) is not
        installed at the module location specified
        in devnum.
```

These two functions add or delete a device as a TDM data source for the specified time *slot* and TDM *bus*. Only one device on a board may be the source for a TDM bus during a time slot. Therefore, it is not necessary to specify the device type or unit number for the delete function.

The following arguments values are used for the IM2T1 module:

slot	Values between 1 and the number of slots configured for the TDM subsystem frame (see **pci_tdm_init**).
bus	One of the macros defined in <**pciutil.h**>, **TDM_BUSA**, **TDM_BUSB**, **TDM_BUSC** and **TDM_BUSD**, corresponding to the four TDM busses.
devtyp	The device type for the IM2T1 the device type is **TDM_DEVIM2T1**. This is a macro defined in <**pciutil.h**>.
devnum	The device unit number encodes the module location and which T1 line for the module as shown in the table below.

IM2T1 TDM device unit number	
Bit 4 = T1 Line	Bits 3:0 = Module
0 = Line A 1 = Line B	0 = A 1 = B 2 = C 3 = D 5 = F

function:	`int` **`pci_tdm_dst_add`**`(`*`tdm, slot, bus, devtyp, devnum`*`)`
	`int` **`pci_tdm_dst_del`**`(`*`tdm, slot, devtyp, devnum`*`)`

args:	`PCI_TDM *`*`tdm`*
	`int` *`slot`*
	`int` *`bus`*
	`int` *`devtyp`*
	`int` *`devnum`*

return:	`1    Buffered TDM Map modified.`
	`0    error indicated.`

errors:	`EINVAL` *`tdm`* `is NULL or does not reference the TDM subsystem of a V6M6, or other arguments are invalid.`
	`ENODEV the specified` *`devtyp`* `(an IM2T1) is not installed at the module location specified in` *`devnum`*`.`

These two functions add or delete a device as a TDM data destination for the specified time *slot* and TDM *bus*. A device may be a destination for only one TDM bus during a time slot. Therefore,it is not necessary to specify the TDM bus for the delete function. The use of the arguments is the same as the **pci_tdm_src_add** function.

```
function:   int pci_tdm_special_add(tdm, slot, devtyp, devnum)
            int pci_tdm_special_del(tdm, slot, devtyp, devnum)

args:       PCI_TDM  *tdm
            int       slot
            int       devtyp
            int       devnum

return:     1    Buffered TDM Map modified.
            0    error indicated.

errors:     EINVAL  tdm is NULL or does not reference the TDM
                    subsystem of a V6M6, or other
                    arguments are invalid.
            ENODEV  the specified devtyp (an IM2T1) is not
                    installed at the module location specified
                    in devnum.
```

The TDM **special** functions are used to add or delete reads of incoming T1 elastic store buffers which do not drive data onto a TDM bus. These may be necessary when not all of the incoming data is being used and the TDM bandwidth is required for other data transfers. It is still necessary to specify unloading of the elastic stores to maintain slot and frame alignment. The use of the arguments is the same as the **pci_tdm_src_add** function.

5.3.3.2 MIPS Application Library

A library of functions provides access to the DS2151Q T1 framer registers and the IM2T1 interrupt mask and status by MIPS processor module applications run under the WM Kernel.

The functions are ported from the V6M6 Host Application Library and have the same names, arguments and return values. The IM2T1 functions implemented for MIPS applications are:

```
im2t1_intenb
im2t1_intstat
im2t1_outparams
im2t1_lirst
im2t1_status
im2t1_xmtsig
im2t1_rcvsig
im2t1_inreg
im2t1_outreg
```

5.3.3.3 Utility Programs

The following programs, in **$CAC/bin**, are provided for the V6M6 PCI carrier boards. They are useful for initializing and maintaining the IM2T1 module. Detailed descriptions of these programs are available in Appendix B or online in UNIX man file format provided that you followed the setup procedures in Section 2.3.2 Programmer and User Setup.

pciinit	initializes the base board and modules. It instructs the on-board microprocessor to configure FPGAs on the module. It also sets up PCI address mappings.
pciinfo	examines the EEROMs on the base board and modules and the flash memory. It then displays all pertinent version and serial number information.
pciflashup	compares flash object (FPGA configurations and microprocessor program) versions in the flash memory with those residing in host files. Depending on the command line options it will either display the version numbers or update the flash memory with any new versions on disk.

5.3.3.4 Diagnostic programs

The following diagnostic programs are being developed for the IM2T1 module. These programs will be distributed in the directory **$CAC/pci/diag**.

pcit1loop	Runs a loop-back test with a single IM2T1 sending data to itself. The data is generated from, received and tested by a processor module on the same baseboard. The T1 signal may be looped back externally or internal to the DS2151Q framer.
pcit1xmt	Transmits a known T1 data pattern generated from a processor module on the same baseboard.
pcit1rcv	Receives T1 data and compares with the known pattern used by **pcit1xmt**.

The **pcit1loop** program is one of the components of the test and burn-in program, **pciburn**.

5.6 IMHDLC

The IMHDLC module provides 64 channels of HDLC processing for the V6M6 PCI module carrier board using two Siemens Munich32 chips. The Munich chips perform layer 2 HDLC formatting and deformatting of data in external memory included on the module. The IMHDLC is available with two amounts of memory: 64K bytes (IMHDLC-64K) or 128K bytes (IMHDLC-128K) for channel map and data storage.

The memory busses of both Munich chips share one side of the dual port RAM. The other side of the dual port RAM and module control registers interface with the V6M6 PCI bus. The host system and/or processors on the V6M6 have access to the module's memory and control registers.

The serial input and output data streams of the Munich chips are connected to the V6M6 TDM Subsystem. Through the TDM interface HDLC data may be transferred between the Munich chips and other serial interfaces on the V6M6 such as T1, E1 and SCSA. The Munich chips are compatible with LAPD ISDN, HDLC, SDLC, and LAPB DMI protocols. Further information on the Munich chips may be found in Siemens User's Manual, "Ics for Communications - Munich32 - PEB 20320"

Information in this manual is divided into the following sections:

5.6.1 IMHDLC Quick Tour

Figure 8-1 is a block diagram of the IMHDLC showing the major components and signal paths. The primary components are the two Siemens Munich32 chips and the Dual Port RAM.

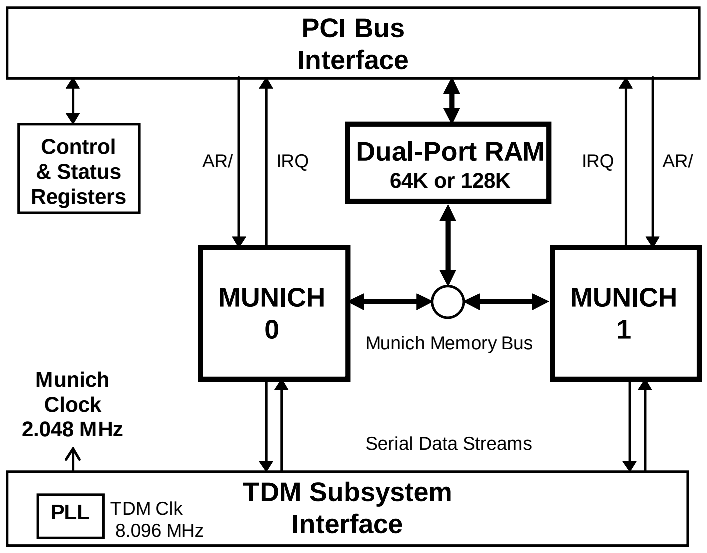


Fig. 8-1: IMHDLC Module Block Diagram

The Munich chips share one side of the Dual Port RAM for their program and data storage using bus arbitration. The other side of the Dual Port RAM is interfaced to the V6M6 PCI bus. V6M6 processor modules or the host system may access the Dual Port RAM through the PCI interface.

The IMHDLC is available with two amounts of Dual Port memory: 64K bytes or 128K bytes. For either amount of physical memory, the IMHDLC occupies 512K bytes of PCI address space to accommodate the use of address aliasing for special operations. These operations, Munich action requests and module interrupt acknowledge, are described in Section 5.6.2.3 Munich Action Requests and Interrupt Acknowledge.

The Munich chips' Identification pins are wired such that Control Start Address for Munich 0 is at address 0 and for Munich 1 at address 4.

Interrupt requests from each Munich chip increment counters in the PCI interface. The counters being non-zero become status signals that may be read and masked to generate PCI interrupts A and or B for the module. The counters are decremented by the interrupt acknowledge operation described in Section 5.6.2.3 Munich Action Requests and Interrupt Acknowledge.

The serial, multiplexed data streams in and out of the Munich chips are interfaced to the V6M6 TDM subsystem. The interface provides flexible mapping of TDM time slots to Munich time slots. This allows the IMHDLC module, in conjunction with IM2T1, IM2E1 and or IMSCSA modules, to extract or generate encoded data channels through various networks. The mapping mechanism is described in section 5.6.4 IMHDLC TDM Interface.

5.6.2 IMHDLC PCI Interface

The PCI side of the Dual Port RAM is accessed in PCI memory space. Registers for control of the module are accessed in PCI configuration space.

The table below shows the base addresses for the different module locations on the V6M6 baseboard. The configuration space base addresses are determined by physical hardware connections. Memory space base addresses are determined by the **pciinit** program based on the memory space utilized by other modules installed on the V6M6 board.

PCI Base Addresses for IMHDLC Modules

PCI Module	Config Space Base Address	Memory Space Base Address
A	0x01000000	Depends on
B	0x00800000	other modules
C	0x00400000	installed
D	0x00200000	and is
E	0x00100000	determined
F	0x00080000	by **pciinit**

5.6.2.1 Configuration, Status and Control

The table below lists the IMHDLC registers which are accessed in PCI configuration space. The addresses or address ranges shown are the offsets from the base address of the module.

IMHDLC Configuration Space Registers

Register Name	Address	Acc	Function / Description
PCI Stat/Cmd	0x04	rw	See PCI Spec for details
Base Addr	0x10	wo	PCI base address of module
Reset	0x40	rw	Munich Reset
Interrupt	0x44	rw	I/O Interrupt stat and mask
Module ID	0x48	rw	V6M6 module number

PCI Stat / Cmd Register:

This register is used for configuring and obtaining status of the module's interface to the PCI bus. The control bits are all either hardwired or programmed by the **pciinit** program.

Two status bits, listed below, are available for PCI error checking. These are cleared by writing a 1 to the bit to be cleared.

```
Bit 30 - Module signaled a System Error
Bit 31 - Module detected a PCI Parity Error
```

Base Address Register:

This register stores the module's base address on the PCI bus. It is programmed by the **pciinit** program and a copy of its contents is stored in the host's device driver and made available to host applications. The IMHDLC is hard-wired to respond in PCI Memory space.

Reset Register:

This register is used to control the reset of the Munich chips. A value of 1 in each bit will assert reset to the respective Munich chip.

```
Bit 0 - Munich 0 RESET
Bit 8 - Munich 1 RESET
```

Interrupt Register:

This register controls the interrupt enables and is used to read interrupt status for the module.

The IMHDLC control logic counts interrupt requests from the Munich chips. Whenever one of the counters is non-zero, a PCI interrupt will be asserted according to the setting of the interrupt mask. This mask is contained in four bits of the IMHDLC Interrupt Register. A 1 in each bit routes the Munich interrupts to the PCI interrupt lines as shown:

Mask Bits in the IMHDLC Interrupt Register

```
Bit 24 - Munich 0 Int to PCI interrupt A
Bit 25 - Munich 1 Int to PCI interrupt A
Bit 26 - Munich 0 Int to PCI interrupt B
Bit 27 - Munich 1 Int to PCI interrupt B
```

The status of the interrupt counters may be read from the Interrupt Register in the bits shown below:

Status Bits in the IMHDLC Interrupt Register

```
Bit 0 - Munich 0 interrupt non-zero
Bit 1 - Munich 1 interrupt non-zero
```

The interrupt status bits are valid regardless of the interrupt mask setting. A 1 indicates that the corresponding interrupt counter is non-zero.

The interrupt counters are decremented by performing an interrupt acknowledge access in memory space. This is described in Section 5.6.2.3 Munich Action Requests and Interrupt Acknowledge.

Module ID Register:

This register contains the module ID (0 through 5) and indicates the module's location on the V6M6 board. It is written by the **pciinit** program.

5.6.2.2 Munich Memory Access

The table below illustrates the memory mapping of the IMHDLC PCI address space.

IMHDLC PCI Memory Map						
PCI Address Bits						
18	17	16	•	•	•	0
Address Aliased	IMHDLC-128K					
Memory Space		IMHDLC-64K				

The physical memory is the PCI side of the Dual Port RAM used for Munich memory (128K bytes for the IMHDLC-128K or 64K bytes for the IMHDLC-64K). Note that the Munich chips' Identification pins are wired such that Control Start Address location for Munich 0 is at address 0 and for Munich 1 at address 4.

Applications written for the host system use the general purpose V6M6 data transfer routines to read and write the memory on the IMHDLC. These are described on section 5.6.3.1.2 Data Transfer Functions.

The PCI interface decodes address bits 17 and 18, along with the direction of the access (read or write) to perform Munich action request and module interrupt acknowledge functions on the Munich chips. These operations are described in section 5.6.2.3 Munich Action Requests and Interrupt Acknowledge.

5.6.2.3 Munich Action Requests and Interrupt Acknowledge

Munich action requests are performed concurrently with memory write operations when address bit 18 is set. Address bit 17 selects which Munich chip will receive the action request: Munich 0 when bit 17 is 0 or Munich 1 when bit 17 is 1.

A special function in the host library, **imhdlc_atten**, is used for this purpose. This function is described in section 5.6.3.1.3 IMHDLC Control Functions. Typically, an action request is performed concurrently with loading the Control Start Address, after loading the Munich's program and initial data.

An interrupt acknowledge is performed concurrently with a memory read operation when address bit 18 is set. This operation decrements the IMHDLC's interrupt counter for the selected Munich chip. Address bit 17 selects which Munich's interrupt counter is decremented: Munich 0 when bit 17 is 0 or Munich 1 when bit 17 is 1. The module's interrupt status for a Munch chip is deasserted when the interrupt counter is at 0.

A special function in the host library, **imhdlc_intack**, described in section 5.6.3.1.3 IMHDLC Control Functions, is used for this purpose. Typically, an interrupt acknowledge is performed concurrently with reading the Munich's current interrupt queue as part of the application's interrupt service routine.

5.6.3 Application Interfaces for Munich Data and Control

Software interfaces to the IMHDLC for application programmers. These are in the form of library functions and/or system calls for host applications, MIPS Kernel applications and MIPS pSOS applications.

5.6.3.1 V6M6 Host Application Library for the IMHDLC

The V6M6 Host Application Library contains functions for transferring data between the host system and PCI memory space. These are used for reading and writing the Dual Port RAM on the IMHDLC. The library also includes special functions for control of the IMHDLC and Munich chips.

These functions all require the use of a pointer to a data type named PCI_MOD structure. A typical application program will declare one or more PCI_MOD pointers and initialize them by calling the **pciopen** function with the name of the module to be accessed. For example:

```
PCI_MOD *pci;

pci = pciopen(“pci0a”);
if (pci == NULL) {
    perror(“pci0a”);
    exit(1);
}
```

Having successfully opened the module, the program will pass the PCI_MOD pointer to the various functions to operate on or transfer data to or from the module.

5.6.3.1.1 IMHDLC Functions Quick Reference

These are the C-callable functions of the V6M6 Host Application Library used for programming the IMHDLC module. Full descriptions begin on page 5-99.

Data Transfer Functions

int **pci_dl_a32b**(*pci, pciadr, nwords, buf*)	Transfers an array **from the host to PCI memory**.
int **pci_up_a32b**(*pci, pciadr, nwords, buf*)	Transfers an array of 32-bit values **from PCI memory to the host**.
int **pci_dl_i32b**(*pci, pciadr, value*)	Transfers a single 32-bit value **from the host to PCI memory**.
u_long **pci_up_i32b**(*pci, pciadr*)	Transfers a single 32-bit value **from PCI memory to the host**.

IMHDLC Control Functions

int **imhdlc_init**(*pci*)	Initializes the IMHDLC module.
int **imhdlc_atten**(*pci, munsel, reladr, value*)	Performs a special memory write generating a Munich **action request** on the IMHDLC module.
int **imhdlc_intenb**(*pci, intmask*)	Sets the interrupt enable mask of the IMHDLC module.
int **imhdlc_intstat**(*pci*)	Reads the interrupt status from the IMHDLC module.
u_long **imhdlc_intack**(*pci, munsel, reladr*)	Performs a special memory read, generating an **interrupt acknowledge** on the IMHDLC module.

Programming IMHDLC TDM Connections

`int` **`pci_tdm_src_add`**`(`*`tdm, slot, bus, devtyp, devnum`*`)`	Adds a device as a TDM data source for the specified time *slot* and TDM *bus*.
`int` **`pci_tdm_src_del`**`(`*`tdm, slot, bus`*`)`	Deletes a device as a TDM data source for the specified time *slot* and TDM *bus*.
`int` **`pci_tdm_dst_add`**`(`*`tdm, slot, bus, devtyp, devnum`*`)`	Adds a device as a TDM data destination for the specified time *slot* and TDM *bus*.
`int` **`pci_tdm_dst_del`**`(`*`tdm, slot, devtyp, devnum`*`)`	Deletes a device as a TDM data destination for the specified time *slot* and TDM *bus*.
`int` **`pci_tdm_special_add`**`(`*`tdm, slot, devtyp, devnum`*`)`	Adds data shift operations, moving data between temporary data registers and HDLC time slots.
`int` **`pci_tdm_special_del`**`(`*`tdm, slot, devtyp, devnum`*`)`	Deletes HDLC data shift operations.

5.6.3.1.2 Data Transfer Functions

Data transfer between the host and the Dual Port RAM is performed using memory mapped I/O. Transfers are limited to those involving 32 bit values. The V6M6 host application library provides the following functions for data transfers.

These functions may not be used to access the PCI addresses for performing Action Requests or Interrupt Acknowledge cycles. Section 5.6.3.1.3 IMHDLC Control Functions describes functions provided specifically for those operations.

function:	`int` **`pci_dl_a32b`** `(pci, pciadr, nwords, buf)`
	`int` **`pci_up_a32b`** `(pci, pciadr, nwords, buf)`

args:	`PCI_MOD  *pci`
	`u_long    pciadr, nwords, buf`

return:	`1          transfer was successful.`
	`0          error indicated.`

errors:	`EINVAL` *`pci`* `is NULL,` *`pciadr`* `out of bounds,` *`nwords`* `is 0, or` *`buf`* `is NULL`
	`EIO` `error in communicating with module.`

The **pci_dl_a32b** function transfers an array from the host program's memory to PCI memory on the module or baseboard referenced by *pci*. The **pci_up_a32b** function transfers an array from PCI memory to the host program's memory.

The *pciadr* argument specifies the PCI memory space address, which may be local memory on the module referenced by *pci* or global memory on the baseboard. *pciadr* must be modulo-4 and must reside within the module's local memory space or the baseboard's global memory space.

Note that this is an absolute PCI address. To write to a location at a given offset from the module's base address, the offset must be added to the module's base address. This value is available in the PCI_MOD structure:

```
PCI_MOD  *pci ;
u_long  pciadr,  offset ;

pciadr  =  pci->config.mod_par_min  +  offset;
```

The *nwords* argument specifies how many 32-bit words to transfer. The *buf* argument specifies the host program's data buffer.

function:	`int `**`pci_dl_i32b`**` (pci, pciadr, value)`

args:	`PCI_MOD *pci`
	`u_long pciadr, value`

return:		
	`1`	`transfer was successful.`
	`0`	`error indicated.`

errors:		
	`EINVAL`	*`pci`* `is NULL or` *`pciadr`* `out of bounds`
	`EIO`	`error in communicating with module.`

The **pci_dl_i32b** function transfers a single 32-bit value from the host to PCI memory on the module or baseboard referenced by *pci*. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* is an absolute PCI address, must be modulo-4 aligned and must reside within the DSP's local memory space or the baseboard's global memory space.

function:	`u_long` **`pci_up_i32b`** `(pci, pciadr)`

args:	`PCI_MOD  *pci`
	`u_long    pciadr`

return:	*`value`* `data uploaded from PCI memory.`
	`0` `error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or` *`pciadr`* `out of bounds`
	`EIO` `error in communicating with module.`

The **pci_up_i32b** function transfers a single 32-bit value from PCI memory on the module or baseboard referenced by *pci* to the host. The *pciadr* argument specifies the PCI memory space address, which may be local memory of the DSP or global memory on the baseboard. Note that *pciadr* must be modulo-4 aligned and reside within the module's local memory space or the baseboard's global memory space.

The function returns the 32-bit value uploaded from PCI memory. Note that an error condition is indistinguishable from a successful value of 0. To avoid this ambiguity, clear the global variable **errno** prior to the function call. Then test it after the function call. For example:

```
u_long  dspadr, data;
extern int  errno;

errno = 0;
data = pci_up_i32b (pci, dspadr);
if  (errno != 0)
    perror(“pci_up_i32b error”);
```

5.6.3.1.3 IMHDLC Control Functions

The following functions are available in the host application library for controlling the IMHDLC module and Munich chips.

function:	`int` **`imhdlc_init`** `(pci)`	
args:	`PCI_MOD *pci`	
return:	`1`	`initialization complete.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference an IMHDLC module.`
	`EIO`	`error in communicating with module.`

The **imhdlc_init** function initializes the IMHDLC module referenced by the PCI_MOD pointer, *pci*. The initialization performs the following steps:

- Resets both Munich chips
- Clears the interrupt counters for both Munich chips.
- Clears the interrupt mask on the module
- Loads the Module ID Register.

function:	`int` **`imhdlc_atten`** `(pci, munsel, reladr, value)`	
args:	`PCI_MOD`	`*pci`
	`int`	`munsel`
	`u_long`	`reladr`
	`u_long`	`value`
return:	`1`	`successful transfer.`
	`0`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference an IMHDLC module, or` *`munsel`* `or` *`reladr`* `are invalid.`
	`EIO`	`error in communicating with module.`

The **imhdlc_atten** function performs a special memory write generating a Munich **action request** on the IMHDLC module referenced by the PCI_MOD pointer, *pci*. The *munsel* argument selects the Munich chip to receive the action request: 0 for Munich 0 or 1 for Munich 1.

The *reladr* and *value* arguments specify the address and data to be written to the Munich memory. Note that *reladr* specifies the offset from the module's base address (rather than an absolute PCI address as used for the general purpose PCI transfer functions).

function:	`int` **`imhdlc_intenb`** `(pci, intmask)`

args:	`PCI_MOD *pci`
	`int intmask`

return:	*`status`* `module interrupt status.`
	`-1 error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or does not reference an IMHDLC module.`
	`EIO error in communicating with module.`

The **imhdlc_intenb** function sets the interrupt enable mask of the IMHDLC module referenced by the PCI_MOD pointer, *pci*. It loads the interrupt mask portion of the module's Interrupt Register. These control the routing of Munich interrupts to the PCI interrupt lines for the module. Bits 4:0 of the *intmask* argument correspond to the following interrupt routing:

```
Bit 0 - Munich 0 Int to PCI interrupt A
Bit 1 - Munich 1 Int to PCI interrupt A
Bit 2 - Munich 0 Int to PCI interrupt B
Bit 3 - Munich 1 Int to PCI interrupt B
```

function:	`int` **`imhdlc_intstat`** `(pci)`	
args:	`PCI_MOD *pci`	
return:	*`status`*	`module interrupt status.`
	`-1`	`error indicated.`
errors:	`EINVAL`	*`pci`* `is NULL or does not reference an IMHDLC module.`
	`EIO`	`error in communicating with module.`

The **imhdlc_intstat** function reads the interrupt status from the IMHDLC module referenced by the PCI_MOD pointer, *pci*. The *status* returned reflects the current state of the Munich interrupt counters from the module's Interrupt Register.

```
Bit 0 - Munich 0 interrupt non-zero
Bit 1 - Munich 1 interrupt non-zero
```

Note that the interrupt bits read in *status* are not affected by the interrupt mask.

function:	`u_long` **`imhdlc_intack`** `(pci, munsel, reladr)`

args:	`PCI_MOD  *pci`
	`int       munsel`
	`u_long    reladr`

return:	*`value`* `data from Munich memory.`
	`0` `error indicated.`

errors:	`EINVAL` *`pci`* `is NULL or does not reference an IMHDLC module, or` *`munsel`* `or` *`reladr`* `are invalid.`
	`EIO` `error in communicating with module.`

The **imhdlc_intack** function performs a special memory read, generating an **interrupt acknowledge** on the IMHDLC module referenced by the PCI_MOD pointer, *pci*. The *munsel* argument selects the Munich chip's interrupt counter to decrement: 0 for Munich 0 or 1 for Munich 1.

The *reladr* argument specifies the address of Munich memory to be read. Note that *reladr* specifies the offset from the module's base address (rather than an absolute PCI address as used for the general purpose PCI transfer functions).

The *value* returned is the 32-bit data read from the module's memory. Note that an error condition is indistinguishable from a successful value of 0. To avoid this ambiguity, clear the global variable **errno** prior to the function call. Then test it after the function call. See the example of using **pci_up_i32b** in section 5.6.3.1.2 Data Transfer Functions.

5.6.3.2 MIPS Kernel Interface

(documentation not yet available)

5.6.3.3 MIPS pSOS Interface

(documentation not yet available)

5.6.4 IMHDLC TDM Interface

Serial data is transferred to and from the Munich chips on the IMHDLC module via the V6M6 TDM subsystem. The IMHDLC module requires a TDM subsystem clock rate of 8 MHz and a TDM frame rate of 8 KHz (or 125 microseconds per frame). This translates to 128 8-bit TDM slots per frame. A typical call to the **tdm_init** function to achieve these parameters would be:

```
pci_tdm_init(tdm, 8, 8, 128, 24)
```

specifying the 8 MHz clock, 8 bits per slot, 128 slots per frame and 24 frames per superframe.

The Munich chips transfer data at a rate of 2 MHz. Due to the different data rates the Munich time slot numbers do not correspond directly with the TDM subsystem time slots. However, the Munich frames are aligned to the TDM subsystem frames. The chart below show the alignment of Munich and TDM slots.

Munich Time Slot	TDM Time Slots
0	1 thru 4
1	5 thru 8
2	9 thru 12
•••	•••
30	121 thru 124
31	125 thru 128

Transferring data from the V6M6 TDM subsystem to the Munich involves capturing 8 bits of TDM data into a temporary register during a TDM time slot. Once captured, the data is shifted, two bits at a time, into the Munich data stream.

Transferring data from the Munich to the V6M6 TDM subsystem involves shifting data from the Munich data stream, 2 bits at a time, into a temporary register. The stored data is then driven onto an 8-bit TDM data time slot. Each capture, shift and drive operation occurs during a TDM slot time as specified in the TDM control Map in section 5.6.4.3 Programming IMHDLC TDM Connections.

Two common ways of using TDM / Munich data transfer are based on ISDN data rates: 64K-bit and 16K-bit Packed.

5.6.4.1 64K-bit Transfers:

A 64K-bit transfer consists of moving 8 bits between a single Munich time slot and a TDM time slot.

For 64K transfers from TDM to Munich, 8 bits of data is first captured from the TDM bus during a single 8-bit TDM time slot. This data is stored in a temporary register and later shifted into a Munich time slot during four subsequent TDM slot times. One of the shift operations may be overlapped with a capture operation for the next outgoing Munich time slot so that the net transfer rate consists of 4 TDM slot times per Munich slot time.

For 64K transfers from Munich to TDM, data is shifted out of the Munich time slot during four consecutive TDM slot times into a temporary register. This data is then driven to the TDM bus during a fifth TDM time slot. The TDM drive operation may be overlapped with one of the shift operations for the next Munich time slot so that the net transfer rate consists of 4 TDM slot times per Munich slot time.

5.6.4.2 16K-bit Packed Transfers:

A 16K-bit Packed transfer consists of moving 2 bits from four different Munich time slots to or from a single 8-bit TDM slot.

For 16K transfers from TDM to Munich, an 8-bit TDM slot is captured, stored, and shifted as for 64K-bit transfers. The difference is that the shift operations are spread out over 16 TDM time slots so that each shift is aligned with the beginning of the next Munich time slot.

For 16K transfers from Munich to TDM, data is shifted out of 4 different Munich time slots, 2 bits at a time, into a temporary register. The shift operations are spread out over 16 TDM slots to align each shift with the beginning of the next Munich time slot. Following the 4 shift operations, the contents of the temporary register is driven onto the TDM during a single 8-bit TDM time slot.

5.6.4.3 Programming IMHDLC TDM Connections

Munich serial data transfers are performed by TDM subsystem control words. The TDM control words are stored in the TDM Map RAM and are distributed to the V6M6 modules over the TDM control bus during each TDM slot time. Control words are loaded into the TDM Map RAM using the TDM functions in the host application library.

There are six TDM map functions used to program control words for the IMHDLC. **pci_tdm_src_add** and **pci_tdm_src_del** are used for adding or deleting connections from an IMHDLC module to the TDM subsystem. **pci_tdm_dst_add** and **pci_tdm_dst_del** are used for adding or deleting connections from the TDM subsystem to an IMHDLC module. **pci_tdm_special_add** and **pci_tdm_special_del** are used for adding or deleting shift operations both for data to and from the Munich chips.

The initial connection map may be programmed with a sequence of calls to **pci_tdm_src_add**, **pci_tdm_dst_add** and **pci_tdm_special_add**. These should be followed by a call to the **pci_tdm_run** function if the TDM subsystem is not yet running. If the TDM subsystem is already is running, use the **pci_tdm_updatemap** function. Changes to the Map may be made using the add and delete functions at any time. These changes are applied at a TDM frame boundary by calling the **pci_tdm_updatemap** function.

function:	`int` **`pci_tdm_src_add`**`(tdm, slot, bus, devtyp, devnum)`
	`int` **`pci_tdm_src_del`**`(tdm, slot, bus)`

args:	`PCI_TDM  *tdm`
	`int       slot`
	`int       bus`
	`int       devtyp`
	`int       devnum`

return:	`1    Buffered TDM Map modified.`
	`0    error indicated.`

errors:	`EINVAL` *`tdm`* `is NULL or does not reference the TDM subsystem of a V6M6, or other arguments are invalid.`
	`ENODEV the module type specified` *`devtyp`* `is not installed at the module location specified in` *`devnum`*`.`

These two functions add or delete a device as a TDM data source for the specified time *slot* and TDM *bus*. Only one device on a board may be the source for a TDM bus during a time slot so it is not necessary to specify the device type or unit number for the delete function.

The **pci_tdm_src_add** function is used to add **DRIVE** operations for the IMHDLC module. These put data shifted from Munich time slots onto the TDM data bus.

The following arguments values are used for the IMHDLC module:

slot	The TDM subsystem slot number, between 1 and 128.
bus	One of the macros defined in <pciutil.h>, TDM_BUSA, TDM_BUSB, TDM_BUSC and TDM_BUSD, corresponding to the four TDM busses.
devtyp	The TDM device type for the IMHDLC is TDM_DEVIMHDLC. This is a macro defined in <pciutil.h>.
devnum	The device unit number encodes the module location, which of the Munich chips on the module, and the transfer operation(s) to be performed during the TDM time slot. See chart below.

function:	`int` **`pci_tdm_dst_add`**`(tdm, slot, bus, devtyp, devnum)`
	`int` **`pci_tdm_dst_del`**`(tdm, slot, devtyp, devnum)`

args:	`PCI_TDM *tdm`
	`int slot`
	`int bus`
	`int devtyp`
	`int devnum`

return:	`1` Buffered TDM Map modified. `0` error indicated.
errors:	`EINVAL` *tdm* is NULL or does not reference the TDM subsystem of a V6M6, or other arguments are invalid. `ENODEV` the module type specified *devtyp* is not installed at the module location specified in *devnum*.

These two functions add or delete a device as a TDM data destination for the specified time *slot* and TDM *bus*. A device may be a destination for only one TDM bus during a time slot so it is not necessary to specify the TDM bus for the delete function.

The **pci_tdm_dst_add** function is used to add **CAPTURE** operations for the IMHDLC module. These capture data from the TDM data bus which will then be shifted into one or more Munich time slots.

The following arguments values are used for the IMHDLC module:

slot	The TDM subsystem slot number, between 1 and 128.
bus	One of the macros defined in <pciutil.h>, TDM_BUSA, TDM_BUSB, TDM_BUSC and TDM_BUSD, corresponding to the four TDM busses.
devtyp	The TDM device type for the IMHDLC is TDM_DEVIMHDLC. This is a macro defined in <pciutil.h>.
devnum	The device unit number encodes the module location, which of the Munich chips on the module, and the transfer operation(s) to be performed during the TDM time slot. See chart below.

function: `int` **`pci_tdm_special_add`**`(tdm, slot, devtyp, devnum)`
`int` **`pci_tdm_special_del`**`(tdm, slot, devtyp, devnum)`

args:
```
PCI_TDM  *tdm
int       slot
int       bus
int       devtyp
int       devnum
```

return:
```
1    Buffered TDM Map modified.
0    error indicated.
```

errors:

EINVAL	*tdm* is NULL or does not reference the TDM subsystem of a V6M6, or other arguments are invalid.
ENODEV	the specified *devtyp* (an IM2T1) is not installed at the module location specified in *devnum*.

These two functions add or delete special TDM operations for certain devices. The operations are performed during the specified TDM time *slot*. Special TDM operations do not involve reading or writing TDM data busses so they have no argument to specify a TDM bus.

The **pci_tdm_special_add** function is used to add **SHIFT** operations for the IMHDLC module. These shift data between the temporary data registers and Munich time slots.

The following arguments values are used for the IMHDLC module:

slot	The TDM subsystem slot number, with values between 1 and 128.
devtyp	The TDM device type for the IMHDLC is TDM_DEVIMHDLC. This is a macro defined in <pciutil.h>.
devnum	The device unit number encodes the module location, which of the Munich chips on the module, and the transfer operation(s) to be performed during the TDM time slot. See chart below.

The *devnum* argument of the TDM map function specifies the module location, Munich chip and the transfer operation(s) to be performed. The chart below describes the encoding.

IMHDLC TDM device unit number		
Bits 7:5 = Operation	Bit 4 = Munich Chip	Bits 3:0 = PCI Module
010 = Drive data to TDM for pci_tdm_src_add or Capture data from TDM for pci_tdm_dst_add 001 = Shift data from Munich 101 = Shift data to Munich	0 = Munich 0 1 = Munich 1	0 = A 1 = B 2 = C 3 = D 4 = E 5 = F

The following macros are defined in <pciutil.h> to simplify the encoding of the *devnum* argument.

TDM_UNIT(x)
: takes the Munich chip number, 0 or 1, and shifts it to the proper bit position.

TDM_HDLC_SHSRC
: produces a 1 in bit 5 to specify a shift operation for data coming from a Munich time slot.

TDM_HDLC_SHDST
: produces a 1 in bit 5 and in bit 7 to specify a shift operation for data going to a Munich time slot.

TDM_HDLC_DRIVE
: produces a 1 in bit 6 to specify a drive operation for TDM source connections.

TDM_HDLC_CAPT
: produces a 1 in bit 6 to specify a capture operation for TDM destination connections.

5.6.4.4 IMHDLC TDM Examples

The first example sets up a transfer from TDM slots 8 and 12 to Munich slots 2 and 3 for the Munich chip 0 on module D. Note the overlapped shift and capture operations during TDM time slot 12.

```
int  mun_mod = 3;      /* module site d */
int  mun_chip = 0;     /* munich chip 0 */

pci_tdm_dst_add(tdm, 8, TDM_BUSA, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_IMHDLC_CAPT);
pci_tdm_special_add (tdm, 9, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
pci_tdm_special_add (tdm, 10, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
pci_tdm_special_add (tdm, 11, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
pci_tdm_special_add (tdm, 12, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
pci_tdm_dst_add(tdm, 12, TDM_BUSA, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_IMHDLC_CAPT);
pci_tdm_special_add (tdm, 13, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
pci_tdm_special_add (tdm, 14, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
pci_tdm_special_add (tdm, 15, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
pci_tdm_special_add (tdm, 16, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHDST);
```

The second example sets up a transfer from Munich slot 0 for Munich chip 0 on module D to TDM slot 15. Note that no other TDM destination operations may be performed on this Munich chip between TDM slots 4 and 15.

```
pci_tdm_special_add (tdm, 1, TDM_DEVIMHDLC,
      mn_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHSRC);
pci_tdm_special_add (tdm, 2, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHSRC);
pci_tdm_special_add (tdm, 3, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHSRC);
pci_tdm_special_add (tdm, 4, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_HDLC_SHSRC);
pci_tdm_src_add(tdm, 15, TDM_BUSA, TDM_DEVIMHDLC,
      mun_mod | TDM_UNIT (mun_chip) | TDM_IMHDLC_DRIV);
```

5.6.5 Support Software

Host programs are provided for V6M6 and IMHDLC support and diagnostics as part of the V6M6 software distribution.

5.6.5.1 Utility Programs

The following programs, in **$CAC/bin**, are provided for the V6M6 PCI carrier boards. They are useful for initializing and maintaining the IM2T1 module. Detailed descriptions of these programs are available in Appendix B or online in UNIX man file format provided that you followed the setup procedures in Section 2.3.2 Programmer and User Setup.

pciinit	Initializes the baseboard and modules. Instructs the on-board microprocessor to configure FPGAs on the module. It also sets up PCI address mappings.
pciinfo	Examines the EEROMs on the baseboard and modules and the flash memory. It then displays all pertinent version and serial number information.
pciflashup	Compares flash object (FPGA configurations and microprocessor program) versions in the flash memory with those residing in host files. Depending on the command line options it will either display the version numbers or update the flash memory with any new versions on disk.

5.6.5.2 Diagnostic programs

The following diagnostic programs are being developed for the IMHDLC module. These programs will be distributed in the directory, $CAC/pci/diag.

pcimemslice	Runs a memory diagnostic with the host and all processors found on the baseboard such that each processor is testing a slice of global memory and local memory on all the other modules. The memory on IMHDLC modules is included in the test.
pcihdlc	Runs serial data diagnostics on the IMHDLC Munich chips. A known data pattern is loaded into Munich memory and transferred onto the TDM subsystem. The TDM data is looped back into the Munich and compared with the original pattern.
pciburn	Runs the set of diagnostics on all V6M6 boards installed in the system or a specified list of boards. It logs the tests that are run and any errors reported by the diagnostics.

10. Host MIPS Kernel Server

10.1 Introduction

The host programs, **mipsrv** and **pcisrv**, initialize the MIPS processors, load the kernel and provide access to host resources for applications running on a MIPS processor board. Users working with the mini-PCI modules on the V6M6 will use **pcisrv** and those using the R3081 mezzanine board will use **mipsrv**. When this document uses the name **pcisrv**, it is referring to both programs.

10.2 Host Server

The purpose of the server is to allow a user that developed a program on a host UNIX system to recompile that code and then execute it on the mezzanine board or a MIPS processor module Server Execution.

10.2.1 Server Execution

The server is started by typing **pcisrv** at the UNIX prompt. Each MIPS processor on a board requires its own server. The program accepts the following command line options:

-b	Boot the kernel on this processor. (PCI only)
-i	Do not initialize the other PCI modules. (PCI only)
-f *n*	Set maximum number of open UNIX files to *n.*
-t *n*	Set tracing level to *n*.
-lm	Load the monitor and debugging kernel into the MIPS board.
-k *file.s3*	Use file.s3 as the alternate kernel file.
-m *file.s3*	Use file.s3 as the alternate monitor file.
-u	Connect to processor without initializing or loading a kernel.
dspdev	The name of the resource, e.g., *dsp0g, dsp0h, pci0a, pci0b*, that this instance of the server will control.

The default *dspdev* for the R3081 mezzanine board is *dsp0g* and the default *dspdev* for the V6M6 is *pci0c*. The **-b** option must be used when starting the kernel on the processor that will boot the board and is a required option for each board in the system. The **-b** and **-i** options apply only to the mini-PCI modules. Multiple occurrences of the **-b** option can corrupt global memory. At this time there is no mechanism to prevent this from happening.

10.2.2 Server Support

The server requires the daemon, **socktable**. This program will be started automatically by **pcisrv** or can be started from the command line or system startup file.

socktable manages global tables of sockets. Each MIPS processor is under the control of a separate instance of pcisrv. By using global memory, **socktable** optimizes socket communication between programs running on different MIPS processors on the same board. Socket communication between programs on different boards will use the IPC channel established by the host.

10.2.3 Functions

Two types of messages are passed between the MIPS kernel and the host.

- File messages are used for file management, I/O and sockets.

  File messages provide host support for applications running on the MIPS processor. The following operations are available for use in application programs.

chdir	chmod	close
creat	dup/dup2	fstat
getenv	link	lseek
mkdir	open	read
rename	rmdir	select
stat	tell	time
truncate	unlink	write

Many commonly used functions in the WM Kernel and pSOS Application Libraries are based on this basic set of operations. For example, **printf**( ), **puts**( ), and **fwrite**( ) are handled by the **write** operation. Similarly, **scanf**(), **gets**(), **fread**( ) and other input functions are handled by the **read** operation.

The server also supports the following socket operations.

accept	bind	connect
listen	socket	

Other functions, such as **socketpair**( ), are implemented by the WM Application Library using these basic host functions.

For a list of available functions see the *WM Kernel User's Guide* or the *WM pSOS User's Guide*.

- Control messages are used for process management and are not available to user applications.

10.3 User Interface

The **pcisrv** program accepts two types of commands:

- Control messages that are transmitted to the kernel running on the MIPS processor. This includes commands that can obtain MIPS information without support from the MIPS kernel.

- Status messages are useful for server debugging and obtaining state information about the server.

When **pcisrv** is started using the default configuration, the kernel commands will be disabled. When these commands are active, **pcisrv** will display the message

```
    Server commands enabled
```

The easiest way to accomplish this is to rename **root.s3** to **Root.s3** before starting **pcisrv**. Once the server is running you can type

```
        run Root
```

This will enable kernel commands and place the MIPS board in its default configuration.

10.3.1 MIPS Kernel Commands

The following commands provided for status and management of MIPS programs. It is sufficient to type the first two letters of the command.

run *filename*	Run filename on the MIPS processor. The server first looks for the file in the current directory and then in the list of directories specified by the environment variable **S3MIPSPATH**. The file must be in Motorola S Record format. If the user enters a filename that does not have an extension, the server will append an ".s3" to the name. The filename argument may be entered with a full or relative path. The list of directories in **S3MIPSPATH** is delimited by colons.
list	List the processes running on the MIPS board. The first two processes belong to the kernel. Process 0 is the idle loop and process 1 is the task requester. The task requester will either be **root.s3** or, in its absence, one built into the kernel. The first user task will be process 2.
abort	Print register dump of the MIPS processor if the kernel crashes.
kill *n*	Kill a MIPS process. Similar to the abort but without the register dump.
dump *n*	Dump the MIPS registers associated with process *n*.

10.3.2 Server Status Commands

The commands listed here are intended for debugging purposes only.

tr [*n*]	Set the tracing level to *n*. The server disables tracing if *n* is zero or is not provided. The tracing is organized so that each bit in *n* corresponds to a different type of message.
ex, qu	Exit the server.
pr *n*	Display the information that the server maintains for a process that is running on the MIPS board.
qs	Print statistics about queue usage.
hv	Download and print the host vector from the MIPS kernel.
mt	Display an area of MIPS memory that has been set aside for trace messages. This area can only be used for messages that are written by the kernel.
ct	Clear the area in MIPS memory where kernel trace messages are written.

10.4 Caveats

There are several limitations in the current implementation. These are:

- The initial current working directory of a MIPS process is the current working directory of the process that executed **pcisrv**. An application can change its current directory by using the **chdir**( ) function. The server will maintain a separate working directory for each process running on the board.

- All files that are read or written by a MIPS application are, in reality, accessed by the server. This means that the limit on the number of open files in a single host process is shared by all MIPS programs. Programs that use a large number of files or programmers that do not balance a “file open” with a “file close” can cause problems for every task running on the processor. The MIPS kernel will close all open files used by a single process when that process terminates.

- Because the standard files (**STDIN, STDOUT,** and **STDERR)** are common to all MIPS programs, the output of functions such as **printf**( ) and **puts**( ) will appear on the terminal that was used to start the server. The messages may be inter-mixed with the output of other tasks. Since the software does not attempt to prevent this, a single line of output may have characters from two or more programs. Applications that are started from **ushell** will not behave in this manner. For a description of the **ushell** program, see the *WM Kernel User’s Guide*.

- The implementation of **pcisrv** and the MIPS kernel precludes the use of the environment variable _**CACDEVICE**. This variable informs the kernel what type of board is being used. A R3081 mezzanine application that uses this string as an argument to **getenv**( ) will get a pointer to the string **dsp** and a V6M6 application will get a pointer to the string **pci.** This is regardless of how the user sets it on the host or whether it is set at all.