Sun Microsystems, Inc.

NICE

TAAC-1 Application Accelerator Technical Notes

Architecture &
Application Development

Contents

Note: Please send comments or questions about this document to:

Application Accelerator Marketing
Sun Microsystems, Inc.
P.O Box 13447
Research Triangle Park, N.C. 27709-3447

1.0 Development of the TAAC-1 Processor

1.1 Design Criteria for the TAAC-1 Processor

The TAAC-1 is an applications accelerator. A highly programmable device, it is designed to make an entire class of applications run faster, not just to speed up a few selected functions. It offers full color display capability, but for tasks that require more processing complexity than a traditional image processor or graphics processor. It also has the floating-point performance of an array processor, but with more flexibility. Thus, it is more optimized (and offers more performance) than a general purpose CPU, but is not so highly optimized that all flexibility is lost.

The TAAC-1 was designed for applications involving spatial or geometric data sets. These applications include high-quality graphics, image processing, and analysis. The design of the TAAC-1 architecture took into account the processing requirements of these applications.

1) Large amounts of data are processed.
2) The data is often structured in 2-D or 3-D arrays.
3) Both integer and floating point is needed.
4) Both vector and scalar processing is needed.
5) Direct display of results is often needed.
6) The processing algorithms are constantly changing so ease of programming is important.
7) Interactive processing is required.

These considerations led to the following features of the TAAC-1:

1) A large amount of memory (8 MByte)
2) Enhanced access to 2D and 3D arrays of memory
3) Integer and floating-point processors
4) Vector and random access to memory
5) Display capability from memory
6) Separate instruction memory
7) Low latency, non-pipelined design
8) Memory-mapped into VME bus
9) C language compiler

In developing a product with these goals, a constant awareness of available technology and market requirements were blended as well. In the case of the TAAC-1, two new technologies were crucial.

Processor technology - Several firms have recently introduced very high performance 32-bit processors that are extensions of the bit-slice (or horizontal) architecture. These families feature high speed (100 nsec or less cycle time), floating-point as well as integer parts, large register files, and multiple I/O ports. The TAAC-1 uses the Texas Instruments 88XX family of processors. This family was selected after hand simulating execution of key algorithms.

The 88XX family has numerous attractive features.

- It is an extension of a known 8-bit family (74AS888,74AS890)
- Its power consumption is low compared to alternatives.
- TI has several IC technologies available, with some parts already moving from AS to CMOS.
- The 88XX performance is excellent and it has a high data throughput.
- The parts are not internally pipelined and have a horizontal as opposed to integrated structure .

Memory technology - The TAAC-1 uses two types of memory: video RAM and static RAM. Video RAM

chips are dual-ported dynamic RAMs, which have very high serial access rates on the second port. They are typically used to provide video rate data for display refresh. The second port can also serve as a very high bandwidth port for vector processing. That is, arrays of floating-point numbers can be accessed for number crunching as well as pixels for display. The memory was divided into two banks with 400 Mbytes/sec vector access to each.

The second element of memory technology involved high-speed static RAMs (SRAMs) for instruction storage. The TAAC-1 is designed with a 16K x 200 bit instruction memory using 50 SRAM chips. The SRAMs made possible a very-long-instruction-word (VLIW) architecture. With it, multiple functional units are controlled by each word. There are 26 different fields within the 200-bit-wide instruction.

The combination of VLIW architecture with non-pipelined low latency parts provides very-high scalar performance, and the high-memory bandwidth yields high vector performance.

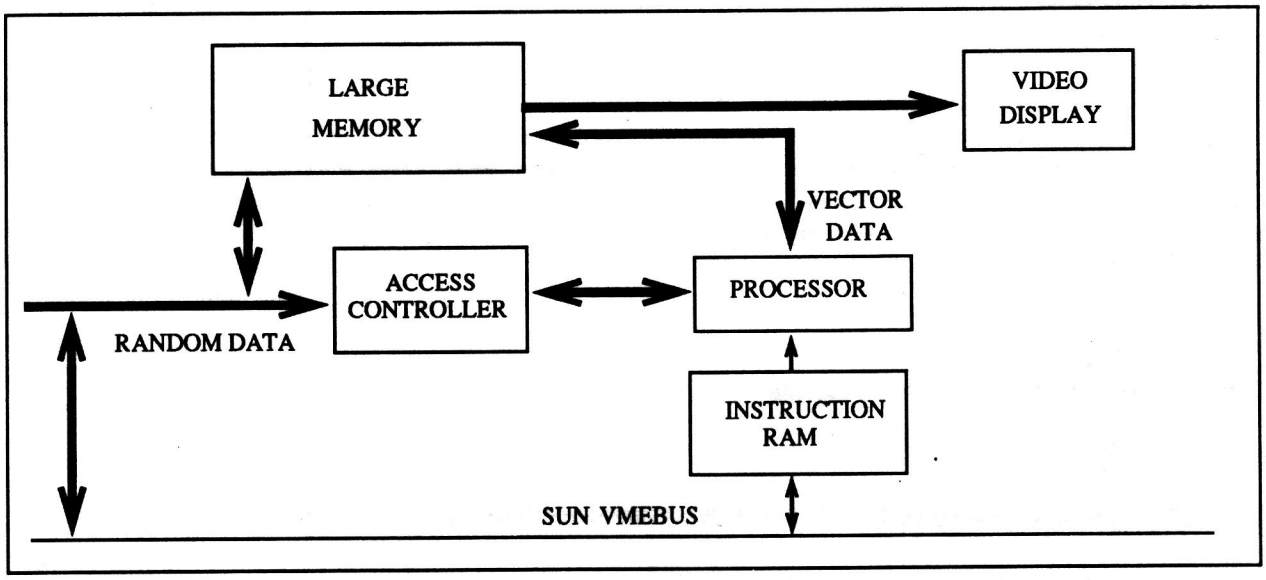


Figure 1. TAAC-1 Architecture

1.2 Processor Architecture

The processor section of the TAAC-1 consists of multiple functional units connected by multiple buses. The functions performed by each of the units and the sources and destinations of the buses are controlled by fields within the 200-bit-wide instruction word.

The processor architecture was developed with two goals. The first was for application code to run as fast as possible. The second was for compiler development to be reasonably straightforward.

A number of algorithms common in the targeted applications were hand-simulated to refine the architecture. The purpose of this exercise was to reduce inner loops to a single instruction, thus providing the same performance that might be expected from dedicated hardware. Two integer ALUs were included; this meant one could be dedicated to address calculations. The integer multiplier/accumulator (MAC) was also included for address computation as well as integer data processing. This architecture kept the free integer ALU and the floating-point unit supplied with a constant flow of data, an important design goal.

The look-up tables (LUTs) serve two purposes. Seed values for Newton-Raphson iteration to compute reciprocals (for division by multiplication) and square roots are provided by a pre-programmed LUT. A user loadable LUT is used for image processing or histogram accumulation.

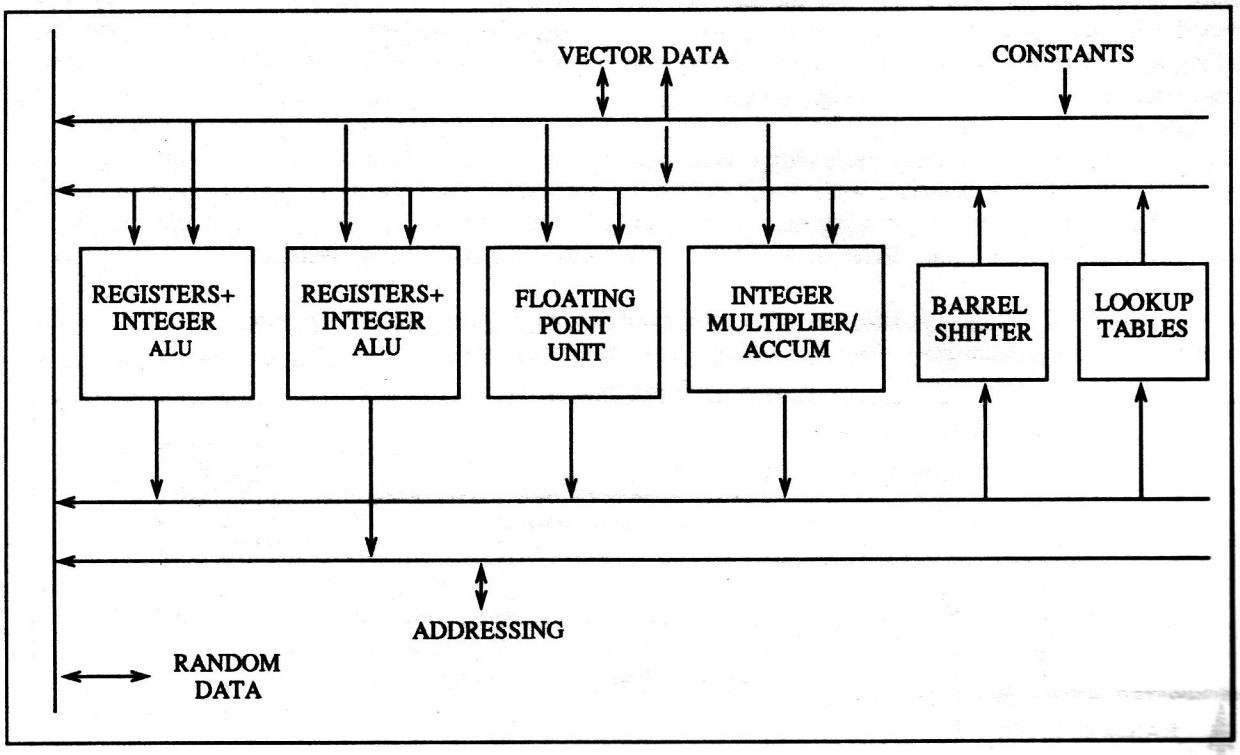


Figure 2. TAAC-1 Processor Block Diagram

Access Processor:

The access processor helps keep data flowing through the processor. It performs two functions.

The first is to act as a smart memory-access controller. It ensures that the programmer is isolated from any concern over memory timing. The controller portion also reduces access time to a minimum in a simple cache-like fashion, so that accesses to nearby regions of memory are faster than arbitrary accesses.

The second function re-arranges and controls physical addresses. This allows programs to access memory in four ways.

1) Linear addressing 2M words (32 bits/word)
2) 2-D addressing 1024 x 2048 pixels (32 bits/pixel)
3) 3-D slice addressing 32 ea. 256 x 256 slices (32 bits/pixel)
4) 3-D dice addressing 128 x 128 x 128 cube (32 bits/voxel)

The last two methods are ideal for 3-D volume data, such as a series of CT scans or a volume of seismic data. Included within the 200-bit-wide instruction word are fields to control loading, incrementing, and decrementing the X, Y, and Z addresses used in the different modes.

An additional circuit inhibits writes based on a comparison of the data to be written with the previously read value. This technique is useful for visible-surface-display algorithms.

Memories:

As mentioned earlier, the memory is constructed using video RAMs for both display and vector processing capabilities. An additional 16K x 32 fast static RAM is also included for stack-and global-variable storage.

The data/image memory has a random access port that is typically controlled by the access processor described above. This port is fairly straightforward, except it is 128 bits wide (multiplexed to 32).

The memory is divided into two banks for vector-port purposes. Each bank has a 128-bit bi-directional bus capable of speeds up to 400 Mbytes/second. The memory array consists of 256 each 64K x 4 video RAMs (although the architecture is designed to accommodate larger memories in the future). In a typical operation, one bank of memory feeds the display, while the other feeds the processor.

An expansion port reserved for future use is also included in the memory architecture, meaning there are a total of eight 128-bit-wide buses in and out of the memory.

1.3 Display Controller Architecture

The display controller has two separate sections. One controls timing, the other converts digital to analog to drive a color monitor.

The timing controller generates pixel clock, horizontal, and vertical signals locked to either an internal oscillator or to an external sync signal. The pixel clock is generated by a phase-locked loop circuit with an extremely wide range (10-100 MHz), but which is very precise (less than 1 nanosecond jitter). The horizontal and vertical sync signals and auxiliary signals, such as video blanking, are generated from bit maps loaded into RAMs addressed by counters. This arrangement gives a maximum amount of flexibility in providing any desired timing characteristics.

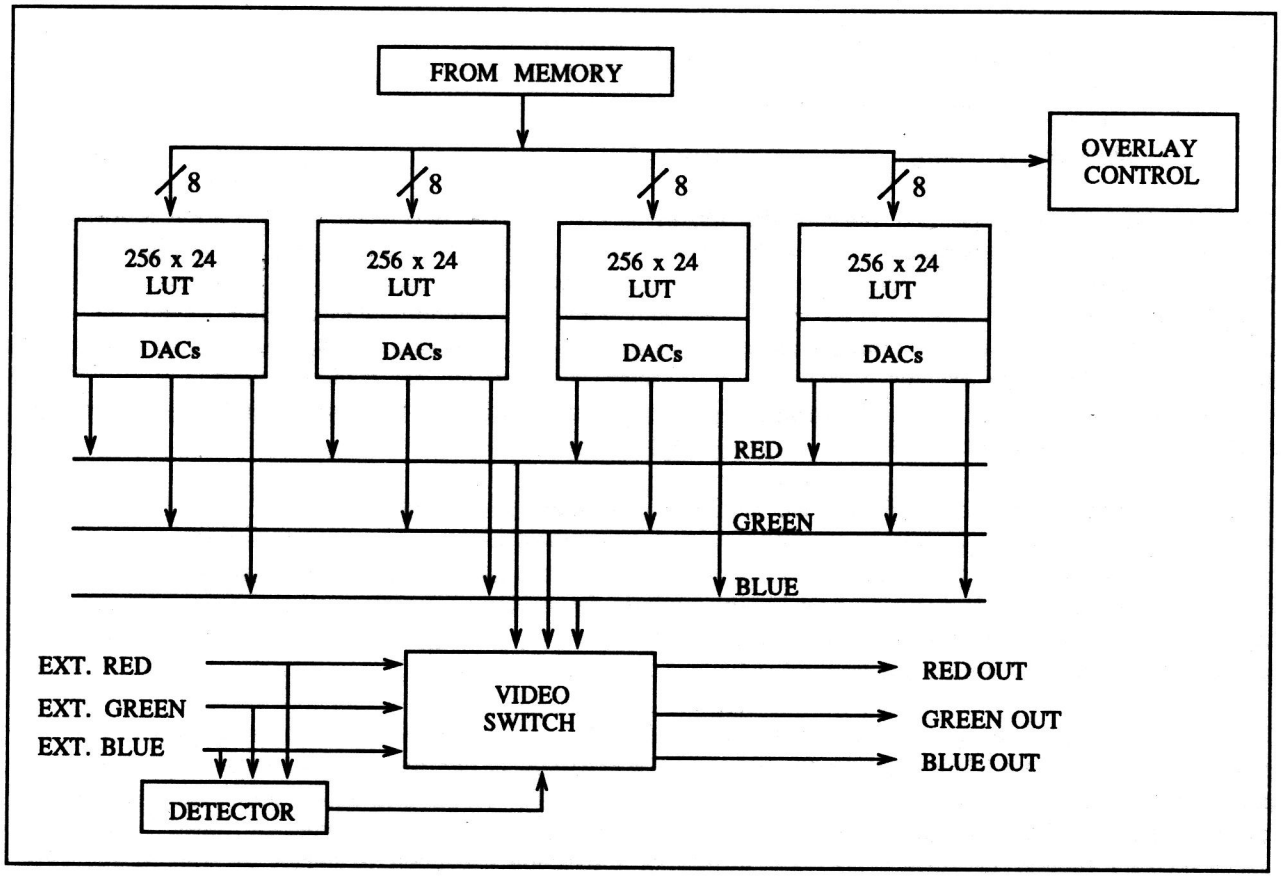


Figure 3. Video Output Block Diagram

The digital-to-analog section consists of four 256 x 24 look-up tables driving a total of twelve digital-to-analog converters. Four converters are summed together on each of the red, green, and blue outputs. Again, this design is intended to give as much flexibility as possible in a small amount of circuitboard space. For example, by loading the color LUTs appropriately, a user can switch from a single-channel pseudo-color display (256 colors from a palette of 16.7 million) to a full color display with 8 bits each of red, green, and blue, plus an 8-bit overlay channel. The overlay circuit functions by turning off the normal display DACs and turning on those belonging to the overlay channel. A bit mask allows user selection of overlay planes. If any of the enabled bits of a pixel is on, then the overlay takes place for that pixel.

Included in the circuitry is a video switch used to insert video generated by the TAAC-1 into an image created elsewhere. For example, a window can be created on the standard Sun color frame buffer and the TAAC-1 generated image is displayed in that window. The video switch circuit responds to a particular color on the input. Whenever that color is detected the video output is switched so that TAAC-1 video is displayed instead of the externally supplied input.

Sun-3 and Sun-4 workstations are each an ideal environment for a device like the TAAC-1. The internal VME bus supports 32-bit data transfers and a 32-bit address space, the same as the TAAC-1. The TAAC-1 is entirely memory-mapped into the Sun address space. Thus, the Sun processor can directly and with no driver overhead read and write image/data memory, control registers, and LUTs. This arrangement allows the TAAC-1 to be treated as just more memory.

1.4 Programming Tools

Because the TAAC-1 is a user-programmable accelerator, it has software tools in conjunction with the hardware. The tools fall in four main areas.

1) assembler/compiler
2) compiler
3) debugger
4) libraries

The assembler, C compiler, and linker/loader for the TAAC-1 were developed by Bit Slice Software of Waterloo, Ontario. The assembler works with separate mnemonic definitions for each of the fields within the 200-bit-wide instruction word. The linker and loader use object modules produced by the assembler. The C compiler is a full implementation of the C language and produces assembly code as output. (Note: The usual Unix I/O and memory allocation routines are not supported on the TAAC-1). Several functions were added to allow access to hardware features, such as X, Y, Z addressing, that are not readily expressed in C. These special functions set fields within the compiler generated instructions. In addition, assembly language code can be placed in-line with the C code.

Although a compiler cannot produce code nearly as efficiently as a skilled programmer using assembler, having a high-level language compiler for the TAAC-1 is an absolute necessity. To begin with, it encourages customers to port code to the accelerator, not rewrite code. Secondly, software development proceeds much faster with such a tool. Developing new library routines, for example, is almost always done by testing the algorithm in the standard Sun program-development environment, porting that code (through re-compilation) to the TAAC-1, and then substituting assembly language code in crucial loops only.

The debugger allows the display and manipulation of values in all registers and memory locations. This process can be seen in a window on the Sun screen. The programmer can for example, single step or set break points because the hardware supports this type of operation. The single step mode offers single assembly-language steps. The debugger gives symbolic reference to variables and program break points. A complementary profiler provides a graphical display of instruction execution frequency as well as statistical information.

Library routines fall into two classes: those that run on the Sun CPU, and those that run on the TAAC-1 processor. The first group comprises useful initialization and control routines and can be linked into user programs running on the Sun. The second group consists of optimized routines for commonly used functions in various applications. Libraries are available for graphics and image processing. A toolkit is available for rendering volumetric data. These libraries are described later in this document.

1.5 Using the TAAC-1

In porting an application or developing a new one with the TAAC-1 accelerator, two steps must be followed. The first is to identify and break out portions of the application to be moved to the TAAC-1. This must be done since the TAAC-1 cannot make any system calls. These separate modules can then be compiled using the TAAC-1 compiler.

The application programmer must then modify the main routines to load data to the TAAC-1 (from memory or disk), start the TAAC-1 portion(s) of the application, and look for results. These host routines are compiled using the normal Sun compiler. At run time, the two sets of object modules are loaded and executed, one in the TAAC-1 and one in the Sun CPU.

TAAC-1 software is designed for dedicated single-task operation which complies with the general modus operandi for users needing an accelerator. No multitasking control software has been written for the TAAC-1.

1.6 Results

The table below shows results from some experimental programs run using the TAAC-1.

	Sun-3/160	Sun-3/160+ TAAC
Ray Tracing	30 min.	30 sec.
2D FFT Routines	8 min.	8 sec.
Adaptive Histogram Equalization	6 min.	4.5 sec.

Library routines for graphics operations give the following results.

3-D transform	312,500 per second
3-D vectors	112,000 per second
3-D polygons (Gouraud shaded & Z-buffered)	15,000 per second

This level of performance is generally associated with hardwired, limited-function hardware rather than a fully programmable computation accelerator such as the TAAC-1.

2.0 Programming the TAAC-1

This section describes the general approach for developing applications on the TAAC-1 application accelerator. It will focus on the type of program that has components running both on the host Sun workstation and on the TAAC-1 - one that involves communication between, and synchronization of, these separate processes. It discusses dividing tasks between the two processors, communication, synchronization, window management, memory allocation on the TAAC-1, and C software development.

The *TAAC-1 Application Accelerator: User Guide* contains more information and goes into much greater detail about most issues described below

2.1 Program Structure

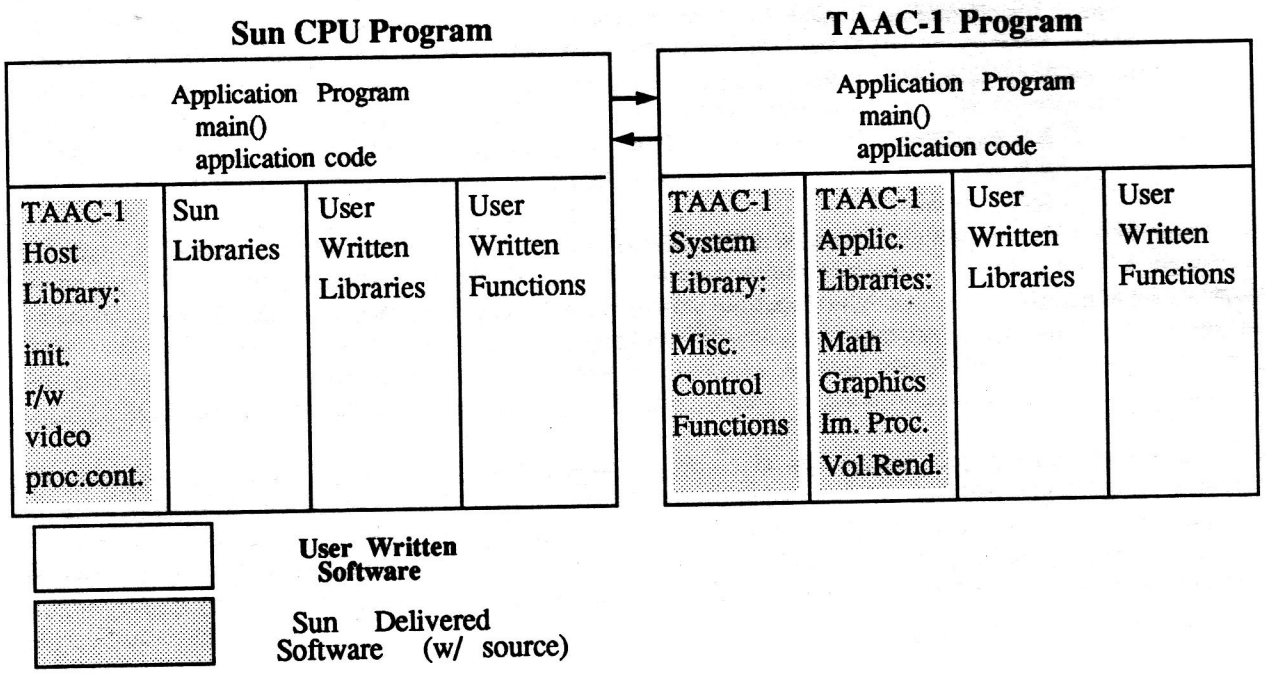
2.1.1 Dividing Tasks between the Sun and TAAC-1

Initially, one must assign portions of the program to run on the host Sun and portions to run on the TAAC-1. The table below provides general guidelines for these assignments.

Sun Modules	TAAC-1 Modules
• Disk I/O routines • User interface routines • Host system libraries • Host TAAC-1 libraries • Window management routines	• Compute intensive routines • TAAC-1 application libraries • TAAC-1 system libraries • Image generation and manipulation routines

As of this writing, TAAC-1 software does not support standard I/O routines. Furthermore, the TAAC-1 is a slave processor and cannot write to the Sun. Therefore, all reading and writing of data from and to the TAAC-1 is done by functions in the Sun routines.

The diagram below gives a conceptual view of the structure of programs which use the TAAC-1.

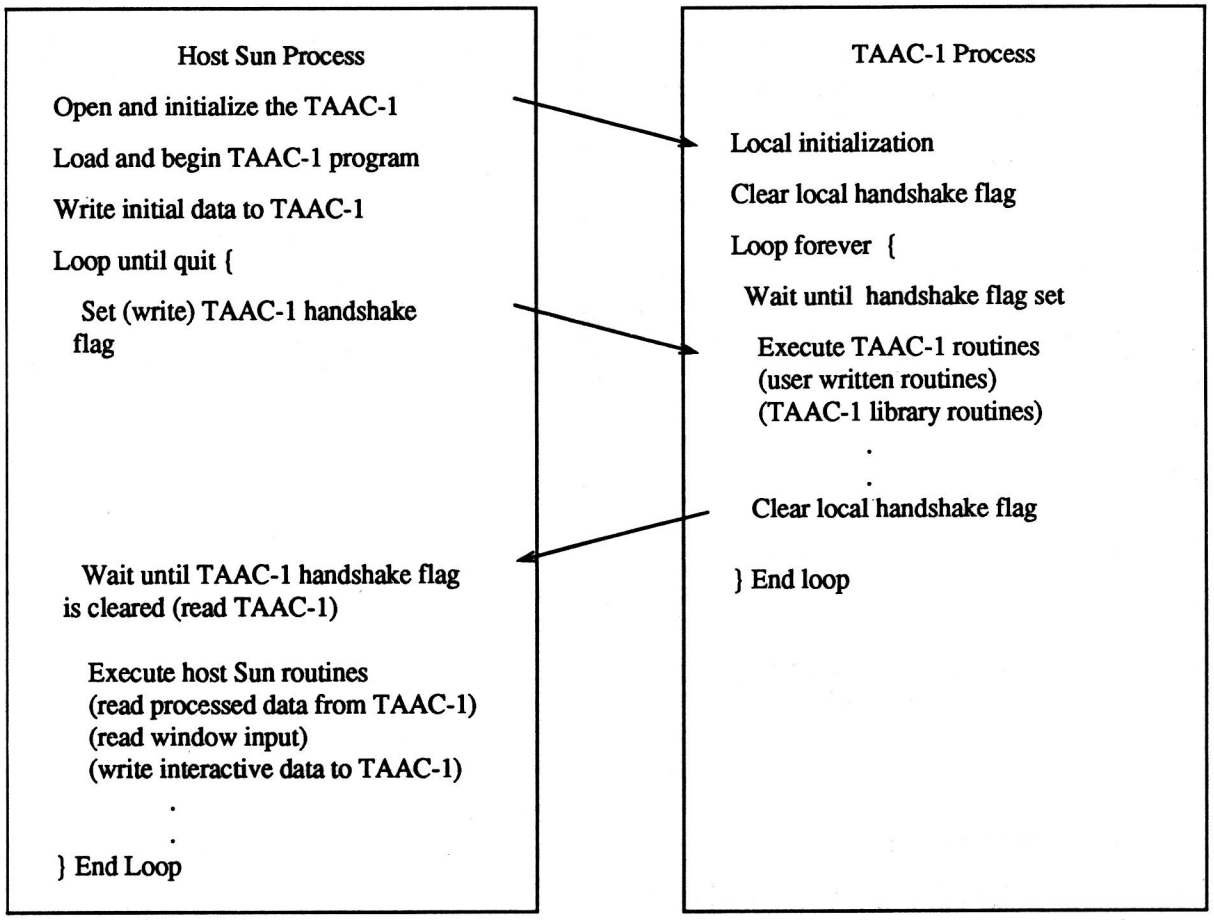


2.1.2 Synchronizing Processes

The host Sun and TAAC-1 processes are asynchronous. One typically uses a simple handshaking protocol for synchronizing the two processes. The figure below illustrates one possible organization for the handshaking.

Note that the ioflag is in TAAC-1 memory. The TAAC-1 process merely tests and/or assigns to a variable name. The host Sun program must read and/or write values to the associated address on the TAAC-1 (more on this below).

In this example the handshaking is done by clearing and setting a flag. It is also possible to assign different values to that flag as a parameter to the host or TAAC-1 program.



To further illustrate this synchronization technique, the program organization described above is duplicated below using program fragments with the appropriate TAAC-1 library routines. (In this example, as in most TAAC-1 demo software, the handshake flag is *ioflag*.)

Host Sun Process

```
#include <taac1/taio.h>
#include "taacfile_map.h"

main( ) {
TA_HANDLE *tah;
     .
     .
if ((tah= ta_open())== NULL ){
     printf("error opening TAAC");
     exit(-1);
     }
if(ta_init(tah)==TA_FAILURE){
     printf("error on TAAC init");
     exit(-1);
     }

if(ta_run(tah, "taacfile.abs")==
 TA_FAILURE){
     printf("error on ta_run");
     exit(-1);
     }

ta_write(tah,&hostvar,
     sizeof(hostvar), TC_taacvar);
     .
     .

while(!done){

     ioflag= 1;
     ta_write(tah,&ioflag,sizeof
  (ioflag), TC_ioflag);
```

TAAC-1 Process

```
#include <taac1/builtin.h>

int ioflag= 0;

main() {
     .
     .
     .

while (1) {

     while(ioflag==0) ;  /* wait
     until set */
     .
     .
     .
     ioflag= 0;

     } /* end while 1 */

} /* end main */
```

2.2 TAAC-1 Memory Usage

Variables used in TAAC-1 programs may be assigned memory in:

- registers (ALU RC or RD) – 64 registers in each ALU
- scratchpad memory (SRAM) – 16K words
- data/image memory (DRAM) – 2M words

ALU Registers

Frequently-used variables should be placed in registers. This significantly reduces the time for variable accesses. Because of the large number of registers available, entire transformation matrices or convolution kernels can be stored in register variables. Designation of register variables is done using the standard C notation:

```
register datatype variablename;
```

One can also designate the registers of a particular ALU or the actual register. For example:

```
register RD integer x;/* specifies a register of the RD alu */
register float y @2;/* specifies register #2 (in RC)      */
```

SRAM

SRAM holds the C stack as well as user-defined global and static variables (which are not specified to use other memory types). Access to this memory is slower than to registers but faster than to DRAM. This is the default location for all global and static variables. Care must be taken not to overwrite the C stack with global or static variables that are too large.

DRAM

DRAM should be used for images and very large data structures. An example of a DRAM variable declaration is:

```
DRAM float z[1000];
```

Note that this does not specify where in the DRAM the variable is to be located. The linker will do this memory allocation; however, one must specify to the linker which portions of DRAM are available for its memory allocation (This is to prevent the linker from using memory being allocated by the programmer for images or other data.). A command line option of the TAAC-1 linker specifies block(s) of DRAM which are *available* to the linker. As an example, to designate two megabyte blocks as available for DRAM variables, one at 0x80000 and one at 0x180000 , one would use the linker command line option shown below:

```
talinc -d 0x80000 0xbffff -d 0x180000 0x1bffff foo.obj ...
```

See the manual entry on the TAAC-1 linker for more information.

The programmer may also access DRAM more directly by specifying a pointer to an actual DRAM memory address or by using the TAAC-1 compiler's *builtin* functions to access memory in 1-D, 2-D or 3-D modes. Both of these approaches are discussed in the user's manual.

2.3 TAAC-1 Data Communication

Reading/Writing Variables in TAAC-1 Programs

TAAC-1 image/data memory is mapped to the Sun virtual memory space. As a result, transfer of data between the Sun and TAAC-1 is straightforward. Utility programs and library routines have been provided to make it simple.

In order to read or write a variable in a TAAC-1 program, one must have the address of the variable in TAAC-1 memory space. To get this information the variable must be made global (see the variable *ioflag* in the TAAC-1 program fragment above). When TAAC-1 programs are linked the names and addresses of all global variables are written to a *.map* file. The entry for ioflag in the .map file might look like

```
SY _ioflag 0x30000000
```

This is the address of the variable ioflag in SRAM (static ram) memory. (See the manual for more on the different types of TAAC-1 data memory). To simplify the usage of this information there is a utility program, *tamakdef*, which reads a .map file and produces a *.h* file whose entry for ioflag might look like

```
#define TC_ioflag 0x30000000
```

By *including* this .h file in one's host program the address of the variable ioflag (or any global variable) is defined to be the variable name prefixed with *TC_* . One can then use any of the host library memory access routines (ta_read(), ta_read_noinc(), ta_read2d(), ta_write(), ta_write_noinc(), ta_write2d()) to read or write this variable.

More Direct Access to TAAC-1 Memory

It is sometimes desirable to avoid the relatively small overhead of the TAAC-1 library read/write routines. For example, if one is reading a large file from disk one would want to provide a destination address in TAAC-1 memory to avoid the two step process of writing to Sun memory and then to TAAC-1 memory. The TAAC-1 host library has two routines for this purpose: ta_map() and ta_use_map().

The TAAC-1's VME interface slave mode register (SMR), which specifies the particular TAAC-1 memory type that is addressable from the Sun, is set by ta_use_map. It is used in conjunction with ta_map() which returns the Sun virtual memory address for a specified TAAC-1 memory address. For example, the following code fragment illustrates reading data from disk and writing it to the beginning of Bank B in TAAC-1 video memory (TAAC-1 address 0x200000).

```
(assume TAAC-1 is open, initialized and handle is tah)

#define NUMINTS 1000
#define TAACDEST 0x200000

{
int *taacdestptr;
FILE *fp;
char *filename;

if((taacdestptr= ta_map(tah,  TAACDEST))==NULL)
exit();
if( ta_use_map(tah, TAACDEST)==TA_FAILURE )
exit();

if( ((fp=fopen(filename,"r")) == NULL)
exit();

if( (fread(taacdestptr, sizeof(int), NUMINTS, fp)) !=NUMINTS )
exit();
}
```

Note that ta_use_map() must be called each time the slave mode register is changed (ie. if a ta_read() or ta_write() is called). These routines can also be used with the address of a variable listed in the .map file as discussed in the section above. We generally recommend the use of the read/write routines provided with the library instead of these lower level routines.

2.4 Window Management

To create a window displaying part of the TAAC-1 frame buffer one must do the following (not neccessarily in this order):

- create a window of the desired size and at the desired screen location
- fill the window with the rgb values used for TAAC-1 keying

- communicate the screen location of the window to the TAAC-1
- specify the portion of the TAAC-1 frame buffer to be displayed
- specify the video parameters for mixed Sun and TAAC-1 video

Creating a window of the proper size and location is a basic Sunview function and will not be discussed here. (Alternatively, that function can be provided by any window system and still interface with the TAAC-1 routines). The TAAC-1 host library includes routines providing the other functions. The key routines are described below.

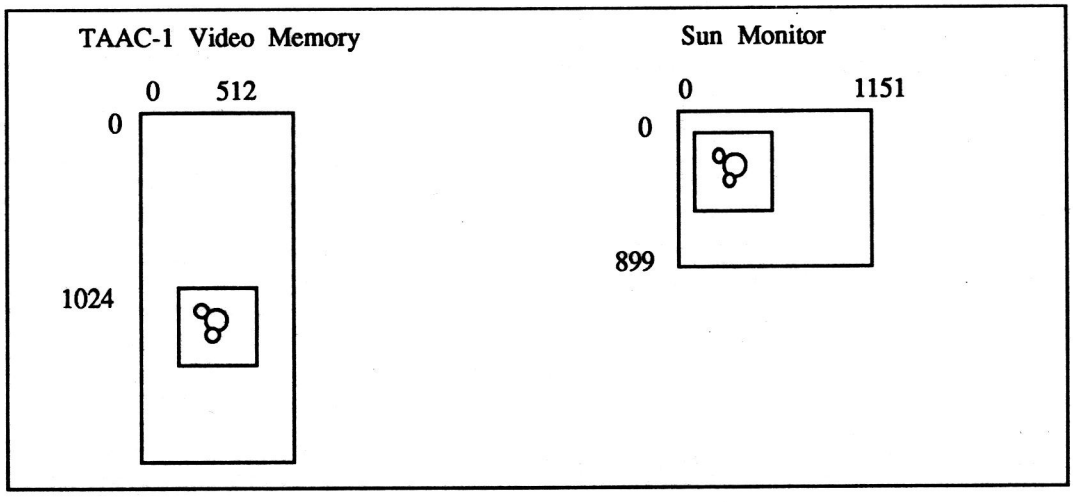
- ta_get_insert_color() - reads the TAAC-1 keying color; use the rgb values returned as the color for the window where the TAAC-1 video is to be displayed.

- ta_set_window() - sets the location and dimensions of the TAAC-1 video window in screen coordinates (pixels). If the view of the TAAC-1 frame buffer is to remain constant as the window is moved, this function should be called to reset the window parameters each time the window's screen location is changed.

- ta_set_view() - sets the 2-D memory location (pixel coordinates) that is to be displayed as the first visible pixel in the upper left corner of the displayed video.

- ta_set_video() - sets the video parameters for mixed TAAC-1 and Sun video needed for this window situation (it can also set Sun-only or TAAC-1-only video).

An example will further clarify the window location functions. Assume the part of the TAAC-1 frame buffer to be displayed begins at the 2-dimensional address x=256, y=1024 in the TAAC-1 video memory and is 512x512 (see the diagram below). The window with the key color is located at x=200, y=200 on the Sun monitor and is also 512x512.



To display this portion of TAAC-1 memory as described, the window calls would be:

```
ta_set_window(tah, 200, 200, 512, 512);
ta_set_view(tah, 256, 1024);
```

If the Sun window were moved 100 pixels to the right with the cursor, the TAAC-1 video memory displayed would shift to x=612 (unless ta_set_window is called again). To keep the same memory displayed one would have to call

```
ta_set_window(tah, 300, 200, 512, 512);
```

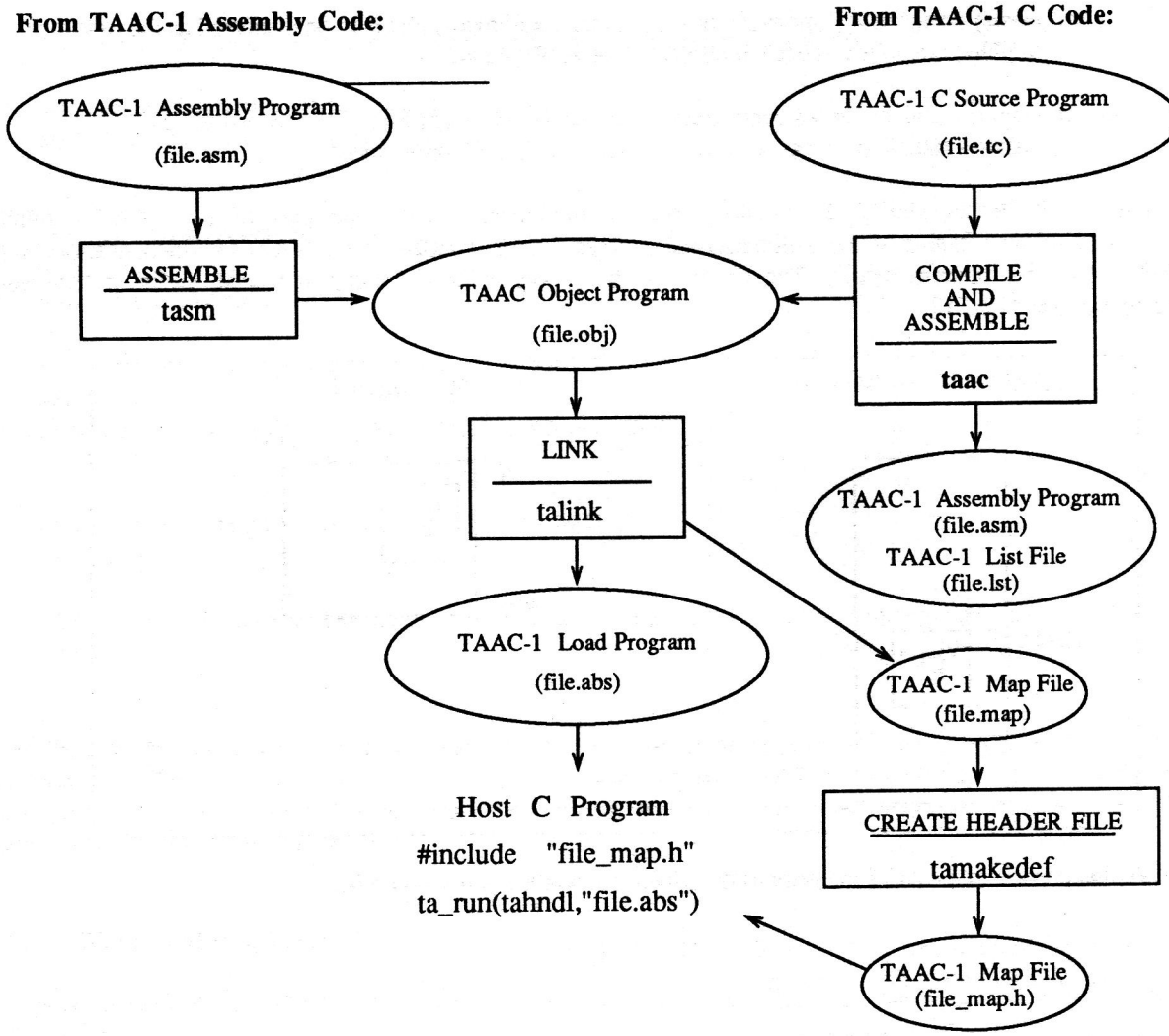
The *hostlib* directory ($TAAC1/hostlib) contains additional unsupported routines for quick and easy set-up of Sunview windows interfaced to the TAAC-1.

2.5 Program Development

Compile, Assemble, Link, Load

The diagram below (from the TAAC-1 user guide) illustrates the development of software for the TAAC-1. As with any C source code, the initial step is to compile and/or assemble (it is very unusual to begin with assembly code for the TAAC-1). The resulting object files are then linked to create a load program (.abs file). The load program is loaded at run-time by the Sun host program using the library routine *ta_run()*, or as a stand-alone routine using the utility program *tarun*.

The linker also produces a .map file with variable and subroutine address locations. The utility program *tamakedef* produces an include file from the .map file. The contents of the include file (described above) simplifies data communication with the TAAC-1.



Linking the TAAC-1 Absolute File with a Sun C Program

One may also link a TAAC-1 absolute file with a C program, eliminating the (slight) inconvenience of having a separate .abs file to load. The utility used is taabs2o which creates a .o file from a .abs file. This file can

then be linked with other .o files.
When using this structure, ta_runm() is used with only the TAAC-1 device handle as an argument instead of ta_run() which requires the .abs filename as an argument. All other steps in the process remain the same.

Program Optimization

There are a number of ways of enhancing program efficiency on the TAAC-1:

- The TAAC-1 has 128 general purpose registers. Using these for frequently accessed variables can greatly increase the speed of data access. Similarly, data that is too large for registers but small enough to fit in SRAM should use SRAM and not DRAM.
- The TAAC-1 compiler provides a number of **built-in functions** for fast access to specialized TAAC-1 components. These functions are actually macros that provide direct access to the lookup tables, the vector ports, and AC register reads and writes, without the overhead of subroutine calls.
- TAAC-1 library routines perform mathematical operations, drawing functions, and control functions. A number of these routines have been hand-optimized for efficiency. You may wish to make a copy of a library routine and modify it to suit your own needs.
- Another way to enhance program efficiency is to insert **in-line assembly code** within a C program. In-line code can be used to tighten inner loops, for example. The TAAC-1 profiler (available fall, 1988) will help to identify the parts of a program that would benefit most from optimization.

2.6 Makefile Organization

The makefile below is included to illustrate typical makefile organization for TAAC-1 software development and can be used as a template. It is designed with the separate .abs file organization.

```
# Host C-compiler
FLOAT =
DEBUG =
COPTS =  -g
CFLAGS = $(DEBUG) $(COPTS) -fsingle
CLIB = -ltaac1
SLIB = -lsuntool -lsunwindow -
    lpixrect

#TAAC-1 C-compiler
TCC = tacc
TALINK = talink
TAMAKEDEF = tamakedef
TFLAGS = -c -fsingle
TLIB = -ltaac1

# host object files
COBJ =  foo1.o foo2.o foo3.o

# TAAC-1 object files
TOBJ = foo4.obj foo5.obj foo6.obj

foo: foo.abs $(COBJ)
$(CC)  $(CFLAGS) -o foo $(COBJ)
    $(TLIB) $(SLIB)

foo_map.h: foo.abs foo.map
$(TAMAKEDEF) foo.map foo_map.h

foo.abs: $(TOBJ)
$(TALINK) -m foo.map $(TOBJ) -o $@
    $(TLIB)
$(TAMAKEDEF) foo.map foo_map.h
$(COBJ): foo.h
$(TOBJ): foo.h
foo1.o foo2.o foo3.o foo4.o:
    foo_map.h
.SUFFIXES: .tc .obj .abs

.obj.abs:
$(TALINK) -m $@.map -o $@ $< $(TLIB)

.tc.obj:
$(TCC) $(TFLAGS) $<

.c.o:
$(CC) $(CFLAGS) -c $< $(CLIB)

$(TOBJ):
```

3.0 TAAC-1 Graphics Library

The TAAC-1 graphics library provides a set of low-level routines that run on the TAAC-1 to perform common 2D and 3D graphics operations. The library routines may be used as delivered or customized by the user to fit specific applications. The delivered source code is complete and well documented so that the routines may be easily extended.

The graphics library consists of several categories of routines: transformation, shading, clipping, and rendering. These routines are intended to be a low-level interface – some users will want to construct higher-level primitives using these routines as building blocks. Many of the routines access a structure called a state table, which provides a central control block for most of the graphics subroutine calls. Multiple state tables can be used to provide fast and easy context switching (e. g., multiple rendering modes such as wireframe and Gouraud shaded polygons). **Note: These routines are callable only from programs running on the TAAC-1; they are not host callable!**

This section contains a matrix that details the functionality available in the library and a list and short description of each of the routines that make up the categories within the library. The document also contains a list of the graphics-related functions that are in the TAAC-1 miscellaneous control function library. These provide control of basic video functions such as color lookup tables and overlays. Mathematical routines for trigonometric functions, square root, reciprocal, etc. are provided in the TAAC-1 math library and are not covered in this document

TRANSFORMATION AND CLIPPING			
	2-D	3-D homogeneous	3-D non-homogeneous
Transform	t_xform2d	t_xformh	t_xformnh t_xform3x4
Vector clip	t_vclip2d	t_fastvcliph	t_fastvclipnh
Polygon clip	t_pclip2d t_fastpclip2d	t_pcliph t_fastpcliph	t_pclipnh t_fastpclipnh t_porthoclip
Project	t_proj2d	t_proj	t_proj
Matrix Multiply		t_mat4mul	t_mat3mul

SHADING	
Pseudocolor	t_pseudo
Full color	t_fastshade

RENDERING			
	2-D	3-D Pseudocolor (hardware z-buffer)	3-D Full color (software z-buffer)
Lines	t_line2d t_line	t_linep	t_linet
Antialiased lines	t_aaline2d t_aalinep2d	t_aalinep	t_aalinet
Antialiased shade-interpolated lines	t_dcline2d		t_dclinet
Wireframe polygons	t_poly2d	t_poly	t_poly
Flat-shaded polygons	t_poly2d	t_poly	t_poly
Gouraud-shaded polygons		t_poly	t_poly
Text	t_rasttext t_rasttextbg		
Circles	t_circle		
Rectangles	t_rect		
BitBlt	t_blit		
Objects		t_obj t_objnh	t_obj t_objnh

3.1 Transformation Functions

These functions apply a spatial transformation to an input vertex list to produce an output vertex list. The transformation is specified by a user generated transformation matrix. The projection functions t_proj and t_proj2d provide simple ways to convert from modeling space to screen space and output vertex coordinates in the format required by the rendering functions. Two square matrix multiplication routines are also provided to facilitate concatenation of matrices.

Name	Description
t_mat3mul	3x3 matrix multiply
t_mat4mul	4x4 matrix multiply
t_proj	project 3-D vertex list
t_proj2d	project 2-D vertex list
t_xform	transform vertex list
t_xformh	transform 3-D homogeneous vertex list
t_xformnh	transform 3-D non-homogeneous vertex list
t_xform2d	transform 2-D vertex list
t_xform3x4	transform 3-D non-homogeneous vertex list

3.2 Clipping Functions

These routines process the input list of vertices to produce an output vertex list that has been clipped. Some of the polygon clippers support optional clipping of vertex colors and/or vertex normals. Unless otherwise noted, clipping is done in floating point in world space.

Name	Description
t_fastpcliph	fast clip polygon (homogeneous)
t_fastpclipnh	fast clip polygon (non-homogeneous)
t_fastpclip2d	fast clip 2-D polygon
t_fastvcliph	fast clip vector list (homogeneous)
t_fastvclipnh	fast clip vector list (non-homogeneous)
t_pcliph	clip 3-D polygon (homogeneous). Optionally clips vertex colors and vertex normals.
t_pclip2d	clip 2-D polygon
t_pclipnh	clip 3-D polygon (non-homogeneous). Optionally clips vertex colors and vertex normals.
t_porthoclip	screen space clip to arbitrary orthogonal planes
t_vclip2d	clip 2-D vector list

3.3 Shading Functions

The shading functions take as input a list of vertex normals and lighting model information from a state table. The dot product of each input normal and the light source vector is formed. This intensity value is then combined with the color obtained from the state table or the input vertex color list and the results written to an output list.

Name	Description
t_fastshade	fast shade vertex list (full color) with optional vertex colors
t_pseudo	shade vertex list (pseudocolor)

3.4 Rendering Functions

The rendering functions take input lists of vertices and color information and draw to image/data memory. These functions include line, polygon, circle, text, and rectangle primitives.

Name	Description
t_aalinep	draw 3-D antialiased line (pseudocolor)
t_aalinet	draw 3-D antialiased line (full color)
t_aaline2d	draw 2-D antialiased line (full color)
t_blit	rectangular area bltblt
t_circle	draw unfilled 2-D circle
t_dclinet	draw 3-D depth-cued antialiased line
t_dcline2d	draw 2-D depth-cued antialiased line
t_erase	fast rectangular area erase
t_line	draw 2-D line
t_linep	draw 3-D hardware z-buffered line
t_linet	draw 3-D software z-buffered line
t_line2d	draw 2-D line
t_obj	render 3-D object with homogeneous coordinates
t_objnh	render 3-D object with non-homogeneous coordinates
t_poly	render 3-D polygon
t_poly2d	render 2-D polygon
t_rasttext	draw raster text
t_rasttextbg	draw raster text with background
t_rect	draw rectangle

3.5 Graphics-Related Functions in the Miscellaneous Control Function Library

Name	Description
t_get_alphafill	read hardware fill state in channel 3 (alpha)
t_get_bitmask	read the current 32-bit DRAM write mask
t_get_bitmask_id	return the current bitmask id
t_get_bitmask_mode	return the current bitmask mode
t_get_blink_mask	read current blink mask for the selected channel
t_get_btcommand	read command register state of the selected DAC channel
t_get_channel_select	read display state of all four video channels
t_get_colormap	read the rgb colormap for the selected video channel's DAC
t_get_overlay_colors	read DAC overlay colors in selected channel
t_get_overlay_mask	read the 8-bit overlay enable mask
t_get_read_mask	read current readmask for the selected channel
t_get_rgbfill	read hardware fill state for channels 0,1,2
t_get_view	read address of first displayed pixel
t_set_alphafill	set hardware fill state for channel 3 (alpha)
t_set_bitmask	set the 32-bit DRAM write mask
t_set_bitmask_id	set current bitmask id
t_set_bitmask_mode	enable/disable bitmask mode
t_set_blink_mask	set current blink mask for the selected channel
t_set_btcommand	set command register state of the selected DAC channel
t_set_channel_select	set display state of all four video channels
t_set_colormap	set the rgb colormap for the selected video channel's DAC
t_set_overlay_colors	set the DAC overlay colors in the selected channel
t_set_overlay_mask	set the 8-bit overlay enable mask
t_set_read_mask	set current readmask for the selected channel
t_set_rgbfill	set hardware fill state for channels 0,1,2 (red,green,blue)
t_set_view	set address of first displayed pixel

4.0 TAAC-1 Image Processing Library

The TAAC-1 image processing library provides a basic set of TAAC routines that can be used as delivered or extended to a specific application. Each image processing user tends to have very specific library requirements tailored to a particular application. The delivered source code will be complete and well documented so that it may be easily extended by the user to address a specific application.

The library is organized into several sections: point functions, geometric functions, information functions, and transform functions. Point functions are those functions that produce an output pixel as a function of a corresponding point in the input image or images (e.g., add, blend, and). Geometric functions are those functions that produce an output pixel based on some spatial transformation of an input pixel or some number of input pixels (e.g., zoom, reflect, control point warping). Functions for statistical analysis produce data about an input image or subimage (e.g. minimum and maximum, histogram). Transformation functions produce an output pixel intensity based on a region of pixel input intensities (e.g., fft, convolution). Functions for Fourier analysis are those functions used to process and examine image frequency content. Morphologic functions are those functions used to highlight or extract structures or specific features within an image. TAAC-1 utility functions are those functions that make use of the special capabilities of the TAAC-1 processor (e.g., read, write, lookup table). **Note: These routines are callable only from programs running on the TAAC-1; they are not host callable!**

For each function the user may specify the image datum by indicating a field width and affect (size of image). File I/O and user interaction (roam, pan, pick, contour, etc.) are not specified and are assumed to be handled by other library routines. Graphics routines such as line and polygon drawing, transformation, and pixel filling are assumed to be addressed by the graphics library.

4.1 Point Functions

All of the point functions generate an output pixel as a function of a single input pixel or pair of input pixels at corresponding locations in the selected image regions. The input regions may overlap each other and/or the output region may overlap either of the input regions. All point functions operate on two-dimensional regions.

Function Name	Description
add	sum two image regions
addconst	two's complement addition of image and constant
and	bitwise and of two image regions
blend	blend the red, green, and blue pixel values based on alpha
divide	signed divide of two image regions, limited to 12-bits max
invert	compute image positive/negative from negative/positive
lin_transf	perform linear transformation of form ax + b
multconst	multiply image by constant
multiply	signed multiply of two image regions
nand	bitwise nand of two image regions
nor	bitwise nor of two image regions
not	bitwise logical inverse of an image region
or	bitwise or of image regions
shift	left and right shift of data fields
subtract	subtract two image regions
udivide	unsigned divide
umultiply	unsigned multiply
xor	bitwise exclusive or of two image regions

4.2 Geometric Functions

The geometric functions perform spatial transformation on two-dimensional input image regions.

Function Name	Description
bicub_interp	spatial interpolation based on bicubic estimation
control	2-D spatial transformation using a control grid
interp	bilinear interpolation zoom and contract
interp_rgb	bilinear interpolation zoom and contract for rgb
reflect	2-D reflection in x or y
rotate2d	2-D rotation about image center
zoom2d	2-D magnification by pixel replication

4.3 Functions for Statistical Analysis

The information functions accept as input an arbitrary rectangular region of pixels and produce information about the region (e.g., minimum, maximum, histogram).

Function Name	Description
extrema	determine minumum and maximum in image region for signed data
hist	determine histogram in image region
moments	mean and variance of image region
uextrema	unsigned extrema

4.4 Functions for Fourier Analysis

The transformation functions produce an output pixel whose intensity depends on a region of input pixels. This includes convolution and transform processing.

Function Name	Description
bandpass	band pass butterworth filter (frequency domain)
bandreject	band pass butterworth filter (frequency domain)
conv_3x3	integer convolve
conv_5x5	integer convolve
conv_NxM	integer convolve
fltconv_NxM	floating point convolve
cmplx_fft1d	1-D floating point fast fourier transform (complex input)
cmplx_fft2d	2-D floating point fast fourier transform (complex input)
cmplx_ifft1d	1-D floating point inverse fast fourier transform (complex input)
cmplx_ifft1d	2-D floating point inverse fast fourier transform (complex input)
fft_1d	1-D floating point fast fourier transform (real integer input)
fft_2d	2-D floating point fast fourier transform (real integer input)
highpass	high pass butterworth filter (frequency domain)
ifft_1d	1-D floating point inverse fast fourier transform (compressed complexl input)
ifft_2d	2-D floating point inverse fast fourier transform (compressed complex input)
log_spectrum	compute log power frequency spectrum of fft
lowpass	high pass butterworth filter (frequency domain)
median_filter	median window filter
user_filter	user specified filter (frequency domain)

4.5 Morphologic Functions

Function Name	Description
connect	segment image based on connected components
contour	trace a contour of points based on label image
dilate	function performs edge "thickening" operations
erode	function performs edge "thinning" operations
threshold	set image pixels based on specified threshold
uthreshold	set image pixels based on specified threshold for unsigned data

4.6 Transformation Functions

Function Name	Description
clahe	interpolated contrast limited adaptive histogram equalization
histeq	equalize histogram of image region
lut_transform	transform an image via look-up table mapping

4.7 Utility Functions

Function Name	Description
read_lut	read look-up table
write_lut	write look-up table

5.0 TAAC-1 Volume Rendering Toolkit

The TAAC-1 volume rendering toolkit provides a set of programs for rendering volumetric data as well as associated utility functions. The toolkit is applicable for a wide range of applications which have 3-dimensional data – medicine, earth resources, biochemistry, meteorology, fluid flow, etc. The programs and functions come with complete, well-documented source code for integration into third party software or extension by end-users requiring additional functionality.

The current toolkit consists of three rendering programs: Cubevu, Rayvu and Cloudvu, providing three different rendering schemes. The three programs use a common data format and utility programs are provided for data manipulation. Multiple data types are supported – single-parameter 8, 16, and 32-bit integer, multi-parameter 8-bit integer and single-parameter 32-bit floating point. Rendering of data larger than can fit in TAAC-1 memory is transparent to the user. The programs also interface with the 2-D image processing routines for extended functionality.

5.1 Cubevu

Cubevu displays and manipulates the visible x, y, and z planes of an orthographics projection of 3-dimensional data in real time. The user can, interactively, move an oblique (arbitrary) slice plane through the cube, displaying the data in an interior portion of the volume. The plane is specified using "pitch" and "yaw" to define the angle of the plane and "push" to determine how far the slice plane is pushed into the volume. Cutting the volume with this slice plane has the effect of cutting away a portion of the cube, at any angle, to expose the interior of the volume. Additionally, the portion of the volume data to be displayed may be restricted along any of the 3 volume axes. These near and far clip planes for each of the 3 axes, together with the arbitrary oblique clip plane effectively allow interaction with 6 orthogonal and 1 oblique slice plane. Interactively clipping with these slice planes is non-destructive, the volume data is not altered. The volume slice planes are generated by either point sampling or interpolation of the volume data for each pixel.

The user also has the capability to:

- specify aspect ratio and scale
- map data to pseudocolor or gray scale
- do intensity windowing
- change orientation of the cube
- do bilinear interpolation for orthogonal slices, trilinear interpolation for oblique slices
- apply a threshold function to the data, rendering only the voxels above the designated threshold
- move multiple slice planes through the cube, one of which may be oblique
- display a selected slice in a separate window for further, 2-D image processing
- record and playback sequences of images generated

Functions planned for the future include milling operations, extrusion clipping, and transparency among others.

5.2 Rayvu

Rayvu does ray tracing of volumetric data. Rays are projected from the user specified eye point into the 3-D data set. Along each ray, trilinearly interpolated samples from the original image slices are compared to user-selected density thresholds. The different density ranges can be rendered as opaque or transparent and are assigned colors. When a density threshold is crossed by a ray, a surface normal is calculated from the local gradient and a light source shading model is applied.

The user also has the capability to:

- specify aspect ratio and scale
- specify lighting parameters
- "plug in" custom routines for ray generation, interpolation, shading, ...
- select among preview, fast, and high quality generation modes

Functions planned for the future include multi-parameter rays, successive refinement during image generation, different rendering schemes, volume clipping/editing, and perspective views, among others.

5.3 Cloudvu

Cloudvu is a rendering program for point primitives. It's input data is sets of points in 3-dimensional space. The points can have an associated normal vector and/or an associated 3-bit or 8-bit pseudocolor value. The program is capable of transforming and rendering 225,000 points per second.

Different display modes include:

- grey scale gradient
- pseudocolor gradient
- Z (depth) shaded
- Z shaded pseudocolor
- direct color display

6.0 Other Available Technical Notes

4 Example of Porting Software to the TAAC-1
Incorporating the TAAC-1 into an existing program with examples of assembly code

5 Look-Up Functions for 12-Bit Images
Using the TAAC-1 programmable Look-up Table for 12-bit to 8-bit image processing

6 Volume Imaging with the TAAC-1
TAAC-1 performance in rendering volumetric data by slicing along a fixed axis or ray casting.

7 The TAAC-1 C Compiler
Brief description of compiler features.

9 Application Acceleration: Development Of The TAAC-1 Architecture
Discussion of design criteria, architecture, and programmability of the TAAC-1 (largely incorporated in this document).

10 VideoTape Recording with the TAAC-1
Technical issues in direct videotape recording of TAAC-1 video.

11 Programming the TAAC-1
Task division, communciation, synchronization of processes, window management, memory usage and software development in using the TAAC-1 (largely incorporated in this document).

12 TAAC-1 Image Processing Library
(Incorporated in this document).

13 Medical image processing on an enhanced workstation
TAAC-1 architecture, software tools, and 2-D and 3-D medical image processing and display applications described.

14 TAAC-1 Graphics Library
(Incorporated in this document.)

15 TAAC-1 Software
Lists software provided for the TAAC-1 including C development tools, host library, and TAAC-1 system, math, graphics and image processing libraries.

The Network Is The Computer ™

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
TLX 287815

For U.S. Sales Office
locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Sun House
31-41 Pembroke Broadway
Camberley
Surrey GU15 3XD
England
0276 62111
TLX 859017

Australia: (02) 436 4699
Canada: 416 477-6745
France: (1) 46 30 23 24
Germany: (089) 95094-0
Japan: (03) 221-7021
Nordic Countries: (08) 764 78 10
Switzerland: (1) 82 89 555
The Netherlands: 02155 24888
UK: 0276 62111

Europe, Middle East, and Africa,
call European Headquarters:
0276 62111

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales

Specification subject to change without notice

Printed in USA 6/88 FE181-0 7.5K