ALACRON

FASTSERIES

ALRT RUNTIME SOFTWARE
PROGRAMMER'S GUIDE AND REFERENCE
USER'S MANUAL

FAST CAPTURE
FAST PROCESSING
FAST RESULTS

FASTSERIES PCI BOARD	*FAST SERIES* PMC
FastVision	FastMem
FastImage 1300	Fast4 1300
FastFrame 1300	Fast I/O 1300

30002-00169

Document Name:	ALRT RT SW Programmer’s Guide & Reference User’s Manual	
Document Number:	30002-00169	
Revision History:	1.2	June 13, 2002

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
Fax: 603-891-2745

Web Site:
http://www.alacron.com/

Email:
sales@alacron.com, or support@alacron.com

TABLE OF CONTENTS

MANUAL FIGURES & TABLES

FIGURE	PAGE	SUBJECT	TABLE	PAGE	SUBJECT
1	1	Trimedia Program Development & Execution on FastSeries			
2	5	The ALRT Execution Environment			

<u>OTHER ALACRON MANUALS</u>

Alacron manuals cover all aspects of FastSeries hardware and software installation and operation. Call Alacron at 603-891-2750 and ask for the appropriate manuals from the list below if they did not come in your FastSeries shipment.

• 30002-00148	ALFAST Runtime Software Programmer's Guide & Reference
• 30002-00150	FastSeries Library User's Manual
• 30002-00153	Fast I/O Hardware User's Manual
• 30002-00155	FastMem Hardware User's Manual
• 30002-00162	FOIL – **F**astSeries **O**bject **I**maging **L**ibrary User's Manual
• 30002-00169	ALRT Runtime Software Programmer's Guide & Reference
• 30002-00170	ALRT, ALFAST & FASTLIB Software Installation Manual for Linux
• 30002-00171	ALRT, ALFAST, & FASTLIB Software Installation for Windows NT
• 30002-00173	FastMem Programmer's Guide & Reference
• 30002-00176	FastImage 1300 Hardware User's Manual
• 30002-00180	Fast4 1300 Hardware User's Manual
• 30002-00184	FastSeries Getting Started Manual
• 30002-00183	FastImage 1300 Camera Integration User's Manual
• 30002-00185	FastVision Hardware User's Manual
• 30002-00186	FastVision Software User's Manual
• 30002-00187	FastFrame 1300 Hardware User's Manual

I. INTRODUCTION

A. Developing TriMedia Programs for FastSeries Boards

Figure 1 diagrams the steps in application development and execution for Alacron's FastSeries boards.

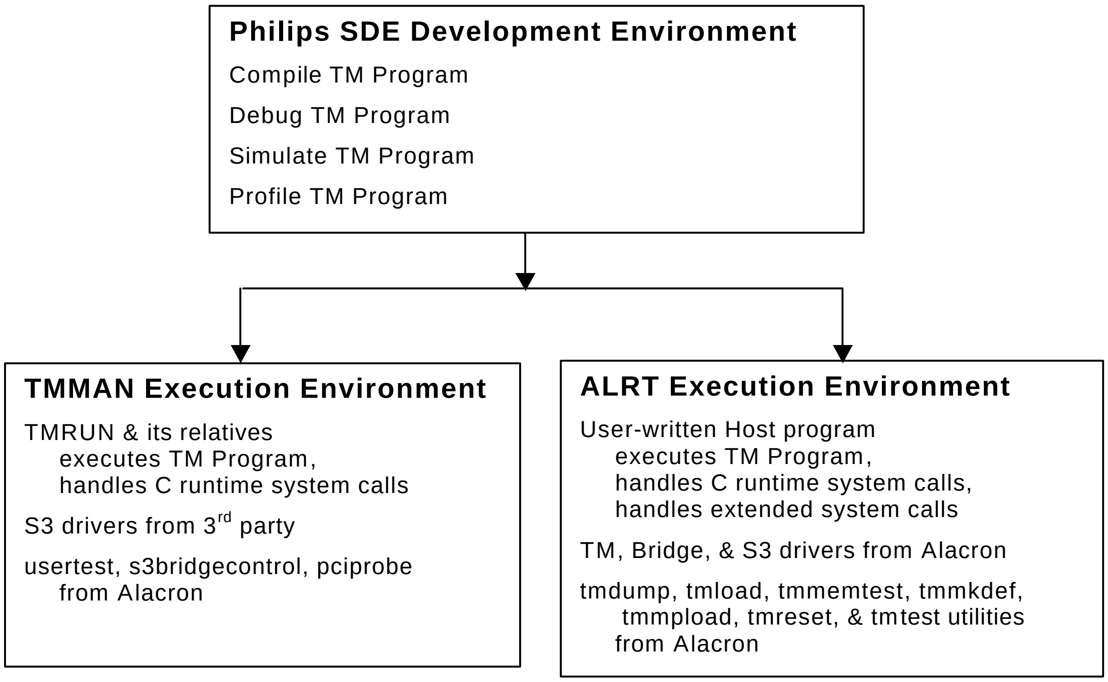
Philips SDE Development Environment
Compile TM Program
Debug TM Program
Simulate TM Program
Profile TM Program
TMMAN Execution Environment
TMRUN & its relatives executes TM Program, handles C runtime system calls
S3 drivers from 3rd party
usertest, s3bridgecontrol, pciprobe from Alacron
ALRT Execution Environment
User-written Host program executes TM Program, handles C runtime system calls, handles extended system calls
TM, Bridge, & S3 drivers from Alacron
tmdump, tmload, tmmemtest, tmmkdef, tmmpload, tmreset, & tmtest utilities from Alacron

Figure 1. TriMedia Program Development and Execution on FastSeries

As diagrammed in Figure 1, all applications use the Software Development Environment (SDE) from Philips TriMedia Corporation to compile and debug TriMedia programs for the FastSeries boards. SDE is supported on several operating systems, and the TriMedia application can be compiled and simulated on any supported system.

Normally, applications are developed in SDE and then executed in the Philips TMMAN execution environment. However, the Alacron FastSeries board can run under operating systems where TMMAN is not supported. You can use the TMMAN execution environment with FastSeries boards under Windows/NT, for example.

Under operating systems not supported by SDE, you use Alacron's ALRT Runtime Environment instead of TMMAN. ALRT was designed specifically for use with the FastSeries products, and provides features and utilities not available from TMMAN. You may prefer ALRT in systems with large numbers of TriMedia processors.

B. Software Development with Philips SDE

The Philips SDE contains utilities to develop, compile, simulate, debug, and profile TriMedia application programs. The listing here may differ in detail from the Philips release you have. Refer to the documentation on the CD ROM with the Philips SDE software for complete information.

1. Program Development

Program development tools from Philips include the following:

- **tmcc**, TriMedia C and C++ compiler driver. To compile and link a C program, **tmcc** runs the C preprocessor **cpp**, the core C compiler **tmccom**, the instruction scheduler **tmsched**, the assembler **tmas**, and the linker **tmld**. Optionally, **tmcc** can strip the final executable with **tmstrip**. To compile and link a C++ program, **tmcc** runs the C++ front end **tmcfe** rather than **cpp**, and links with the C++ library **libC++.a** in addition to the standard C library **libC.a**.
- **cpp**, GNU C-compatible compiler preprocessor. **cpp** is the standard macro preprocessor, providing header files, macro expansion, conditional compilation, and line control. **cpp** is invoked by the **tmcc** program automatically, and is not normally executed directly.
- **tmccom**, TriMedia compiler. **tmcom** reads a single preprocessed C source file and writes an output file containing an intermediate code representation of the file as decision trees. **tmcom** is invoked by the **tmcc** program automatically, and is not normally executed directly.
- **tmsched**, TriMedia instruction scheduler. **tmsched** takes the intermediate format file produced by **tmcom** and a machine description file provided by the user, and generates an assembly-language file containing the scheduled instructions. **tmsched** is invoked by the **tmcc** program automatically, and is not normally executed directly.
- **tmas**, TriMedia assembler. **tmas** assembles TriMedia assembly code for specific processor descriptions, and creates object files that can be simulated using **tmsim**. **tmas** is invoked by the **tmcc** program automatically, and is not normally executed directly.
- **tmld**, TriMedia program linker. **tmld** links the various types of executable files. **tmld** is invoked by the **tmcc** program automatically, and is not normally executed directly.
- **tmcfe**, TriMedia C++ front end. **tmcfe** reads a C++ source file and expands it into a C source file on the standard output. **tmcfe** is invoked by the **tmcc** program automatically, and is not normally executed directly.
- **tmsize** prints the size in bytes of a TriMedia object file.
- **tmcomp**, TriMedia object file compressor. **tmcomp** compresses the text section of an executable TriMedia object file. The resulting text section has a memory width of 8 bits as opposed to 344 bits for the uncompressed file. tmcomp updates the reference tables and the symbol tables of the output file to reflect the new address values. The compressed instruction format is Philips proprietary.
- **tmstrip**, removes symbol information from a TriMedia object file. The stripped file retains only the symbols required by the simulator **tmsim** for simulated execution of an executable object file, or required by the host downloader for successful downloading and execution of the program on a TriMedia.

- **tmar**, TriMedia archive librarian. **tmar** builds libraries of TriMedia object files, prints the contents of a library, deletes or replaces modules in a library, extracts modules from a library, and prints modules from a library.
- **tmlib**, TriMedia format library builder. **tmlib** builds a TriMedia format library from a set of input files.
- **tmnm**, prints symbol table from TriMedia object file or library. The list can be sorted alphabetically or otherwise, and the symbols can be selected.
- **tmranlib**, builds symbol table for TriMedia intermediate file archive.

2. Program Simulation, Debug, and Profiling

Philips utilities for simulating, debugging, and profiling the application include:

- **tmsim**, TriMedia cycle accurate machine level simulator. **tmsim** simulates the DSPCPU as described in the input machine description file, including the PCSW, DPC, SPC, and CCOUNT registers. **tmsim** also simulates MMIO space, memory and cache control registers, the vectored interrupt controller, four timers and debug support (instruction and data breakpoints). Audio in/out, video in/out, ssi, jtag, vld, and icp peripherals are simulated only when enabled through a command line option. **tmsim** also provides operating system support, catches and reports exceptions, and includes commands useful for debugging.
- **tmcanal**, TriMedia cache analysis tool. **tmcanal** reads a tracefile produced by the simulator **tmsim** and provides a graphic interface to view the cache behavior during program execution.
- **tmdbg**, TriMedia source-level debugger. **tmdbg** provides GUI or command-based debugging of a TriMedia executable program that has been compiled with the **–g** option to **tmcc**. The debug symbol table generated with the **–g** option contains the names of all the source files (which can be browsed in the debugger) and extensive debugging information. **tmdbg** also includes commands for debugging pSOS+ applications; these require the pSOS+ monitor to be linked into the application.
- **tmdump**, dump TriMedia format object modules in readable format. **tmdump** dumps a specified portion of a file. **tmdump** displays binary and string data, string identifiers, and section (segment) names alongside numeric identifiers.
- **tmprof**, generate estimated execution profile. The default tracefile, named **mon.out**, is generated either by running a program compiled with the **–ptm** or **–lprof** option to **tmcc**, or by simulating a program using the **–statfile** option to **tmsim**.
- **tmdtprof**, TriMedia profile information lister. **tmdtprof** generates an ASCII readable form of the profile information contained in its input file. The profile data is typically in the file **dtprof.out** generated by the execution of a program compiled with the **–p** option to **tmcc**.

C. TMMAN Execution Environment

The Philips TMMAN execution environment supplies drivers, libraries, and utilities for loading and executing TriMedia programs.

1. TMMAN Drivers and Libraries

The Philips TMMAN execution environment drivers and libraries for loading and executing TriMedia programs include:

- **TMMan.sys**, kernel mode driver. **TMMan.sys** provides the TMMan functionality, including support for multiple boards.
- **TMMan32.dll**, User Mode Win32 DLL that provides the TMMan Host API to Win32 applications
- **TMMan.a**, the target component of TMMan. This static library links to boot applications on the TriMedia. It provides the TMMan functionality on the target.
- **TMCRT.dll**, TriMedia C Runtime Server. This module accepts requests from the target and services them. These requests are Unix level 2 I/O calls that are generated by the TriMedia executable. It uses TMMan messaging to communicate with the target.
- **Driver.exe**, a helper utility for installing the kernel mode driver in the system. Needed only for install and uninstall operations.

2. TMMAN Utilities

The TMMAN environment supplies utilities for loading and executing TriMedia programs including:

- **TMMon**, command-based interface for executing programs on the TriMedia processor. Functions via calls to **TMMan** programs, and by default uses **TMRun** as its console.
- **TMRun**, command line utility for downloading and running executables on the TriMedia processor. Used by **TMMon** as the TriMedia console.
- **TMmpRun**, multiprocessor version of TMRun. Enables multiprocessor cluster downloading on multiple TriMedia boards in the system.

3. Philips Documentation

Documentation on program development with Philips SDE and execution with the TMMAN environment is provided on the CD-ROM from Philips North America.

4. S3 Driver

The Alacron FastSeries boards use an S3 ViRGE GX2 device for video output. The driver for this component must be obtained from the vendor, S3 Corporation; neither Alacron nor Philips supplies a driver for this device under TMMAN.

Instructions for configuring the S3 driver for FastSeries are given in the *ALFAST Programmers Guide & Reference*.

5. Alacron TMMAN Utilities

To enable the FastSeries boards to perform optimally in the TMMAN environment, Alacron supplies three utility programs that run under TMMAN. They are:

- **usertest**, a test program that verifies the correct installation of the hardware and software.
- **s3bridgecontrol**, a program that conditions the PCI bridge and S3 devices on the FastSeries board.
- **pciprobe**, a device driver for the PCI bridge device on the FastSeries board

Instructions for running **usertest** are given in the installation manuals (both HW and SW).

Instructions for using **s3bridgecontrol** and **pciprobe** are given in the *ALFAST Programmers Guide & Reference*.

D. The ALRT Runtime Environment

The Alacron Runtime (ALRT) software supports execution of a TriMedia program on an Alacron FastSeries board under operating systems where the Philips SDE components are unavailable. ALRT consists of four components: the Host Runtime software, the TriMedia Runtime software, the Device Driver, and a set of utilities parallel to TMRUN. This manual documents the ALRT components.

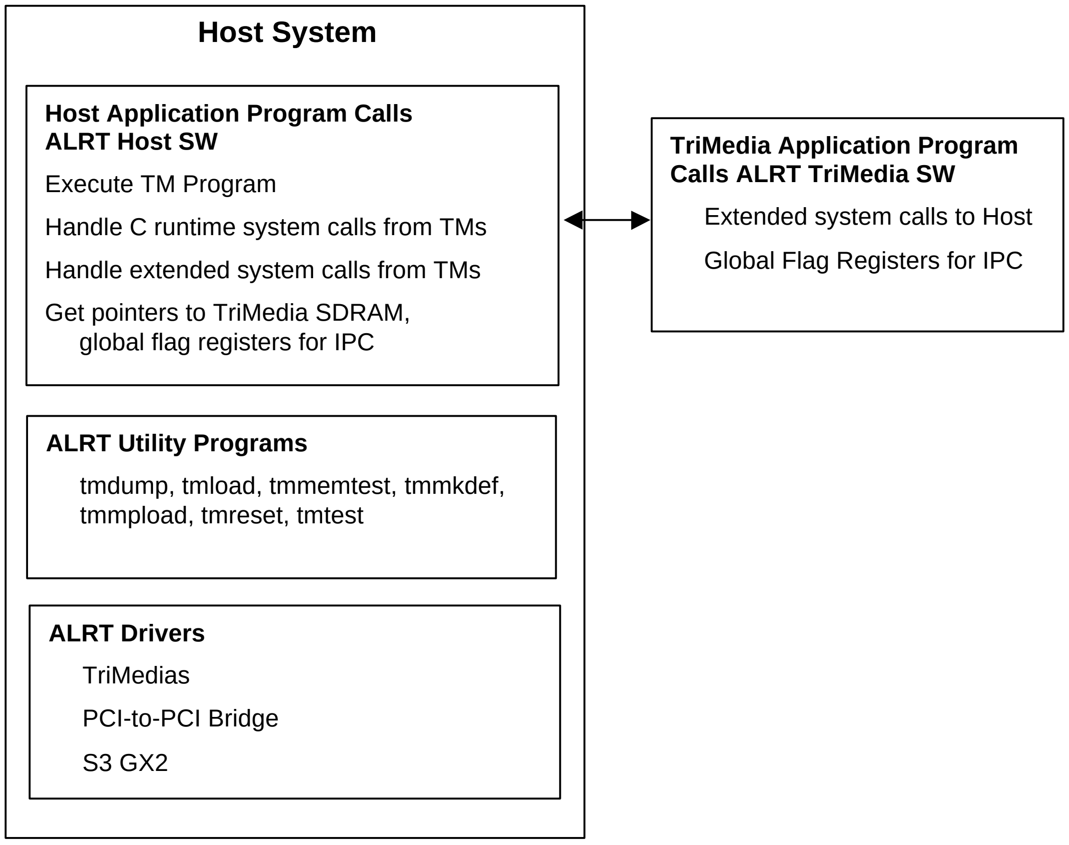
Host System
Host Application Program Calls ALRT Host SW
Execute TM Program
Handle C runtime system calls from TMs
Handle extended system calls from TMs
Get pointers to TriMedia SDRAM, global flag registers for IPC
ALRT Utility Programs
tmdump, tmload, tmmemtest, tmmkdef, tmmpload, tmreset, tmtest
ALRT Drivers
TriMedias
PCI-to-PCI Bridge
S3 GX2
TriMedia Application Program Calls ALRT TriMedia SW
Extended system calls to Host
Global Flag Registers for IPC

Figure 2. The ALRT Execution Environment

1. ALRT Host Software

The ALRT Host Software provides a Host API for the Alacron FastSeries boards operating without Philips TMMAN. The Host software consists of a host device and a Host application library.

a) Host Device

The Host software opens a single instance of the device driver to access all the TriMedia processors in the system.

b) Host Library Functions

The Host application library is provided as a library called **alfast.a**. Applications must include the file **allib.h.**Included library functions are:

Aladdr	Retrieve the address of a target program variable
Alclose	Close a device
Aldev	Select current device
Alflagregs	Return pointer to flag registers
alget*	Read byte/word/long single element or array
Almapload	Load a program file on to a processor
Almapsdram	Return pointer to SDRAM
Alopen	Open a device (processor)
Alputargs	Pass argc/argv arguments to target processor
alset*	Write byte/word/long single element or array
Alsyscall	Process system calls from a target processor
Alsyscallext	Process system call, with extensions
alsyscallserver	Spin off a system call server thread
alsyscallserver_stop	Stop a system call server thread
alsyscallserver_wait	Suspend program execution until thread completes
VtoP	Convert target virtual to physical address

c) Driver Interface Functions

A set of operating system independent device driver interface functions are provided; these are called from within the Host application program. These functions are for program development rather than production code.

Open_alfast	open device
Close_alfast	close device
Seek_alfast	set read/write seek address
Read_alfast	read from SDRAM
Write_alfast	write to SDRAM
ioctl_alfast	perform I/O control function
Dev_alfast	return currently selected device

These functions provide lower level access to the underlying device driver. The functions are documented in a Reference section but (except for **ioctl_alfast**) are not recommended for use by applications. The functions **alopen, alclose**, **alget***, **alset***, and **almapsdram** perform the same functions. The **ioctl_alfast** function provides a number of useful functions including determination of device configuration, waiting on system call request interrupts, reading and writing MMIO space registers, and accessing mapped device memory.

d) Utility Programs

In addition to the functions that are called from within a Host program, a number of OS-level executable utility programs are provided:

tmload	Load a TM program
tmdump	Display TM's MMIO registers
memtest	Run quick memory test on a TM's SDRAM
tmmkdef	List TM's I/O mappings
tmreset	Reset TM
tmmpload	Load multiple TM programs

The source code to **tmload** is provided in the Examples section of this manual and as sample code in the directory **examples/host/tmload**.

e) TriMedia Program Execution

Target (TriMedia) programs begin execution immediately at entry point `main` following loading. The ALRT Host library has no calls to begin execution. Since the target code never terminates, there are no return value or status functions.

f) Mapped Virtual Memory

The Host device library permits the direct mapping of TriMedia SDRAM into the Host program's virtual address space. The application can dereference pointers that point to the physical memory containing the program and data of the TriMedia processor. The functions **almapsdram** and **alflagregs** both return pointers to mapped SDRAM. **almapsdram** returns a pointer to a specific address in SDRAM. The address may be derived by using **aladdr** to compute a symbol value, or by having the TriMedia pass address information to the host at runtime (using flag registers or extended system calls).

g) Flag Registers

Ten contiguous 32 bit uncached memory locations are reserved for application use. These locations are referred to as the *flag registers*. The host software can use **alflagregs** to establish a mapped address that may be used to access these locations. The flag registers are initialized to zero when a TriMedia program is loaded using **almapload**.

h) Thread Functions for System Call Service

The host library provides a simple set of thread control functions to service system call requests. The host function **alsyscallserver** starts a thread. The application might then use **alsyscallserver_wait** to wait on completion of this thread. When the thread is no longer needed, it is terminated with **alsyscallserver_stop**.

i) Host Resident DMA Buffer

The ALFAST runtime provides an API call for allocating a Host resident DMA buffer. The function call specifies the size in bytes of the buffer to be allocated. A second function maps the DMA buffer into user space. A third call frees the DMA buffer and the mapping. During TM program load, the DMA buffer information is exported to a global structure in the TriMedia processor's memory. The TM processor may directly reference the buffer addresses, or perform PCI DMA to the buffer.

2. ALRT TriMedia Library

The ALRT TriMedia library provides an application programming interface (API) to be used by the TriMedia processors in the Alacron Runtime environment for non Philips SDE supported hosts. The ALRT TriMedia library is provided as part of the ALFAST runntime library **libalfast.a**. The standard ALFAST include file (**alfast_tm.h**) should be included in C application source code to provide all declarations required for use of the library. The functionality added for the ALRT environment is as follows:

- Send extended system call to Host
- Communicate with other TriMedias via global flags provided by the driver.

a) Extended System Calls

A library function **tmfast_syscall** is provided to allow a TM program to communicate with a host extended system call handler. The **tmfast_syscall** function takes a ***cmd*** argument which is an integer code. Since the same mechanism is used for C runtime support, the ***cmd*** code must be unused by existing functions. ***cmd*** values greater than or equal to 2000 (decimal) may be used; a predefined token **SYSCALL_EXT_START** may be used by both Host and TriMedia programs as a base for extended system call codes.

b) Flag Registers

Each TM processor has a region of global shared memory. This memory is uncached, and accessible to all TM processors, as well as the host. It is available for application specific use. The function **tmfast_flagregs** returns a pointer to the flag registers on the local TM; the function **tmfast_flagregs_m** returns a pointer to the flag registers on other TM processors.

c) Memory Management

The TriMedia application is responsible for all memory management. Standard TM library calls (**malloc**, **free**, etc.) are available for dynamic allocation.

d) C Runtime Support

The ALRT TriMedia Runtime library supports a subset of C runtime functions that require Host action: **open**, **close**, **fstat**, **stat**, **isatty**, **read**, **write**, **lseek**, **time**, and **exit**.

C runtime functions that do not require Host support are fully supported (e.g., **strcpy**).

3. ALRT Device Driver

The ALRT Device Driver operates only with the Alacron Fast Series PCI boards. The FastSeries boards have one or more TriMedia processors, an S3/ViRGE display, and a PCI-to-PCI bridge device, each appearing as a distinct physical PCI device.

The ALRT driver probes and attaches to all three devices.

Access to the TM processors is via kernel mapping of the 64 KB MMIO register space for each TM, kernel mapping of a 64 KB SDRAM communications buffer for each TM, user mapping of 64 KB MMIO register space, user mapping of all SDRAM (8/16 MB), kernel access to PCI configuration space registers, handling TM to Host interrupts in the kernel, asignaling waiting user threads using condition variables, and allocation and mapping of kernel resident DMA buffers for TM to host data transfer. The number and size of buffers is specified at driver load time.

Access to the S3/Virge display device consists of user mapping of video display memory (4 MB) and user mapping of control register space (64KB). No support is provided for S3 to host interrupts.

Access to the PCI to PCI bridge consists of reading and writing PCI configuration registers for possible reconfiguration.

The ALRT device driver is implemented as separate drivers, one for the TMs, one for the S3 and one for the bridge. The ALRT device driver supports multiple FastSeries boards. The device driver supports multiple thread access, with the restriction that only one thread is allowed to wait on the TM interrupt event. If multiple threads wait on TM interrupts, unpredictable results occur.

II. PROGRAMMER'S GUIDE

This Programmer's Guide shows how the Host program controls the TriMedia program via the ALRT Host software, and how the TriMedia program performs IPC with the Host and other TriMedias via the ALRT TriMedia Runtime software

A. ALRT Host Program Functions

The Host program in the ALRT environment controls the TriMedia program via calls to **alfast.a** library functions. To access the library, the Host program includes header file **allib.h**.

1. Open TriMedia Processors

The Host software opens a single instance of the ALRT Device Driver to access up to eight TriMedia processors in the system. Processors are numbered from 0 to NPROCS–1. Processor 0 must be the master TriMedia on the board; only the master processor programs the PAL devices on the board.

To open each of the processors in turn, the Host program makes calls to:

int **alopen**(int *proc*)

The function returns SUCCESS (0) for successful open, FAILURE (-1) otherwise.

The Host program can open processor 0 through 7 as follows:

```
#include allib.h
#define NPROCS 8
int i, ret;
for (i=0; i<NPROCS; i++) {
       ret = alopen (i);
       if (ret){
       printf("error opening proc %d.\n", i);
       exit(1);
       }
}
```

2. Close TriMedia Processors

When the Host program is preparing to exit, it closes each processor with calls to:

int **alclose**(int *proc*)

3. Select Active Processor

The calls to **ioctl_alfast** (in the previous section) and to **alsyscall** (at the end of this section on Host software functions) have *proc* arguments to specify which processor to use. The remaining Host software functions do not contain a *proc* argument, but instead operate on the one processor that is currently *active*. The Host program selects the active processor with a call to:

int **aldev** (int *proc*)

4. Pass Command-Line Arguments to Processors

The Host program can pass **argc**/**argv** command-line parameters to the target processor. The Host stores the array of arguments pointed to by **argv** in Host system non-volatile memory (not on the stack). Then int **alputargs** (int *argc*, char **argv*[])

The call must be made after device open (**alopen**) but prior to invoking **almapload** for the target processor. The array of data referenced by the ***argv*** parameter must remain present in memory until after the **almapload** call. The arguments are passed to the target processor using the system call service mechanism, so the Host application must enter a wait for system call loop to utilize this facility.

Here is an example:

```
int main (int argc, char *argv[])
{
int proc = 0;
char *my_argv[argc-1];
int i;
/* copy all but first argument */
for (i=1; i<argc; i++) {
my_argv[i-1] = argv[i];
       }
if (alopen (proc)) error ();
aldev (proc);
/* pass all but first argument */
alputargs (argc – 1, my_argv);
if (almapload (infile, 0)) error ();
/* begin waiting for system calls */
```

5. Load and Execute TriMedia Programs

To load the program into each TriMedia and begin execution, the Host program calls:

int **almapload** (char **filename*, int *stacksize*)

Function **almapload** loads the processor board executable file given by *filename* into TriMedia memory. Argument *filename* should be a file residing within the Host file system. The file should be an executable image built using compiler tools specifically designed for the TriMedia. The *stacksize* argument is ignored.

Upon completion of **almapload**, the on-board processor reset is deasserted and program execution begins. On the TriMedia processor, execution begins at function `main`.

Processor 0 is the master TriMedia on the board. The master processor programs the PAL devices on the board. The program on Processor 0 must be loaded and executed before any other processors are activated. After processor 0, the remaining TriMedias may be started in any order.

6. Get Address of Symbol in TriMedia Program

To obtain the virtual address corrresponding to a symbol in the TriMedia program most recently loaded, the Host program calls

ADDR **aladdr** (char **name*)

The **aladdr** function returns the virtual address of the variable given by the string input argument *name*. The symbol table information in the currently loaded target executable file is examined to resolve the address.

Function **aladdr** returns a valid address for a particular processor only after a call to **almapload** referencing that processor.   Only the symbols loaded by the last call to **almapload** may be referenced by **aladdr**.

The returned value is a virtual address and should be converted to a physical address (see **VtoP**) before use in functions such as **alget[bwl][a]**, and **alset[bwl][a]**.

Note: There is significant overhead in executing the **aladdr** call. If the address of a symbol is to be used more than once, it is recommended that **aladdr** be called once and the value saved in a program variable.

7. Virtual and Physical Addresses

The TM processor uses physical addresses to access its two memory resources, the MMIO space and the SDRAM space. The physical locations of these resources are assigned by the BIOS or OS as part of the PCI bus configuration. References to TM program code and data addresses in SDRAM are adjusted to the physical PCI addresses when each TM is loaded using **almapload**.

In the ALRT library, **aladdr** returns as a virtual address the offset from start of SDRAM—the variable's address with the program image relocated to a TM with SDRAM located at PCI address 0. To convert from the virtual address returned by **aladdr** to the physical address used by Host calls to **alget** and **alset**, the program calls **VtoP** (see below). The physical address is the actual PCI address.

A problem can arise when the TM processor passes a target virtual address to the Host. In this library, a target virtual address that the TM processor supplies must be treated as a physical address, and not ever passed to **VtoP**. **VtoP** should be used only for addresses that come from **aladdr**.

Note: **aladdr** returns addresses relative to the base of SDRAM. To compute a physical address, the application would add the SDRAM base address, obtained using **ioctl_alfast (IOCTL_GET_SDRAM_BASE)**. This works unless the symbol being looked up has been reassigned to uncached memory (using the **tmld –sectionproperty data=uncached** directive). In this case, the actual address of the symbol is relocated to high SDRAM addresses, and the value returned by **aladdr** is not valid. The address must be passed at runtime.

TriMedia caching of SDRAM is another concern. Unless explicitly linked to an uncached section, data used by the TriMedia programs are cached. To enable the host to see data written to SDRAM by the TriMedia, a cache write-back must be executed by the TriMedia (see TriMedia SDE function **_cache_copyback**). Similarly, to enable the TriMedia to see data written to SDRAM by the host, a cache invalidate must be executed (using **_cache_invalidate**)

8. Convert Virtual Address to Physical Address

To convert virtual addresses returned by **aladdr** to physical addresses for the **alget** and **alset** functions, the Host program calls:

ADDR **VtoP** (ADDR *vaddr*)

The address conversion is performed on the processor that was selected by the **aldev** function.

9. Read Values from TriMedia Memory

The Host program can read a single value or an array of values from TriMedia memory using the physical address. The memory transfer is performed on the processor that is selected by the **aldev** function.The calls have formats that depend on the type of value to be returned,

int **algetb** (ADDR *physadd*)   // Read 8-bit value

int **algetw** (ADDR *physadd*)	// Read 16-bit value
long **algetl** (ADDR *physadd*)	// Read 32-bit value
int **algetba** (ADDR *physadd*, char **buf*, int *n*)	// Read array of 8-bit values
int **algetwa** (ADDR *physadd*, short **buf*, int *n*)	// Read array of 16-bit values
long **algetla** (ADDR *physadd*, long **buf*, int *n*)	// Read array of 32-bit values

The **alget** functions transfer data from the processor memory buffer to a host buffer. Buffers shared by the host and target processors should be uncached.

The **algetba**, **algetwa**, and **algetla** functions read arrays of bytes, 16-bit words or 32-bit longwords from the processor memory buffer specified by argument *physadd* into the host buffer specified by argument *buf.* The size of both buffers must be at least *n* bytes, words, or longwords as appropriate.

The **algetb**, **algetw**, and **algetl** functions read a single byte, word or longword from processor memory into Host memory. The value read from processor physical address *physadd* is the function return value.

10. Write Values to TriMedia Memory

The Host program can write a single value or an array of values to TriMedia memory using the physical address. The memory transfer is performed on the processor that was selected by the **aldev** function. The calls have formats that depend on the type of value to be written.

int **alsetb** (ADDR *physadd*, int *value*)	// Write 8-bit value
int **alsetw** (ADDR *physadd*, int *value*)	// Write 8-bit value
long **alsetl** (ADDR *physadd*, long *value*)	// Write 8-bit value
int **alsetba** (ADDR *physadd*, char **buf*, int *n*)	// Write array of 16-bit values
int **alsetwa** (ADDR *physadd*, short **buf*, int *n*)	// Write array of 16-bit values
long **alsetla** (ADDR *physadd*, long **buf*, int *n*)	// Write array of 16-bit values

The **alset** functions transfer data from host memory to processor memory.

The **alsetba**, **alsetwa**, and **alsetla** functions write arrays of bytes, 16-bit words or 32-bit longwords to the processor board buffer specified by argument *physadd* from the host buffer specified by argument *buf.* The size of both buffers must be at least *n* bytes, words or longwords as appropriate.

The **alsetb**, **alsetw**, and **alsetl** functions transfer a single byte, word or longword between processor board memory and host memory. The **alset** functions write the byte, word or longword in argument *value* to physical address *physadd*.

Buffers that are shared by the host and target processors should be uncached. Functions **alsetb**, **alsetw**, and **alsetl** may perform a read-modify-write in processor board memory. Data could be overwritten if user software allows the host and processor board application to modify the same buffer in processor board memory simultaneously.

11. Pointers to TriMedia SDRAM and Flag Registers

The Host program can obtain virtual memory pointers to TriMedia SDRAM locations, and to the set of global flag registers maintained by the ALRT driver for each TriMedia processor in the system.

The Host program can obtain the physical address of a location in TriMedia SDRAM in several ways. Given a physical address *physaddr* in the SDRAM of a given TriMedia (*dev*), the Host can get a virtual pointer to that address by calling:

void ***almapsdram** (int *dev*, unsigned long *physaddr*)

The ALRT driver maintains a set of global 32-bit Flag Registers for each of the TriMedia processors opened by the Host program (with **alopen**). The flag registers are located in uncached space. The Host program can get a pointer to the flag registers for a given TriMedia *dev* by calling:

volatile unsigned long ***alflagregs**(int *dev*)

12. Handle System Calls from the TriMedias

Once the program has been started on a given TriMedia, the Host program can service normal or extended system calls from that processor.

a) Wait for Interrupt

The TriMedia interrupts the Host when it has a system call pending. The Host program calls **ioctl_alfast** to halt program execution while waiting for an interrupt from a particular TriMedia:

int ioctl_alfast (int *dev*, IOCTL_INTWAIT);

The **ioctl_alfast** returns when the system call request has been received.

b) Normal System Calls

The TriMedia program initiates a normal system call (no data passed with the system call request) for any of the C runtime library functions listed above. To handle normal system calls (after waiting for the interrupt as shown earlier), the Host program calls:

Int **alsyscall** (int *proc*)

The call services any normal system call requests that are pending for processor *proc*, then returns. The function does not wait.

alsyscall returns 0 for success, non-zero if the TM program requests an **exit** call.

c) Extended System Calls

When the TM program needs to send data along with the system call, it uses TM Runtime library call **tmfast_syscall** to pass a request number (greater than **SYSCALL_EXT_START** to avoid conflict with the built-in requests) and an array of unsigned long values. The Host program handles extended system call requests (after waiting for the interrupt as shown earlier) with a call to:

int **alsyscallext** (int *proc*,

int (**callback*)(int *dev*, int *request*, unsigned long *args*[])

This function services both normal and extended system calls from the TriMedia. If an extended system call has been requested, **alsyscallext** invokes the function *callback* (if non-null), passing the request number and the array of arguments. **alsyscallext** also returns 0 for success, non-zero when the TM program requests an **exit** call.

13. Using System Call Threads

Instead of waiting explicitly for a system call interrupt, the Host program can launch a thread to handle system calls from each processor.

To spin off a system call thread, the Host program calls:

int **alsyscallserver** (int *dev*, int (**callback*)(int *dev*, int *request*,
unsigned long *args*[]), char **cwd*)

The thread handles standard system calls in the normal manner, while passing extended system calls to routine *callback* (if non-null), using exactly the same argument passing as in **alsyscallext** above. Argument *cwd* should be NULL to use the current working directory of the calling thread as the current working directory.

Once launched, each thread monitors its assigned processor for system call interrupts, while Host program execution continues. Each thread terminates when its corresponding processor program calls exit() or encounters an exception. If it is desired to synchronize the Host program and thread termination, the Host program calls

int alsyscallserverwait (int *dev*)

This function does not return until the system call thread for the given processor terminates.

To terminate a thread before its normal completion, the Host program calls

int alsyscallserverwait (int *dev*)

This function terminates the server for the given processor, leaving other threads unchanged.

B. ALRT TriMedia Program Functions

In the ALRT environment, the application includes a Host program to open, load, and start the TriMedia processors on the Alacron board. The TriMedia program in the ALRT environment calls the functions described in this section to communicate with the Host program. The TM Runtime library API is provided in library **libalfast.a**; the TriMedia program should include header file **alfast_tm.h**.

The application is responsible for all memory management. Standard TM library calls are available for dynamic allocation (**malloc**, **free**, etc.)

Host support is provided for the subset of C runtime functions listed earlier: **open**, **close**, **fstat**, **stat**, **isatty**, **read**, **write**, **lseek**, **time**, and **exit**

C runtime functions that do not require Host support are fully supported (e.g., **strcpy**).

1. Send Extended System Call to Host

In addition to the standard system calls listed earlier in this section, the application can include extended system calls for custom processing by the Host. The extended system call can pass a small number (currently ten) of unsigned long argument values for the Host program to process. To generate the system call request, the TriMedia program loads the values into an array, then calls

int **tmfast_syscall** (int *request*, int *nargs*, unsigned long *args*[], int **pret*)

The *request* should be a value greater than **MAX_SYSCALL_ARGS**, to avoid conflicts with standard system call requests. The call includes *nargs*, the number of arguments passed in array *args*. The Host program sets up a handler for extended system calls; the Host handler can determine how many arguments to expect based on the *request* number. Upon successful return from **tmfast_syscall**, variable *pret* (if non-NULL) contains the value returned by the Host handler.

2. Flag Registers

The ALRT driver maintains a set of global 32-bit Flag Registers for each of the TriMedia processors opened by the Host program (with **alopen**). The flag registers are located in uncached space.

The global 32 bit flag registers can serve as communication links among the TriMedias. Each TriMedia processor can read and write its own flag registers, and can also read and write the flag registers of another TriMedia when the processor number (*proc* in the **alopen** call by the Host) is known.

To obtain a pointer to its own flag registers, the TriMedia program calls

volatile unsigned long ***tmfast_flagregs** (void)

This call returns a pointer to the array of flag registers set up for the calling processor. When the **tmfast_flagregs** function returns, read/write references to the flag register array are valid. For example,

```
#include <alfast_tm.h>
extern char buf[];
volatile unsigned long *flagregs = tmfast_flagregs;
flagregs[0] = (*unsigned long) buf;
```

The TriMedia can also obtain a pointer to the flag registers of another processor in the system. The TriMedia program must somehow know the integer processor number used by the Host program to open the other TM (that is, the *dev* argument used in the Host's **alopen** call). The TM program calls

volatile unsigned long ***tmfast_flagregs_m** (int *dev*)

The function **tmfast_flagregs_m** returns a pointer to the beginning of the flag register array for processor *dev*, making the array accessible for reading or writing.

```
#include <alfast_tm.h>
// Read global structure for TM proc #2
int dev2 = 2;
extern char buf[];
volatile unsigned long flagregs2 = tmfast_flagregs_m (dev2);

flagregs2[0] = (*unsigned long) buf;
```

C. *Building Programs on Linux Systems*

1. Compiling Device Driver

The driver object files have been compiled with the Linux kernel version shown in the file **$ALFAST/linux/driver/kernel.version**

If a different kernel version is being used, it may be desirable to recompile the driver object files. This is accomplished as follows:

```
cd $ALFAST/linux/driver
make
cd $ALFAST/linux/pciprobe
make
```

2. Building Host Programs

Host application programs are built using the Linux C compiler (gcc) and are linked with Alacron provided libraries. The following is the standard makefile used to build host applications:

```
ROOTDIR      = $(ALFAST)
include $(ROOTDIR)/tools/include/linuxhost.inc

all:   app

OBJS   = app.o

app:   $(OBJS) $(LIBS)
       $(CC) –o $@ $(OBJS) $(LIBS)
```

3. Building TriMedia Programs

Developing the TriMedia programs for execution under ALRT is essentially the same as when targetting Philips TMMAN. Both the TriMedia libraries from Philips and the ALFAST libraries from Alacron can be accessed. PSOS+ and PSOS+M are supported.

TriMedia programs are compiled using cross development tools provided by Philips. The Philips software is (at present) not available to run under Linux hosts, therefore TriMedia program development is performed on supported hosts such as Windows NT, or Solaris.

When building TriMedia programs targeted for TMMAN, the user would compile with **–host WinNT.** When compiling for ALRT, the program would be compiled with **–host nohost –tmconfig=$ALFAST/lib/tcs20/make/tmconfig** flags, and must be linked with Alacron provided host communication module (**hcomm.o**). Makefiles are provided for use under Windows using **NMAKE**, and under Unix hosts using **make**.

4. Sample MakeFiles

The following are sample makefiles:

NMAKE File

```
ROOTDIR             = ..\..\..
!include $(ROOTDIR)\tools\include\tmle.inc
all:   ntsc.out

OBJS          = $(OBJDIR)\ntsc.o

ntsc.out:     $(OBJS) $(LIBS)
copy $(LIBDIR)\hcomm.o
$(LD) -o $@ $(OBJS) $(LDFLAGS) $(LDTAIL)
clean:
•      del $(OBJS)

clobber: clean
•      del ntsc.out
```

Make File

```
ROOTDIR             = ../../..
include $(ROOTDIR)/tools/include/tmle-mak.inc

all:   ntsc.out

OBJS         = $(OBJDIR)/ntsc.o

ntsc.out:    $(OBJS) $(LIBS)
       $(LD) -o $@ $(OBJS) $(LDFLAGS) $(LDTAIL)

clean:
       -rm -f $(OBJS)

clobber: clean
       -rm -f ntsc.out
```

5. Makefile Conventions for FastSeries Software

The following summarizes makefile naming conventions:

- makefiles for use with NMAKE are prefaced with ***nmake***
- makefiles for use with Unix make are prefaced with ***make***
- makefiles to build TriMedia programs to run with TMMAN under WINNT have suffix ***tmnt***
- makefiles to build TriMedia programs to run with ALRT little-endian have suffix ***tmle***
- makefiles to build TriMedia pregrams to run with ALRT big-endian have suffix ***tmbe***
- makefiles to build Host programs running under TMMAN or ALRT have suffix ***winnt***
- makefiles to build host programs running under Solaris/sparc have suffix ***sol***
- makefiles to biuld host programs running under Solaris/x86 have suffix ***solx86***
- makefiles to build host programs running under Linux/x86 have suffix ***linux***
- makefiles to build host programs running under Linux/PPC have suffix ***linuxppc***

So the following makefiles will be encountered:

nmake.tmnt	NMAKE TriMedia to run with TMMAN under WINNT
nmake.tmle	NMAKE TriMedia to run with ALRT little-endian
nmake.tmbe	NMAKE TriMedia to run with ALRT big-endian
nmake.winnt	NMAKE Host to run TMMAN or ALRT under WINNT
make.tmle	make TriMedia to run with ALRT little-endian
make.tmbe	make TriMedia to run with ALRT big-endian
make.sol	make Host to run with ALRT under Solaris/sparc
make.solx86	make Host to run with ALRT under Solaris/x86
make.linux	make Host to run with ALRT under Linux/x86
make.linuxppc	make Host to run with ALRT under Linux/PPC

III. PROGRAM EXAMPLES

A. tmload.c

The **tmload** program is a simple host application that loads a TM program.

```
/*****************************************************************************
**
** File:     tmload.c - load a program
**
** Copyright © 2000; Alacron Inc.
**
** Description:
**
** History:
**
*****************************************************************************/

/*----------------------- HEADER FILES ----------------------------------*/
#include <allib.h>
/*----------------------- PRIVATE CONSTANTS ------------------------------*/
/*----------------------- PRIVATE MACROS ---------------------------------*/
/*----------------------- PRIVATE TYPES ----------------------------------*/
/*----------------------- PRIVATE DATA -----------------------------------*/
static int dev = 0;
static char *ifname;

/*----------------------- PUBLIC DATA ------------------------------------*/
/*----------------------- PRIVATE ROUTINE REFERENCES ---------------------*/
PRIVATE void usage (void);
/*----------------------- PUBLIC ROUTINES --------------------------------*/

/*****************************************************************************
**
**  EXPORT -
**
**  Description:
**
*****************************************************************************/

int main (int argc, char *argv[])
{
int i;
int rval;
int target_argc;
char **target_argv;

for (i = 1; i < argc; i ++)
if (*argv[i] == '-')
switch (*(argv[i]+1))
                    {
case 'd':
if (++i < argc)
dev = atoi (argv[i]);
break;
default:
usage ();
/* no return */
                    }
```

```
else
             {
ifname = argv[i];
break;
             }

if (!ifname)
usage ();
target_argc = argc - i;
target_argv = &argv[i];

rval = alopen (dev);
if (rval)
      {
printf (“ERROR: can’t open device %d\n”, dev);
return 1;
      }
aldev (dev);
alputargs (target_argc, target_argv);
rval = almapload (ifname, 0);
if (rval)
      {
printf (“ERROR: almapload %s failed: errorcode %d\n”, ifname, rval);
return 1;
      }

for (;;)
      {
rval = ioctl_alfast (dev, IOCTL_INTWAIT);
if (rval)
             {
printf (“WAIT: failed, interrupted maybe\n”);
return 0;
             }
else
             {
rval = alsyscall (dev);
if (rval)
                    {
printf (“TM program terminated\n”);
break;
                    }
             }
      }

alclose (dev);
return 0;
}


/*----------------------- PRIVATE ROUTINES ------------------------------*/

/***************************************************************************
**
**  PRIVATE - usage -
**
**  Description:
**
***************************************************************************/

PRIVATE void usage (void)
{
printf (“Usage: tmload [-d dev] [-w] program\n”);
```

```
exit (1);
}
```

B. Extended System Call Example

This simple Host and TriMedia application demonstrates the use of extended system calls. File **shared.h** is shared by the Host and TriMedia programs, file **sctest.c** is the Host program, and **scmtest.c** is the TriMedia Program

1. shared.h

```
#define CMD_SHOWSTRING        (SYSCALL_EXT_START + 0)
#define CMD_SHOWINT           (SYSCALL_EXT_START + 1)
```

2. sctest.c

```
/************************************************************************
**
** File:     sctest.c - load extended system call test program
**
** Copyright © 1999; Alacron Inc.
**
** Description:
**
** History:
**
*************************************************************************/

/*----------------------- HEADER FILES ---------------------------------*/
#include <allib.h>
#include “shared.h”

/*----------------------- PRIVATE CONSTANTS ----------------------------*/
/*----------------------- PRIVATE MACROS -------------------------------*/
/*----------------------- PRIVATE TYPES --------------------------------*/
/*----------------------- PRIVATE DATA ---------------------------------*/
static int dev = 0;
static char *ifname = “tmsctest.out”;

/*----------------------- PUBLIC DATA ----------------------------------*/
/*----------------------- PRIVATE ROUTINE REFERENCES -------------------*/
PRIVATE int ext_handler (int dev, int cmd, unsigned long args[]);
/*----------------------- PUBLIC ROUTINES ------------------------------*/

/*************************************************************************
**
**  EXPORT - main
**
**  Description:
**
*************************************************************************/

int main (int argc, char *argv[])
{
int i;
int rval;

for (i = 1; i < argc; i ++)
if (*argv[i]  == ‘-‘)
```

```
switch (*(argv[i] + 1))
                    {
case ‘d’:
if (++i < argc)
dev = atoi (argv[i]);
break;

                    }

       /*
•      open device
        */

rval = alopen (dev);
if (rval)
       {
printf (“host: ERROR: can’t open device %d\n”, dev);
exit (1);
       }
aldev (dev);

       /*
•      load program TM file
        */

rval = almapload (ifname, 0);
if (rval)
       {
printf (“host: ERROR: almapload %s failed: error code %d\n”,
ifname, rval);
exit (1);
       }

       /*
•      Main wait loop
        *
•      wait for system call interrupt
        *
•      process system call request, using ext_handler() for
•      extended system calls.
        */

for (;;)
       {
rval = ioctl_alfast (dev, IOCTL_INTWAIT);
if (rval)
       {
printf (“WAIT: failed, signal maybe\n”);
return 0;
              }
else
              {
rval = alsyscallext (dev, ext_handler);
if (rval)
                    {
printf (“TM program terminated\n”);
break;
                    }
              }
       }

alclose (dev);
return 0;
```

```
}


/*---------------------- PRIVATE ROUTINES ------------------------------*/

/****************************************************************************
**
**  PRIVATE - ext_handler -
**
**  Description:
**
**     Handle application specific system calls.  The TM passes a command
**     up to MAX_SYSCALL_ARGS 32 bit arguments.  This function processes
**     the command, returning a value that is returned to the TM.
**
***************************************************************************/

PRIVATE int ext_handler (int dev, int cmd, unsigned long args[])
{
int rval = 0;
switch (cmd)
       {
case CMD_SHOWSTRING:
       {
char *s;
s = almapsdram (dev, args[0]);
printf (“host: from %d: %s\n”, dev, s);
rval = 100;
break;
       }

case CMD_SHOWINT:
       {
printf (“host: from %d: %ld %ld\n”, dev, args[0], args[1]);
rval = 200;
break;
       }

default:
rval = -1;
       }

return rval;

}
```

3. tmsctest.c

```
#include <alfast_tm.h>
#include “shared.h”

int main (int argc, char *argv[])
{
unsigned long args[MAX_SYSCALL_ARGS];
int rval;
int retval;

args[0] = (unsigned long) “this is a test ...”;
_cache_copyback ((void*) args[0], strlen ((char*)args[0] + 1));
rval = tmfast_syscall (CMD_SHOWSTRING, 1, args, &retval);
if (rval)
       {
printf (“tm: tmfast_syscall failed\n”);
exit (1);
       }
printf (“tm: CMD_SHOWSTRING returned %d\n”, retval);
args[0] = 11111;
args[1] = 22222;
rval = tmfast_syscall (CMD_SHOWINT, 2, args, &retval);
if (rval)
       {
printf (“tm: tmfast_syscall failed\n”);
exit (1);
       }
printf (“tm: CMD_SHOWINT returned %d\n”, retval);

}
```

IV. ALRT HOST SOFTWARE REFERENCE

A. Function Summary

aladdr	retrieve the address of a target program variable
aldev	select current device
alclose	close a device
alflagregs	return pointer to flag registers
alget*/alset*	read/write byte/word/long single element or array
almapload	load a program file on to a processor
almapsdram	return pointer to SDRAM
alopen	open a device (processor)
alputargs	pass argc/argv arguments to target processor
alsyscall	process system calls from a target processor
alsyscallext	process system call, with extensions
alsyscallserver	spin off a system call server thread
alsyscallserver_stop	stop a system call server thread
alsyscallserver_wait	wait on a system call server thread
VtoP	convert target virtual to physical address

aladdr

C Usage:

```
#include <allib.h>
ADDR   aladdr (char *name)
```

Arguments

name	Name of a program global symbol

Description:

The **aladdr** function returns the virtual address of the variable given by the string input argument *name*. The symbol table information in the currently loaded target executable file is examined to resolve the address.

Function **aladdr** may only be called after a call to **almapload**. Only the symbols loaded by the most recent call to **almapload** may be referenced by **aladdr.**

The virtual address returned by **aladdr** is the variable's address taken as though the program image were relocated to a TM with SDRAM located at PCI address 0. In other words, the virtual address from **aladdr** is the offset from SDRAM start for that processor.

The returned virtual address should be converted to a physical address with a call to **VtoP** for use in functions **alget[bwl][a]**, and **alset[bwl][a]**.

NOTE: There is significant overhead in executing the **aladdr** call. If the address of a symbol is to be used more than once, it is recommended that **aladdr** be called once and the value saved in a program variable.

Return Values:

Function **aladdr** returns a 32 bit unsigned address, or zero if the variable address could not be found.

Example:

```
ADDR virtadd;
virtadd = aladdr("_shared_buffer");
if (virtadd == 0) {
printf ("FATAL: aladdr failed\n");
       }
```

alclose

C Usage:

```
#include <allib.h>
int    alclose (int proc)
```

Arguments:

proc	Target processor unit number

Description:

The **alclose** function closes target processor specified by the *proc* argument. This function should be called before the Host application exits.

Return Values:

None.

Example:

```
int proc=0;
alclose(proc);
```

aldev

C Usage:

```
#include <allib.h>
int    aldev (int proc)
```

Arguments

proc	Target processor unit number

Description:

The **aldev** function selects the processor specified by the *proc* argument to be the active device for subsequent calls (until the next **aldev** call). This function should be called after the processor is opened using **alopen**.

Return Values:

SUCCESS	Function succeeded
FAILURE	Function failed, device not opened

Example:

```
int proc;

for (proc=0; proc<8; proc++) {
       alopen(proc);
       aldev(proc);
       almapload(“prog”, 0x10000);
       }
```

alflagregs

C Usage:

```
#include <allib.h>
volatile unsigned long *alflagregs (int dev)
```

Arguments

dev	The processor to access

Description:

This function returns a host mapped virtual address that may be used to directly access the flag registers on TriMedia *dev*. The **MAX_FLAG_REGS** locations may be used in any way desired by the application

Return Values:

non-null	Pointer mapped flag register
null	The mapping failed

algetb, algetw, algetl, algetba, algetwa’ algetla’ alsetb. Alsetw. Alsetl, alsetba. Alsetwa & alsetla

C Usage:

```
#include <allib.h>
int          algetb  (ADDR physadd)
int          algetw  (ADDR physadd)
long   algetl  (ADDR physadd)
int          algetba (ADDR physadd, char  *buf, int n)
int          algetwa (ADDR physadd, short *buf, int n)
long   algetla (ADDR physadd, long  *buf, int n)
int          alsetb  (ADDR physadd, int  value)
int          alsetw  (ADDR physadd, int  value)
long   alsetl  (ADDR physadd, long value)
int          alsetba (ADDR physadd, char  *buf, int n)
int          alsetwa (ADDR physadd, short *buf, int n)
long   alsetla (ADDR physadd, long  *buf, int n)
```

Arguments

physadd	A physical address in the processor program. Use function **aladdr** to get a program virtual address and **VtoP** to translate a virtual address to its corresponding physical address.
buf	A pointer to a host buffer
n	The number of bytes, words, longwords to be transferred
value	The byte, word or long value to write to processor address physadd.

Description:

These functions transfer data between a Host program and a buffer in target processor memory. The **alget** functions transfer data from the processor buffer to a host buffer while the **alset** functions transfer data from host memory to processor memory. The memory transfer is performed on the processor that is selected by the **aldev** function. These functions allow the host to read and write processor memory.

The **algetba**, **algetwa**, **algetla**, **alsetba**, **alsetwa**, and **alsetla** functions transfer arrays of bytes, 16-bit words or 32-bit longwords between the processor buffer specified by argument *physadd* and the host buffer specified by argument *buf.* The size of both buffers must be at least *n* bytes, words, or longwords as appropriate.

The **algetb**, **algetw**, **algetl**, **alsetb**, **alsetw**, and **alsetl** functions transfer a single byte, word or longword between processor board memory and host memory. In the case of the **alget** functions, the value read from processor board physical address *physadd* is the function return value. The **alset** functions write the byte, word or longword in argument *value* to physical address *physadd*.

Buffers that are shared by the host and target processors should be uncached. Functions **alsetb**, **alsetw**, and **alsetl** may perform a read-modify-write in processor memory. Data could be overwritten if user software allows the host and TriMedia application to modify the same buffer in processor memory at the same time.

Return Values:

algetb	byte at address *physadd*
algetw	16-bit word at address *physadd*
algetl	32-bit longword at address *physadd*
algetba	SUCCESS or FAILURE
algetwa	SUCCESS or FAILURE
algetla	SUCCESS or FAILURE
alsetb	none
alsetw	none
alsetl	none
alsetba	SUCCESS or FAILURE
alsetwa	SUCCESS or FAILURE
alsetla	SUCCESS or FAILURE

Example:

```
/* Host Program /
char hostbuf[SIZE];              / input-output data /
ADDR V_func;
ADDR V_procadd, P_procadd;

V_func = aladdr(“_func”);
V_procadd = aladdr(“_global_array”);
P_procadd = VtoP(V_procadd);

alsetba(P_procadd, hostbuf, SIZE);      / copy input to TM /
alcall(V_func, 1, (long) V_procadd);    / process data /
alwait();

algetba(P_procadd, hostbuf, SIZE);      / retrieve output
data */
```

almapload

C Usage:

```
#include <allib.h>
int almapload (char *filename, long stacksize)
```

Arguments:

filename	TriMedia executable file residing on Host disk file system.
stacksize	Not used by ALRT

Description:

Function **almapload** loads the TriMedia executable file given by *filename* into the active processor memory using memory management. Argument *filename* should be a file residing within the Host file system. The file should be a TriMedia executable image. The *stacksize* argument is ignored.

Upon completion of **almapload**, the on-board TriMedia processor reset is deasserted and program execution begins at the function `main`.

After a call to **almapload**, the Host program may look up the virtual addresses of TM program symbols using the function **aladdr**. Only the symbols loaded by the most recent call to **almapload** may be referenced by **aladdr**.

The TM processor requires access to two resources, the MMIO space, and the SDRAM space. The physical location of these resources are assigned by the BIOS or OS as part of the PCI bus configuration. The TM uses the physical addresses of the resources when accessing them. Since the TM program code and data are executed out of SDRAM, all references to these addressed items must be adjusted to the actual PCI address where they are located. **almapload** does this for each TM loaded.

Return Values:

SUCCESS	Function succeeded
FAILURE	Function failed, device not opened

Example:

```
alopen (0);
aldev (0);
almapload(“filename”, 0x0L);
```

almapsdram

C Usage:

```
#include <allib.h>
void *almapsdram (int dev, unsigned long addr)
```

Arguments

Dev	The processor to access
addr	Physical address to convert

Description:

This function returns a host mapped virtual address that may be used to indirectly access the SDRAM physical address *addr* on TriMedia device *dev*.

Return Values:

non-null	Pointer to requested SDRAM
null	The address was invalid

Example:

```
int dev = 0;
volatile unsigned long *flags = alflagregs (dev);
char *buf;

/*--- wait for TM to pass us its buffer address ---*/
while (!flags[0])
       ;

/*--- Stash the buffer address ---*/

buf = almapsdram (dev, flags[0]);
if (!buf)
       printf (“ERROR: almapsdram failed\n”);
```

alopen

C Usage:

```
#include <allib.h>
int    alopen (int proc)
```

Arguments

Proc	The target processor unit number. Normally, the first target is unit 0.

Description:

The **alopen** function opens the target processor specified by argument *proc*. The value 0 is returned if the open is successful, -1 otherwise. This function must be called on a processor before any other Host library functions can use that processor.

Return Values:

SUCCESS	Device present. Open succeeded.
FAILURE	Open failed.

Example:

```
int proc;
int fd[MAX_TM_DEVS];

for (proc=0; proc<MAX_TM_DEVS; proc++) {
       fd[proc] = alopen (proc);
       if (fd[proc]==SUCCESS) {
              printf (“Processor[%d] opened\n”, proc);
              }
       }
```

alputargs

C Usage:

```
#include <allib.h>
int alputargs (int argc, char *argv[])
```

Arguments

Argc	number of arguments passed in argv
Argv	array of pointers to null terminated argument strings

Description:

This function is used to pass argc/argv parameters to the active target processor. It must be called after device open (**alopen**) and device selection (**aldev**) but prior to program load (**almapload**). The array of ***argv*** strings must reference data that remains present until after the **almapload** call. The arguments are passed to the target processor using the system call service mechanism, so the host application must enter a wait for system call loop to utilize this facility.

Return Values:

SUCCESS	Function succeeded
FAILURE	Function failed, device not opened

Example:

```
int main (int argc, char *argv[])
{
int proc = 0;

       if (alopen (proc)) error ();
       aldev (proc);
       /* pass all but first argument */
       alputargs (argc – 1, argv + 1);
       if (almapload (infile, 0)) error ();

       /* begin waiting for system calls */
}
```

alsyscall

C Usage:

```
#include <allib.h>
int alsyscall (int proc)
```

Arguments:

proc	The processor whose system calls are to be serviced

Description:

This function services any system call requests that are pending for the processor given by *proc*. If the target has requested an **exit()** call, **alsyscall** returns non-zero, otherwise a 0 value is returned.

Return Values:

SUCCESS	Function succeeded
non-zero	an Exit call was requested

Example:

alsyscallext

C Usage:

```
#include <allib.h>
int alsyscallext (int proc, int (*pf)(int proc, int request,
                    unsigned long args[])
```

Arguments:

proc	The processor whose system calls are to be serviced
pf	pointer to extended system call handler
request	system call request number
args	array of 32 bit arguments

Description:

This function services any system call requests (including extended system calls) pending for the processor given by *proc*. If the target has requested an **exit()** call, **alsyscallext** returns non-zero, otherwise a 0 value is returned. If an extended system call is requested, **alsyscallext** will invoke the function ***pf*** if non-null, passing the system call number and up to **MAX_SYSCALL_ARGS** unsigned long arguments. Token **SYSCALL_EXT_START** may be used by both Host and TriMedia programs as the base code for extended system calls.

Return Values:

SUCCESS	Function succeeded
non-zero	an Exit call was requested

Example:

```
int userext (int dev, int cmd, unsigned long args[])
       {
       switch (cmd)
       {
       case SYSCALL_EXT_START:
           printf (“got command 2000: args %08X %08X%08X\n”,
              args[0], args[1], args[2]);
       default:
              return –1;
       }
}

// In main program
int dev = 0;
int rval;

for (;;)
{
       rval = ioctl_alfast (dev, IOCTL_INTWAIT);
       if (rval)
       {
```

```
                printf (“awoken by signal\n”);
                return;
        }
        rval = alsyscallext (dev, userext);
        if (rval)
        {
                printf (“TM %d terminated\n”);
                return;
        }
}
```

alsyscallserver

C Usage:

```
#include <allib.h>
int alsyscallserver (int dev, int (*callback)(int dev, int
request, unsigned long args[]), char *cwd)
```

Arguments

dev	The processor whose system calls are to be serviced
callback	pointer to extended system call handler
request	system call request number
args	array of 32 bit arguments
cwd	current working directory

Description:

This function spins off a thread to handle system call requests from the TriMedia processor given by *dev*. The argument *callback*, if non-null, is invoked (as in **alsyscallext**) when an extended system call request is encountered. The argument *cwd*, if non-null specifies a filesystem path to a directory that the system call service thread uses as the current working directory. If *cwd* is null, then the current working directory for the calling thread is used.

Return Values:

SUCCESS	Function succeeded
non-zero	The system call server thread could not be started

Example:

```
int rval;
int dev = 0;

alopen (dev);
aldev (dev);
almapload (dev, “program.out”, 0);
alsyscallserver (dev, NULL, NULL);
alyscallserver_wait (dev);
```

alsyscallserver_stop

C Usage:

```
#include <allib.h>
int alsyscallserver_stop (int dev)
```

Arguments

Dev	Processor whose system calls are being handled.

Description:

This function terminates the system call server for the processor given by *dev*.

Return Values:

SUCCESS	Function succeeded
non-zero	The wait failed

Example:

```
int dev = 0;
alsyscallserver (dev, NULL, NULL);
alsyscallserver_stop (dev);
```

alsyscallserver_wait

C Usage:

```
#include <allib.h>
int alsyscallserver_wait (int dev)
```

Arguments

dev	Processor whose system calls are to be serviced

Description:

This function waits until the host thread started (with **alsyscallserver**) for processor *dev* has completed. The call to **alsyscallserver_wait** returns when the TriMedia program completes, either by calling **exit()** or by encountering an exception.

Return Values:

SUCCESS	Function succeeded
non-zero	The wait failed

Example:

```
int dev = 0;

alsyscallserver (dev, NULL, NULL);
alsyscallserver_wait (dev);
```

VtoP

C Usage:

```
#include <allib.h>
ADDR   VtoP (ADDR vaddr)
```

Arguments

vaddr	A virtual address

Description:

Function **VtoP** returns the physical address corresponding to the virtual address given by *vaddr*. **VtoP** returns the value zero if the virtual address is not mapped to a physical address.

Host calls to **alget** and **alset** require the physical PCI address. The physical address is the actual PCI address. Function **VtoP** returns the physical address corresponding to a given virtual address for the active processor.

When the TM processor passes a target virtual address to the Host, the target virtual address should be treated as a physical address, and not passed to **VtoP**. **VtoP** should be used only for addresses that come from **aladdr**.

Return Values:

VtoP	physical address of *vaddr* on processor 0
Zero	Address not valid

Example:

```
ADDR Vadd, Padd;
Vadd = aladdr(“symbol_name”);
Padd = VtoP(Vadd);
```

V. ALRT HOST DRIVER-LEVEL FUNCTIONS

The functions documented in this section provide lower level access to the ALRT device driver. These functions are independent of the operating system. Except for **ioctl_alfast**, these functions are not recommended for use by applications. The same functionality is available via the ALRT Host Library API. The driver-level functions may be of assistance in some development situations.

The following device driver interface functions are provided:

close_alfast	close device
ioctl_alfast	perform I/O control function
open_alfast	open device
read_alfast	read from SDRAM
seek_alfast	set read/write seek address
write_alfast	write to SDRAM

close_alfast

C Usage:

```
#include <allib.h>
int close_alfast (int dev)
```

Arguments

dev	The processor to close

Description:

This function closes the TriMedia device given by *dev*.

Return Values:

SUCCESS	Function succeeded
FAILURE	Close failed

Example:

```
int dev = 0;
int rval;

rval = close_alfast (dev);
if (rval)
printf (“ERROR: close_alfast %d failed\n”);
```

NOTE: Applications should use **alclose** rather than **close_alfast**.

ioctl_alfast

C Usage:

```
#include <allib.h>
int ioctl_alfast (int dev, int cmd, …)
```

Arguments:

dev	The processor to access
cmd	The I/O command to execute

Description:

This function performs various I/O control functions on the TM processor *dev* (previously opened with a call to **alopen**). Here are the available commands in alphabetical order.

int **ioctl_alfast** (*dev*, **ALF_DMABUF_MAP**, *dmabuf_map_t *map*)

```
typedef struct {
      unsigned long vaddr;
      int len;
} dmabuf_map_t;
```

This function is used to map a kernel-resident DMA buffer into user address space.

Note: Kernel-resident DMA buffers are not supported at Release 1.4 of ALRT.

int **ioctl_alfast** (*dev*, **ALF_IOCTL_DMABUF_ALLOC**, dmabuf_alloc_t **dmabuf*)

```
typedef struct {
      int len;
} dmabuf_alloc_t;
```

This function requests that an allocation of a kernel-resident DMA buffer of ***len*** bytes be made. The allocation will fail if kernel resources are not available or a DMA buffer has been previously allocated without being freed.

Note: Kernel-resident DMA buffers are not supported at Release 1.4 of ALRT.

int **ioctl_alfast** (*dev*, **ALF_IOCTL_DMABUF_FREE**)

This function frees a kernel resident DMA buffer that was allocated using **IOCTL_DMABUF_ALLOC**. Any mapping made with **IOCTL_DMABUF_MAP** is freed.

Note: Kernel-resident DMA buffers are not supported at Release 1.4 of ALRT.

int **ioctl_alfast** (*dev*, **ALF_IOCTL_DMABUF_QUERY**, *dmabuf_query_t *query*)

```
typedef struct {
      unsigned long physaddr;
      int len;
} dmabuf_query_t;
```

This function queries the current kernel DMA buffer attributes, placing the results in *query*. The element `physaddr` is set to the address that an I/O device would use to access the DMA buffer.

Note: Kernel-resident DMA buffers are not supported at Release 1.4 of ALRT.

int **ioctl_alfast** (*dev*, **IOCTL_ASSERT_RESET**)

This function asserts the processor reset of the TriMedia given by the argument *dev*.

int **ioctl_alfast** (*dev*, **IOCTL_DEASSERT_RESET**)

This function deasserts the processor reset of the TriMedia given by argument *dev*.

int **ioctl_alfast** (*dev*, **IOCTL_GET_CLOCK_SPEED**, unsigned long **presult*)

This function returns the processor clock speed in Hz of the TriMedia given by *dev*. The result is written to the address given by *presult*

int **ioctl_alfast** (*dev*, **IOCTL_GET_DINFO**, get_dinfo_t **presult*)

This function fills the `get_dinfo_t` structure pointed to by *presult*. The `get_dinfo_t` structure is declared as follows:

```
typedef struct {
      unsigned long dev
      unsigned long sdram_base;
      unsigned long sdram_size;
      unsigned long mmio_base;
      unsigned long mmio_size;
      unsigned long ireq;
      int bus;
      int slot;
      char version[VERSION_LENGTH]
} get_dinfo_t;
```

Where:

Dev	TriMedia device
Sdram_base	physical address of SDRAM
Sdram_size	size in bytes of SDRAM
mmio_base	physical address of MMIO space
mmio_size	size in bytes of MMIO space
Irq	Interrupt request line used by TriMedia
Bus	PCI bus
Slot	PCI slot (device)
Version[]	text string identifying driver version

int **ioctl_alfast** (*dev*, **IOCTL_GET_FLAGREGS_BASE**, volatile unsigned long **presult*)

This function returns a host mapped virtual address of the `flagregs[]` region of shared memory. The value returned may be used as a pointer by the host software.

Note: this function is identical in functionality to **alflagregs**

int **ioctl_alfast** (*dev*, **IOCTL_GET_MMIO_BASE**, unsigned long **presult*)

This function returns the physical base address of the MMIO space of the TriMedia given by *dev*. The result is written to the address given by *presult.*

int **ioctl_alfast** (*dev*, **IOCTL_GET_SDRAM_BASE**, unsigned long **presult*)

This function returns the physical base address of the SDRAM space of the TriMedia given by *dev*. The result is written to the address given by *presult.*

int **ioctl_alfast** (*dev*, **IOCTL_GET_SDRAM_SIZE**, unsigned long **presult*)

This function returns the size in bytes of the SDRAM space of the TriMedia given by *dev*. The result is written to the address given by *presult.*

int **ioctl_alfast** (*dev*, **IOCTL_INTWAIT**)

This function waits for a system call interrupt from the TriMedia given by *dev*.

int **ioctl_alfast** (*dev*, **IOCTL_MMIO_READ**, int *offset*, unsigned long **presult*)

This function reads the MMIO register of device *dev* offset by *offset* bytes. The result is stored into the address given by argument *presult*.

int **ioctl_alfast** (*dev*, **IOCTL_MMIO_WRITE**, int *offset*, unsigned long *value*)

This function writes *value* to the MMIO register offset by *offset* bytes of the TriMedia given by *dev*.

Return Values:

SUCCESS	Function succeeded
non-zero	an Exit call was requested

Example:

open_alfast

C Usage:

```
#include <allib.h>
int open_alfast (int dev)
```

Arguments:

dev	The processor to open

Description:

This function attempts to open the TriMedia device given by ***dev***.

Return Values:

SUCCESS	Function succeeded
FAILURE	Open failed

Example:

```
int dev = 0;
int rval;

rval = open_alfast (dev);
if (rval)
      printf (“ERROR: open_alfast %d failed\n”);
```

NOTE: Applications should use **alopen** rather than **open_alfast**.

read_alfast

C Usage:

```
#include <allib.h>
int read_alfast (int dev, void *buf, unsigned n)
```

Arguments

dev	The processor to open
buf	host buffer address
n	number of bytes to transfer

Description:

This function reads *n* bytes from the SDRAM of device *dev* to the host buffer *buf*. The TriMedia SDRAM address used is the one specified by the current seek address as set by **seek_alfast** or by another **read_alfast** or **write_alfast**.

Upon completion, the current seek address is updated to the byte following the last byte read.

The request will fail if the range given by the current seek address and length falls outside the range of SDRAM for the device.

Return Values:

SUCCESS	Function succeeded
FAILURE	Read failed

Example:

```
int dev = 0;
int rval;
char buf[1000];

ADDR addr = 0xdc000000;
rval = seek_alfast (dev, addr);
if (!rval)
       rval = read_alfast (dev, buf, sizeof (buf));
if (rval)
       printf (“ERROR: seek or read failed\n”);
```

NOTE: Applications should use **alget*** rather than **read_alfast**.

seek_alfast

C Usage:

```
#include <allib.h>
int seek_alfast (int dev, ADDR addr)
```

Arguments

dev	The processor to open
addr	Seek address to set

Description:

This function sets the current seek address for subsequent calls to **read_alfast** and **write_alfast**. The value *addr* is set for *dev*. *addr* is a processor physical address.

Return Values:

SUCCESS	Function succeeded
FAILURE	Seek failed

Example:

```
int dev = 0;
int rval;
ADDR addr = 0xdc000000;
rval = seek_alfast (dev, addr);
if (rval)
printf (“ERROR: seek_alfast %d failed\n”);
```

NOTE: Application programs should use the **alget*** and **alset*** functions rather than **seek_alfast**.

write_alfast

C Usage:

```
#include <allib.h>
int write_alfast (int dev, void *buf, unsigned n)
```

Arguments

dev	The processor to open
buf	host buffer address
n	number of bytes to transfer

Description:

This function writes *n* bytes from the host buffer *buf* to the SDRAM of device *dev*. The TriMedia SDRAM address used is the one specified by the current seek address as set by **seek_alfast** or by another **read_alfast** or **write_alfast**.

Upon completion, the current seek address is updated to the byte following the last byte written.

The request will fail if the range given by the current seek address and length falls outside the range of SDRAM for the device.

Return Values:

SUCCESS	Function succeeded
FAILURE	Read failed

Example:

```
int dev = 0;
int rval;
char buf[1000];
ADDR addr = 0xdc000000;
rval = seek_alfast (dev, addr);
if (!rval)
rval = write_alfast (dev, buf, sizeof (buf));
if (rval)
printf (“ERROR: seek or write failed\n”);
```

NOTE: Applications should use **alset*** rather than **write_alfast**.

VI. UTILITY PROGRAMS

This reference section provides information on the following utility programs. These programs are executed at the OS prompt.

tmdump	Display TM's MMIO registers
tmload	Load a TM program
tmmemtest	Run quick memory test on a TM's SDRAM
tmmkdef	List TM's I/O mappings
tmmpload	Load multiple TM programs
tmreset	Reset TM
tmtest	Verify HW & SW installation, list TM configurations

tmdump

Usage:

```
tmdump [-d device]
```

Command Line Options:

-d device	Specifies which TM processor to display

Description:

The **tmdump** program displays the MMIO registers of the selected TM processor; if no *dev* is specified, processor 0 is the default. Output is directed to host standard output.

tmload

Usage:

```
tmload [-d dev] program arg1 arg2 …
```

Command Line Options:

-d dev	Specifies which TM processor to load and run on
program	TM executable file to load
arg1, …	Arguments passed to **main (argc, argv)** of the TM program

Description:

The **tmload** program loads the program given by argument *program* on the TM processor given by argument **–d** *dev. dev* is an integer ranging from 0 to the number of TMs less 1; if no *dev* is specified, processor 0 is the default. Arguments *arg1*, … are passed to the TM program's **main** entry point.

Following successful loading, **tmload** waits for interrupts from the TM indicating a system call request or termination.

tmmemtest

Usage:

```
tmmemtest [-d dev]
```

Command Line Options:

-d *dev*	Specifies which TM processor to run the memory test on

Description:

The **tmmemtest** program performs a single write/read memory test of the selected TM's SDRAM, reporting the number of byte compare errors detected. If no *dev* is specified, processor 0 is the default.

tmmkdef

Usage:

```
tmmkdef
```

Command Line Options:

	none

Description:

The **tmmkdef** program displays hardware mappings and settings for all TM processors. Output is directed to host standard out and will show one line per TM in a format similar to the following:

```
133333333 0xDB000000 0xDB800000 0x00800000
133333333 0xDA000000 0xDA800000 0x00800000
133333333 0xD3000000 0xD3800000 0x00800000
133333333 0xD2000000 0xD2800000 0x00800000
133333333 0xD0000000 0xD0800000 0x00800000
133333333 0xD1000000 0xD1800000 0x00800000
```

Column 1 is the processor clock frequency, column 2 is the MMIO base address, column 3 is the SDRAM base address, and column 4 the SDRAM size.

tmmpload

Usage:

```
tmmpload –exec prog1 arg1 … -exec prog2 arg1 …
```

Command Line Options:

-exec	option delimiter introducing the program and arguments for each processor in turn
prog1,2	TM program to load on the first, second, … TriMedia
arg1, …	Arguments for each program instance

Description:

The **tmmpload** program is used to load multiple TriMedia programs with a single command. **tmmpload** can load a different program executable on each TM, each with its own set of arguments. The program and arguments for each TM are introduced on the command line by an –`exec` option.

The following example starts the programs **a.out, b.out, c.out,** and **d.out** on TriMedia's 0, 1, 2 and 3. Each program is passed an arbitrary set of arguments:

```
tmmpload –exec a.out 0 –exec b.out 1 1 –exec c.out 2 2 2 –
exec d.out 3 3 3 3
```

tmreset

Usage:

```
tmreset [-d dev | -a]
```

Command Line Options:

-d *dev*	Specifies which TM processor to reset
-a	Reset all TM processors

Description:

The **tmreset** program performs a hardware reset on the TM processor selected with **–d** *dev*) or on all processors (with **–a**). If no processor is specified, processor 0 is the default.

tmtest

Usage:

```
tmtest
```

Command Line Options:

	none

Description:

The **tmtest** program runs a short series of host and TriMedia programs to establish the correct configuration and software installation. (Note: this program is identical to the **usertest** program that may be run under TMMAN). When properly installed a display appears similar to the following:

VII. ALRT TRIMEDIA RUNTIME REFERENCE

A. Error Control

Most all library function will return with a result code. The following is a list of (some of the) result codes:

ALF_RUNTIME_NOERROR	returned if successful – has the value zero
ALF_RUNTIME_ERROR_NOMEM	a memory allocation failure occurred.
ALF_RUNTIME_ERROR_BADPARM	an invalid parameter was passed to a function
ALF_RUNTIME_ERROR_BUSY	the underlying subsystem was busy and unavailable
ALF_RUNTIME_ERROR_OVERFLOW	an output buffer overflow occurred
ALF_RUNTIME_ERROR_TIMEOUT	the function did not complete in the expected time
ALF_RUNTIME_ERROR_NOTIMPLEMENTED	the requested function is not implemented in this version of the library

B. Functions

tmfast_flagregs	Return pointer to TM's own flag registers
tmfast_flagregs_m	Return pointer to flag registers on other TM
tmfast_syscall	Send extended system call to Host program

tmflagregs

Name

`tmfast_flagregs`

Usage

```
#include <alfast_tm.h>
volatile unsigned long *tmfast_flagregs (void)
```

Description

This function returns a pointer to the flag registers.

Return Values

non-null	returned pointer to flag registers
Null	error occurred

Examples

```
/*--- Set flagregs 0 to address of data buffer ---*/

extern char buf[];
volatile unsigned long flagregs = tmfasg_flagregs ();

flagregs[0] = (unsigned long) buf;
```

tmfast_flagregs_m

Name

```
tmfast_flagregs_m
```

Usage

```
#include <alfast_tm.h>
volatile unsigned long *tmfast_flagregs_m (int node)
```

Description

This function returns a pointer to the flag registers of the TM processor given by argument ***node***.

Return Values

non-null	returned pointer to flag registers
Null	error occurred

Examples

```
/*--- Set flagregs 0 of TM 2 to address of data buffer ---*/

extern char buf[];
volatile unsigned long *flagregs2 = tmfasg_flagregs_m (2);

flagregs2[0] = (unsigned long) buf;
```

mfast_syscall

Name:

```
tmfast_syscall
```

Usage:

```
#include <alfast_tm.h>int tmfast_syscall (int cmd, int
nargs, unsigned long args[], int *pret)
```

Description:

This sends the requested extended system call to the host. The argument ***cmd*** specifies system call number, ***args[]*** contains ***nargs*** arguments that are passed to the host. ***args*** must contain **MAX_SYSCALL_ARGS** arguments or less Upon completion ****pret*** (if non-null) will contain the system call return value.

Return Values:

ALF_RUNTIME_NOERROR	no error
ALF_RUNTIME_ERROR_BADPARM	invalid parameter was passed

Examples:

```
#define CMD1  2000
#define CMD2  2001
unsigned long args[3];
int rval;
int sc_return;
args[0] = first_arg;
args[1] = next_arg;
args[2] = last_arg;

/* 3 arguments, and a return */
rval = tmfast_syscall (CMD1, 3, args, &sc_return);
if (rval)
      printf (“ERROR: syscall failed\n”);
else
      printf (“syscall returned %d\n”, sc_return);

/* no args or return on this call */rval = tmfast_syscall
(CMD2, 0, NULL, NULL);
```

VIII. ALRT DEVICE DRIVER REFERENCE

The ALRT Device Driver is implemented as three separate drivers, one each for the TriMedias, the S3 ViRGE, and the PCI-to-PCI bridge.

A. Trimedia Device Driver

int **open** (char **path*, int *oflag*, …)

The open function establishes a connection with the TM device given by ***path***. The device name for TM processors is **/dev/alfast_tm<n>** where **<n>** indicates the device instance.

int **close** (int *filedes*)

The close function disconnects access to the device.

void ***mmap** (void **addr*, size_t *len*, int *prot*, int *flags*, int *filedes*, off_t *off*)

This function is used to map TM device resources into the application address space. Refer to Unix man-pages for details on the call. The ***off*** argument specifies the offset into the address space. Since there are two distinct address spaces for the TM (one for MMIO, one for SDRAM), the high order nibble of ***off*** specifies which address space to map.

0x0xxxxxxx	map SDRAM
0x1xxxxxxx	map MMIO

Int **ioctl** (int *filedes*, **ALF_IOCTL_INTWAIT**)

This function blocks the calling thread until a TM interrupt is received.int **ioctl** (int *filedes*, **ALF_IOCTL_GET_DINFO**, Unix_get_dinfo_t **dinfo*)

```
typedef struct {
       int dev;
       unsigned long sdram_base;
       unsigned long sdram_size;
       unsigned long mmio_base;
       unsigned long mmio_size;
       int bus;
       int devfunc;
       char version[VERSION_LENGTH];
} Unix_get_dinfo_t;
```

This function retrieves device information from the driver using the **Unix_get_dinfo_t** structure.

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_READ_BYTE**, unsigned char **result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_READ_SHORT**, unsigned short **result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_READ_LONG**, unsigned long **result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_WRITE_BYTE**, unsigned char *result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_WRITE_SHORT**, unsigned short *result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_WRITE_LONG**, unsigned long *result*)

These functions perform PCI configuration space reads and writes to the TM device.

Int **ioctl** (int *filedes*, **ALF_IOCTL_DMABUF_ALLOC**, dmabuf_alloc_t **dmabuf*)

```
typedef struct {
int len;
} dmabuf_alloc_t;
```

This function requests that an allocation of a kernel resident dma buffer of ***len*** bytes be made. The allocation will fail if kernel resources are not available, or a dma buffer has been previously allocated without being freed.

Int **ioctl** (int *filedes*, **ALF_IOCTL_DMABUF_FREE**)

This function frees a kernel resident DMA buffer that was allocated using **IOCTL_DMABUF_ALLOC**. Any mapping made with **IOCTL_DMABUF_MAP** will be freed.int **ioctl** (int *filedes*, **ALF_IOCTL_DMABUF_QUERY**, *dmabuf_query_t *query*)

```
typedef struct {
      unsigned long physaddr;
      int len;
} dmabuf_query_t;
```

This function queries the current kernel DMA buffer attributes, placing the results in ***query***. The element ***physaddr*** is set to the address that an I/O device would use to access the dma buffer.int **ioctl** (int *filedes*, **ALF_DMABUF_MAP**, *dmabuf_map_t *map*)

```
typedef struct {
      unsigned long vaddr;
      int len;
} dmabuf_map_t;
```

This function is used to map the dma buffer into user address space.

B. S3/Virge Device Driver

int **open** (char **path*, int *oflag*, …)

The open function establishes a connection with the S3 device given by ***path***. The device name for S3 is **/dev/alfast_t3<n>** where **<n>** indicates the device instance.

Int **close** (int *filedes*)

The close function disconnects access to the device.

void ***mmap** (void **addr*, size_t *len*, int *prot*, int *flags*, int *filedes*, off_t *off*)

This function is used to map S3 device resources into the application address space. Refer to Unix man-pages for details on the call. The ***off*** argument specifies the offset into the address space. Since there are two distinct address spaces for the S3 (one for display memory, and control registers), the high order nibble of ***off*** specifies which address space to map.

0x0xxxxxxx	map display memory
0x1xxxxxxx	map control registers

int **ioctl** (int *filedes*, ALF_IOCTL_PCICFG_READ_BYTE, unsigned char *result)

int **ioctl** (int *filedes*, ALF_IOCTL_PCICFG_READ_SHORT, unsigned short *result)

int **ioctl** (int *filedes*, ALF_IOCTL_PCICFG_READ_LONG, unsigned long *result)

int **ioctl** (int *filedes*, ALF_IOCTL_PCICFG_WRITE_BYTE, unsigned char result)

int **ioctl** (int *filedes*, ALF_IOCTL_PCICFG_WRITE_SHORT, unsigned short result)

int **ioctl** (int *filedes*, ALF_IOCTL_PCICFG_WRITE_LONG, unsigned long result)

These functions perform PCI configuration space reads and writes to the S3 device.

C. PCI to PCI Bridge Device Driver

int **open** (char **path*, int *oflag*, …)

The open function establishes a connection with the bridge device given by ***path***. The device name is **/dev/alfast_bridge<n>** where **<n>** indicates the device instance.

int **close** (int *filedes*)

The close function disconnects access to the device.

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_READ_BYTE**, unsigned char **result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_READ_SHORT**, unsigned short **result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_READ_LONG**, unsigned long **result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_WRITE_BYTE**, unsigned char *result*)

int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_WRITE_SHORT**, unsigned short *result*)

Int **ioctl** (int *filedes*, **ALF_IOCTL_PCICFG_WRITE_LONG**, unsigned long *result*)

These functions perform PCI configuration space reads and writes to the TM device.

IX. <u>TROUBLESHOOTING</u>

There are several things you can try before you call Alacron Technical Support for help.

_____ Make sure the computer is plugged in. Make sure the power source is on.

_____ Go back over the hardware installation to make sure you didn't miss a page or a section.

_____ Go back over the software installation to make sure you have installed all necessary software.

_____ Run the Installation User Test to verify correct installation of both hardware and software.

_____ Run the user-diagnostics test for your main board to make sure it's working properly.

_____ Insert the Alacron CD-ROM and check the various Release Notes to see if there is any information relevant to the problem you are experiencing.

The release notes are available in the directory: **\usr\alacron\alinfo**

_____ Compile and run the example programs found in the directory: **\usr\alacron\src\examples**

_____ Find the appropriate section of the Programmer's Guide & Reference or the Library User's Manual for the particular library and problem you are experiencing. Go back over the steps in the guide.

_____ Check the programming examples supplied with the runtime software to see if you are using the software according to the examples.

_____ Review the return status from functions and any input arguments.

_____ Simplify the program as much as possible until you can isolate the problem. Turning off any operations not directly related may help isolate the problem.

_____ Finally, first **save your original work**. Then remove any extraneous code that doesn't directly contribute to the problem or failure.

X. ALACRON TECHNICAL SUPPORT

Alacron offers technical support to any licensed user during the normal business hours of 9 a.m. to 5 p.m. EST. We offer assistance on all aspects of processor board and PMC installation and operation.

A. Contacting Technical Support

To speak with a Technical Support Representative on the telephone, call the number below and ask for Technical Support:

Telephone: **603-891-2750**

If you would rather FAX a written description of the problem, make sure you address the FAX to Technical Support and send it to:

Fax: **603-891-2745**

You can email a description of the problem to support@alacron.com

Before you contact technical support have the following information ready:

_____ Serial numbers and hardware revision numbers of all of your boards. This information is written on the invoice that was shipped with your products.

_____ Also, each board has its serial number and revision number written on either in ink or in bar-code form.

_____ The version of the ALRT, ALFAST, or FASTLIB software that you are using.

_____ You can find this information in a file in the directory: **\usr\alfast\alinfo**

_____ The type and version of the host operating system, i.e., Windows 98.

_____ Note the types and numbers of all your software revisions, daughter card libraries, the application library and the compiler

_____ The piece of code that exhibits the problem, if applicable. If you email Alacron the piece of code, our Technical-Support team can try to reproduce the error. It is necessary, though, for all the information listed above to be included, so Technical Support can duplicate your hardware and system environment.

B. ***Returning Products for Repair or Replacements***

Our first concern is that you be pleased with your Alacron products.

If, after trying everything you can do yourself, and after contacting Alacron Technical Support, you feel your hardware or software is not functioning properly, you can return the product to Alacron for service or replacement. Service or replacement may be covered by your warranty, depending upon your warranty.The first step is to call Alacron and request a "Return Materials Authorization" (RMA) number.This is the number assigned both to your returning product and to all records of your communications with Technical Support. When an Alacron technician receives your returned hardware or software he will match its RMA number to the on-file information you have given us, so he can solve the problem you've cited.

When calling for an RMA number, please have the following information ready:

_____ Serial numbers and descriptions of product(s) being shipped back

_____ A listing including revision numbers for all software, libraries, applications, daughter cards, etc.

_____ A clear and detailed description of the problem and when it occurs

_____ Exact code that will cause the failure

_____ A description of any environmental condition that can cause the problem

All of this information will be logged into the RMA report so it's there for the technician when your product arrives at Alacron.Put boards inside their anti-static protective bags. Then pack the product(s) securely in the original shipping materials, if possible, and ship to:

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Clearly mark ***the outside of your package:***

Attention **RMA #80XXX**

Remember to include your return address and the name and number of the person who should be contacted if we have questions.

C. Reporting Bugs

We at Alacron are continually improving our products to ensure the success of your projects. In addition to ongoing improvements, every Alacron product is put through extensive and varied testing. Even so, occasionally situations can come up in the fields that were not encountered during our testing at Alacron.

If you encounter a software or hardware problem or anomaly, please contact us immediately for assistance. If a fix is not available right away, often we can devise a work-around that allows you to move forward with your project while we continue to work on the problem you've encountered.

It is important that we are able to reproduce your error in an isolated test case. You can help if you create a stand-alone code module that is isolated from your application and yet clearly demonstrates the anomaly or flaw.

Describe the error that occurs with the particular code module and email the file to us at:

support@alacron.com

We will compile and run the module to track down the anomaly you've found.

If you do not have Internet access, or if it is inconvenient for you to get to access, copy the code to a disk, describe the error, and mail the disk to Technical Support at the Alacron address below.

If the code is small enough, you can also:

FAX the code module to us at 603-891-2745

If you are faxing the code, write everything large and legibly and remember to include your description of the error.

When you are describing a software problem, include revision numbers of all associated software.

For documentation errors, photocopy the passages in question, mark on the page the number and title of the manual, and either FAX or mail the photocopy to Alacron.

Remember to include the name and telephone number of the person we should contact if we have questions.

Alacron Inc.
71 Spit Brook Road, Suite 200
Nashua, NH 03060
USA

Telephone: 603-891-2750
FAX: 603-891-2745

Web site:
http://www.alacron.com/

Electronic Mail:
sales@alacron.com
support@alacron.com