FASTSERIES

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

FAST CAPTURE
FAST PROCESSING
FAST RESULTS

FASTSERIES PCI BOARD

FastVision
FastImage 1300
FastFrame 1300

FAST SERIES PMCs

FastMem
Fast4 1300
Fast I/O 1300

30002-00148

COPYRIGHT NOTICE

Copyright ã 2002 by Alacron Inc.

All rights reserved. This document, in whole or in part, may not be copied, photocopied, reproduced, translated, or reduced to any other electronic medium or machine-readable form without the express written consent of Alacron Inc.

Alacron makes no warranty for the use of its products, assumes no responsibility for any error, which may appear in this document, and makes no commitment to update the information contained herein. Alacron Inc. retains the right to make changes to this manual at any time without notice.

Document Name: ALFAST RT SW Programmer's Guide & Reference Manual

Document Number: 30002-00148

Revision History: 2.1 April 21, 2003

Trademarks:

Alacronâ is a registered trademark of Alacron Inc.
AltiVecÔ is a trademark of Motorola Inc.
Channel LinkÔ is a trademark of National Semiconductor
CodeWarriorâ is a registered trademark of Metrowerks Corp.
FastChannelâ is a registered trademark of Alacron Inc.
FastSeriesâ is a registered trademark of Alacron Inc.
Fast4â, FastFrame 1300â, FastImageâ, FastI/Oâ, and FastVision® are registered trademarks of Alacron Inc.
FireWireÔ is a registered trademark of Apple Computer Inc.
3MÔ is a trademark of 3M Company
MS DOSâ is a registered trademark of Microsoft Corporation
SelectRAMÔ is a trademark of Xilinx Inc.
SolarisÔ is a trademark of Sun Microsystems Inc.
TriMediaÔ is a trademark of Philips Electronics North America Corp.
Unixâ is a registered trademark of Sun Microsystems Inc.
VirtexÔ is a trademark of Xilinx Inc.
WindowsÔ, Windows 95Ô, Windows 98Ô, Windows 2000Ô, and Windows NTÔ are trademarks of Microsoft

All trademarks are the property of their respective holders.

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

Copyright Notice ……………………………………………………………………………….. *ii*
Table of Contents ……………………………………………………………………………… *iii*
Manual Figures & Tables …………………………………………………………..……… *viii*
Other Alacron Manuals ………………………………………………………………………….. *ix*

I. INTRODUCTION ***1***

A. Alacron FastSeries Processor Boards **1**

B. FastSeries Hardware Configurations **2**
1. Processor and Memory 2
2. Analog and Digital Input 2
3. Analog and Digital Output 2
4. PMC Connectors 3
5. Alacron PMC Daughter cards 3

C. Development and Execution Environments **4**
1. Philips Software Development Environment (SDE) 4
2. TMMAN Execution Environment 4
3. ALRT Execution Environment 4

D. Using the SDE Library for Capture **5**

E. ALFAST Support Libraries **5**
1. Capture Library 6

F. Build Instructions for SDE **8**

G. Building Programs on Linux Systems **9**
1. Compiling Device Driver 9
2. Building Host Programs 9
3. Building TriMedia Programs 9

II. PROGRAMMER'S GUIDE ***11***

A. Capture Library Functions **11**
1. Include File 11
2. Initialize Capture Library 11
3. Load Capture Profile 11
4. Query and Set Capture Attributes 12
5. Set Up Capture Parameters 13
6. Setup for "Raw" Data 13
7. Declare Callback Function 15
8. Start the Capture Operation 16
9. Retrieve Status and Buffer Information 16
10. Stop Continuous Capture Operation 17
11. Capture Library Example 17
12. Board Configuration Function 18
13. Timing and Delay Functions 18

B. Digital Output Library Functions **19**
1. Include File 19
2. Initialize Digital Output Library 19
3. Load and Access Digital Output Profile 20
4. Query and Set Digital Output Attributes 20
5. Set Up Digital Output Parameters 21
6. Declare Callback Function 22

7. Digital Output Driver Enable and Disable ... 23
8. Start the Digital Output Operation ... 24
9. Retrieve Status and Buffer Information ... 24
10. Stop Continuous Digital Output Operation ... 24
11. Digital Output Example ... 24

C. S3 Streaming Library Functions ... 25
1. Initialize S3 Streaming Library ... 25
2. Set Up LPB Input ... 26
3. Set Up Streaming Output Window Parameters ... 26
4. Start the Video Streaming Operation ... 27
5. Retrieve Status Information ... 27
6. Stop Continuous Video Streaming Operation ... 27
7. Video Streaming Example ... 27

D. TV Output Library Functions ... 28
1. Initialize TV Output Library ... 30
2. Allocate New Control Context ... 30
3. Set Up Control Variables ... 30
4. Turn on TV Output ... 31
5. Turn Off TV Output ... 31
6. Query TV Output Parameters ... 31
7. Reading and Setting the TV Output Color Palette ... 32
8. Synchronizing TV Output with Screen Redraw ... 32
9. TV Output Example ... 32

E. Analog Output Library Functions ... 33
1. S3 ViRGE GX2 Driver Controls for Windows NT ... 33
2. S3 ViRGE GX2 Driver Controls for Linux ... 34
3. S3 Bridge Control ... 34
4. Include File ... 34
5. Open S3 Video Memory ... 34
6. Close S3 Video Memory ... 35
7. Library Version ... 35
8. Handling Errors ... 35
9. Getting Display Parameters ... 36
10. TriMedia-Controlled Mode Functions ... 36
11. Using a Graphics Primitive Buffer in SDRAM ... 37
12. Using a Graphics Primitive Buffer in Display Memory ... 38
13. Getting Integer Color Values ... 38
14. Drawing Graphic Objects in a GRP Buffer ... 39
15. Drawing Text Objects in the GRP Buffer ... 40
16. Advanced Functions ... 42

III. PROGRAM EXAMPLES ... 43

A. Host Program Example ex1.c ... 43

B. TriMedia Program Example frame.c ... 45

C. TriMedia Program Example analog.c ... 48

D. TriMedia Program Example ntsc.c ... 51

E. TriMedia Program Example vidout.c ... 57

F. TriMedia Program Example send.c ... 63

G. TriMedia Program Example recv.c ... 65

H. TriMedia Program Example vdatest.c ... 68

IV. CAPTURE LIBRARY REFERENCE 72
A. Include Files 72
B. Libraries 72
C. Quick Reference 72
D. Returns and Error Codes 72
alf_capture_attach 74
alf_capture_load_profile 75
alf_capture_query_attribute 76
alf_capture_set_attribute 77
alf_capture_set_control 78
alf_capture_start 80
alf_capture_status 83
alf_capture_stop 84
alf_config_query 85
alf_palreg_write 86
alf_timing_start 87
alf_timing_elapsed 88
alf_timing_msec_elapsed 89
alf_timing_msec_delay 90
alf_timing_usec_delay 91
V. DIGITAL OUTPUT LIBRARY REFERENCE 92
A. Include Files 92
B. Libraries 92
C. Quick Reference 92
D. Returns and Error Codes 93
alf_digout_attach 94
alf_digout_enable 95
alf_digout_disable 96
alf_digout_load_profile 97
alf_digout_query_attribute 98
alf_digout_set_attribute 99
alf_digout_set_control 100
alf_digout_start 101
alf_digout_status 103
alf_digout_stop 104
VI. S3 VIDEO STREAMING LIBRARY REFREFERENCE 105
A. Include Files 105
B. Libraries 105
C. Quick Reference 105
D. Returns and Error Codes 106
alf_vstream_attach 107
alf_vstream_set_lpb 108
alf_vstream_set_window 109
alf_vstream_start 111
alf_vstream_status 112
alf_vstream_stop 113
VII. TV OUTPUT LIBRARY REFERENCE 114

A. Include Files **114**

B. Libraries **114**

C. Quick Reference **114**

D. Returns and Error Codes **114**

alf_tvout_attach 116
alf_tvout_control 117
alf_tvout_new 119
alf_tvout_params 121
alf_tvout_rdpalette 122
alf_tvout_start 123
alf_tvout_stop 124
alf_tvout_wait_eof 125
alf_tvout_wrpalette 126

VIII. ICP SHARING LIBRARY REFERENCE ***127***

A. Include Files **127**

B. Libraries **127**

C. Quick Reference **127**

D. Returns and Error Codes **127**

alf_icp_acquire 129
alf_icp_attach 131
alf_icp_release 132

IX. ANALOG OUTPUT LIBRARY REFERENCE ***133***

A. Include Files **133**

B. Libraries **133**

C. Quick Reference **133**

D. Handling Errors **134**

grp_alloc 135
grp_buf_width 136
grp_char_width 137
grp_cls 138
grp_dot_dash_line 139
grp_draw_buf 140
grp_draw_char 141
grp_draw_string 142
grp_fill 143
grp_free 144
grp_free_font 145
grp_line 146
grp_load_font 147
grp_point 148
grp_point_addr 149
grp_rect_fill 150
grp_string_width 151
vda_close 152
vda_cls 153
vda_errno 154
vda_errorprt 155
vda_grp_vram 156

vda_grp_vram_free ... 157
vda_host_controlled ... 158
vda_ioctl(VDA_GET_PARAMS) ... 159
vda_ioctl(VDA_SET_RESOLUTION) ... 160
vda_ioctl(VDA_WAIT_EOF) ... 161
vda_ioctl(VDA_WAIT_WRITE) ... 162
vda_map_rgb ... 163
vda_open ... 164
vda_read ... 165
vda_region_copy ... 167
vda_sync ... 169
vda_version ... 170
vda_write ... 171
vda_write_yuv ... 173

X. CAPTURE PROFILE FORMAT ... 175

A. Example Capture Profile ... 175

B. Section Names ... 180

C. Input Line Format ... 180

D. Conditional Input Lines ... 180

E. Include Files ... 181

F. Comments ... 181

XI. DIGITAL OUTPUT PRIFILE FORMAT ... 182

XII. VDA INITIALIZATION FILE ... 184

A. Sample vda.ini File ... 184

B. Conditional Input Lines ... 185

C. Include Files ... 185

D. Format of Input Lines ... 185

E. Local S3 on FastSeries Board ... 185

F. Other Display Controller ... 186

XIII. TROUBLESHOOTING ... 187

XIV. ALACRON TECHNICAL SUPPORT ... 188

A. Contacting Technical Support ... 188

B. Returning Products for Repair or Replacements ... 189

C. Reporting Bugs ... 190

MANUAL FIGURES & TABLES

FIGURE	PAGE	SUBJECT	TABLE	PAGE	SUBJECT
1	1	Generalize FastSeries Processor Board in a PC Chassis			
2	4	Development and Execution Environments			
3	6	ALFAST Runtime Support Libraries			
4	6	ALFAST Capture Library			
5	8	S3 Libralies and S3 Operating Modes			

OTHER ALACRON MANUALS

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.

- 30002-00150 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. Alacron FastSeries Processor Boards

Figure 1 is a generalized block diagram of a FastImage or FastFrame board.

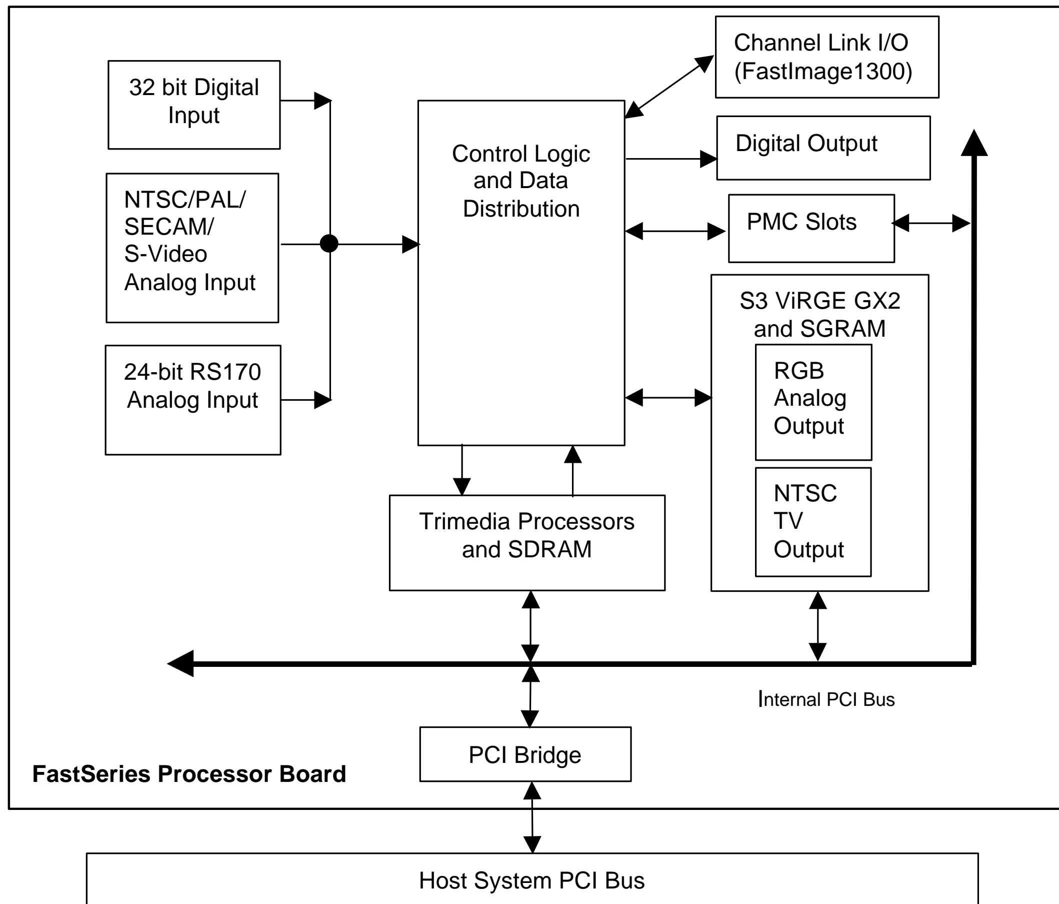

Figure 1. Generalized FastSeries Processor Board in a PC Chassis

B. FastSeries Hardware Configurations

The FastImage and FastFrame processor boards are available in a variety of input, output, and processor/memory configurations. The Hardware User's Guide for each system gives complete details. This section briefly reviews the configuration options.

1. Processor and Memory

The FastSeries boards can be configured with TriMedia processors as follows:

- The FastImage1100 board can have one to four TriMedia TM1100 processors.
- The FastImage1300 board can have one or four TriMedia TM1300 processors.
- The FastFrame board can have one or two TM1100 processors.
- The Fast4 PMC daughter card can have one to four TM1300 processors.
- The FastIO PMC daughter card has one TM1100 processor.
- 8MB or 16MB SDRAM for each processor (up to 32MB on the FastFrame). All TriMedia on a board have the same size SDRAM.

2. Analog and Digital Input

The FastImage and FastFrame processor boards and the FastIO PMC card have four inputs, TAP1 through TAP4. Each TAP (8 bits data, 2 bits control, pixel clock) can be configured for analog input or for digital input (but not both). TAP1, TAP2, and TAP3 can be populated as either digital input or RS170 compatible analog input. TAP4 can be either digital input or composite/component analog input (NTSC/PAL/SECAM or S-Video).

a) Analog Input

The Analog input option includes the SAA7111A Video Input Processor, three eight-bit analog input channels with A/D, RS422 digital video control and clock inputs (for analog line scan or other non-composite sync cameras), and RS422 digital camera/strobe control and clock outputs (for analog line scan or other non-composite sync cameras).

b) Digital Input

The Digital input option includes 32 bits of RS422 differential digital input. RS422 receivers are standard; differential LVDS receivers is available as options.

3. Analog and Digital Output

The FastImage and FastFrame boards can be configured with analog and/or digital output, or with no output.

a) Analog Output

The analog output option includes the S3 VirGE GX2 graphics accelerator with associated BIOS ROM and 4MB of SGRAM.

b) TV Output

Analog RGB output can be directed to the system console. NTSC/PAL/S-Video output can simultaneously go to the TV monitor connector.

c) Digital Output and FastChannel

The digital output drivers and connectors on the Fast Series board drive external hardware interfaces or digital video recorders. The FastImage1100 provides 32 bits of RS422 differential digital output drivers with RS422 digital video control and clock outputs. The FastFrame1100 provides 16 bits of digital output. On the FastImage1300, the FastChannel interface provides both digital output and digital input, 32 bits in either direction.

The digital output lines are shared with other signals, depending on the model:

- On the FastFrame1100, eight of the digital output lines are shared with the S3 Video Streaming lines to the S3 in 16-bit mode, and the other eight are shared with the EVIP analog input device in 16-bit mode.
- On the FastImage1100, the digital output lines are shared with PMC connector #1, the standard PMC.
- On the FastImage1300, the data and control lines to the digital output drivers (the FastChannel outputs) are also routed to the Channel Link outputs. FastChannel inputs are not affected.

d) Channel Link

On the FastImage1300, a Channel Link interface is provided for direct connection of digital cameras that support this industry standard.

4. PMC Connectors

The FastImage1100 and FastImage 1300 provide two PMC slots for connecting daughter cards to the internal PCI bus. The FastFrame1100 has one PMC daughter card slot.

The PMC connector signals are shared with other I/O on the FastImage models:

- On the FastImage1100, the digital output lines are shared with PMC connector #1, the standard PMC connector.
- On the FastImage1300, the Channel Link interface shares its receive lines with PMC connector #2, the connector used for the Fast4 or Fastmem daughter cards.

5. Alacron PMC Daughter cards

Alacron PMC daughter cards include:

- The Fast4 supplies one to four additional TriMedia and associated SDRAM.
- The FastIO supplies additional digital or analog inputs.
- The FastMem supplies up to 512 MB of global SDRAM.

The FastImage, FastFrame, FastIO, and Fast4 products are described in the Hardware Users Manuals listed at the end of this Introduction section.

C. Development and Execution Environments

Figure 2 diagrams the components of the software environments available from Alacron or from Philips TriMedia Corporation for developing and running TriMedia application programs on the FastSeries processor boards.

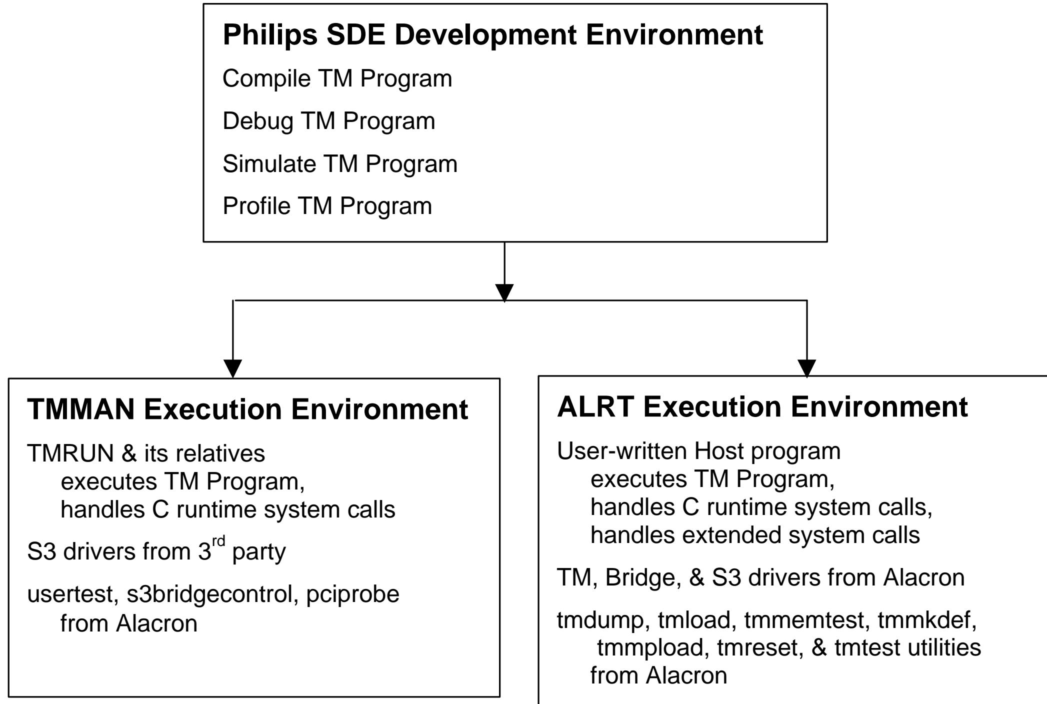

Figure 2. Development and Execution Environments

1. Philips Software Development Environment (SDE)

The Philips Software Development Environment (SDE) for the TriMedia processor contains utilities to develop, compile, simulate, and debug TriMedia application programs. SDE is supported on a variety of operating systems. Refer to the documentation on the CD ROM with the Philips SDE software for complete information.

2. TMMAN Execution Environment

The Philips TMMAN execution environment provides utilities for loading and executing Trimedia programs. It is based on a set of drivers and libraries for loading and executing TriMedia programs. Refer to the documentation on the CD ROM with the Philips SDE software for complete information.

3. ALRT Execution Environment

ALRT (Alacron Runtime) is a runtime environment for use with Alacron Fast Series boards equipped with Trimedia processors. ALRT provides runtime support under a variety of host operating system environments, including Solaris/Sparc, Solaris/x86, Linux, LinuxPPC, and WinNT. ALRT is designed to be:

- Independent from Philips TMMAN runtime.
- Portable and retargetable to new host operating systems with minimal effort.

- Able to run TriMedia programs developed under Philips SDE.

The ALRT software consists of a Host library and a TriMedia library.

The ALRT Host library is based on a subset of the Alacron processor board API. In addition, the ALRT Host library supports the TriMedia C runtime library.

The ALRT TriMedia library provides a small set of functions for IPC with the Host library.

Refer to the *ALRT Runtime Software Programmers Guide & Reference* for complete information.

D. Using the SDE Library for Capture

The Video Input Device Component allows Philips SDE library functions to access the SAA7111A video decoder for data capture. This feature is relevant for applications that wish to use the Philips functions instead of Alacron's ALFAST capture library functions. Two issues require consideration:

- PAL hardware configuration must be performed; since the Philips SDE functions don't do this. The easiest way to accomplish this is to have a TM program perform **alf_capture_attach, alf_capture_load_profile**, then terminate (note, PALS may be loaded only by the first TM processor). A subsequent program may then use SDE Video Input functions (viOpen, etc.)
- It is necessary to link the board support package for the FastSeries board. This may be accomplished by linking the file **alfast20.o** from the $(ALFAST)\lib directory. When this is done, SDE example code such as **vitest** and **exIoVTransICP** will compile and operate correctly.

E. ALFAST Support Libraries

Figure 3 diagrams the ALFAST Runtime Support Libraries, which are documented in this manual. The ALFAST libraries are supported in both the TMMAN and ALRT execution environments.

The ALFAST Runtime Support Libraries are the following:

- The Alacron ALFAST Capture library handles capture operations through one of the input channels
- The Alacron ALFAST S3 Libraries. The ALFAST S3 libraries in turn control the execution of the S3 GX2 graphics accelerator in various display modes.
- The Alacron ALFAST Digital Output Library directs data to the digital output port
- The optional Alacron FastSeries Library provides an extensive selection of vector and image-processing functions. The Function Library is ordered separately from the ALFAST Runtime Support Libraries.

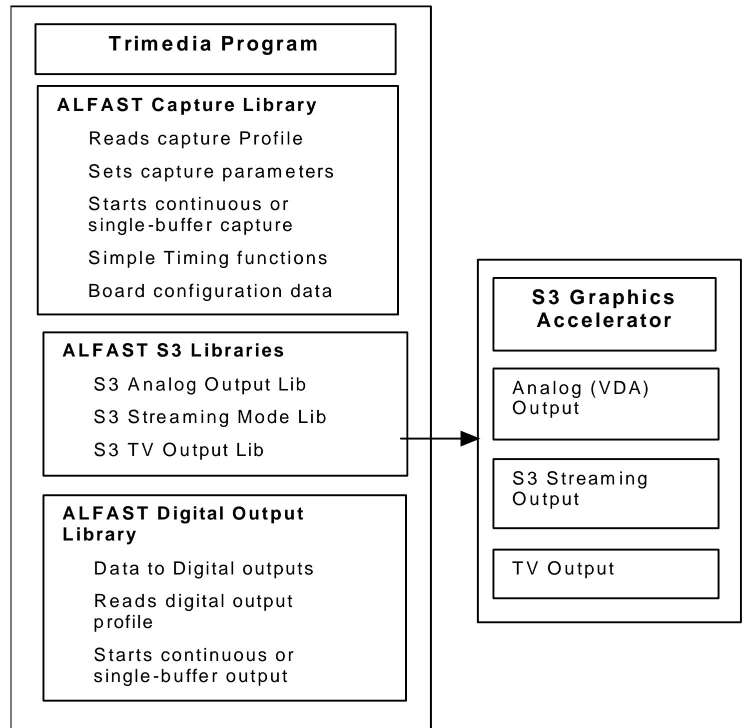

Figure 3. ALFAST Runtime Support Libraries

1. Capture Library

The ALFAST Capture Library provides a simple API for the TriMedia program to capture buffers of data from a specific input channel.

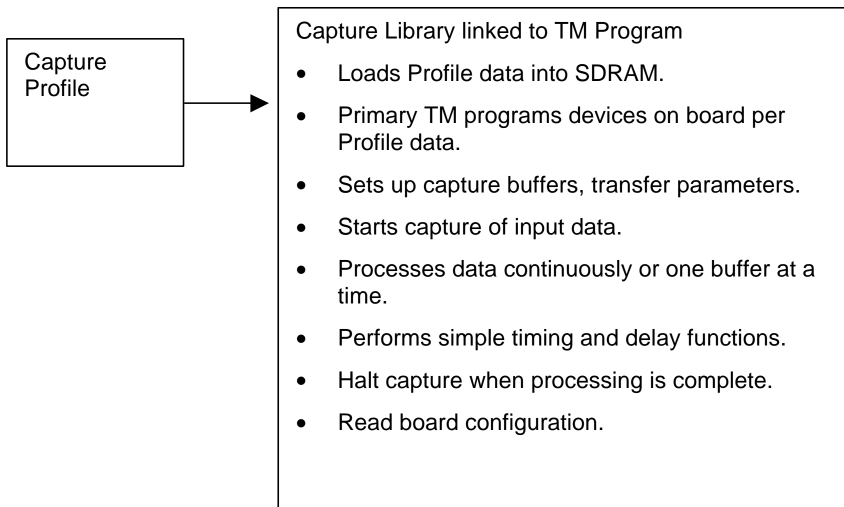

Figure 4. ALFAST Capture Library

Capture library functions allow the TriMedia program to:

- Read a Capture Profile from PC memory into TriMedia SDRAM. The Trimedia program uses the Profile data to determine buffer sizes and other capture parameters. The Primary TriMedia on the FastImage or FastFrame processor board also uses the data from the Profile to program the control and data path logic on the board.
- Specify the number and size of buffers to be allocated for the capture.
- Select Continuous or Single-buffer capture mode. In Continuous capture mode, data flows into a ring of buffers continuously. In Single-buffer mode, only one buffer of input data is captured.
- Set up a callback function to be executed when a buffer of data is ready to be processed. Callbacks can be used in both Continuous and Single-buffer mode to speed handling.
- Start single or continuous capture.
- Poll for completion status of any capture operation (as an alternative to using a callback function), and obtain capture buffer addresses.
- Stop a continuous capture operation.
- Perform simple timing and delay functions.
- Access FastSeries board configuration data. The TriMedia processor can retrieve data such as the board type, processor number, PCI base addresses, and PCI IRQ level. In addition, the processor can determine the presence of digital input, analog input, digital output, and analog output channels, when that information is useful.

a) S3 Analog Output Library

The S3 Analog Output library (Figure 5) allows the TriMedia program to access the system console display in one of two modes—Host-controlled and TriMedia-controlled. In Host-Controlled mode, the S3 serves as the PC system display. The TriMedia programs change the display by directly or indirectly writing S3 display memory, using an API based on Alacron's VDA daughter card library. Display parameters can be controlled by screen utilities in Windows/NT. In TriMedia-controlled mode, the TriMedia can clear the display and set the display resolution via the S3.

b) S3 Streaming Mode Library

The S3 Streaming Mode library (Figure 5) allows the TriMedia program to control the S3 in Streaming mode. In this mode, input video data connects from the crosspoint switch (FastImage) or FPGA (FastFrame) to the Streaming Input port on the S3, then to a window on the console. The S3 can be set up to perform alpha blending and resizing of the Streaming mode image in the window.

> Note: The S3 Streaming Mode Library is not supported by early versions of the hardware.

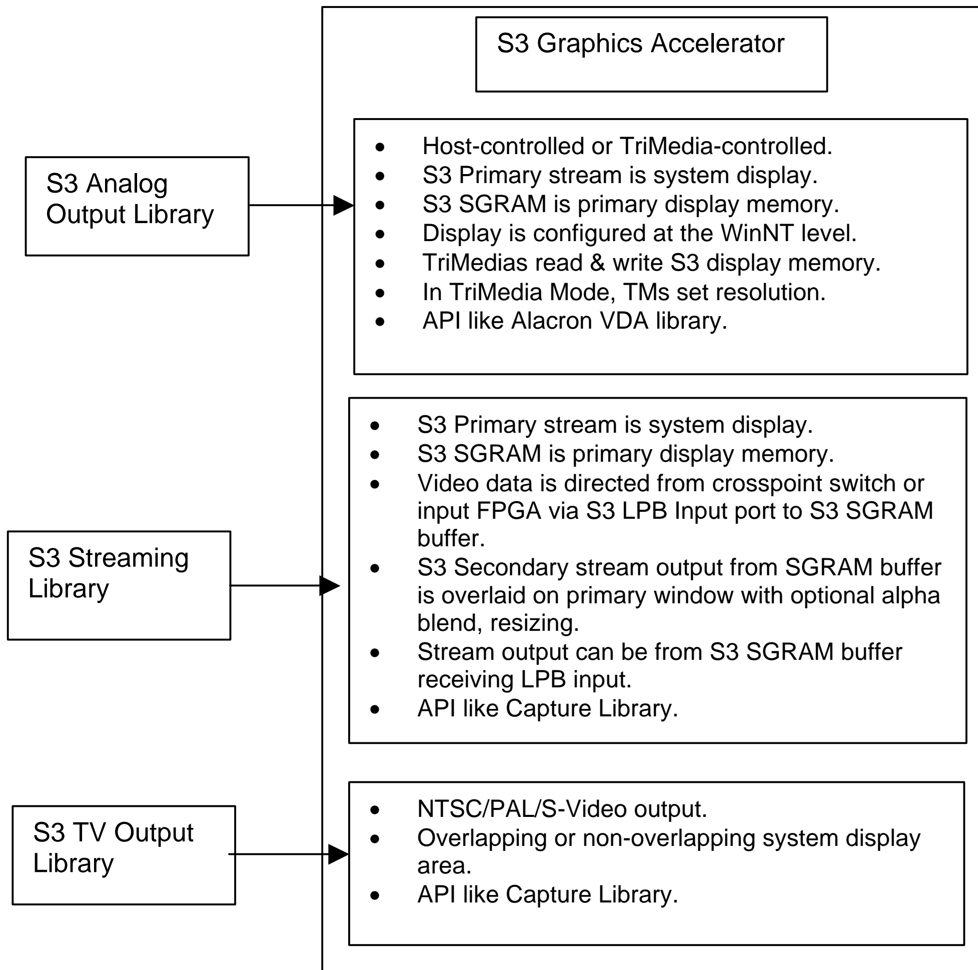

Figure 5. S3 Libraries and S3 Operating Modes

c) S3 TV Output Library

The FastImage and FastFrame boards can direct S3 display memory to a television-compatible video output stream (NTSC, NTSC-Japan or PAL) in composite or S-Video formats. The TV output image shares the video memory with the main display; the memory regions can be overlapping or non-overlapping.

d) Digital Output Library

The Digital Output library allows the TriMedia program to direct digital data from the crosspoint switch or FPGA to the Digital output drivers. The configuration is specified in a Digital Output profile (similar to the Capture profile discussed above), which is set into the programmable devices by TriMedia 0. The ALFAST Digital Output runtime software provides an API to set up and operate the digital output functionality of the Fast Series boards.

F. Build Instructions for SDE

TM applications are built using the standard Philips SDE provided compilers and linkers. The following is a typical NMAKE makefile.

```
# we presume that environment variables:
#       ALFAST  WinNT subdirectory of ALFAST
```

```
installation
#       TCS             Philips Trimedia installation directory

ENDIAN          = -el
CC              = tmcc
CFLAGS          = -I$(ALFAST)\..\include $(ENDIAN)
LD              = tmcc
LDFLAGS = $(ENDIAN) -host WinNT -L$(ALFAST)\..\lib

.c.o:
        $(CC) -c $(CFLAGS) $*.c
OBJS            = test1.o test2.o test3.o

test.out:       $(OBJS)
        $(LD) -o $@ $(LDFLAGS) $(OBJS) -lvda -lalfast
```

The file **$(ALFAST)\..\tools\include\tmnt.inc** contains the standard NMAKE declarations. The preceding makefile may be rewritten using declarations from **tmnt.inc**:

```
ROOTDIR         = $(ALFAST)
!include $(ROOTDIR)\tools\include\tmnt.inc

OBJS            = test1.o test2.o test3.o

test.out:       $(OBJS)
        $(LD) -o $@ $(LDFLAGS) $(OBJS) $(LDTAIL)
```

G. 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 targeting 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**.

II. PROGRAMMER'S GUIDE

This Programmer's Guide shows how the TriMedia program accesses the Alacron ALFAST Runtime Libraries.

This revision of the ALFAST Programmer's Guide describes the Capture Library, the Digital Output Library, the S3 Streaming Mode Library, the S3 TV Output Library, and the S3 Analog Output Library.

A. Capture Library Functions

The program on a TriMedia processor (the *TM program*) accesses the input channels on the FastSeries board via calls to the ALFAST Capture Library.

1. Include File

The header file **alfast_tm.h** contains all needed definitions for the ALFAST Capture Library for the TriMedia program. The path to this file is **%ALFAST%\include**, where **%ALFAST%** represents the install directory for the runtime libraries.

2. Initialize Capture Library

Each TriMedia processor that is to receive input from the input channel accesses the Capture library. The first step is to allocate a variable of type **handle**, for example:

```
#include <alfast_tm.h>
...
handle CapHan;
```

The TM program next initializes the Capture Library and obtains a handle for use in later Capture library functions with a call to:

handle **alf_capture_attach** (int *device_instance*)

In all TM programs, the *device_instance* should be 0. For example:

```
CapHan = alf_capture_attach(0);
```

3. Load Capture Profile

The details about the data to be captured are provided to the TM program via a Capture profile in PC memory. The Capture library uses the profile information to set up all the programmable devices on the FastSeries board.

To read the profile from PC memory into TriMedia memory, the TM program calls

int **alf_capture_load_profile**(handle *h*, char **profile_filename*)

Argument *h* is the handle returned by **alf_capture_attach**. Argument *profile_filename* is the name of the profile (enclosed in quotes). The Capture library searches for the profile first in the current directory, then in **%ALFAST%\lib\capture**, where **%ALFAST%** is the install directory for the runtime software. For example:

```
ret = alf_capture_load_profile(CapHan, "NTSC60.cap");
```

Capture profiles have the following attributes available for reading by the application:

"**INPUT_NCOMP**"	integer number of elements in the pixel (e.g., 3 for RGB, 1 for monochrome)

"**INPUT_PIXELS_PER_LINE**"	integer	acquired pixels per line
"**INPUT_LINES_PER_BUFFER**"	integer	acquired lines per input buffer
"**INPUT_PIXEL_SIZE**"	integer	bytes per pixel
"**INPUT_NX**"	integer	number of usable pixels in X direction
"**INPUT_NY**"	integer	number of usable pixels in Y direction
"**INPUT_XOFFSET**"	integer	byte offset to first usable pixel
"**INPUT_YOFFSET**"	integer	line offset to first usable line
"**INPUT_SOURCE**"	string	"**RGB Analog**" "**RGB Digital**" "**Monochrome Analog**" "**Monochrome Digital**" "**NTSC Analog**" "**PAL Analog**" "**SECAM Analog**" "**S-Video**"

Notes:

Attributes INPUT_NX, INPUT_NY, INPUT_XOFFSET, and INPUT_YOFFSET are not supported in Release 1.4.1 of ALFAST.

TriMedia processor 0 on the board is the primary processor; only this processor can program devices on the board such as crosspoint switches, FPGAs, CPLDs, MAXDACs, and UARTs. If the processor executing the alf_capture_load_profile function is processor 0, then, in addition to reading the profile attributes into TriMedia memory, *the Capture library executes internal commands to program all devices on the FastSeries board as specified by the profile attributes*.

4. Query and Set Capture Attributes

The TriMedia program can query the value of any attribute from the capture profile loaded with **alf_capture_load_profile**. The program first sets up a blank structure of type **alf_attribute_t**. The definition for this structure type is:

```
typedef struct {
        union {
                unsigned long attr_lvalue;
                float attr_fvalue;
                char *attr_svalue;
                } attr_value;
        int type;
        } alf_attribute_t;
```

Then the program calls the following routine, passing it the capture handle from the **alf_capture_attach** call, the name of the attribute to be returned, and the address of the **alf_attribute_t** variable.

int **alf_capture_query_attribute** (handle *h*, char **name*, alf_attribute_t **p*)

When the function returns, the structure has been filled in with the value, which can be an integer, a float, or a string. The type of value returned can be read from the type element of the **alf_attribute_t** structure. The following predefined tokens may be used to match the type element: **ALF_ATTRIBUTE_LVALUE**, **ALF_ATTRIBUTE_FVALUE**, **ALF_ATTRIBUTE_SVALUE**.

When the type of value is known, the program can use the following defined tokens to access the union element with the desired type:

```
#define lvalue attr_value.attr_lvalue
#define fvalue attr_value.attr_fvalue
#define svalue attr_value.attr_svalue
```

For example:

```
alf_attribute_t p_attrib;
char profile_source[20];

ret = alf_capture_query_attribute(CapHan, "INPUT_SOURCE",
                                                      &p_attrib);
if (p_attrib.type == ALF_ATTRIBUTE_SVALUE)
       memset (profile_source, p_attrib.svalue);
```

It may be desired to set the value of an attribute from within the TriMedia program. The program first sets up an **alf_attribute_t** structure with a value of the correct type assigned to the appropriate union element, and the type member set to a token for the value type. The program then calls the following function, passing it the capture library handle, the name of the attribute to be set, and the address of the attribute structure containing the value to be set.

int **alf_capture_set_attribute** (handle *h*, char **name*, alf_attribute_t **p*)

For example:

```
alf_attribute_t p_attrib;
p_attrib.lvalue = 1024;
p_attrib.type = ALF_ATTRIBUTE_LVALUE;

ret = alf_capture_set_attribute(CapHan, "INPUT_NX",
                                              &p_attrib);
```

5. Set Up Capture Parameters

The TriMedia application program sets up parameters to control the capture operation in an **alf_capture_control_t** structure. The **alf_capture_control_t** structure has the following elements:

```
typedef struct {
       int alf_capture_nbuf;  // Number of buffers
       int alf_capture_bufsize;      // Size of buffer in bytes
       int alf_capture_mode;  // Mode = Logical OR of
                                            // ALF_CAPTURE_CONTINUOUS,
                                            // ALF_CAPTURE_SINGLE,
                                            // ALF_CAPTURE_ODD_FIELD,
                                            // ALF_CAPTURE_EVEN_FIELD,
                                            // ALF_CAPTURE_INTERLACE
} alf_capture_control_t;
```

Set structure element **alf_capture_mode** to specify continuous or single-buffer capture. The use of the field capture modes for **alf_capture_mode** and the values of **alf_capture_ bufsize** and **alf_capture_nbuf** depend on the **INPUT_SOURCE** attribute in the Capture profile.

6. Setup for "Raw" Data

When the **INPUT_SOURCE** is Monochrome Digital, RGB Digital, Monochrome Analog, or RGB Analog, the **alf_capture_mode** can be either **ALF_CAPTURE_CONTINUOUS** or **ALF_CAPTURE_SINGLE**. These input sources are stored in TriMedia memory in "raw" format, that is, one frame per buffer of byte data. The internal organization of the data (pixel size, RGB fields, etc.) depends on the source, and the program must deal with them accordingly.

Monochrome Digital, RGB Digital, Monochrome Analog, and RGB Analog data sources should use a negative value (<0) for **alf_capture_bufsize**; this value sets the buffer size to one frame per buffer (the product of Capture profile attributes INPUT_PIXEL_SIZE * INPUT_PIXELS_PER_LINE * LINES_PER_BUFFER) automatically.

Monochrome Digital, RGB Digital, Monochrome Analog, and RGB Analog input programs could specify the number of buffers to use for the raw data. In single-buffer operation, one buffer will do. For continuous capture, the program needs a minimum of two buffers. The callback routine (see **alf_capture_start**) is called to process the first while the second fills. If the algorithm can complete each buffer before the second is ready, two buffers will do. The program can specify any number of buffers, which are filled and refilled in a round-robin order.

a) Setup for "Planar" Data

When INPUT_SOURCE is NTSC Analog, PAL Analog, SECAM Analog, or S-video, **alf_capture_mode** specifies how to handle the interlaced fields in odd and even frames, using the OR (**|**) of **ALF_CAPTURE_CONTINUOUS** or **ALF_CAPTURE_SINGLE** with one of the following tokens:

ALF_CAPTURE_ODD_FIELD for odd fields only, one field per buffer.

ALF_CAPTURE_EVEN_FIELD for even fields only, one field per buffer.

ALF_CAPTURE_ODD_FIELD | ALF_CAPTURE_EVEN_FIELD captures

both odd and even fields in alternating buffers.

ALF_CAPTURE_INTERLACE captures both odd and even fields and re-interlaces them in a single, double-length buffer. The first line in the buffer is the first line of the odd field, the second line in the buffer is the first line of the even field, the third line in the buffer is the second line of the odd field, the fourth line in the buffer is the second line of the even field, and so on.

When INPUT_SOURCE is NTSC Analog, PAL Analog, SECAM Analog, or S-video, **alf_capture_bufsize** is ignored. The buffer size is computed from INPUT_PIXELS_PER_LINE and INPUT_LINES_PER_BUFFER in the Capture profile. Each component of YUV is stored in a separate buffer (the "planar" format required by the TriMedia). YUV input is always 4:2:2, so U and V are half the length of Y. If ALF_CAPTURE_INTERLACE is specified, INPUT_LINES_PER_BUFFER should specify the number of lines per field; **alf_capture_set_control** will allocate buffers large enough for both fields.

> For all interlaced field options, the value of INPUT_LINES_PER_BUFFER should be the same as the number of lines per odd or even field (e.g., 240 lines for a 480-line interlaced frame); the ALF_CAPTURE_INTERLACE option automatically creates the frame-length buffer from the value for each field separately.

When INPUT_SOURCE is NTSC Analog, PAL Analog, SECAM Analog, or S-video, the element **alf_capture_nbuf** indicates the number of *sets* of YUV triple buffers needed by the algorithm. In single-buffer operation, one buffer set will do. For continuous capture, the program needs a minimum of two buffer sets. The callback routine (see **alf_capture_start**) is called to process the first set while the second set fills. If the algorithm can complete each buffer set before the second is ready, two buffer sets will do. The program can specify any number of buffer sets, which are filled and refilled in a round-robin order.

When the control structure is set up, the application calls:

int **alf_capture_set_control**(handle *h*, alf_capture_control_t **p*)

Now, subsequent capture operations will use this set of parameters. The buffers specified in the control structure are allocated in TriMedia memory. Pointers to the buffers are passed automatically to the callback function, described in the next section. Pointers can also be retrieved with the status-monitoring function described later.

7. Declare Callback Function

A capture operation can be monitored with a callback function or by polling the capture status. The polling method is described later in this chapter. The application program declares a callback function like the following:

```
void *alf_cap_callback (int cause, alf_capture_buf_t *buf
{
// Code here to check the cause, handle the data.
}
```

The callback function is called after each buffer of data has been captured in both Continuous and Single-buffer capture modes, when continuous capture has been stopped by **alf_capture_stop**, or when an error occurs. The *cause* and *buf* parameters are passed to the callback by the library. The following predefined values for *cause* are used:

ALF_CALLBACK_BUFFER_READY indicates a buffer is ready to process.

ALF_CALLBACK_STOPPED indicates capture was stopped with **alf_capture_stop**.

ALF_CALLBACK_ERROR indicated that an error occurred during capture.

When the cause is **ALF_CALLBACK_BUFFER_READY**, the argument ***buf*** points to a structure of type **alf_capture_buf_t** containing information on the buffer that most recently completed. The **alf_capture_buf_t** structure has the elements:

int alf_capture_buf_idx	The buffer index, ranging from zero (0) to the number of buffers minus 1 (nbuf-1).
int **alf_capture_field**	In the case of interlaced frames, this value specifies 0 for odd field, and 1 for even field.
Alf_buf_t **alf_capture_bufs**[3]	This is an array of **alf_buf_t** structures, each containing buffer information (see below).

For “raw” data transfers (Monochrome and RGB Digital and Analog inputs), only one **alf_buf_t** structure is used (accessed as **alf_capture_bufs[0]**). Predefined macro **alf_buf** allows a simpler form of reference. The macro definition is:

```
#define alf_buf        alf_capture_bufs[0]
```

Now, if *buf* is a pointer to an **alf_capture_buf_t**, the callback routine can access members of the first structure in the **alf_capture_bufs** array of **alf_buf_t** structures with the construct:

```
buf->alf_buf.structure_member
```

Which is equivalent to:

```
buf->alf_capture_bufs[0].structure_member
```

For YUV data transfers (NTSC, PAL, SECAM, and S-video inputs), three buffer structures are used. Use predefined tokens for the **alf_capture_bufs** indexes: **ALF_CAPTURE_BUFY** for Y, **ALF_CAPTURE_BUFU** for U, and **ALF_CAPTURE_BUFV** for V. Predefined macros **alf_bufY**, **alf_bufU**, and **alf_bufV** simplify references to the array of buffer structures:

```
#define alf_bufY       alf_capture_bufs[ALF_CAPTURE_BUFY]
#define alf_bufU       alf_capture_bufs[ALF_CAPTURE_BUFU]
#define alf_bufV       alf_capture_bufs[ALF_CAPTURE_BUFV]
```

Now, the callback routine can access these arrays of buffer structures with a construct such as:

```
buf->alf_bufU.structure_member
```

This construct is equivalent to:

```
buf->alf_capture_bufs[ALF_CAPTURE_BUFU].structure_member
```

The **alf_buf_t** structure contains the parameters for the buffer of data. Each **alf_buf_t** structure has the following members:

void ***alf_buf_data**	Pointer to the cache-aligned data buffer.
int **alf_buf_stride**	Row to row stride.
int **alf_buf_nrows**	Number of rows in the data buffer.
int **alf_buf_ncols**	Number of columns in the data buffer.
int **alf_buf_size**	Size in bytes of the data buffer.

Continuing the example, the callback routine (and the main program as well; see note below) can access the data in the buffer with a reference such as:

```
buf->alf_bufY.alf_buf_data           // Buffer of Y (luma) data
```

Note: The argument ***buf*** to the callback function points to static data, which remains a valid reference outside of the callback context.

8. Start the Capture Operation

Next, the application program calls

```
int alf_capture_start(handle h,
        void (*alf_cap_callback (int cause, alf_capture_buf_t *buf))
```

Capture begins continuously or in single-buffer mode. In either case, this function returns immediately (capture is asynchronous). Completion of the capture can be monitored with a callback function as described in the previous section or by polling for status as described next (for polling, the **alf_start_capture** call contains a NULL callback routine).

9. Retrieve Status and Buffer Information

To monitor capture status by polling or to obtain pointers to the buffers of captured data, the TriMedia program sets up a blank **alf_capture_status_t** structure. The definition for this structure type is:

```
typedef struct {
```

```
        int alf_capture_running;      // true if capture is running
        int alf_capture_bufno; // current capture buffer number
        int alf_capture_last_bufno;
                              // last completed buffer number
        alf_capture_buf_t *alf_capture_buflist;
                              // ptr to structure with buffer ptrs
        int alf_capture_error; // error flags
} alf_capture_status_t;
```

Then, the application program calls

int **alf_capture_status**(handle *h*, alf_capture_status t **alf_capture_p*)

When this function returns, the structure pointed to by *alf_capture_p* contains status on the capture operation. The application can reference any of the elements of the structure. The buffer pointers are returned in the structure pointed to by element **alf_capture_buflist**.

The call to **alf_capture_status** clears the error flags after returning their value.

Note: Error flags are undefined at Release 1.4.1.

10. Stop Continuous Capture Operation

Single-buffer captures terminate after one buffer has been captured. To terminate a continuous capture operation, the TriMedia application calls

int **alf_capture_stop**(handle *h)*

The continuous capture operation is halted. The callback function registered with **alf_capture_start**, is called with *cause* value ALF_CALLBACK_STOPPED.

11. Capture Library Example

Here is an example of some of the Capture library calls in a code fragment.

```
// (Outside main Routine)
...
// Set up callback function
void *alf_cap_callback(int cause, alf_capture_buf_t *buf) {
        // Code here to determine the cause of the callback
        // and process the buffer retrieved via buf.
}
...
// (Inside main routine)
...
// Attach to capture library
handle CapHan = alf_capture_attach((int)0);

// Load capture configuration profile
int ret = alf_capture_load_profile(CapHan, "NTSC60.cap");

// Create control structure and set it up
alf_capture_control_t CapCon;
CapCon.alf_capture_nbuf = 2;
CapCon.alf_capture_bufsize = 0; // Ignored for NTSC
CapCon.alf_capture_mode = (ALF_CAPTURE_CONTINUOUS |
                              ALF_CAPTURE_ODD);

ret = alf_capture_set_control(CapHan, &CapCon);
// Start continuous capture with callback
ret = alf_capture_start(CapHan, alf_callback);
...
// Stop continuous capture
ret = alf_capture_stop(CapHan);
```

The Program Examples chapter later in this manual lists and briefly explains the example programs that come with the distribution.

12. Board Configuration Function

The TriMedia program can obtain configuration data about the FastSeries processor board. First, it sets up a blank **alf_config_t** structure. This structure has the definition:

```
typedef struct {
        int alf_config_board_type;          // ALF_CONFIG_FI/FD/FF/F4
        char alf_config_board revision[9];
                                    // 1.2.3.4 (null terminated)
        int alf_config_serial_number[10];
                                    // EEPROM s/n (null terminated)
        int alf_config_processor_number;    // Processor number
        int alf_config_processor_type;
                                    // ALF_CONFIG_TM1000/1100/1300
        int alf_config_processor_stepping;
                                    // TriMedia processor stepping
        int alf_config_clock_speed;   // integer Mhz
        int alf_config_SDRAM_size;          // integer Mbytes
        int alf_config_PCI_bus;             // PCI bus number
        int alf_config_PCI_devfunc;         // PCI device/function number
        int alf_config_PCI_IRQ;                     // PCI IRQ
        int alf_config_EEPROM_processor_number;     // Processor number
                                                    from EEPROM
        unsigned long alf_config_SDRAM_base;        // SDRAM base address
        unsigned long alf_config_MMIO_base;   // MMIO base address
        int alf_config_digital_in;    // Digital input present
        int alf_config_analog_in;           // Analog input present
        int alf_config_digital_out;   // Digital output present
        int alf_config_analog_out;    // Analog output present
} alf_config_t;
```

Then, the program passes a pointer to the blank structure in a call to:

int **alf_config_get**(alf_config_t **alf_config_p*)

The Capture library fills in the structure with the Board Configuration data, then returns.

13. Timing and Delay Functions

The Capture library includes functions to allow the TriMedia application to perform simple timing and delay functions.

a) Timing Functions

To perform a timing operation, the application program allocates a variable of type **alf_timing_t,** then initializes the timer variable with a call to:

void **alf_timing_start** (alf_timing_t **ts*)

When this function returns, the timer variable *ts* have received an initial (internal) value from the system clock. Then, after the section of code to be timed, the application can obtain the calculated elapsed time in seconds by calling:

float **alf_timing_elapsed** (alf_timing_t **ts*)

The value returned is the number of seconds since the call to **alf_timing_start** as a floating-point value. If instead the timing is desired in milliseconds, the application calls:

unsigned long **alf_timing_msec_elapsed** (alf_timing_t **ts*)

The value returned is the number of milliseconds since the call to **alf_timing_start** as an unsigned long value.

Multiple calls may be made to the elapsed timing functions with a given timing variable. The value returned is always the elapsed time from the call to **alf_timing_start** with that timing variable.

b) Delay Functions

The program can introduce a simple delay. To specify a delay in milliseconds, the TriMedia application calls:

void **alf_timing_msec_delay** (int *nmsec*)

To delay for a period of microseconds instead, the application calls:

void **alf_timing_usec_delay** (int *nusec*)

> **Note**. Applications using pSOS+ should not use these delay functions, since they perform a busy-wait. pSOS+ based applications should use the pSOS+ sleep/wakeup or event functions to perform delays.

B. Digital Output Library Functions

The Digital Output hardware configuration is specified in a Digital output profile read by all TriMedia. TriMedia 0 then uses the digital output profile to program the crossbar and set up other devices. Attributes of the digital output profile may be read and queried by any TriMedia.

> **Note:** Just as with the Capture profile, the Digital Output profile must be read and loaded into the programmable devices by TriMedia 0 before any other TriMedia can begin digital output operations.

Digital output via the output drivers can be enabled and disabled in the hardware, under the control of the Digital output profile. The program on TriMedia 0 can independently enable and disable the drivers.

The application specifies the output mode and number of output buffers for the library to allocate. The output buffer addresses are returned to the application. Digital Output operates in one of two modes: Continuous mode, where the output buffers are continuously directed to the output, and Single mode, where a single buffer is output.

The Digital Output buffer format is identified in the configuration profile as RAW or YUV. When RAW is specified, a single component is allocated. When YUV is specified, three components are allocated, one each for the Y, U and V components. Since YUV output is always 4:2:2 layout, the U and V components are half the length of the Y component.

When Digital Output is started, the application provides a callback function. This function is invoked at the completion of output of one buffer, and at the initiation of output of the next buffer. The callback is passed a structure with information identifying the buffer most recently completed or the buffer being started.

When (continuous) digital output is no longer needed, the application can stop the output.

1. Include File

The header file **alfast_tm.h** contains all needed definitions for the ALFAST Digital Output Library for the TriMedia program. The path to this file is **%ALFAST%\include**, where **%ALFAST%** represents the install directory for the runtime libraries.

2. Initialize Digital Output Library

Each TriMedia processor that is to use the digital output channel accesses the Digital Output library. The first step is to allocate a variable of type **handle**, for example:

```
#include <alfast_tm.h>
...
handle DigOutHan;
```

The TM program next initializes the Digital Output Library and obtains a handle for use in later Digital Output library functions with a call to:

handle **alf_digout_attach**(int *device_instance*)

In all TM programs, the *device_instance* should be 0. For example:

```
DigOutHan = alf_digout_attach(0);
```

3. Load and Access Digital Output Profile

The details about the data to be output are provided to the TM program via a Digital Output Configuration profile in PC memory. The Digital Output library uses the profile information to set up all the programmable devices on the FastSeries board.

To read the profile from PC memory into TriMedia memory, the TM program calls

int **alf_digout_load_profile**(handle *h*, char **profile_filename*)

Argument *h* is the handle returned by **alf_capture_attach**. Argument *profile_filename* is the name of the profile (enclosed in quotes). The Digital Output library searches for the profile first in the current directory, then in **%ALFAST%\lib\digout**, where **%ALFAST%** is the install directory for the runtime software. For example:

```
ret = alf_digout_load_profile(DigOutHan, “example2.dop”);
```

Digital Output configuration profiles have the following attributes available for reading by the application:

DDS_OUTPUT	1 = output enabled, 0 = disabled
DDS_FREQ	Frequency (MHz) of internally-generated clock.
DIGOUT_MODE	RAW or YUV
NROWS	Number of rows in the output (RAW = 1).
NCOLS	Number of columns in output (RAW = buffersize)
INTPRI	Interrupt priority
YUV_MODE	TriMedia-specific YUV mode
YUV_VIDEO_STD	NTSC, PAL, or NONE
IMAGE_VERT_OFFSET	Vertical offset in pixels of buffer within window
IMAGE_HORZ_OFFSET	Horizontal offset in pixels of buffer within window

Note: TriMedia processor 0 on the board is the primary processor; only this processor can program devices on the board such as crosspoint switches, FPGAs, CPLDs, MAXDACs, and UARTs. If the processor executing the **alf_digout_load_profile** function is processor 0, then, in addition to reading the profile attributes into TriMedia memory, *the Digital Output library executes internal commands to program all devices on the FastSeries board as specified by the profile attributes.*

4. Query and Set Digital Output Attributes

The TriMedia program can query the value of any attribute from the digital output profile loaded with **alf_digout_load_profile**. The program first sets up a blank structure of type **alf_attribute_t**. The definition for this structure type is:

```
typedef struct {
```

```
        union {
               unsigned long attr_lvalue;
               float attr_fvalue;
               char *attr_svalue;
               } attr_value;
        int type;
        } alf_attribute_t;
```

Then the program calls the following routine, passing it the capture handle from the **alf_digout_attach** call, the name of the attribute to be returned, and the address of the alf_attribute_t variable.

int **alf_digout_query_attribute** (handle *h*, char **name*, alf_attribute_t **p*)

When the function returns, the structure has been filled in with the value, which can be an integer, a float, or a string. The type returned can be read from the **type** element of the **alf_attribute_t** structure. The following predefined tokens may be used to match the **type** element: **ALF_ATTRIBUTE_LVALUE**, **ALF_ATTRIBUTE_FVALUE**, **ALF_ATTRIBUTE_SVALUE**.

When the type of value is known, the program can use the following defined tokens to access the union element with the desired type:

```
#define lvalue attr_value.attr_lvalue
#define fvalue attr_value.attr_fvalue
#define svalue attr_value.attr_svalue
```

For example:

```
alf_attribute_t p_attrib;
char profile_source[20];

ret = alf_digout_query_attribute(DigOutHan, "DIGOUT_MODE",
                                                     &p_attrib);
if (&p_attrib->type == ALF_ATTRIBUTE_SVALUE) {
       memcpy (profile_source, p_attrib.svalue;
       }
```

It may be desired to set the value of an attribute from within the TriMedia program. The program first sets up an **alf_attribute_t** structure with a value of the correct type assigned to the appropriate union element, and the **type** member set to a token for the value type. The program then calls the following function, passing it the digital output library handle, the name of the attribute to be set, and the address of the attribute structure containing the value to be set.

int **alf_digout_set_attribute** (handle *h*, char **name*, alf_attribute_t **p*)

For example:

```
alf_attribute_t p_attrib;
&p_attrib->lvalue = 1024;
p_attrib.type = ALF_ATTRIBUTE_LVALUE;

ret = alf_digout_set_attribute(CapHan, "NCOLS",
                                              &p_attrib);
```

5. Set Up Digital Output Parameters

The TriMedia application program sets up parameters to control the capture operation in an **alf_digout_control_t** structure. The **alf_digout_control_t** structure has the following elements:

```
typedef struct {
       int alf_digout_nbuf;   // Number of buffers
```

```
        int alf_digout_mode;   // Mode = ALF_DIGOUT_CONTINUOUS,
                                       // ALF_DIGOUT_SINGLE,
} alf_digout_control_t;
```

Set structure element **alf_digout_mode** to specify continuous or single-buffer capture.

The element **alf_capture_nbuf** sets the number of buffers to allocate. When the **DIGOUT_MODE** attribute in the Digital Output profile is YUV, each buffer contains the Y, U, and V components. Since YUV output is always 4:2:2, the U and V components are half the length of the Y component. When **DIGOUT_MODE** is RAW, each buffer contains just a single component.

When the control structure is set up, the application calls:

int **alf_digout_set_control**(handle *h*, alf_digout_control_t **p*)

Now, subsequent digital output operations will use this set of parameters. The buffers specified in the control structure are allocated in TriMedia memory. Pointers to the buffers are passed automatically to the callback function, described in the next section. Pointers may also be retrieved with the status-monitoring function described later.

6. Declare Callback Function

A digital output operation can be monitored with a callback function or by polling the capture status. The polling method is described later in this chapter. The application program declares a callback function like the following:

```
void *alf_dout_callback (int cause, alf_digout_buf_t *buf) {

// Code here to check the cause, handle the data.

}
```

The callback function is called upon the completion of output from one buffer and again at the initiation of output from the next buffer, or when an error occurs. The *cause* and *buf* parameters are passed to the callback by the library. The following predefined values for *cause* are used:

ALF_DIGOUT_START indicates a buffer is ready to transmit.

ALF_DIGOUT_COMPLETE indicates a buffer has been completely transmitted.

ALF_DIGOUT_ERROR indicated that an error occurred during capture.

When the cause is ALF_DIGOUT_COMPLETE, the argument ***buf*** points to a structure of type **alf_digout_buf_t** containing information identifying the index of the buffer that most recently completed, and the actual **alf_buf_t** elements that were most recently completed. When cause is ALF_DIGOUT_START, the argument ***buf*** points to a structure of type **alf_digout_buf_t** containing information identifying the buffer that is now being started.

The **alf_digout_buf_t** structure has the elements:

int **alf_digout_buf_idx**	The index of the buffer, ranging from 0 to the number of buffers – 1
alf_buf_t **alf_digout_bufs**[4]	This is an array of **alf_buf_t** structures, each containing buffer information (see below.)

The **alf_buf_t** structure is the same one used for capture buffers. Each **alf_buf_t** structure has the following members:

void ***alf_buf_data**	Pointer to the cache-aligned data buffer
int **alf_buf_stride**	Row to row stride
int **alf_buf_nrows**	Number of rows in the data buffer
int **alf_buf_ncols**	Number of columns in the data buffer
int **alf_buf_size**	Size in bytes of the data buffer

For “raw” data transfers (Monochrome and RGB Digital output), only one **alf_buf_t** structure is used (accessed as **alf_digout_bufs[0]**). Predefined macro **alf_digout_buf** allows a simpler form of reference. The macro definition is:

```
#define alf_digout_buf alf_digout_bufs[0]
```

Now, if *buf* is a pointer to an **alf_digout_buf_t**, the callback routine can access the first structure in the **alf_digout_bufs** array of **alf_buf_t** structures with:

```
buf->alf_digout_buf.structure_member
```

Which is equivalent to:

```
buf->alf_digout_bufs[0].structure_member
```

YUV data (NTSC, PAL, SECAM, and S-video) uses three buffer structures. Use predefined tokens for the **alf_digout_bufs** indexes: **ALF_DIGOUT_BUFY** for Y, **ALF_DIGOUT_BUFU** for U, and **ALF_DIGOUT_BUFV** for V. Predefined macros **alf_digout_bufY**, **alf_ digout_bufU**, and **alf_ digout_bufV** simplify references to the array of buffer structures:

```
#define alf_digout_bufY  alf_capture_bufs[ALF_CAPTURE_BUFY]
#define alf_digout_bufU  alf_capture_bufs[ALF_CAPTURE_BUFU]
#define alf_digout_bufV  alf_capture_bufs[ALF_CAPTURE_BUFV]
```

Now, the callback accesses these arrays of buffer structures with:

```
buf->alf_digout_bufU.structure_member
```

This construct is equivalent to:

```
buf->alf_digout_bufs[ALF_DIGOUT_BUFU].structure_member
```

Continuing the example, the callback routine (and the main program as well; see note below) can access the data in the buffer with a reference such as:

```
buf->alf_digout_bufY.alf_buf_data     // Buffer of Y (luma) data
```

Note: The argument ***buf*** to the callback function points to static data, which remains a valid reference outside of the callback context.

7. Digital Output Driver Enable and Disable

The digital output drivers can be enabled and disabled in the hardware, under the control of the **DDS_OUTPUT** attribute in the Digital Output profile. The drivers can be initially enabled (**DDS_OUTPUT** = 1) or disabled (**DDS_OUTPUT** = 0). The program on TriMedia 0 can enable the drivers with a call to:

int **alf_digout_enable** (handle *h)*

If the drivers were disabled, they become enabled. To disable the drivers again, TM0 calls:

int **alf_digout_disable** (handle *h)*

8. Start the Digital Output Operation

To start data output from the buffers, the application program calls

int **alf_digout_start**(handle *h*,

```
void (*alf_dout_callback (int cause, alf_digout_buf_t *buf))
```

Digital output begins continuously or in single-buffer mode. In either case, this function returns immediately (display is asynchronous). Completion of the transfer can be monitored with a callback function as described in the previous section or by polling for status as described next (for polling, the **alf_digout_start** call contains a NULL callback routine).

9. Retrieve Status and Buffer Information

To obtain pointers to the output buffers or to monitor digital output status by polling, the TriMedia program sets up a blank **alf_digout_status_t** structure. The definition for this structure type is:

```
typedef struct {
        int alf_digout_running;
                // true if digital output is running
        int alf_digout_bufno;
                        // current digital output buffer number
        int alf_digout_last_bufno;
                                // last processed buffer (now free)
        alf_digout_buf_t *alf_digout_buflist;
                                // ptr to array of buffer ptrs
} alf_digout_status_t;
```

Then, the application program calls

int **alf_digout_status**(handle *h*, alf_digout_status t **alf_digout_p*)

When this function returns, the structure pointed to by *alf_digout_p* contains status on the digital output operation. The application can reference any of the elements of the structure. The buffer pointers are returned in an structure pointed to by element **alf_digout_buflist** (see Declare Callback Function for details).

10. Stop Continuous Digital Output Operation

Single-buffer displays terminate after one buffer has been output. To terminate a continuous digital output operation, the TriMedia application calls

int **alf_digout_stop**(handle *h)*

The continuous output operation is halted.

11. Digital Output Example

Here is an example of some of the Digital Output library calls in a code fragment.

```
// (Outside main Routine)
...
// Set up callback function
void *alf_dout_callback(int cause, alf_digout_buf_t *buf) {
        // Code here to determine the cause of the callback
```

```
        // and process the buffer retrieved via buf.
}
...
// (Inside main routine)
...

// Attach to digital output library
handle DigOutHan = alf_digout_attach((int)0);

// Load digital output configuration profile
int ret = alf_digout_load_profile(DigOutHan, "example2.dop");

// Create control structure and set it up
alf_digout_control_t DoutCon;
DigOutCon.alf_digout_nbuf = 2;
DigOutCon.alf_digout_mode = (ALF_DIGOUT_CONTINUOUS);
ret = alf_digout_set_control(DigOutHan, &DigOutCon);
// Start continuous output with callback
ret = alf_digout_start(DigOutHan, alf_dout_callback);
...
// Stop continuous output
ret = alf_digout_stop(DigOutHan);
```

The Program Examples chapter later in this manual lists and briefly explains the example programs that come with the distribution.

C. S3 Streaming Library Functions

Video Output Streaming provides an API for configuring the S3 ViRGE/GX2 display chip to accept video data streams, either from the TM processors (which would use Digital Output), or directly from the Capture hardware using suitable FPGA/Crossbar configurations.

Video Output Streaming takes a data stream from the S3 Local Peripheral Bus (LPB) and stores it into buffers allocated in the Video Memory. This LPB input may be RGB or YUV format. Independently, the Secondary Stream processor (SS) in the S3 chip takes an arbitrary image stored in the Video Memory, performs color space conversion (if needed) and resizing, then merges the result with the Primary video stream that drives the display monitor. By directing the SS to take data from the buffers being filled by the LPB, the LPB input data may be displayed on the screen, without upsetting the data being managed by the Host in the primary display memory. With the capability of the SS, the Video Streaming window may be of any size and positioned anywhere on the display.

NOTE: The Fast Series boards currently have 4 MB of Video memory. Depending upon the selected display resolution, the amount of Video Memory available for Video Streaming may be limited. For example, a buffer 1024 by 768 of 24-bit RGB pixels requires 2.25 MB, leaving 1.75 MB for Video Streaming. **alf_vstream_set_window** may fail with ALF_ERROR_ALLOC if the available Video Memory is not sufficient.

The program on a TriMedia processor accesses the S3 Streaming output on the FastSeries board via calls to the ALFAST S3 Streaming Library.

1. Initialize S3 Streaming Library

Each TriMedia that is to use S3 Streaming Output accesses the Video Output Streaming library by allocating a variable of type **handle**, for example:

```
#include <alfast_tm.h>
...
handle StreamHan;
```

The TM program next initializes the Streaming Library and obtains a handle for use in later library functions with a call to:

handle **alf_vstream_attach**(int *device_instance*)

In all TM programs, the *device_instance* should be 0. For example:

```
StreamHan = alf_vstream_attach(0);
```

2. Set Up LPB Input

The TriMedia application program sets up parameters to control the Local Peripheral Bus operation in an **alf_vstream_lpb_t** structure:

```
typedef struct {
        int alf_lpb_nx;
                // Number of X pixels in input data stream
        int alf_lpb_ny;
                // Number of Y pixels in input data stream
        int alf_lpb_mode;
                // RGB or YUV input data stream
        int alf_lpb_offset_x; // Input X-offset
        int alf_lpb_offset_y; // Input Y-offset
        int alf_lpb_mode;
                // ALF_LPB_YCBCR,
                // ALF_LPB_YUV16,
                // ALF_LPB_KRGB16,
                // ALF_LPB_YUV,
                // ALF_LPB_RGB16,
                // ALF_LPB_RGB24,
                // ALF_LPB_XRGB32
        unsigned long alf_lpb_address;
                // Force LPB address
} alf_vstream_lpb_t;
```

The program then calls the following function, passing it the handle and a pointer to the **alf_vstream_lpb_t** structure containing the parameters to be set.

int **alf_vstream_set_lpb** (handle *h*, alf_vstream_lpb_t **pc*)

This routine allocates the addresses in S3 Video Memory for the LPB input buffer from space immediately following that used by the S3 primary display.
If (*pc*->**alf_lpb_address**) is non-zero, it is taken as the offset to apply to the start of S3 Video Memory, overriding the default address.

3. Set Up Streaming Output Window Parameters

The TriMedia application program sets up parameters to control the Video Streaming operation in an **alf_vstream_window_t** structure. The structure has the following elements:

```
typedef struct {
        int alf_ss_in_x0;
// Upper left X coordinate of input buffer to be displayed
int alf_ss_in_y0;
// Upper left Y coordinate of input buffer to be displayed
int alf_ss_in_nx; // Width of input buffer to be displayed
int alf_ss_in_ny; // Height of input buffer to be displayed
int alf_ss_out_x0;
// Upper left X coordinate of screen location of SS window
int alf_ss_out_y0;
// Upper left Y coordinate of screen location of SS window
int alf_ss_out_nx; // Width of SS window
int alf_ss_out_ny; // Height of SS window
int alf_lpb_nx; // Number of X pixels in input data stream
int alf_ss_keying;      // ALF_SS_KEY_WINDOW,
                        // ALF_SS_KEY_COLOR,
                        // ALF_SS_KEY_CHROMA
```

```
unsigned long alf_ss_keydata[4];
                        // Keying dependent data (TBD)
} alf_vstream_control_t;
```

NOTE: **alf_ss_keying** and **ss_keydata** are ignored at ALFAST Release 1.4.1. **ALF_SS_KEY_WINDOW** is the only mode supported.

When the control structure is set up, the application calls:

int **alf_vstream_set_window** (handle *h*, alf_vstream_window_t **p*)

Now, subsequent video streaming operations will use the window specified by this set of parameters; the input region of interest is scaled to the SS window. The **alf_vstream_set_window** function may be called any number of times after video streaming has started (see next section), to resize or relocate the region of interest

4. Start the Video Streaming Operation

Next, the application program calls

int **alf_vstream_start**(handle *h*)

Video streaming begins continuously. This function returns immediately (streaming is asynchronous). Completion of the operation can be monitored by polling for status as described next.

5. Retrieve Status Information

To monitor Video Streaming status, the TriMedia program sets up a blank **alf_vstream_status_t** structure. The definition for this structure type is:

```
typedef struct {
        int alf_vstream_running;
                        // true if video streaming is running
        alf_vstream_lpb_t alf_vstream_lpb;
                // LPB information
        alf_vstream_window_t alf_vstream_window;
                // Window information
} alf_vstream_status_t;
```

Then, the application program calls

int **alf_vstream_status**(handle *h*, alf_vstream_status t **alf_vstream_p*)

When this function returns, the structure pointed to by *alf_vstream_p* contains status on the video streaming operation. The application can reference any of the elements of the structure.

NOTE: Function **alf_vstream_status** is not supported at ALFAST Release 1.4.1.

6. Stop Continuous Video Streaming Operation

To halt a continuous video streaming operation, the TriMedia application calls

int **alf_vstream_stop**(handle *h)*

The continuous display operation is terminated.

7. Video Streaming Example

Here is an example of some of the Video Streaming library calls in a code fragment.

```
// (Inside main routine)
...
// Attach to Video Streaming library
handle VstreamHan = alf_vstream_attach((int)0);

// Create LPB structure and set it up
alf_vstream_lpb_t Vstreamlpb;

Vstreamlpb.alf_lpb_nx = 1024;
Vstreamlpb.alf_lpb_ny = 768;
Vstreamlpb.alf_lpb_mode = ALF_LPB_YUV;

ret = alf_vstream_set_lpb(VstreamHan, &Vstreamlpb);

// Create window structure and set it up
alf_vstream_window_t VstreamWin;

VstreamWin.alf_ss_in_x0 = 0;
VstreamWin.alf_ss_in_y0 = 0;
VstreamWin.alf_ss_in_nx = 512;
VstreamWin.alf_ss_in_ny = 384;
VstreamWin.alf_ss_out_x0 = 0;
VstreamWin.alf_ss_out_y0 = 0;
VstreamWin.alf_ss_out_nx = 512;
VstreamWin.alf_ss_out_ny = 384;
VstreamWin.alf_ss_keying = ALF_SS_KEY_WINDOW;

ret = alf_vstream_set_window(VstreamHan, &VstreamWin);

// Start continuous display
ret = alf_vstream_start(VstreamHan);
...
// Stop continuous output
ret = alf_vstream_stop(VstreamHan);
```

The Program Examples chapter later in this manual lists and briefly explains the example programs that come with the distribution.

D. TV Output Library Functions

FastSeries boards equipped with the S3 display have the capability of generating television compatible video output streams. The ALFAST API for TV Output defined to configure and enable TV output supports the following output formats:

- 525 line, interlaced, 60 Hz, NTSC and NTSC-Japan composite and S-Video component (640 x 432 pixels)
- 625 line, interlaced, 50 Hz, PAL composite and S-Video, (640 x 525)

The TV Output video is generated by the S3. It operates independently from the S3's standard VGA output, which is controlled with the Analog Output library API. The TV Output and VGA outputs share the same 4-MB video memory, but may be configured to use overlapping or non-overlapping regions of video memory.

Setting the TV Output buffer address (the offset into video memory for TV Output image data) to 0 creates an overlapped display corresponding to the upper left hand corner of the VGA display. When the TV Output uses an overlapping video memory region, the TV Output configuration is set up so that the bits per pixel and row stride values match those of the VGA output.

When non-overlapped video regions are desired, the user may configure the TV Output to have a different bits per pixel value from the VGA, and choose a stride value that is efficient for the desired mode.

The TV Output image resolution is fixed by the Output format selected. NTSC and NTSC-Japan are set to 640 x 432 pixels, PAL is set to 640 x 525 pixels.

When configured for 8-bit pixels the TriMedia program initializes the 256 location TV Output palette. This palette memory is separate from the palette used for S3 VGA output. The functions **alf_tvout_wrpalette** and **alf_tvout_rdpalette** may be used to initialize the TV Output palette.

It is often desirable to synchronize image data updates with the video refresh. The function **alf_tvout_wait_eof** may be used to wait until the beginning of vertical retrace of the TV Output.

1. Initialize TV Output Library

Each TriMedia that is to use S3 TV Output accesses the TV Output library by allocating a variable of type **handle**, for example:

```
#include <alfast_tm.h>
...
handle TVHan;
```

The TM program next initializes the Streaming Library and obtains a handle for use in later library functions with a call to:

handle **alf_tvout_attach**(int *device_instance*)

The attach function is called with a single parameter, *device_instance*, which will always be 0, since there is only one S3 device on the FastSeries boards. If successful, a ***handle*** will be returned to be used with the remainder of the routines.

In all TM programs, the *device_instance* should be 0. For example:

```
TVHan = alf_tvout_attach(0);
```

2. Allocate New Control Context

The TriMedia program next allocates a TV Output control context by setting up a pointer to an **alf_tvout_t** structure and making a call to:

int **alf_tvout_new** (handle *h*, alf_tvout_t ***ptvout*)

Note that the *address* of the pointer is passed. This function allocates a control context of type **alf_tvout_t** and initializes it with default values. The default values (all of type `int`) are:

alf_tvout_mode	NTSC
alf_tvout_output	composite video
alf_tvout_flicker	flicker filter enabled, value 2
alf_tvout_bpp	same bits per pixel as the VGA display
alf_tvout_buffer_address	address of start of VGA display
alf_tvout_stride	same stride as VGA display

When the **alf_tvout_t** structure is no longer needed, it may be deallocated with the standard library function **free**:

```
free (h);
```

3. Set Up Control Variables

If the default values are correct, the program simply passes the **alf_tvout_t** structure unchanged to the function **alf_tvout_control**. Otherwise the program uses assignment statements to change those elements that require new values.

int **alf_tvout_control** (handle *h*, tvout_t **ptvout*)

The following table summarizes the **alf_tvout_t** control structure variables. Details may be found in the TV Output Library Reference section describing **alf_tvout_control**.

alf_tvout_mode	ALF_TVOUT_NTSC	selects 525 line NTSC

	ALF_TVOUT_NTSCJ	selects 525 line NTSC-Japan
	ALF_TVOUT_PAL	selects 625 line PAL
alf_tvout_output	ALF_TVOUT_COMPOSITE	selects composite video output
	ALF_TVOUT_SVIDEO	selects separated video output
alf_tvout_flicker	0-6	enable flicker filter, extent of filter, higher value is more filtering
	ALF_TVOUT_DISABLE	disable flicker filter
alf_tvout_bpp	<integer value>	bits per pixel, 8, 15, 16, 24, or 32
	ALF_TVOUT_USE_VGA	use the bpp setting from the VGA output
alf_tvout_buffer_address	ALF_TVOUT_USE_VGA	use the starting buffer address from the VGA output
	ALF_TVOUT_JUST_AFTER_VGA	specify a starting buffer address that immediately follows the end of the VGA output
	<offset value>	explicitly specify a starting buffer address (value is an offset from the beginning of video memory)
alf_tvout_stride	ALF_TVOUT_USE_VGA	use the row stride from the VGA output
	ALF_TVOUT_USE_DEFAULT	use a row stride that results in contiguous memory use depending on the TV Output Mode (640 pixels for NTSC, NTSCJ and PAL)
	<stride value>	explicitly specify a stride in bytes

4. Turn on TV Output

The TriMedia program starts TV Output by calling:

int **alf_tvout_start** (handle *h*)

5. Turn Off TV Output

The TriMedia program stops TV Output by calling:

int **alf_tvout_stop** (handle *h*)

NOTE: If the S3 VGA output format is changed (using Analog Output library function **vda_ioctl (VDA_SET_RESOLUTION)**), the TV Output should be reinitialized with calls to **alf_tvout_stop, alf_tvout_control** and **alf_tvout_start.**

6. Query TV Output Parameters

Using the **alf_tvout_params** function, the values of various parameters may be determined. The syntax is:

int **alf_tvout_params** (handle *h*, tvout_params_t **params*)

The elements of the **tvout_params_t** structure are all of type `int`:

alf_tvout_nx	number of pixels per display row
alf_tvout_ny	number of display rows
alf_tvout_hstride	number of bytes per pixel
alf_tvout_vstride	number of bytes between the start of successive rows
alf_tvout_bpp	bits per pixel (8, 15, 16, 24, 32)

alf_tvout_buffer_address	address of start of TVOUT image display (this is an actual address, not an offset)

7. Reading and Setting the TV Output Color Palette

For 8-bit color modes, the pixel value is an index into the TV Output color palette, each of the 256 colors in the TV Output palette is a **vda_rgb_t** structure:

```
typedef struct {
        unsigned char vda_red;
        unsigned char vda_green;
        unsigned char vda_blue;
} vda_rgb_t;
```

The TriMedia program allocates space for an array of **vda_rgb_t** structures. The size of the array, *n*, is the maximum number of colors to be read or written at one time, not necessarily 256.

To read n colors from the TV Output palette, the TriMedia program calls:

int **alf_tvout_rdpalette** (handle *h*, int *idx*, vda_rgb_t **pal*, int *n*)

To write or set n colors into the TV Output palette, the program calls:

int **alf_tvout_wrpalette** (handle *h*, int *idx*, vda_rgb_t **pal*, int *n*)

NOTE: The TV Output color palette is completely separate from the palette used by the S3 VGA Analog Output library.

8. Synchronizing TV Output with Screen Redraw

To have the TV Output begin concurrently with the start of screen refresh, the program inserts a call to the following function just before **alf_tvout_start**:

int **alf_tvout_wait_eof** (handle *h*)

This function waits for the end of the TV Output display end to be reached just prior to beginning vertical retrace. When **alf_tvout_wait_eof** returns, **alf_tvout_start** executes, producing synchronized screen data updates.

9. TV Output Example

```
int rval;
handle h;
alf_tvout_control_t *ptv;
alf_tvout_params_t params;

h = alf_tvout_attach (0);
if (ALF_ISERROR (h))
{
        printf ("ERROR: alf_tvout_attach failed\n");
        return -1;
}
rval = alf_tvout_new (h, &ptv);
if (ALF_IS_ERROR (rval))
{
        printf ("ERROR: alf_tvout_new failed\n");
        return -1;
}
ptv->alf_tvout_output = ALF_TVOUT_SVIDEO;
rval = alf_tvout_control (h, ptv);
if (ALF_IS_ERROR (rval))
{
        printf ("ERROR: alf_tvout_control failed\n");
```

```
        return -1;
}

rval = alf_tvout_start (h);
if (ALF_IS_ERROR (rval))
{
        printf ("ERROR: alf_tvout_start failed\n");
        return -1;
}

rval = alf_tvout_params (h, &params);
if (ALF_IS_ERROR (rval))
{
        printf ("ERROR: alf_tvout_params failed\n");
        return -1;
}
        printf ("nx %d ny %d\n", params.alf_tvout_nx, params.alf_tvout_ny);

printf ("bpp %d hstride %d vstride %d\n", params.alf_tvout_bpp,
params.alf_tvout_hstride, params.alf_tvout_vstride);
```

E. Analog Output Library Functions

The Analog Output Library allows TriMedia processors to access the S3 primary memory in two modes. In Host-controlled mode, the Host Operating System uses the S3 for the system display. In TriMedia-controlled mode, another device is the system display, and the TriMedia program can clear the display and set the resolution of the S3 analog output. The API for all Analog Output library functions is based on Alacron's VDA daughter card library.

1. S3 ViRGE GX2 Driver Controls for Windows NT

In Host-controlled mode, the Windows NT operating system exercises the S3 driver for the ViRGE GX2 analog output device. Depending on your FastImage or FastFrame configuration, you may need to set up the display via the S3 ViRGE GX2 driver.

From the Start Menu in Windows NT, select **Settings->ControlPanel->Display**. The Display Properties window appears. Click on the **S3** tab. The S3 driver window appears, with three checkboxes and an Advanced Settings button.

Select the **CRT** checkbox to have output go to the system console. A checkmark appears when the CRT output is enabled.

Select the **TV** checkbox to have output go to the NTSC/PAL/S-Video output connector. A checkmark appears when the TV output is enabled.

When both **CRT** and **TV** output are enabled, select the **Optimal Timing** checkbox to have the CRT console output and the TV monitor output each run at its own rate. A checkmark appears when Optimal Timing is enabled. Optimal Timing lets the main CRT screen display its entire area while the TV monitor scrolls and pans to display all regions. Turning Optimal Timing off runs both displays at the same rate, and now they both scroll/pan to display the screen.

When the hardware and driver are configured for TV output (NTSC/PAL/S-video), you can make further adjustments by clicking on the **Advanced Settings** button. The Advanced Settings Properties window appears.

- From the TV Type panel, select one of the three buttons **NTSC**, **PAL**, or **NTSC Japan** as appropriate.
- From the TV Output Signal panel, select the **Composite** or **S-Video** button as appropriate.

- In the Flicker Filter panel, select the **On** button to have the GX2 average adjacent lines of pixels to eliminate flicker in one-line objects. The Flicker Filter is required when the picture is not pre-filtered video data. Use the **Minimum...Maximum** slider to adjust the filter operation; some testing may be required to find the optimal setting for a given data set.

When all S3 driver configuration selections have been made, click Apply at the lower right of the window you are on. The system asks you if the setting is to become permanent; by default, a change is temporary so that a disastrous setting can be recovered.

2. S3 ViRGE GX2 Driver Controls for Linux

In Host-controlled mode, the Linux operating system exercises the S3 driver for the ViRGE GX2 analog output device. Under Linux, you must configure X Windows for the display adapter you are using. The X configuration can specify that the S3 is to be used as the X display device. Under Redhat 6.1, the **Xconfigurator** program performs this function.

3. S3 Bridge Control

For TriMedia-controlled mode, it is necessary to isolate the local PCI bus on the FastSeries board from the PC primary PCI bus. This is required so that the TM processors may execute PCI I/O cycles to the S3 device and not have the cycles reflected on the primary PCI bus, which would access other VGA devices in the system.

The ALFAST Runtime software includes a native device driver (**pciprobe.sys**), and a service (**S3BridgeControl**), which are installed and run at system boot time. They identify any Alacron S3 devices, and reprogram the FastSeries board PCI-to-PCI bridge device.

A side effect of the reprogramming of the bridge is that PCI I/O cycles are not passed from the primary PCI bus to the FastSeries PCI bus. This can be a problem only if a PMC module is installed with a device that requires I/O access. Memory access to the FastSeries PCI bus operates normally.

When the **S3BridgeControl** service is run, a log file is created in the directory given by the environment variable TMPDIR, with file name **S3BridgeControl.log**.

Under Windows NT, you can specify that one or more FastSeries bridges should not be reprogrammed by editing registry keys that contain an “Enable” variable. Setting Enable to 0 keeps the corresponding S3 device from being reprogrammed. The registry keys are:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\S3BridgeControl\Device0
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\S3BridgeControl\Device1
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\S3BridgeControl\Device2
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\S3BridgeControl\Device3
```

Use the NT-supplied registry editor program (**regedit**) to alter the Enable values, or to add S3 devices beyond Device3.

Under Linux, **S3bridgecontrol** always reprograms the bridges, and you cannot selectively disable the reprogramming.

4. Include File

To access the routines and structures in the Analog Output library, the application includes the header file **vda.h**, which is located in the install directory **%ALFAST%\include**.

5. Open S3 Video Memory

Any TriMedia processor that wants to use the S3 VGA display opens the device with a call to

int **vda_open** (int *device*, int *initialize*)

The *device* argument should always be 0 (zero). The *initialize* argument is ignored.

a) VDA Initialization File

The Analog Input VDA library supports access to the Fast Series S3 device, and in addition provides a mechanism for accessing other PCI memory mapped VGA display adapters. **vda_open** accesses an ini-file **%ALFAST%\lib\vda.ini** to obtain information on accessing both kinds of devices. See the chapter VDA Initialization File Format later in this manual for a specification.

6. Close S3 Video Memory

When the processor no longer needs access to the S3 display, it calls

int **vda_close** (int *dev*)

The *dev* argument should be 0 (zero) for all processors.

7. Library Version

To obtain the current version of the Analog Output library, the program can reference the global variable:

extern char **vda_version**[]

8. Handling Errors

Error messages from the Analog Output library resemble the messages returned by the VDA library. All (non-void) functions return zero upon successful completion or -1 upon failure. On failure, an error code may be retrieved using:

int **vda_errno** (int *dev*)

The application may enable the automatic display of any detected errors using:

int **vda_errorprt** (int *dev*)

Each function listed in the reference section provides information on each of the possible errors detected by each routine.

Potential errors include the following.

Definition	Explanation
VDA_NO_ERROR	No outstanding error exists
VDA_E_NODEV	Device specifier indicates a device that does not exist
VDA_E_NOTOPEN	Device specifier is not currently opened
VDA_E_OPENED	Device specifier is already opened
VDA_E_DEV	Invalid device specifier provided
VDA_E_CMD	Invalid IO command specified
VDA_E_CANT	The requested command is invalid for this device
VDA_E_BADPARAM	A parameter is invalid for this command
VDA_E_MODE	Invalid request for current configuration

VDA_E_FAILURE	Command failed

9. Getting Display Parameters

Once connected to the S3 display, the application may query the device configuration using the **VDA_GET_PARAMS** command to **vda_ioctl**.

int **vda_ioctl** (int *dev*, **VDA_GET_PARAMS**, vda_params_t **pp*)

First, the program declares a blank parameter structure **vda_params_t**, then passes a structure pointer to the function. For example:

```
vda_params_t p_params;
vda_ioctl(0, VDA_GET_PARAMS, &p_params)
```

When the function returns, the application can read the **vda_params_t** elements:

```
typedef struct {
        int device;    // VDA device for this structure (always 0)
        int xres;      // Number of pixels per horizontal line
        int yres;      // Number of lines per display
        int pixel_size;        // Number of bits per pixel (8, 15, 16, or
                               24)
        int hstride;   // Number of bytes per pixel
        int vstride;   // Byte offset from one line to the next
        unsigned long buffer_address;
                       // Address of the display buffer.  This address
                       // may be accessed by the TriMedia processor.
} vda_params_t;
```

For example, after the sequence of calls above, the reference **p_params.hstride** is the pixel size of the display in bytes.

10. TriMedia-Controlled Mode Functions

To query whether the display is Host-controlled or TriMedia-controlled, the TriMedia program calls:

int **vda_host_controlled** (int *dev*)

This function returns TRUE when the display is Host-controlled, FALSE when the display can be controlled by the TriMedia.

a) Setting VDA Resolution

When the S3 device is not Host controlled, and it is on the TM's local bus, the Analog Output library lets the TriMedia set the display resolution:

int **vda_ioctl** (dev, **VDA_SET_RESOLUTION**, char **resolution_string*)

This call sets the S3 device for display resolution given by a *resolution_string* of the form:

"xres**x**yres**x**bpp_framerate"

For example, the string "800x600x16_72" sets the display to 800 x 600 pixels, 16 bit pixels (65526 colors), with a frame rate of 72 Hz.

The function call can specify an arbitrary resolution. It is possible to configure the S3 device for nonstandard display formats or frame rates. For example, "1024x1024x8_85" would set up a 1k x 1k, 8 bit display at 85 Hz. Note that not all monitors are capable of displaying high frame or line rates.

VDA resolution settings are persistent. The S3 configuration that is set by one TriMedia program remains in effect even after the TM program terminates. Another TM program may start up, call **vda_open**, and **vda_ioctl (VDA_GET_PARAMS)** to determine the current xres, yres, and bpp.

b) Default VDA Resolution

If the S3 device has never been initialized, the first call to **vda_open** initializes the device to the following default resolution:

800 x 600, 16 bit pixels, 72 Hz frame rate

c) Clearing the VDA Display

The function **vda_cls** allows the program in TriMedia-controlled mode to clear the display screen of device ***dev*** to the color given by ***color.*** The function uses PCI DMA to speed this operation.

int **vda_cls** (int *dev*, int *which_buf*, int *color*)

Argument *dev* is the device opened with **vda_open**; *which_buf* must be **VDA_DISPLAY**; *color* is an integer color value such as that returned by function **vda_map_rgb**. The following example clears the screen to a light gray color:

```
vda_cls (dev, VDA_DISPLAY, vda_map_rgb (dev, 100, 100, 100));
```

11. Using a Graphics Primitive Buffer in SDRAM

In either Host-controlled or TriMedia-controlled mode, the Trimedia program can create graphics in the display using objects called *graphics primitives*. Graphics primitives can be created in a scratchpad buffer in TriMedia SDRAM, then copied from the buffer to the display on command. The SDRAM method is described in the next section. A second method is to obtain a pointer to the S3 display memory and write directly into it; see the later section Using a Graphics Primitive Buffer in Display Memory for details on direct access.

a) Create GRP Buffer in SDRAM

To work from a graphics primitive buffer in SDRAM, the TriMedia program declares a pointer to a **grp_buf_t** structure, then gets the structure allocated as the return value from the function:

grp_buf_t ***grp_alloc** (int *nrows*, int *ncols*, int *size*)

The *nrows* and *ncols* arguments specify the dimensions of the buffer in pixels, and *size* is the number of bytes per pixel. For example:

```
grp_buf_t *p_grp = grp_alloc (512, 512, 8);
```

To set up a scratchpad buffer that is the same size as the display, the TriMedia program can use **vda_params_t** parameter elements returned by **vda_ioctl (VDA_GET_PARAMS)**:

```
p_grp = grp_alloc (p_params.yres, p_params.xres,
                              p_params.hstride);
```

In addition to setting up the structure, the **grp_alloc** call allocates a buffer of the specified size in Trimedia SDRAM and returns the base address of the buffer in the **grp_buf_t** structure. For SDRAM buffers, the structure elements refer to the scratchpad buffer in SDRAM:

```
typedef struct grp_buf {
        unsigned long grp_base;
        // Base address of upper left pixel in buffer.
        int grp_xres;           // X resolution (pixels per row)
        int yres;       // Y resolution (pixels per frame)
        int grp_hstride;        // Bytes per pixel
        int grp_vstride;        // Bytes per row
} grp_buf_t;
```

The entire structure gets passed to the draw and copy routines. When the program no longer needs the buffer, it can free the memory with:

int **grp_free** (grp_buf_t **pb*)

b) Copy GRP Objects from SDRAM into S3 Display Memory

The details on drawing graphics and text objects in a Graphics Primitive buffer are covered later in this section. When a scratchpad buffer in SDRAM has been drawn, the Trimedia program copies it to display memory with a call to:

int **vda_sync** (int *dev*, grp_buf_t **pb*, **VDA_DISPLAY**)

The **vda_sync** function actually translates the **grp_buf_t** elements, then calls another function, **vda_write**, to perform the copy. The syntax of this more general routine is given in the Analog Output Library Reference later in this manual.

12. Using a Graphics Primitive Buffer in Display Memory

Instead of (or in addition to) using a scratchpad buffer in SDRAM for graphics operations, the TriMedia program in either Host- or TriMedia-controlled mode can arrange to draw directly into S3 Video display memory. The program declares a pointer to a blank **grp_buf_t** structure as before, then calls:

grp_buf_t ***vda_grp_vram** (int *dev*)

For direct drawing, the returned structure elements refer to the system display as follows:

```
typedef struct grp_buf {
        unsigned long grp_base;
        // Base address of upper left pixel in display.
        int grp_xres;           // X resolution (pixels per row)
        int yres;       // Y resolution (pixels per frame)
        int grp_hstride;        // Bytes per pixel
        int grp_vstride;        // Bytes per row
} grp_buf_t;
```

After the **vda_grp_vram** function returns, any drawing operations on the **grp_buf_t** object appear immediately on the display.

When the application no longer needs direct drawing ability, it can free the **grp_buf_t** object with a call to:

void **vda_grp_vram_free** (grp_buf_t **pb*)

13. Getting Integer Color Values

Several graphics routines take a color argument (type **int**). To determine the integer value representing a desired color combination expressed as a triplet of RGB intensity values from 0 to 255, the application calls:

int **vda_map_rgb** (int *dev*, int *red*, int *green*, int *blue*)

The *dev* argument is the VDA device identifier returned by a successful call to **vda_open**. Arguments *red*, *green*, and *blue* are integer values in the range from 0 through 255 decimal. **vda_map_rgb** returns an integer value representing the color in the current resolution.

For example, to find the integer value representing the color white in the current resolution, the program would use:

```
int dev = 0;
int white;
white = vda_map_rgb (dev, 255, 255, 255);
```

14. Drawing Graphic Objects in a GRP Buffer

The drawing routines operate on Graphics Primitive Buffers, which can be in SDRAM or S3 Display memory.

a) Clear Buffer to Background

To clear the entire buffer to a background color, call:

int **grp_cls** (grp_buf_t **pb*, int *color*)

The **grp_buf_t** pointer *pb* must have been set up using **grp_alloc** or **vda_grp_vram**. The *color* must be a value returned by **vda_map_rgb** or its equivalent.

b) Draw a Point at X, Y Location

To draw a point in the GRP buffer at a location identified by it's X and Y coordinates in the buffer, the program calls:

int **grp_point** (grp_buf_t **pb*, int *color*, int *x*, int *y*)

The **grp_buf_t** pointer *pb* must have been set up using **grp_alloc** or **vda_grp_vram**. The *color* must be a value returned by **vda_map_rgb** or its equivalent. The pixel at coordinates (*x*, *y*) is set to the *color*.

c) Draw a Point at Address

To draw a one-pixel point in the GRP buffer at a location identified by its address in the buffer, the program calls:

int **grp_point_addr** (grp_buf_t **pb*, unsigned long *addr*, int *color*)

The **grp_buf_t** pointer *pb* must have been set up using **grp_alloc** or **vda_grp_vram**. The *color* must be a value returned by **vda_map_rgb** or its equivalent. The pixel at absolute address *addr* in the buffer is set to the *color*.

d) Draw Solid Line

To draw a solid line in the GRP buffer, the program calls:

int **grp_line** (grp_buf_t **pb*, int *color*, int *x0*, int *y0*, int *x1*, int *y1*)

The **grp_buf_t** pointer *pb* must have been set up using **grp_alloc** or **vda_grp_vram**. The *color* must be a value returned by **vda_map_rgb** or its equivalent. The line is drawn in the *color* from the pixel at coordinates (*x0*, *y0*) up to (but not including) the pixel at coordinates (*x1*, *y1*).

e) Draw Dotted-Dashed Line

To draw a dotted, dashed, or dot-dash line in the buffer, the program calls:

int **grp_dot_dash_line** (grp_buf_t **pb*, int *x0*, int *y0*, int *x1*, int *y1*,
int *fg*, int *bg*, int *ddtab*[], int *n_ddtab*)

The **grp_dot_dash_line** function draws a dot-dashed line from coordinate *x0, y0*, to (but not including) coordinate *x1, y1* in the display buffer given by *pb*.

A dot-dashed line is implemented as an alternating series of foreground and background pixels. For example, a dotted line might be 5 pixels foreground followed by 15 pixels background, 5 pixels foreground, 15 pixels background, and so forth.

The argument *ddtab* is an array of integer elements defining a dot-dash pattern to be repeated over the length of the line; *n_ddtab* specifies the number of elements in the array (that is, the length of the repeating pattern). The even elements of array *ddtab* specify numbers of foreground pixels in the line; these elements receive the color given by argument *fg*. The odd elements of *ddtab* specify numbers of background pixels, receiving color *bg*. A value of –1 for foreground color *fg* or background color *bg* leaves the current color unchanged.

For example, the dotted line above can be expressed as a repeating sequence of the two integers 5 and 15. In this case, *ddtab* would be initialized as follows:

```
int ddtab[] = {5, 15};
```

The **grp_buf_t** pointer *pb* must have been set up using **grp_alloc** or **vda_grp_vram**. The foreground and background colors *fg* and *bg* must be values returned by **vda_map_rgb** or its equivalent.

f) Fill Rectangle

To draw a rectangle in the buffer, the program calls:

int **grp_rect_fill** (grp_buf_t **pb*, int *color*, int *x0*, int *y0*, int *x1*, int *y1*)

The **grp_buf_t** pointer *pb* must have been set up using **grp_alloc** or **vda_grp_vram**. The *color* must be a value returned by **vda_map_rgb** or its equivalent. The rectangle is drawn in the specified *color* with one corner at the pixel with coordinates (*x0*, *y0*) and the other corner at the pixel (*x1*, *y1*).

g) Draw Polygon

To draw an arbitrary polygon in the buffer, the program calls:

int **grp_fill** (grp_buf_t **pb*, int *color*, int *x[]*, int *y[]*, int *npoints*)

The **grp_buf_t** pointer *pb* must have been set up using **grp_alloc** or **vda_grp_vram**. The *color* must be a value returned by **vda_map_rgb** or its equivalent. The polygon is drawn in the *color* with its first vertex at the pixel with coordinates (*x*[0], *y*[0]), the next vertex at (*x*[1], *y*[1]) and so on. The last vertex, with coordinates (*x*[*npoints* -1], *y*[*npoints* -1]), is connected to the first vertex to complete the polygon.

15. Drawing Text Objects in the GRP Buffer

The text drawing functions use font descriptions read from files in X11 BDF format (ASCII-encoded). BDF-format font description files may be obtained from X-Windows sources. The font description file can define up to 256 characters.

a) Load or Unload Font Description File

The TriMedia program uses a font descriptor structure (**grp_font_t**) to hold the font information that is downloaded from each file. The program can load as many fonts as desired. For each font, the program declares a pointer to a blank **grp_font_t** structure then loads it with the contents of the font description file by calling:

grp_font_t ***grp_load_font** (char **font_file*)

For example:

grp_font_t *font = grp_load_font (“10x20.bdf”);

When this function returns, pointer `font` can be passed as the font argument to the text drawing routines. When a given font is no longer needed, the descriptor can be freed with a call to:

int **grp_free_font** (grp_font_t **font*)

b) Text Objects

The drawing routines handle three types of text objects:

- A single character in integer representation (‘a’, ‘b’, etc.)
- A string of characters terminated with a null character (“This is a string.”)
- A fixed-length array or buffer of **char** values (char buf[2] = “ab”;)

c) Determine Size of Text Object

The program may need to know the size of an object to be displayed. To find the width in pixels of a single character in a given font, call:

int **grp_char_width** (grp_font_t **font*, int *character*)

The return value is the width. For example:

int char_wide = grp_char_width (default_font, ‘A’);

To find the width in pixels of a null-terminated string, call

int **grp_string_width** (grp_font_t **font*, char **string*)

For example:

```
int str_wide = grp_string_width (default_font,
                                  “Printed string.”);
```

To find the width in pixels of an array of characters, call

int **grp_buf_width** (grp_font_t **font*, char *buf*[], int *length*)

For example:

```
char text_buf[] = “ABC”;
int buf_wide = grp_buf_width (default_font, &text_buf, 3);
```

d) Draw Character, String, or Array

To draw a single character in a graphics buffer, the TriMedia program calls:

void **grp_draw_char** (grp_font_t **font*, grp_buf_t **pb*, int *x*, int *y*,
int *character*, int *fg*, int *bg*)

The font and buffer pointers *font* and *pb* are those returned by **grp_load_font** and **grp_alloc**, respectively. Arguments *x* and *y* specify the pixel location of the upper left corner of the character. The character is rendered in foreground color *fg* against a bounding rectangle of background color *bg*, both of which are integer values returned by **vda_map_rgb** or the equivalent. For example:

```
int ret = grp_draw_char (default_font, p_grp, 100, 100,
                          ‘A’, white, black);
```

To draw a null-terminated string of characters in a graphics buffer, the TriMedia program calls:

void **grp_draw_string** (grp_font_t **font*, grp_buf_t **pb*, int x, int y, char **string*, int *fg*, int *bg*)

The arguments for this call are the same as for a single character, except for the substitution of **char *string** for **int *character***.

For example:

```
ret = grp_draw_string (default_font, p_grp, 100, 100,
                                    "Print this!", red, green);
```

To draw a fixed-length array of characters in a graphics buffer, the TriMedia program calls:

void **grp_draw_buf** (grp_font_t **font*, grp_buf_t **pb*, int x, int y, char *buf*[], int *size*, int *fg*, int *bg*)

For example:

```
char text_buf[15] = "Now print this."
ret = grp_draw_string (default_font, p_grp, 100, 100,
                                    text_buf, 5, blue, yellow);
```

16. Advanced Functions

In addition to the general-purpose function **vda_write** (mentioned earlier with **vda_sync**), the Analog Output library provides routine **vda_write_yuv** for writing YUV data from a buffer to display memory. Function **vda_ioctl**(int *dev*, **VDA_WAIT_WRITE**) allows the TM program to pause program execution until the previous **vda_write** or **vda_write_yuv** has returned.

The program can read an arbitrary region of the S3 video display memory with a call to routine **vda_read**. The program can copy a rectangle from one area of S3 video display memory to another with a call to routine **vda_region_copy**.

The parameters for these calls are given in the Analog Output Library reference later in this manual.

III. PROGRAM EXAMPLES

A. Host Program Example ex1.c

Here is a listing of the Host program example source file **ex1.c**, located in **examples\host\ex1**.

This program executes one or more programs on one or more TriMedia, using a syntax like **tmmprun**:

ex1 –exec *pgm1 arg(s)* –**exec** *pgm2 arg(s)* ...

```
/*****************************************************************************
**
**  File:      ex1.c - Host example - tmmprun
**
**  Copyright © 1999; Alacron Inc.
**
**  Description:
**
**      This is a simplified version of the tmmprun program, which allows
**      us to load a separate program on each TM processor.
**
**  History:
**
**      29-Jun-99, tjc:       Created
**
*****************************************************************************/

/*----------------------- HEADER FILES ------------------------------------*/

#include <alfast_nt.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <tmmanapi.h>
#include <tmcrt.h>

/*----------------------- PRIVATE CONSTANTS -------------------------------*/

#define MAXDEV        32

/*----------------------- PRIVATE MACROS ----------------------------------*/

/*----------------------- PRIVATE TYPES -----------------------------------*/

/*----------------------- PRIVATE DATA ------------------------------------*/
static Int32 handles[MAXDEV];
static CRunTimeParameterBlock crtparams[MAXDEV];
static UInt32 crt_handles[MAXDEV];

/*----------------------- PUBLIC DATA -------------------------------------*/

/*----------------------- PRIVATE ROUTINE REFERENCES ----------------------*/

PRIVATE int start (int dev, int argc, char *argv[]);
PRIVATE void wait_for_completion (int dev);

/*----------------------- PUBLIC ROUTINES ---------------------------------*/

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

```
	int ndev = 0;
	int idx = -1;

	/*--- initialize the C runtime support ---*/

	cruntimeInit ();

	/*--- extract arguments, and start running TM program ---*/

	for (i = 1; i < argc; )
	{
		if (strcmp (argv[i], "-exec") == 0)
		{
			if (++i < argc)
			{
				idx = i;
				while ((++i < argc) && (strcmp (argv[i], "-exec"))
!= 0)
					;
			}

			if (ndev < MAXDEV)
			{
				rval = start (ndev, i - idx, &argv[idx]);
				if (rval)
				{
					printf ("Aborting\n");
					return 0;
				}
				ndev ++;
			}
		}
		else
			i ++;
	}

	/*--- now wait for each to complete ---*/

	for (i = 0; i < ndev; i ++)
		wait_for_completion (i);
	/*--- close each device ---*/

	for (i = 0; i < ndev; i ++)
		tmmanDSPClose (handles[i]);
	return 0;
}

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

PRIVATE int start (int dev, int argc, char *argv[])
{
	int rval;

	/*--- open TriMedia processor, get handle ---*/

	rval = tmmanDSPOpen (dev, &handles[dev]);
	if (rval != statusSuccess)
	{
		printf ("ERROR: tmmanDSPOpen device %d failed\n", dev);
		return -1;
	}

	/*--- hold TriMedia processor in reset ---*/

	tmmanDSPStop (handles[dev]);

	/*--- load TriMedia program in default DRAM locations ---*/

	rval = tmmanDSPLoad (handles[dev], constTMManDefault, argv[0]);
	if (rval != statusSuccess)
	{
```

```
                printf (“ERROR: tmmanDSPLoad device %d failed\n”, dev);
                return -1;
        }

        /*--- set up C runtime parameters and completion event ---*/

        memset (&crtparams[dev], 0, sizeof (crtparams[dev]));
        crtparams[dev].OptionBitmap = 0;
        crtparams[dev].StdInHandle = (DWORD) GetStdHandle (STD_INPUT_HANDLE);
        crtparams[dev].StdOutHandle = (DWORD) GetStdHandle (STD_OUTPUT_HANDLE);
        crtparams[dev].StdErrHandle = (DWORD) GetStdHandle (STD_ERROR_HANDLE);
        crtparams[dev].OptionBitmap |= constCRunTimeFlagsUseSynchObject;
        crtparams[dev].VirtualNodeNumber = dev;
        crtparams[dev].CRTThreadCount = 1;
        crtparams[dev].SynchronizationObject =
                (UInt32) CreateEvent ( NULL, FALSE, FALSE, NULL);

        if (!crtparams[dev].SynchronizationObject)
        {
                printf (“ERROR: CreateEvent failed\n”);
                return -1;
        }

        rval = cruntimeCreate (dev, argc, &argv[0], &crtparams[dev],
                &crt_handles[dev]);

        if (!rval)
        {
                printf (“ERROR: cruntimeCreate failed\n”);
                return -1;
        }

        /*--- start program running on TriMedia (deassert reset) ---*/

        rval = tmmanDSPStart (handles[dev]);
        if (rval != statusSuccess)
        {
                printf (“ERROR: tmmanDSPStart failed\n”);
                return -1;
        }

        return 0;
}

PRIVATE void wait_for_completion (int dev)
{
        int exitcode;

        WaitForSingleObject ((HANDLE) crtparams[dev].SynchronizationObject,
                INFINITE);

        /*--- halt TriMedia processor, shut down C runtime, close handle ---*/

        tmmanDSPStop (handles[dev]);
        cruntimeDestroy (crt_handles[dev], &exitcode);

        CloseHandle ((HANDLE) crtparams[dev].SynchronizationObject);
}
```

B. TriMedia Program Example frame.c

TriMedia example source file **frame.c** is located in **examples\target\ex1**.

```
/***************************************************************************
**
**  File:       frame.c - digital frame camera example
**
**  Copyright © 1999; Alacron Inc.
```

```
**
**  Description:
**
**      Input consists of multiple 8 bit taps, each fed to a separate
**      TM processor.
**
**      Input is taken to be 256 level grayscale, can copied to the
**      VGA display using the ICP.  VGA parameters are set in the file
**      vga.ini.
**
**  History:
**
**      24-Jun-99, tjc:         Created
**      03-Mar-00, tjc:         Added voltatile to input_buf
**
*************************************************************************/

/*----------------------- HEADER FILES -----------------------------------*/

#include <alfast_tm.h>
#include <vda.h>

/*----------------------- PRIVATE CONSTANTS ------------------------------*/

#define PROFILE                "tmc9700.cap"

/*----------------------- PRIVATE MACROS ---------------------------------*/

/*----------------------- PRIVATE TYPES ----------------------------------*/

/*----------------------- PRIVATE DATA -----------------------------------*/

static int capture_dev = 0;
static int vda_dev = 0;

/*
 *      buffer ready indicator
 */

static volatile int input_ready;
static alf_capture_buf_t * volatile input_buf;

/*----------------------- PUBLIC DATA ------------------------------------*/

/*----------------------- PRIVATE ROUTINE REFERENCES ---------------------*/

PRIVATE void frame_handler (int cause, alf_capture_buf_t *buf);
PRIVATE void frame_process (alf_capture_buf_t *input_buf);
PRIVATE int setup_display (void);
PRIVATE void copy_display (alf_capture_buf_t *input_buf);

/*----------------------- PUBLIC ROUTINES --------------------------------*/

EXPORT int main (int argc, char *argv[])
{
        handle hc;
        int rval;
        alf_capture_control_t capture;

/*--- Initialize the display variables ---*/

        rval = setup_display ();
        if (rval)
        exit (-1);

        /*--- attach to capture device ---*/

        hc = alf_capture_attach (capture_dev);
        if (ALF_ISERROR (hc))
        {
                printf ("ERROR: alf_capture_attach %d failed\n", capture_dev);
```

```
        exit (1);
    }
    printf ("alf_capture_attach %d ok\n", capture_dev);

    /*--- Load the profile ---*/

    rval = alf_capture_load_profile (hc, PROFILE);
    if (ALF_ISERROR (rval))
    {
        printf ("ERROR: alf_capture_load_profile %s failed\n", PROFILE);
        printf ("Continuing anyway\n");
    }
    printf ("alf_capture_load_profile ok\n");

    /*--- setup capture control ---*/
    memset (&capture, 0, sizeof (capture));
    capture.alf_capture_nbuf = 2;
    capture.alf_capture_bufsize = -1;
    capture.alf_capture_mode = ALF_CAPTURE_CONTINUOUS;

    rval = alf_capture_set_control (hc, &capture);
    if (ALF_ISERROR (rval))
    {
        printf ("ERROR: alf_capture_set_control failed\n");
        exit (1);
    }
    printf ("alf_capture_configure ok\n");

    /*--- Start the capture ---*/

    rval = alf_capture_start (hc, frame_handler);
    if (ALF_ISERROR (rval))
    {
        printf ("ERROR: alf_capture_start failed\n");
        exit (1);
    }

    /*
     *      Main loop -
     *
     *      wait for input_ready to go true, then process buffer
     */

    input_ready = 0;
    for (;;)
    {
        while (!input_ready)
            ;

        frame_process (input_buf);

        input_ready = FALSE;
    }

}

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

/*****************************************************************************
**
**  PRIVATE - frame_handler - buffer ready handler
**
**  Description:
**
**      If the previous buffer has been handled, then save buffer pointer and
**      set ready flag.
**
*****************************************************************************/

PRIVATE void frame_handler (int cause, alf_capture_buf_t *buf)
{
```

```
        if (!input_ready)
        {
                input_buf = buf;
                input_ready = TRUE;
        }
}

/***********************************************************************
**
**  PRIVATE - frame_process - process the new buffer
**
**  Description:
**
**      Accumulate statistics, and display
**
***********************************************************************/

PRIVATE void frame_process (alf_capture_buf_t *input_buf)
{
copy_display (input_buf);
}

/***********************************************************************
**
**  PRIVATE - setup_display - read display parameters
**
**  Description:
**
***********************************************************************/

PRIVATE int setup_display (void)
{
        int rval;

        rval = vda_open (vda_dev, 0);
        if (rval)
        {
                printf ("ERROR: vda_open failed\n");
                return -1;
        }

        return ALF_NOERROR;
}

/***********************************************************************
**
**  PRIVATE - copy_display - copy current image buffer to display
**
**  Description:
**
***********************************************************************/

PRIVATE void copy_display (alf_capture_buf_t *input_buf)
{
        vda_write (vda_dev, VDA_DISPLAY, VDA_GRAY8,
                input_buf->alf_buf.alf_buf_data,
                input_buf->alf_buf.alf_buf_stride,
                0, 0,
                input_buf->alf_buf.alf_buf_nrows,
                        input_buf->alf_buf.alf_buf_ncols);
}
```

C. *TriMedia Program Example analog.c*

Here is a listing of the TriMedia program example source file **analog.c**, located in **examples\target\ex2**. The program logic is the same as in the **frame.c** example.

```
/**********************************************************************
```

```
**
** File:      analog.c - analog frame camera example
**
** Copyright © 1999; Alacron Inc.
**
** Description:
**
**      Input consists of up to 3 analog RS170 type inputs, each directed
**      to a separate TM processor.
**
**      Input is taken to be 256 level grayscale, can copied to the
**      VGA display using the ICP.  VGA parameters are set in the file
**      vga.ini.
**
** History:
**
**      24-Jun-99, tjc:       Created
**      03-Mar-00, tjc:       Changed globals to volatile
**
*****************************************************************************/

/*----------------------- HEADER FILES ------------------------------------*/

#include <alfast_tm.h>
#include <vda.h>

/*----------------------- PRIVATE CONSTANTS -------------------------------*/

#define PROFILE               "analog.cap"

/*----------------------- PRIVATE MACROS ----------------------------------*/

/*----------------------- PRIVATE TYPES -----------------------------------*/

/*----------------------- PRIVATE DATA ------------------------------------*/

static int capture_dev = 0;
static int vda_dev = 0;

/*
.*      buffer ready indicator
 */

static volatile int input_ready;
static alf_capture_buf_t * volatile input_buf;

/*----------------------- PUBLIC DATA -------------------------------------*/

/*----------------------- PRIVATE ROUTINE REFERENCES ----------------------*/

PRIVATE void analog_handler (int cause, alf_capture_buf_t *buf);
PRIVATE void analog_process (alf_capture_buf_t *input_buf);
PRIVATE int setup_display (void);
PRIVATE void copy_display (alf_capture_buf_t *input_buf);

/*----------------------- PUBLIC ROUTINES ---------------------------------*/

EXPORT int main (int argc, char *argv[])
{
       handle hc;
       int rval;
       alf_capture_control_t capture;

       /*--- Initialize the display variables ---*/

       rval = setup_display ();
       if (rval)
       exit (0);

       /*--- attach to capture device ---*/
```

```
        hc = alf_capture_attach (capture_dev);
        if (ALF_ISERROR (hc))
        {
                printf ("ERROR: alf_capture_attach %d failed\n", capture_dev);
                exit (1);
        }
        printf ("alf_capture_attach %d ok\n", capture_dev);

        /*--- Load the profile ---*/

        rval = alf_capture_load_profile (hc, PROFILE);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture_load_profile %s failed\n", PROFILE);
                printf ("Continuing anyway\n");
        }
        printf ("alf_capture_load_profile ok\n");

        /*--- setup capture control ---*/

        memset (&capture, 0, sizeof (capture));
        capture.alf_capture_nbuf = 2;
        capture.alf_capture_bufsize = -1;
        capture.alf_capture_mode = ALF_CAPTURE_CONTINUOUS;

        rval = alf_capture_set_control (hc, &capture);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture_set_control failed\n");
                exit (1);
        }
        printf ("alf_capture_configure ok\n");

        /*--- Start the capture ---*/

        rval = alf_capture_start (hc, analog_handler);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture_start failed\n");
                exit (1);
        }

        /*
         *      Main loop -
         *
        .*      wait for input_ready to go true, then process buffer
         */

        input_ready = 0;
        for (;;)
        {
                while (!input_ready)
                        ;
                analog_process (input_buf);
                input_ready = FALSE;
        }

}

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

/***************************************************************************
**
**  PRIVATE - analog_handler - buffer ready handler
**
**  Description:
**
**      If the previous buffer has been handled, then save buffer pointer and
**      set ready flag.
**
***************************************************************************/
```

```
PRIVATE void analog_handler (int cause, alf_capture_buf_t *buf)
{
        if (!input_ready)
        {
                input_buf = buf;
                input_ready = TRUE;
        }
}

/***************************************************************************
**
**  PRIVATE - analog_process - process the new buffer
**
**  Description:
**
**      Accumulate statistics, and display
**
***************************************************************************/

PRIVATE void analog_process (alf_capture_buf_t *input_buf)
{
copy_display (input_buf);
}

/***************************************************************************
**
**  PRIVATE - setup_display - read display parameters
**
**  Description:
**
***************************************************************************/

PRIVATE int setup_display (void)
{
        int rval;

        rval = vda_open (vda_dev, 0);
        if (rval)
        {
                printf ("ERROR: vda_open failed\n");
                return -1;
        }

        return ALF_NOERROR;
}

/***************************************************************************
**
**  PRIVATE - copy_display - copy current image buffer to display
**
**  Description:
**
***************************************************************************/

PRIVATE void copy_display (alf_capture_buf_t *input_buf)
{
        vda_write (vda_dev, VDA_DISPLAY, VDA_GRAY8,
        input_buf->alf_buf.alf_buf_data,
                input_buf->alf_buf.alf_buf_stride,
                0, 0,
                input_buf->alf_buf.alf_buf_nrows,
                input_buf->alf_buf.alf_buf_ncols);
}
```

D. TriMedia Program Example ntsc.c

Here is a listing of the TriMedia program example source file **ntsc.c**, located in **examples\target\ex3**. The program uses the Host-controlled analog output library API. The input mode (odd fields only) is unique to the NTSC environment.

This program accepts two optional arguments. The **–r** *resolution* argument specifies a display resolution. the **–d** *device* argument specifies an alternate VDA device.

```
/***************************************************************************
**
** File:       ntsc.c - NTSC input example
**
** Copyright © 1999; Alacron Inc.
**
** Description:
**
**      Setup for NTSC input using SAA7111A.  Acquire odd field of input
**      data and display on the system's VGA screen. The VGA parameters
**      are stored in an ini-file, called vga.ini.
**
** History:
**
**      24-Jun-99, tjc:        Created
**      25-Aug-99, tjc:        Modified to use VDA library
**      03-Mar-00, tjc:        Changed some globals to be volatile
**
***************************************************************************/

/*----------------------- HEADER FILES ------------------------------------*/

#include <alfast_tm.h>
#include <vda.h>

/*----------------------- PRIVATE CONSTANTS -------------------------------*/

#define PROFILE                "ntsc60.cap"

#define NBUFS          3

/*----------------------- PRIVATE MACROS ----------------------------------*/

/*----------------------- PRIVATE TYPES -----------------------------------*/

/*----------------------- PRIVATE DATA ------------------------------------*/
static int capture_dev = 0;
static int vda_dev = 0;
static char *resolution_string = NULL;
static int display_xoffset = 0;
static int display_yoffset = 0;
static vda_params_t vda_params;

/*
 *      buffer ready indicator
 */

static volatile int input_ready;
static alf_capture_buf_t * volatile input_buf;

/*
 *      frame counting variables
 */

static int oddcount;
static int evencount;

/*----------------------- PUBLIC DATA -------------------------------------*/

/*----------------------- PRIVATE ROUTINE REFERENCES ----------------------*/

PRIVATE void ntsc_handler (int cause, alf_capture_buf_t *buf);
```

```
PRIVATE void ntsc_process (handle h, alf_capture_buf_t *input_buf);
PRIVATE void ntsc_stats (alf_capture_buf_t *input_buf);
PRIVATE int setup_display (char *resolution_string);
PRIVATE int capture_zero_buffers (handle hc);
PRIVATE void zero_buf (alf_buf_t *pb);

/*---------------------- PUBLIC ROUTINES --------------------------------*/

EXPORT int main (int argc, char *argv[])
{
        int i;
        handle hc;
        int rval;
        alf_capture_control_t capture;
        alf_attribute_t value;

        for (i = 1; i < argc; i ++)
                if (*argv[i] == '-')
                        switch (*(argv[i]+1))
                        {
                        case 'r':
                                if (++i < argc)
                                        resolution_string = argv[i];
                                break;
                        case 'd':
                                if (++i < argc)
                                        vda_dev = atoi (argv[i]);
                                break;
                        }

        /*--- Initialize the display ---*/

        setup_display (resolution_string);

        /*--- attach to capture device ---*/

        hc = alf_capture_attach (capture_dev);

        if (ALF_ISERROR (hc))
        {
                printf ("ERROR: alf_capture_attach %d failed\n", capture_dev);
                exit (1);
        }
        printf ("alf_capture_attach %d ok\n", capture_dev);

        /*--- Load the NTSC 60 Hz profile ---*/

        rval = alf_capture_load_profile (hc, PROFILE);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture_load_profile %s failed\n", PROFILE);
                exit (1);
        }
        printf ("alf_capture_load_profile ok\n");

        /*--- setup display x and y offset (to center display in screen) ---*/

        rval = alf_capture_query_attribute (hc, "INPUT_PIXELS_PER_LINE", &value);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: can't find IMAGE_PIXELS_PER_LINE\n");
                exit (1);
        }
        printf ("pixels_per_line %d\n", value.lvalue);
        display_xoffset = (vda_params.xres - value.lvalue) / 2;
        if (display_xoffset < 0)
                display_xoffset = 0;
        rval = alf_capture_query_attribute (hc, "INPUT_LINES_PER_BUFFER", &value);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: can't find INPUT_LINES_PER_BUFFER\n");
```

```
		exit (1);
	}
	printf ("lines_per_buffer %d\n", value.lvalue);

	display_yoffset = (vda_params.yres - value.lvalue*2) / 2;
	if (display_yoffset < 0)
		display_yoffset = 0;

	/*--- setup capture control ---*/

	memset (&capture, 0, sizeof (capture));
	capture.alf_capture_nbuf = NBUFS;
	capture.alf_capture_bufsize = 0;	/* for YUV this is ignored */
	capture.alf_capture_mode = ALF_CAPTURE_CONTINUOUS | ALF_CAPTURE_INTERLACE;

	rval = alf_capture_set_control (hc, &capture);
	if (ALF_ISERROR (rval))
	{
		printf ("ERROR: alf_capture_set_control failed\n");
		exit (1);
	}

	/*--- Zero out the input buffers ---*/

	capture_zero_buffers (hc);

	/*--- Start the capture ---*/

	rval = alf_capture_start (hc, ntsc_handler);
	if (ALF_ISERROR (rval))
	{
		printf ("ERROR: alf_capture_start failed\n");
		exit (1);
	}

	/*
	 *	Main loop -
	 *
	 *	wait for input_ready to go true, then process buffer
	 */

	input_ready = FALSE;
	for (;;)
	{
		while (!input_ready)
			;
		ntsc_process (hc, input_buf);
		input_ready = FALSE;
	}

}

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

/*******************************************************************
**
**  PRIVATE - ntsc_handler - buffer ready handler
**
**  Description:
**
**	If the previous buffer has been handled, then save buffer pointer and
**	set ready flag.
**
*******************************************************************/

PRIVATE void ntsc_handler (int cause, alf_capture_buf_t *buf)
{
	if (!input_ready)
	{
		input_buf = buf;
		input_ready = TRUE;
```

```
        }
}
/**********************************************************************
**
**  PRIVATE - ntsc_process - process the new buffer
**
**  Description:
**
**      Accumulate statistics, and display
**
**********************************************************************/

PRIVATE void ntsc_process (handle h, alf_capture_buf_t *input_buf)
{
        int rval;

        ntsc_stats (input_buf);

        rval = vda_write_yuv (vda_dev, VDA_DISPLAY, VDA_COPY,

                input_buf->alf_bufY.alf_buf_data,
                input_buf->alf_bufY.alf_buf_stride,
                input_buf->alf_bufU.alf_buf_data,
                input_buf->alf_bufU.alf_buf_stride,
                input_buf->alf_bufV.alf_buf_data,
                input_buf->alf_bufV.alf_buf_stride,
                display_xoffset, display_yoffset,
                input_buf->alf_bufY.alf_buf_nrows, input_buf-
>alf_bufY.alf_buf_ncols);
        if (rval)
        {
                printf ("ERROR: vda_write_yuv failed - %d\n", rval);
                exit (1);
        }

}

/**********************************************************************
**
**  SHARED - ntsc_stats - accumulate statistics
**
**  Description:
**
**********************************************************************/

PRIVATE void ntsc_stats (alf_capture_buf_t *input_buf)
{
        static alf_timing_t ts;
        static int iflag = FALSE;

        if (!iflag)
        {
                alf_timing_start (&ts);
                iflag = TRUE;
        }

        if (input_buf->alf_capture_field == 0)
                oddcount ++;
        else
                evencount ++;

        if ((oddcount > 100) || (evencount > 100))
        {
                float elapsed;
                elapsed = alf_timing_elapsed (&ts);
                alf_timing_start (&ts);
                printf ("elapsed %f oddcount %d (%.2f/sec) evencount %d(%.2f/sec)
\n",
                        elapsed,
                        oddcount, ((float)oddcount) / elapsed,
                        evencount, ((float)evencount) / elapsed );
```

```
                oddcount = 0;
                evencount = 0;
        }
}

/***************************************************************************
**
**  PRIVATE - setup_display - read display parameters
**
**  Description:
**
***************************************************************************/

PRIVATE int setup_display (char *resolution_string)
{
        int rval;

        rval = vda_open (vda_dev, 0);
        if (rval)
        {
                printf ("ERROR: vda_open failed - %d\n", rval);
                exit (1);
        }

        /*
         *      If the display is not host controlled, then
         *      try to set the requested resolution
         */

        if (!vda_host_controlled (vda_dev))
        {
                if (resolution_string != NULL)
                {
                        rval = vda_ioctl (vda_dev, VDA_SET_RESOLUTION,
resolution_string);
                        if (rval)
                                printf ("WARNING: VDA_SET_RESOLUTION failed-using "
                                "current setup\n");
                }


                vda_cls (vda_dev, VDA_DISPLAY, vda_map_rgb (vda_dev, 100, 100,
100));
        }

        rval = vda_ioctl (vda_dev, VDA_GET_PARAMS, &vda_params);
        if (rval)
        {
                printf ("ERROR: VDA_GET_PARAMS failed\n");
                exit (1);
        }

        return ALF_NOERROR;
}


PRIVATE int capture_zero_buffers (handle hc)
{
        int rval;
        alf_capture_status_t status;
        alf_capture_buf_t *pb;
        int i;

        rval = alf_capture_status (hc, &status);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture_status failed\n");
                return -1;
        }

        for (i = 0; i < NBUFS; i ++)
```

```
        {
                pb = &status.alf_capture_buflist[i];
                zero_buf (&pb->alf_bufY);
                zero_buf (&pb->alf_bufU);
                zero_buf (&pb->alf_bufV);
        }

        return 0;
}

PRIVATE void zero_buf (alf_buf_t *pb)
{
        memset (pb->alf_buf_data, 0, pb->alf_buf_size);
        _cache_copyback (pb->alf_buf_data, pb->alf_buf_size);
        _cache_invalidate (pb->alf_buf_data, pb->alf_buf_size);
}
```

E. *TriMedia Program Example vidout.c*

TriMedia example source file **vidout.c** is located in **examples\target\ex4**.

```
/***************************************************************************
**
** File:        vidout.c - generate video output
**
** Copyright © 1999; Alacron Inc.
**
** Description:
**
**      This example sets up a YUV video output from TM0 to be received
**      by TM1 using the NTSC example (example 3).  The example operates by
**      allocate DO_NBUFS buffers, and rendering (using VDA functions) a
**      sequence of video "frames".   The buffers are cycled through round
**      robin using Digital Output API functions.
**
** History:
**
** 12-Aug-99, tjc:     Created
**      03-Mar-00, tjc:        Changed some globals to volatile
**
****************************************************************************/

/*---------------------- HEADER FILES -----------------------------------*/
#include <alfast_tm.h>
#include <vda.h>
#include <tm1/tmDMA.h>
#include <tmlib/tmlibc.h>
#include <tm1/tmICP.h>
#include <assert.h>
#include <math.h>

/*---------------------- PRIVATE CONSTANTS ------------------------------*/

/*
 *      NBUFS for digital output
 */

#define DO_NBUFS             16

/*
 *      Name of the Digital output profile
 */

#define VO_PROFILE           "yuv.dop"

/*
 *      Dimensions of the output data
 */

#define NR                          256
```

```
#define NC                              256

/*
 *      A useful constant
 */

#define PI      3.14159265358979323846

/*----------------------- PRIVATE MACROS ---------------------------------*/

#define RGB(r,g,b)     ( (®<<16) | ((g) << 8) | (b))

/*----------------------- PRIVATE TYPES ----------------------------------*/

/*----------------------- PRIVATE DATA -----------------------------------*/

static grp_buf_t image;                              / SDRAM image buffer */
static volatile int startcount[DO_NBUFS];    /* counter for callbacks */
static volatile int completecount[DO_NBUFS];/* counter for callbacks */

static alf_digout_status_t do_status;

/*----------------------- PUBLIC DATA ------------------------------------*/

/*----------------------- PRIVATE ROUTINE REFERENCES ---------------------*/

PRIVATE int setup_digital_output (handle *ph);
PRIVATE void do_handler (int cause, alf_digout_buf_t *pbuf);
PRIVATE int start_digital_output (handle h);
PRIVATE int lim255 (int value);
PRIVATE int rgb2yuv (unsigned char *a, int ia, unsigned char *y, int iy, unsigned
char *u, int iu, unsigned char *v, int iv, int nr, int nc);
PRIVATE void polar2cart (int radius, int angle, int *dx, int *dy);

/*----------------------- PUBLIC ROUTINES --------------------------------*/

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

        /*--- Set up digital output ---*/
        rval = setup_digital_output (&ho);
        if (ALF_ISERROR (rval))
        {
                printf (“ERROR: setup_digital_output failed\n”);
                exit (1);
        }

        /*--- Start up digital output ---*/

        rval = start_digital_output (ho);
        if (ALF_ISERROR (rval))
        {
                printf (“ERROR: start_digital_output failed\n”);
                exit (1);
        }

        /*--- That’s all, we wait for the operator to type <return> ---*/

        getchar ();
{
extern int vointcount;
printf (“intcount %d\n”, vointcount);
}

        /*--- Stop digital output ---*/

        rval = alf_digout_stop (ho);
        if (ALF_ISERROR (rval))
```

```
	{
		printf ("ERROR: stop_digital_output failed\n");
		exit (1);
	}

	/*--- Print out the interrupt counts ---*/

	for (i = 0; i < DO_NBUFS; i ++)
		printf ("%6d %6d\n", startcount[i], completecount[i]);

}

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

/**************************************************************************
**
**  PRIVATE - setup_digital_output - set up digital output
**
**  Description:
**
**************************************************************************/

PRIVATE int setup_digital_output (handle *ph)
{
	handle h;
	int rval;
	alf_digout_control_t dsetup;
	alf_digout_buf_t *pbuf;
	int i;
	int j;
	int x0;
	int y0;
	int dy;
	int dx;

	/*--- Attach to DIGOUT device ---*/

	h = alf_digout_attach (0);
	if (ALF_ISERROR (h))
	{
		printf ("ERROR: alf_digout_attach failed\n");
		exit (1);
	}

	/*--- Load the profile ---*/

	rval = alf_digout_load_profile (h, VO_PROFILE);
	if (ALF_ISERROR (rval))
	{
		printf ("ERROR: alf_digout_load_profile failed %d\n", rval);
		exit (1);
	}

	/*
	 *	Prepare alf_digout_control_t argument we want:
	 *	-	DO_NBUFS buffers
	 *	-	run continuous
	 */

	memset (&dsetup, 0, sizeof (dsetup));
	dsetup.alf_digout_nbuf = DO_NBUFS;
	dsetup.alf_digout_mode = ALF_DIGOUT_CONTINUOUS;

	/*--- Setup DIGOUT control ---*/

	rval = alf_digout_set_control (h, &dsetup);
	if (ALF_ISERROR (rval))
	{
		printf ("ERROR: alf_digout_set_control failed\n");
		exit (1);
	}
```

```
        /*
         *      Retrieve buffer addresses
         */

        rval = alf_digout_status (h, &do_status);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_digout_status failed\n");
                exit (1);
        }

        /*
         *      Fill a pattern into each buffer
         *
         *      This pattern consists of a small filled rectangle, drawn
         *      on a gray background, it's position rotating around a
         *      circle.
         */

        image = grp_alloc (NR, NC, 3);

        pbuf = do_status.alf_digout_buf_list;
        for (i = 0; i < DO_NBUFS; i ++)
        {
                pbuf = &do_status.alf_digout_buf_list[i];

/*--- Zero the buffers ---*/

        memset (pbuf->alf_digout_bufY.alf_buf_data, 0,
                pbuf->alf_digout_bufY.alf_buf_size);
        memset (pbuf->alf_digout_bufU.alf_buf_data, 128,
                pbuf->alf_digout_bufU.alf_buf_size);
        memset (pbuf->alf_digout_bufV.alf_buf_data, 128,
                pbuf->alf_digout_bufV.alf_buf_size);

        /*--- Clear the image to gray ---*/

        grp_cls (image, RGB (80, 80, 80));

        /*--- Draw a couple of lines ---*/

        grp_line (image, RGB (200, 200, 200), 0, 0, NC-1, NR-1);
        grp_line (image, RGB (200, 200, 200), 0, NR-1, NC-1, 0);

        /*--- Compute the x and y coordinate to draw box ---*/

        polar2cart (50, i*360/DO_NBUFS, &dx, &dy);

        x0 = NC/2 + dx - 10;
        y0 = NR/2 + dy - 10;

        /*
         *      Draw the box - in this case, we draw horizontal lines
         *      blue for even lines, green for odd lines
         */

        for (j = 0; j < 20; j ++)
                grp_line (image,
                        (y0+j)&1 ? RGB (0, 255, 0) : RGB (0, 0, 255),
                        x0, y0 + j, x0 + 20, y0 + j);

        /*
         *      we convert the RGB image to YUV (someday we might have
         *      a library function
         */

        rgb2yuv ((unsigned char*) image->grp_base, image->grp_vstride,

                pbuf->alf_digout_bufY.alf_buf_data,
                pbuf->alf_digout_bufY.alf_buf_stride,
```

```
                pbuf->alf_digout_bufU.alf_buf_data,
                pbuf->alf_digout_bufU.alf_buf_stride,
                pbuf->alf_digout_bufV.alf_buf_data,
                pbuf->alf_digout_bufV.alf_buf_stride,
                image->grp_yres,
                image->grp_xres);

        /*
         *      remember to copyback the cache - otherwise the
         *      the DIGOUT DMA hardware won't see the real data
         */

                _cache_copyback (pbuf->alf_digout_bufY.alf_buf_data,
                pbuf->alf_digout_bufY.alf_buf_size);
                _cache_copyback (pbuf->alf_digout_bufU.alf_buf_data,
                pbuf->alf_digout_bufU.alf_buf_size);
                _cache_copyback (pbuf->alf_digout_bufV.alf_buf_data,
                pbuf->alf_digout_bufV.alf_buf_size);
        }

        *ph = h;
        return ALF_NOERROR;

}


/*****************************************************************************
**
**  PRIVATE - do_handler - DIGOUT interrupt handler
**
**  Description:
**
**      In this example we don't really do anything other than increment
**      some counters.
**
*****************************************************************************/

PRIVATE void do_handler (int cause, alf_digout_buf_t *pbuf)
{
        switch (cause)
        {
        case ALF_DIGOUT_COMPLETE:
                completecount[pbuf->alf_digout_buf_idx] ++;
                break;
        case ALF_DIGOUT_START:
                startcount[pbuf->alf_digout_buf_idx] ++;
                break;
        }
}

/*****************************************************************************
**
**  PRIVATE - start_digital_output - start up digital output
**
**  Description:
**
**      Just call the DIGOUT startup function
**
*****************************************************************************/

PRIVATE int start_digital_output (handle h)
{
        int rval;

        rval = alf_digout_start (h, do_handler);
        if (ALF_ISERROR (rval))
                return rval;
        return ALF_NOERROR;
}

#define B               0
```

```
#define G               1
#define R               2

/***************************************************************************
**
**  PRIVATE - lim255 - limit value to 0-255
**
**  Description:
**
***************************************************************************/

PRIVATE int lim255 (int value)
{
        if (value < 0)
                return 0;
        if (value > 255)
                return 255;
        return value;
}

/***************************************************************************
**
**  PRIVATE - rgb2yuv - convert RGB image to YUV
**
**  Description:
**
***************************************************************************/

PRIVATE int rgb2yuv (unsigned char *a, int ia, unsigned char *y, int iy, unsigned
char *u, int iu, unsigned char *v, int iv, int nr, int nc)
{
        int i;
        int j;
        int u0, u1, v0, v1;
        unsigned char *ps;

        for (i = 0; i < nr; i ++)
        {
                /*--- compute Y component ---*/
                ps = a;
                for (j = 0; j < nc; j ++, ps += 3)
                y[j] = (((30*ps[R]) + (59*ps[G]) + (11*ps[B])) + 50)/ 100;
                /*--- compute UV components, subsamping by 2 ---*/
                ps = a;
                for (j = 0; j < nc; j += 2, ps += 6)
                {
                        u0 = ps[B] - y[j] + 128;
                        u1 = ps[B+3] - y[j+1] + 128;
                        u[j/2] = lim255 ((u0 + u1) / 2);

                        v0 = ps[R] - y[j] + 128;
                        v1 = ps[R+3] - y[j+1] + 128;
                        v[j/2] = lim255 ((v0 + v1) / 2);
                }
                a += ia;
                y += iy;
                u += iu;
                v += iv;
        }

        return 0;
}

/***************************************************************************
**
**  SHARED - polar2cart - convert polar to cartesian coordinate
**
**  Description:
**
***************************************************************************/
```

```
PRIVATE void polar2cart (int radius, int angle, int *dx, int *dy)
{
	*dx = (int) (radius * cos ( (((float)angle) / 360.0) * 2.0 * PI) + 0.5);
	*dy = (int) (radius * sin ( (((float)angle) / 360.0) * 2.0 * PI) + 0.5);
}
```

F. TriMedia Program Example send.c

TriMedia program **send.c** is located in **examples\target\ex5**.

```
/**************************************************************************
**
** File:       send.c - send blocks of data using Digital Output
**
** Copyright © 1999; Alacron Inc.
**
** Description:
**
**      This example fills a buffer of data with a pseudo random pattern
**      then transmits it out the Digital Output port.  The Cross bar is
**      configured to driver Data Valid with data bit 7.
**
**      On the send side, we need to allocate a slightly larger buffer so
**      that the data may be framed with D7 set to zero.
**
** History:
**
**      20-Aug-99, tjc:       Created
**
**************************************************************************/

/*----------------------- HEADER FILES ------------------------------------*/

#include <alfast_tm.h>
#include "shared.h"

/*----------------------- PRIVATE CONSTANTS -------------------------------*/

#define DIGOUT_PROFILE "send.dop"
#define NBUFS                 2
#define DIGOUT_DEVICE  0

/*----------------------- PRIVATE MACROS ----------------------------------*/

/*----------------------- PRIVATE TYPES -----------------------------------*/

/*----------------------- PRIVATE DATA ------------------------------------*/

static alf_buf_t buffers[NBUFS];
static volatile int doneflag;
static volatile int lastidx;

/*----------------------- PUBLIC DATA -------------------------------------*/

/*----------------------- PRIVATE ROUTINE REFERENCES ----------------------*/

PRIVATE void send_callback (int cause, alf_digout_buf_t *pbuf);
PRIVATE void fillpattern (unsigned char *p, int n);

/*----------------------- PUBLIC ROUTINES ---------------------------------*/

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

int main (int argc, char *argv[])
```

```
{
        int i;
        handle h;
        int rval;
        alf_digout_control_t ctrl;
        alf_digout_status_t status;

        /*--- Attach DIGOUT device ---*/

        h = alf_digout_attach (DIGOUT_DEVICE);
        if (ALF_ISERROR (h))
        {
                printf ("ERROR: alf_digout_attach failed\n");
                exit (1);
        }

        /*--- Load profile ---*/

        rval = alf_digout_load_profile (h, DIGOUT_PROFILE);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_digout_load_profile failed\n");
                exit (1);
        }

        /*--- initialize control parameters ---*/

        memset (&ctrl, 0, sizeof (ctrl));
        ctrl.alf_digout_nbuf = NBUFS;
        ctrl.alf_digout_mode = ALF_DIGOUT_SINGLE;
        rval = alf_digout_set_control (h, &ctrl);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_digout_set_control failed\n");
                exit (1);
        }

        /*--- Retrieve buffer addresses ---*/

        rval = alf_digout_status (h, &status);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_digout_status failed\n");
                exit (1);
        }

        for (i = 0; i < NBUFS; i ++)
                buffers[i] = status.alf_digout_buf_list[i].alf_digout_buf;

        /*--- Fill buffers ---*/

        for (i = 0; i < NBUFS; i ++)
        {
                fillpattern (buffers[i].alf_buf_data, buffers[i].alf_buf_size);
        }

        /*--- Start ---*/

        for (;;)
        {
                /*--- Fire it off ---*/

                doneflag = 0;
                rval = alf_digout_start (h, send_callback);
                if (ALF_ISERROR (rval))
                {
                        printf ("ERROR: alf_digout_start failed\n");
                        exit (1);
                }

                /*--- wait for completion ---*/
```

```
                while (!doneflag)
                        ;
                printf ("done lastidx = %d\n", lastidx);
                fillpattern (buffers[lastidx].alf_buf_data,
                buffers[lastidx].alf_buf_size);
                printf ("Type <cr> to send next\n");
                getchar ();
        }

}

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

/*****************************************************************************
**
**  PRIVATE - send_callback -
**
**  Description:
**
*****************************************************************************/

PRIVATE void send_callback (int cause, alf_digout_buf_t *pbuf)
{
        if (cause == ALF_DIGOUT_COMPLETE)
        {
                doneflag ++;
                lastidx = pbuf->alf_digout_buf_idx;
        }
}

/*****************************************************************************
**
**  PRIVATE - fillpattern -
**
**  Description:
**
*****************************************************************************/

PRIVATE void fillpattern (unsigned char *p, int n)
{
        int i;
        int offset = (n - BLOCKSIZE) / 2;
        static int seed = 0;

        srand (seed);
        memset (p, 0, n);
        p[offset] = seed | 0x80;
        for (i = 1; i < BLOCKSIZE; i ++)
                p[i + offset] = rand () | 0x80;

        _cache_copyback (p, n);
        seed = (++seed % 128);
}
```

G. *TriMedia Program Example recv.c*

TriMedia program **recv.c** is located in **examples\target\ex5**.

```
/****************************************************************************
**
** File:       recv.c - receive using RAW input
**
** Copyright © 1999; Alacron Inc.
**
** Description:
**
```

```
**      This program is set up to receive RAW input from the send program
**
** History:
**
**      20-Aug-99, tjc:         Created
**      03-Mar-00, tjc:         Changed some globals to volatile
**
*****************************************************************************/

/*----------------------- HEADER FILES ------------------------------------*/

#include <alfast_tm.h>
#include "shared.h"

/*----------------------- PRIVATE CONSTANTS -------------------------------*/

#define RECV_PROFILE_NAME     "recv.cap"
#define RECV_DEVICE                  0
#define RECV_NBUFS                   2

/*----------------------- PRIVATE MACROS ----------------------------------*/

/*----------------------- PRIVATE TYPES -----------------------------------*/

/*----------------------- PRIVATE DATA ------------------------------------*/

static volatile int doneflag;
static alf_capture_buf_t * volatile pbuf;
static volatile int intcount;

/*----------------------- PUBLIC DATA -------------------------------------*/

/*----------------------- PRIVATE ROUTINE REFERENCES ----------------------*/

PRIVATE void recv_callback (int cause, alf_capture_buf_t *pb);
PRIVATE void checkbuf (unsigned char *p, int n);

/*----------------------- PUBLIC ROUTINES ---------------------------------*/

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

EXPORT int main (int argc, char *argv[])
{
        handle h;
        alf_capture_control_t ctrl;
        int rval;

        /*--- Attach to capture device ---*/

        h = alf_capture_attach (RECV_DEVICE);
        if (ALF_ISERROR (h))
        {
                printf ("ERROR: alf_capture_attach failed\n");
                exit (1);
        }

        /*--- Load profile ---*/

        rval = alf_capture_load_profile (h, RECV_PROFILE_NAME);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture_load_profile failed\n");
                exit (1);
        }
```

```
        /*--- Setup capture control argument ---*/

        memset (&ctrl, 0, sizeof (ctrl));
        ctrl.alf_capture_nbuf = RECV_NBUFS;
        ctrl.alf_capture_bufsize = BLOCKSIZE;
        ctrl.alf_capture_mode = ALF_CAPTURE_CONTINUOUS;
        rval = alf_capture_set_control (h, &ctrl);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture_set_control failed\n");
                exit (1);
        }

        /*
         *      Start
         */

        rval = alf_capture_start (h, recv_callback);
        if (ALF_ISERROR (rval))
        {
                printf ("ERROR: alf_capture start failed\n");
                exit (1);
        }

        printf ("Running\n");

        for (;;)
        {
                alf_buf_t *pb;
                doneflag = FALSE;
                while (!doneflag)
                        ;
                printf ("got a buffer %d\n", pbuf->alf_capture_buf_idx);
                pb = &pbuf->alf_buf;
                checkbuf (pb->alf_buf_data, pb->alf_buf_size);
                printf ("%02X %02X %02X %02X %02X\n",
                        ((unsigned char*) pb->alf_buf_data)[0],
                        ((unsigned char*) pb->alf_buf_data)[1],
                        ((unsigned char*) pb->alf_buf_data)[2],
                        ((unsigned char*) pb->alf_buf_data)[3],
                        ((unsigned char*) pb->alf_buf_data)[4]);
                memset (pb->alf_buf_data, 0, pb->alf_buf_size);
                _cache_copyback (pb->alf_buf_data, pb->alf_buf_size);
                _cache_invalidate (pb->alf_buf_data, pb->alf_buf_size);

        }

}

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

/***************************************************************************
**
**  PRIVATE - recv_callback -
**
**  Description:
**
***************************************************************************/

PRIVATE void recv_callback (int cause, alf_capture_buf_t *pb)
{
        intcount ++;
        if (!doneflag)
        {
                doneflag = TRUE;
                pbuf = pb;
        }
}

/***************************************************************************
**
```

```
**  PRIVATE - checkbuf - check buffer
**
**  Description:
**
****************************************************************************/

PRIVATE void checkbuf (unsigned char *p, int n)
{
        int seed = p[0] & 0x7F;
        int i;
        unsigned char exp;
        int errcount = 0;
#define MAXERRCOUNT    10

        srand (seed);
        for (i = 1; i < BLOCKSIZE; i ++)
        {
                exp = rand () | 0x80;
                if (exp != p[i])
                {
                        errcount ++;
                        if (errcount == MAXERRCOUNT)
                                printf ("Error list terminated\n");
                        else    if (errcount < MAXERRCOUNT)
                                printf ("Error[%d] got %02X exp %02X\n", i, p[i],
exp);
                }
        }
        if (!errcount)
                printf ("No errors\n");
}
```

H. *TriMedia Program Example vdatest.c*

TriMedia program **vdatest.c** is located in **examples\target\ex6**.

```
/***************************************************************************
**
** File:        vdatest.c - simple VDA test program
**
** Copyright © 1995; Alacron Inc.
**
** Description:
**
**      This program performs some simple VDA operations - first
**      directly to the screen, then repeated to an SDRAM buffer
**      and copied using vda_sync()
**
** History:
**
**      30-Aug-99, tjc:         Created
**      15-Oct-99, tjc:         Added -r and -d flags
**
****************************************************************************/

/*----------------------- HEADER FILES ------------------------------------*/

#include <alfast_tm.h>
#include <vda.h>

/*----------------------- PRIVATE CONSTANTS -------------------------------*/

#define DELAY_MSEC              700

/*----------------------- PRIVATE MACROS ----------------------------------*/

/*----------------------- PRIVATE TYPES -----------------------------------*/

/*----------------------- PRIVATE DATA ------------------------------------*/
```

```
/*---------------------- PUBLIC DATA ------------------------------------*/

/*---------------------- PRIVATE ROUTINE REFERENCES ----------------------*/

PRIVATE void dotest (int dev, grp_buf_t *pgrp, int syncflag);
PRIVATE void delay (void);

/*---------------------- PUBLIC ROUTINES --------------------------------*/

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

int main (int argc, char *argv[])
        {
        int i;
        int rval;
        int dev = 0;
        grp_buf_t *pgrp_display;
        grp_buf_t *pgrp_buffer;
        char *resolution_string = NULL;

        for (i = 1; i < argc; i ++)
                if (*argv[i] == '-')
                        switch (*(argv[i]+1))
                        {
                        case 'r':
                                if (++i < argc)
                                        resolution_string = argv[i];
                                break;
                        case 'd':
                                if (++i < argc)
                                        dev = atoi (argv[i]);
                        }

        vda_errorprt (dev, 1);
        vda_open (dev, 0);

        /*--- Set resolution and clear screen if not host controlled ---*/

        if (!vda_host_controlled (dev))
        {
                if (resolution_string != NULL)
                {
                        rval = vda_ioctl (dev, VDA_SET_RESOLUTION,
resolution_string);
                        if (rval)
                                printf ("WARNING: VDA_SET_RESOLUTION failed, using "
                                        "current setup\n");
                }
                vda_cls (dev, VDA_DISPLAY, vda_map_rgb (dev, 100, 100, 100));
        }

        /*--- create grp_buf_t pointing to the screen ---*/

        pgrp_display = vda_grp_vram (dev);

        /*--- alter it to be upper left quarter screen ---*/

        pgrp_display->grp_xres /= 2;
        pgrp_display->grp_yres /= 2;

        /*--- allocate a local graphics buffer ---*/

        pgrp_buffer = grp_alloc (pgrp_display->grp_yres, pgrp_display->grp_xres,
        pgrp_display->grp_hstride);
```

```
        /*--- run test directly to display ---*/

        dotest (dev, pgrp_display, 0);

        /*--- run test to local buffer ---*/

        dotest (dev, pgrp_buffer, 1);

        /*--- Clean up ---*/

        vda_close (dev, 0);
        grp_free (pgrp_display);
        grp_free (pgrp_buffer);

        exit (0);
}

/*****************************************************************************
**
**  PRIVATE - dotest - perform test
**
**  Description:
**
*****************************************************************************/

PRIVATE void dotest (int dev, grp_buf_t *pgrp, int syncflag)
{
        grp_font_t *pf;
        int x[4];
        int y[4];
        int ddtab[10];
        int i;

        printf ("cls red\n");
        grp_cls (pgrp, vda_map_rgb (dev, 220, 0, 0));
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);
        delay ();

        printf ("cls green\n");
        grp_cls (pgrp, vda_map_rgb (dev, 0, 220, 0));
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);
        delay ();

        printf ("cls blue\n");
        grp_cls (pgrp, vda_map_rgb (dev, 0, 0, 220));
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);
        delay ();

        printf ("draw lines\n");
        grp_line (pgrp, vda_map_rgb (dev, 255, 255, 255), 0, 0, 49, 49);
        grp_line (pgrp, vda_map_rgb (dev, 255, 255, 255), 0, 49, 49, 0);
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);

        printf ("region copy\n");
        vda_region_copy (dev, VDA_DISPLAY, 0, 0, VDA_DISPLAY, 50, 0, 49, 49);
        vda_region_copy (dev, VDA_DISPLAY, 0, 0, VDA_DISPLAY, 0, 50, 49, 49);
        vda_region_copy (dev, VDA_DISPLAY, 0, 0, VDA_DISPLAY, 50, 50, 49, 49);
        delay ();

        printf ("draw text\n");
        grp_cls (pgrp, vda_map_rgb (dev, 80, 80, 80));

        pf = grp_load_font ("10x20.bdf");
        if (pf == NULL)
        {
                printf ("ERROR: grp_load_font failed\n");
                exit (1);
        }

        grp_draw_string (pf, pgrp, 100, 100, "testing 123",
```

```
                        vda_map_rgb (dev, 0, 255, 255),
                        vda_map_rgb (dev, 0, 0, 0));
        grp_draw_string (pf, pgrp, 100 + grp_string_width (pf, "testing 123"), 100,
                        "testing 123",
                        vda_map_rgb (dev, 0, 255, 255),
                        vda_map_rgb (dev, 80, 80, 80));
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);
        delay ();

        grp_free_font (pf);

        printf ("fill a triangle\n");
        x[0] = 100;
        y[0] = 200;
        x[1] = 200;
        y[1] = 200;
        x[2] = 150;
        y[2] = 100;
        x[3] = 100;
        y[3] = 200;
        grp_cls (pgrp, vda_map_rgb (dev, 80, 80, 80));
        grp_fill (pgrp, vda_map_rgb (dev, 255, 0, 255), x, y, 4);
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);
        delay ();

        printf ("draw dot-dash lines\n");
        ddtab[0] = 10;
        ddtab[1] = 5;
        ddtab[2] = 5;
        ddtab[3] = 5;

        grp_cls (pgrp, vda_map_rgb (dev, 80, 80, 80));
        for (i = 100; i < 200; i += 10)
                grp_dot_dash_line (pgrp, 100, i, 300, i,
                        vda_map_rgb (dev, 255, 255, 255),
                        vda_map_rgb (dev, 0, 0, 0), ddtab, 4);
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);
        delay ();

        printf ("draw diagonal line using points\n");

        grp_cls (pgrp, vda_map_rgb (dev, 80, 80, 80));

        for (i = 100; i < 200; i ++)
        grp_point (pgrp, vda_map_rgb (dev, 220, 0, 200), i, i);
        if (syncflag) vda_sync (dev, pgrp, VDA_DISPLAY);
        delay ();

}


PRIVATE void delay (void)
{
        alf_timing_msec_delay (DELAY_MSEC);
}
```

IV. CAPTURE LIBRARY REFERENCE

The Capture Library functions allow the TriMedia program to capture data via one of the input channels on a FastSeries processor board such as the FastImage. This chapter provides reference information for each of the functions, listed alphabetically.

A. Include Files

The standard include file **alfast_tm.h** for the ALFAST Runtime library may be found in the **include** directory in the ALFAST software directory (typically **\usr\alfast**).

The environment variable **ALFAST** must be set to point to the ALFAST library install directory.

B. Libraries

Environment	Libraries
Windows NT, Windows 95/97	**%ALFAST%\lib\libalfast.a**

C. Quick Reference

The following table lists the functions in the Capture Library

Function	Summary
alf_capture_attach	Initialize capture library
alf_capture_load_profile	Load a capture profile
alf_capture_query_attribute	Query profile attribute
alf_capture_set_attribute	Set profile attribute
alf_capture_set_control	Set capture parameters
alf_capture_start	Start image capture
alf_capture_status	Retrieve capture status
alf_capture_stop	Stop capture
alf_config_query	Retrieve board configuration
alf_palreg_write	Write directly to PAL device JTAG register
alf_timing_start	Initialize a duration timing
alf_timing_elapsed	Compute elapsed time in seconds
alf_timing_msec_elapsed	Compute elapsed time in milliseconds
alf_timing_msec_delay	Delay for milliseconds
alf_timing_usec_delay	Delay for microseconds

D. Returns and Error Codes

A consistent return and error reporting mechanism is provided for this library. Most functions return zero (ALF_NOERROR) to indicate completion without errors, or a negative integer error code for failure. The attach routines return a positive valued handle value, or a negative error code. The “register read” functions return an 8 bit integer value in the range 0 to 255 if successful, or a negative error code if the read failed.

The macro **ALF_ISERROR(*x*)** evaluates to TRUE if ***x*** is an error value, FALSE on ALF_NOERROR.

Error codes are:

ALF_ERROR_ALLOC
ALF_ERROR_BADARG
ALF_ERROR_BADHANDLE
ALF_ERROR_BUSY
ALF_ERROR_CHECKSUM
ALF_ERROR_EEPROM
ALF_ERROR_IIC
ALF_ERROR_NODATA
ALF_ERROR_NOFILE
ALF_ERROR_NOINIT
ALF_ERROR_NOTCAPABLE
ALF_ERROR_NOTIMPLEMENTED
ALF_ERROR_NOTRUNNING
ALF_ERROR_SDE
ALF_ERROR_SSI
ALF_ERROR_SYNC
ALF_ERROR_TIMEOUT

alf_capture_attach

C Usage:

```
#include <alfast_tm.h>

handle alf_capture_attach (int device_instance)
```

Arguments

device_instance	Input device instance number, typically 0

Description:

Initialize capture library.

The **alf_capture_attach** function establishes a connection with the Capture library. The value returned from the attach routine is an instance “handle” (of type **handle**) which is used in subsequent calls to the functions in the library. The attach function may be called multiple times.

When attaching, a *device instance* is specified. The device instance is an integer (0, 1, ..) identifying the specific device instance when there is more than one instance. Device instance value 0 should be passed to the attach routine.

Return Values

handle	Capture library handle for success.
error code	Error in processing command.

Example

```
#include <>
…
handle CapHan;
…
CapHan = alf_capture_attach((int)0);
if (ALF_ISERROR(CapHan)){
       printf “Attach Error”;
       exit;
       }
```

alf_capture_load_profile

C Usage:

```
#include <alfast_tm.h>

int alf_capture_load_profile (handle h, char *filename)
```

Arguments

h	Capture library handle returned by **alf_capture_attach()**
Filename	Name of Capture profile.

Description:

Load capture profile attributes into TriMedia memory. If the processor is TriMedia 0, the processor issues commands to program all the programmable devices on the FastImage or FastFrame board. *Filename* is the name of the profile in PC memory; the capture library searches for this file first in the current directory, then in **%ALFAST%\lib\capture**, where **%ALFAST%** points to the install directory for the runtime libraries. Capture profiles have the following attributes available for reading by the application:

"**INPUT_NCOMP**"	integer number of elements in the pixel (e.g. 3 for RGB, 1 for monochrome)	
"**INPUT_PIXELS_PER_LINE**"	integer	acquired pixels per line
"**INPUT_LINES_PER_BUFFER**"	integer	acquired lines per input buffer
"**INPUT_PIXEL_SIZE**"	integer	bytes per pixel
"**INPUT_NX**"	integer	number of usable pixels in X direction
"**INPUT_NY**"	integer	number of usable pixels in Y direction
"**INPUT_XOFFSET**"	integer	byte offset to first usable pixel
"**INPUT_YOFFSET**"	integer	line offset to first usable line
"**INPUT_SOURCE**"	string	"**RGB Analog**" "**RGB Digital**" "**Monochrome Analog**" "**Monochrome Digital**" "**NTSC Analog**" "**PAL Analog**" "**SECAM Analog**" "**S-Video**"

NOTE: Attributes **INPUT_NX**, **INPUT_NY**, **INPUT_XOFFSET**, and **INPUT_YOFFSET** are not supported in Release 1.4.1 of ALFAST.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_capture_query_attribute

C Usage:

```
#include <alfast_tm.h>

typedef struct {
        union {
                unsigned long attr_lvalue;
                float attr_fvalue;
                char * attr_svalue;
                }attr_value;
        int type;
} alf_attribute_t;
int alf_capture_query_attribute (handle h, char *name,
                alf_attribute_t *p})
```

Arguments

H	Capture library handle returned by **alf_capture_attach()**
name	Quoted string identifying the attribute value to retrieve.
p	Pointer to attribute structure (union).

Description:

Find the value of one of the Capture profile attributes. Refer to **alf_capture_load_profile()** for a listing of the attributes. If the type of the attribute is not known, the program can query element **type**, using tokens **ALF_ATTRIBUTE_LVALUE**, **ALF_ATTRIBUTE_FVALUE**, and **ALF_ATTRIBUTE_SVALUE** for matching. When the type of value is known, defined tokens access the union element with the desired type:

```
#define lvalue  attr_value.attr_lvalue
#define fvalue  attr_value.attr_fvalue
#define svalue  attr_value.attr_svalue
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

Example:

```
// Attach to capture library and Load capture configuration profile
handle CapHan = alf_capture_attach((int)0);
int ret = alf_capture_load_profile(CapHan, "alf_profile_1.cap");
// Query attributes to determine buffer size
alf_attribute_t pixels_line, lines_buffer, pixel_size;
ret = alf_capture_query_attribute(CapHan, "INPUT_PIXELS_PER_LINE",
pixels_line);
ret = alf_capture_query_attribute(CapHan, "INPUT_LINES_PER_BUFFER",
                              &lines_buffer);
ret = alf_capture_query_attribute (CapHan, "INPUT_PIXELS_SIZE",
&pixel_size);
printf ("Buffer size = %d\n", (pixels_line.lvalue *
                              lines_buffer.lvalue *
        pixel_size.lvalue));
```

alf_capture_set_attribute

C Usage:

```
#include <alfast_tm.h>

typedef struct {
        union {
                unsigned long attr_lvalue;
                float attr_fvalue;
                char * attr_svalue;
                }attr_value;
        int type;
} alf_attribute_t;

int alf_capture_set_attribute (handle h, char *name,
                       alf_attribute_t *p})
```

Arguments

h	Capture library handle returned by **alf_capture_attach()**
name	Quoted string identifying the attribute value to set.
p	Pointer to attribute structure.

Description:

Set the value of one of the Capture profile attributes to the value preloaded in the **alf_attribute_t** structure. See **alf_capture_load_profile()** for a listing of attributes. The **type** of attribute is specified using tokens **ALF_ATTRIBUTE_LVALUE**, **ALF_ATTRIBUTE_FVALUE**, and **ALF_ATTRIBUTE_SVALUE**. When the type of value is known, defined tokens access the union element with the desired type:

```
#define lvalue  attr_value.attr_lvalue
#define fvalue  attr_value.attr_fvalue
#define svalue  attr_value.attr_svalue
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

Example:

```
// Attach to capture library
handle CapHan = alf_capture_attach((int)0);
// Load capture configuration profile
int ret = alf_capture_load_profile(CapHan, “alf_profile_1.cap”);
// Query attribute to determine buffer size
alf_attribute_t pixels, new_pixels;
ret = alf_capture_query_attribute (CapHan, “INPUT_PIXELS_PER_LINE”,
&pixels);
&new_pixels->lvalue = (int)((&pixels->lvalue)/2)
new_pixels.type = ALF_ATTRIBUTE_LVALUE;
ret = alf_capture_set_attribute (CapHan, “INPUT_PIXELS_PER_LINE”,
&new_pixels);
```

alf_capture_set_control

C Usage:

```
#include <alfast_tm.h>

int alf_capture_set_control (handle h, alf_capture_control_t *pc)
```

Arguments

H	Capture library handle returned by **alf_capture_attach()**
Pc	Pointer to capture control structure with data to be set up.

Description:

Set up control parameters for the capture operation. The program sets up an **alf_capture_control_t** structure using the following elements:

```
typedef struct {
        int alf_capture_nbuf;  // Number of buffers
        int alf_capture_bufsize;   // Size of buffer, in bytes
        int alf_capture_mode;  // Mode = Logical OR of
                                 // ALF_CAPTURE_CONTINUOUS,
                                 // ALF_CAPTURE_SINGLE,
                                 // ALF_CAPTURE_ODD_FIELD,
                                 // ALF_CAPTURE_EVEN_FIELD,
                             // ALF_CAPTURE_INTERLACE
        } alf_capture_control_t;
```

The values of **alf_capture_nbuf** and **alf_capture_bufsize** and the use of the Odd and Even Field specifiers depend on the INPUT_SOURCE attribute in the Capture profile.

Monochrome Digital, RGB Digital, Monochrome Analog, or RGB Analog input programs can specify the number of buffers to use for the raw data. In single-buffer operation, one buffer will do. For continuous capture, the program needs a minimum of two buffers. The callback routine (see **alf_capture_start**) is called to process the first while the second fills. If the algorithm can keep up complete each buffer before the second is ready, two buffers will do. The program can specify any number of buffers, which are filled and refilled in a round-robin order.

Monochrome Digital, RGB Digital, Monochrome Analog, or RGB Analog data sources should use a negative value (<0) for **alf_capture_bufsize**; this value sets the buffer size to one frame per buffer (the product of Capture profile attributes INPUT_PIXEL_SIZE * INPUT_PIXELS_PER_LINE * LINES_PER_BUFFER) automatically.

Structure element **alf_capture_mode** specifies continuous or single-buffer capture. When the INPUT_SOURCE is Monochrome Digital, RGB Digital, Monochrome Analog, or RGB Analog, only **ALF_CAPTURE_CONTINUOUS** or **ALF_CAPTURE_SINGLE** may be specified as the capture mode **alf_capture_mode**. These input sources are stored in TriMedia memory in “raw” format, that is, one frame per buffer of byte data. The internal organization of the data (pixel size, RGB fields, etc.) depends on the source, and the program must deal with them accordingly.

When INPUT_SOURCE is NTSC Analog, PAL Analog, SECAM Analog, or S-video, **alf_capture_bufsize** is ignored. The buffer size is taken from INPUT_PIXELS_PER_LINE, and INPUT_LINES_PER_BUFFER from the Capture profile. Each component of YUV is stored in a separate buffer (the "planar" format required by the TriMedia). YUV input is always 4:2:2, so U and V are half the length of Y.

When INPUT_SOURCE is NTSC Analog, PAL Analog, SECAM Analog, or S-video, the element **alf_capture_nbuf** indicates the number of *sets* of YUV triple buffers needed by the algorithm. In single-buffer operation, one buffer set will do. For continuous capture, the program needs a minimum of two buffer sets. The callback routine (see **alf_capture_start**) is called to process the first set while the second set fills. If the algorithm can complete each buffer set before the second is ready, two buffer sets will do. The program can use any number of buffer sets, which are filled in a round-robin order.

Structure element **alf_capture_mode** specifies continuous or single-buffer capture. When INPUT_SOURCE is NTSC Analog, PAL Analog, SECAM Analog, or S-video, the **alf_capture_mode** must additionally specify how to handle the interlaced fields in odd and even frames. The value of **alf_capture_mode** becomes the logical OR (using the **|** sign) of **ALF_CAPTURE_CONTINUOUS** (or less commonly

ALF_CAPTURE_SINGLE) with one of the following tokens:

ALF_CAPTURE_ODD_FIELD to capture the odd fields only, one field per buffer.

ALF_CAPTURE_EVEN_FIELD to capture the even fields only, one field per buffer.

ALF_CAPTURE_ODD_FIELD | ALF_CAPTURE_EVEN_FIELD to capture both odd and even fields in alternating buffers.

ALF_CAPTURE_INTERLACE to capture both odd and even fields and re-interlace them in a single, double-length buffer.

If ALF_CAPTURE_INTERLACE is specified, INPUT_LINES_PER_BUFFER should specify the number of lines per field; **alf_capture_set_control** allocates buffers large enough for both fields. For all interlaced field options, the value of INPUT_LINES_PER_BUFFER should be the same as the number of lines per field (odd or even); the ALF_CAPTURE_INTERLACE option automatically creates the frame-length buffer from the value for each field separately.

Example:

```
// Attach to capture library
handle CapHan = alf_capture_attach((int)0);
// Load capture configuration profile
int ret = alf_capture_load_profile(CapHan, "NTSC_60.cap");
alf_capture_control_t CapCon;
CapCon.alf_capture_nbuf = 2;
CapCon.alf_capture_bufsize = -1; // Ignored for NTSC
CapCon.alf_capture_mode = (ALF_CAPTURE_CONTINUOUS |
                               ALF_CAPTURE_INTERLACE);
ret = alf_capture_set_control(CapHan, &CapCon);
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_capture_start

C Usage:

```
#include <alfast_tm.h>

int alf_capture_start (handle h,
               void (*alf_callback (int cause, alf_capture_buf_t *buf))
```

Arguments

h	Capture library handle returned by **alf_capture_attach()**
alf_callback	Callback routine for buffer ready or capture status
cause	Reason for calling callback routine, see below for details.
buf	Pointer to structure of buffer information

Description:

Starts image capture as specified in the Digital Output configuration profile loaded with **alf_capture_load_profile**, and via the **alf_capture_control_t** structure set up with **alf_capture_set_control**.

If continuous capture was set up, capture runs until a call is made to **alf_capture_stop**. If single-buffer capture was set up, a single buffer of data is captured. In both cases, the **alf_capture_start** function returns immediately (the capture operation is asynchronous to processing by the TriMedia).

The application program declares a callback routine like the following:

```
void *alf_callback (int cause, alf_capture_buf_t *buf) {

// Code here to check the cause, handle the buffer

}
```

The callback function is called after each buffer of data has been captured, when continuous capture has been stopped by **alf_capture_stop**, or when an error occurs. The *cause* and *buf* parameters are passed to the callback by the library. The following predefined values for *cause* are available:

ALF_CALLBACK_BUFFER_READY indicates a buffer is ready to process.

ALF_CALLBACK_STOPPED indicates capture stopped with **alf_capture_stop**.

ALF_CALLBACK_ERROR indicated that an error occurred during capture.

When the cause is **ALF_CALLBACK_BUFFER_READY**, the argument ***buf*** points to a structure of type **alf_capture_buf_t** containing information on the buffer that most recently completed.

The **alf_capture_buf_t** structure has the elements:

int **alf_capture_buf_idx**	The buffer index, ranging from 0 to the number of buffers – 1
int **alf_capture_field**	In the case of interlaced frames, this value specifies 0 for odd field, and 1 for even field
Alf_buf_t **alf_capture_bufs**[3]	This is an array of **alf_buf_t** structures, each containing buffer information. (See below.)

For "raw" data transfers (Monochrome and RGB Digital and Analog inputs), only one **alf_buf_t** structure is used (accessed as **alf_capture_bufs[0]**). Predefined macro **alf_buf** allows a simpler form of reference. The macro definition is:

```
#define alf_buf        alf_capture_bufs[0]
```

Now, if *buf* is a pointer to an **alf_capture_buf_t**, the callback routine can access members of the first structure in the **alf_capture_bufs** array of **alf_buf_t** structures with the construct:

```
buf->alf_buf.structure_member
```

Which is equivalent to:

```
buf->alf_capture_bufs[0].structure_member
```

For YUV data transfers (NTSC, PAL, SECAM, and S-video inputs), three buffer structures are used. Use predefined tokens for the **alf_capture_bufs** indexes: **ALF_CAPTURE_BUFY** for Y, **ALF_CAPTURE_BUFU** for U, and **ALF_CAPTURE_BUFV** for V. Predefined macros **alf_bufY**, **alf_bufU**, and **alf_bufV** simplify references to the array of buffer structures:

```
#define alf_bufY       alf_capture_bufs[ALF_CAPTURE_BUFY]
#define alf_bufU       alf_capture_bufs[ALF_CAPTURE_BUFU]
#define alf_bufV       alf_capture_bufs[ALF_CAPTURE_BUFV]
```

Now, the callback routine can access these arrays of buffer structures with a construct such as:

```
buf->alf_bufU.structure_member
```

This construct is equivalent to:

```
buf->alf_capture_bufs[ALF_CAPTURE_BUFU].structure_member
```

The **alf_buf_t** structure contains the parameters for the buffer of data. Each **alf_buf_t** structure has the following members:

void ***alf_buf_data**	Pointer to the cache-aligned data buffer
int **alf_buf_stride**	Row to row stride
int **alf_buf_nrows**	Number of rows in the data buffer
int **alf_buf_ncols**	Number of columns in the data buffer
int **alf_buf_size**	Size in bytes of the data buffer

Continuing the example, the callback routine (and the main program as well; see note below) can access the data in the buffer with a reference such as:

```
buf->alf_bufY.alf_buf_data        // Buffer of Y (luma) data
```

Note: The argument to the callback function ***buf*** points to static data, which remains a valid reference outside of the callback context.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

See Also:

alf_capture_stop

alf_capture_status

C Usage:

```
#include <alfast_tm.h>

int alf_capture_status (handle h, alf_capture_status_t *pstatus)
```

Arguments

h	Digitasl Output library handle returned by **alf_capture_attach()**
pstatus	Pointer to digital output status structure to receive status information.

Description:

Retrieve the status of the current digital output operation, continuous or single-buffer. The TriMedia main program uses this routine to gain access to the capture buffers.

The application allocates an empty **alf_capture_status_t** structure, then calls the **alf_capture_status** function. On return, the structure has information in the following elements:

```
typedef struct {
        int alf_capture_running;  // true if capture is running
        int alf_capture_bufno;// current capture buffer number
        int alf_capture_last_bufno;       // last completed buffer
        alf_capture_buf_t *alf_capture_buflist;
                                  // ptr to array of buffer ptrs
        int alf_capture_error;// error flags
} alf_capture_status_t;
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_capture_stop

C Usage

```
#include <alfast_tm.h>

int alf_capture_stop (handle h)
```

Arguments

h	Capture library handle returned by **alf_capture_attach()**

Description

Stop the current continuous capture operation.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_config_query

C Usage

```
#include <alfast_tm.h>

int alf_config_query (alf_config_t *pconf)
```

Arguments

pconf	Pointer to board configuration structure to receive information.

Description

Retrieve information about the FastSeries board configuration. The **alf_config_t** structure that is written upon successful return from this function has the following elements:

```
typedef struct {
       int alf_config_board_type;    // ALF_CONFIG_FI/FD/FF/F4
       char alf_config_board revision[9];
                                     // 1.2.3.4 (null terminated)
       int alf_config_serial_number[10];// EEPROM s/n (null terminated)
       int alf_config_processor_number;      // Processor number
       int alf_config_processor_type;        // ALF_CONFIG_TM1000/1100/1300
       int alf_config_processor_stepping;// TriMedia processor stepping
       int alf_config_clock_speed;   // integer Mhz
       int alf_config_SDRAM_size;            // integer Mbytes
       int alf_config_PCI_bus;               // PCI bus number
       int alf_config_PCI_devfunc;   // PCI device/function number
       int alf_config_PCI_IRQ;               // PCI IRQ
       int alf_config_EEPROM_processor_number;
                                     // processor number from EEPROM
       unsigned long alf_config_SDRAM_base;  // SDRAM base address
       unsigned long alf_config_MMIO_base;   // MMIO base address
       int alf_config_digital_in;    // Digital input present
       int alf_config_analog_in;     // Analog input present
       int alf_config_digital_out;   // Digital output present
       int alf_config_analog_out;    // Analog output present
} alf_config_t;
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

Example

```
int ret;
alf_config_t config;
ret = alf_config_query(&config);
printf (“Memory size = %d \n”, config.alf_config_SDRAM_size);
```

alf_palreg_write

C Usage

```
#include <alfast_tm.h>

void alf_palreg_write (int palno, int regno, int value,)
```

Arguments

palno	1 = FPGA #1, 2 = FPGA #2
regno	JTAG Register number
value	Value to be written to register

Description

Accesses JTAG register in capture FPGAs (Xilinx devices).

alf_timing_start

C Usage

```
#include <alfast_tm.h>

void alf_timing_start (alf_timing_t *ts)
```

Arguments

ts	Pointer to timing variable to calculate timing information.

Description

Initialize a duration timer. Upon return, the timing variable *ts* set to an initial value.

alf_timing_elapsed

C Usage

```
#include <alfast_tm.h>

float alf_timing_elapsed (alf_timing_t *ts)
```

Arguments

Ts	Pointer to timing variable to calculate timing information.

Description

Returns the current elapsed value in seconds since the most recent call to **alf_timing_start** with this **alf_timing_t** variable.

Return Values

float	Elapsed time in seconds.

Example

```
alf_timing_t timer;
float elapsed_secs;

alf_timing_start (&timer);
...
elapsed_secs = alf_timing_elapsed(&timer);
```

alf_timing_msec_elapsed

C Usage

```
#include <alfast_tm.h>

unsigned long alf_timing_msec_elapsed (alf_timing_t *ts)
```

Arguments

ts	Pointer to timing variable to calculate timing information.

Description

Returns the current elapsed value in milliseconds since the most recent call to **alf_timing_start** with this **alf_timing_t** variable.

Return Values

unsigned long	Elapsed time in milliseconds

Example

```
alf_timing_t timer;
unsigned long elapsed_msecs;

alf_timing_start (&timer);
...
elapsed_msecs = alf_timing_msec_elapsed(&timer);
```

alf_timing_msec_delay

C Usage

```
#include <alfast_tm.h>

void alf_timing_msec_delay (int nmsec)
```

Arguments

nmsec	Number of milliseconds to delay.

Description

Delay (pause the TriMedia program) for the specified value in milliseconds. The function returns after *nmsec* milliseconds have elapsed.

alf_timing_usec_delay

C Usage

```
#include <alfast_tm.h>

void alf_timing_usec_delay (int nusec)
```

Arguments

nusec	Number of microseconds to delay.

Description

Delay (pause the TriMedia program) for the specified value in microseconds. The function returns after *nusec* microseconds have elapsed.

V. DIGITAL OUTPUT LIBRARY REFERENCE

The Digital Output Library functions allow the TriMedia program to output data via the digital output channel on a FastSeries processor board such as the FastImage. This chapter provides reference information for each of the functions, listed alphabetically.

A. Include Files

The standard include file **alfast_tm.h** for the ALFAST Digital Output library may be found in the **include** directory in the ALFAST software directory (typically **\usr\alfast**).

The environment variable **ALFAST** must be set to point to the ALFAST library install directory.

B. Libraries

Environment	Libraries
Windows NT, Windows 95/98	**%ALFAST%\lib\libalfast.a**

C. Quick Reference

The following table lists the functions in the Digital Output Library.

Function	Summary
alf_digout_attach	Initialize digotal output library
alf_digout_load_profile	Load a digital output profile
alf_digout_query_attribute	Query profile attribute
alf_digout_set_attribute	Set profile attribute
alf_digout_set_control	Set digital output parameters
alf_digout_start	Start digital output
alf_digout_status	Retrieve digital output status
alf_digout_stop	Stop digital output

D. Returns and Error Codes

A consistent return and error reporting mechanism is provided for this library. Most functions return zero (ALF_NOERROR) to indicate completion without errors, or a negative integer error code for failure. The attach routines return a positive valued handle value, or a negative error code.

The macro **ALF_ISERROR(*x*)** evaluates to TRUE if ***x*** is an error value, FALSE on ALF_NOERROR.

Error codes are:

ALF_ERROR_ALLOC
ALF_ERROR_BADARG
ALF_ERROR_BADHANDLE
ALF_ERROR_BUSY
ALF_ERROR_CHECKSUM
ALF_ERROR_EEPROM
ALF_ERROR_IIC
ALF_ERROR_NODATA
ALF_ERROR_NOFILE
ALF_ERROR_NOINIT
ALF_ERROR_NOTCAPABLE
ALF_ERROR_NOTIMPLEMENTED
ALF_ERROR_NOTRUNNING
ALF_ERROR_SDE
ALF_ERROR_SSI
ALF_ERROR_SYNC
ALF_ERROR_TIMEOUT

alf_digout_attach

C Usage:

```
#include <alfast_tm.h>

handle alf_digout_attach (int device_instance)
```

Arguments

device_instance	Input device instance number, typically 0

Description:

Initialize digital output library.

The **alf_digout_attach** function establishes a connection with the Digital Output library. The value returned from the attach routine is an instance "handle" (of type handle) which is used in subsequent calls to the functions in the library. The attach function may be called multiple times.

When attaching, a *device instance* is specified. The device instance is an integer (0, 1, ..) identifying the specific device instance when there is more than one instance. Device instance value 0 should be passed to the attach routine.

Return Values

handle	Digital Output library handle for success.
error code	Error in processing command.

Example

```
#include <alfast_tm.h>
…
handle DigOutHan;
…
DigOutHan = alf_digout_attach(0);
if (ALF_ISERROR(DigOutHan)){
        printf "Attach Error";
        exit;
        }
```

alf_digout_enable

C Usage:

```
#include <alfast_tm.h>

int alf_digout_enable (handle h)
```

Arguments

H	Digital Output library handle returned by **alf_digout_attach()**

Description:

The Digital Output Hardware has a hardware output driver enable line. This line is initialized from the Digital Output configuration profile using the **DDS_OUTPUT** entry. The application program on TriMedia 0 can maintain direct control using **alf_digout_enable** and **alf_digout_disable**.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_digout_disable

C Usage:

```
#include <alfast_tm.h>

int alf_digout_disable (handle h)
```

Arguments

H	Digital Output library handle returned by **alf_digout_attach()**

Description:

The Digital Output Hardware has a hardware output driver enable line. This line is initialized from the Digital Output configuration profile using the **DDS_OUTPUT** entry. The application program on TriMedia 0 can maintain direct control using **alf_digout_enable** and **alf_digout_disable**.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_digout_load_profile

C Usage:

```
#include <alfast_tm.h>

int alf_digout_load_profile (handle h, char *filename)
```

Arguments

h	Digital Output library handle returned by **alf_digout_attach()**
filename	Name of Digital Output configuration profile.

Description:

Load digital output profile attributes into TriMedia memory. Filename is the name of the profile in PC memory; the digital output library searches for this file first in the current directory, then in **%ALFAST%\lib\digout**, where **%ALFAST%** points to the install directory for the runtime libraries.

Digital Output configuration profiles have the following attributes available for reading by the application:

DDS_OUTPUT	1 = output enabled, 0 = disabled
DDS_FREQ	Frequency (MHz) of internally-generated clock.
DIGOUT_MODE	RAW or YUV
NROWS	Number of rows in the output (RAW = 1).
NCOLS	Number of columns in output (RAW = buffersize)
INTPRI	Interrupt priority
YUV_MODE	TriMedia-specific YUV mode
YUV_VIDEO_STD	NTSC, PAL, or NONE
IMAGE_VERT_OFFSET	Vertical offset in pixels of buffer within window
IMAGE_HORZ_OFFSET	Horizontal offset in pixels of buffer within window

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_digout_query_attribute

C Usage:

```
#include <alfast_tm.h>

typedef struct {
        union {
                unsigned long lvalue;
                float fvalue;
                char *svalue;
                }attr_value;
        int type;
} alf_attribute_t;

int alf_digout_query_attribute (handle h, char *name,
                alf_attribute_t *p})
```

Arguments

h	Digital output library handle returned by **alf_digout_attach()**
name	Quoted string identifying the attribute value to retrieve.
p	Pointer to attribute structure.

Description:

Find the value of one of the Digital Output configuration profile attributes. Refer to **alf_digout_load_profile()** for a listing of the attributes. If the type of the attribute is not known, the program can query the structure element **type**, using the predefined tokens **ALF_ATTRIBUTE_LVALUE**, **ALF_ATTRIBUTE_FVALUE**, and **ALF_ATTRIBUTE_SVALUE** for matching. When the type of value is known, defined tokens access the union element with the desired type:

```
#define lvalue  attr_value.attr_lvalue
#define fvalue  attr_value.attr_fvalue
#define svalue  attr_value.attr_svalue
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

Example:

```
// Attach to digital output library
handle DigOutHan = alf_digout_attach((int)0);
// Load digital output configuration profile
int ret = alf_digout_load_profile(DigOutHan, "alf_profile_2.cap");
// Query attributes to determine output mode
alf_attribute_t mode;
ret = alf_digout_query_attribute (DigOutHan, "DIGOUT_MODE", &mode);
printf ("Output mode = %s\n", mode.svalue);
```

alf_digout_set_attribute

C Usage:

```
#include <alfast_tm.h>

typedef struct {
        union {
                unsigned long lvalue;
                float fvalue;
                char *svalue;
                }attr_value;
                int type;
} alf_attribute_t;

int alf_digout_set_attribute (handle h, char *name,
                alf_attribute_t *p})
```

Arguments

H	Digital output library handle returned by **alf_digout_attach()**
name	Quoted string identifying the attribute value to set.
P	Pointer to attribute structure.

Description:

Set the value of one of the Digital Output configuration profile attributes to the value preloaded in the **alf_attribute_t** structure. Refer to **alf_digout_load_profile()** for a listing of the attributes. The type of the attribute is specified by setting the structure element **type**, using the predefined tokens **ALF_ATTRIBUTE_LVALUE**, **ALF_ATTRIBUTE_FVALUE**, and **ALF_ATTRIBUTE_SVALUE**. When the type of value is known, defined tokens access the union element with the desired type:

```
#define lvalue  attr_value.attr_lvalue
#define fvalue  attr_value.attr_fvalue
#define svalue  attr_value.attr_svalue
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

Example:

```
// Attach to Digital Output library
handle DigOutHan = alf_digout_attach((int)0);
// Load Digital Output configuration profile
int ret = alf_digout_load_profile(DigOutHan, "alf_profile_2.cap");
// Set DIGOUT_MODE attribute to "raw".
alf_attribute_t mode;
mode.svalue = "raw";
mode.type = ALF_ATTRIBUTE_SVALUE;
ret = alf_digout_set_attribute (DigOutHan, "DIGOUT_MODE",
&mode);
```

alf_digout_set_control

C Usage:

```
#include <alfast_tm.h>

typedef struct {
                int alf_digout_nbuf;       // Number of buffers
                int alf_digout_mode;       // Mode = Logical OR of
                                           // ALF_DIGOUT_CONTINUOUS,
                                           // ALF_DIGOUT_SINGLE,
                } alf_digout_control_t;

int alf_digout_set_control (handle h, alf_digout_control_t *pc)
```

Arguments

H	Digital Output library handle returned by **alf_digout_attach()**
Pc	Pointer to digital output control structure with data to be set up.

Description:

Set up control parameters for the capture operation in the .**alf_digout_control_t** structure. Element **nbufs** is the number of buffers desired. Structure element **alf_digout_mode** specifies continuous or single-buffer output.

When DIGOUT_MODE is RAW,.a single component is allocated per buffer. For YUV output data, three components per buffer are allocated. Since the YUV data is always 4:2:2, the U and V components are half the size of the Y component.

Example:

```
// Attach to digital output library
handle DigOutHan = alf_digout_attach(0);

// Load digital output configuration profile
int ret;
ret = alf_digout_load_profile(DigOutHan, "example2.dig");

//Set digital output control
alf_digout_control_t DigOutCon;
DigOutCon.alf_digout_nbuf = 3;
DigOutCon.alf_digout_mode = ALF_DIGOUT_CONTINUOUS;
ret = alf_digout_set_control(DigOutHan, &DigOutCon);
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_digout_start

C Usage:

```
#include <alfast_tm.h>

int alf_digout_start (handle h,
               void (*alf_callback (int cause, alf_digout_buf_t *buf))
```

Arguments

H	Digital Output library handle returned by **alf_digout_attach()**
alf_callback	Callback routine for buffer or transfer status
cause	Reason for calling callback routine, see below for details.
buf	Pointer to structure with completed buffer(s)

Description:

Starts image digital output as specified in the Digital Output configuration profile loaded with **alf_digout_load_profile**, and via the **alf_digout_control_t** structure set up with **alf_digout_set_control**. If continuous transfer was set up, digital output runs until a call is made to **alf_digout_stop**. If single-buffer transfer was set up, a single buffer of data is output. In both cases, the **alf_digout_start** function returns immediately (the digital output operation is asynchronous to processing by the TriMedia).

The application program declares callback routine **alf_callback** like the following:

```
void *alf_dout_callback (int cause, alf_digout_buf_t *pbuf) {
// Code here to check the cause, handle the buffer

}
```

The callback is called with the cause set to **ALF_DIGOUT_COMPLETE** and pbuf pointing (via the **alf_digout_buf_t** structure elements) to the buffer just completed. The callback is also invoked just prior to initiating transfer of the next buffer, with the cause set to ALF_DIGOUT_START and the argument ***pbuf*** pointing to the buffer that is being started.

The **alf_digout_buf_t** structure contains the following elements:

int **alf_digout_buf_idx**	The buffer index, ranging from 0 to the number of buffers – 1
int **alf_digout_bufs**	This is an array of **alf_buf_t** structures, each containing specific buffer information. (See below.)

The **alf_buf_t** structure elements specify:

alf_buf_t **alf_buf_data**	Pointer to the cache aligned data buffer
int **alf_buf_stride**	Row to row stride in bytes
int **alf_buf_nrows**	Number of rows in the data buffer
int **alf_buf_ncols**	Number of columns in the data buffer

int **alf_buf_size** Size in bytes of the data buffer

For “raw” data transfers (Monochrome and RGB Digital output), only one **alf_buf_t** structure is used (accessed as **alf_digout_bufs[0]**). Predefined macro **alf_buf** allows a simpler form of reference. The macro definition is:

```
#define alf_digout_buf alf_digout_bufs[0]
```

Now, if *buf* is a pointer to an **alf_digout_buf_t**, the callback routine can access the first structure in the **alf_digout_bufs** array of **alf_buf_t** structures with:

```
buf->alf_digout_buf.structure_member
```

Which is equivalent to:

```
buf->alf_digout_bufs[0].structure_member
```

For YUV data transfers (NTSC, PAL, SECAM, and S-video inputs), three buffer structures are used. Use predefined tokens for the **alf_digout_bufs** indexes: **ALF_DIGOUT_BUFY** for Y, **ALF_DIGOUT_BUFU** for U, and **ALF_DIGOUT_BUFV** for V. Predefined macros **alf_digout_bufY**, **alf_ digout_bufU**, and **alf_ digout_bufV** simplify references to the array of buffer structures:

```
#define alf_digout_bufY  alf_capture_bufs[ALF_CAPTURE_BUFY]
#define alf_digout_bufU  alf_capture_bufs[ALF_CAPTURE_BUFU]
#define alf_digout_bufV  alf_capture_bufs[ALF_CAPTURE_BUFV]
```

Now, the callback routine can access these arrays of buffer structures with a construct such as:

```
buf->alf_digout_bufU.structure_member
```

This construct is equivalent to:

```
buf->alf_digout_bufs[ALF_DIGOUT_BUFU].structure_member
```

Continuing the example, the callback routine (and the main program as well; see note below) can access the data in the buffer with a reference such as:

```
buf->alf_digout_bufY.alf_buf_data        // Buffer of Y (luma) data
```

Thus, the **alf_buf_t** structure contains all the pertinent data about the buffer of data, so the callback need not look any of it up.

Note: The argument to the callback function ***buf*** points to static data, which remains a valid reference outside of the callback context.

Return Values

ALF_NOERROR Success.
negative error code Error in processing command.

See Also:

alf_digout_stop

alf_digout_status

C Usage:

```
#include <alfast_tm.h>

int alf_digout_status (handle h, alf_digout_status_t *pstatus)
```

Arguments

h	Digital Output library handle returned by **alf_digout_attach()**
pstatus	Pointer to digital output status structure to receive status information.

Description:

Retrieve the status of the current digital output operation, continuous or single-buffer.

The application allocates an empty **alf_digout_status_t** structure, then calls the **alf_digout_status** function. On return, the structure has information in the following elements:

```
typedef struct {
        int alf_digout_running;    // true if digital output is running
        int alf_digout_bufno;  // current buffer number
        int alf_digout_last_bufno;// last completed buffer number
        alf_digout_buf_t *alf_capture_buflist;
                                   // ptr to array of buffer ptrs
        int alf_digout_error;  // error flags
} alf_digout_status_t;
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_digout_stop

C Usage:

```
#include <alfast_tm.h>

int alf_digout_stop (handle h)
```

Arguments:

h	Capture library handle returned by **alf_digout_attach()**

Description:

Stop the current continuous digital output operation.

Return Values:

ALF_NOERROR	Success.
negative error code	Error in processing command.

VI. S3 VIDEO STREAMING LIBRARY REFRENCE

The S3 Video Streaming Library functions allow the TriMedia program to display data via the Lpcal Peripheral Bus interface to the S3 chip on a FastSeries processor board such as the FastImage. This chapter provides reference information for each of the functions, listed alphabetically.

A. Include Files

The standard include file **alfast_tm.h** for the ALFAST S3 Video Streaming library may be found in the **include** directory in the ALFAST software directory (typically **\usr\alfast**).

The environment variable **ALFAST** must be set to point to the ALFAST library install directory.

B. Libraries

Environment	Libraries
Windows NT, Windows 95/98	**%ALFAST%\lib\libalfast.a**

C. Quick Reference

The following table lists the functions in the Capture Library

Function	Summary
alf_vstream_attach	Initialize video streaming library
alf_vstream _set_lpb	Set LPB parameters
alf_vstream _set_window	Set window parameters
alf_vstream _start	Start video streaming
alf_vstream _status	Retrieve video streaming status
alf_vstream _stop	Stop video streaming

D. Returns and Error Codes

A consistent return and error reporting mechanism is provided for this library. Most functions return zero (ALF_NOERROR) to indicate completion without errors, or a negative integer error code for failure. The attach routines return a positive valued handle value, or a negative error code.

The macro **ALF_ISERROR(*x*)** evaluates to TRUE if ***x*** is an error value, FALSE on ALF_NOERROR.

Error codes are:

ALF_ERROR_ALLOC
ALF_ERROR_BADARG
ALF_ERROR_BADHANDLE
ALF_ERROR_BUSY
ALF_ERROR_CHECKSUM
ALF_ERROR_EEPROM
ALF_ERROR_IIC
ALF_ERROR_NODATA
ALF_ERROR_NOFILE
ALF_ERROR_NOINIT
ALF_ERROR_NOTCAPABLE
ALF_ERROR_NOTIMPLEMENTED
ALF_ERROR_NOTRUNNING
ALF_ERROR_SDE
ALF_ERROR_SSI
ALF_ERROR_SYNC
ALF_ERROR_TIMEOUT

alf_vstream_attach

C Usage:

```
#include <alfast_tm.h>

handle alf_vstream_attach (int device_instance)
```

Arguments

device_instance	Input device instance number, typically 0

Description:

Initialize S3 Video Streaming library.

The **alf_vstream_attach** function establishes a connection with the S3 Video Streaming library. The value returned from the attach routine is an instance "handle" (of type handle) which is used in subsequent calls to the functions in the library. The attach function may be called multiple times.

When attaching, a *device instance* is specified. The device instance is an integer (0, 1, ..) identifying the specific device instance when there is more than one instance. Device instance value 0 should be passed to the attach routine.

Return Values

handle	S3 Streaming library handle for success.
error code	Error in processing command.

Example

```
#include <alfast_tm.h>
…
handle VstreamHan;
…
VstreamHan = alf_vstream_attach(0);
if (ALF_ISERROR(VstreamHan)){
        printf "Attach Error";
        exit;
        }
```

alf_vstream_set_lpb

C Usage:

```
#include <alfast_tm.h>

int alf_vstream_set_lpb (handle h, alf_vstream_lpb_t *pc)
```

Arguments

H	Video Streaming library handle returned by **alf_vstream_attach()**
pc	Pointer to streaming LPB control structure with data to be set up.

Description:

Set up Local Peripheral Bus (LPB) parameters for the video streaming operation. The **alf_vstream_lpb_t** structure has the following elements:

```
typedef struct {
        int alf_lpb_nx;        // Width of LPB stream in pixels
        int alf_lpb_ny;        // Height of LPB stream in pixels
        int alf_lpb_offset_x;  // Pixels to skip at start of each line
        int alf_lpb_offset_y;  // Lines to skip from the top of field
        int alf_lpb_mode;      // ALF_LPB_YCBCR
                               // ALF_LPB_YUV16
                               // ALF_LPB_KRGB16
                               // ALF_LPB_YUV
                               // ALF_LPB_RGB16
                               // ALF_LPB_RGB24
                               // ALF_XRGB32
        unsigned long alf_     lpb-address;
                               // Force video memory LPB address
} alf_vstream_lpb_t;
```

Set the relevant elements to the values desired, then pass a pointer to the completed structure in the **alf_vstream_set_lpb** function. The video formats for **alf_lpb_mode** are explained in the S3 documentation. The function allocates addresses in video memory for the LPB input buffer from space immediately following that is used for the S3 primary display. If lpb_address is non-zero, this value is used as an offset into video memory, overriding the default address.

Example:

```
// Attach to video streaming library
handle VstreamHan = alf_vstream_attach(0);

alf_vstream_lpb_t VstreamLpb;
// Set alf_vstream_lpb_t elements to input and
// output window parameters (code not shown).
ret = alf_digout_set_control(VstreamHan, &VstreamLpb;
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_vstream_set_window

C Usage:

```
#include <alfast_tm.h>

int alf_vstream_set_window (handle h, alf_vastream_window_t *pc)
```

Arguments

h	Video Streaming library handle returned by **alf_vstream_attach()**
pc	Pointer to streaming window structure with data to be set up.

Description:

Set up window parameters for the video streaming operation.

The **alf_vstream_window_t** structure has the following elements:

```
typedef struct {
       int alf_ss_in_x0;
// Upper left X coordinate of input buffer to be displayed
int alf_ss_in_y0;
// Upper left Y coordinate of input buffer to be displayed
int alf_ss_in_nx; // Width of input buffer to be displayed
int alf_ss_in_ny; // Height of input buffer to be displayed
int alf_ss_out_x0;
// Upper left X coordinate of screen location of SS window
int alf_ss_out_y0;
// Upper left Y coordinate of screen location of SS window
int alf_ss_out_nx; // Width of SS window
int alf_ss_out_ny; // Height of SS window
int alf_lpb_nx; // Number of X pixels in input data stream
int alf_ss_keying;     // ALF_SS_KEY_WINDOW,
                       // ALF_SS_KEY_COLOR,
                       // ALF_SS_KEY_CHROMA
unsigned long alf_ss_keydata[4];
                       // Keying dependent data (TBD)
} alf_vstream_window_t;
```

Set the relevant elements to the values desired, then pass a pointer to the completed structure in the **alf_vstream_set_window** function. This function may be called repeatedly after the streaming starts to dynamically resize and reposition the window in the display area.

NOTE: **ALF_SS_KEY_WINDOW** is the only supported keying mode at ALFAST Release 1.4.1. Parameters **alf_ss_keying** and **alf_ss_keydata** are ignored.

Example:

```
// Attach to video streaming library
handle VstreamHan = alf_vstream_attach(0);

alf_vstream_window_t VstreamWin;
// Set alf_vstream_window_t elements to
// output window parameters (code not shown).
ret = alf_digout_set_window(VstreamHan, &VstreamWin);
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_vstream_start

C Usage:

```
#include <alfast_tm.h>

int alf_vstream_start (handle h)
```

Arguments

h	Video Streaming library handle returned by **alf_vstream_attach()**

Description:

Starts image video streaming with the Local Peripheral Bus (LPB) set up as specified with **alf_vstream_set_lpb**. The window is as specified with **alf_vstream_set_window**.

Video streaming runs until a call is made to **alf_vstream_stop**.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

See Also:

alf_vstream_stop

alf_vstream_status

C Usage:

```
#include <alfast_tm.h>
int alf_vstream_status (handle h, alf_vstream_status_t
*pstatus)
```

Arguments

h	Video streaming library handle returned by **alf_vstream_attach()**
pstatus	Pointer to video streaming status structure to receive status information.

Description:

Retrieve the status of the current video streaming operation.

The application allocates an empty **alf_vstream_status_t** structure, then calls the **alf_vstream_status** function. On return, the structure has information in the following elements:

```
typedef struct {
        int alf_vstream_running;  // true if video streaming is running
        alf_vstream_lpb_t alf_vstream_lpb; // LPB data
        alf_vstream_window_t alf_vstream_window; // Window data
} alf_vstream_status_t;
```

NOTE: The **alf_vstream_status** function is not supported at ALFAST Release 1.4.1.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_vstream_stop

C Usage

```
#include <alfast_tm.h>
int alf_vstream_stop (handle h)
```

Arguments

h	Video Streaming library handle returned by **alf_vstream_attach()**

Description

Stop the current S3 video streaming operation.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

VII. TV OUTPUT LIBRARY REFERENCE

A. Include Files

The standard include file **alfast_tm.h** for the ALFAST TV Output library may be found in the **include** directory in the ALFAST software directory (typically **\usr\alfast**).

The environment variable **ALFAST** must be set to point to the ALFAST library install directory.

B. Libraries

Environment	Libraries
Windows NT, Windows 95/98	**%ALFAST%\lib\libalfast.a**

C. Quick Reference

The following table lists the functions in the TV output Library

Function	Summary
alf_tvout_attach	Initialize TV output library
alf_tvout_control	Set TV output parameters
alf_tvout_new	Allocate TV output control context
alf_tvout _params	Retrieve video streaming status
alf_tvout_rdpalette	Read 256-entry color palette used for 8-bit color LUT
alf_tvout _start	Start video streaming
alf_tvout _stop	Stop video streaming
alf_tvout_wait_eof	Synchronize image data with window refresh
alf_tvout_wrpalette	Write 256-entry color palette used for 8-bit color LUT

D. Returns and Error Codes

A consistent return and error reporting mechanism is provided for this library. Most functions return zero (ALF_NOERROR) to indicate completion without errors, or a negative integer error code for failure. The attach routines return a positive valued handle value, or a negative error code.

The macro **ALF_ISERROR(*x*)** evaluates to TRUE if ***x*** is an error value, FALSE on ALF_NOERROR.

Error codes are:

ALF_ERROR_ALLOC
ALF_ERROR_BADARG
ALF_ERROR_BADHANDLE
ALF_ERROR_BUSY
ALF_ERROR_CHECKSUM
ALF_ERROR_EEPROM
ALF_ERROR_IIC
ALF_ERROR_NODATA

ALF_ERROR_NOFILE
ALF_ERROR_NOINIT
ALF_ERROR_NOTCAPABLE
ALF_ERROR_NOTIMPLEMENTED
ALF_ERROR_NOTRUNNING
ALF_ERROR_SDE
ALF_ERROR_SSI
ALF_ERROR_SYNC
ALF_ERROR_TIMEOUT

alf_tvout_attach

C Usage:

```
#include <alfast_tm.h>
handle alf_tvout_attach (int device_instance)
```

Arguments

device_instance	Input device instance number, typically 0

Description:

Initialize TV output library.

The **alf_tvout_attach** function establishes a connection with the S3 TV output library. The value returned from the attach routine is an instance "handle" (of type handle) which is used in subsequent calls to the functions in the library.

When attaching, a *device instance* is specified. Since the FastSeries boards have one S3 device, instance value 0 should be passed to the attach routine.

Return Values

handle	TV Output library handle for success.
error code	Error in processing command.

Example

```
#include <alfast_tm.h>
…
handle TVHan = alf_tvout_attach(0);
if (ALF_ISERROR(TVHan)){
        printf "Attach Error";
        exit;
        }
```

alf_tvout_control

C Usage:

```
#include <alfast_tm.h>

int alf_tvout_control (handle h, alf_tvout _t *pc)
```

Arguments

H	TV output library handle returned by **alf_tvout_attach()**
Pc	Pointer to filled-in TV output control structure to be set up.

Description:

Set up control context for the TV output operation.

The **alf_tvout_t** structure has the following elements:

```
typedef struct {
        int alf_tvout_mode;    // ALF_TVOUT_NTSC,
                               // ALF_TVOUT_PAL
                               // ALF_TVOUT_NTSCJ (Japan)
        int alf_tvout_output;  // ALF_TVOUT_COMPOSITE
                               // ALF_TVOUT_SVIDEO
        int alf_tvout_flicker;// ALF_TVOUT_DISABLE = disable
                               // 0-6 = lines to filter
        int alf_tvout_bpp;     // Bits per pixel (8, 15, 16, 24, or 32)
        int alf_tvout_buffer_address;
                               // ALF_TVOUT_USE_VGA
                               // ALF_TVOUT_JUST_AFTER_VGA
                               // Integer offset value
        int alf_tvout_stride;
                               // ALF_TVOUT_USE_VGA
                               // ALF_TVOUT_USE_DEFAULT
                               // Integer stride value
} alf_tvout_t;
```

The default values of these elements are set by a prior call to **alf_tvout_new**. The TriMedia program can assign non-default values as desired, then pass a pointer to the completed structure in the **alf_tvout_control** function. The elements are as follows.

alf_tvout_mode

TV Output mode may set to one of three options. **ALF_TVOUT_NTSC** selects 60 HZ 525 interlaced display. **ALF_TVOUT_PAL** selects 50 HZ 625 line interlaced display, and ALF_TVOUT_NTSCJ selects NTSC Japan of 60 HZ, 525 line interlaced display.

alf_tvout_output

This argument selects composite video output (**ALF_TVOUT_COMPOSITE**), or separated CY/CV (**ALF_TVOUT_SVIDEO**).

alf_tvout_flicker

This argument enables flicker filtering. Values from 0 to 6 select varying amounts of flicker filtering. An argument of ALF_TVOUT_DISABLE disables flicker filtering.

alf_tvout_bpp

The TV Output buffer bits per pixel may be selected using this argument. Values of 8, 15, 16, 24, and 32 may be used. If the value **ALF_TVOUT_USE_VGA** is passed in, the current S3 VGA display pixel size is used.

alf_tvout_buffer_address

This argument specifies the offset into Video Memory for the start of the TV Output image buffer. A value of **ALF_TVOUT_USE_VGA** selects the current S3 VGA display address, a value of **ALF_TVOUT_JUST_AFTER_VGA** selects video memory immediately following the S3 VGA display image buffer. An integer value is used as an offset from the start of video memory.

alf_tvout_stride

This argument selects the row stride in bytes for the TV Output image. A value of **ALF_TVOUT_USE_VGA** selects the current S3 VGA stride, a value of **ALF_TVOUT_USE_DEFAULT** selects the stride based on the TV Output mode, which is 640 pixels for all modes.

Example:

```
int ret;
// Attach to TV output library
handle TVHan = alf_tvout_attach(0);
if (ALF_ISERROR(TVHan)){
        printf "tvout_attach error";
        exit;
        }

alf_tvout_t *tv_p;
ret = alf_tvout_new(TVHan, &tv_p);
if (ALF_ISERROR(TVHan)){
        printf "tvout_new error";
        exit;
        }

tv_p->alf_tv_output = ALF_TV_SVIDEO;
ret = alf_tvout_control(TVHan, tv_p);
if (ALF_ISERROR(ret)){
        printf "tvout_control error";
        exit;
        }
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_tvout_new

C Usage:

```
#include <alfast_tm.h>

int alf_tvout_new (handle h, alf_tvout _t **pc)
```

Arguments

H	TV output library handle returned by **alf_tvout_attach()**
Pc	Pointer to address of empty TV output control structure to be set up.

Description:

Allocate control context for the TV output operation. On return, the **alf_tvout_t** structure is set to the following default element values:

```
typedef struct {
        int alf_tvout_mode;    // ALF_TVOUT_NTSC
        int alf_tvout_output;  // ALF_TVOUT_COMPOSITE
        int alf_tvout_flicker; // 2 (lines to filter)
        int alf_tvout_bpp;     // ALF_TVOUT_USE_VGA
        int alf_tvout_buffer_address;     // ALF_TVOUT_USE_VGA
        int alf_tvout_stride; // ALF_TVOUT_USE_VGA
} alf_tvout_t;
```

Allocate a pointer to an **alf_tvout_t** structure, then pass the address of the pointer to the structure in the **alf_tvout_new** function. When the function returns, the structure has received the default values shown above. See **alf_tvout_control** for details.

The **alf_tvout_t** structure allocated by **alf_tvout_new** can be deallocated with a call to the standard library function **free**.

Example:

```
handle TVHan;
int ret;
alf_tvout_t *tv_p;

TVHan = alf_tvout_attach(0)
if(ALF_ISERROR(TVHan)){
        printf("tvout attach failed"\n);
        exit();
        }

ret = alf_tvout_new(TVHan, &tv_p);
if(ALF_ISERROR(ret)){
        printf("tvout new failed"\n);
        exit();
        }
…
ret = free(TVHan);
```

Return Values

ALF_NOERROR   Success.

negative error code	Error in processing command.

alf_tvout_params

C Usage:

```
#include <alfast_tm.h>

int alf_tvout_params (handle h, alf_tvout_params_t *params)
```

Arguments

H	TV output library handle returned by **alf_tvout_attach()**
params	Pointer to TV output parameter structure to receive status information.

Description:

Retrieve the values of the current TV output operation.

The application allocates an empty **alf_tvout_params_t** structure, then calls the **alf_tvout_params** function. On return, the structure has information in the following elements:

```
typedef struct {
        int alf_tvout_nx;      // Number of pixels per display row
        int alf_tvout_ny;      // Number of display rows
        int alf_tvout_hstride; // Number of bytes per pixel
        int alf_tvout_vstride; // Number of bytes between
                               // the start of successive rows
        int alf_tvout_bpp;     // Bits per pixel (8, 15, 16, 24, 32)
        int alf_tvout_buffer_address;
                               // Address of start of TV output
                       // image display.
        } alf_vstream_status_t;
```

NOTE: **alf_tvout_buffer address** is the physical address of the display, not an offset.

Example:

```
handle TVHan;
int ret;
alf_tvout_params_t tv_params;

// alf_tvout_attach, alf_tvout_new, alf_tvout_control, other functions
ret = alf_tvout_params(TVHan, &tv_params);
if(ALF_ISERROR(ret)){
        printf("tvout params failed"\n);
        exit();
        }
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_tvout_rdpalette

C Usage:

```
#include <alfast_tm.h>

int alf_tvout_rdpalette( handle h, int index, vda_rgb_t *pal, int n);
```

Arguments:

h	A TV output handle returned by **alf_tvout_attach**
pal	A pointer to a palette buffer with at least *n* entries.
index	The start index in the palette array to retrieve
n	The number of palette array indices to retrieve

Description:

The **alf_tvout_rdpalette** function allows the application to retrieve all (or a portion) of the selectable palette. The palette is a 256 element array of structures. Each structure has a field for red, green and blue pixel values. The palette type, `vda_rgb_t`, is defined in the include file alfast_tm.h. It has three unsigned char elements, one each for red, blue, and green.

Argument *n* specifies the number of entries in the palette to retrieve. The entries in the palette are read starting at *index*. The RGB specifiers are read into the palette array provided by *pal*.

Return Values:

0	Function successful
-1	Function failed. VDA_E_BADPARAM VDA_E_MODE

Example:

```
handle tv_h = alf_tvout_attach(0);
int iret;
vda_rgb_t pal
int n = 256;          / read 256 entries */
int start_idx = 0;

pal = (vda_rgb_t *) malloc(n * sizeof(struct vda_rgb_t));

iret = alf_tvout_rdpalette (tv_h, start_idx, pal, n);
if (iret) {
        printf("read palette failed\n");
        }
```

alf_tvout_start

C Usage:

```
#include <alfast_tm.h>

int alf_tvout_start (handle h)
```

Arguments

h	TV output library handle returned by **alf_tvout_attach()**

Description:

Starts TV output set up as specified with **alf_tvout_control**.

TV output runs until a call is made to **alf_tvout_stop**.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_tvout_stop

C Usage

```
#include <alfast_tm.h>

int alf_tvout_stop (handle h)
```

Arguments

h	TV output library handle returned by **alf_tvout_attach()**

Description

Stop the current S3 TV output operation.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_tvout_wait_eof

C Usage

```
#include <alfast_tm.h>

int alf_tvout_wait_eof (handle h)
```

Arguments

h	TV output library handle returned by **alf_tvout_attach()**

Description

Pauses program execution until just prior to the beginning of vertical retrace. Used to synchronize screen data updates with the beginning of the screen redraw.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_tvout_wrpalette

C Usage:

```
#include <alfast_tm.h>
int alf_tvout_wrpalette(handle h, int index, vda_rgb_t *pal, int n);
```

Arguments:

h	A handle returned by **alf_tvout_attach**
pal	A pointer to a palette buffer with at least *n* entries.
index	The index in the palette array where modification will start.
n	The number of palette array indices to modify.

Description:

The **alf_tvout_wrpalette** function allows the application to modify all (or a portion) of the selectable palette. The palette is a 256 element array of structures. Each structure has an unsigned char field each for red, green and blue. The palette type, `vda_rgb_t`, is defined in the include file alfast_tm.h.

Argument *n* specifies the number of entries in the palette to modify. The entries in the VDA palette are modified starting at *index*. The RGB specifiers are contained in the array provided by *pal*.

Return Values:

0	Function successful
-1	Function failed.

Example:

```
handle tv_h = alf_tvout_attach(0);
int iret;
vda_rgb_t pal
int n = 256;          / modify 256 entries */
int start_idx = 0;

pal = (vda_rgb_t *) malloc(n * sizeof(struct vda_rgb_t));

/* set-up palette for gradations of red */
for (i=0; i<256; i++) {
        pal[i].vda_red = i;
        pal[i].vda_blue = 0;
        pal[i].vda_green = 0;
        }

iret = alf_tvout_wrpalette (h, start_idx, pal, n);
if (iret) {
        printf("write palette failed\n");
        }
```

VIII. ICP SHARING LIBRARY REFERENCE

The Analog Output library uses the TriMedia Image Co-Processor (ICP) to transfer image data to the VGA display buffer. Since applications may have their own needs to use the ICP, the Philips ICP library code must be shared. A simple API is available with ALFAST 1.2 or later releases to share the ICP.

A. Include Files

The standard include file **alfast_tm.h** for the ALFAST Image Co-Processor (ICP) Sharing library may be found in the **include** directory in the ALFAST software directory (typically **\usr\alfast**).

The environment variable **ALFAST** must be set to point to the ALFAST library install directory.

B. Libraries

Environment	Libraries
Windows NT, Windows 95/98	**%ALFAST%\lib\libalfast.a**

C. Quick Reference

The following table lists the functions in the ICP sharing library

Function	Summary
alf_icp_acquire	Acquire access to ICP
alf_icp_attach	Initialize Image Co-Processor
alf_icp_release	Release access to ICP

D. Returns and Error Codes

A consistent return and error reporting mechanism is provided for this library. Most functions return zero (ALF_NOERROR) to indicate completion without errors, or a negative integer error code for failure. The attach routines return a positive valued handle value, or a negative error code.

The macro **ALF_ISERROR(*x*)** evaluates to TRUE if ***x*** is an error value, FALSE on ALF_NOERROR.

Error codes are:

ALF_ERROR_ALLOC
ALF_ERROR_BADARG
ALF_ERROR_BADHANDLE
ALF_ERROR_BUSY
ALF_ERROR_CHECKSUM
ALF_ERROR_EEPROM
ALF_ERROR_IIC
ALF_ERROR_NODATA
ALF_ERROR_NOFILE
ALF_ERROR_NOINIT
ALF_ERROR_NOTCAPABLE
ALF_ERROR_NOTIMPLEMENTED
ALF_ERROR_NOTRUNNING

ALF_ERROR_SDE
ALF_ERROR_SSI
ALF_ERROR_SYNC
ALF_ERROR_TIMEOUT

alf_icp_acquire

C Usage:

```
#include <alfast_tm.h>

int alf_icp_acquire (handle h, void (*callback)(), int *instance);
```

Arguments

H	TV output library handle returned by **alf_icp_attach()**
Pc	Pointer to filled-in TV output control structure to be set up.

Description:

This function acquires access to the ICP. ALF_NOERROR is returned if the ICP is available, ALF_ERROR_BUSY is returned if the ICP is in use by other code, or the VDA library. The argument ***callback*** is passed in by the application call, and becomes a callback invoked by the ICP interrupt handler upon ICP completion. ***callback*** may be NULL, in which case no callback is made from the interrupt handler. The SDE "instance" variable for the ICP is returned to the caller at the location pointed to by ***instance***.

Example:

```
handle h;
h = alf_icp_attach (0);
if (ALF_IS_ERROR (h))
{
        printf ("ERROR: could not attach ICP\n");
        return -1;
}

void icp_handler (void)
{
./* ICP completion code here */
}

int icpinstance;
int rval;
icpImageColorConversion_t image;

rval = alf_icp_acquire (h, icp_handler, &icpinstance);
if (ALF_IS_ERROR (h))
{
        printf ("ERROR: alf_icp_acquire failed\n");
        return -1;
}

memset (&image, 0, sizeof (image));
image.yBase = yarray;
image.uBase = uarray;
image.vBase = varray;
image.yInputStride = ystride;
image.yvInputStride = uvstride;
image.inputHeight = nr;
image.inputWidth = nc;
image.outputWidth = nc;
image.filterBypass = icpBYPASS;
```

```
image.outputPixelOffset = 0.0
image.inFormat = vdfYUV422Planar;
image.outFormat = vdfRGB24;
image.littleEndian = 1;
image.overlayEnable = 0;
image.bitMaskEnable = 0;
image.outputDestination = icpPCI;

rval = icpCOlorConversion (icpinstance, &image);
if (rval)
{
        printf ("ERROR: icpColorConversion failed\n");
        return -1;
}

while (icpCheckBUSY ())
        ;

alf_icp_release (icphandle);
```

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

alf_icp_attach

C Usage:

```
#include <alfast_tm.h>

handle alf_icp_attach (int device_instance)
```

Arguments

device_instance	Input device instance number, typically 0

Description:

This function opens the performs ICP initialization (calling icpOpen), returning a handle that the application uses to establish access to the ICP. As there is only one ICP device, the argument ***device_instance*** should always be 0.

This function calls icpOpen, icpInstanceSetup, and icpLoadCoeff. icpInstanceSetup is called with interrupt priority 5. icpLoadCoeff is called with the default ICP coefficient tables.

Return Values

handle	ICP sharing library handle for success.
error code	Error in processing command.

Example

```
#include <alfast_tm.h>
…
handle TVHan = alf_tvout_attach(0);
if (ALF_ISERROR(TVHan)){
        printf "Attach Error";
        exit;
        }
```

alf_icp_release

C Usage:

```
#include <alfast_tm.h>

int alf_icp_release (handle h)
```

Arguments

h	ICP sharing library handle returned by **alf_icp_attach()**

Description:

Releases access to the ICP.

Return Values

ALF_NOERROR	Success.
negative error code	Error in processing command.

IX. ANALOG OUTPUT LIBRARY REFERENCE

The VDA-like Analog Output Library functions allow the TriMedia program to display data on the system console. Most functions assume the display is under the control of the Host processor; a few allow the TriMedia to control the display in "native mode". This chapter provides reference information for each of the functions, listed alphabetically.

A. Include Files

The standard include file **vda.h** for the ALFAST Host-Controlled Analog Output Library may be found in the **include** directory in the ALFAST software directory (typically **\usr\alfast**).

The environment variable **ALFAST** must be set to point to the ALFAST library install directory.

B. Libraries

Environment	Libraries
Windows NT, Windows 95/98	**%ALFAST%\lib\libvda.a**

C. Quick Reference

The following table lists the functions in the Host-Controlled Analog Output Library Library

Function	Summary
grp_alloc	Allocate image buffer in SDRAM
grp_buf_width	Return width of buffer data in pixels
grp_char_width	Return width of single character in pixels
grp_cls	Clear image buffer to background color
grp_dot_dash_line	Draw dotted or dashed line in image buffer
grp_draw_buf	Draw characters from array in image buffer
grp_draw_char	Draw single characters in image buffer
grp_draw_string	Draw string of characters in image buffer
grp_fill	Draw filled polygon in image buffer
grp_free	Free image buffer in SDRAM
grp_free_font	Free font image in memory
grp_line	Draw line in image buffer
grp_load_font	Load font image from file
grp_point	Draw point in image buffer
grp_point_addr	Draw point at address in image buffer
grp_rect_fill	Draw filled rectangle in image buffer
grp_string_width	Return width of null-terminated string in pixels
vda_close	Close VDA device access
vda_cls	Clear display to color
vda_errno	Return error value
vda_errorprt	Display errors on console
vda_grp_vram	Allocate image buffer in Video Memory
vda_grp_vram_free	Free image buffer in Video Memory

vda_host_controlled	Return TRUE if display is host-controlled, FALSE if display is TM native controlled.
vda_ioctl(VDA_GET_PARAMS)	Get configuration information
vda_ioctl(VDA_SET_RESOLUTIO N)	Set video resolution (“native” mode)
vda_ioctl(VDA_WAIT_EOF)	Pause program until end-of-field received
vda_ioctl(VDA_WAIT_WRITE)	Pause program until vda_write or vda_write_yuv complete
vda_map_rgb	Get color pixel value in current resolution
vda_open	Open VDA device
vda_read	Read Video Memory
vda_region_copy	Copy rectangular region in Video Memory
vda_sync	Copy SDRAM image buffer into Video Memory
vda_version	Library version
vda_write	Write RGB data to Video Memory
vda_write_yuv	Write YUV data to Video Memory

D. Handling Errors

Error messages from the Host-controlled library resemble the messages returned by the VDA library. All (non-void) functions return zero upon successful completion or -1 upon failure. On failure, an error code may be retrieved using:

int **vda_errno** (int *dev*)

The application may enable the automatic display of any detected errors using:

int **vda_errorprt** (int *dev*)

Each function listed in the reference section provides information on each of the possible errors detected by each routine. Potential errors include the following.

Definition	Explanation
VDA_NO_ERROR	No outstanding error exists
VDA_E_NODEV	Device specifier indicates a device that does not exist
VDA_E_NOTOPEN	Device specifier is not currently opened
VDA_E_OPENED	Device specifier is already opened
VDA_E_DEV	Invalid device specifier provided
VDA_E_CMD	Invalid IO command specified
VDA_E_CANT	The requested command is invalid for this device
VDA_E_BADPARAM	A parameter is invalid for this command
VDA_E_MODE	Invalid request for current configuration
VDA_E_FAILURE	Command failed

grp_alloc

C Usage:

```
#include <vda.h>

grp_buf_t *grp_alloc (int nrows, int ncols, int size);
```

Arguments:

nrows	Number of rows in Image Buffer
ncols	Number of columns in Image Buffer
size	Pixel size of Image Buffer in bytes

Description:

The **grp_alloc** function creates a Graphics Primitive Buffer Descriptor that describes an Image Buffer located in processor board DRAM. The DRAM buffer is allocated off of the processor board memory heap. The mode of the memory buffer allocated is set to uncached. Image Buffers of this type are intended to be used as scratch buffers onto which graphics objects are rendered. When the image is ready for display, the application should call **vda_sync** (or **vda_write**) to display the image.

An Image Buffer is described by a Graphics Primitive Buffer Descriptor, which is contained in a structure of type `grp_buf_t`. On success, this function returns a pointer to a `grp_buf_t, which` contains information that describes the newly created Image Buffer. The `grp_buf_t` pointer may be passed to other Graphics Primitives to modify the Image Buffer with graphics objects. On failure, this function will return `NULL`.

The Image Buffer created has *nrows* rows and *ncols* columns. Its pixel size is specified by the *size* argument.

Return Values

NULL	Function failed due to insufficient free memory
Non-NULL	Pointer to allocated Graphics Primitive Buffer Descriptor

Example:

```
grp_buf_t *gbuf;

gbuf = grp_alloc ( 100, 100, 1 );
if ((int) gbuf == (int) NULL) {
        printf ("grp alloc failed\n");
        }
```

See also:

grp_free

grp_buf_width

C Usage:

```
#include <vda.h>

int grp_buf_width (grp_font_t *pf, char *s, int n);
```

Arguments:

Pf	a Font Descriptor
S	a buffer containing text
N	the size in bytes of buffer *s*

Description:

The **grp_buf_width** function is used to obtain the width (in pixels) of the text specified by argument *s*. The text buffer has a size of *n* bytes.

The size of the text buffer is measured using the font specified by the Font Descriptor *pf*. A Font Descriptor may be obtained from the function **grp_load_font**.

Return Values

Length of text measured in pixels

Example:

```
#define TEXT "ABCD"
#define N      4
int size;
grp_font_t *pf;

pf = grp_load_font("font.fil");
if ((int) pf == (int) NULL) {
       printf ("grp load font failed\n");
       }
size = grp_buf_width (pf, TEXT, N);
```

grp_char_width

C Usage:

```
#include <vda.h>
int grp_char_width (grp_font_t *pf, int c);
```

Arguments:

Pf	a Font Descriptor
C	a character

Description:

The **grp_char_width** function is used to obtain the width (in pixels) of the character specified by argument *c*. Argument *c* is the integer (ASCII) value of the character.

The size of the character is measured using the font specified by the Font Descriptor *pf*. A Font Descriptor may be obtained from the function **grp_load_font**.

Return Values

size of the character measured in pixels

Example:

```
int size;
grp_font_t *pf;

pf = grp_load_font("font.fil");
if ((int) pf == (int) NULL) {
        printf ("grp load font failed\n");
        }
size = grp_buf_width (pf, (int) 'E');
```

grp_cls

C Usage:

```
#include <vda.h>

int grp_cls (grp_buf _t *pb, int color);
```

Arguments:

pb	a Graphics Primitive Buffer Descriptor
color	a color value

Description:

The Image Buffer specified by argument *pb* is initialized to the pixel value specified by argument *color*.

The Graphics Primitive Buffer Descriptor may be obtained from a call to **grp_alloc** or **vda_grp_vram**.

Return Values

0	Function succeeded
-1	Function failed

Example:

```
int iret;
grp_font_t *pf;

pf = grp_load_font("font.fil");
if ((int) pf == (int) NULL) {
        printf ("grp load font failed\n");
        }
iret = grp_cls (pf, (int) 0);
```

grp_dot_dash_line

C Usage:

```
#include <vda.h>
int grp_dot_dash_line (grp_buf_t *pb, int x0, int y0, int x1,
            int y1, int fg, int bg, int ddtab[], int n_ddtab);
```

Arguments:

Pb	a Graphics Primitive Buffer Descriptor
x0	x-coordinate of the first pixel pair
y0	y-coordinate of the first pixel pair
x1	x-coordinate of the second pixel pair
y1	y-coordinate of the second pixel pair
Fg	foreground color
Bg	background color
ddtab	an array of integers which describe a series of dots and dashes
n_ddtab	the element size of ddtab

Description:

The **grp_dot_dash_line** function draws a dot-dashed line from coordinate *x0, y0*, to coordinate *x1, y1* in the display buffer given by *pb*. The argument *ddtab* is an array of *n_ddtab* integer elements. The dot-dashed line is implemented as an alternating series of foreground and background pixels. For example, 5 pixels foreground followed by 15 pixels background, 5 pixels foreground, 15 pixels background, etc,..

The even elements of array *ddtab* specify the number of foreground pixels in the line while the odd elements of *ddtab* specify the number of background pixels. For example, the line above would be expressed by the following series of integers: 5, 15, 5, 15, etc, … In this case, *ddtab* would be initialized as follows:

```
int ddtab[] = { 5, 15, 5, 15, 5, 15, 5, 15};
```

The *pb* argument is a Graphics Primitive Buffer Descriptor, which may be obtained from a call to **grp_alloc** or **vda_grp_vram**.

Return Values

0	Function succeeded
-1	Function failed

Example:

In the example below, a pattern of 5 pixels foreground, 2 pixels background, 2 pixels foreground, 2 pixels background could be specified as:

```
int line [] = {5, 2, 2, 2};
#define NEL(x) (sizeof (x) / sizeof (x[0]))

grp_dot_dash_line (pb, 0, 0, 100, 100, line, NEL(line));
```

Note, that the pixel at coordinates (x1, y1) is not drawn.

grp_draw_buf

C Usage:

```
#include <vda.h>
void grp_draw_buf (grp_font_t *pf, grp_buf_t *pb, int x, int y,
                              char s[], int n, int fg, int bg);
```

Arguments:

pf	a Font Descriptor
pb	a Graphics Primitive Buffer Descriptor
x	x-coordinate of the first pixel pair
y	y-coordinate of the first pixel pair
s	an array of text
n	the element size of *s*
fg	foreground color
bg	background color

Description:

The **grp_draw_buf** function draws *n* characters in the text array *s* to the Image Buffer specified by *pb*. The text is drawn using the font specified by *pf*. Characters are drawn with foreground color *fg* and background color *bg*.

The *pb* argument is a Graphics Primitive Buffer Descriptor, which may be obtained from a call to **grp_alloc** or **vda_grp_vram**. A Font Descriptor, *pf*, may be obtained from the function **grp_load_font**.

Return Values

None

Example:

```
#define LEN    12
grp_font_t *pf;
grp_buf_t *pb;
char string[LEN]="ABCDEFGHIJKL";

pf = grp_load_font("font.fil");
if ((int) pf == (int) NULL) {
        printf ("grp load font failed\n");
        }

pb = grp_alloc ( 512, 512, 1 );
if ((int) pb == (int) NULL) {
        printf ("grp alloc failed\n");
        }

grp_draw_buf (pf, pb, 10, 10, string, LEN, 80, 0);
```

grp_draw_char

C Usage:

```
#include <vda.h>
void grp_draw_char (grp_font_t *pf, grp_buf_t *pb, int x,
                                  int y, int c, int fg, int bg);
```

Arguments:

Pf	a Font Descriptor
Pb	a Graphics Primitive Buffer Descriptor
X	x-coordinate of the first pixel pair
Y	y-coordinate of the first pixel pair
C	a character of text
Fg	foreground color
Bg	background color

Description:

The **grp_draw_char** function draws the character *c* to the Image Buffer specified by *pb*. The text is rendered using the font given by *pf*. The upper left corner of the character will be aligned on the coordinate specified by arguments *x* and *y*. The character is drawn with foreground color *fg* and background color *bg*.

The *pb* argument is a Graphics Primitive Buffer Descriptor that may be obtained from a call to **grp_alloc** or **vda_grp_vram**. A Font Descriptor, *pf*, may be obtained from the function **grp_load_font**.

Return Values

None

Example:

```
grp_font_t *pf;
grp_buf_t *pb;

pf = grp_load_font("font.fil");
if ((int) pf == (int) NULL) {
       printf ("grp load font failed\n");
       }

pb = grp_alloc ( 512, 512, 1 );
if ((int) pb == (int) NULL) {
       printf ("grp alloc failed\n");
       }

grp_draw_char (pf, pb, 100, 100, (int) 'T', 180, 0);
```

grp_draw_string

C Usage:

```
#include <vda.h>
void grp_draw_string (grp_font_t *pf, grp_buf_t *pb, int x,
                                  int y, char *s, int fg, int bg);
```

Arguments:

pf	a Font Descriptor
pb	a Graphics Primitive Buffer Descriptor
x	x-coordinate of the first pixel pair
y	y-coordinate of the first pixel pair
s	an array of text
fg	foreground color
bg	background color

Description:

The **grp_draw_string** function draws the text in the null terminated string *s* to the Image Buffer specified by *pb*. The text is rendered using the font given by *pf*. The upper left corner of the text block will be aligned on the coordinate specified by arguments *x* and *y*. The characters are drawn with foreground color *fg* and background color *bg*.

The *pb* argument is a Graphics Primitive Buffer Descriptor which may be obtained from a call to **grp_alloc** or **vda_grp_vram**. A Font Descriptor, *pf*, may be obtained from the function **grp_load_font**.

Return Values

None

Example:

```
grp_font_t *pf;
grp_buf_t *pb;

pf = grp_load_font("font.fil");
if ((int) pf == (int) NULL) {
        printf ("grp load font failed\n");
        }

pb = grp_alloc ( 512, 512, 1 );
if ((int) pb == (int) NULL) {
        printf ("grp alloc failed\n");
        }

grp_draw_string (pf, pb, 100, 100, "Text to Draw", 180, 20);
```

grp_fill

C Usage:

```
#include <vda.h>
int grp_fill (grp_buf_t *pb, int color, int x[], int y[], int npts);
```

Arguments:

pb	a Graphics Primitive Buffer Descriptor
color	a pixel value specifying a desired color
x	x-coordinate of the first pixel pair
y	y-coordinate of the first pixel pair
npts	number of points

Description:

The **grp_fill** function fills a polygon, in the Image Buffer specified by *pb*, with the pixel value *color*. The polygon is bounded by the lines, which connect its vertices. A list of vertices is provided to **grp_fill** using arguments *x* and *y*.

The vertices of the polygon are expressed as a series of coordinate points (x_i,y_i). The x-coordinate is supplied in array argument x[i] and the y-coordinate is in array argument y[i]. The index i runs from 0 to (*npts* –1)

The *pb* argument is a Graphics Primitive Buffer Descriptor which may be obtained from a call to **grp_alloc** or **vda_grp_vram**. The color argument, *color*, may be obtained from the function **vda_map_rgb**.

Return Values

0	Function succeeded
-1	Function failed

Example:

The following code example fills a triangular area:

```
int x[] = {0, 0, 100};
int y[] = {0, 100, 100};

grp_fill (pgrp, FILL_COLOR, x, y, 3);
```

grp_free

C Usage:

```
#include <vda.h>
void grp_free (grp_buf_t *pb);
```

Arguments:

pb	a Graphics Primitive Buffer Descriptor

Description:

The **grp_free** function frees all allocated memory associated with the Graphics Primitive Buffer Descriptor *pb*. The argument pg was obtained from the function using **grp_alloc**.

> Note, ***grp_free*** must not be called with a display buffer that was not allocated using ***grp_alloc***.

Return Values

None

Example:

```
grp_free (pb);
```

See also:

Grp_alloc

grp_free_font

C Usage:

```
#include <vda.h>
void grp_free_font (grp_font_t *pf);
```

Arguments:

pf a Font Descriptor

Description:

The **grp_free_font** function frees all dynamically allocated memory used by the font *pf*. The buffers used by *pf* were previously allocated by a call to **grp_load_font**.

Return Values

None

Example:

```
grp_free_font (pf);
```

See also:

grp_load_font

grp_line

C Usage:

```
#include <vda.h>
int grp_line (grp_buf_t *pb, int color, int x0, int y0, int x1,
                                                         int y1);
```

Arguments:

Pb	a Graphics Primitive Buffer Descriptor
color	a pixel value denoting a desired color
x0	the x coordinate of the first pixel
y0	the y coordinate of the first pixel
x1	the x coordinate of the second pixel
y1	the y coordinate of the second pixel

Description:

The **grp_line** function draws a solid line in the Image Buffer specified by *pb*. The line is drawn from coordinate (x_0, y_0) to ($x_{1,}, y_{1,}$) using pixel value *color*.

Note, the pixel at endpoint coordinate (x_1, y_1) is not drawn.

Return Values

0	Function succeeded
-1	Function failed

Example:

```
grp_line (pb, 0x80, 0, 0, 100, 100);
```

grp_load_font

C Usage:

```
#include <vda.h>
grp_font_t *grp_load_font (char *filename);
```

Arguments:

filename	a BDF font file

Description:

The **grp_load_font** function allocates and reads the font description data contained in the host file *filename.* If successful, file *filename* is opened and font description information is loaded into a Font Descriptor of type **`grp_font_t`**. A pointer to the newly created Font Descriptor is returned. This pointer may be used as an input argument to all functions, which require a **`grp_font_t`**. The application should store each different font that is to be used in a unique Font Descriptor.

The file *filename* must be an X11 standard formatted BDF font file. BDF format is ASCII and thus is fully portable across various operating systems.

Return Values

NULL	Function failed. Insufficient free memory.
Non-NULL	A pointer to a Font Descriptor

Example:

```
grp_font_t *pf;

pf = grp_load_font ("font.fil");
if ((int) pg == (int) NULL) {
        printf ("load font failed\n");
        }
```

grp_point

C Usage:

```
#include <vda.h>
int grp_point (grp_buf_t *pb, int color, int x, int y);
```

Arguments:

pb	A Graphics Primitive Buffer Descriptor
color	A pixel value specifying a desired color
x	x-coordinate of the first pixel pair
y	y-coordinate of the first pixel pair

Description:

The **grp_point** function draws a single pixel into the Image Buffer specified by *pb*. The pixel is drawn at coordinates *(x,y)* using the pixel value denoted by *color*.

Return Values

0	Function succeeded
-1	Function failed

Example:

```
grp_point (pb, 0x80, 0, 99);
```

grp_point_addr

C Usage:

```
#include <vda.h>
int grp_point_addr (grp_buf_t *pb, unsigned long p, int color);
```

Arguments:

pb	a Graphics Primitive Buffer Descriptor
color	a pixel value specifying a desired color
p	x-coordinate of the first pixel pair

Description:

The **grp_point_addr** draws a single pixel to Image Buffer *pb* at the address given by *addr* using pixel value *color*.

Return Values

0	Function succeeded
-1	Function failed

Example:

```
int iret

/* where ADDRESS is defined as an address in an Image Buffer */
iret = grp_point_addr (pb, 0x99, ADDRESS);
```

grp_rect_fill

C Usage:

```
#include <vda.h>
int grp_rect_fill (grp_buf_t *pb, int color, int x0, int y0,
                                                      int x1, int y1);
```

Arguments:

pb	a Graphics Primitive Buffer Descriptor
color	a pixel value denoting a desired color
x0	the x coordinate of the first pixel
y0	the y coordinate of the first pixel
x1	the x coordinate of the second pixel
y1	the y coordinate of the second pixel

Description:

The **grp_rect_fill** function fills a rectangular region in the Image Buffer specified by *pb*. The rectangle is initialized to the pixel value specified by the argument *color*.

The rectangle is defined by a pair of coordinates (x_0, y_0) and (x_1, y_1). The two points reference opposite corners of the rectangle.

Return Values

Example:

```
grp_rect_file (pb, 0x100, 0, 0, 100, 100);
```

grp_string_width

C Usage:

```
#include <vda.h>
int grp_string_width (grp_font_t *pf, char *s);
```

Arguments:

Pf	a Font Descriptor
S	a pointer to a null-terminated string

Description:

The grp_string_width function returns the width (in pixels) of the null terminated string s. The width is computed using the font specified by the Font Descriptor pf.

Return Values

size of string

Example:

```
int size;
size = grp_rect_file (pf, "Null-Terminated String");
```

vda_close

C Usage:

```
#include <vda.h>
int vda_close ( int dev, int reset );
```

Arguments:

dev	A device number that was passed to vda_open
reset	Ignored by ALFAST

Description:

The **vda_close** routine removes the connection designated by argument *dev*. The *reset* parameter is ignored.

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_FAILURE VDA_E_NOTOPEN

Example:

```
int iret;
iret = vda_close (VDA_DEV_0_0, 1);
if (iret) {
        printf("vda close failed\n");
        }
```

See also:

vda_open

vda_cls

C Usage:

```
#include <vda.h>
int vda_cls (int dev, int which_buf, int color);
```

Arguments:

dev	A device number that was passed to vda_open
which_buf	Must be VDA_DISPLAY to clear the display
color	Integer representing color value to be written.

Description:

The **vda_cls** routine clears the video display to the requested color.

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_FAILURE VDA_E_NOTOPEN

Example:

```
int iret;
iret = vda_cls (dev, VDA_DISPLAY, vda_map_rgb(dev, 100, 100, 100)););
if (iret) {
        printf("vda cls failed\n");
        }
```

vda_errno

C Usage:

```
#include <vda.h>
int vda_errno (int dev, );
```

Arguments:

dev	A device number that was passed to vda_open

Description:

The **vda_errno** routine returns the current error value for the specified device on this processor. Errors are stored for each processor and device combination.

The **vda_errno** function returns -1 on failure - indicating that the device specified is invalid. On success, a positive error value is returned. All errors are defined in vda.h.

Return Values

Positive value	Function successful - error returned
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_DEV VDA_E_FAILURE VDA_E_NODEV

Example:

```
int error;
error = vda_errno (VDA_DEV_0_0);
if (error > 0) {
        printf("error = %d\n",error);
        }
```

vda_errorprt

C Usage:

```
#include <vda.h>
int vda_errorprt ( int dev, int enable_flag );
```

Arguments:

dev	A device number that was passed to vda_open	
enable_flag	0	Disable error detection
	non-zero	Enable error detection

Description:

The **vda_errorpt** routine instructs the VDA library to automatically display any detected errors for the device specified. The invocation of this routine may occur at any time, including prior to **vda_open** and after **vda_close**. In multi-processor systems, each processor thread maintains a separate copy of the error printing state.

> Note: Error message generation should not be enabled unless the host application is prepared to service printed messages.

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_DEV VDA_E_FAILURE VDA_E_NODEV

Example:

```
int iret;
iret = vda_errorport (VDA_DEV_0_0, 1);
if (iret) {
        printf("vda errorport failed\n");
        }
```

vda_grp_vram

C Usage:

```
#include <vda.h>
grp_buf_t *vda_grp_vram (int dev)
```

Arguments:

Dev	A device number that was passed to vda_open

Description:

The ***vda_grp_vram*** function creates a Graphics Primitive Buffer Descriptor which directly references the S3 display memory. This descriptor, of type `grp_buf_t`, may be passed to the Graphics Primitives to modify the currently displayed VDA image.

Return Values

Not Null	A pointer to a Graphics Primitive Buffer Descriptor
NULL	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_DEV VDA_E_BADPARAM VDA_E_NOMEM VDA_E_NOTOPEN

Example:

```
grp_buf_t *gb;
int iret;

gb = vda_grp_vram (VDA_DEV_0_0);
if ((int) gb == (int) NULL) {
        printf("vda grp vram failed\n");
        }
```

vda_grp_vram_free

C Usage:

```
#include <vda.h>
void vda_grp_vram_free (grp_buf_t *pb)
```

Arguments:

Pb	A Graphics Primitive Buffer Descriptor

Description:

The ***vda_grp_vram_free*** function frees a Graphics Primitive Buffer Descriptor previously allocated using ***vda_grp_vram.***

Return Values

None

Example:

```
vda_grp_vram_free (gb);
```

See also:

vda_grp_vram

vda_host_controlled

C Usage:

```
#include <vda.h>
int vda_host_controlled (int dev)
```

Arguments:

dev	VDA device opened with vda_open

Description:

The ***vda_host_controlled*** function returns TRUE when the system VDA controller has been opened and is being controlled from the Host (O/S) level, or FALSE when the VDA device can be controlled by the TriMedia in “native” mode.

Return Values

int	TRUE (1) or FALSE (0) for success VDA_E_NOTOPEN if dev is not opened

Example:

```
vda_grp_host_controlled (dev);
```

See also:

vda_open
vda_ioctl(VDA_SET_RESOLUTION)

vda_ioctl(VDA_GET_PARAMS)

C Usage:

```
#include <vda.h>
int vda_ioctl(int dev, VDA_GET_PARAMS, vda_params_t *params);
```

Arguments:

dev	A device number that was passed to **vda_open**
params	A pointer to a VDA parameter buffer

Description:

The **VDA_GET_PARAMS** command allows the application to retrieve configuration information about the device specified by *dev*. When the function returns, the application can read the **vda_params_t** elements:

```
typedef struct {
        int device;    // VDA device for this structure (always 0)
        int xres;      // Number of pixels per horizontal line
        int yres;      // Number of lines per display
        int pixel_size;        // Number of bits per pixel (8, 15, 16,
        or 24)
        int hstride;   // Number of bytes per pixel
        int vstride;   // Byte offset from one line to the next
        unsigned long buffer_address;
                       // Address of the display buffer.  This address
                       // may be accessed by the TriMedia processor.
} vda_params_t;
```

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are specific to this **vda_ioctl** function: VDA_E_BADPARAM

Example:

```
int iret;
vda_params_t *params;

params = (vda_params_t *) malloc (sizeof(vda_params_t));
iret = vda_ioctl (VDA_DEV_0_0, VDA_GET_PARAMS, params);
if (iret) {
        printf("vda set resolution failed\n");
        }
```

vda_ioctl(VDA_SET_RESOLUTION)

C Usage:

```
#include <vda.h>
int vda_ioctl( int dev, VDA_SET_RESOLUTION, char *resolution_string);
```

Arguments:

Dev	A device number that was passed to **vda_open**
resolution_string	String specifying resolution, pixel size, frame rate

Description:

this command sets the display resolution, pixel size, and frame rate to those specified.

Resolution-string has the format:

*xres**x**yresxbpp_framerate*

For example, the string "800x600x16_72" sets the display to 800 x 600 pixels, 16-bits per pixel, and a frame rate of 72HZ. The command can set up any arbitrary resolution.

The default resolution upon **vda_open** is 800 x 600 pixels, 16-bits per pixel, and a frame rate of 72HZ.

> **NOTE:**
> 1. **VDA_SET_RESOLUTION** is valid only when the S3 display is NOT Host-controlled.
> 2. Not all monitors are capable of displaying high frame or line rates.

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code.

Example:

```
int iret;
iret = vda_ioctl (dev, VDA_SET_RESOLUTION, "800x600x16_72");
if (iret) {
        printf("vda set resolution failed\n");
        }
```

See also:

vda_host_controlled

vda_ioctl(VDA_WAIT_EOF)

C Usage:

```
#include <vda.h>
int vda_ioctl( int dev, VDA_WAIT_EOF)
```

Arguments:

dev	A device number that was passed to **vda_open**

Description:

When called, this command will pend until the end-of-field indication is detected.

NOTE: **VDA_WAIT_EOF** is not supported at ALFAST .

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code.

Example:

```
int iret;
iret = vda_ioctl (dev, VDA_WAIT_EOF);
if (iret) {
        printf("vda wait eof failed\n");
        }
```

vda_ioctl(VDA_WAIT_WRITE)

C Usage:

```
#include <vda.h>
int vda_ioctl( int dev, VDA_WAIT_WRITE)
```

Arguments:

Dev	A device number that was passed to **vda_open**

Description:

When called, this command will pend until the previous call to **vda_write** or **vda_write_yuv** completes.

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code.

Example:

```
int iret;
iret = vda_ioctl (dev, VDA_WAIT_WRITE);
if (iret) {
        printf("vda wait write failed\n");
        }
```

vda_map_rgb

C Usage:

```
#include <vda.h>
int vda_map_rgb (int dev, int r, int g, int b)
```

Arguments:

dev	A device number that was passed to **vda_open**
r	pixel red value (0-255 decimal)
g	pixel green value
b	pixel blue value

Description:

The **vda_map_rgb** function returns a pixel value that most closely matches the color specified by the *r*, *g*, and *b* arguments. If one of the 8 bit modes is selected, the current palette is scanned for the closest match. For other pixel formats, the pixel value is computed directly from RGB.

Return Values

Pixel value

Example:

```
int ret;
int dev = 0;
int pixel_value;

ret = vda_open (dev, 0);

pixel_value = vda_map_rgb (dev, 5, 150, 255);
```

vda_open

C Usage:

```
#include <vda.h>
int vda_open ( int dev, int initialize );
```

Arguments:

dev	Zero for the on-board S3 VDA, 1 or higher for additional display types.
initialize	Ignored

Description:

The **vda_open** function will determine if the device referred to by *dev* is installed and operating. If *dev* is present, it will setup a connection to the device on behalf of the application. Argument *dev* specifies a device; device 0 is the on-board S3 VDA chip. See VDA Initialization File below for details on specifying another display.

vda_open must be invoked prior to attempting to access other VDA functions, the only exception being **vda_errorprt**.

The *initialize* parameter is ignored by the ALFAST runtime software.

VDA Initialization File

The Host-controlled Analog Input library accesses the Fast Series S3 device, but can also access other PCI memory mapped VGA display adapters. When accessing a host controlled S3 device on the Fast Series board, the VDA library gets all its information by directly reading S3 registers. When another VGA adapter is used, this information must be provided as entries in an ini-file. The device argument to **vda_open** indexes an ini-file **%ALFAST%\lib\vda.ini** to obtain information on the device. See the later section vda.ini File Format for a sample **vda.ini** file:

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_DEV VDA_E_NODEV VDA_E_OPENED VDA_E_NOTINITTED VDA_E_FAILURE

Example:

```
int iret;
iret = vda_open (VDA_DEV_0_0, 1);
if (iret) {
        printf("vda open failed\n");
        }
```

vda_read

C Usage:

```
#include <vda.h>
int vda_read ( int dev, int which_buf, int conversion, int x,
               int y, void *buf, int vstride, int nr, int nc );
```

Arguments:

dev	The constant passed to **vda_open** that is used to specify the device.
which_buf	A constant VDA_DISPLAY.
conversion	A constant that specifies the data conversion mode: VDA_COPY.
x	The x-coordinate of the region to be read
y	The y-coordinate of the region to be read
buf	The target buffer
vstride	The row length of the image in bytes
nr	The number of pixels per row
nc	The number of pixels per column

Description:

The **vda_read** command allows an application to retrieve a section of Video Memory. The region returned starts at the pixel offset (*x, y*) into the drawable buffer and encompasses the region covered from (*x, y*) through (*x*+*nc-1, y*+*nr-1*) inclusive. When invoked by a C application, the data is stored in the buffer *buf* in row major order. The (*x,y*) tuple is based upon a two-dimensional display array starting with (0,0) in the upper left corner of the displayable region. The *vstride* argument specifies the number of bytes between the first points in each row of the image to be copied.

The *which_buf* parameter indicates from where the reads are performed. The only value is VDA_DISPLAY to read from the S3 display memory.

The *conversion* value must be VDA_COPY (data move without conversion).

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_NOTOPEN VDA_E_BADPARAM VDA_E_ MODE VDA_E_FAILURE

Example:

```
int iret;
int which_buf;
int conversion;
int x, y;
char buf;
int vstride;
int nr;
int nc;

/ The following code fragment, reads a 100 by 100 submatrix
 * (starting at offset 10,10) of a 256 by 256 image */
x = 10;
y = 10;
nr = 100;
nc = 100;
vstride = 256;

buf = (char *) malloc (nr * nc * sizeof(char));
conversion = VDA_COPY;
which_buf = VDA_BUF0;
iret = vda_read (VDA_DEV_0_0, which_buf, conversion, x, y, buf,
vstride, nr, nc);
if (iret) {
        printf("vda read failed\n");
        }
```

vda_region_copy

C Usage:

```
#include <vda.h>
int vda_region_copy ( int dev, int src_buf, int sx, int sy, int
       vstride, int dst_buf, int dx, int dy, int nr, int nc );
```

Arguments:

dev	The constant passed to **vda_open** that is used to specify the device.
src_buf	A constant VDA_DISPLAY for S3 display memory
sx	The x-coordinate for the source buffer
sy	The y-coordinate for the source buffer
vstride	The vertical stride in bytes
dst_buf	A constant VDA_DISPLAY for S3 display memory
dx	The x-coordinate for the destination buffer
dy	The y-coordinate for the destination buffer
nr	Number of columns (row length) in pixels
nc	Number of rows in pixels

Description:

The **vda_region_copy** function is provided to enable applications to copy rectangular regions of data between the Video Memory buffers specified by *src_buf* and *dst_buf*. The source and destination buffer identifiers are both VDA_DISPLAY.

> When copying between overlapping buffers in the S3 display memory, the value for sy must be greater than the value for dy. Otherwise data may be lost.

The *dev* parameter identifies a particular VDA device. The *sx, sy* parameters are the starting x and y coordinates for the source buffer. Argument *vstride* is the stride between vertical rows. The *dx, dy* parameters are the starting x and y coordinates for the destination buffer, and *nr, nc* represent the number of pixels per row and the number of rows to copy respectively.

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_NOTOPEN VDA_E_BADPARAM VDA_E_FAILURE

Example:

```
int iret;

iret = vda_region_copy (VDA_DEV_0_0, VDA_DISPLAY, 256, 256, 1024,
VDA_DISPLAY, 0, 0, 1024, 1024);
if (iret) {
        printf("vda region copy failed\n");
        }
```

vda_sync

C Usage:

```
#include <vda.h>
int vda_sync (int dev, grp_buf_t *pb, int which_buf)
```

Arguments:

dev	The constant passed to **vda_open** that is used to specify the device.
Pb	A Graphics Primitive Buffer Descriptor in DRAM
which_buf	The constant VDA_DISPLAY to write the DRAM image into S3 display memory

Description:

The **vda_sync** function writes the DRAM GPB Buffer specified by argument *pb* to the VDA device specified by *dev*. The GPB must have been allocated with a call to **grp_alloc**. The destination of the write is the constant VDA_DISPLAY for the S3 display memory. The movement of image data is executed by using the **vda_write** library function.

The pixel size of the VDA resident image must match the pixel size of the Image Buffer specified by *pb*. Pixel format conversions are not supported.

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_DEV VDA_E_NOTOPEN VDA_E_BADPARAM VDA_E_NOMEM

Example:

```
int iret;
grp_buf_t *pb = grp_alloc(100, 100, 1);
// Commands here to load graphics into the GRP.

iret = vda_sync (dev, pb, VDA_DISPLAY);
if (iret) {
        printf("vda sync failed\n");
        }
```

vda_version

C Usage:

```
extern char vda_version [];
```

Description:

The VDA_version string will contain the information pertaining to the build revision and date of the product build. The format of the string is as follows:

VDA: Version m.n of dd-mmm-yy hh:mm

Where *m.n* is the software major and minor revision numbers, *dd-mmm-yy hh:mm* is the day, month, year and time of the build.

Note: vda_version is a global string variable that may be referenced externally by a user application. It is not a function.

Return Values

none

Example:

```
extern char vda_version{};
printf ("VDA version: %s\n",(char *)vda_version);
```

vda_write

C Usage:

```
#include <vda.h>
int vda_write ( int dev, int which_buf, int conversion, void *buf,
                int vstride, int x, int y, int nr, int nc );
```

Arguments:

dev	The constant passed to **vda_open** that is used to specify the device.
which_buf	A constant VDA_DISPLAY to write to S3 display memory.
conversion	A constant that specifies the data conversion mode: VDA_COPY, VDA_GRAY8. See description below.
x	The x-coordinate of the region to be written
y	The y-coordinate of the region to be written
buf	The source buffer
vstride	The row length of the image in bytes
nr	The number of pixels per row
nc	The number of pixels per column

Description:

The **vda_write** command allows an application to overwrite a section of Video Memory. The region written starts at the pixel offset (*x, y*) into the VDA target buffer and encompasses the region covered from (*x, y*) through (*x+nc-1, y+nr-1*) inclusive. The data is retrieved from buffer *buf* in row major order, when invoked by a C application. The (*x,y*) tuple is based upon a two-dimensional display array starting with (0,0) in the upper left corner of the VDA target image. The vstride argument specifies the number of bytes between the first points in each row of the source image.

The *which_buf* parameter indicates the destination of the write. The only value is VDA_DISPLAY to write to S3 display memory.

The *conversion* value may be one of:

Define	Description
VDA_COPY	Data move without conversion
VDA_GRAY8	8-bit grayscale to color conversion (uses ICP)

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_NOTOPEN VDA_E_BADPARAM VDA_E_ MODE VDA_E_FAILURE

Example:

```
int iret;
int which_buf;
int conversion;
int x;
int y;
char buf;
int vstride;
int nr;
int nc;

/ The following code fragment, reads a 100 by 100 submatrix
 * (starting at offset 10,10) of a 256 by 256 image /
x = 10;
y = 10;
nr = 100;
nc = 100;
vstride = 256;

/ assume that buf is initialized with data that will be written */
buf = (char *) malloc (nr * nc * sizeof(char));

conversion = VDA_COPY;
which_buf = VDA_DISPLAY;
iret = vda_read (VDA_DEV_0_0, which_buf, conversion, x, y, buf,
vstride, nr, nc);
if (iret) {
        printf("vda write failed\n");
        }
```

vda_write_yuv

C Usage:

```
#include <vda.h>
int vda_writeyuv (int dev, int which_buf, int conversion,
        void *ybuf, int iy, void *ubuf, int iu, void *vbuf, int iv,
              int x0, int y0, int nr, int nc );
```

Arguments:

dev	The VDA device passed to **vda_open**.
which_buf	A constant VDA_DISPLAY to write to S3 display memory.
conversion	A constant data conversion mode: VDA_COPY.
ybuf	Pointer to buffer of Y-data
iy	Row stride for Y
ubuf	Pointer to buffer of U-data
Iuy	Row stride for U
vbuf	Pointer to buffer of V-data
iv	Row stride for V
x0	The x-coordinate of the Video Memory to be written
y0	The y-coordinate of the Video Memory to be written
nr	The number of rows to write
nc	The number of columns to write

Description:

The **vda_write_yuv** command allows an application to overwrite a section of Video Memory with data from a YUV buffer. The region written starts at the pixel offset (*x0, y0*) into the VDA target buffer and encompasses the region covered from (*x0, y0*) through (*x0+nc-1, y0+nr-1*) inclusive. The data is retrieved from the three buffers *ybuf*, *ubuf*, and *vbuf* in row major order, when invoked by a C application. The (*x,y*) tuple is based upon a two-dimensional display array starting with (0,0) in the upper left corner of the VDA target image. The iy, iu, and iv arguments specify the number of bytes between the first points in each row of the source components.

The *which_buf* parameter indicates the destination of the write. The only allowed value is **VDA_DISPLAY** to write to S3 display memory.

The *conversion* value must be **VDA_COPY** (no conversion).

Return Values

0	Function successful
-1	Function failed. Call **vda_errno** to obtain an error code. The following error codes are returned: VDA_E_NOTOPEN VDA_E_BADPARAM VDA_E_ MODE VDA_E_FAILURE

Example:

In this example, the arguments for the YUV write are the ones set in the **alf_capture_buf_t** structure returned to the capture callback routine.

```
int rval;

rval = vda_write_yuv (vda_dev, VDA_DISPLAY, VDA_COPY,
        input_buf->alf_bufY.alf_buf_data,
        input_buf->alf_bufY.alf_buf_stride,
        input_buf->alf_bufU.alf_buf_data,
        input_buf->alf_bufU.alf_buf_stride,
        input_buf->alf_bufV.alf_buf_data,
        input_buf->alf_bufV.alf_buf_stride,
        display_xoffset,
        display_yoffset,
        input_buf->alf_bufY.alf_buf_nrows,
        input_buf->alf_bufY.alf_buf_ncols);
if (rval)
{
        printf ("ERROR: vda_write_yuv failed - %d\n", rval);
        exit (1);
}
```

X. CAPTURE PROFILE FORMAT

Capture configuration information is stored in a text file. The file format closely follows the *ini-file* style of Windows 3.x.

A. Example Capture Profile

The following is a sample Capture Profile:

```
#---------------------------------------------------------------------------
#
#       Capture - Specifications for capture
#
#---------------------------------------------------------------------------

[Pals]
!if FastImage
PAL_ICUBE_2 = string "7111clk.bin"
PAL_ICUBE_1 = string "7111xpt.bin"
PAL_XILINX_1 = string "analogPAL1.bin"
PAL_XILINX_2 = string "analogPAL2.bin"
!endif
!if FastFrame
PAL_ICUBE_2 = string "FFntsCLK.bin"
PAL_XILINX_1 = string "FFntsc.bin"
!endif
[Capture]
INPUT_NCOMP = int 1
INPUT_PIXELS_PER_LINE = int 704
INPUT_LINES_PER_BUFFER = int 240
INPUT_PIXEL_SIZE = int 1
INPUT_NX = int 352
INPUT_NY = int 240
INPUT_XOFFSET = int 0
INPUT_YOFFSET = int 0
INPUT_SOURCE = string "NTSC Analog"
NUMBER_OF_BUFFERS = int 2
MULTI_TAP = int 0

#
#       ADC clamp source
#

ADC_CLAMP_1 = int 1
ADC_CLAMP_2 = int 2
ADC_CLAMP_3 = int 3

#
#       VIDEO INPUT
#
#       VIMODE can be raw or yuv
#

VIMODE = string yuv

#
#       VIRAW_mode - Video input raw modes
#
#       2       viStTREAM8
#       3       viSTREAM10s
#       4       viSTREAM10U
#       5       viMESSAGE
#

VIRAW_mode = int 2

#
```

```
#       YUV setup variables
#

VIYUV_thresholdReachedEnable = int 0
VIYUV_cositedSampling = int 1

#
#       VIYUV_mode - Video input YUV modes
#
#       0       viFULLRES
#       1       viHALFRES
#

VIYUV_mode = int 0
VIYUV_yThreshold = int 0
VIYUV_startX = int 4
VIYUV_startY = int 11

#
#       Video input interrupt priority
#

interruptPriority = int 0

#
#       Programmable clocks
#

PROG_CLK1 = float 10.0e6
PROG_CLK2 = float 10.0e6

#
#       Slow Timer values
#

ST1C1 = int 0
ST1C2 = int 0
ST1T = int 0                  # 0 - ext trigger, 1 - free running
ST2C1 = int 0
ST2C2 = int 0
ST2T = int 0                  # 0 - ext trigger, 1 - free running

#
#       S/W Sync
#

SWSYNC = int 0

#
#       External Trigger using GPOUT
#
#       0       pulse high
#       1       pulse low
#

EXT_TRIGGER = int 0
EXT_TRIGGER_MASK = int 1

#---------------------------------------------------------------------------
#
#       MAX521 -
#
#       values specified in volts, operation is unipolar
#
#---------------------------------------------------------------------------

[MAX521]
ENABLE = int 0
VREF = float 4.096
ADC_A_TOP = float 2.9
ADC_A_BOT = float 0.9
```

```
ADC_B_TOP = float 2.9
ADC_B_BOT = float 0.9
ADC_C_TOP = float 2.9
ADC_C_BOT = float 0.9
ADC_CLAMP = float 0.9

#---------------------------------------------------------------------------
#
#       SAA7111 -
#
#       The following SAA7111A locations are read-only
#
#       SA00 - Chip version
#       SA1A - Text slicer status
#       SA1B - Decoded text slicer bytes
#       SA1C - Decoded text slicer bytes
#       SA1F - Status byte
#
#---------------------------------------------------------------------------

[SAA7111]
ENABLE = int 1
# use MODE = 4 for S-video
SA02_MODE = int 0
SA02_GUDL = int 3
SA02_FUSE = int 3
SA03_GAI = int 0x10040000
SA03_GAFIX = int 0
SA03_HOLDG = int 0
SA03_WPOFF = int 0
SA03_VBSL = int 1
SA03_HLNRS = int 0
SA06_HSB = int 0
SA07_HSS = int 0
SA08_VNOI = int 0
SA08_HPLL = int 0
SA08_VTRC = int 1
SA08_EXFIL = int 0
SA08_FSEL = int 0
SA08_AUFD = int 1
SA09_APER = int 0
SA09_UPTCV = int 0
SA09_VBLB = int 0
SA09_BPSS = int 1
SA09_PREF = int 0
SA09_BYPS = int 0
SA0A_BRIG = int 128
SA0B_CONT = int 0x47
SA0C_SATN = int 0x40
SA0D_HUEC = int 0
# Information taken from SDE vitest
#
#      two decriptions are given, the first for 60 Hz,
#      the second for 50 Hz
#
# Chroma Control       NTSC M, PAL BGHI (0x01)
SA0E_CHBW = int 1
SA0E_FCTC = int 0
SA0E_DCCF = int 0
SA0E_CSTD = int 0
SA0E_CDTO = int 0
# Chroma Control       PAL 4.43, NTSC 4.43 (0x11)
#SA0E_CHBW = int 1
#SA0E_FCTC = int 0
#SA0E_DCCF = int 0
#SA0E_CSTD = int 1
#SA0E_CDTO = int 0

# Chroma Control       NTSC 4.43, PAL N (0x21)
#SA0E_CHBW = int 1
#SA0E_FCTC = int 0
```

```
#SA0E_DCCF = int 0
#SA0E_CSTD = int 2
#SA0E_CDTO = int 0

# Chroma Control       NTSC N, PAL M (0x31)
#SA0E_CHBW = int 1
#SA0E_FCTC = int 0
#SA0E_DCCF = int 0
#SA0E_CSTD = int 3
#SA0E_CDTO = int 0

# Chroma Control       PAL 4.43, SECAM (0x51)
#SA0E_CHBW = int 1
#SA0E_FCTC = int 0
#SA0E_DCCF = int 0
#SA0E_CSTD = int 1
#SA0E_CDTO = int 1

SA10_YDEL = int 0
SA10_VRLN = int 1
SA10_HDEL = int 0
SA10_OFTS = int 3
SA11_COLO = int 0
SA11_VIPB = int 0
SA11_OEHV = int 1
SA11_OEYC = int 1
SA11_COMPO = int 0
SA11_FECO = int 0
SA11_CM99 = int 0
SA11_GPSW = int 1
SA12_AOSL = int 0
SA12_DIT = int 0
SA12_RGB888 = int 0
SA12_CBR = int 0
SA12_TCLO = int 0
SA12_RTSE = int 2
SA13_BCLO = int 0
SA13_BCHI = int 0
SA13_CCTR = int 0
SA13_VCTR = int 0
SA15_VSTA = int 0
SA16_VSTO = int 0

#---------------------------------------------------------------------------
#
#       BT261 - Brooktree BT261 line lock controller
#
#       we have three of these
#
#       The following BT261 locations are read-only
#
#       CR05    capture strobe
#       CR12    Reset lock loss status bit
#       SA07    status register
#
#---------------------------------------------------------------------------

[BT261_1]
ENABLE = int 0
CR0_horizontal_counter_control = int 1
CR0_sync_detect_select = int 3
CR0_clock_input_select = int 0
CR1_interlaced = int 1
CR1_CLOCK_output_disable = int 0
CR1_CSYNC_output_disable = int 0
CR1_VSYNC_output_disable = int 0
CR1_HSYNC_output_disable = int 0
CR1_phase_comparator_input = int 1
CR1_phase_limit_enable = int 0
CR2_phase_lock_pixel_count = int 1
CR2_pixel_clock_mask_enable = int 1
```

```
CR2_lock_override = int 1
CR2_pixel_clock_select = int 0
CR3_phase_lock_line_count = int 32
SA04_VSYNC_sample = int 100
SA05_OSC_COUNT_LOW = int 6
SA06_OSC_COUNT_HIGH = int 6
SA08_HSYNC_start = int 56
SA08_HSYNC_stop = int 0
SA0C_CLAMP_start = int 96
SA0E_CLAMP_stop = int 64
SA10_ZERO_start = int 113
SA12_ZERO_stop = int 771
SA14_FIELD_start = int 188
SA16_FIELD_stop = int 563
SA18_noise_gate_start = int 88
SA1A_noise_gate_stop = int 772
SA1C_HCOUNT = int 793
[BT261_2]
ENABLE = int 0
CR0_horizontal_counter_control = int 1
CR0_sync_detect_select = int 2
CR0_clock_input_select = int 0
CR1_interlaced = int 1
CR1_CLOCK_output_disable = int 0
CR1_CSYNC_output_disable = int 0
CR1_VSYNC_output_disable = int 0
CR1_HSYNC_output_disable = int 0
CR1_phase_comparator_input = int 1
CR1_phase_limit_enable = int 0
CR2_phase_lock_pixel_count = int 1
CR2_pixel_clock_mask_enable = int 1
CR2_lock_override = int 1
CR2_pixel_clock_select = int 0
CR3_phase_lock_line_count = int 32
SA04_VSYNC_sample = int 100
SA05_OSC_COUNT_LOW = int 6
SA06_OSC_COUNT_HIGH = int 6
SA08_HSYNC_start = int 56
SA08_HSYNC_stop = int 0
SA0C_CLAMP_start = int 96
SA0E_CLAMP_stop = int 64
SA10_ZERO_start = int 113
SA12_ZERO_stop = int 771
SA14_FIELD_start = int 188
SA16_FIELD_stop = int 563
SA18_noise_gate_start = int 88
SA1A_noise_gate_stop = int 772
SA1C_HCOUNT = int 793
[BT261_3]
ENABLE = int 0
CR0_horizontal_counter_control = int 1
CR0_sync_detect_select = int 2
CR0_clock_input_select = int 0
CR1_interlaced = int 1
CR1_CLOCK_output_disable = int 0
CR1_CSYNC_output_disable = int 0
CR1_VSYNC_output_disable = int 0
CR1_HSYNC_output_disable = int 0
CR1_phase_comparator_input = int 1
CR1_phase_limit_enable = int 0
CR2_phase_lock_pixel_count = int 1
CR2_pixel_clock_mask_enable = int 1
CR2_lock_override = int 1
CR2_pixel_clock_select = int 0
CR3_phase_lock_line_count = int 32
SA04_VSYNC_sample = int 100
SA05_OSC_COUNT_LOW = int 6
SA06_OSC_COUNT_HIGH = int 6
SA08_HSYNC_start = int 56
SA08_HSYNC_stop = int 0
SA0C_CLAMP_start = int 96
```

```
SA0E_CLAMP_stop = int 64
SA10_ZERO_start = int 113
SA12_ZERO_stop = int 771
SA14_FIELD_start = int 188
SA16_FIELD_stop = int 563
SA18_noise_gate_start = int 88
SA1A_noise_gate_stop = int 772
SA1C_HCOUNT = int 793
```

B. Section Names

The file is separated into sections with headers of the format:

[*section name***]**

C. Input Line Format

Input lines are defined for each kind of section and have the following format (":=" means "is defined as...".

input-line := attribute **=** type value

attribute:= simple string

type := **integer** | **string**

value := number | simple string | quoted string

number := decimal number | **0x**hex number // Examples: 0, -1, 0xBEEF

simple string := non-quoted string that contains no space characters.

quoted string := double-quoted (") string that might contain spaces.

D. Conditional Input Lines

The Capture profile can have input lines for one or more FastSeries board types. Each set of input lines is enclosed in a conditional with the syntax:

!if variable

// Input lines to apply when variable is TRUE

[**!else**

// Input lines to apply when variable is FALSE]

!endif

These conditionals enable and disable the processing of the enclosed input lines. The entry *variable* is one of the following fixed values:

0 Always FALSE

1 Always TRUE

FastImage TRUE if the TM processor is on a FastImage board

FastFrame TRUE if the TM processor is on a FasfFrame board

Fast4 TRUE if the TM processor is on a Fast4 board

FastIO TRUE if the TM processor is on a FastIO board

E. Include Files

Nested inclusion of files may be accomplished through the use of the **!include** directive. The syntax is:

!include *file*

Where *file* specifies the file to be included. When attempting to open *file*, the library will first examine the current directory, followed by the directory containing the file issuing the **!include** directive.

F. Comments

Any line beginning with a pound-sign (#) is a comment (not interpreted as an input line).

XI. DIGITAL OUTPUT PRIFILE FORMAT

Here is a sample Digital Output profile. The format is the same as the Capture Profile.

```
[Pals]
!if FastImage
PAL_ICUBE_1 = string "vo2vixpt.bin"
PAL_ICUBE_2 = string "vo2viclk.bin"
!endif
!if FastFrame
PAL_ICUBE_2 = string "FFvoCLK.bin"
PAL_XILINX_1 = string "FFvo2vi.bin"
!endif
[Digout]

#
#       DDS_FREQ        output frequency
#

DDS_OUTPUT = int 1
DDS_FREQ = float 27.0e6

#
#       DIGOUT_MODE = yuv | raw
#

DIGOUT_MODE = string yuv

#
#       These values are set, regardless of DIGOUT_MODE
#

NROWS = int 400
NCOLS = int 512
INTPRI = int 3

#
#       These values are set if DIGOUT_MODE is yuv
#

# YUV_MODE =   0       vo422_COSITED_UNSCALED
#                              1       vo422_INTERSPERSED_UNSCALED
#                              2       vo420_UNSCALED
#                              4       vo422_COSITED_SCALED
#                              5       vo422_INTERSPERSED_SCALED
#                              6       vo420_SCALED
#

YUV_MODE = int 0
# YUV_VIDEO_STD = ntsc | pal | none
YUV_VIDEO_STD = string ntsc

#
#       The next section is only used if Video Standard is not ntsc or pal. In
#       those two cases, these values are preset.
#

FRAME_PRESET =  int 0
FIELD2_START =  int 0
FRAME_LENGTH =  int 0
F2OLAP =  int 0
F1OLAP =  int 0
F1VIDEO_LINE =  int 0
F2VIDEO_LINE =  int 0
VIDEO_PIXEL_START =  int 0
FRAME_WIDTH =  int 0
FREQUENCY =  int 0
LITTLE_ENDIAN =  int 0
HBE_IEN =  int 0
```

```
URUN_IEN =  int 0
RUN_IEN =  int 0
PLL_S =  int 0
PLL_T =  int 0
CLKOUT =  int 0
SYNC_MASTER =  int 0

#
#       Image offset
#

IMAGE_VERT_OFFSET = int 56
IMAGE_HORZ_OFFSET = int 192

#
#       Overlay - if enabled
#

OVERLAY_ENABLED = int 0
OVERLAY_STARTX = int 0
OVERLAY_STARTY = int 0
OVERLAY_HEIGHT = int 0
OVERLAY_WIDTH = int 0
OVERLAY_ALPHA0 = int 0
OVERLAY_ALPHA1 = int 0

#
#       ChromaKeying (evo, TM1100 or later)
#

CHROMA_KEY_ENABLE = int 0
CHROMA_KEY_KEYY = int 0
CHROMA_KEY_KEYU = int 0
CHROMA_KEY_KEYV = int 0
CHROMA_KEY_MASKY = int 0
CHROMA_KEY_MASKUV = int 0

#
#       ClipSetup (evo, TM1100 or later)
#

CLIP_ENABLE = int 0
CLIP_HIGHCLIPUV = int 0
CLIP_LOWCLIPUV = int 0
CLIP_HIGHCLIPY = int 0
CLIP_LOWCLIPY = int 0

#
#       GenLockSetup (evo, TM1100 or later)
#

GENLOCK_ENABLE = int 0
GENLOCK_SLAVE_DELAY = int 0
```

XII. VDA INITIALIZATION FILE

The Analog Output library supports access to the Fast Series local S3 device, and in addition provides a mechanism for accessing other PCI memory-mapped VGA display adapters. When accessing a Host-controlled S3 device on the Fast Series board, the Analog Output library is able to determine all required information by directly reading S3 registers. When another VGA adapter is being used, this information must be provided as entries in an ini-file. When opening a VDA device using **vda_open**, the *device* argument accesses an ini-file (**%ALFAST%\lib\vda.ini**), which contains information on accessing the device.

A. Sample vda.ini File

The following is a sample **vda.ini** file:

```
#-----------------------------------------------------------------
#
#       vda.ini - VDA library configuration file
#
#       This file contains information about VDA devices.  Each
#       device is described by a section [VDAn].
#
#-----------------------------------------------------------------

!if 1
[VDA0]
type = string s3
locate = string localbus

#--- Monitor section: change as needed
h_retrace = float 4.0e-6            # minimum H retrace
v_retrace = float 500e-6            # minimum V retrace
h_sync_polarity = int 1                     # 1 = +
v_sync_polarity = int 1                     # 1 = +
h_sync_width = float 1.2e-6         # H sync width
v_sync_width = float 52.0e-6  # V sync width

!endif
!if 0
[VDA0]

#       setup for 15 bit 32k color mode
control = string host
locate = string fixed
buffer_address = int 0xEC000000
xres = int 1024
yres = int 768
hstride = int 2
vstride = int 2048
pixel_size = int 16
!endif



!if 0
[VDA1]

#       setup for 24 bit 16M color
control = string host
locate = string fixed
buffer_address = int 0xd4000000
xres = int 1024
yres = int 768
hstride = int 3
vstride = int 3072
pixel_size = int 24
!endif
```

B. Conditional Input Lines

The **vda.ini** file has input lines for one or more VDA devices. Each set of input lines is enclosed in a conditional with the syntax:

!if *variable*

// Input lines to apply when *variable* is TRUE

[**!else**

// Input lines to apply when *variable* is FALSE]

!endif

These conditionals enable and disable the processing of the enclosed ini-file input lines. The entry *variable* is one of the following fixed values:

0	Always FALSE
1	Always TRUE
FastImage	TRUE if the TM processor is on a FastImage board
FastFrame	TRUE if the TM processor is on a FasfFrame board
Fast4	TRUE if the TM processor is on a Fast4 board
FastIO	TRUE if the TM processor is on a FastIO board

Only one VDA device can be enabled at a time, so only one conditional *variable* can be TRUE. The default is to use the local S3 as the system display. Edit the **vda.ini** file and set TRUE the conditional enclosing the device input lines you want to use. Set the other devices FALSE.

C. Include Files

Nested inclusion of files may be accomplished through the use of the **!include** directive. The syntax is:

!include *file*

Where *file* specifies the file to be included. When attempting to open *file*, the library will first examine the current directory, followed by the directory containing the file issuing the **!include** directive.

D. Format of Input Lines

Input lines in the **vda.ini** file have the general format:

parameter **=** *type value*

Available *parameters* depend on whether the device is the local S3 or another device. *Type* can be **string**, **int**, or **float**. *Value* corresponds to the type.

E. Local S3 on FastSeries Board

In the example **vda.ini** file listed earlier in this chapter, the default VDA device 0 (the [VDA0] section) is the S3 on the Fast Series board. The following are valid input lines for the local S3.

Parameter **type** is **string s3** (equivalent to **"s3"**).

Parameter **locate** parameter is **“localbus”**, indicating that it is an S3 located on the same Fast Series board as the TM processor.

The **localbus** section defines monitor parameters for the S3 device:

h_retrace	Minimum horizontal retrace (sec)
v_retrace	Minimum vertical retrace (sec)
h_sync_polarity	Horizontal sync polarity (1 is +, 0 is -)
v_sync_polarity	Vertical sync polarity (1 is +, 0 is -)
h_sync_width	Horizontal sync width (sec)
v_sync_width	Vertical sync width (sec)

Within the **localbus** section, the attribute ***control*** is ignored (see Other Display Adapters below). The determination of whether the S3 device is under host control (used as the system display) is made at runtime.

F. Other Display Controller

VDA device 1 (the [VDA1] section) represents an alternate display type.

- Parameter **control** is **“host”**, indicating that the Host is to control the device. This line is ignored by the library, but may remain useful as a comment.
- Parameter **type** is not specified. The determination of display adapter type is made at runtime.
- Parameter **locate** is set to **“fixed”**, which tells the VDA library that the actual parameters to access the device are provided in subsequent ini-file entries.
- Parameter **buffer_address** is the PCI address of the display buffer on the VGA adapter.
- Parameter **xres** is the number of pixels per line.
- Parameter **yres** is the number of lines.
- Parameter **hstride** is the number of bytes per pixel.
- Parameter **vstride** is the line to line stride in bytes.
- Parameter **pixel_size** is the pixel size in bits (8, 15, 16, 24, 32).

XIII. TROUBLESHOOTING

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.

XIV. 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