USER'S MANUAL

1.2

P·I·X·A·R

Overviews

Tutorials

Reference Documents

Advanced Topics

Appendices

Glossary

SUPPLEMENTARY DOCUMENTS

MEMO: Pixar User's Manual
TO: Pixar Customers
FROM: Pixar Documentation Group
DATE: December 2, 1986

Welcome to the Pixar User's Manual. This book contains hardware and software overviews, programming tutorials, reference documents and additional material from the Pixar Tech Memo files. We hope you will find this material helpful. Also note that the Pixar Software Release 1.2 contains plenty of source code which will be very helpful in writing your own software for the Pixar Image Computer.

Please contact us if you have any questions regarding the material in this manual.

TABS: You will find two kinds of tabs to help divide the traditional UNIX sections into subsections. The major tabs have the familiar meaning, while the minor tabs correspond to subsections (e.g., specific tutorials).

BUGS: Mail in the pink comment forms, or use electronic mail to submit on-line comments and suggestions (e.g., *mail pixar!bugs*).

NOTE:

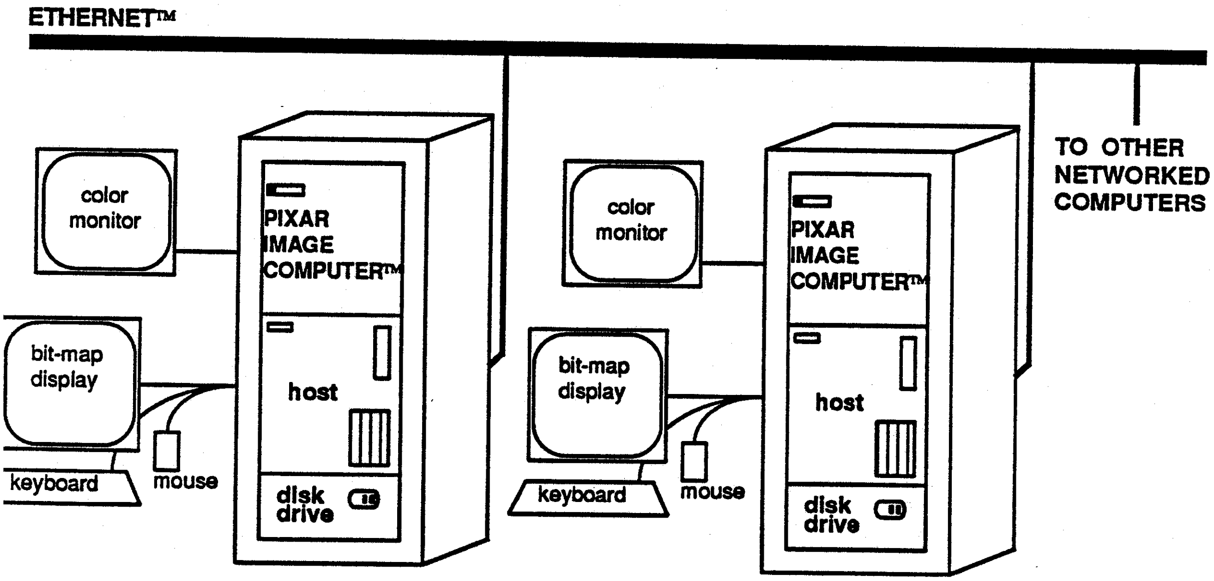
The information in this manual is for informational purposes only and is subject to change without notice. Use, duplication or disclosure by the Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013.

Pixar Software Overview
November 3, 1986

The Pixar Image Computer™ provides an innovative and powerful architecture for performing image-computing operations. The software for the Pixar Image Computer provides a collection of tools that utilize the full capabilities of the hardware to perform a wide range of standard image-computing operations quickly and efficiently, and powerful development tools that allow additional software to be easily developed for the system.

The Pixar Image Computer is closely coupled to a host computer that provides those facilities and functions that do not directly relate to image-computing. The host computer provides network access, a program development environment, and general resources required by application programs (e.g., operating system, file system, mass storage, etc.). The Pixar Image Computer can be interfaced to a variety of host computers, including Sun, Symbolics, Digital Equipment Corporation, and Silicon Graphics computer systems.

Pixar offers a Pixar Development System™ that includes a Sun-3™ computer, bitmapped workstation, 380 MByte disk, with the Sun UNIX™ operating system and Pixar Software configured for use by software developers and research laboratories. Other configurations and specialized application programs may be purchased from Pixar OEMs and VARs who offer the Pixar Image Computer in conjunction with their products. A development environment that includes Pixar Image Computers is shown below.



All image-computing operations are performed using software and resources of both the host computer and the Pixar Image Computer. The design of the closely coupled interface between the two machines allows application programs to be implemented on the host computer with concurrent image processing performed by the Pixar Image Computer. The host computer program provides the data for, and controls the overall operation of, the Pixar Image Computer. Table 1 lists the actions of the host computer and the corresponding actions of the Pixar Image Computer for a typical application program:

Table 1.

Application Program Actions		
Step	*Host Computer*	*Pixar Image Computer*
1.	Pixar Device Open	Initialization
2.	Load Pixar Software	none
3.	Transfer Data To Pixar	Accept Data From Host
4.	Invoke Image Computing Operation	Perform Image Computations
5.	Interact with User	Display Image on Video Monitor
6.	Retrieve Data From Pixar	Transfer Data To Host
7.	Pixar Device Close	none

The host computer/Pixar Image Computer interface simplifies the development of complex image-computing applications. The host computer interface provides for high bandwidth data transfer between the host and Pixar Image Computer, under polled and interrupt control. The Pixar software provides facilities to allocate and deallocate Pixar resources, load Pixar software, transfer data to and from the Pixar, as well as view images from any portion of the Pixar Image Memory using the video output capabilities of the Pixar Image Computer. In addition, many of the most commonly needed image-computing operations have also been provided as utility programs, available for immediate use without any programming required.

The software provided with the Pixar Image Computer may be divided into five classes:

- Utility Programs
- Development Libraries
- Development Tools for Pixar Programming
- System Software
- Contributed Software

Utility Programs

Utility programs are provided for the most common image operations performed using the Pixar Image Computer. These utilities are provided with a wide range of options available, allowing general user access to many advanced capabilities of the Pixar Image Computer before any application-specific software is written. Included are:

- Display Utilities:

cbars	generates standard color bars in an image memory window.
clr	clears an image memory window to specified value(s).
gamma	sets gamma-corrected colormap.
guide	displays a field-guide alignment pattern in an image memory window.
loop	sequences preloaded animation frames.
ramp	displays a ramped pattern horizontally or vertically in an image memory window.
tool	allows the image memory contents to be examined interactively.
video	modifies the configuration of the video controller.

- Image Input/Output Utilities:

gt	gets (loads) an image into the image memory from a picture file on mass storage.
gtinfo	displays information describing a picture file.
sv	saves all or part of the Image Memory to a picture file on mass storage.

- Image Processing Utilities:

blur	applies a box filter to an image memory window.
clamp	clamps the contents of an image memory window to [0,1].
conv	convolves an image memory window with arbitrary 3x3 filter.
copy	copies an image memory window to another window.
hg	generates a histogram of the pixels in an image memory window.
merge	merges two image memory windows using compositing operators.
perm	permutes image memory rows/columns, allowing clamping, inversion, etc.
resize	resizes an image memory window onto a destination window.
rotate	rotates and scales an image memory window onto a second window.
scale	scales the RGBA intensities of a image memory window.

- Miscellaneous Utilities:

pixinit	initializes the Pixar hardware and the software configuration tables.

The utility programs allow a large number of arguments to be specified, and allow functions to be performed over selected portions of the Image Memory. Both the utility programs and the Pixar development libraries use *logical frame buffers* and *pixel windows* to partition the image memory.

Development Libraries

Development libraries are provided for both the host computer and the Channel Processor(s) (Chap^{TM}) of the Pixar Image Computer. These development libraries provide high-level and low-level library functions for diverse uses that include the allocation and management of Pixar resources (Chaps, Chap program memory, image memory, scatchpad memory), data transfer operations, pixel operations, image processing functions, etc. The large variety of library functions available enables many applications to be written without the need to program the Pixar Image Computer directly. The libraries provided are:

- Host Libraries:

libchad a library of functions that manage the allocation and deallocation of Pixar resources. Also included are procedures to initiate execution of Chap procedures and to synchronize host/Chap program execution.

libpicio a library of functions for encoding, decoding, loading, and unloading image "picture" files. Image "pictures" are areas of Image Memory that are stored in digital form on the file system of the host computer. Most image "pictures" are encoded using published run-length encoding techniques to facilitate the interchange of data among Pixar Image Computers and other image gathering/processing equipment.

libpirl a library of functions for manipulating rectangular pixel windows of the Pixar image memory. The libpirl functions provide a powerful set of building blocks that perform many of the most commonly required image processing functions. Included are library functions to perform arithmetic operations, boolean algebra, linear algebra, filtering, clamping, convolution, merging, and copy operations on Image Memory pixel windows.

libpixar a library of functions for accessing the Pixar Image Computer at a low level. Library functions are provided that allow direct access to all memories, registers, and buses that can be accessed by the host computer.

- Chap Libraries:

libchad a library of Chap functions that service the requests made by the host *libchad* functions.

libpicio a library of Chap functions that service the requests made by the host *libpicio* functions.

libpip a library of Chap functions that perform common image processing operations using the Chap. Many of the host *libpirl* image processing library functions invoke the *libpip* Chap functions to actually perform the image processing operation.

libpm a library of Chap functions that perform common arithmetic operations using the Chap. A variety of arithmetic operations are provided, including addition, subtraction, multiplication, division, square root, absolute value, 4*x*4 matrix multiplication, etc.

libpt a library of Chap functions that provide a variety of methods for transfering pixels between the Pixar Image Memory and a Chap's scratchpad memory.

libpx a library of Chap functions that perform geometric transformations on images. Functions are provided to change the size of an image using linear, quadratic, or cubic interpolations. Functions are also provided that perform image rotations and warping.

The Chap libraries are implemented to utilize the SIMD Chap architecture and, in general, make effective use of the hardware features of the Pixar Image Computer. Source code is provided to all Development Libraries to encourage the further development of additional host and Chap libraries and user applications.

Development Tools for Pixar Programming

Development tools are provided to enable custom applications to be developed for the Chap Processor(s) of the Pixar Image Computer. The tools provided include a Chap assembler, dynamic loader, debugger, and other miscellaneous tools that aid in program development and debugging. Additional tools may also be provided by the various host environments. For example, a LISP compiler for the Chap is available from Symbolics for the Symbolics environment. The standard tools provided are:

- Programming Tools:

chas the Chap assembler. *Chas* takes one or more input files and generates a relocatable object file suitable for use with the Chap linkage editor, *chld*, the Chap dynamic loader, *chload*, or that may be incorporated into an object-code library using the standard UNIX *ar* command and the *chranlib* tool.

chc the Chap compiler. *Chc* is analogous to the UNIX *cc* command. *Chc* is used to assemble and link *Chas* programs. Like *cc*, *chc* invokes the C preprocessor. *Chc* then assembles the code into relocatable object files and links them with other object files to form executable modules.

chld the linkage editor for the Chap. *Chld* combines several object programs into one, resolving external references, and loading object files from libraries if necessary to resolve all external references.

chload the loader for the Chap. *Chload* links and relocates one or more relocatable object files created by *chas* or *chld*, downloading the resulting program into the Chap and executing it.

chranlib the archive converter for Chap libraries. *Chranlib* converts archives produced by the standard UNIX *ar* utility, to a library form that *chload* can load efficiently.

- Debugging Aids:

charm allows a user to interactively interrogate the state of a Chap, load and link Chap code, and control execution of programs running in a Chap. The user interface and facilities of *charm* are similar to the UNIX *adb/fP debugger.*

chcmp compares the contents of a Chap object file against the contents of Chap instruction and scratchpad memories.

chconfig interacts with the operating system's Pixar hardware configuration tables. The hardware configuration of each Pixar Image Computer is automatically determined when the operating system is initially booted, and the tables reflect that configuration unless explicitly modified using *chconfig*. Using these tables to determine the existing hardware configuration allows configuration-independent programs to be written that can take advantage of additional Pixar resources (e.g., Image Memory, multiple Chaps, etc.) when they are available.

chd disassembles Chap object-code.

chmap prints the symbol table associated with a Chap. This symbol table, used by *chload*, reflects the known contents of the Chap's instruction and scratchpad memories. *Chmap* may also be used to initialize a Chap's symbol table.

chnm outputs the name list (symbol table) of a Chap object file or object-code library.

chsize outputs the sizes of the various segments of a Chap object file.

System Software

System software is provided with each Pixar Image Computer to handle the low-level interfacing to the host operating system and hardware. This software includes:

- Pixar Software Installation Procedures
- Pixar Device Drivers
- Pixar Diagnostics

The Pixar Device Driver provided for each host provides closely coupled access to the Pixar Image Computer. The close coupling is provided by means of a memory-mapped hardware interface, which is mapped directly into a user's virtual address space by a "device open" call to the Pixar device driver. This direct mapping allows I/O operations to be initiated by each user process without the overhead normally associated with I/O operations.

The Pixar software utilizes this facility extensively, enabling host computer software to, in effect, make remote procedure calls to software that resides in the Pixar Image Computer to perform image-computing operations. Parameters and other data may be passed efficiently and conveniently directly between the host and the hardware interface. Also, synchronization of the host and Pixar software can be accomplished by polling, or by means of a host interrupt that can be generated under program control by the Pixar Image Computer. This allows the host and Pixar Image Computer to operate concurrently for optimal throughput.

Contributed Software

Software developed by Pixar or by individual users of the Pixar Image Computer can be contributed for general distribution by Pixar. Software in this category need not undergo the stringent product testing and documentation effort normally required for software products. This provides a means to distribute a variety of demonstrations, prototypes, specialized functions, images, data sets, etc., that might otherwise be restricted from distribution. Pixar ensures that all source code, compilation and installation procedures, and a minimal level of documentation is available, but otherwise assumes no responsibility or liability for the software.

Software in this category that is available to all Pixar customers includes:

- Demonstrations:

fft is a mouse-driven demonstration of real-time fast Fourier transforms computed and displayed by the Pixar Image Computer

tree demonstrates the compositing features of the Pixar Image Computer using trees to show the speed with which complex scenes can be composited. The trees themselves were rendered on a general-purpose computer using a technique developed at Pixar called "particle systems".

videotool is a Sunview window-based demonstration that allows interactive selection of image memory and video features of the Pixar Image Computer.

cubetool is a Sunview window-based demonstration of the use of image memory to store and view the orthogonal faces of a three-dimensional cube represented by a series of "stacked" images. The view may be altered by interactively moving a viewing plane through the volume of data. This is useful for viewing sets of images, such as CT slices, seismic data, etc.

magtool is a Sunview window-based demonstration that allows the interactive examination of a sequence of images using a "magnifying glass" whose magnifying power and position are under user control.

- Development Tools:

libaargs is a library of argument-parsing functions that allows User Utilities with standard and optional argument lists to be easily implemented (most Pixar User Utilities use this development tool).

- Images:

1984 the *1984* image of the pool balls that appeared on the cover of the July 1984 issue of *Science 84* magazine. This image was rendered on a general-purpose computer using a technique developed at Pixar called stochastic sampling or distributed ray tracing.

Andre and Wally B.
a series of rendered images from the *Andre and Wally B.* film initially presented at the 1984 SIGGRAPH conference. These images may be used in conjunction with the *loop* Utility Program and *magtool* demonstration program to demonstrate frame-sequential animation.

Antenna Volume
a series of images that form a three-dimensional cube of data that may be viewed using the *cubetool* demonstration. The antenna volume represents the radiation pattern from a phased array antenna that was processed to represent a three-dimensional volume at Pixar.

Additional software that is developed at Pixar will be considered for inclusion in the contributed software classification. All customers are also encouraged to submit their demonstrations, programming prototypes, etc. for inclusion and distribution to the benefit of all Pixar users.

Conclusion

The Pixar Image Computer software provides a wide variety of development tools that allow image-processing and computer-graphics algorithms to be implemented easily and efficiently. The software allows use of the power of the Channel Processor(s), while retaining high-level programming constructs and sophisticated development tools. The flexible design of the Pixar Image Computer and powerful software provides a combination that can help customers discover the new possiblities image-computing brings them.

Pixar, Pixar Image Computer, and Pixar Development System are trademarks of Pixar.
Sun and Sun-3 are trademarks of Sun Microsystems, Inc.
UNIX is a trademark of AT&T.
Chap is a trademark of Pixar.

Pixar Hardware Overview

November, 1986

Introduction

The Pixar Image Computer hardware provides an innovative and powerful achitecture for performing image computing operations. The strength of the product is its integration of fast hardware and software libraries tailored for image computing. The software for the Pixar Image Computer provides a collection of tools to utilize the full capabilities of the hardware. These tools enable the user to perform a wide range of standard image computing operations quickly. They form a powerful development system that allows additional software to be easily developed.

This overview examines the hardware in a top-down fashion. See the software overview for details of the software that comes standard with a Pixar Image Computer.

Image computing is the combination of image processing and computer graphics. An image computer combines the capabilities of image analysis with image synthesis in a single machine.

System

The Pixar Image Computer uses a closely-coupled host computer to provide those functions not directly related to image computing. The host provides network access, a program development environment and general resources required by application programs. The Pixar Development System consists of a Pixar Image Computer, host computer, disk-based file system, an RGB monitor, tape back-up, modem (for diagnostics), and a rack. The host computer includes provides a bit-mapped display, keyboard, and mouse. All Pixar Development Systems include training credits for Pixar programming classes. Additions include extra memory, processor boards, monitors, disks, and tape drives. A typical system configuration is shown below.

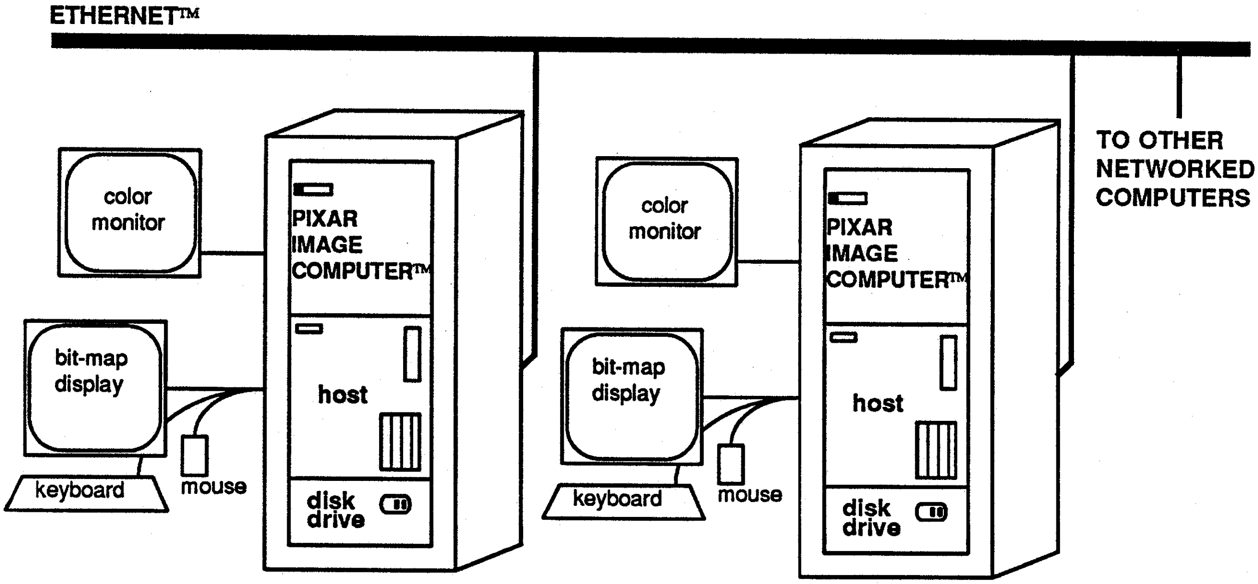


Fig. 1 Pixar Image Computer and host

The Pixar Image Computer comes with an extensive software package for developing applications. Pixar software is developed in-house under UNIX 4.2 in both C and assembly language. Pixar provides a development system and several sets of libraries as described in the Pixar Software Overview.

Pixar Image Computer Hardware

The Pixar Image Computer consists of a 21 inch-high, rack-mounted box with 12 board slots. The minimum system contains six boards: one CHAP, one Video, one Memory Controller, and three 8Mbyte memory boards. It is expandable to three CHAP, two Video and six memory boards. The host can be up to 30 feet from the Pixar Image Computer. The Pixar runs on 200-250 volts AC and will dissipate, worst case, almost 3KW.

The Pixar Image Computer Development System. consists of the Pixar Image Computer, the SUN 3/180™, tape drive, 380Mbyte disk, a high-resolution RGB monitor, bit-mapped display, SUN UNIX™, PIXAR software, a rack and all the necessary cabling.

The host connection is made through an interface card that plugs into the host's I/O bus. Fig. 2 shows an example using a SUN 3 host. The interface card connects the host bus, which might be VME, QBus, or Multibus, to the SYSBUS. The SYSBUS is a 16-bit 2Mbyte/second bandwidth bus over which instructions and data are sent to the PIXAR.

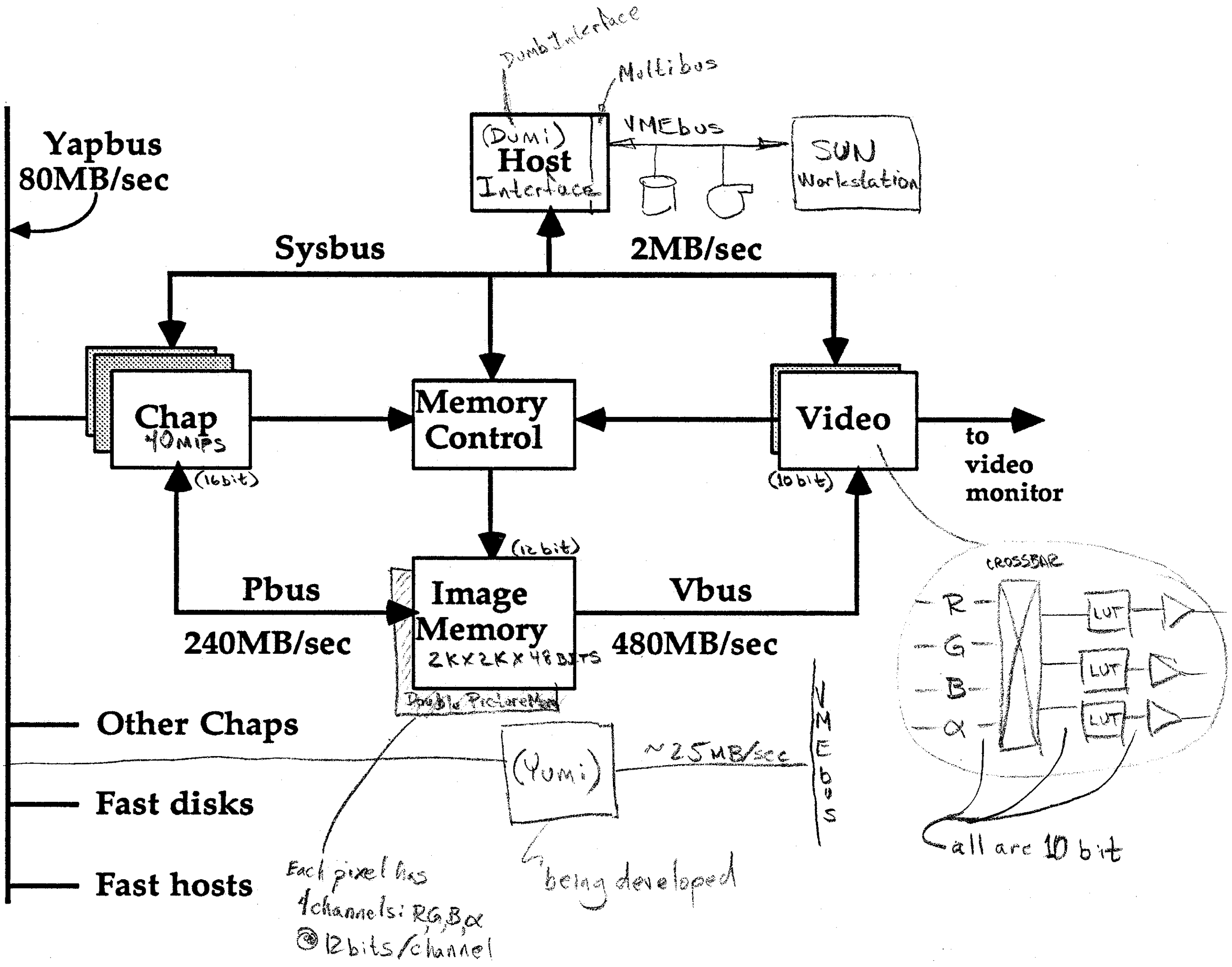


Fig. 2 System block diagram

As shown in Fig. 2 on the previous page, the YAPBUS offers an extremely high-bandwidth connection to a wide range of peripherals. This 80Mbytes/second bandwidth bus connects the Pixar Image Computer to other Pixar Image Computers, fast disks, and other high speed peripherals/computers.

The Pixar Image Computer primarily consists of a fast processor tightly coupled to a large memory. As shown in Fig. 2, the CHAP communicates with the memory over the PBUS. This 240Mbytes/second bandwidth bus provides the tight coupling between the memory and the processor. The CHAP processor receives instructions from the host computer, controls the YAPBUS, and computes on the data in the image memory.

The video board reads image data out of the memory to refresh the display. Data transfers from the memory to the Video board over the VBUS, which has a bandwidth of 480Mbytes/second.

The memory boards are 8Mbytes each, using 256K by 1 bit DRAMs. Each board is dual-ported, so that the Pbus and Vbus can have concurrent access to the 480Mbyte/second bandwidth of the memory system.

The memory controller receives requests for memory resources from the CHAPs or Video boards. It then arbitrates and schedules the data transfers when the appropriate resources are available.

The Architecture

Pixar's graphics software experience strongly influenced the design of the CHAP. This experience dictated several design objectives:

- Hardware support to facilitate programming
- Generality of hardware implementation
- Parallel nature of many image computing operations
- Efficiently manipulate the pixel data type
- Support multiple CHAPs in a system

The following technical description shows how these objectives were accomplished.

The Pixel Data Structure

All the components of a Pixar Image Computer are designed to accommodate the pixel data structure. Pixels are stored in memory as four 12-bit quantities, one each for the Red, Green, Blue, and Alpha components, or "channels", of a picture. Together, these four channels define the color (RGB) and transparency (Alpha) of any particular pixel in memory. Images are stored on the disk this way—as four separate pictures. This scheme allows the programmer to use the memory in many different ways. In medical imaging applications, for example, the four channels hold four monochrome pictures rather than a single furll-color picture. The Pixar's programmability makes different interpretations of the pixel data structure possible. The section on the memory will cover uses of the pixel data structure in more detail.

Chap

The CHAP has one processor for each of the Red, Green, Blue, and Alpha channels. The CHAP is a SIMD machine—for Single Instruction, Multiple Data. Four processors execute the same instruction on four values simultaneously. These four values can be the four components of a single pixel (RGBA), four entries in a table, the same component from four adjacent pixels, etc. Since each processor runs at 10 MIPS, the total speed is 40 MIPS. Control of the individual processors can be explicit but is normally transparent to the programmer.

Many computers use a Von Neumann architecture, in which instructions and data share the same memory. The CHAP is based on a Harvard architecture, with instruction memory separate from the scratchpad, or data memory.

The CHAP contains two types of hardware elements: vector and scalar. The vector elements are so named because there are four of each element: one for the Red, Green, Blue and Alpha channels. There is only one of each scalar element per CHAP.

Vector elements:

ALU
Multiplier
Pbus buffer
YAPBUS buffer
Scratchpad memory
Write crossbar
Read crossbar

Scalar elements:

Control unit
Address Generator
Program memory

The CHAP moves pixels from register to register via the two vector buses (Abus and Mbus). The scalar bus (Sbus) can receive a value from one particular vector element (e.g., the Red multiplier) or send a single value out to all. The processor clock has an 85 nanosecond cycle. All pixels follow the same path around the CHAP for a given loop in the program, with new pixels entering the path every couple of clock cycles. This is called pipelining—one of the keys to the CHAP's speed in processing large amounts of data. Figure 3 shows the main elements of the Chap from the programmer's point of view.

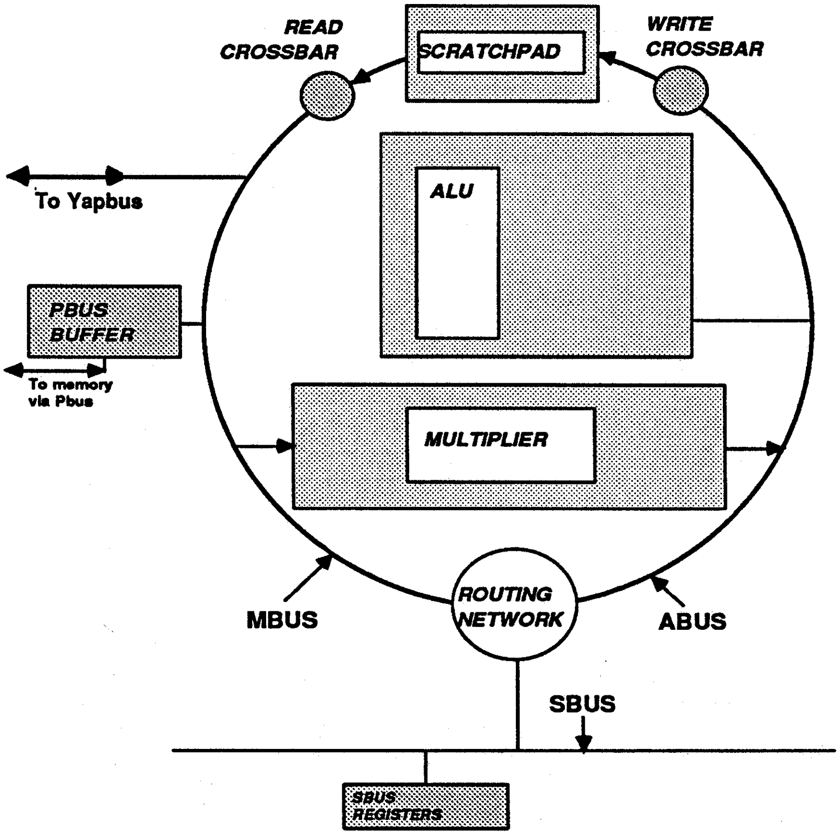


Fig. 3. Simplified Chap Programmer's Model

Vector Elements

ALU. The ALU (Arithimetic Logic Unit) is a 16-bit bipolar AMD 29116A chip, with 32 working registers, an accumulator, and a status register. A single data path connected to the Abus carries both incoming operands and outgoing results. Normally, all four ALUs execute the same instruction on the four components of a single pixel, but the hardware can automatically specify one or more ALUs not to participate, in which case those pixel components pass through untouched. This is useful, for instance, in processing the Red, Green, and Blue channels, while holding the Alpha channel unchanged. Four "Runflags" determine which ALUs will execute the current instruction. The runflags are simply a set of four bits, set by the evaluation of conditional statements in a program, that signifies which ALUs are enabled.

Multiplier. The 16-bit multiplier has a single input line and a single output line. A value from the Mbus goes to one of two input registers (the multiplicands), called MULTX and MULTY. Two ticks after the values A and B are put into MULTX and MULTY from the Mbus, the 32-bit product AB appears in the two output registers facing the Abus. The MSP register contains the most significant part of the product, while the LSP contains the least significant part.

Pbus buffer. The Pbus buffer moves pixels in bursts of 16 or 32 pixels between the CHAP and the picture memory. Four pixels are transferred at a time, using a 10 MHz transfer rate. The buffer has two sets of registers: one set facing the memory and the other set facing the CHAP. The pixels in the CHAP half of the buffer at any one time are said to be CHAPside, while pixels in the Pbus half are said to be Memoryside. In actuality, the Pbus buffer receives pixels one at a time from the CHAP, four at a time from the picture memory, but waits until it has either 16 or 32 to send a burst of (four pixel) Pbus transfers.

The datapaths and components in the CHAP are all 16 bits wide, to accommodate higher precision in intermediate products. Therefore, when a 12-bit quantity, such as a color component, comes into the CHAP from the picture memory, the Pbus buffer extends the 12-bit number to a 16-bit number using a special scheme. The special scheme involves extending the fixed-point representation of an integer from [0,1) to [-.5, 1.5), providing exact representations of zero and unity.

Yapbus buffer. The Yapbus buffer has an 80Mbytes/second bandwidth. The Yapbus transfers 64-bit quantities between CHAPs, or between a CHAP and a peripheral device.

Scratchpad. The scratchpad is a high-speed memory that stores groups of pixels during a program's execution. Though the scratchpad can hold 16K pixels (16 scan lines on a high-resolution monitor), it typically holds a scan line or two of pixels at a time. There are actually four scratchpad memories in the CHAP, just as there are four ALUs, making the scratchpad a vector component. Each scratchpad is 16 bits wide by 16K deep. For storing pixel data, scratchpad addresses can be tessellated to satisfy many different kinds of requests for component information. The crossbars and the address generator handle the details of tessellation, allowing the programmer to think of the scratchpad as a single, versatile pixel memory.

Write crossbar. The write crossbar is a destination on the Abus (see fig 3). It is the port through which data passes on the way to the scratchpad. The purpose of the write crossbar is to handle the details of tessellating the pixels in scratchpad, thereby providing four possible storage modes:

Pixel —Write values from the four Abuses (e.g., the four components of a single full-color pixel—R, G, B, A) separately to four scratchpad locations.

Component —Write the same component (e.g., Red) of four adjacent pixels in four scratchpad locations.

Broadcast —Write one Abus value (e.g., Red) into one scratchpad location.

Untessellated —Same as Pixel, only without tessellation.

Read crossbar. The read crossbar is a source on the Sbus. The read crossbar "untessellates" the pixels coming out of scratchpad, according to four possible access modes:

Pixel —Read four scratchpad values (e.g., the four components of a single full-color pixel—R, G, B, A) onto the Mbus.

Component —Read the same component (e.g., Red) of four adjacent pixels from four scratchpad locations.

Broadcast —Read a single scratchpad location to all four Mbuses, thereby sending out four copies of a single value.

Index —Read four scratchpad values from four explicit addresses supplied by the programmer.

Scalar components

The scalar components all reside on the Sbus. There is only one Sbus, or scalar bus, while there are four each of the Mbus and Abus. Through the Sbus a programmer can access the control unit, the address generator, the runflags, memory for communication with the host, and various other registers.

Control Unit. The Control Unit handles the details of processor execution and data flow within the Chap. Specifically, it allows the ALUs to execute based on the state of the runflags and distributes the signals to clock data into and out of computing elements, such as the multiplier. The control unit also contains several registers, such as the program counter, related to execution.

Program memory. The address space for the program memory is 16K words, each of which is 96 bits wide. With the exception of multi-line statements, each line of a Chap program translates into a single instruction word. The program memory can hold programs over 16,000 lines long. This memory is completely separate from scratchpad memory and is controlled as a separate resource by the software in the Pixar software release.

Picture Memory

Also called a frame buffer, the 24-Megabyte picture memory can store a 2048 x 2048 (48-bit pixel) image, and can be expanded to 48Megabytes. Though the address space is actually linear, the memory is conceptually divided into tiles, each of which is 32 x 32 pixels. A programmer can configure the memory to any rectangular shape (measured in tiles) to hold any image, whether in landscape, square, or portrait format. The picture memory is designed to facilitate the pixel data structure, though it is quite flexible in handling other data formats as well.

As mentioned above, the pixel data structure contains a fourth component, called the Alpha channel. The Alpha channel may contain arbitrary data as dictated by the software. One use of the Alpha channel is to include coverage (i.e., opacity—the opposite of transparency) information at each pixel. In this scheme, the alpha channel determines the degree to which a pixel obscures an existing pixel when one picture is laid on top of another. An alpha value of 1.0 corresponds to full coverage, while an alpha of 0.0 means the underlying pixel is to remain unchanged. For many applications, this use of the alpha channel makes it easy to do anti-aliasing, layering, filtering, touch-up, semi-transparent surfaces, and composition (matting and merging).

Other uses of the Alpha channel include storing four monochrome images in memory at the same time, providing Z-axis information (for volume applications), temporary scratch area, object tags, priority information, etc. The Alpha channel gives the programmer added flexibility in dealing with real-world complex data sets.

The memory is tessellated to give fast access horizontally or vertically. One memory access pulls a burst of 16 or 32 pixels. Each pixel contains 48 bits of data (12 bits each for the Red, Green, Blue, and Alpha channels). The access time to memory is three microseconds, but blocks of pixels can be read at a rate of four six-byte pixels every 100 nanoseconds. This is not the theoretical maximum, but the actual rate running real code.

Video board

Video features. The video board sends scan lines from the memory to the video monitor. Since the memory has more pixels than the monitor can display, the video board sends a subrectangle, called a Pixel Window, of the memory to the monitor. Two video boards may independently display any two rectangular portions of memory. Genlock circuitry allows video synch to coordinate the monitor's signal with other devices in the system. The video board has some special features, controlled by programs running on the host computer:

—Flexible formats (1024 x 768 interlaced, 525-line RGB, NTSC compatible, PAL)
—Hardware cursor, user-designed, 128 x 128 pixels
—Integer zoom
—Roam through the entire picture memory
—Colormap loading
—10 bits per DAC (Digital-to-Analog-Converter)

NTSC and Genlock

The Pixar Image Computer provides broadcast-quality video output for an NTSC encoder. A synch input locks the output video timing to other video signals. One genlock board is necessary for each video board in the Pixar Image Computer.

Memory Controller

The memory controller board makes data transfer flexible and convenient for the programmer. It coordinates data flow between the picture memory and its two customers: the CHAP and video board. When the CHAP or Video board makes a request for access to the memory, the memory controller arbitrates the request and allocates memory resources on a "first-come, first-served" basis. The memory controller is normally transparent to the programmer.

I/O for the Pixar Image Computer

Yapbus. A high-speed channel developed by Pixar connects the Pixar Image Computer to various peripheral devices and other computer systems. The channel, called the Yapbus (for "Yet Another Pixar BUS"), can transfer 64-bit data words at 10MHz, for a burst transfer rate of 80Mbytes/second. Control for the Yapbus is completely distributed, with a zero-collision transfer protocol guaranteed by a distributed arbitration scheme. Up to 16 devices may be connected on a single network.

Third party peripherals

Devices that will plug in to the SUN™ VME™ backplane can be used by the *Pixar Image Computer*. These include off-the-shelf frame grabbers, film printers, etc.

Conclusion

The Pixar Image Computer is a proven design, manufactured using state-of-the-art technology, that outperforms both general and special-purpose machines in image computing applications. Its large software library and multi-slot design make it flexible and expandable to meet new requirements as customers discover the new possibilities image computing brings them. The Pixar Image Computer is a complete system, developed and supported to meet the needs of those who demand the highest quality digital pictures.

The Pirl Tutorial:
Image Manipulation on the Pixar Image Computer

PIXAR

ABSTRACT

This document discusses **Pirl**, a library of subroutines for manipulating images on the Pixar Image Computer. It is a tutorial intended to enable programmers to invoke and combine basic image operations. We discuss the motivation for the package and where it fits into the Pixar software environment, then provide a series of examples showing how **Pirl** is used, concluding with pointers to lower-level packages which enable programmers to extend the **Pirl** library. In addition, this tutorial documents a number of graphics data types which are used widely in the Pixar software release. Accordingly, it is required first reading for new Pixar programmers.

November 18, 1986

Table of Contents

1. Introduction 1
2. Using **Pirl** 3
3. Getting Started: an Example Program 3
4. Compiling and Running *Pirl* Programs 5
5. A Functional *Pirl* Program 7
6. Using Different Tile Blocks 8
7. Making New Pixel Windows 9
8. Error Handling 10
9. The *RGBAPixelType* Data Type 11
10. Clearing a Pixel Window 12
11. Channel Masks 13
12. Displaying Pictures from Disk 14
13. Storing a Picture 16
14. Picture Formats 18
15. Merging Pictures: PirlMerge() 19
16. Writing into Pixel Windows I: Subrectangle Filling 21
17. Writing into Pixel Windows II: Line Drawing 24
18. Where To Go From Here 27

The Pirl Tutorial:
Image Manipulation on the Pixar Image Computer

PIXAR

1. Introduction

Pixar Software Overview provides an overall view of the Pixar software release: its structure, conventions and motivations. We begin by reviewing the basic concepts central to **Pirl**.

The three most important components of the Pixar Image computer are the image memory, the video board, and the Chap (**Chan**nel **P**rocessor). The video board does not concern us here. The Chap is the workhorse of the Pixar Image Computer; it contains the four AMD 29116 processors which justify the term *image computer*, together with instruction memory for storing programs and *scratchpad* memory for the use of these processors. The image memory is where the pictures are stored. The hardware of the image memory consists of a large number of **tiles**, each 32 pixels square. The tiles of the image memory are organized by the software into rectilinear regions known as *tile blocks* and *pixel windows*, with which **Pirl** is primarily concerned.

Most frame buffers have a 2-dimensional addressing scheme, with a fixed width and height in pixels. In contrast, the image memory of the Pixar Image Computer (hereinafter, the PIC) can be dynamically reconfigured to a nearly-arbitrary variety of aspect ratios. The basic unit of this organization is the tile. The basic PIC contains 4 million pixels, or 4 thousand tiles.

A **tile block** is a software entity organizing tiles into a two-dimensional structure for display. It consists of a contiguous series of tiles divided into a number of rows. The first tiles in a tile block (the first row) may contain, for example, the first 32 lines of an image, the second row the second 32 lines, and so on. Thus, on the standard PIC display monitor, a single tile block of 24 rows and 32 columns of tiles can be used for the 768-line, 1024-pixels-per-line display.

Once a tile block is defined, by allocating (*nrows* times *ncolumns*) contiguous tiles, a rectangular region of pixels (not necessarily a multiple of 32 in either dimension) can be allocated from the tile block anywhere within it. For a tile block which coincides with the display screen (as above), a number of pixel windows may be defined for smaller subregions of the display, forming in effect a number of various-sized "viewports" for accessing the display.

As another example, a tile block could be defined with 32 columns (the display width), and enough rows to fill the entire frame buffer (128 in a four-million-pixel PIC), with non-overlapping pixel windows, the size of the display, "stacked" vertically in the tile block, so that they could be successively put on display without moving pixels around.

The pixel window concept is the foundation of **Pirl**. A **pixel window** is a displayable, rectangular set of pixels in a tile block. If the pixels in a pixel window are considered as an image, then **Pirl** is an image-manipulation package; all functional **Pirl** routines operate on images. The basic idea is to provide a set of useful, simple operations on images so that they can be composed into more complex operations.

2. Using Pirl

Since **Pirl** is a collection of functions assembled into a library, the standard UNIX† documentation style is used. All routines discussed here are summarized more tersely in Section 3H of the Pixar Programmer's Manual.

The source of the tutorial examples discussed below are in the directory */usr/pixar/doc/tutorial/pirl*. To run them, you should create a directory of your own (referred to hereinafter as your working directory), then copy the files as they are discussed.

3. Getting Started: an Example Program

Figure 3-1 shows the complete source of an example program using **Pirl**, as contained in the file *lazybum.c*. It does nothing, but it does it right: it is the minimal correct **Pirl** program. Understanding it and getting it to run is your first exercise.

```
#include <pirl.h>

/* lazybum: a Pirl program which does nothing */
main ( )
{
    PirlBegin ( CHAP0, STD_TB );
    PirlEnd( );
}
```

Figure 3-1: *lazybum* Program Listing (*lazybum.c)*

All program files using **Pirl** pixel windows and defined constants must include the header file *<pirl.h>* in */usr/pixar/include.*

Since **Pirl** maintains a (simple) runtime environment, a program must initialize, and later deallocate, this environment. The former task is accomplished by the call to **PirlBegin**(), explained by the following digression:

In the Pixar software environment, tile blocks and pixel windows are fully dynamic objects, and a program can define several tile blocks, several pixel windows per tile block, and so on. Additionally, a given system can have several PICs attached to it, each PIC can include up to 3 Chaps, and tile blocks and pixel windows can be distributed at will among these (well, almost at will). However, managing all these structures is a significant burden on the programmer (and understanding it is a significant burden on the new Pixar programmer), and the flexibility is rarely useful. **Pirl** eases this burden considerably by imposing the following simplifying constraint: one PIC, one tile block. Both are maintained almost invisibly in the **Pirl** environment. In **PirlBegin**(), you say which Chap is to be used (**CHAP0** above), and you describe the tile block to be used. Thereafter, all operations use them automatically.

In the above call to **PirlBegin**(), **CHAP0** tells **Pirl** to use the first Chap of those attached to the first PIC running on the host. It is a token which allows other Chaps and

† UNIX is a Trademark of Bell Laboratories.

other PICs to be used, but no harm will come of always using **CHAP0** in the absence of good reasons to do otherwise. **STD_TB** simply specifies that **Pirl**'s tile block should be the one discussed above: 32 tiles wide, 24 high beginning at the first tile in image memory. Other possible values for this argument are **BIG_TB** and **HUGE_TB**, which are also 32 tiles wide but are 128 and 256 tiles high, respectively. The former is the largest tile block available on a standard PIC with 4 million pixels, while the latter will just fit on the largest available PIC (8 million pixels).

PirlBegin() also performs the auxiliary service of putting its standard tile block up on the display. The upper left pixel of the tile block is displayed at the upper left corner of the display.

The complementary call to **PirlBegin**() is **PirlEnd**(). In general, each will be called once during a program: **PirlBegin** at the beginning and **PirlEnd** at the end. Calling **PirlEnd**() at the end of a program is a matter of good citizenship, since it clears up some dynamic resources on the Chap which would otherwise persist and eventually require the Chap to be cleared with the not-so-friendly shell command **'chmap -i'.** You don't need to understand that sentence. Just make a habit of making **PirlEnd**() the last line in your program.

4. Compiling and Running *Pirl* Programs

Figure 4-1 shows part of a *makefile* (if this term is unfamiliar, go back to the UNIX Programmer's Manual) to compile *lazybum.* It illustrates common requirements for **Pirl** programs specifically, and Pixar programs in general.

```
CFLAGS = -g -I. -I/usr/pixar/include
LIBDIR = /usr/pixar/host/lib
LIBS = $(LIBDIR)/libpirl.a $(LIBDIR)/libpicio.a $(LIBDIR)/libpirl.a \
       $(LIBDIR)/libchad.a $(LIBDIR)/libpixar.a -lm -lg

all : lines fill getpic lazybum merge plaster savepic \
       skinny skinny2 testpat testpat2 wrong

lines : lines.o
       cc -o lines lines.o $(LIBS)

fill : fill.o
       cc -o fill fill.o $(LIBS)

getpic : getpic.o
       cc -o getpic getpic.o $(LIBS)

lazybum : lazybum.o
       cc -o lazybum lazybum.o $(LIBS)

merge : merge.o
       cc -o merge merge.o $(LIBS)

plaster : plaster.o
       cc -o plaster plaster.o $(LIBS)

savepic : savepic.o
       cc -o savepic savepic.o $(LIBS)

skinny : skinny.o
       cc -o skinny skinny.o $(LIBS)

skinny2 : skinny2.o
       cc -o skinny2 skinny2.o $(LIBS)

testpat : testpat.o
       cc -o testpat testpat.o $(LIBS)

testpat2 : testpat2.o
       cc -o testpat2 testpat2.o $(LIBS)

wrong : wrong.o
       cc -o wrong wrong.o $(LIBS)
```

Figure 4-1: *makefile* for tutorial

The overall directory structure of the Pixar software release is discussed in *Pixar Software Overview.* This makefile uses two of its principal directories: */usr/pixar/include*, as invoked by the **CFLAGS** macro, is the master directory for header files, used by practically all Pixar programs. In the case of *lazybum*, the compiler must look there to find *pirl.h.* /usr/pixar/host/lib (see **LIBDIR** above) contains all host program archives. Any program using **Pirl** be linked using three of these archives: *libpirl.a* is obvious. All **Pirl** programs use **Chad** code, contained in *libchad.a*, and **Chad**, in turn, uses **libpixar.a**.

If *lazybum.c* and *makefile*, both from */usr/pixar/doc/tutorial/pirl*, are now in your private directory, then make *lazybum* by telling the shell to 'make lazybum'. If all goes well, then you can run it with 'lazybum'. Run *lazybum*, but first do 'chmap -i' to re-initialize the Chap. This is the "hard reset" command to clear out all programs and data on the Chap, and perform other initialization. *chmap*, or any other **Pirl** program, may tell you that it can't open the device, that */dev/chap0* is busy. This indicates that another user has the Chap open. In such cases, all you can do is track this person down and ask them when they'll be finshed. Sometimes you will even have Chap processes in background which have the Chap open, in which case you'll have to ask yourself when you'll be finished.

When you start *lazybum*, you may see a series of messages telling you that certain modules of native Chap code are being loaded from various libraries. This is the Pixar runtime environment. Under various other shell programs, you will see other modules being loaded.

Run *lazybum* again, this time without doing 'chmap -i' first. The module-loading messages do not appear this time. This is Pixar's *dynamic loader* in action, or rather inaction: all the code of the runtime environment was loaded the first time through. Since the dynamic loader maintains a system-wide symbol table, it knows what modules are loaded, and where, so that any given module need only be loaded once, no matter how many times it is used.

When you run *lazybum*, you will notice that, although it may be doing nothing, it is doing it very slowly; it will probably take several seconds for it to return, even on an unloaded system, even the second time through with no modules being loaded. This overhead, which you will notice in any program using **Pirl** or **Chad**, is a constant which should become insignificant in long-running programs.

One final thing to notice about *lazybum*: its size. Do 'ls -l lazybum', and you will see an enormous (>400K) executable file. This should give you an idea how much software supports **Pirl**. This 400K, too, is pretty much a fixed cost of using **Chad**. **Pirl** is a relatively thin layer on top of this.

Depending on the setting of the environment variable CHAPDEBUG. If CHAPDEBUG=6, the functions names will be echoed on *stderr*. If CHAPDEBUG != 6, or does not exist, these messages will not appear.

5. A Functional *Pirl* Program

Now that you've made a **Pirl** program work, you can make one that does something. Figure 5-1 shows the source of *testpat.c*, which you should now copy from the tutorial directory */usr/pixar/doc/tutorial/pirl*. Obviously, the only difference between this program and *lazybum* is the line calling **PirlCbars** (). When you compile this program by saying 'make testpat' (which you should now do), then run it, you will see a broadcast color pattern on the display.

```
#include <pirl.h>

/* testpat: Use Pirl to put a test pattern up on the monitor */
main ()
{
        PirlBegin (CHAP0, STD_TB);
        PirlDisplay (ThePirlPW, 0, 0);
        PirlCbars (ThePirlPW, NORMAL);
        PirlEnd ();
}
```

which CHAP to use

which Tile Block

STD_TB = 1024 X 768

TB_DESCRIP(1024, 32, 24)

start tile, for 768 tile blocks

Define my own tile Block

which Pixel window

Figure 5-1: *testpat* Program Listing (*testpat.c*)

PirlDisplay() causes the pixel window given by its first argument to be displayed on the monitor with its upper left pixel at the top left of the display. The last two arguments, if non-zero, give an offset from this positioning: positive arguments move the display origin right and down in the pixel window.

PirlCbars() fills a pixel window with a broadcast test pattern.

This program may be boring, but it *is* informative. First, it illustrates the general form of a **Pirl** call: an operation aimed directly at a pixel window of image memory. Other operations may use more than one operand pixel window, and they frequently have arguments which are not pixel windows, but they always have at least one pixel window whose pixels will be modified.

The second interesting thing is that **PirlCbars** () and **PirlDisplay()** use **ThePirlPW.** This object, defined in *pirl.h* as type *PirlPW,* is **Pirl**'s default pixel window. It was created by **PirlBegin** () as a service to users who need only the simplest access to the pixels of the display. It will always coincide with the tile block specified by the second argument of **PirlBegin** (), which in this case is the entire display.

6. Using Different Tile Blocks

You might find that none of the three standard tile blocks given above suits your needs. Maybe an image in a file is exceptionally wide; maybe you want to assemble an animation of small images; maybe you would rather avoid overwriting a tile block which starts with the first tile by allocating one from further down the frame buffer; maybe you are simply of independent mind. For whatever reason, you *can* get **Pirl** to use a different tile block.

There are two ways to do this. You can use the **TB_DESCRIP** macro, as defined in *pirl.h*. Suppose you want a tile block 16 tiles wide and 24 tiles wide, beginning with tile number 1024. Then the program *skinny*, shown in Figure 6-1, will draw the previous broadcast color bars in that tile block.

```
#include <pirl.h>
#include <cbars.h>

/* skinny: Use Pirl to display a narrow test pattern */
main ()
{
    PirlBegin (CHAP0, TB_DESCRIP( 1024, 16, 24));
    PirlDisplay (ThePirlPW, 0, 0);
    PirlCbars (ThePirlPW, NORMAL);      /* Normal color bars */
    PirlEnd ();
}
```

Figure 6-1: *skinny* Program Listing (*skinny.c*)

Compile ('make skinny') and run the program. As promised, **ThePirlPW** fills the tile block, which is now half the width of the display, but just as tall. Note the strangeness of the display. If you block out the right half of the display, the bars should look 'right' (i.e., skinny).

> **Exercise**: Modify *skinny.c* to use a tile block, beginning with tile #0, which is 48 tiles wide and 24 high, then compile it and run it. These three instances of color bars and the way they are displayed should give you important clues to how the video scan out of the frame buffer actually works. Can you tell why the 'skinny' color bars look the way they do? Hint: consider that tiles in memory really are strung together in sequential addressing, and that, for the case of unmodified 'skinny', the first tile in the second row is number 1040 in the frame buffer.

7. Making New Pixel Windows

It is all very well and good to define novel tile blocks, but pixel windows are much more useful. Not only is it necessary to define more than one pixel window to use **Pirl** routines which have several operands, but pixel windows are convenient, in that they impose independent coordinate systems on the pixels of a tile block.

'skinny2.c' is a program that has ideas of what a pixel window should be, independent of **ThePirlPW**. It is shown in Figure 7-1. Note first that we are back to the original tile block **STD_TB**. What is really new here is the *PirlPW* **OurPW**. It is declared in the first line of **main** () and allocated immediately after the program enters the **Pirl** environment, by **PirlNewPW** (). This function takes five arguments: a pointer to the *PirlPW*, followed by the minimum and maximum coordinates in x and y of the pixel window, in the coordinate system of the tile block. As for all Pixar software, this coordinate space has its origin at the upper left corner, with positive values moving right and down. This pixel window, then has its upper left corner at the upper left corner of the tile block (0, 0), and is 512 pixels wide and 768 high. Note carefully the distinction between (for example) *xmax* and *xsize*: *xmax* = *xmin* + *xsize* - 1. Forgetting this relation can cause the most vexatious off-by-one errors.

```
#include <pirl.h>
#include <cbars.h>

/* skinny2: Use Pirl to display another skinny test pattern */
main ()
{
        PirlPW OurPW;

        PirlBegin (CHAP0, STD_TB);
        PirlDisplay (ThePirlPW, 0, 0);
        PirlNewPW(&OurPW, 0, 511, 0, 767);
        PirlCbars ( OurPW, NORMAL);          /* Normal color bars */
        PirlEnd ();
}
```

Figure 7-1: *skinny2.c* Program Listing

Now copy and compile 'skinny2.c'. Before you run it, though, run the program */usr/pixar/host/bin/clr* (if you haven't already done so, it will make your life much easier if you add 'usr/pixar/host/bin' to your Shell's search path) to clear the display. When you run 'skinny2', the left half of the display will have the same skinny color bars as in 'skinny'. But the right half of the display will remain dark. This is directly related to the fact the tile block is now the size of the display. Do you understand the video now?

Exercise: Now that you know how to declare one pixel window, modify parts of the tile block, then put color bars into them with **PirlCbars** (). **DIRE WARNING**: be absolutely certain that none of the pixel window coordinates lie outside the range [0,1023] for x-coordinates and [0,767] in y, for these are the boundaries of **STD_TB**. You don't know how to handle error conditions yet, so ignoring this stricture will surely cause your program to bomb. If you feel like it, though, you can do a null check on the value of **OurPW**.

8. Error Handling

All **Pirl** routines obey a simple convention: a return value of zero is normal; all other values are error codes. A diagnostic message for the nature of the error can be printed by calling **PirlErrReport()** with a file pointer to either *stderr* or *stdout* (or a pointer to any other file that's open for output).

PirlErrReport() tells you the source file in which the error occurred and the line number within that file. It also describes the nature of the error. Typically, a **Pirl** error condition occurs as a result of a Chad error (**Chad**, recall, is the lower-level package which **Pirl** uses). In these cases, you will see two error reports: first, the **Chad** error that caused the **Pirl** error, then the **Pirl** error itself. Frequently, there is no difference between the two errors. The location of the **Chad** is usually of academic interest only; you really want to know where the trouble lies in *your* source file(s).

The obvious method of using **PirlErrReport** () would be to include a call to it inside a conditional on the return value. However, this is not only tedious to type but tiring to read, and so *pirl.h* includes a macro, **CHECK()** which sets a global variable, **PirlLastErr** (also defined in *pirl.h*), and jumps to the label *error*, which can contain arbitrary error-handling code. The program *wrong.c* (see Figure 8-1) illustrates this. Note the **exit** () call which now appears; you don't want execution to fall through to the error-handling block after normal execution.

```
#include <pirl.h>
#include <cbars.h>

/* wrong: Try unsuccessfully to show a test pattern on the monitor */
main ()
{
        PirlPW OurPW;

        CHECK (PirlBegin ( CHAP0, STD_TB));
        CHECK (PirlDisplay ( ThePirlPW, 0, 0));
        CHECK (PirlNewPW ( &OurPW, 0, 1024, 0, 768));
        CHECK (PirlCbars ( OurPW, NORMAL));
        CHECK (PirlEnd ());
        exit (1);
error:
        PirlErrReport ( stderr );
        PirlEnd ();
}
```

Figure 8-1: *wrong.c* Program Listing

Run this program in your directory. The error here is the previously-noted off-by-one error in the declaration of the pixel window: we are asking for a pixel window 1025 pixels wide and 769 lines high from a tile block which is only 1024x767. Note that the error message tells on what line of the source file the error occurred.

Exercise: introduce error-handling code, as above, into the version of 'skinny2' which took multiple pixel windows. Modify it further to enter a loop, reading pixel window coordinates from standard input, declaring a pixel window and putting color bars into it. What happens when you open the 33rd pixel window?

9. The *RGBAPixelType* Data Type

The pixel window is not the only data type that **Pirl** routines take. In addition to the usual scalar, vector and matrix *ints*, *floats* and *doubles*, there is a special type, declared in */usr/pixar/include/pixeldef.h* as follows:

```
typedef short int PixelDataType;

typedef struct {
        PixelDataType Red;
        PixelDataType Green;
        PixelDataType Blue;
        PixelDataType Alpha;
} RGBAPixelType;
```

There is nothing mysterious about this structure, but it is pervasive, used throughout Pixar software to move pixel values around on the host and to the Chap.

Actually, there is *one* thing mysterious about *RGBADataTypes*. They are not actually integers to the Chap. Pixel values on the Chap run from -.5 to 1.5, so that 1 bit (of the 12 used for each channel of a pixel) is sacrificed to overflow prevention. The net result is that a 12-bit pixel contains 10 bits of fraction. Since it is probably not economical to modify the host to compute using this representation as a hardware data type, short integers are used, with conversion macros supplied between floating-point and pixel types.

Further information on Chap number representation, if needed, can be found in the **Chap Programming Tutorial**. From the viewpoint of host programming, though, it is sufficient to know about the existence of the macros **DBL2PXL** () and **PXL2DBL** (). The former takes a floating-point value (either **float** or **double**), converting it to *PixelDataType* (strictly speaking, it works for *ints*, too, but you have no business using integers in this context). The latter makes the opposite conversion.

10. Clearing a Pixel Window

Now that you are sick of color bars, we can use an *RGBAPixelType* for a new operation: filling a pixel window with a color. See Figure 10-1. This program should be obvious. One point, though: the **RGAPixelType** structure 'ThePirlPW' is passed by address, rather than value. This is universally true of Pixar software: no structures are ever passed directly. Rather, pointers to an instance of the structure are passed. Of course, this reveals that *PirlPW*s are actually pointers to structures.

```
#include <pirl.h>
#include <pixeldef.h>

/* fill: fill a pixel window with a color */
main ()
{
        float red = .7, green = .3, blue = .5;
        RGBAPixelType color;

        CHECK ( PirlBegin ( CHAP0, STD_TB));
        CHECK ( PirlDisplay( ThePirlPW, 0, 0));

        color.Red   = DBL2PXL ( red );
        color.Green = DBL2PXL ( green );
        color.Blue  = DBL2PXL ( blue );
        color.Alpha = DBL2PXL ( 1.0 );

        CHECK ( PirlClear ( ThePirlPW, &color ) );
        CHECK ( PirlEnd () );
        exit (1);
error:
        PirlErrReport ( stderr );
        PirlEnd ( );
}
```

Figure 10-1: *fill.c* Program Listing

An important digression: in the **Pixar Programmer's Manual** documentation for **Pirl** routines, pixel windows are declared as *ChadPW* pointers. The *PirlPW* data type is **typedefed** to a *ChadPW*. The only distinction is that the former are allocated through **Pirl**, and as a result **PirlEnd**() can release all pixel windows.

> **Exercise:** Modify *fill.c* to use a color that is passed to it, either on the command line or through standard input. The values given to **DBL2PXL** () should not be outside the range -.5 **up to but not including** +1.5. What happens when you give values outside the range 0 to 1.0?

> **Exercise: PirlSweepX** () takes a pixel window and an array of *RGBAPixelType*s, using the array as a scan line and sweeping it vertically in the pixel window. Modify *fill.c* again to fill such an array with some color pattern (note that the array must have as many elements as the pixel window has pixels in a scan line) and call **PirlSweepX** () or **PirlSweepY** (). Are you impressed with the speed of the PIC yet?

11. Channel Masks

There are situations in which you would want to write some subset of the channels. Three monochromatic pictures, for example, can be "stacked" in the red, green and blue planes of a single pixel window. The PIC allows the channels of a pixel window to be written independently by using a **channel mask,** a bit mask which restricts the subset of channels available for modification. The program in Figure 11-1 illustrates.

```
#include <pirl.h>
#include <pixeldef.h>
#include <cbars.h>

/* testpat2: show test pattern on monitor using channel mask. */
main ()
{
        float red = .7, green = .3, blue = .5;
        RGBAPixelType color;
        int mask;

        CHECK(PirlBegin ( CHAP0, STD_TB));

        color.Red = DBL2PXL( 0.0);
        color.Green = DBL2PXL( 0.0);
        color.Blue = DBL2PXL( 0.0);
        color.Alpha = DBL2PXL( 0.0);

        /* CHECK(PirlClear ( ThePirlPW, &color)); */

        printf ("color? ");
        switch(getchar()) {
        case 'r':
        case 'R':
            mask = REDMASK;
            break;
        case 'g':
        case 'G':
            mask = GREENMASK;
            break;
        case 'b':
        case 'B':
            mask = BLUEMASK;
            break;
        default:
            fprintf(stderr, "I don't know that color.\n");
            PirlEnd ();
            exit(-1);
        }

        CHECK(PirlSetChannelMask ( ThePirlPW, mask));
        CHECK(PirlCbars ( ThePirlPW, NORMAL));
        CHECK(PirlEnd ());
        exit (1);
error:
        PirlErrReport( stderr );
        PirlEnd ();
```

Figure 11-1: *testpat2.c* Program Listing

Clear the display using 'clr', then run 'testpat2' with the different channel masks. Note that channels other than the one in the mask are unaffected by the new color bars.

12. Displaying Pictures from Disk

You are probably interested in communicating pictures between image memory and files on external media, primarily disk files. This and the next two sections discuss using **Pirl** with the Pixar **picio** package for image storage and retrieval.

Figure 12-1 shows a program to read a picture from a file into a pixel window. The first new thing about this program is the header file *picio.h*, which defines the structure type *PFILE*. In the Pixar image file format, a picture file begins with a header block giving descriptive information about the file, which is copied into the *PFILE* structure when the picture is opened by **PicOpen**(), along with a regular **UNIX**-style file pointer. The program uses this information in the call to **PirlNewPW** () to open a pixel window the same size as the image on disk. After opening the pixel window, the program calls **PirlGetPic** () to read the image from the disk.

```
#include <pirl.h>
#include <picio.h>

#define PICFILE "andre.pic"

/* getpic: Read a picture from a file into a pixel window */
main ( )
{
    PFILE *pic;
    PirlPW OurPW;

    pic = PicOpen ( PICFILE, "r");
    if ( !pic ) {
        fprintf ( stderr,"Can't get picture file %s\n", PICFILE );
        exit ( -1 );
    }

    CHECK( PirlBegin ( CHAP0, STD_TB) );
    CHECK( PirlDisplay( ThePirlPW, 0, 0));

    CHECK( PirlNewPW( &OurPW, 0, pic->Pwidth-1, 0, pic->Pheight-1) );
    CHECK( PirlGetPic ( pic, OurPW, 0, 0 ) );

    CHECK( PirlEnd ( ) );
    PicClose ( pic );
    exit ( 1 );
error:
    PirlErrReport ( stderr );
    PirlEnd ( );
    if ( pic )
        PicClose ( pic );
}
```

Figure 12-1: *getpic.c* Program Listing

To run this program, you must (besides copying the source and *make*'ing it) copy the picture file *andre.pic* from */usr/pixar/doc/tutorial/pirl.*

PirlGetPic () takes four arguments: the *PFILE* picture-file pointer, the *PirlPW* which is the destination of the picture, and x and y offsets. This latter pair, here (0, 0), is most useful when the size of the pixel window differs from the size of the picture. The offset gives the number of pixels by which the picture is moved vertically and horizontally before writing the data into the pixel window. Thus, if the offset had been (-10, -10), the origin of the picture would be at (-10, -10) in the coordinate system of the pixel window, and so the first 10 rows and 10 columns of the picture would have been invisible.

> **Exercise:** Modify *getpic.c* to take six arguments from the command line: a rectangle defining the pixel window, and an offset for the picture. Experiment with various window sizes and picture offsets to get a feel for what they do.

> **Exercise:** Change the program further to call **PicFind** () instead of **PicOpen** (). The latter uses the Shell environment variable **PIXPATH** as a series of directories to search for the given file. Make your **PIXPATH** include */usr/pixar/doc/tutorial/pirl*, then delete *andre.pic* from your working directory and recompile and run *getpic* to get the picture again. **PIXPATH** enables your installation to keep good pictures accessible, while economizing disk space, by maintaining them in a set of standard places.

> **Exercise:** The program *gtinfo*, included in the Pixar Software Release, is a Shell program which summarizes the header on a picture file. Check out a few of the picture files in */usr/pixar/doc/tutorial/pirl* and use your modified *getpic* to look at them onscreen. Try tiling them on the display so you can see several at once.

13. Storing a Picture

Once you have an interesting display, you can try saving a picture. The program in Figure 13-1 does that:

```
#include <pirl.h>
#include <picio.h>

#define PICFILE "our.pic"
#define ALLREAD 664 /* mode for pic file */

/* savepic: Save a picture to a file on disk */
main ()
{
        PFILE *pic;

        PicSetPsize (1024, 768);
        pic = PicCreat(PICFILE, ALLREAD);
        if(!pic) {
            fprintf(stderr,"Can't create picture file %s\n", PICFILE);
            exit(-1);
        }

        CHECK(PirlBegin ( CHAP0, STD_TB));
        CHECK(PirlDisplay ( ThePirlPW, 0, 0));

        CHECK(PirlSavePic ( pic, ThePirlPW, 0, 0));

        CHECK(PirlEnd ());
        PicClose( pic);
        exit (1);
error:
        PirlErrReport( stderr );
        PirlEnd ();
        if(pic)
            PicClose( pic);
}
```

Figure 13-1: *savepic.c* Program Listing

There are three significant differences between this program and *getpic*. First, the program declares the size of the picture to be output with **PicSetPsize** (). Then, the picture file *our.pic* is created with **PicCreat** (). Finally, the picture is saved with **PirlSavePic** ().

The semantics of **PirlSavePic** () are as follows: the origin of the specified picture (the upper left pixel saved) comes from the pixel in the PW specified in the last two arguments. The rectilinear region of pixels which are actually saved is the *smaller* of 1) the area of the PW to the right and below the given pixel; and 2) the size of the image, as specified by **PicSetPsize** (). If this seems confusing, just accept that this arrangement does what you want: if you declare an image size that matches the PW size and give an offset of (0,0) to **PirlSavePic** (), then the whole PW is saved. If you want to save a portion of a PW, then you can create a picture file of the appropriate size using **PicCreat** (), and pass the offset to the desired region to **PirlSavePic** ().

You should make and run *savepic*, but only if you have lots of disk space on your file system. If you find that there is not enough space to store the whole display, you can reduce the size of the image in **PicSetPsize** () and/or open up a smaller pixel window.

> **Exercise:** Modify this program to write a smaller part of the display rather than the entirety of it as represented by **ThePirlPW**. Then use your modified version of *getpic* to place it in different locations on the display.

14. Picture Formats

savepic illustrates the protocol of picture creation: the parameters of the image are declared, then the picture is created. Since there are many options for picture storage, this mechanism is used to avoid having a long list of parameters to **PicCreat()** which rarely differ from the default. The following lists the characteristics associated with each image, the routines used to set them, and their standard values. Again, all of these routines must be called *before* **PicCreat** ().

- **image size:** The number of pixels of width and height of the image is set, as above, by **PicSetPSize** (). These are arbitrary positive integers, where 512 by 488 is assumed.
- **tile size:** Pictures can be stored as one monolithic block of pixels, or subdivided into *tile*s. **PicSetTsize** () specifies the size in x and y of the tiles used. It needn't divide the picture size evenly. 512 by 488 is the assumed tile size.
- **offset:** By default, the upper left pixel of an image has coordinate (0, 0). The function **PicSetOffset** () resets this origin, effectively offsetting the pixels of the image for all subsequent retrievals. This can be used to break a huge picture into a number of picture files.
- **format:** Several subsets of the red, green, blue and alpha channels of an image may be stored. This is most useful for 1) not wasting the space required to store the alpha information of pictures that don't use it, and 2) storing monochrome pictures. The format is set by passing **PicSetFormat** () one of **PF_RGBA, PF_RGB, PF_R, PF_G, PF_B** and **PF_A,** as defined in *picio.h.* By default, all four channels of an image are saved.
- **label:** Each file can have an ASCII string associated with it. This can be whatever information the creator of the file deems useful, up to 255 characters. Set by **PicSetLabel** ().
- **pixel format:** The pixel channel values of an image may be stored as either 8- or 12-bit quantities. Additionally, the image may be either **dumped** (each pixel is written individually) or **run-length-encoded** (horizontal series of identical pixels are written as a single pixel plus a replication factor. Dump mode is preferred for complex pictures with very little replication, since it saves the replication byte. The possibilities (again defined in *picio.h*) are **PS_8BIT** (8-bit runlength-encoded), **PS_12BIT** (12-bit ditto), **PS_8DUMP** (8-bit dumped) and **PS_12DUMP**. The default pixel format is **PS_8BIT**.
- **matting indicator:** The semantics of the alpha channel are recorded in the image using its *matting indicator*, which is set with **PicSetPMatting** (). The two possible values are **PM_MTB** ("matted to black") and **PM_NONE**. The former indicates that the red, green and blue values in the picture have been premultiplied by alpha to facilitate merging. The latter indicates that the color channels are uncorrelated with respect to alpha, and so the image is unsuitable for merging.

15. Merging Pictures: PirlMerge()

The **Pirl** functionality discussed up to now applied to the "visible" red, green and blue channels in the frame buffer. We now turn to the uses of the alpha channel for compositing images together. The program in Figure 15-1, *merge.c*, takes two images and uses their alpha, or coverage, information to merge them intelligently.

Note the new header file, *merge.h*, included after *pirl.h*. It is basically used to define **MergeOpOver**, the token which tells the operation **PirlMerge** () is to perform. *merge.c* also defines the *RGBAPixelType* variable 'RGBAOne'. Note that it is initialized using **DBL2COF**(), where before we used **DBL2PXL**(). The *coefficient* data in the Chap is similar to a pixel: it is also a fixed-point quantity, but with two more bits in the fraction. Coefficients are frequently used to represent scaling factors. 'RGBAOne' is initialized to the Pixar coefficient representation of 1.0 in all four channels, and its address is passed as the last two arguments to **PirlMerge** (). It is used by the function as factors for weighting the two input images *PW1* and *PW2*; the four channel values of each pixel in *PW1* are multiplied by the four channels of the *RGBAPixelType* pointed to by the fifth argument, and likewise for *PW2*. We use 'RGBAOne' here to indicate that both PWs are used with their full weight, but it should be obvious how to make, for example, a "ghost image" of either of the two inputs.

Note the check that the two input images are identical in size. This is a requirement for **PirlMerge** (), and it is an error for them to differ.

When you run 'merge', note the difference in time between reading the two images from disk, versus the speed at which merging occurs. This is a fair benchmark of the Chap's speed compared to image input; merging is a fairly compute-intensive operation, yet it happens **much** faster than I/O.

There are 14 merge operations used to combine two images into a third. There are three sources of explanation about these operations: The manual page for **PirlMerge** () in Section 3H of the **Pixar Programmer's Manual** gives a terse explanation which will enable you to experiment with the various options. There is a more generous explanation in **Compositing Digital Images**, found under **Advanced Topics** in the first volume of the **Pixar Programmer's Manual.** The definitive reference, though, is the paper **Compositing Digital Images**, by Tom Porter and Tom Duff, in **SIGGRAPH '84**, from which the former is derived. The **SIGGRAPH** paper includes color illustrations.

> **Exercise**: copy *merge* into your working directory, then *make* and run it (remember to copy *genesis.pic* as well). Once it works to your satisfaction, modify *merge.c* as follows: declare a local *RGBAPixelType* variable, say 'RGBAFrac', then enclose the call to **PirlMerge** () in a **for** loop from 0 to 10, which sets the four channels of 'RGBAFrac' to (**DBL2PXL**(i/10.0)) before calling **PirlMerge** () with the address of 'RGBAFrac' as the fourth argument. The net effect should be to fade the foreground in over the background. If you feel like it, you can go on to more elaborate stunts like fading the channels in independently.

> **Exercise**: modify *merge.c* to take a command-line argument for a merge operator, and experiment with the various operations, so that you can get a feeling for what they do.

```
#include <pirl.h>
#include <picio.h>
#include <merge.h>
#include <pixeldef.h>

#define PIC1 "andre.pic"
#define PIC2 "genesis.pic"

RGBAPixelType RGBAOne =
    { DBL2COF(1.0), DBL2COF(1.0), DBL2COF(1.0), DBL2COF(1.0) };

/* merge: Merge two pixel windows into a third. */
main ( )
{
        PFILE *pic1, *pic2;
        PirlPW PW1, PW2;
        int pwidth, pheight;

        pic1 = PicOpen ( PIC1, "r" );
        if( !pic1 ) {
            fprintf ( stderr,"Can't get picture file %s\n", PIC1 );
            exit ( -1 );
        }
        pwidth = pic1->Pwidth;
        pheight = pic1->Pheight;

        pic2 = PicOpen ( PIC2, "r" );
        if ( !pic2 ) {
            fprintf ( stderr,"Can't get picture file %s\n", PIC2 );
            exit ( -1 );
        }
        if ( (pic2->Pwidth != pwidth) || (pic2->Pheight != pheight) ) {
            fprintf ( stderr,
"merge size mismatch: %s is %d by %d, but %s is %d by %d\n",
               PIC1, pwidth, pheight,
               PIC2, pic2->Pwidth, pic2->Pheight );
            exit ( -1 );
        }

        CHECK ( PirlBegin ( CHAP0, BIG_TB ) );
        CHECK ( PirlDisplay ( ThePirlPW, 0, 0));

        CHECK ( PirlNewPW ( &PW1, 0, pwidth-1, 0, pheight-1 ) );
        CHECK ( PirlGetPic ( pic1, PW1, 0, 0 ) );

        CHECK ( PirlNewPW ( &PW2, pwidth, pwidth*2-1, 0, pheight-1) );
        CHECK ( PirlGetPic ( pic2, PW2, 0, 0 ) );

        CHECK ( PirlMerge ( PW1, PW2, PW2,
                          MergeOpOVER, &RGBAOne, &RGBAOne ) );

        CHECK ( PirlEnd ( ) );
        exit ( 1 );
error:
        PirlErrReport ( stderr );
        PirlEnd ( );
 }
```

Figure 15-1: *merge.c* Program Listing

16. Writing into Pixel Windows I: Subrectangle Filling

You now know how to create pixel windows, modify them whole, and move them to and from disk files, but we have not provided a capability for finer control, namely how to read and write a subset of the pixels in a pixel window from the host computer. The capabilities of **Pirl** for doing this are limited to 1) filling a rectangle in a PW with the contents of two-dimensional buffer (described in this section) and 2) line drawing. The principal reason for this limitation is that the communication channel (the **Sysbus**) between the host and the PIC is relatively slow (approx. 2 Mbytes/second), **especially** when compared with the speed of the Chap. Consequently, if you use the host to generate pixels and simply use the PIC as a display device, then it is not being used up to its capabilities. If the host is really being used to generate pictures, it is a generally more efficient use of the PIC to generate the images into a disk file, and use (say) **PirlGetPic**() to display them.

(Actually, a little reflection will demonstrate that these restrictions aren't so bad, after all. Single-pixel reads and writes *can* be accomplished, if need be, by using a degenerate rectangle 1 pixel on a side.)

Figure 16-1 shows a program, *plaster.c*, for writing a series of random rectangles into a pixel window.

plaster generates randomly-colored, random-sized rectangles and places them at random locations in the display, continuing until Return is typed at the keyboard (if you redirect standard input, it only generates one rectangle). It looks for input, without waiting for it to occur *a la* **getchar** (), by using the **ioctl** () function to determine the number of characters waiting on input, quitting when this number is non-zero.

PirlPutRect() takes as arguments a pixel window, a buffer filled with pixel values, and minimum/maximum value pairs for x and y. These last represent the rectangle to be written from the rectangle buffer.

Note that we could have filled the buffer by using the code fragment

```
for(x = 0; x < xsize; x++)
    for(y = 0; y < ysize; y++)
        PxlBuf[y][x] = color;
```

We could have, but it would have been wrong unless *xsize* was equal to **MAXWID** (defined at the beginning of *plaster.c*. The reason is that **PirlPutRect**() expects a two-dimensional buffer of x- and y-dimensions equal to the rectangle specified in its arguments. Since we can't redeclare the *PxlBuf* buffer to the appropriate dimensions each time the rectangle is re-randomized, we use C's loose typing rules to fake a two-dimensional buffer with a 1-dimensional one. Any C reference book should contain enough information to enable you to do this.

When you run *plaster*, you'll observe that the speed is indifferent. This is because it is loading a full buffer of pixels (rather than passing a single pixel and telling the Chap to fill a rectangle with it). This gives you an idea of the rate of data-loading into the Chap, since the code to do this buffer loading is fairly optimal.

Note the **#undef** statement on line 3 of *lines.c*. This is an (admittedly ugly) patch to a problem with **Chad**, hence with **Pirl**: a conflict with the definition of **B0** in the system's *ioctl.h* header file. In this case, as in most cases, the conflict is resolved by undefining it before including the latter header file. Of course, if **B0** is actually *used* in

```
#include <pirl.h>
#include <math.h>
#undef B0
#include <sys/ioctl.h>
#include <pixeldef.h>

double drand();

#define MAXWID 100
#define min(a,b)  (((a)<(b))?(a):(b))

RGBAPixelType PxlBuf[MAXWID][MAXWID];

/* plaster: write random rectangles to a pixel window */
main ( )
{
    RGBAPixelType color;
    register RGBAPixelType *pxlptr;
    int xloc, yloc, xsize, ysize, awaiting, npxls,
       pwminx, pwmaxx, pwminy, pwmaxy;

    CHECK ( PirlBegin ( CHAP0, STD_TB ) );
    CHECK ( PirlDisplay ( ThePirlPW, 0, 0));
    pwminx = ThePirlPW->xmin;
    pwmaxx = ThePirlPW->xmax;
    pwminy = ThePirlPW->ymin;
    pwmaxy = ThePirlPW->ymax;

    srand ( 1 );
    color.Alpha = DBL2PXL ( 1.0 );
    do {
        xloc  = pwminx + rand() % (pwmaxx-pwminx+1);
        yloc  = pwminy + rand() % (pwmaxy-pwminy+1);
        xsize  = rand() % MAXWID;
        xsize = min(xsize, (pwmaxx-xloc)); /* clip to ThePirlPW */
        ysize  = rand() % MAXWID;
        ysize = min(ysize, (pwmaxy-yloc)); /* clip to ThePirlPW */

        color.Red   = rand() % 2048; /* 2048 is the max pixel value */
        color.Green = rand() % 2048;
        color.Blue  = rand() % 2048;

        pxlptr =  PxlBuf;
        npxls = xsize * ysize;
        while (npxls--)
            *(pxlptr++) = color;

        CHECK( PirlPutBuf ( ThePirlPW, PxlBuf,
                          xloc, xloc+xsize-1, yloc, yloc+ysize-1 ));

        /* Get the number of characters waiting at input */
        ioctl ( fileno(stdin), FIONREAD, &awaiting );
    } while (!awaiting); /* Continue until anything is typed */

    CHECK ( PirlEnd ( ) );
    exit ( 1 );
error:
    PirlErrReport ( stderr );
    PirlEnd ( );
```

Figure 16-1: *plaster.c* Program Listing

the **Chad** sense in the source file, this fix breaks down (and rather badly, at that). However, you will virtually never use it in the course of using **Pirl** exclusive of **Chad.**

> **Exercise:** Since the dimensions of the rectangles passed to **PirlPutRect()** can't in general be predicted at compile-time, you should satisfy yourself that you can manipulate a one-dimensional array as if it were 2-d. When you have done this, modify *plaster.c* to draw a diagonal line between opposite corners of the pixel buffer before passing it down to the Chap.

> **Exercise: PirlGetRect()** is the symmetric opposite of **PirlPutRect()**. Modify the original *plaster.c* so that, instead of filling the rectangle buffer with a single color, it pulls the buffer out of the frame buffer using **PirlGetRect()**, and writes it into a second random location. Note that, for writing, the size should not be reset (so that the same rectangle is written as was read), and don't forget to make sure that the output rectangle fits into *ThePirlPW* (that is, make sure that (xloc+xsize < TBX) and (yloc+ysize < TBY)).

17. Writing into Pixel Windows II: Line Drawing

The **Pirl** library contains a very general package for drawing lines into frame-buffer memory on a PIC, with complete anti-aliasing. A simple program to do so is seen in Figure 17-1.

```
#include <pirl.h>

#define MAXNPTS 50
int LineNum = 0, LinesOpen = 0;
typedef struct { float x, y; } FloatPt;

/* lines: Read a script to draw lines into the display */
main ()
{
    PirlPW OurPW;
    char cmd[80];
    FloatPt points[MAXNPTS], polygon[MAXNPTS], offset;
    int npts = 0;

    CHECK (PirlBegin ( CHAP0, STD_TB));
    CHECK (PirlDisplay( ThePirlPW, 0, 0));
    CHECK (PirlBeginLines ( ) );
    LinesOpen = 1;
    while ( scanf( "%80s", cmd) == 1) {
        switch ( cmd[0] ) {
        case 'm':       /*  %f %f: Re-initialize sequence at point */
            npts = 0;
        case 'l':       /* %f %f: Add point to sequence */
            if ( npts < MAXNPTS )
                getpt(&(points[npts++]));
            else
                fprintf(stderr,"WARNING: too many points in polygon\n");
            break;
        case 'd':       /* %f %f: Draw with offset */
            getpt(&offset);
            offsetpoly(points, &offset, polygon, npts);
            PirlPolyLine(ThePirlPW, npts, polygon);
            break;
        }
        LineNum++;
    }
    CHECK (PirlEndLines ( ) );
    CHECK (PirlEnd ());
    exit (1);
error:
    PirlErrReport (PirlLastErr);
    barf ( );
}
```

Figure 17-1a: *lines.c* Program Listing (part 1)

Like **Pirl** itself, the line-drawing package maintains an environment, entered with **PirlBeginLines**() and exited with **PirlEndLines**(). The lines are drawn by the function **PirlPolyLine**(), which takes as arguments a pixel window, a count, and an array of 'count' x/y pairs. It draws ('count'-1) connected line segments, beginning at the first point and ending at the last.

lines reads standard input, expecting any of three commands: the point to the series;

```
offsetpoly(points, offset, polygon, n)
FloatPt *points, *offset, *polygon;
int n;
{
    while ( n--) {
       polygon->x = points->x + offset->x;
       polygon->y = points->y + offset->y;
       polygon++; points++;
    }
}

getpt(pt)
FloatPt *pt;
{
       if(scanf("%f %f", &(pt->x), &(pt->y)) != 2) {
           fprintf(stderr,"Bad input format on line %d\n", LineNum);
           barf ( );
       }
}

barf ( )
{
       if ( LinesOpen )
           PirlEndLines ( );
       PirlEnd ();
}
```

Figure 17-1b: *lines.c* Program Listing (part 2)

and the 'draw' command causes the current series to be drawn into the pixel window.

> **Exercise:** *make* 'lines' and run it using the script file *octagons.l.* Make a script file of your own to do some line drawing.

> **Exercise:** There is no checking done to make sure points fit inside the pixel window. Add a routine, called after **offsetpoly** (), to clip the points to the pixel window boundary.

The line-drawing environment referred to above includes four attributes of lines, which are initialized by **PirlBeginLines** () and can be reset while the environment is open, to wit:

- **color:** The color of the lines, initially white, is set with **PirlSetLineColor**(), by passing it a pointer to an **RGBAPixelType**.
- **mode:** The mode in which lines are overwritten into the pixels of the frame buffer is controlled by calls to **PirlSetLineMode**(), which is passed a *PirlPlaceMode* value from *<pirl.h>*, one of:

  PM_MERGE: merge the pixels of the lines over the values already in each pixel, according to the coverage of the line in the pixel.

  PM_REPLACE: replace pixels with the matted-to-black values of the line, regardless of the coverage the line has over each pixel.

  PM_MAX: use the maximum of the line value and the existing pixel value.

The drawing mode is **PM_REPLACE** by default.

- **width:** The width of the lines, initially 1.0 pixels and limited to positive values less than 256, is set by passing **PirlSetLineWidth**() a floating-point value.
- **edge:** The "profile" of the lines drawn by **PirlPolyLine** ( ) controlled by **PirlSetLineAttributes** ( ). This function has many arguments, but the edge description basically has two characteristics: the function describing the "profile function", the falloff of the line's intensity from its center (termed the **edge** characteristic), and the function used for anti-aliasing the line during rasterization (the **filter** characteristic). Each of these two is specified by a token, the width of the line, and a function pointer. The edge token is one of

  PE_GIVEN
  PE_HARD *(default)*
  PE_RANDOM
  PE_FELTTIP

  The filter token is one of:

  PF_GIVEN: the filter is specified by a table, specified by the *filterwidth* and *filterfunctionptr* parameters. The former gives the number of entries in the table (the half-width of the filter, since it is assumed symmetric), and the latter a falloff function: each member of the array denoted by *filterfunctionptr* is an attenuation factor between 0 and 1.
  PF_SINC: the filter is a sinc (*sine*(x)/x)
  PF_NONE: no filter is used
  PF_BOX: a box filter (1 inside its width, 0 elsewhere) *(default)*

Note that **PirlSetLineAttributes** need only be called if any other than the default edge descriptor (PE_HARD, PF_BOX) is desired.

> **Exercise**: Modify *lines.c* to accept commands which control the four line characteristics, then experiment. Try drawing different-colored lines of different widths. Try redrawing lines over themselves with different modes. See what the different edge descriptors do.

18. Where To Go From Here

By now, you should be fairly fluent in using **Pirl**. More important, you should have a good understanding of its capabilities and limitations. The Appendix lists the set of functions in the current **Pirl** library; it is the first place to look before you try to go beyond what has been covered here.

Defining the operations in **Pirl** is an ongoing process. In addition to future Pixar software releases, which will attempt to provide generally-useful functionality, the Pixar software development environment encourages programmers to add new functions to **Pirl** by providing access to the tools with which **Pirl** is built. Among these, the logical one to explore after **Pirl** is **Chad**.

Chad is designed to support the process of developing and running programs programs for the PIC by managing the its Chap processor, allocating and maintaining resources, setting registers and loading and running programs with one easy-to-use protocol. **Pirl** is essentially a special-purpose front end to **Chad**, and so **Chad** is the answer to the limitations of **Pirl** as discussed in this tutorial: learn and use it when you need more than one tile block, use more than one Chap, or read and write the PIC's image memory directly. **Chad** is introduced in *Programming with Chad.*

Chad gives the programmer full access to all existing libraries of Chap programs and routines. The current Pixar development environment includes the assembler, **chas**; a dynamic loader that loads Chap functions as programs require them, an interactive debugger, **charm**; and an archive maintainer, **chranlib**, in the spirit of the **UNIX** program *ranlib.*

Programming the Pixar Image Computer With *Chad*: A Tutorial Introduction

PIXAR

ABSTRACT

The essential introduction to general purpose programming on the Pixar Image Computer. **Chad** provides a simple environment for writing host programs which run programs on the Channel Processor, providing more general access to the programming power of the Chap, and making it feasible to develop programs on the host and move them to the Chap with a minimum of pain and aggravation.

November 18, 1986

Table of Contents

1. Introduction 1
2. Essential Hardware 3
3. Basics of *Chad* 3
4. Nuts and Bolts 5
5. The *Chad* Environment 6
6. Error Handling 7
7. Resource Allocation: *ChadAlloc*() 8
An Example: PirlBegin() 10
8. Freeing Resources: *ChadFree*() 11
9. Leaving *Chad* 12
10. Writing to Resources: *ChadWrite*() 13
11. Executing Chap Programs: *ChadGo*(), etc. 15
PirlClear() 16
12. Determining a Resource's Chap: *ChadOwner*() 18
13. Repeated Execution of *Pirl* Routines 19
14. Resource Reallocation: *ChadCheck*() 20
15. Reading from Resources: *ChadRead*() 21
The Reliability of *ChadRead*() Values 22
16. More Resource Deallocation: *ChadReset*() and *ChadBackup*() 24
17. Do-it-yourself Synchronization: *ChadCPUBusy*() and *ChadWaitCPU*() 26
18. *Chad* Internals for the Curious 27
19. Further Explorations 28
20. Appendix: Summaries of Routines 29
ChadBegin() 29
ChadEnd() 29
ChadAlloc() 30
ChadCheck() 30
ChadFree() 30
ChadReset() 30
ChadBackup() 31
ChadLibs() 31
ChadWrite() 32

ChadRead() 32
ChadGo() 33
ChadCPUBusy(), ChadCPUWait() 33
ChadOwner() 34
ChadErrReport() 34

Programming the Pixar Image Computer With *Chad*: A Tutorial Introduction

PIXAR

1. Introduction

Like a high-powered sports car, the principal design goal of the Pixar Image Computer (henceforth PIC) is speed. Occasionally, you may miss the cigarette lighter or AM-FM tape deck found in the family sedan, but for getting from 0 to 60 as fast as possible, it has no equal.

Writing programs to run on the PIC which get the most from its pipelined, parallel architecture is a different experience compared to programming in conventional languages. The design decisions going into the Chap have opted for speed and flexibility at a cost in programmer responsibility. The programming environment called **Chad** is designed to aid a number of strategies for making programming the Pixar more pleasant and productive in the context of a host software environment.

One way of introducing the power of the PIC into conventional programming environments has been presented in **The Pirl Tutorial. Pirl** is a system for manipulating rectilinear regions of frame buffer memory from the host without ever considering the PIC except to apportion frame buffer memory. However, **Pirl** is not designed to be a universal solution, and so some system must provide closer control over the PIC.

A second strategy for using the Chap is to use the substantial body of software already written for it. This strategy requires only a smooth means for host programs to control Chap programs.

A third strategy for smoother Chap programming applies equally well to other special-purpose hardware: use it the way it's intended, primarily for those repetitive, computation-intensive tasks which wreak havoc on most general purpose computing engines. In other words, use it for inner loops. Many such are already written and available (the Pixar Software Release contains more than 60,000 lines of Chap code).

Finally, there is the migration strategy: debug algorithms by host programming, then migrate code to the Chap as the inner loops make themselves known. The motivations and benefits for this strategy (and its cost) are similar to machine-language programming on a host. The time spent by the CPU is wildly disproportionate to the total amount of code involved in a comparison with the program as a whole.

This document is a tutorial on the Pixar host-interface environment **Chad**, which is a tool for implementing all these strategies. The prerequisite for making sense of this tutorial are familiarity with the UNIX† operating system and the C programming language. You should also have read at least a large portion of **The *Pirl* Tutorial** to

† UNIX is a Trademark of Bell Laboratories.

familiarize yourself with the Pixar software environment, at least to the extent of understanding backreferences to it. A reading of *Pixar Software Overview* would also be helpful, although the following section explains the essential aspects of the PIC.

Every work of literature requires a theme, and this manual is no exception. The theme here is **Pirl**. **Pirl** is built using **Chad**, and an explication of the former's internals is an excellent way to get to know the latter. Consequently, most of the code examples are from working **Pirl** routines, and there are frequent digressions comparing the **Pirl** universe to that of **Chad**. At the end, you should have a nearly complete understanding of how **Pirl** operates, plus the ability to extend it by using **Chad**. Finally, you will have a toolkit which enables you to go on and develop full-blown Chap programs by using **The Chap Programming Tutorial**. We finish with an Appendix which condenses the information presented into a quick-reference format.

2. Essential Hardware

We are concerned here primarily with the **Chap** (**Channel Processor**), the portion of a Pixar with which the host deals in moving image data to and from image memory, and which operates on that data using four 10 MIPS processors, a 16K array of instruction memory (referred to herein as *RAM*) for controlling them, and 64K 16-bit words of auxiliary, or *Scratchpad*, memory, plus such supporting devices as an elaborate bus structure and four 16-bit multipliers. Each processor, or *ALU* for short, includes a bank of 32 ALU *registers*. The scratchpad memory contains 16 *base registers* and 16 *index registers*, the distinction between which may remain obscure for the time being. Note that the ALU registers are quadruple, since there are four processors per Chap, but the base and index registers are scalar, since values are fetched from the scratchpad (generally) using a single address.

The Chap is a **SIMD** (single-instruction, multiple-data) processor: all four ALU's on a Chap execute the same instruction, but the data on which they operate is different; all four processors may read simultaneously from four different locations in scratchpad, which is organized such that, when a bank of locations contains successive sets of four values each (for the Red, Green, Blue and Alpha values of a pixel), the four processors may read:

-- each from the same location (*broadcast* mode)
-- from the Red, Green, Blue and Alpha channels of a pixel (*pixel* mode)
-- from Red (or Green, etc.) in four successive pixels (*component* mode)
-- from a table indexed independently by the individual channel values of a pixel (*index* mode)

The interface between the host and a Chap consists, for our purposes, of 16 Registers which are mapped into the memory of the host: the *sysbus* registers, whose name refers to the Chap's system bus. There is also the notion of the *Virtual Data Registers*, or *VDR*s. On the host side, these appear as an array of 256 memory locations. The implementation is in fact quite different, but the distinction is irrelevant and confusing for now.

3. Basics of *Chad*

Once an image is available on the PIC (that is, a pixel window *a la* **Pirl** has been allocated and filled with the appropriate pixel values), dozens of routines can be called for altering that image, and they all use the same paradigm. Namely, for each scan line in the image:

1: repeatedly read the scan line from the image into scratchpad memory.
2: perform necessary processing on the pixels of the scan line in parallel.
3: write the scan line out to image memory.

Of course, exceptions abound to this generalization, but it is generally much more efficient to block-load pixels into scratchpad, operate on them there, then block-write them back into image memory. This tells us what information is required for a Chap program to do its job:

- the code for the program itself must be loaded into RAM.
- the operating buffer in scratchpad must be set aside, and the Chap program told where it can be found.
- other parameters, such as the location and size of the image, must be passed to the

program.

The process involved in those three steps is, in a nutshell, what **Chad** is designed to support. The Pixar Software Release includes a set of libraries of Chap code. Routines and programs in this library can be retrieved by name and, given that name, **Chad** invokes a dynamic loader to load the code during execution of the host program. Similarly, **Chad** handles requests to allocate blocks of scratchpad memory for buffers or tables, and will load them with host-generated data if necessary. Since Chap routines usually take their parameters in the base, index, ALU and sysbus registers, **Chad** provides translucent access to these registers for reading and writing.

Executing a Chap program thus reduces to five basic tasks, each performed by a single routine:

1) Allocate and load program RAM and scratchpad buffer space (performed with **ChadAlloc()**).
2) Write any tables in scratchpad and registers required by the routine (**ChadWrite()**).
3) Run the routine and wait for it to return (**ChadGo()**).
4) Read any output tables or return registers (**ChadRead()**).
5) Free scratchpad buffers, and maybe instruction RAM too (**ChadFree()**).

The is the basic functionality around which the rest of **Chad** revolves. Most of the rest of it is motivated by a consideration we have not yet mentioned. The 64K of scratchpad memory is a common bottleneck: when it runs out, **Chad** fails and programs don't run. This is not just a problem within a single process. The dynamic loader which loads Chap code and parcels out scratchpad memory is a global, systemwide mechanism. This means that when one process uses up the scratchpad, nobody can get to it. The same problem can theoretically occur with instruction memory, but in practice it virtually never does.

The usual way of dealing with out-of-memory problems is to reset the Chap, clearing both its instruction and scratchpad memories. While less than elegant, this method *is* effective. However, it implies that some way must be devised of "revoking" the resources **Chad** had allocated, so that a running program can efficiently verify the continued survival of its resources. This implies some sort of scheme involving shared memory between the program and **Chad**. In fact, this *is* the solution: allocation requests cause **Chad** to provide a pointer to a data structure, held by **Chad** but accessible by the program, part of which gives an address, referring either to a scratchpad or RAM location, as appropriate, which is given an illegal value when the corresponding resource is destroyed. Several routines support the maintenance of these structures. They will be discussed below.

One final note before plunging into specifics: Since a single PIC can support several Chaps, a set of resources must be maintained for each Chap, and **Chad** must be told which Chap is being addressed by routines. As a result, most **Chad** reoutines include as argument a predefined token, typed *ChapID*, which indicates which Chap is meant.

4. Nuts and Bolts

In the following sections, routines comprising **Chad** are presented and explained at length, interspersed with example **Pirl** routines using the **Chad** routines under discussion. In the course of the discussion we will be exploring the internals of **Pirl**, since that is more or less the canonical use of **Chad**. The Appendix will repeat the routine declarations, with a more terse explanation. That section is intended as a reference to be used while actually programming with **Chad**.

5. The *Chad* Environment

Unlike **Pirl**, which is limited to one Chap per program, **Chad** will maintain an independent environment for any subset of the Chaps attached to its system. This environment is entered by calling **ChadBegin()** and exited with **ChadEnd()**. The former is declared thus:

```
ChadError ChadBegin ( chapid, exclusive )
ChapID chapid;
int exclusive;
```

The Chaps attached to the system are referred to by a token of type *ChapID*: **CHAP0**, **CHAP1**, etc. Every system with a PIC has a **CHAP0**, and it is the one 99% of all **Chad** programs will use. The *exclusive* flag indicates, if non-zero, that the Chap is not to be open by any other process as long as this one has it open. Even under non-exclusive access, shared access is limited to processes owned by the same user. In almost all cases, *exclusive* should be 0, at least for debugging, since *charm*, the Chap debugger, gets excluded too.

6. Error Handling

Chad has an error-reporting convention identical to **Pirl**'s. When they encounter an error, all **Chad** routines set *ChadLastErr* to an error code (always negative) and return that value, which provides an efficient check on success of completion. Once detected, an error condition can be explained by calling **ChadErrReport** () with the file pointer which will receive the error description (*stdout* or *stderr* recommended for interactive applications).

ChadError **ChadErrReport** (fp)
-- describe the **Chad** error whose code is in *ChadLastErr* to the file *fp*
FILE *fp;

You can call **ChadErrReport** () without any further ado after a routine fails. In fact, a macro, **ASSERT**(), is provided for use in **Chad**. When called with a **Chad** routine as its argument, and that routine returns an error value, then **ASSERT**() calls **ChadErrReport**() and returns from the current procedure.

It is frequently necessary to do more error handling than a simple report-and-return. The **CHECK** () macro is provided for these cases. Like **ASSERT** (), it takes a single argument, an error code as presumably returned by a **Chad** function. It does *not*, however, do an error report, nor does it return from the current function. Instead, when its argument is negative, **CHECK**() sets the global variable *PirlLastErr* to the error code, and branches to the label 'error'. This allows you to place, presumably at the end of your routine, code to perform any cleanup tasks before your error return.

Once *PirlLastErr* is set (by **CHECK**() or however), **PirlErrReport**() will describe the error it indicates. It is perfectly feasible to assign a **Chad** error code to *PirlLastErr*. **Chad**'s and **Pirl**'s error codes are distinct from one another, and **PirlErrReport**() calls **ChadErrReport** () if *PirlLastErr* is a **Chad** error.

7. Resource Allocation: *ChadAlloc*()

Chad maintains control over resources of several different types on its Chaps: dynamic scratchpad space, executable instruction memory, frame buffer tile blocks and pixel windows are all managed through **Chad**. The application program gets a pointer to a host structure denoting these resources when they are allocated, and uses this pointer in subsequent transactions with **Chad**.

A single routine handles all allocation of **Chad** resources: **ChadAlloc**(). It takes a variable number of arguments. The first is a *ChapID*. The last is the predefined token **NIX**. The remaining arguments come in groups, each group signifying allocation of one resource. An argument group begins with a token specifying the requested resource, followed by a handle (a pointer to a pointer) which will receive a pointer to the resource, followed by a type-dependent, but fixed, set of arguments parametrizing the resource:

```
ChadError ChadAlloc (chapid,
[TB, tbpp, firsttile, tileswide, tileshigh,]
[PW, pwpp, tbpp, xmin, xmax, ymin, ymax,]
[SPAD, blockpp, nwords,]
[RAM, pcpp, sym,]
[PIXELS, blockpp, npixels,]
 NIX)

ChadSpad *(*blockpp);
ChadPC *(*pcpp);
ChadTB *(*tbpp);
ChadPW *(*pwpp);
int nwords, npixels, firsttile, tileswide, tileshigh;
char *sym;
```

The first group (denoted with the **TB** token) allocates a *tile block*, giving the first tile, the number of tiles in a row, and the number of rows. The request here allocates and organizes (*tileswide* times *tileshigh*) tiles, beginning with number *firsttile*, such that the block has (32 times *tileswide*) pixels on each of (32 times *tileshigh*) scan lines.

The second group (**PW**) allocates a subset of these pixels (a *pixel window*) with dimensions *xmin, xmax, ymin* and *ymax*. The *tbpp* parameter to this group is a tile block handle, so that the tile block can be allocated and used in the same call to **ChadAlloc**().

The third group (**SPAD**) allocates *nwords* of Scratchpad memory, and sets the pointer whose address is *blockpp* to point to a **Chad** scratchpad block descriptor.

The fourth argument group (**RAM**) above loads a routine into instruction RAM. The *ChadPC* pointer whose address is passed is made to point to a descriptor giving an address in instruction memory of the named routine. The use of *ChadPC*s is explained in Section 11.

The fifth argument group (**PIXELS**) also allocates space in the Chap's scratchpad memory, but in groups of four words, aka *pixel*s. Furthermore, the allocated space obeys the alignment requirements for access to the data as pixels.

The argument groups to **ChadAlloc**() may appear in any order, any may be omitted and all may be repeated at will, so that one call can be made to provide an arbitrary number of resources. The only hard requirement is that the last argument be **NIX**.

An Example: PirlBegin()

Look at Figure 7-1 for an example showing the above routines. This is a subset of the actual definition of **PirlBegin**(). It uses the header file */usr/pixar/include/pirl.h.* Most programs which use **CHECK**() or generate **Pirl** error codes will do likewise. The appropriate header definitions are in */usr/pixar/include/chad.h,* which is included implicitly by *pirl.h.*

```
#include <pirl.h>

/*
 *  PirlBegin(): open the Pirl environment on the given chap
 */

ChadTB ThePirlTB; /* global, default tile block   */
ChadPW ThePirlPW; /*   "      "    pixel window */

PirlError PirlBegin(chap, tile0, nxtiles, nytiles)
ChapID chap;
int tile0, nxtiles, nytiles;
{
    int pwxlimit = (nxtiles*32)-1,
        pwylimit = (nytiles*32)-1;

    CHECK( ChadBegin( chap, 0 ) );
    CHECK( ChadAlloc( chap,
             TB, &ThePirlTB, tile0, nxtiles, nytiles,
             PW, &ThePirlPW, &ThePirlTB, 0, pwxlimit, 0, pwylimit,
             NIX ) );
    return( PIRL_NO_ERROR );
error:
    return( PirlLastErr );
}
```

Figure 7-1: *PirlBegin* Program Listing

PirlBegin() opens the **Chad** environment, as outlined above, by calling **ChadBegin**() for a designated Chap, without excluding other programs from using the Chap.

The two **Chad** routines called by **PirlBegin**() are both enclosed in the **CHECK**() macro, and the last statement, reachable only by branching to *error*, returns *PirlLastErr*. Thus, any program which calls **PirlBegin**() can simply check that its return value is non-negative, calling **PirlErrReport**() if so.

At the beginning of the routine, *pwxlimit* and *pwylimit* are computed and set to be the maximum x and y coordinates for a pixel window in the given tile block. **ChadAlloc**() uses the given tile block parameters to allocate a tile block, *ThePirlTB*, which is global to **Pirl** and is available to all programs using it. Ditto *ThePirlPW*, which is a pixel window encompassing all pixels within *ThePirlTB*.

8. Freeing Resources: *ChadFree*()

Once a program is finished with its Chap resources, it is responsibile for deallocating them so that other routines (and, more important, other programs) can use them. **Chad** resources are deallocated with **ChadFree**(). **ChadFree**() is another routine which takes an arbitrary number of arguments, each of them either a resource pointer as allocated by **ChadAlloc**(), or a terminating **NIX**.

> *ChadError* **ChadFree** ([blockp,] [pcp,] [tbp,] [pwp,] **NIX**)
> -- free the given resources *AND* their associated host structures
> *ChadSpad* *blockp;
> *ChadPC* *pcp;
> *ChadTB* *tbp;
> *ChadPW* *pwp;

Naturally, the resource pointers given by **ChadAlloc**() and freed by **ChadFree**() point to some structure on the host computer. The question then arises: does **ChadFree**() deallocate only the Chap resource, leaving the host structure valid so that the *addr* field can be tested, or does it also deallocate the host structure, thereby possibly creating a dangling pointer?

The answer is the former. The host pointer points to the ozone after **ChadFree**(). Thus, **Chad** uses the *alloc* model in assuming that if you explicitly deallocate something, you don't expect to be using it again.

9. Leaving *Chad*

The symmetric opposite of **ChadBegin()** is **ChadEnd()**:

ChadError **ChadEnd** (chapid)
-- close out the **Chad** environment
ChapID chapid;

ChadEnd() does **not** deallocate any resources. This must be done explicitly by the host program.

A good example of the use of **ChadFree()** and **ChadEnd()** is **PirlEnd()**, the routine which cleans up the **Pirl** environment when a program is through with it (shown in Figure 9-1). Since **PirlNewPW()** allocates pixel windows for **Pirl** users, **Pirl** acts as a good citizen and deallocates all such pixel windows (recorded in the static array *pws* and counted by the static **int** *npws*), in addition to *ThePirlTB* and *ThePirlPW*. **PirlEnd()** then calls **ChadEnd()**.

```
/*
 *  PirlEnd(): exit Pirl, closing all open pixel windows
 */
PirlEnd()
{
    CHECK( ChadFree ( ThePirlPW, ThePirlTB, NIX ));
    while ( npws )
        CHECK( ChadFree ( pws[--npws], NIX ));
    CHECK(ChadEnd());
    return ( PIRL_NO_ERROR );
error:
    return ( PirlLastErr );
}
```

Figure 9-1: *PirlEnd* Program Listing

10. Writing to Resources: *ChadWrite*()

Thus far, we have covered allocating and deallocating resources, but no mention has been made of how to move information between the host and the Chap. As for allocation/deallocation, **one** routine is used for all writing from the host to the Chap: **ChadWrite**(). Again we see the use of multiple argument groups in the argument list, terminated by **NIX**:

ChadError **ChadWrite** (chapid,
-- write to a **Chad** resource
[**SPAD**, blockp, val, offset,]
[**SPADARRAY**, blockp, vals, nwords, offset,]
[**PIXELS**, blockp, pxvals, npixels, offset,]
[**SYSBUS<0..15>**, val,]
[**R<0..31>**, proc, val,]
[**B<0..15>**, val,]
[**I<0..15>**, val,]
NIX)

ChapID chapid;
*ChadSpad *blockp;*
int *proc, nwords, npixels, offset;*
CHAPVAL val, vals[ ];
RGBAPixelType pxvals[ ];

Notice that **ChadWrite**() uses several more resource types than **ChadAlloc**() (you may have been wondering what happened to the variety of types discussed at the beginning). The various register types of the Chap are not allocated or protected in any way, and can be set by any program or routine with the will to do so. Hence the tokens **SYSBUS0**, ..., **SYSBUS15** (for writing one of the 16 sysbus registers), **R0**, ..., **R31** (the 32 registers of the Chap's AMD 29116 ALU), **B0**, ..., **B15**, and **I0**, ..., **I15** (the base and index registers -- 16 apiece -- of the Chap's scratchpad memory). There is no access to the ALUs' accumulators.

The sysbus, base and index register writes all have the same format: the appropriate token, followed by the value to be written. Since there are four ALUs on a Chap, the ALU register writes must also include *proc*, which is a bitmask specifying which register of four specified by the token will be written. **Chad** defines four bit constants with the obvious meanings: **CHAD_PROCR**, **CHAD_PROCG**, **CHAD_PROCB** and **CHAD_PROCA**. These may be bitwise-or'ed to write to any subset of the registers; **CHAD_ALLPROCS** is the union of them all

The registers are typically used to pass parameters to routines in instruction RAM. This action (writing to Chap registers) is a key element in the ability **Chad** gives to run any program on the Chap from the host: most Chap routines take their parameters in registers, so programs consisting of calls to Chap routines can easily be written for the host. This gives you the ability to easily prototype programs using UNIX programming tools, and gradually migrate programs down to the Chap as the need appears.

You can most easily determine the actual parameters expected by a particular routine from the source file itself. Each routine is described on a manual page in Section 3C of the **Pixar Programmer's Manual**, but the specification of which parameter goes into what register is given by a comment in the source file. These exist in the subdirectories of */usr/pixar/chap/src/lib* for the various libraries.

The other operands of **ChadWrite**() write to scratchpad resources under various formats (these, of course, must have been previously allocated with **ChadAlloc**()). The first argument is always *blockp*, of type *ChadSpad*, and each Spad write also requires an *offset* from the base of the scratchpad block at which the write originates. A write to **SPAD** requires only one further argument, the value to be written. Writes to **SPADTAB** and **SPADARRAY** both take an *array* of values, a count of the number of values to write, as well as the offset into the scratchpad block. The difference between them is that the former performs an untessellated write, suitable for use as a table, and the latter write is tessellated. Finally, a set of pixel values may be written by including a **PIXELS** specification to **ChadWrite**(). In this case, *pxvals* is an array of *RGBAPixelType* elements, and *npixels* gives a number of *pixels* to write, as opposed to a number of words (a pixel is four words).

A word about the data written by **ChadWrite**(): since scratchpad words are 16 bits wide, only the first 16 bits of the values written are significant. However, all integers are 32 bits wide under UNIX running on Vax or Sun systems. This becomes significant on systems which support smaller word sizes either for passing as parameters (for the scalar writes) or for packing in arrays (for the vector writes **SPADTAB, SPADARRAY** and **PIXELS**).

11. Executing Chap Programs: *ChadGo*(), etc.

To do real work on the Chap, we need only one more routine. **ChadGo()** is the sole routine which executes Chap programs. It takes a single argument, a *ChadPC* pointer giving the address of the routine in scratchpad:

ChadError **ChadGo** (pcp)
-- run a routine on the *ChadPC*s Chap, returning if the *ChadPC*
specified by *pcp* has become invalid.
ChadPC *pcp;

PirlClear()

We can now give an example of a **Pirl** routine which actually does something useful:

```
#include <pixeldef.h>
#include <pirl.h>

PirlError PirlClear(pw, color)
ChadPW *pw;
RGBAPixelType *color;
{
    ChadPC *PWClear;
    ChadSpad *pxl;

    CHECK( ChadAlloc ( CHAP0,
        RAM, &PWClear, "PWClear",
        PIXELS, &pxl, 1,
        NIX));

    CHECK( ChadWrite( ChadOwner( pw ),
        B0, pw->addr,
        B1, pxl->addr,
        PIXELS, pxl, color, 1, 0,
        NIX ) );

    CHECK( ChadGo ( PWClear ) );
    CHECK( ChadFree ( pxl, PWClear, NIX ) );
    return ( PIRL_NO_ERROR );

error:
    return ( PirlLastErr );
}
```

Figure 11-3: *PirlClear* Program Listing

PirlClear() sets every pixel in a given pixel window to a particular color. This program includes the header file *pixeldef.h* to use the pixel descriptor **RGBAPixelType**, and *pirl.h* for the various predefined constants, especially error codes, used by **Pirl**. *pirl.h* also implicitly includes *chad.h*, which would otherwise have to be included explicitly.

PirlClear() allocates two resources for its use, then frees them before returning. *PWClear* is a **RAM** resource: a program on the Chap which is included in one of the standard archives in which **Chad** expects to find code. It is allocated by giving a *ChadPC* handle, and the name of the routine, "PWClear". **Chad** looks through its archives for a routine with that label, loads the routine into the instruction memory of the Chap, *and resolves all unresolved references in the routine*. Thus the *ChadPC* pointer comes back ready to run. The dynamic loader used by **Chad** is also efficient, maintaining a system-wide symbol table so that resident routines are used to resolve references in new code, ensuring that no routine is loaded twice.

The other resource allocated by **PWClear** is space for a single pixel in scratchpad, whose address is given by *pxl*. This pixel is used as a parameter to the routine **PWClear**, telling it the color with which to fill the window.

After allocating its resources, **PirlClear**() uses **ChadWrite**() to set up the parameters of **PWClear**. The Chap routines in the Pixar Software Release are overwhelmingly register-based, which is the main reason **Chad** provides access to them. **PWClear** takes three parameters; when it is called, the base registers **B0** and **B1** should be set to the address in scratchpad of a pixel window and a pixel, respectively. Note the use of the *addr* field of the *ChadPW* and **ChadSpad** structures to give the actual locations to which these structures refer. The **PIXELS** part of the **ChadWrite**() call moves a pixel value down to the scratchpad. The 1 and 0 here refer to the number of pixels to be written, and the offset from the address given by *spad* at which they will be written. We are only transferring one pixel here, but obviously any number can actually be moved.

Chad leaves a certain amount of responsibility to the programmer. Here, that responsibility is exemplified by the fact that *no checking is performed to ensure that the pixels written actually fit into the space previously allocated in scratchpad.*

The **PWClear** Chap routine is actually invoked by the **ChadGo**() host routine. **ChadGo**() waits for any previously-running routine to complete before starting **PWClear**, and returns immediately after invoking it. In this way, **Chad** allows fully parallel operation between the Chap and the host. The host can go on to do whatever preparation it needs for invoking the next routine, up to the point of other **Chad** interactions, and advanced programmers can even create host programs that converse with Chap routines.

Before returning, **PirlClear**() frees its pixel and **RAM** resources. As usual, any error condition in the **Chad** routines causes a branch to the *error* label at the end of the routine, which returns *PirlLastError*, the error code set by the **CHECK**() macro.

The version of **PirlClear**() in *libpirl* actually differs from the code shown here. The latter could stand some improvements, which form the subject of the next four sections.

12. Determining a Resource's Chap: *ChadOwner*()

extern *ChapID* **ChadOwner**();

The first problem with our **PirlClear**() is its use of **CHAP0** in **ChadAlloc**(). Since many Chaps can be attached to a host, it is inappropriate for a general library routine to assume that it will run on **CHAP0**.

Fortunately, **Chad** remembers on what Chap a given resource was defined, and will tell you if you use the macro **ChadOwner**(). Thus, we could replace the line

```
CHECK( ChadAlloc ( CHAP0,
```

with

```
CHECK( ChadAlloc ( ChadOwner ( pw ),
```

defined. **ChadOwner**() replaces a pointer to a **Chad** structure with the *ChapID* on which the resource was allocated. *It works for any resource pointer obtained via* **ChadAlloc***()*. Although it is a macro, **ChadOwner**() is equivalent to the declaration below:

extern *ChapID* **ChadOwner**();

13. Repeated Execution of *Pirl* Routines

A second weakness of **PirlClear()** as defined here comes up when the function is used repeatedly (say in the 'plaster' demo program in **The** *Pirl* **Tutorial**). It seems wasteful to reallocate and re-free all its resources every time it is called, when every time it uses the same routine, **PWClear**, and just a single pixel. This is a tradeoff between the cost of repetitive allocation and the size of resources which are left allocated between invocations of a routine. There is no good universal policy, so **Chad** leaves it up to the application. **Pirl**, unfortunately, cannot be so lax.

A reasonable alternative, avoiding reallocation, is shown in Figure 13-1.

```
#include <pixeldef.h>
#include <pirl.h>

PirlError PirlClear(pw, color)
ChadPW *pw;
RGBAPixelType *color;
{
    static ChadPC *PWClear = (ChadPC *) 0;
    static ChadSpad *spad;
    ChapID chapid = ChadOwner ( pw );

    if ( !PWClear )   /* The first time through */
        CHECK( ChadAlloc ( chapid,
            RAM, &PWClear, "PWClear",
            PIXELS, &spad, 1,
        NIX ) );

    CHECK( ChadWrite( chapid,
        B0, pw->addr,
        B1, spad->addr,
        PIXELS, spad, color, 1, 0,
        NIX ) );

    CHECK( ChadGo ( PWClear ) );
    return ( PIRL_NO_ERROR );

error:
    return ( PirlLastErr );
}
```

Figure 13-1: a second *PirlClear* Program Listing

The essential elements here are:

- *PWClear* and *spad* are now **static** variables, so that they are preserved from one call to another.
- *PWClear* is (unnecessarily, actually) initialized to 0, providing a check on the prior existence of the resource so that it can be allocated the first time the routine is called.
- **ChadFree()** has now disappeared. Since deallocating resources often causes the host to wait until any Chap routines return, this version of **PirlClear()** has the advantage that it returns before **PWClear** finishes running on the Chap.

14. Resource Reallocation: *ChadCheck*()

PirlClear() now has a significant new bug: what if *spad* or **PWClear** have been deallocated between one invocation and the next? Their pointers will still be valid, but the corresponding resources will have gone away. For reasons of efficiency and consistency, neither **ChadWrite**() nor **ChadGo**() makes an effort to check or reallocate the resources passed to them. **PirlClear**() is on its own.

The answer is **ChadCheck**():

ChadError **ChadCheck** (chapid, [blockp,] [pcp,] [tbp,] [pwp,] **NIX**)
ChapID chapid;
ChadSpad *blockp;
ChadPC *pcp;
ChadTB *tbp;
ChadPW *pwp;

ChadCheck() takes a variable-sized argument list, beginning with a *ChapID* and ending with **NIX**, with other arguments being pointers to **Chad** structures. The routine assures the continued survival of the associated resource, reallocating it as needed.

You don't need **ChadCheck**() to check a resource's validity. The *addr* field of any **Chad** structure is always set to -1 when the resource is deallocated. This *addr* field check is predeclared as the macro **CHAD_RSRCOK**() which takes as argument a **Chad** structure pointer and checks that the pointer is non-zero **and** that it references a valid (i.e. non-deallocated) resource, giving a nonzero value only when both conditions are true.

So now we can replace

```
if ( !PWClear )  /* The first time through */
    CHECK( ChadAlloc ( chapid,
        RAM, &PWClear, "PWClear",
        PIXELS, &spad, 1,
        NIX));
```

with

```
if ( !PWClear )  /* The first time through */
    CHECK( ChadAlloc ( chapid,
        RAM, &PWClear, "PWClear",
        PIXELS, &spad, 1,
        NIX));
else {
    if ( !CHAD_RSRCOK ( PWClear) );
        CHECK( ChadCheck ( chapid, PWClear, NIX));
    if ( !CHAD_RSRCOK ( spad ) );
        CHECK( ChadCheck ( chapid, spad, NIX ) );
```

Notice how little information **ChadCheck**() requires compared to **ChadAlloc**(). This is because **Chad** also stores the parameters of a resource as passed to the latter in the structure itself, so the application program need not remember them. However, you should be aware that **ChadCheck**() *has no idea what data was in any scratchpad location* that was deallocated. You must be prepared to reload any such data yourself.

This concludes our discussion of **PirlClear**(). We now return you to our presentation of higher-level **Chad** functions.

15. Reading from Resources: *ChadRead*()

The routine symmetric to **ChadWrite** () is **ChadRead**():

```
ChadError ChadRead(chapid,
-- read from a Chad resource
[ SPAD, blockp, valp, offset, ]
[ SPADARRAY, blockp, vals, nwords, offset, ]
[ SPADTAB, blockp, vals, nwords, offset, ]
[ PIXELS, blockp, pxvals, npixels, offset, ]
[ SYSBUS<0..13>, valp, ]
[ R<0..31>, proc, valp, ]
[ B<0..15>, valp, ]
[ I<0..15>, valp, ]
  NIX)

ChapID chapid;
ChadSpad *blockp;
int offset, nwords, npixels, proc;
RGBAPixelType pxvals[ ];
unsigned short int *valp, vals[ ];
```

You can expect to call **ChadWrite** () much more often than **ChadRead** (), since data usually flows from the host to the Chap. The major exception is reading error codes resulting from the execution of Chap routines. Another exception is the Chap routine which compiles a histogram of the pixel values in an image, **PirlHistogram**() (Figure 15-1).

PirlHistogram() uses a compromise allocation plan. Since the scratchpad table used to compile the histogram is rather large (and the routine is rather less likely to be called frequently than **PirlClear**()), the scratchpad buffer is allocated and deallocated at every call, but the **RAM** resource (the **PwHistogram** routine) remains in the instruction memory between calls. Most **Pirl** routines use this plan, and in fact it is a good one: one should hesitate to repeatedly load and unload programs from the Chap, since there is plenty of instruction **RAM** for any existing purpose and, by comparison, allocation of scratchpad is quick. Scratchpad is also the scarcest resource on the PIC.

After all resources are allocated, things proceed in the obvious way, until the call to **ChadRead**(). Here, *size* 32-bit words, or (*size**2) 16-bit Chap words, are read by asking **ChadRead**() to read (*size*/2) **PIXELS**. This works because each pixel contains four words, for Red, Green, Blue and Alpha. **PirlHistogram**() uses **PIXELS** instead of scratchpad words to explicitly force the scratchpad table to be aligned on a four-word boundary. The reasons for this are best left until you have perused the **Chap Programming Tutorial**.

The loop following the **ChadRead**() call swaps 16-bit words of the 32-bit output table. The reason is the difference between the 32-bit integer representation used on the host and the double-precision representation of the Chap software.

```
# include <pixeldef.h>
# include <pirl.h>

PirlError PirlHistogram( pw, histogram, size, component )
ChadPW *pw;
int histogram[], size, component;
{
    static ChadPC       *PwHistogram = 0;
    ChadSpad            *spadhistogram, *spad;
    int i, n;

    n = pw->xmax - pw->xmin + 1;

    CHECK( ChadAlloc ( ChadOwner(pw),
        PIXELS, &spad, n,
        PIXELS, &spadhistogram, size/2,
        NIX));

    if ( !PwHistogram )
        CHECK( ChadAlloc ( ChadOwner(pw),
           RAM, &PwHistogram, "PWHistogram",
           NIX) );

    else if ( PwHistogram->addr < 0 )
        CHECK( ChadCheck ( ChadOwner(pw), PwHistogram, NIX ) );

    CHECK( ChadWrite ( ChadOwner(pw),
        B0, pw->addr,
        B1, spadhistogram->addr,
        B2, spad->addr,
        R0, CHAD_ALLPROCS, size,
        R1, CHAD_ALLPROCS, component,
        NIX));

    CHECK( ChadGo( PwHistogram ) );

    CHECK( ChadRead( ChadOwner(pw),
        PIXELS, spadhistogram, histogram, size/2, 0,
        NIX));

    /* words need to be swapped */
    for( i=0; i<size; i++ ) {
        n = histogram[i];
        histogram[i] = (0xffff & (n  >> 16)) | ((n & 0xffff) << 16);
    }

    CHECK( ChadFree ( spad, spadhistogram, NIX ) );
    return( PIRL_NO_ERROR );

error:
    return ( PirlLastErr );
}
```

Figure 15-1: *PirlHistogram* Program Listing

The Reliability of *ChadRead*() Values

Chad seeks not to interfere with the state of the Chap, which means changing no registers and writing only its own private scratchpad locations. However, **Chad** has to use *some* register to do its work. It uses the ALUs' accumulators, and so these are not available from the host. For compatibility with previously-existing Chap routines which use the accumulator to return an error code, though, **Chad**'s last act before returning to its busy-wait state is to set Sysbus12 to the value of *acc*[0]. To check for error returns, then you can simply use **ChadRead**() to read **SYSBUS12**. However, you should beware that **Chad** also uses Sysbus12 for other purposes, so its value is useful *only when accessed immediately after the return of the routine generating the error code*. Thus, it is not necessary to call another **SYSBUS12**-setting routine for it to go away. It is not even necessary to call a Chap routine; **Chad** itself may trash this value if it is not fetched immediately.

With this *caveat*, though, you should be able to reliably read any Chap resource to which **Chad** gives you access.

16. More Resource Deallocation: *ChadReset*() and *ChadBackup*()

ChadFree() deallocates **Chad** resources on an individual basis. It is sometimes necessary, usually in the context of error recovery, to perform more broad deallocation. This is why frequent checks for the continued viability of resources is good programming practice.

There are three more general methods of freeing **Chad** resources. Two use **ChadReset**() and the third, **ChadBackup**():

ChadError **ChadReset** (chapid, [**RAM**,] [**SPAD**,] [**TB**,] [**PW**,] **NIX**)
ChapID chapid;

ChadReset() takes a set of resource tokens as previously used in **ChadWrite**() and **ChadRead**(). It deallocates *every* instance of those resources. That is,

```
ChadReset ( chapid, PW, NIX );
```

deallocates every pixel window declared on the specified Chap by the process in which the call is made. No other pixel windows are affected, even those allocated by processes now dead. You can see how important it is for programs to free their resources; there is no other way for **Chad** to know when a resource is no longer needed.

The other, more drastic, form of **ChadReset**() is to call it with no tokens at all, i.e.

```
ChadReset ( chapid, NIX );
```

This form *completely clears all instruction memory and scratchpad, including all pixel windows and tile blocks.* The difference between this and

```
ChadReset ( chapid, RAM, SPAD, PW, TB, NIX );
```

is that this latter form yanks all resources out from under any other processes which might be sharing the Chap. Of course, it also frees up the resources remaining from prior processes, which is the real reason to call it. When you have exclusive use of a Chap, this form of **ChadReset**() is safe (albeit inconvenient when you have to wait for your Chap programs to reload). When the process calling it is cooperating with another process using the same Chap (whether using **Chad** or not), this action requires the utmost care and cooperation between the programs so that they both know their resources have disappeared.

The third broad deallocation action uses **ChadBackup**():

ChadError **ChadBackup** (structp)
union {
 ChadSpad spad;
 ChadPC pc;
 ChadTB tb;
 ChadPW pw;
} *structp;

ChadBackup() is commonly used in the context of error recovery. It frees the given resource, plus *every resource that was allocated since it was allocated.* Thus a routine would place a call to **ChadBackup()** immediately after the *error* label, giving the first structure it allocated. This is a convenience, since not every structure allocated by the routine need be named. More important, however, this strategy provides a sneaky way for a high-level routine, or program, to deallocate all the **static** resources allocated by any routines "inner" to it.

A word about **Chad**'s general deallocation policy: resources may be deallocated implicitly, but *never the corresponding host structure.* In other words, **Chad** never creates dangling pointers on the host without the knowledge of the application program. Since only **ChadFree()** restricts its actions to resources explicitly named by the calling routine, it alone removes the **Chad** structure associated with the resource. Since **ChadReset()** and **ChadBackup()** can free resources not appearing in their argument lists, they do no deallocation of the corresponding host structure. They only affect the Chap.

17. Do-it-yourself Synchronization: *ChadCPUBusy*() and *ChadWaitCPU*()

As mentioned above, **Chad** never tries to do two things at once, waiting until there is no program executing on the Chap before setting another program running or writing to any register or scratchpad location. The mechanics of this may remain magical, but there are times when you may want to check yourself whether a routine has completed rather than enter some **Chad** routine which hangs awaiting completion. The following two macros are for this purpose:

ChadCPUBusy (chapid)
-- is the given Chap currently executing?
ChapID chapid;

ChadWaitCPU (chapid)
-- busy-wait until !ChadCPUBusy (chapid)
ChapID chapid;

The former is a simple conditional which is nonzero if the Chap is currently busy (i.e., if **ChadGo**() or **ChadWrite**() would hang). The latter is a busy-wait on **ChadCPUBusy**(). It is unfortunate, but true, that the interrupt-generating capability of the Chap is reserved for other purposes (like your own programs), making this busy-wait necessary.

18. *Chad* Internals for the Curious

This section is included to give those who care to know a glimpse into the way **Chad** conducts its business.

Chad is built upon *libpixar,* a library of low-level routines which use the Chap's diagnostic registers to control the state of the Chap.

The host-side component of **Chad** is primarily a storage manager which allocates and keeps track of the host-side structures corresponding to the Chap resources managed by the dynamic loader: *ChadPC*s, *ChadPW*s, etc. These structures record the parameters used to create the resources so that they can be conveniently recreated. The structures also include a header, invisible to the application, giving the Chap on which the resource exists, list pointers, and the time stamp used by **ChadBackup**().

The Chap side of **Chad** consists of a primarily of a simple monitor which busy-waits for accesses of the Virtual Data Registers. The low bit of Sysbus13 is 0 during this busy-wait state and is 1 at all other times. The busy conditional **ChadCPUBusy**() is nothing but a check of this bit.

The various classes of **Chad** resource are mapped into the high-numbered VDRs, such that when a VDR is accessed, the **Chad** monitor branches to the appropriate handling routine. For scalar data (scalar scratchpad locations and registers), this consists of a simple data transfer between the appropriate entity and Sysbus15. Multiple data words are transferred either through single instances of scalar transfers (returning to the monitor meantime) or via simple cooperative routines, depending on the protocol established between the internals of the host side and the Chap side. For programs, this is a branch to the routine itself; all parameters are assumed to have been previously loaded.

19. Further Explorations

The foregoing discussion should enable you to understand the source in the existing **Pirl** library. This is a good source of code to modify in order to create new routines. The source to all **Pirl** routines is in the directory */usr/pixar/host/src/lib/libpirl*, and you are encouraged to examine, modify and put it to new uses.

Of particular use is the library of source code for the Chap contained in the subdirectories of */usr/pixar/chap/src/lib*. These directories provide parameter descriptions for the multitude of Chap routines, thus giving you a source of routines for the Chap to be invoked by your host programs.

To get the most out of these routines, however, and to write your own, you will want to become familiar with programming the Chap directly. The Pixar Software Release contains a complete set of tools for doing this, including an assembler, an interactive debugger, an archiver and, of course, the dynamic loader used by **Chad**. The **Chap Programming Tutorial** will return you to hands-on programming and get you into the internals of the Pixar Image Computer. If the prospect of "assembly-level programming" is daunting to you, we encourage you to withhold judgement until completing that tutorial.

20. Appendix: Summaries of Routines

This appendix is a terse summary of the previous material. It is provided for use as a quick reference while writing programs using **Chad.**

ChadBegin()

ChadError **ChadBegin** (chapid, exclusive)
ChapID chapid;
int exclusive;

ChadBegin() must be called once for each Chap a program expects to use under **Chad.** It restarts the **Chad** monitor, which means that if any processes are sharing a Chap under **Chad,** they must all be in the **Chad** environment before any of them begins executing Chap programs.

ChadEnd()

ChadError **ChadEnd** (chapid)
-- close out the **Chad** environment
ChapID chapid;

ChadAlloc()

ChadError **ChadAlloc** (chapid,
[**TB**, tbpp, firsttile, tileswide, tileshigh,]
[**PW**, pwpp, tbpp, xmin, xmax, ymin, ymax,]
[**SPAD**, blockpp, nwords,]
[**RAM**, pcpp, sym,]
[**PIXELS**, blockpp, npixels,]
NIX)

ChadSpad *(*blockpp);
ChadPC *(*pcpp);
ChadTB *(*tbpp);
ChadPW *(*pwpp);
int nwords, npixels, firsttile, tileswide, tileshigh;
char *sym;

ChadCheck()

ChadError **ChadCheck** (chapid, [blockp,] [pcp,] [tbp,] [pwp,] **NIX**)
ChapID chapid;
ChadSpad *blockp;
ChadPC *pcp;
ChadTB *tbp;
ChadPW *pwp;

ChadFree()

ChadError **ChadFree** ([blockp,] [pcp,] [tbp,] [pwp,] **NIX**)
-- free the given resources *AND* their associated host structures
ChadSpad *blockp;
ChadPC *pcp;
ChadTB *tbp;
ChadPW *pwp;

ChadFree() is the only deallocation routine which frees up the host structure as well as the Chap resource, making all pointers passed to it dangle.

ChadReset()

ChadError **ChadReset** (chapid, [**RAM**,] [**SPAD**,] [**TB**,] [**PW**,] **NIX**)
ChapID chapid;

With any token arguments, **ChadReset**() frees up the Chap resources associated with all the associated resource classes *for a singe process*. With only **NIX** as an argument, **ChadReset**() performs a hard reset of the Chap, removing all resources on the Chap of all types *for all processes*. This form is to be used only with great caution.

ChadBackup()

ChadError **ChadBackup** (structp)
union {
ChadSpad spad;
ChadPC pc;
ChadTB tb;
ChadPW pw;
} *structp;

ChadLibs()

ChadError **ChadLibs** (lib1, ..., libN, **NIX**)
-- include the given Chap-code archives in **Chad**'s search path
char *lib1, ..., libN;

ChadLibs() adds its arguments to the front of the search path in argument order. Therefore the last named archive will be searched first.

ChadWrite()

```
ChadError ChadWrite ( chapid,
-- write to a Chad resource
[ SPAD, blockp, val, offset, ]
[ SPADARRAY, blockp, vals, nwords, offset, ]
[ PIXELS, blockp, pxvals, npixels, offset, ]
[ SYSBUS<0..15>, val, ]
[ R<0..31>, proc, val, ]
[ B<0..15>, val, ]
[ I<0..15>, val, ]
 NIX)

ChapID chapid;
ChadSpad *blockp;
int proc, nwords, npixels, offset;
CHAPVAL val, vals[ ];
RGBAPixelType pxvals[ ];
```

ChadWrite() waits for the completion of any running Chap program before performing its reads.

SPAD and **SPADARRAY** writes are tesselated. **SPADTAB** writes are not, and are therefore suitable for index-mode lookup tables. If this makes no sense to you, then you shouldn't be using **SPADTAB** writes.

ChadRead()

```
ChadError ChadRead(chapid,
-- read from a Chad resource
[ SPAD, blockp, valp, offset, ]
[ SPADARRAY, blockp, vals, nwords, offset, ]
[ SPADTAB, blockp, vals, nwords, offset, ]
[ PIXELS, blockp, pxvals, npixels, offset, ]
[ SYSBUS<0..13>, valp, ]
[ R<0..31>, proc, valp, ]
[ B<0..15>, valp, ]
[ I<0..15>, valp, ]
  NIX)

ChapID chapid;
ChadSpad *blockp;
int offset, nwords, npixels, proc;
RGBAPixelType pxvals[ ];
unsigned short int *valp, vals[ ];
```

SPAD and **SPADARRAY** writes are tesselated. **SPADTAB** writes are not, and are therefore suitable for index-mode lookup tables. If this makes no sense to you, then you shouldn't be using **SPADTAB** writes.

ChadGo()

ChadError **ChadGo** (pcp)
-- run a routine on the *ChadPC*s Chap, returning if the *ChadPC*
specified by *pcp* has become invalid.
ChadPC *pcp;

ChadGo() always waits for the previously-executing Chap program to complete before starting another.

ChadCPUBusy(), ChadCPUWait()

ChadCPUBusy (chapid)
-- is the given Chap currently executing?
ChapID chapid;

ChadWaitCPU (chapid)
-- busy-wait until !ChadCPUBusy (chapid)
ChapID chapid;

ChadOwner()

extern *ChapID* **ChadOwner();**

ChadOwner() is a macro which gives the *ChapID* of the Chap on which a **Chad** resource is defined.

ChadErrReport()

ChadError **ChadErrReport** (fp)
-- describe the **Chad** error whose code is in *ChadLastErr* to the file *fp*
FILE *fp;

ChadErrReport() prints a message summarizing the last **Chad** error to the file given in its argument.

Chap Programming Tutorial

PIXAR

ABSTRACT

A companion to **The** *Pirl* **Tutorial** and *Programming with Chad*, this document introduces Chap programming at the lowest level, acquainting the reader with the programming environment of the Pixar Image Computer for writing native Chap programs. After completing this tutorial, the reader should be able to write and run programs on the Chap with impunity. We assume at least a cursory understanding of the Pixar software environment and its concepts as presented in *Pixar Software Overview*, and familiarity with **Chad**, as presented in *Programming with Chad*. UNIX† literacy is also assumed.

November 18, 1986

† UNIX is a Trademark of Bell Laboratories.

Table of Contents

1. Introduction 1
2. Review of Chap Architecture 2
3. Sample1: Fundamentals 3
Preprocessor definitions 4
Assembler directives 4
Data Representation 4
Scratchpad access: Broadcast Mode 5
Scratchpad Access: Pixel Mode 6
Introducing the Sequencer 7
Note on Instruction Packing 8
End of Program 8
3.1. Assembling sample1 8
References 8
4. The Debugger 9
4.1. Note on Radix 9
4.2. Load the Program 9
4.3. Display the Segment (load) Map 9
4.4. Display the Symbol Table 10
4.5. Displaying Scratchpad Memory 10
4.6. Number Representation 11
4.7. Accessing Program Memory 12
4.8. Runtime Control 13
4.9. Breakpoints 14
4.10. Examining registers 14
4.11. Changing the Radix 15
4.12. Getting out of Charm 15
Addendum for the Curious 16
References 16
5. Sample2: Paint a Colored Rectangle 17
5.1. Condition Codes and Program Flow 18
5.1.1. Syntax of "if ... else ... fi" 18
5.1.2. How Conditionals Work on the Chap 18

5.1.3. "All or nothing" control 18
5.1.4. Low Level Sequencer Control 19
5.2. Subroutines 19
5.2.1. Library Subroutines: Sharing Registers 19
5.2.2. Library Subroutines: Calling Conventions 19
initialize 20
AllocTB 20
OpenPW 21
SFxCopy 21
5.3. Assembling sample2 22
5.4. Running sample2 23
6. Sample3: In Which the Host Plays a Role 24
6.1. Sample3: Chap side 24
Outer loop 24
Synchronization 25
Synchronization loop 25
6.2. The host program 25
General remarks 26
Signaling: Chap to host 27
6.3. Running Sample3 27
References 28
7. Sample4: Transparent Colored Rectangles 29
More storage 29
The alpha channel 29
Selecting the alpha channel 29
Setting the runflag 29
Filling **new_row** 31
Merge the Rectangle 31
Using the multiplier 32
8. Further Topics 34
Component addressing 34
9. Programming Pitfalls 36
4-Way Registers, Scalar Registers, and Runflags 36
"**Runflag =**" vs. **force** 36
Library calls and runflags 36
10. Sample5: The host part completed 37
10.1. Relocation 37
10.2. Link editing 37

10.3. The Dynamic Loader 38
10.3.1. The Dynamic Loader's Symbol Table 38
10.3.2. Creating Relocatable Object Modules 38
10.3.3. Dynamic Loading From Charm 38
10.3.4. Searching Libraries 40
10.3.5. Miscellaneous Commands 40
10.3.6. Dynamic Loading From The Host 41
ChapLoadGo 41
Running *sample5s* 42
11. Chap Routines 43
General Remarks 43
11.1. Advanced Chas Programming 44
11.1.1. Virtual Data Registers 44
Suggestions for VDR Usage 46
11.2. The Dynamic Loader 47
ChapMMan 47
ChapBeginLoad and **ChapEndLoad** 47
ChapSymLookup 48
ChapSetArchives 48
ChapLoad 48
ChapRunAsync 49
11.3. Memory Management 50
ChapGetConfig 50
Allocating and Freeing Chap Resources 51
Resetting 51
11.4. Hardware Interrupts 52
12. Using **Chad** 53
12.1. General Remarks 53
12.2. The Cost of Using *Chad* 53
12.3. Data Transfer to Running Chap Programs 53
12.4. *Chad* and the Chap Libraries 55
12.5. Discussion: the *Chad* Development Process. 56
13. Video Routines 58
General Remarks 58
13.1. Display Parameters 58
VideoSetDisplay 58
Channel Crossbar 59
Commentary on Figure 13-1 60

VideoZoom 60
13.2. Colormap Demo 60
Computing the Colormap 61
Sample Program *contour* 62
13.3. Hardware Cursor 63
Manipulating the Active Cursor 63
Conclusion 65
14. Miscellaneous Routines 66
14.1. The memory controller routines 66
14.2. Disk buffer routines 67
Setting Up the Dumi 67
Opening the Disk Buffer 68
Cleaning Up 69
15. Conclusions 70

Chap Programming Tutorial

PIXAR

Marin County, CA

1. Introduction

The **Chap** (**Channel Processor**) is the processing unit of the Pixar Image Computer, its "CPU". This tutorial will prepare you to write and debug programs for the **Chap,** on the **Chap,** as well as programs running in cooperation between the host and the Chap.

The tutorial consists of a sequence of sample programs. Each program illustrates one or more features of the Pixar Image Computer and its programming environment. The programs incrementally develop the simple task of drawing colored rectangles into the framebuffer memory. The programs are written in Chas, the Chap assembly language. You are shown how to assemble the programs using Chas and load them using **Chad**, the Pixar host library for managing the Chap introduced in *Programming with Chad.* You will also be introduced to **Charm**, the debugger for the Chap. The first few samples are entirely devoted to writing and running programs independent of **Chad**. A later section will teach you how to incorporate what you have learned with this system for automatically running Chap programs from within host C programs, and another section describes the services provided by **Chad** to complete the process of writing applications involving cooperation between Chap and host programs.

There are optional exercises to test your understanding, and references to more complete sources of documentation. The source code for all sample programs is distributed with the system software in the directory */usr/pixar/doc/tutorial/chap*. You are encouraged to copy these programs and tinker with them as you read the tutorial.

The first part of this tutorial takes you through five examples. These deal with using Chas and Charm to make rectangles appear on the screen. The second part discusses advanced topics for the Chap, and programming the video and Dumi boards. We suggest you work through the tutorial from start to finish; the few days you spend here will provide a solid background for writing your own programs for the Pixar Image Computer. You may find it helpful to refer to the Chas Assembler Reference Manual, especially its appendix discussing the sequencer.

2. Review of Chap Architecture

Recall that the Chap is a single-instruction, multiple-data-stream machine consisting of four identical processors. Each processor contains an Arithmetic Logic Unit, or **ALU** for short. Each has 32 general-purpose registers, known to the assembler as `r0` through `r31`. These and other registers associated with each processor are sometimes referred to as **4-way** registers, since there are 4 of each in each Chap. There is a 64K-by-16-bit scratchpad memory shared by all processors. Access is controlled by an array of 16 base and 16 index registers named `b0` through `b15` and `i0` through `i15`, respectively. There are four modes for accessing the scratchpad memory. There is a large framebuffer memory connected to the Chap through a high-speed bus. Scratchpad and framebuffer are organized so that four consecutive 16-bit words of the former map to the red, green, blue, and alpha channels of a single pixel. The alpha channel is a fourth channel that may contain arbitrary data. The examples contained in this tutorial use it to store transparency information. A single sequencer controls the execution states of all four processors.

3. Sample1: Fundamentals

The first sample program, *sample1*, begins the task of drawing a rectangle on the screen and the second, *sample2*, completes it. We learn:

- the syntax of a simple program
- how to use *broadcast* and *pixel* modes of accessing scratchpad
- how to write a simple program loop.

sample1 is shown in Figure 3-1. The line numbers are not part of the source file, but are included here for reference. A discussion of the program follows.

```
  1        /*
  2         * Prepare to draw a rectangle:
  3         * Create one horizontal span for a rectangle.
  4         */
  5      #define   rx0      r8
  6      #define   ry0      r9
  7      #define   rxsize  r10
  8      #define   rysize  r11
  9      #define   rcolor  r12

/* .data section: fill scratchpad */

 10             .data
 11      color:
 12            .pixel      .35E, .22E, .8E, 0
 13      rect:
 14            .pixel      100, 250, 100, 250

/* .bss section: allocate scratchpad space for one scan line. */

 15            .bss
 16      one_row:
 17            .space 1024*4

/* .text section: assemble text into Chas instructions. */

 18            .text
 19      sample1:
 20            b0 = rect
 21            i0 = 1
 22            rx0 = @b0;  b0 = b0 + i0   /* read x0 */
 23            acc = @b0;  b0 = b0 + i0   /* read x1 */
 24            rxsize = acc - rx0         /* compute x dimension */
 25            rxsize = rxsize + 1;       /* include end pt */
 26            ry0 = @b0;  b0 = b0 + i0   /* read y0 */
 27            acc = @b0;  b0 = b0 + i0   /* read y1 */
 28            rysize = acc - ry0         /* compute y dimension */
 29            rysize = rysize + 1;       /* include end line */
 30            b0 = color                 /* read color */
 31            rcolor = (b0)
 32            b0 = one_row
 33            i0 = 4
 34            loop rxsize do                /* build one row */
 35                  (b0) = rcolor; b0 = b0 + i0
 36            done
 37            bpt                           /* end of program */
```

Figure 3-1: Sample1 Program Listing

Preprocessor definitions [lines 1 to 9]

Notice that comments and macro definitions are the same as in the C programming language. This is not surprising since the same preprocessor is applied before assembling the source files.

We use the `#define` facility to make the program more readable by substituting mnemonic names for the register names.

Assembler directives [lines 10 to 19]

Assembler directives are keywords identified by a leading *dot* (.). The directives `.data`, `.bss`, and `.text` mark the three main divisions of the program. These tell the assembler where to assemble the data or instructions that follow. The divisions are:

`.data`	initialized memory locations in scratchpad
`.bss`	reserved data locations (initialized to zero) in scratchpad
`.text`	instructions loaded into the instruction memory

Labels may be placed at the beginning of any line in any of these segments. A label is a name followed by a colon. Names are strings composed of alphanumeric and/or underbar ('_') characters (the first character cannot be a digit).

The `.pixel` directive tells the assembler to assemble the following four values into scratchpad memory using *tessellated* storage. This means that when you retrieve these data using pixel-mode accesses, processor 0 will get the first value, processor 1 will get the second, etc.

The `.bss` directive allocates space to be initialized to zero at load time. Since each pixel consists of four words, the example above makes room for a row of 1024 pixels by allocating 1024*4 words. Notice that the assembler can deal with constant expressions such as '1024*4.'

Data Representation [lines 12 to 14]

To paint a colored rectangle, we need to have some data: a color and a rectangle.

Note that the components of `color` are specified by a decimal fraction followed by E or e. The range of component values that correspond to visible colors is [0.0E,1.0E] where 1.0E is full on and 0.0E is full off. This numbering scheme will be described in more detail later. The first value is the red component of the color, the second is the green, the third the blue. The fourth value is not used in sample1, but later, in sample4, we will use it for transparency information.

`rect` through `rect`+3 contain the four coordinates of the rectangle: lower and upper x-values, and lower and upper y-values. These four values will be referred to as x0, x1, y0, and y1, respectively, in the following discussion. This order (x0, x1, y0, y1) is important, for this is the order in which we will want to access the numbers later.

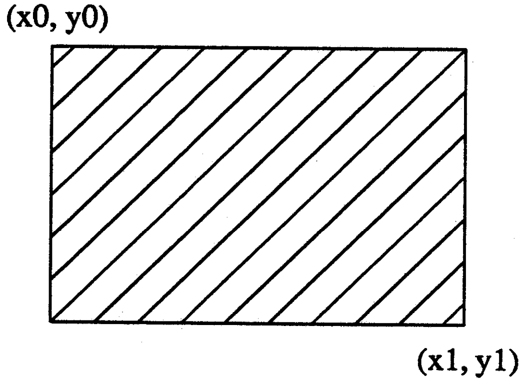
The default radix for specifying integers is decimal, but the assembler also accepts octal and hexadecimal format. This example also shows that the values of a `.pixel` directive need not be actual RGBA color components of a pixel.

Figure 3-2 shows a logical map of memory after data is loaded.

	R	G	B	A
color:	.35	.22	.8	0
rect:	100	250	100	250
	⋮	⋮	⋮	⋮

Figure 3-2: Scratchpad Memory Contents

Figure 3-3 shows the relation of the rectangle to the registers used in the computation.



register	contents
rx0 (r8)	x0
ry0 (r9)	y0
rxsize (r10)	x1 - x0
rysize (r11)	y1 - y0

Figure 3-3: Rectangle Coordinates

Scratchpad access: Broadcast Mode [lines 20 to 27]

The first task of our program is to calculate the dimensions of the rectangle. To do this we must retrieve the corner points stored in `rect`. We want each of the four processors to receive the complete rectangle description, so we use *broadcast* mode to access it. Broadcast-mode read distributes the same 16-bit word from scratchpad to each of the four processors. Figure 3-4 illustrates this mode. Broadcast mode is represented by an '@' preceding a base register. In this case we set `b0` to the address of the rectangle, and `i0` to 1. This choice of the index register lets us distribute consecutive words of scratchpad to all four processors.

The sequence of events in this part of the program [lines 20 to 29] is:

- Load `rx0` with coordinate x0 from `rect` and increment `b0` to `rect`+1.
- Load `acc` (the accumulator—a special ALU register) with x1 from `rect`+1 and increment `b0` again.
- Set `rxsize` to x1–x0.

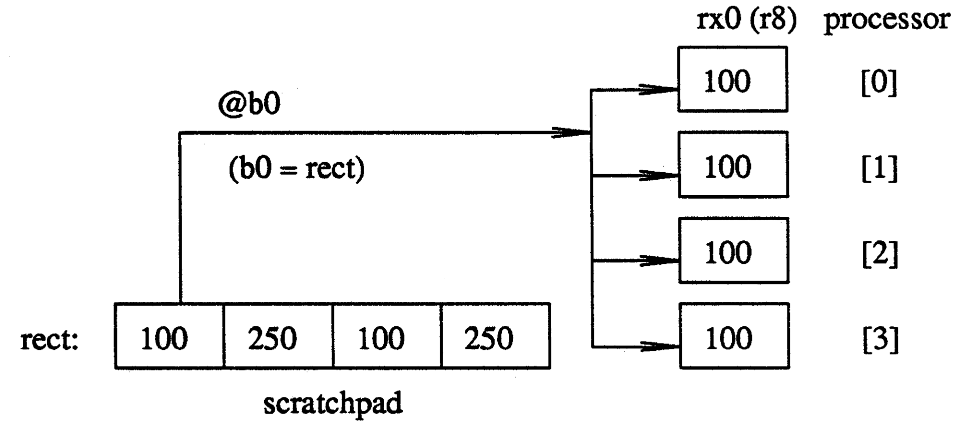


Figure 3-4: Broadcast Mode Read: "`rx0 = @b0`"

- Add 1 to **`rxsize`**; this is the X dimension of the rectangle, which contains both beginning and end points.
- The Y dimension is calculated and handled in an analogous way.

Notice that we use **acc** because the ALU is a single operand ALU. Two general purpose registers cannot be named on the right hand side of an assignment statement. Hence, **`acc=r1 - r2`** becomes the following two statements executed in sequence:

```
acc=r1
acc=acc - r2
```

Scratchpad Access: Pixel Mode [lines 30 to 36]

The next section of the program uses pixel mode to access the scratchpad. The statement

```
rcolor = (b0)
```

is a pixel-mode read, the effect of which is illustrated in Figure 3-5. Each processor is loaded with one of the four components of the pixel accessed. Processor 0 gets the red component, processor 1 gets the green, processor 2 gets the blue, and processor 3 gets the alpha component.

Inside the loop [line 35] is a pixel-mode write into the scratchpad space reserved for the horizontal line, **`one_row`**. Notice that the index register **`i0`** is now set to 4; this is the usual setting when pixel-mode access is used, since four words are transferred in each access. Figure 3-6 illustrates a pixel-mode write.

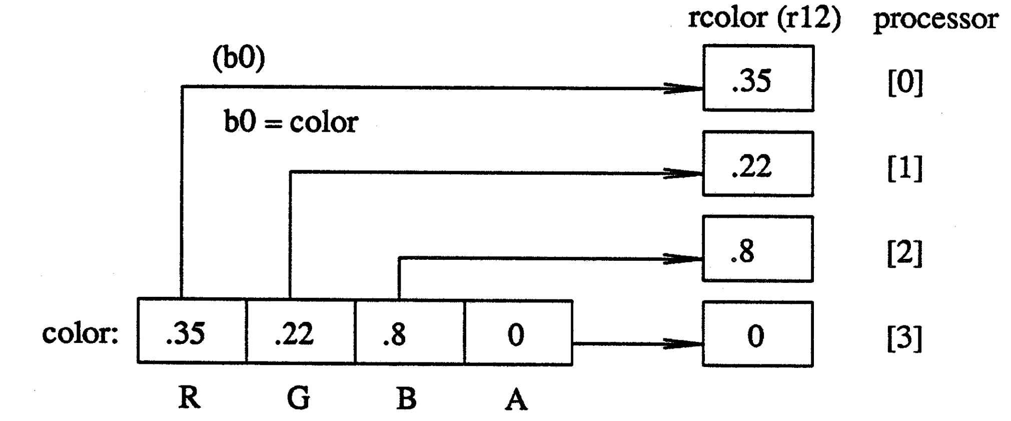


Figure 3-5: Pixel Mode Read: "`rcolor = (b0)`"

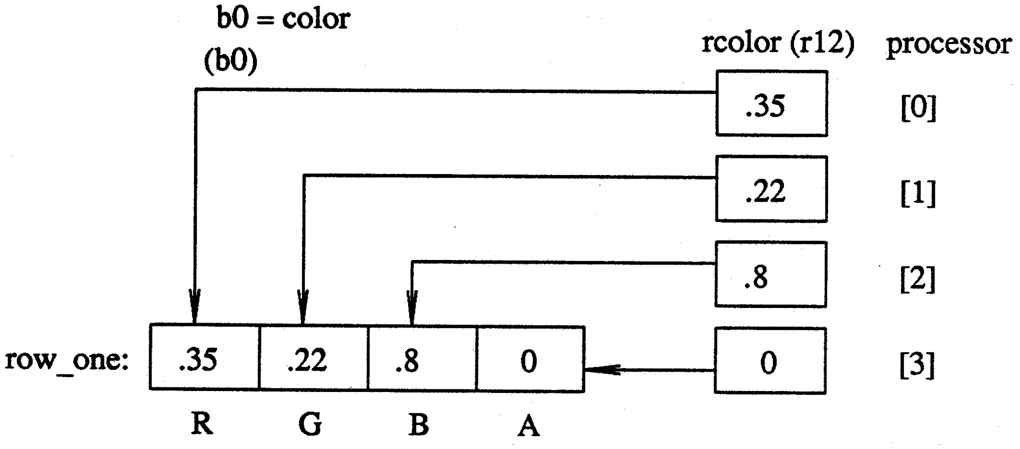


Figure 3-6: Pixel Mode Write: "`(b0) = rcolor`"

Introducing the Sequencer [lines 34 to 36]

The Chap has a single sequencer to control the execution of all four processors. The sequencer uses a single loop counter, which is shared by the processors. The construct

```
loop rxsize do
            ...
done
```

executes the statements inside the loop **`rxsize`** times*.

* There is a subtlety involved here: `rxsize` is a 4-way register, while there is only one loop

Note on Instruction Packing

In general, all the statements on one line are assembled into one microcode instruction. Multiple statements are separated by semicolons. You may also specify that multiple lines are to be assembled into one instruction by enclosing them in curly brackets ({}). There is a limit to what can be specified in one instruction. This will be covered in **Chas**. For now, it is sufficient to know that the assembler will complain if you ask for too much.

End of Program [line 37]

This program ends with the breakpoint statement

```
bpt
```

This is a convenient way to end a program fragment so you can test it with the debugger. When this instruction is executed, control will return to the debugger.

3.1. Assembling sample1

At this point, UNIX users should confirm that their **PATH** environment variable includes the directory */usr/pixar/host/bin*. This will give you access to all the programs discussed subsequently, including the Chas assembler *chc*, the loader *chload*, the debugger *charm* and others.

The sample program shown above is contained in the file ''sample1.s''. Assemble the program with the command

```
% chc sample1.s –o sample1.out
```

The command shown runs the macro preprocessor, assembles the source file sample1.s, and produces the executable object file sample1.out.

The *chc* command is the Chap analog of the UNIX† *cc* command. *Chc* is used to compile and link Chas programs. Like *cc*, *chc* invokes the C-preprocessor. *Chc* then assembles the Chas programs into relocatable object files and links them with other object files to form executable modules.

References

The standard reference on Chas is *The* **Chas** *Reference Manual*. Its appendix on sequencer instructions gives a detailed description of the Chap instruction set. The manual pages of the *Pixar User's Manual* describe *chc* more fully.

counter. This means that only one of the four values of `rxsize` is used as the loop count. By default the register associated with processor 0 is used. But, in general, any of the processors' registers may be used. Processor 1 can be chosen by specifying:

```
loop rxsize[1] do
        ...
done
```

† UNIX is a Trademark of Bell Laboratories.

4. The Debugger

We use the debugger **Charm** to run and test our sample program. Before running the debugger, give the Unix command

```
% chmap -i
```

This will initialize the Chap symbol table. Later, we will see how the dynamic loader allows several users to have programs loaded in the Chap at once, but for our present purposes we want to have the machine to ourselves.

To begin a debugger session use the command 'charm'. Charm identifies itself, then prompts for input:

```
Chap Runtime Monitor, version 3.1 of Tue Aug 19 21:11:26 PST 1986
>
```

A debugger session for the first sample program is shown below. Users of the standard UNIX debugger **adb** will notice a strong resemblance in Charm's command syntax. The approach below is to show sample commands rather than a systematic exposition. See the Charm manual for details.

4.1. Note on Radix

The default radix used by Charm **in most cases** is hexadecimal. Exceptions will be noted in the tutorial. Input values whose hexadecimal form begin with an alphabetic digit must be preceded by a '0' to be properly interpreted as numbers and not strings. The default radix may be changed if desired (see "Changing the Radix," below).

4.2. Load the Program

The **:l** command loads an object module into the Chap.

```
> :l sample1.out
loading sample1.out...
stopped at 198:         push sample1;
```

The second message following the command tells us the processor had previously stopped at instruction 198 – a fact that is of no interest to us.

4.3. Display the Segment (load) Map

You can display a listing of the current memory allocations with the **$m** command.

```
> $m
file name       text        data
sample1.out     0-14        0-100f
Free: 16363 instructions 15356 pixels
```

The **text** column shows where the file's instructions are loaded in instruction RAM. The **data** column shows where the file's (combined) **.bss** and **.data** segments are loaded in scratchpad. These numbers are in hexadecimal format, while the **Free** statistics are in decimal.

4.4. Display the Symbol Table

The **$l** command causes the debugger to display a table of the value, type, and name of each locally defined symbol. Type this command using “l” for “list”, not the digit “1”:

```
> $l
sample1.out: 21 instructions at 0 data [0..1010]
    0 t sample1.o          0 d color              4 d rect
   10 b one_row            0 t sample1
```

The first line repeats the statistics given by the **$m** command. The local symbols, their value and type are then listed. The meaning of each type is shown in Table 4-1.

Type	Segment
t, T	text
b, B	bss
d, D	data
a, A	absolute

Table 4-1: Symbol Types

Symbols in upper-case are *externally visible*. That is, a symbol may be referenced outside the file in which it is defined. As an exercise, in sample program 2 we create symbols of this type. External symbols may be displayed with the **$e** command.

4.5. Displaying Scratchpad Memory

The general form of the command to read scratchpad is

address,count/format

address specifies the address of the first word to be displayed, *count* the number of times the following format is employed. If the address is omitted, the default value is 0. If the count is omitted, its default value is 1. *format* is a string of characters that specifies how to display the data. Some common format characters and their meanings are shown in Table 4-2. Lower-case characters refer to tessellated data; upper-case, to untessellated data.

Character	Meaning
x,X	Print a value in hex
d,D	Print a value in decimal
o,O	Print a value in octal
f,F	Print a value as coefficient type
e,E	Print a value as pixel type
a	Print the current address in symbolic format

Table 4-2: Format Characters

An optional integer *n* can precede any format character. This is equivalent to repeating that character *n* times.

We examine the loaded data:

```
>color,4/x
0:
color:      2cc       1c2       666       0
>color/4d
0:
color:      716       450       1638      0
>
rect:
rect:       100       250       100       250
>one_row/20d
one_row:
one_row:    0         0         0         0
            0         0         0         0
            0         0         0         0
            0         0         0         0
            0         0         0         0
```

There are a couple of things to notice here. First, observe that omitting the count and preceding the format string by '4' achieves the same effect as specifying a count of 4. Also, observe that a carriage return in response to the prompt causes the previous command to be repeated beginning at the current address. (*rect* follows *color* in memory). However, when the command is repeated, the count is set to 1. Thus, it is preferable to incorporate the count into the format part (as noted above) when you are examining large arrays of memory. That way, you can just hit carriage returns to see each row.

4.6. Number Representation

The assembler recognizes three fixed-point formats: *integer*, *coefficient*, and *pixel*. Integers have no fractional bits. Coefficient values contain 14 bits of fraction and have an implicit binary point between bits 13 and 14. Pixel values contain 11 bits of fraction with an implicit binary point between bits 10 and 11. Figure 4-1 compares the three types.

Pixel values are designed to be compatible with the 12-bit frame buffer (12 bits in each channel). The high-order four bits are discarded when a value is moved to the framebuffer, and the resulting 12 bits are interpreted as a value in the range [–.5, 1.5)*.

Coefficient values have a range of [–2, 2). They are useful for scaling constants, matrix elements, etc. A coefficient is represented in Chas programs as a decimal value containing a fraction followed by an 'F' or 'f' (for Fourteen bits). A pixel value is represented by a decimal value with fraction followed by an 'E' or 'e' (for Eleven bits). The values of `color` in sample1 are specified in the 'E' notation (see Figure 4-1). These values are converted by Chas to the 16-bit fixed-point word of the Chap. The hexadecimal (decimal) values for `color` are:

* The notation for intervals used here uses square brackets to denote closed intervals (which contain the endpoint) and parentheses to denote open intervals (which don't). Thus, [.5, 1.5) is the set of *x* such that *x* is greater than or equal to .5 and less than 1.5.

color: 2cc (716) 1c2 (450) 666 (1638) 0 (0)

Consider the value 716. The binary representation is

0000 0010 1100 1100

Inserting the binary point, the number becomes

0000 0.010 1100 1100 (binary) = 0.35 (decimal)

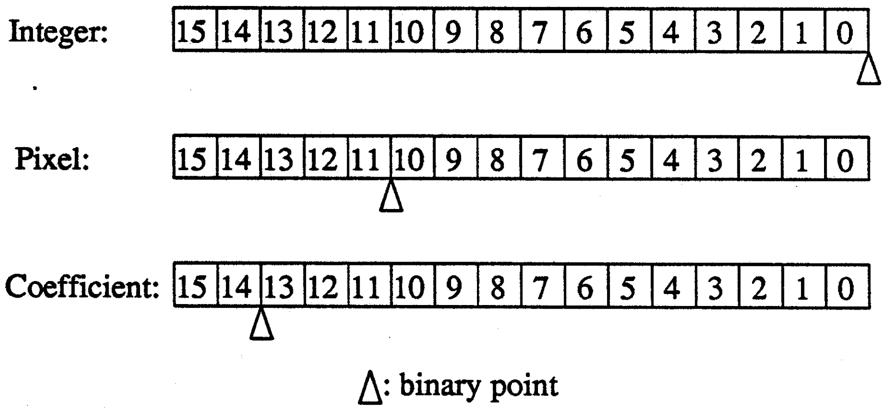


Figure 4-1: Chap Number Types

4.7. Accessing Program Memory

The general form of the command to read instruction memory is:

address,count?format

This command works the same as that for reading scratchpad locations.

An additional format character, **i**, is useful in that it interprets the accessed word as an instruction. Sample1 is short enough to print it out completely:

```
>sample1,13?ia
0:      b0 = rect;
1:      i0 = color+1;
2:      b0++; r8 = @b0;4 ticks;
3:      b0++; acc = @b0; 4 ticks;
4:      !sce; special; r8 = acc- r8;
5:      r10 = acc - r10;
6:      r10 = color+1 + r10; 2 ticks;
7:      b0++; r9 = @b0;4 ticks;
8:      b0++; acc = @b0; 4 ticks;
9:      !sce; special; r9 = acc- r9;
0a:     r11 = acc - r11;
0b:     r11 = color+1 + r11; 2 ticks;
0c:     b0 = 0;
0d:     r12 = (b0); 4 ticks;
0e:     b0 = one_row;
0f:     i0 = rect;
10:     push r10[0]; 2 ticks;
11:     whiledo4 not lczero otherwise 14; 2 ticks;
12:     (b0) = r12; b0++; 2 ticks;
13:     !mce; continue;2 ticks;
14:     bpt;
```

The disassembler produces code similar, but not identical to, the original source program. This *a* format causes the assembled program starting at sample1 (its entry point) to be "disassembled." The shorthand "b0++" stands for "b0 = b0 + i0." On line 2, "4 ticks" is the amount of time the assembler has allocated to execute this instruction. Note that the constant 1 appears as `color`+1 [line 1]. This is because the value of symbol `color` is 0. Charm assumes all constants are addresses, and tries to interpret them as offsets from known addresses. Finally, notice that the `loop–done` construct has been translated into its lower-level equivalent [10-13]. One can learn quite a bit about the assembler by comparing its preassembled code to its disassembled counterpart, or by reading the Chas document.

4.8. Runtime Control

The **:r** command starts program execution at the *address* specified.

```
>sample1:r
/dev/chap0: running
stopped at 14:     bpt;
```

Charm returns automatically when it encounters a breakpoint. This breakpoint is the one we inserted at the end of our sample program.

We may examine the results by listing the contents of `one_row`:

```
> one_row/20d
one_row:
one_row:    716      450      1638     0
            716      450      1638     0
            716      450      1638     0
            716      450      1638     0
            716      450      1638     0
```

The listing above shows the expected result: `one_row` has been filled with the contents of `color`. This list is `rxsize` lines long.

4.9. Breakpoints

Breakpoints may be inserted with the **:b** command and deleted with the **:d** command. We'll insert a breakpoint at the beginning of the loop at location 11 (hexadecimal):

```
> 11:b
```

then, restart the processor:

```
> sample1:r
/dev/chap0: running
breakpoint   11:     bpt; whiledo* not lc zero otherwise 14; 2 ticks;
```

The processor stops at the new breakpoint. We can single step one instruction using the :s command:

```
> :s
stopped at    10:     (b0) = r12; b0++; 2 ticks;
```

or continue execution with the :c command:

```
> :c
/dev/chap0: running
breakpoint   11:     bpt; whiledo* not lc zero otherwise 14;  2 ticks;
```

The program returns to the breakpoint after executing one iteration of the loop. Finally, we can delete the breakpoint:

```
> 11:d
```

then restart, executing the entire program and arriving at the end:

```
> sample1:r
/dev/chap0: running
stopped at 14:     bpt;
```

4.10. Examining registers

ALU registers can be examined with the **$r** command. To look only at those registers used by sample1:

```
> 8,5$r
r8[0]     64      r8[1]     64      r8[2]     64      r8[3]     64
r9[0]     64      r9[1]     64      r9[2]     64      r9[3]     64
r10[0]    96      r10[1]    96      r10[2]    96      r10[3]    96
r11[0]    96      r11[1]    96      r11[2]    96      r11[3]    96
r12[0]    2cc     r12[1]    1c2     r12[2]    666     r12[3]    0
acc[0]    0fa     acc[1]    0fa     acc[2]    0fa     acc[3]    0fa
sp        0
lc        0fa
rf        0f
12:       bpt;
```

The other registers displayed are the four accumulators (**acc**), the stack pointer (**sp**), the loop counter (**lc**), and the runflag (**rf**). Finally, the instruction currently pointed to by the program counter is displayed.

In a similar manner, the **$a** command may be used to display the base and index registers. Try it.

4.11. Changing the Radix

The **$d** command changes the default radix.

```
> 0a$d
radix=10  base ten
```

changes the default radix of the debugger to decimal. Note that the new radix must be specified in terms of the old one. Now input will be interpreted in base 10 and output displayed in base 10. Try the **$r** and **$a** commands to examine the registers. Notice that the register values are now displayed in decimal.

> **Exercise**: You've seen how the color (.35E, .22E, .8E, 0E) gets loaded into scratchpad. You can also change these values from within Charm. The command for writing a single word of scratchpad memory at address **foo** with the value val is:
>
> ```
> > foo/w val
> ```
>
> Your assignment is to overwrite **color**, **color**+1, and **color**+2 with new red, green, and blue values. Then rerun the program and verify that your new color has been used to fill **one_row**.

4.12. Getting out of Charm

To end the debugger session, enter the quit command:

```
> $q
```

or type ^D (Control-D).

Addendum for the Curious

Perhaps now you are wondering what the syntax of Charm commands is. Charm commands generally follow this format:

[*address*][,*count*]*verb modifier*

where bracketed arguments are (usually) optional. In this context, the *format* part of the memory access command is a *modifier* of the verbs '/' and '?'. Table 4-3 shows the most common verbs and their interpretations.

Verb	Interpretation
/	display scratchpad memory
?	display instruction memory
=	display the value of *address*
$	miscellaneous printing commands
:	runtime control

Table 4-3: Command verbs

References

The standard reference on Charm is the **Chap Runtime Monitor Reference Manual** in the **Pixar User's Manual.**

5. Sample2: Paint a Colored Rectangle

We finish the work we began in sample1 and paint a colored rectangle into the framebuffer memory. In the process, we learn about the following topics:

- condition codes and runflags
- subroutines
- calling and linking library subroutines

The listing of sample2 is shown in Figure 5-1. Sections of code unchanged from sample1 have been omitted. Once again, references to the listing are in square brackets.

```
 1  /*
 2   * Draw rectangle of fixed size & color into frame buffer
 3   */
 4      ...
 5  #define rtb      r13          /* new regs for library calls  */
 6  #define rpw      r14
 7      ...                       /* same as sample1 until ...   */
 8      acc = @b0; b0 = b0 + i0   /* read x1                     */
 9      if negative  (acc = acc - rx0) then
10          acc = -1; bpt
11      else
12          rxsize = acc          /* compute x dimension         */
13      fi
14      ry0 = @b0;   b0 = b0 + i0 /* read y0                     */
15      acc = @b0;   b0++         /* read y1                     */
16      if negative  (acc = acc - ry0) then
17          acc = -2; bpt
18      else
19          rysize = acc + 1      /* compute y dimension         */
20      fi
21      ...                       /* build one row               */
22      jsr initialize            /* subroutine jump             */
23      loop rysize do            /* copy one_row to framebuffer */
24          b0 = rpw
25          b1 = one_row
26          r0 = rxsize
27          r1 = rx0
28          r2 = ry0
29          jsr SFxCopy           /* library routine             */
30          ry0 = ry0 + 1         /* go to next line of rect     */
31      done
32      bpt
33  initialize:                   /* subroutine definition       */
34      jsr initstack
35      r0 = 0
36      r1 = 32
37      r2 = 32
38      jsr AllocTB
39      rtb = b0
40      r0 = 0
41      r1 = 1023
42      r2 = 0
43      r3 = 767
44      jsr OpenPW
45      rpw = b0
46      return
```

Figure 5-1: Sample2 Program Listing

5.1. Condition Codes and Program Flow

Chas includes high-level conditional constructs such as **if-then-else-fi**. In this sample program, a test is added to detect a negative value for the length of the rectangle [lines 9-13]. If this value is negative, then −1 is loaded into the accumulator, and the program halts using the **bpt** statement .

5.1.1. Syntax of "if ... else ... fi"

Note that the condition code is not in parentheses, while the value to be tested is. The **fi** and **else** statements must be on lines by themselves. The **else** clause is optional, but the **fi** is not. **zero**, **positive**, **overflow**, and **carry** are other valid condition codes. (see p. 13 of Chas ref. manual)

5.1.2. How Conditionals Work on the Chap

Conditional statements allow the programmer flexible control over which processors execute given blocks of code. The key to this control is the **runflag**. This is a 4-bit value that determines which of the processors executes the current instruction. The runflag is one of the three values which is maintained by the sequencer on the system stack, along with the loop counter and the program counter. Recall from the Charm session that the runflag (known as **rf**) is printed, along with the loop counter and the program counter, by the **$r** command.

In normal operation, the runflag is 1111 (binary) indicating all four processors are to run. The low order bit corresponds to processor 0 (the "red" processor), the next bit to processor 1, etc. Thus, a runflag value of 1000 indicates that only processor 3 is running.

Conditional statements generate a runflag to identify those processors that satisfy the condition. In the usual mode, this runflag is then ANDed with the runflag on the top of the stack, and pushed onto the top of stack. This is the effective runflag until the conditional block is terminated or another sequencer instruction pushes new values on the stack.

5.1.3. "All or nothing" control

In the way described above, processors may be selectively turned on and off depending on the data they contain. Using the key words **any** or **all** it is possible to exercise other types of control. For example,

```
if any !zero then
            ...
fi
```

will force all or none of the processors to execute the block. If any of the processors satisfies the condition code, then the runflag of 1111 is generated, forcing all processors to be active; else 0000 is generated. For example, suppose we want to process all pixels which are not zero in all 4 channels. Then we would use a construct as shown above. The **all** modifier works in a similar way. Any condition code may be preceded by a '**!**' or '**not**' to negate it.

5.1.4. Low Level Sequencer Control

The `if ... else ... fi` statement is a high-level construct. Serious assembler programmers will want access to the low-level constructs. See the appendix at the end of the **Chas Reference Manual** for a table of these.

5.2. Subroutines

The program sample2 calls a subroutine with the statement [line 22]:

```
jsr initialize
```

The `initialize` subroutine appears following the main program in the listing. `initialize` is an example of a user-defined subroutine. It calls three library subroutines [lines 34,38,44] to set up access to the framebuffer. Before going any further, it is important to understand the conventions associated with subroutines.

5.2.1. Library Subroutines: Sharing Registers

Library subroutines must follow certain rules with respect to the use of Chap registers. All the library routines supplied with the Pixar Image Computer follow these rules, and you are encouraged to do likewise in any library routines you write. Registers in the Chap are divided into three classes:

- *Volatile* registers are always available; a subroutine can use a volatile register without saving and restoring its contents.
- *Sacred* registers must be preserved; a subroutine must save the contents of these registers before using them and must restore their contents before returning.
- *Off-limits* registers are reserved for system use and may never be changed by user routines.

Table 5-4 shows the registers in each class.

class	ALU	base	index
volatile	`acc, r0-r7`	`b0-b3`	`i0-i3`
sacred	`r8-r29`	`b4-b13`	`i4-i13`
off-limits	`r30, r31`	`b14, b15`	`i14, i15`

Table 5-4: Register Volatility

This convention explains why we began allocating registers with `r8`, instead of `r0` in example 1. The sacred registers are guaranteed to be preserved across library subroutine calls, such as `SFxCopy`.

5.2.2. Library Subroutines: Calling Conventions

Subroutines pass parameters using ALU, base and index registers. Library subroutines, by convention, use registers in numerical order within each type. That is, **r0** is used before any other ALU register, etc.

Each library routine is described in a Unix manual page provided with the Pixar Image Computer. These descriptions follow a format which respects this convention. Particular examples are given below.

`initialize` [lines 33-46]

The first line [line 34] calls the library subroutine `initstack` to set up the stacks that push and pop groups of registers. Any program using library routines needs to include this call to `initstack`. This initializes the system stack, which is used to push and pop registers. Whenever a library routine uses sacred registers, it pushes the values of these registers upon entry and pops them upon exit. User routines may use these stack routines for the same purpose. For future reference, here are the stacking subroutines:

Registers	Save with	Restore with
sacred base	`pushb`	`popb`
sacred index	`pushi`	`popi`
sacred alu	`pushr`	`popr`
volatile	`pushv`	`popv`

Table 5-5: Register Stacking Subroutines

Any Chap program that transfers data to or from the framebuffer (via the Pbus) will probably use the library subroutines provided for this purpose, such as `SFxCopy`. To use these subroutines, the user must call initialization routines to allocate a piece of the framebuffer and define a window within that piece. This is the reason for the calls [lines 38,44].

`AllocTB` [lines 35-38]

The framebuffer memory is organized as a linear array of *tiles*, each of which is a square with 32 pixels on a side. In this ''raw'' state, it is not possible to create two-dimensional pictures since the tiles do not have well-defined vertical (up and down) neighbors. A call to `AllocTB` gives a rectangular form to this one-dimensional list forming a ''tile block'' and returning a pointer to the tile block.* This is done by providing a starting tile number, and a width and height (in tiles) of the desired rectangle. Given this information, it is possible to compute which tiles are vertically adjacent. The parameters used [lines 35-37] create an area 1024 (32 times 32) pixels on a side, and are sufficient for most ordinary applications.

If you examine the Pixar manual page for TB(3C) you will find the following:

int* **AllocTB**(firsttile, tilewidth, tileheight)
register firsttile, tilewidth, tileheight;

The actual calling sequence in a program doesn't look like this, since the assembler doesn't support argument passing to subroutines. In a real program, the desired values must be explicitly placed in the proper registers before jumping to the library routine. What the first line means is that `AllocTB` returns its result in **b0**. The second line says that `AllocTB` expects its three parameters in `r0,r1,r2`. We know the actual registers used from the convention that parameters are passed in the lowest-numbered

* In addition to calling `AllocTB` you may also need to set the video board of the Pixar Image Computer to agree with this call. In our case, the standard setting of the video board (an image 32 tiles or 1024 pixels wide) agrees with the call to `AllocTB` made here so nothing needs to be done.

registers first. The correspondence between parameters and registers is:

parameter	register	contents
firsttile	r0	upper-left tile
tilewidth	r1	tiles across the block
tileheight	r2	tiles down the block

AllocTB returns a pointer to the tile-block data structure in **b0**. The sample2 program saves this value in register **rtb** [line 39].

OpenPW [lines 40-44]

A *pixel window* is a rectangular window within a tile block. One of the parameters to every framebuffer library subroutine is a pixel-window identifier. All accesses are then clipped to this window. You may have more than one pixel window open within a tile block. **OpenPW** creates a pixel window within a specified tile block. The manual page for **OpenPW** (PW(3C)) reads:

```
base OpenPW(tb, minx, maxx, miny, maxy)
base tb;
register minx, maxx, miny, maxy;
```

Remember, this is a symbolic representation of the calling sequence and does not show how a call appears in a real program. The correspondence between parameters and registers is:

parameter	register	contents
tb	b0	pointer to tile block
minx	r0	minimum X value of window
maxx	r1	maximum X value of window
miny	r2	minimum Y value of window
maxy	r3	maximum Y value of window

Note that the X and Y values are in pixels, not tiles. The parameters used in sample2, [lines 40-43], create the window shown in Figure 5-2. This window happens to coincide with the standard video display of the Pixar Image Computer, but could be smaller or larger. The pointer to the pixel window is returned in **b0**. The sample2 program saves this value in register **rpw** [line 45] so that it may pass it to **SFxCopy** (3C) later.

After saving the pixel-window pointer, **initialize** executes a **return** statement [line 46] and execution resumes at [line 23] where a loop begins. The main feature of this loop [lines 23-31] is a call to a library subroutine, **SFxCopy**.

SFxCopy [line 29]

Library subroutines drive the Pbus, which is between the Chap and the framebuffer. **SFxCopy** is one of several subroutines available for transferring data from scratchpad to framebuffer. 'SFx' indicates that this is a scratchpad to framebuffer copy, in the x, or horizontal, direction. There is an analogous subroutine, **SFyCopy**, which copies into the framebuffer in a vertical direction. The manual pages have the following entry for **SFxCopy**:

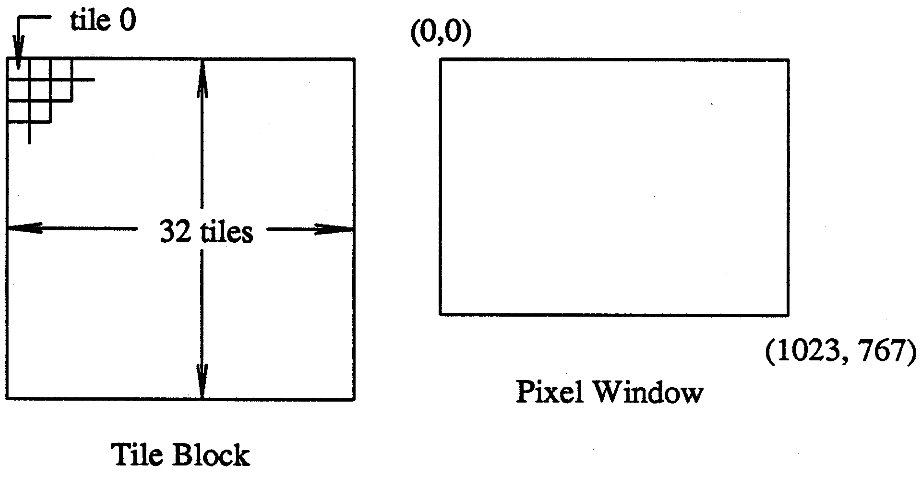


Figure 5-2: Tile Block and Pixel Window

SFxCopy(pw, source, n, x, y [, z])
int *pw;
pixel *source;
register n, x, y [, z]

The correspondence between parameters and registers is:

parameter	register	contents
pw	**b0**	Pixel window
source	**b1**	Address of scanline
n	**r0**	The number of pixels to be copied
x	**r1**	X of starting point in pixel window
y	**r2**	Y of starting point in pixel window
z	**r3**	Volume slice (optional)

Instructions [lines 24-28] set up the parameter values for the call to **SFxCopy**. The horizontal span we put into **one_row** is being repeatedly copied into the framebuffer. Each iteration includes incrementing the Y address of the target. The result is that a rectangle is painted into the frame buffer one line at a time.

5.3. Assembling sample2

To create an executable version, issue the command:

```
% chc sample2.s –o sample2.out /usr/pixar/chap/lib/libpx.a \
/usr/pixar/chap/lib/libpt.a /usr/pixar/chap/lib/libpG.a \
/usr/pixar/chap/lib/libpm.a
```

This assembles the program and links the library subroutines into a single loadable module named ''sample2.out.'' Notice the libraries to be searched to locate references to the library subroutines. The libraries included here contain almost all of the available

Chap library subroutines.

You can simplify the process of assembling the sample programs by using the **make** command. The file 'makefile' in the tutorial directory contains the commands that assemble the sample programs. For example, to assemble sample2.out, use the command

```
% make sample2.out
```

This has the same effect as the explicit command above.

5.4. Running sample2

As an alternative to using Charm to load and run the program, we can also use the host program **chload**. The commands

```
% chmap –i
% chload sample2.out
```

will load sample2.out into the Chap and set the processor running at instruction 0. Do you see a light-violet rectangle in the upper-left corner of your display?

Now run Charm. Change the values of `color` and `rect`, then restart the program from the beginning and observe the results. Charm knows about the symbols in your program, even though you didn't use Charm to load the program. The symbol table is maintained independently. See the discussion of the dynamic loader below for details.

Exercise: Using the `.globl` directive. It is possible to isolate subroutines in separate files. In order to make the subroutine *foo* visible to programs in other files, it is necessary to include the statement

```
.globl foo
```

Try applying this to sample2, by making a separate file for the `initialize` subroutine.

Exercise: Using SFyCopy. There is no reason to prefer the 'x' to the 'y' direction. Rewrite sample2 so that `SFyCopy` is used to access the framebuffer.

Exercise: A Random Rectangle Generator. Modify sample2 to display rectangles an endless sequence of random size, location, and color. Use the random-number-generator library function `rrand`, which produces four random 16-bit integers in `acc`. Compare your solution to sample2a.s in the tutorial directory.

[Hints: generate x0, y0, xsize, and ysize directly, by masking the result of `rrand` to appropriate ranges. For example, x0 and y0 can have 10 bits of significance (0-1023), while the x and y dimensions could restricted to 8 bits (0-255). As in the language C, the masking operator is '&'. Refer to the next sample program to see how to generate a sequence of rectangles under user control.]

6. Sample3: In Which the Host Plays a Role

Our goal now is a program on the host machine that will accept rectangle descriptions from the user sending them to a Chap program for painting them.

The program sample3 builds on sample2. The new features are:

- synchronization of host and Chap programs
- data transfer between host and Chap
- low-level sequencing constructs in Chas

We discuss the Chap program first, then the host program that calls it.

6.1. Sample3: Chap side

Figure 6-1 shows the incremental changes made in sample2 to produce sample3.

```
 1              ...
 2              .text
 3        sample3:
 4              jsr initialize
 5              while true do            /* program loop */
 6                    sysbus<13> = 0
 7                    push               /* wait for signal */
 8                          acc = sysbus<13>
 9                    dowhile zero
10
11                    b0 = rect
12                    i0 = 1
13                    @b0 = sysbus<0>; b0 = b0 + i0
14                    @b0 = sysbus<1>; b0 = b0 + i0
15                    @b0 = sysbus<2>; b0 = b0 + i0
16                    @b0 = sysbus<3>; b0 = b0 + i0
17                    b0 = color
18                    i0 = 1
19                    @b0 = sysbus<4>; b0 = b0 + i0
20                    @b0 = sysbus<5>; b0 = b0 + i0
21                    @b0 = sysbus<6>; b0 = b0 + i0
22                    ...
23                    interrupt = 1
24              done                  /* program loop end */
25              bpt
```

Figure 6-1: Sample3 Chap Program Listing

Outer loop [lines 5-24]

In order to make the Chap program continually draw rectangles, we bracket the program within an endless loop:

```
while true do
       /* main program */
done
```

true is another condition code, used here to ensure that the program loops indefinitely. Like the **if** statement in sample2, the **while** statement affects the runflag, so that only the processors satisfying the condition code execute the statements within. The call to **initialize** [Line 4] lies outside the loop, since it should be performed only once.

Synchronization

Two programs running on separate machines require some synchronization protocol if they are to work together. One popular scheme is to use a shared memory location. With this approach, one party signals the other by writing a preset value to a common memory location. In the case of the Chap and its associated host, a set of 16 Sysbus registers can be read or written by either machine. These registers are known to the assembler as **sysbus**<*value*>, where *value* is in the range 0 to 15. Sample3 uses **sysbus<13>** to receive a signal from the host. Each time the Chap arrives at the top of the main loop, it sets this register to zero and waits until the value changes.

Synchronization loop [lines 7-9]

The following code detects when **sysbus<13>** changes to a non-zero value.

```
push
        acc = sysbus<13>
dowhile zero
```

This is an example of a "low-level" sequencing structure, so-called because, unlike the constructs we have encountered so far, it makes explicit reference to the sequencer. (The "high-level" constructs are built up from low-level constructs by the assembler.) The stack has three fields that are always pushed and popped in unison: the loop counter, the runflag, and the program counter.

The **push** statement stacks the values of the loop counter, the runflag, and the address of the next instruction. The **dowhile** statement evaluates a condition code, in this case generated by the previous ALU operation [line 8]:

```
acc = sysbus<13>
```

If the condition is true, that is **sysbus<13>** is still zero, the address of the next instruction to be executed is obtained from the top of the stack. If not, the stack is popped and execution proceeds to the next instruction following the **dowhile**. Even though the loop counter and the runflag had no bearing on the execution, they are pushed and popped also.

Once execution has passed beyond the **dowhile** delay, the program obtains the rectangle description from seven Sysbus registers, where the host program has written them, and stores the values in the **rect** and **color** data structures [lines 11-21]. The code inherited from sample2 to paint the rectangle is then executed. This done, execution goes back to the top of the loop after executing the statement:

```
interrupt = 1
```

Let's turn our attention to the host side before we explain the purpose of this statement.

6.2. The host program

The host program is written in C. Figure 6-2 shows a listing.

```
 1      #include <pixar/pixar.h>
 2      #include <stdio.h>
 3
 4      main()
 5      {
 6            int x0, y0, x1, y1;
 7            double R, G, B;
 8            CHAP *chap;
 9
10            chap = ChapOpen("/dev/chap0",1);
11            ChapMMan(chap,0);
12            for (;;)      {
13                  printf("enter x0, x1, y0, y1, R, G, B0);
14                  if (scanf("%d%d%d%d%lf%lf%lf",
15                              &x0, &x1, &y0, &y1, &R, &G, &B) < 7)
16                        break;
17                  ChapSetSDR(chap, 0, x0);
18                  ChapSetSDR(chap, 1, x1);
19                  ChapSetSDR(chap, 2, y0);
20                  ChapSetSDR(chap, 3, y1);
21                  ChapSetSDR(chap, 4, (short)((1 << 11) * R));
22                  ChapSetSDR(chap, 5, (short)((1 << 11) * G));
23                  ChapSetSDR(chap, 6, (short)((1 << 11) * B));
24                  ChapSetSDR(chap, 13, 1);
25                  ChapWaitForInterrupt(chap);
26            }
27            ChapClose(chap);
28      }
```

Figure 6-2: Sample3 Host Program Listing

General remarks

This host program uses *libpixar*, the lowest-level library in the Pixar Software Release for host control over the Chap (higher-level packages are **Chad** and **Pirl**; the former is built largely of routines from *libpixar*). *libpixar* includes routines for accessing the Chap's registers, loading instruction and scratchpad memory, controlling the PIC's video board, and managing the global symbol table. The routines in *libpixar* are distinguished by names beginning with 'Chap'. Section 11 of this document is devoted to discussing *libpixar*.

Notice the statement

```
#include <pixar/pixar.h>
```

This header file defines a number of structures and constants used by programs calling routines in *libpixar*. For many simple Chap interface programs, this will be the only file included.

Every host program that interfaces to the Chap must **open** the Chap before it does anything and close it when it is finished [line 10]. The call to **ChapOpen** returns a pointer to a (data) structure. This pointer is used whenever we access the chap.

The call to **ChapMMan** [line 11] ensures that interrupts from the Chap will be passed directly to the host program and not intercepted by the operating system. See the manual pages for further documentation on this procedure.

In this sample program, the exchange of data between host and Chap takes place using the **sysbus** registers, a bank of 16 16-bit registers that can be directly read or written by either machine. The host can set the sysbus register **n** to the value *v* with the statement

```
ChapSetSDR(chap, n, v);
```

and can read the value of that register by the statement

```
v = ChapGetSDR(chap, n)
```

Observe how the host program mirrors the structure of the Chap program. There is a main loop during which one rectangle is processed. In the host, this involves prompting for and receiving a set of numbers describing the location and color of the rectangle [lines 13-14], and then sending the rectangle to the sysbus registers in the Chap. Each rectangle is passed to the Chap using the first 4 of these registers [lines 17-20]. Notice how the floating-point color values are converted to fixed-point *pixel* type before transfer [lines 21-23].* After each rectangle has been sent, the host program sets `sysbus<13>` to a nonzero value. This signals the Chap, as we have already described. Next we will see how the Chap signals the host.

Signaling: Chap to host

When synchronizing the host and Chap, it is a good idea not to force the waiting host into a ``tight loop'', where it polls a memory location over and over. Instead, the Chap signals with a hardware *interrupt.* Until it receives an interrupt, the host program pauses at the statement

```
ChapWaitForInterrupt(chap);
```

The Chas statement

```
interrupt = 1
```

raises a hardware interrupt that is sent to the host program. When the host program receives this interrupt, it is released from its wait state and proceeds to the top of the loop.

Notice that the synchronization presented here is asymmetric. The Chap signals the host by raising interrupts; the host signals the Chap by writing a memory location. Consequently, the host does not have to poll – the Chap does. This is appropriate since the host typically is supporting several users and thus has less time to waste than the Chap.

6.3. Running Sample3

Assemble both programs shown above using the make commands 'make sample3.out' and 'make sample3s'.

Using Charm, load sample3.out and start it running. Type ^C (Control-C). This interrupts the processor. The program should be in the `dowhile` loop, waiting for the host to signal. Restart the program and leave charm by entering ^Z (Control-Z).

Run sample3s on the host. Answer the prompts with values for the rectangle coordinates and the color values. Observe the display. You should be able to place rectangles

* This is equivalent to using the Chad macro **DBL2PXL.**

arbitrarily around the screen. Type ^D (Control-D) to exit.

> **Exercise: Graceful Exit.** Replace the `true` condition code in the `while ... do` construction [lines 5-24] so that the Chap program drops out of its main loop when the user signals termination. (Hint: use a special non-zero value in `sysbus<13>` to indicate completion).

> **Exercise: Error conditions.** As written, the sample Chap program terminates if it is passed an invalid rectangle description. Devise a way to make it recover from this error, either by correcting the rectangle description or (more difficult) by signaling the specific error to the host.

References

The *Chas Reference Manual* contains a handy table of the low-level sequencer instructions. The manual pages for **libpixar** (3H) are useful for an overview of the available low-level host interface routines.

7. Sample4: Transparent Colored Rectangles

This final sample Chap program shows how to use the fourth (*alpha*) channel to implement transparency. We extend sample3 to let the user *merge* rectangles of variable transparency into framebuffer memory. We introduce the following new topics:

- use of the multiplier, and
- use of the alpha channel to achieve transparency.

Figure 7-1 shows the incremental changes in the Chap program. The host program remains the same as in sample3, except that now it also prompts for an alpha value.

More storage [lines 2-6]

Notice that we have allocated space for *two* rows of pixel data. **old_row** is needed to hold the existing framebuffer image, while **new_row** is needed to hold the new rectangle data to merge over the old data.

The alpha channel

We interpret the alpha channel (**color**[3]) as an 11-bit fraction measuring the opacity of the rectangle. A value of 1 (that is 1.0E or 0x800) means full opacity and a value of 0 means full transparency. The library subroutines follow this convention when they merge two arrays of pixel data. They also expect the red, green, and blue components to be pre-scaled by the alpha channel. For example, if the alpha channel is 0, the other channels must also be set to 0 for the library subroutines to work properly.

The program segment to pre-scale the color data is [lines 13-17]:

```
b0 = rcolor[3]
multx = (alpha)b0
multy = (comp)rcolor
nop
rcolor = msp; runflag = 7
```

Selecting the alpha channel [lines 13-14]

When assigning from a 4-way register to a scalar device, such as a base register, any of the four processors may be chosen by appending to the register name the processor's number enclosed in square brackets. **rcolor[3]** accesses the alpha channel of the color. We have to put the alpha value into a scalar register such as **b0** before loading it into the multiplier input. This is because there is no direct data path between the registers of different processors (i.e., between **r0[0]** and **r0[1]**). The data must first be sent to a commonly accessible register, such as **b0**.

Setting the runflag [line 17]

Recall that the runflag is a 4-bit value where a '1' bit allows the associated processor to execute instructions. It is possible to set the runflag for a single instruction with the **runflag** assignment statement. Since it is not meaningful to scale the alpha value by itself, we want the alpha processor disabled when we get the scaled values from the multiplier. A runflag of 7 (0111 binary) does this.

Note well that the **runflag =** construct does **not** set the runflag directly: it obtains the new value of the runflag by performing a bitwise AND of the old value and

```
      ...
      .bss
old_row:
      .space 1024*4
new_row:
      .space 1024*4
      .text
sample4:       .globl sample4
      ...
            @b0 = sysbus<7>; b0 = b0 + i0
      ...
      rcolor = (b0)

/* prescale rgb by alpha */

      b0 = rcolor[3]             /* load the alpha           */
      multx = (alpha)b0
      multy = (comp)rcolor       /* load the rgb             */
      1 ticks                    /* wait for result          */
      rcolor = msp; runflag = 7
      b0 = new_row
      i0 = 4
      loop rxsize do             /* build one row            */
            (b0) = rcolor; b0 = b0 + i0
      done
      loop rysize do
            b0 = rpw             /* get the old row          */
            b1 = old_row
            r0 = rxsize
            r1 = rx0
            r2 = ry0
            jsr FxSCopy
            b0 = new_row         /* merge new over old       */
            b1 = old_row
            b2 = old_row
            r0 = rxsize
            r1 = 1.0F
            r2 = 1.0F
            jsr SSMergeOver
            b0 = rpw             /* write out the merged row */
            b1 = old_row
            r0 = rxsize
            r1 = rx0
            r2 = ry0
            jsr SFxCopy
            ry0 = ry0 + 1
      done
          ...
      bpt
```

Figure 7-1: Sample4 Program Listing

the given constant. This reduces to an assignment only when the runflag was 1111 (binary) before.

Filling new_row [lines 18-22]

With the newly computed value of rcolor, we build up new_row just as we once built one_row.

Merge the Rectangle [lines 23-44]

To perform a merge, we call SFxCopy's partner, FxSCopy, to copy the existing framebuffer data into the scratchpad at old_row [lines 24-29]. Next we merge old_row with new_row [lines 30-36].

Chap library subroutines provide a variety of ways to combine two pixel values. These subroutines have names beginning with SSMerge (the 'SS' indicates scratchpad to scratchpad merging). For this example, we use the subroutine SSMergeOver to combine the new values with the existing values.

SSMergeOver merges a specified number of pixels from one scratchpad location (the foreground) over the same number of pixels in another location (the background), writing the result into a third scratchpad location (the target). The calling sequence is:

```
SSMergeOver(frgd, bkgd, target, n, Lf, Lb)
pixel* frgd, bkgd, target;
register n, Lf ,Lb;
```

The correspondence between parameters and registers is:

parameter	register	contents
frgd	b0	foreground pixels
bkgd	b1	background pixels
target	b2	target
n	r0	number of pixels
Lf	r1	foreground weighting coefficient
Lb	r2	background weighting coefficient

The results are computed according to the equation

$$R = Lf * F + (1 - Fa) * Lb * B$$

where

F = value of foreground pixel
B = value of background pixel
Fa = alpha value of foreground pixel
Lf = weighting factor of foreground
Lb = weighting factor of background

Lf and Lb are coefficients, provided to give an overall relative weighting to the foreground and background pixels in the output.

After merging the two lines, we copy the result to the framebuffer as before using SFxCopy [lines 37-42] and go on to the next line [line 43], until the full rectangle is composed.

Using the multiplier

Each processor includes a multiplier. The multiplier has two inputs from the Mbus: **multx** and **multy**. The output is a 32-bit value, with the high-order 16 bits (most significant part) contained in **msp** and the low-order 16 bits (least significant part) in **lsp**, which feed the Abus. The output of the multiplier is available one instruction cycle after loading the second input. The **nop** (signifying **no operation**) statement [line 16] is used to wait for the product to be ready.

It is possible and usually necessary to shift the inputs to the multiplier so that the product will be in one of the standard fixed-point formats. Let us assume we want to multiply an alpha value (11 bits of fraction) by a color value (11 bits of fraction) and produce a product that also contains 11 bits of fraction. If we were to multiply the alpha and color values without shifting, the 32-bit result would have 11 + 11 = 22 bits of fraction. The **msp** (the high-order 16 bits of the product) would contain only 6 bits of fraction, which is not a standard fixed-point format. However, if before multiplying the values we were to shift the alpha value two bits left giving us a value with 13 bits of fraction and shift the color value three bits left giving us a value with 14 bits of fraction, the product would contain 13 + 14 = 27 bits of fraction. Now the **msp** contains 11 bits of fraction and is in the standard pixel-component format. **Warning:** *these left shifts can cause high-order bits of the multiplicands to be lost.* You must always consider the range of the data involved before blithely asserting these shifts. In some cases it is necessary to (painfully) reconstruct a 16-bit result from the two 16-bit outputs of the multipliers. This loss is not a problem here since the numbers shifted are pixel components, so that their significance is contained in their low-order 12 bits.

The number of bits to shift is specified by prefacing the value when it's loaded into either **multx** or **multy** with one of the following modifiers enclosed in parentheses:

modifier	left shift	type of input
coeff	0	for coefficients with 14 bits of fraction (default)
comp	2	color pixel values
alpha	3	alpha pixel values

The shift modifier can be preceded by either **unsigned** or **signed** to control whether the multiplicand is interpreted as an unsigned magnitude or as a signed number. The default modifier is (**unsigned coeff**).

Exercise: Run sample4 (following the procedure for sample3). Remember that an alpha value of 1.0 gives an opaque rectangle, and 0, a fully transparent one. Specify overlapping rectangles to verify that the transparency works.

Exercise: In `SSMergeOver`, notice the parameters *Lf* and *Lb*. We have set these parameters to 1.0 for all our merges. Experiment with different values to get a feel for their semantics.

Exercise: Read the manual pages for `SSMerge`. Experiment with other merge subroutines instead of `SSMergeOver`.

Exercise: **Random Rectangles Revisited.** In sample2, one of the exercises presented an alternative program in sample2a.s for generating an endless stream of random rectangles. sample4a.s updates that program to include random transparency. Make sample4a.out and run it stand-alone. Do you notice anything funny? Your assignment is to figure out what the bug is and explain it. For the answer, see * below.

*Answer: The alpha channel of *mask2* should be 0x7ff, not 0x7fff.

8. Further Topics

Component addressing

This tutorial has covered two addressing modes available in the Chap: *broadcast* and *pixel*. There is a third popular mode, *component*, which accesses a single color channel in four successive pixels. The default channel is red. For example

```
r0 = <b0>
```

reads the red channel while

```
r0 = <b0,1>
```

reads the green channel of the four pixels starting at the address given by `b0`. Figure 8-1 illustrates component mode read. Component mode write results if the arrows in the figure are reversed.

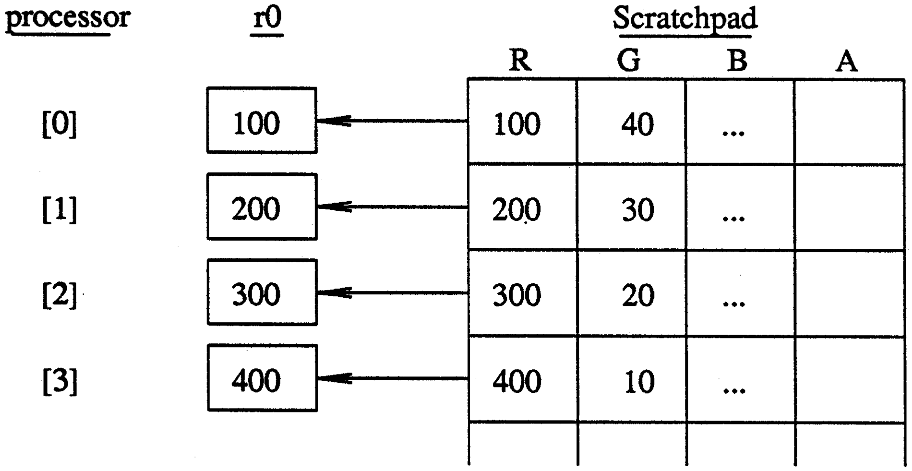


Figure 8-1: Component Mode Read: "`r0 = <b0>`"

Exercise: Go back to the sample programs and invent a variation that uses component addressing somewhere. For example, instead of using pixel mode writes to fill `one_row`, use separate component mode writes for each channel.

The following code shows how to fill the red component of `one_row` with the red component of `color`.

```
b1 = color
i1 = 1
b0 = one_row
i0 = 16
rcolor = @b1;     b1 = b1 + i1
acc = rxsize
push
        <b0> = rcolor;  b0 = b0 + 10
        acc  = acc - 4
dowhile !negative
```

Here we introduce a new looping construct: `push ... dowhile`. Look it up in the table on sequencer instructions in the *Chas Reference Manual* to see how it works. The ''loop counter'' in this case is the `acc` register. It begins with the value `rxsize`, the width of the rectangle. Each iteration within this loop writes the red channel of four pixels; hence the loop counter is decremented by four on each iteration.

Notice that the number of pixels written by the above code segment is always a multiple of 4. This is generally true of component mode access. For this reason, whenever you use component reads or writes, it is a good practice to allocate scanline storage in scratchpad to be aligned on 16-word (4-pixel) boundaries, by using the `.align` directive (see the *Chas Reference Manual*). Otherwise, the programmer has to to treat the end conditions of his loops with extra care to avoid clobbering contiguous storage.

There is a small but tenacious quirk associated with component accesses: the index register specified by the instruction must contain the value 16 for the access (read or write) to succeed. Notice that the above code segment satisfies that condition, since `i0` is set to 16 before entering the loop, so `b0` will advance four pixels following each write. Generally, it is a good idea to use `i0` for this purpose, since it is the default index register specified by the assembler. In this way, if you make a component access but do not increment the base register, the index register specified on the instruction will be `i0`. You will have made sure that it contains the value 16, and all will be well.

9. Programming Pitfalls

If you have worked through this tutorial this far, you have learned to appreciate the unique 4-way architecture of the Chap. The same features that give new elegance and speed to your programs, however, can sometimes cause new problems. This brief discussion will attempt to bring to your attention some of the less obvious features of the Chap.

4-Way Registers, Scalar Registers, and Runflags

Some registers, such as the general purpose ALU ones, are present in each of the four processors; others, such as the base and index ones, exist only on the Sbus. This division underlies the powerful addressing options available to the user. However, this can also create subtle bugs in your programs. Mainly, you need to **be careful when assigning to a scalar register from a 4-way register**. Two things may go wrong: first, you may neglect to specify the intended processor to use as source. Here, use the syntax already introduced. For example, say

```
b0 = r0[2]
```

to use processor 2 as the source. Second, even if the intended processor is specified correctly, the runflag may not allow that processor to be active; hence the assignment will not occur. To overcome the second possibility, the `force` statement (see the table of sequencer instructions in the appendix to the *Chas Reference Manual*) is available. This will force the runflag to take a specified 4-bit value. Use this inside a 4-way conditional clause to turn on a given processor when you cannot be sure that it will be active.

"`Runflag =`" vs. `force`

Note that the `runflag` statement already introduced differs from the `force` statement: the former **ands** a runflag with the current runflag for only the current instruction, while the latter actually pushes a new runflag on the stack. Question: why is it impossible to solve the problem raised in this discussion using only the `runflag` statement? For the answer, see * below.

Library calls and runflags

A related point concerns the use of library routines inside of conditional clauses. In general, **library routines expect the runflag to be fully on**, that is, to have the binary value 1111, or the hexadecimal value **0xf**. Thus, the `force` command should be used before calling a library routine if some processor may be inactive. This command must be paired with a `pop` command following return from the library call.

* Answer: The runflag statement will *not* turn on processors; it can only turn them off.

10. Sample5: The host part completed

This final example completes the host side of the program by having it automatically load the Chap code, as needed, each time the program is run. We introduce the following new topics:

- the dynamic loader
- the dynamic loader's symbol table

To explain the actions of the host program, we must introduce two important concepts: *relocation* and *link-editing*.

10.1. Relocation

The Chap assembler generates files containing *relocatable* object code. That is, files generated by **chas** contain sufficient information to allow the instructions contained in the file to be loaded into instruction memory at any location. On the other hand, the output of the Chap link editor, **chld**, is usually a non-relocatable object file; all instructions are generated with the assumption they will be loaded starting at location 0*.

10.2. Link editing

When a program is constructed from multiple files, it is common to find code in one file referencing symbols in other files. For example, a common data structure might be defined in one file as,

```
        .globl DataStructure
DataStructure: .space 44
```

while in another file it is referenced with,

```
b0 = DataStructure;
```

Similarly, the practice of placing subroutines in individual files and then organizing them into libraries involves the use and definition of externally visible symbols.

The process of resolving references to external symbols is termed *link-editing*. At its simplest, the link-editing process entails filling in references to the undefined symbols with the symbols' values. The Chap link-editor normally performs this process when invoked from *chc*. As we will see, however, there are several advantages to delaying this work until code needs to be loaded into a Chap.

* Chld does support options to fix the base of the code and/or data segments at locations other than zero. To generate a relocatable output file, see **chld** (1).

10.3. The Dynamic Loader

The dynamic loader allows you to link-edit and relocate object files at program execution time. In all our previous work, we have performed these operations prior to loading our code into the Chap. As a result, we have always loaded a non-relocatable, completely linked object module into the Chap. By delaying these two steps until we actually load the code, we gain several advantages:

- code may be loaded at any available location
- references to external symbols may be resolved by linking to symbols already loaded in the Chap
- files never contain out-of-date library routines

The first two points are key. Programs using more code than can fit in the Chap's instruction and/or scratchpad memories must resort to some 'paging' scheme; the dynamic loader provides a significant part of the mechanism needed to implement such a scheme. Further, by allowing symbol references to be bound at runtime, based on the existing contents of the Chap's instruction and scratchpad memories, multiple unrelated modules may coexist in a Chap; this speeds the operation of programs calling commonly used library routines, since they need not reload code into the Chap.

10.3.1. The Dynamic Loader's Symbol Table

Key to the operation of the dynamic loader is a host-resident symbol table file maintained for each Chap. This file contains information describing the current contents of the Chap's instruction and scratchpad memories. In normal operation, a symbol table file is updated only when code is loaded into or unloaded from a Chap. However, sophisticated systems that allocate resources at runtime, such as scratchpad memory, must keep up-to-date records in the symbol table.

10.3.2. Creating Relocatable Object Modules

To insure that your code is relocatable you should either supply the **–c** flag to chc,

```
% chc –c module1.s module2.s ...
```

or the **–r** flag to *chc* (for *chld*),

```
% chc –r module1.s module2.s ...
```

The first scheme is preferred as monolithic object modules usually contain copies of library routines that may already be present in the Chap. As there is no way to "break apart" a monolithic module at load time, attempting to load such a module, when copies of its routines are already resident, will lead to errors and nothing being loaded.

10.3.3. Dynamic Loading From Charm

The **:l** command in charm invokes the dynamic loader. The **$m**, **$l**, and **$e** commands display information derived from the contents of the Chap's symbol table. To load and link-edit multiple files from within Charm, use multiple **:l** commands. In a previous Exercise, you were asked to separate the initialization code into a separate file. Suppose you have placed this code in a file *init.s*. Then, to load the completed program, you could either link edit the two files with:

```
% chc –r –o sample2.out init.s sample2.s /usr/pixar/chap/lib/libpG.a \
/usr/pixar/chap/lib/libpt.a /usr/pixar/chap/lib/libpx.a /usr/pixar/chap/lib/libpm.a
```

and load it as before, or assemble the two files into two relocatable object modules

```
% chc –c init.s sample2.s
```

and then load the two individually:

```
% setenv CHAPDEBUG 6 †
% charm
Chap Runtime Monitor, version 3.1 of Tue Aug 19 21:11:26 PST 1986
> :l init.o
loading init.o...
loading stack.o from /usr/pixar/chap/lib/libpG.a...
loading pw.o from /usr/pixar/chap/lib/libpt.a...
loading reciprocal.o from /usr/pixar/chap/lib/libpG.a...
loading rec15_256.o from /usr/pixar/chap/lib/libpG.a...
stopped at reciprocal32+0a8:        bpt;
> :l sample2.o
loading sample2.o...
loading sfxc.o from /usr/pixar/chap/lib/libpt.a...
stopped at sample2+2d:    bpt;
```

Note that, as each file was loaded, the dynamic loader automatically added various files from the libraries. Each file referenced symbols defined in files contained in these libraries. Had we loaded our files in the opposite order, the following would have occurred:

```
% charm
Chap Runtime Monitor, version 3.1 of Tue Aug 19 21:11:26 PST 1986
> :l sample2.o
loading sample2.o...
Undefined:
initialize
loading sfxc.o from /usr/pixar/chap/lib/libpt.a...
loading pw.o from /usr/pixar/chap/lib/libpt.a...
loading reciprocal.o from /usr/pixar/chap/lib/libpG.a...
loading rec15_256.o from /usr/pixar/chap/lib/libpG.a...
stopped at InqTB+4:        i0 = acc[0]; 2 ticks;
> :l init.o
loading init.o...
loading stack.o from /usr/pixar/chap/lib/libpG.a...
stopped at InqTB+4:        i0 = acc[0]; 2 ticks;
```

After the load of sample2.o, the loader notified us that some file contained a reference to a symbol, **initialize**, it could not resolve from any of the libraries. This undefined

† This enables the **loading** debug messages, which are usually suppressed. See *ChapLoad*(3H).

reference was resolved after loading the second file.

10.3.4. Searching Libraries

In the above examples, we saw the dynamic loader automatically ``pull in'' library routines as they were needed. How did it know where to look for these routines? For the moment, we'll consider this problem under charm. The answer is slightly different when loading code from a program, as we will soon find out.

Charm sets up a default list of libraries for use by the loader. To see this list, type the **:a** command:

```
>:a
/usr/pixar/chap/lib/libpip.a:/usr/pixar/chap/lib/libpx.a:
usr/pixar/chap/lib/libpt.a:/usr/pixar/chap/lib/libpG.a:
/usr/pixar/chap/lib/libpm.a:usr/pixar/chap/lib/libcolor.a:
/usr/pixar/chap/lib/libchad.a
```

The list of libraries includes all the standard libraries, as well as an extra library, **libcolor.a**, which contains routines for performing color correction.

To alter this list, you can use either the **:a** or **:A** commands. The former replaces the current list of libraries with a new one, while the latter *adds* a list of libraries to the existing list (for either command, duplicate library names are ignored). Thus, if we were to place our initialization code in a private archive, say ``libinit.a'', we could automate our loading of sample2 as follows.

```
% ar cr libinit.a init.o
% chranlib libinit.a
% charm
Chap Runtime Monitor, version 3.1 of Tue Aug 19 21:11:26 PST 1986
>:A libinit.a
/usr/student/libinit.a:/usr/pixar/chap/lib/libpip.a:
/usr/pixar/chap/lib/libpx.a:usr/pixar/chap/lib/libpt.a:
/usr/pixar/chap/lib/libpG.a:/usr/pixar/chap/lib/libpm.a:
/usr/pixar/chap/lib/libcolor.a:/usr/pixar/chap/lib/libchad.a
>:l sample2.o
loading sample2.o...
loading init.o from /usr/student/libinit.a...
loading sfxc.o from /usr/pixar/chap/lib/libpt.a...
loading stack.o from /usr/pixar/chap/lib/libpG.a...
loading pw.o from /usr/pixar/chap/lib/libpt.a...
loading reciprocal.o from /usr/pixar/chap/lib/libpG.a...
loading rec15_256.o from /usr/pixar/chap/lib/libpG.a...
stopped at AllocTB+0b:    @b1 + i0 = r1; b1 = color+2; 2 ticks;
```

10.3.5. Miscellaneous Commands

There are several other charm commands related to the dynamic loader and the symbol table. Two of particular interest are the **:u** command, which *unloads* the specified file (i.e., removes it from the symbol table and frees up any associated resources), and the **$u** command which displays any undefined symbols in code currently loaded in the Chap. While the dynamic loader notifies you of undefined symbols, their existence does not preclude your executing the code loaded into the Chap. This fact can be particularly useful in prototyping a large program, as you can construct your program one piece at a time and load files into the Chap as they are ready to be tested. The dynamic loader places a breakpoint at any instruction referencing an undefined symbol, so if your program should execute an instruction attempting to utilize an undefined symbol, it will stop before performing a potentially incorrect operation.

10.3.6. Dynamic Loading From The Host

We can achieve the same results from within the host program. Figure 10-1 shows a listing of the incremental changes in the host program beyond sample4. The Chap microcode remains the same.

```
 1      #include <pixar/pixar.h>
 2      #include <stdio.h>
 3
 4      main()
 5      {
 6              int x0, x1, y0, y1, t;
 7              double R, G, B, A;
 8              CHAP *chap;
 9
10              chap = ChapOpen("/dev/chap0",1);
11              ChapMMan(chap, 0);
12              if (ChapLoadGo(chap, "sample4.o", "sample4") < 0) {
13                      printf("Unable to load file.\n");
14                      exit(1);        }
15              for (;;) {
16                      ...
17              }
18              ChapClose(chap);
19      }
```

Figure 10-1: Sample5s Program Listing

There is a single host procedure that allows us to achieve the same effect as the series of charm commands described in the previous section.

`ChapLoadGo` [line 12]

The calling convention for this routine is:

ChapLoadGo(chap, file, entry)
CHAP *chap;
char *file, *entry;

where the C typedef `CHAP` is defined in the standard include file [line 1]. The *file* specified must be a relocatable object file created by *chas*(1) or *chld*(1). *ChapLoadGo* checks to see if the symbol *entry* is present in the Chap; if not, *file* is loaded and *ChapLoadGo* attempts to resolve any undefined references by searching a standard set of libraries, the same libraries that charm searches when loading files. Upon successfully completing this task, the Chap is set running at the instruction labeled by the *entry* symbol.

We check for a negative return code from *ChapLoadGo*, which indicates an error during loading. For further details, refer to the manual entry *ChapLoadGo*(3H).

Running *sample5s*

The rest of the program *sample5s* is the same as the previous host sample programs. As before, use the *make* utility to create an executable version. Run it and draw some rectangles to convince yourself that **`ChapLoadGo`** loads the code and sets it running.

11. Chap Routines

In Section 6, in the discussion and sample program involving chap-host cooperation, we introduced *libpixar*. In this section we fill in more details, to provide the programmer a firmer ability to write programs using it. There are four main sections in this more detailed discussion:

- Advanced Chas Programming
- Dynamic Loading
- Memory Management
- Interrupts

This section goes into more detail than most of this document. Much of this detail is unnecessary in the context of **Chad**, as discussed in the next section. That discussion is kept separate so that readers using those environments which do not support **Chad** can ignore it.

All the routines discussed below have descriptions in section 3H of the Pixar manual pages. The only routines in that section absent from this discussion are those used for diagnostic purposes.

General Remarks

Any host program that accesses a Chap must make a call to **ChapOpen**. The calling convention for this is:

```
CHAP* ChapOpen(chapname, shared)
char *chapname;
int shared;
```

Typically, the name of the chap is ''/dev/chap*n*,'' where *n* is the number of the Chap. If *shared* is non-zero, the Chap is opened in shared access; this means that other processes owned by the same user may open the same Chap. Otherwise, no other processes may use this Chap until the current process has terminated or has made a call to **ChapClose**. A zero return value means that the attempt to open the Chap has been unsuccessful.

Otherwise, **ChapOpen** returns a pointer to a CHAP data structure. This structure is defined in the include file *<pixar/chapdiag.h>*. However, the ordinary user will have no need to consult this file.

With this introduction concluded, we can go on to discuss a new way of transferring data from the host to the Chap.

11.1. Advanced Chas Programming

The sample programs beginning with *sample3s* used the 16 Sysbus registers to transfer the rectangle descriptions to the Chap. Recall that the host used **sysbus<13>** to signal the Chap that a rectangle had been passed. These 16 registers are physically resident, shared-access registers on the Pixar Image Computer. Another technique, for transferring data directly from a disk file into the Chap, is discussed in a subsequent section, under ‘‘Disk Buffer Routines.’’

11.1.1. Virtual Data Registers

There is an alternative, hardware-assisted technique for effecting larger and faster data tranfers through the use of **virtual data registers**. Figures 11-1 and 11-2 contain listings of simple host and Chap programs that use the virtual data registers to transfer an array of 256 entries from host to Chap. The Chap program then computes the squares of these numbers and returns them to the host. As before, bracketed numbers refer to the numbered lines in these listings.

```
 1    #include <pixar/pixar.h>
 2
 3    main()              {
 4          CHAP *chap;
 5          register int i;
 6          unsigned short *Cvdr, array[256];
 7
 8          chap = ChapOpen("/dev/chap0", 1);
 9          Cvdr = (unsigned short *) ChapVdregBase (chap);
10          ChapLoadGo(chap, "chvdr.o", "vdr_demo");
11          for  (i=0; i<256; ++i)
12                Cvdr[i] = i;
13          for  (i=0; i<256; ++i)
14                array[i] = *Cvdr;
15          for  (i=0; i<256; ++i)
16                printf("%d\n",array[i]);
17
18          ChapClose(chap);
19    }
```

Figure 11-1: Virtual Data Register Demo: Host Program Listing (*vdr.c*)

First, look at the host program. The macro **ChapVdregBase()** returns the address of the virtual data register array [line 9]. We use **ChapLoadGo**, introduced in Section 10.6, to load the Chap program and start it running [line 10]. Then we execute a loop 256 times, writing the loop counter *i* into the *i*th entry of the virtual data register array.

Now look at the corresponding Chap code in Figure 11-2. Note that the Chap program is somewhat more complicated than the host one. To understand why, we need to explain what happens when the host attempts to write to a virtual data register. The host waits until a special status bit in the Chap, the *sysrel* bit, is turned on, meaning that the sysbus is released. Then it pokes the specified index, or address (in this case *i*), into **sysbus<14>**, and the specified value (in this case also *i*) into **sysbus<15>**. To the host program, however, it appears as an ordinary array assignment statement. The size of

```
 1    #define  WAIT_NOT_BUSY                                $\
 2         push;                                            $\
 3              dowhile !sysbus busy;  1 ticks;             $\
 4              1 ticks
 5
 6    #define  WAIT_BUSY                                    $\
 7         push;                                            $\
 8              dowhile sysbus busy;  1 ticks;              $\
 9              1 ticks
10
11         .bss
12    array:    .space 256
13         .text
14    vdr_demo:
15         b0 = array
16         loop 256 do
17              WAIT_NOT_BUSY
18              acc = sysbus<14>
19              i0 = acc & 0xff
20              @b0+i0 = sysbus<15>
21              sysrel = 1
22         done
23
24         b0 = array
25         i1 = 4
26         loop 64 do
27              acc = (b0)
28              multx = acc
29              multy = acc
30              1 ticks
31              (b0) = lsp;       b0 = b0 + i1
32         done
33
34         b0 = array
35         i0 = 1
36         loop 256 do
37              WAIT_NOT_BUSY
38              acc = @b0; b0 = b0 + i0
39              sysbus<15> = acc
40              sysrel = 1
41              WAIT_BUSY
42         done
43         bpt
```

Figure 11-2: Virtual Data Register Demo: Chap Program Listing (*chvdr.s*)

the virtual data register array is 256. This fact explains why exactly 256 elements are transferred to the Chap in the sample program.

On its side, the Chap code waits until it detects that the host has accessed the Sysbus, via the macro **`WAIT_NOT_BUSY`**[lines 1-5]. Notice the '$\' used as at the end of each line. This construct is necessary when defining macros in order to satisfy both the C preprocessor, used by *chc*, and the assembler itself, since newlines are significant to the latter.

The **`dowhile`** construct used for this wait is similar to the looping mechanisms introduced earlier, and is described in the *Chas Reference Manual* on sequencer instructions. The **`sysbus`** condition code evaluates **true** when the host reads or writes from the Sysbus. When this condition code indicates that an access has occurred, the Chap escapes from this loop, and reads the address/value pair out of **sysbus<14>** and **sysbus<15>**, and makes the appropriate store. The statement **sysrel = 1** [line 21] releases the Sysbus. The host can not write a new value until the Sysbus has been released by the Chap.

Two subtleties of VDR usage not otherwise apparent here are worth emphasizing. First: the Chap program must waste no time after it obtains its data before asserting **sysrel = 1**. Otherwise, the bus between the Chap and the host may time out, causing a bus error in the host program. A few minutes with a debugger is usually sufficient to diagnose the problem, but it is still to avoid it in the beginning. The sneakiest aspect of this problem is that it is host-dependent, since the timeout interval varies from host to host.

The second hidden virtue of this code is seen on line 19: only the lower eight bits of the VDR address are used in assigning to **i0** from **acc**. Since there are only 256 VDRs, this merely seems redundant, but you should be careful to do this in your programs, since the upper 24 bits of **sysbus<14>** may contain arcane and irrelevant information.

The program above shows the advantage of using the virtual data registers as opposed to the ordinary Sysbus register access of previous sections: the synchronization of the two processes proceeds automatically via the *sysrel* bit. The host is synchronized by the hardware *sysrel* bit to wait until the transfer has been completed on the Chap before sending more data. The loop [lines 16-22] in Figure 11-2 corresponds to the loop [lines 11-12] in Figure 11-1.

Once the host-to-Chap transfer is complete, the Chap program computes the squares of the array elements [lines 24-32]. At that point, the data is transferred back from the Chap to the host via the virtual data registers ([lines 13-14] in the host and [lines 34-42] in the Chap). Notice that in this direction, the address field is not used by the host. This reflects the fact that when reading the virtual data registers, the host does not have access to the contents of **sysbus<14>**. Once again, the host reads are automatically synchronized with the Chap writes, since a host read cannot take place until the Sysbus has been explicitly released by the Chap program.

Suggestions for VDR Usage

In many cases where a linear array is being copied from host to Chap, the address field can be ignored, thus reducing the time spent by the Chap. For example, this is the case in the loop [lines 16-22], which could be rewritten by auto-incrementing the base register:

```
b0 = array
i0 = 1
loop 256 do
        WAIT_NOT_BUSY
        @b0 = sysbus<15>;  b0 = b0 + i0
        sysrel = 1
done
```

This strategy allows the programmer to transfer arbitrarily long arrays, since now the 8-bit limitation on the address passed in **sysbus<14>** no longer matters.

11.2. The Dynamic Loader

Figure 11-3 shows a listing of a sample program closely related to *sample5s*, in which we introduced the use of the dynamic loader from a host program. *sample5s* used the library routine **ChapLoadGo** to dynamically load the Chap code into the Chap and start it running. The program *sample6s* listed here achieves the same result using a series of more general purpose library routines.

```
 1      #include <pixar/pixar.h>
 2      #include <stdio.h>
 3
 4      #define        DEFARCHS "/usr/pixar/host/lib/libpt.a: \
 5         /usr/pixar/host/lib/libpx.a:/usr/pixar/host/lib/libpG.a"
 6      #define       UCODE        "sample4.o"
 7
 8      main()
 9      {
10             int x0, x1, y0, y1;
11             double R, G, B, A;
12             CHAP *chap;
13             LoadSym **sp;
14             u_short pc;
15
16             chap = ChapOpen("/dev/chap0",1);
17             ChapMMan(chap,0);
18             ChapBeginLoad(chap);
19             sp = ChapSymLookup(chap, UCODE);
20             if (*sp == 0) {
21                    ChapLoadLocs locs;
22
23                    ChapSetArchives(DEFARCHS);
24                    bzero(&locs, sizeof (locs));
25                    if (ChapLoad(chap, UCODE, &locs) != 0)
26                           exit(-1);
27                    pc = locs.cl_entry;
28             } else
29                    pc = (*sp)->ls_tbase;
30             ChapEndLoad(chap);
31             ChapRunAsync(chap, pc);
32             for (;;) {
33                    ...
34             }
35             ChapClose(chap);
36      }
```

Figure 11-3: Sample6s Program Listing (*sample6s.c*)

ChapMMan [line 17]

The function of this call is explained below, in Section 11.4, "Hardware Interrupts."

ChapBeginLoad and **ChapEndLoad** [lines 18, 30]

Any calls to the dynamic loading routines should be bracketed by these calls. They speed up the process by avoiding redundant file accesses and locks.

ChapSymLookup [line 19]

Before attempting to load code we first check the symbol table to see if the file has already been loaded. The dynamic loader automatically records the name of each file loaded into the Chap. Be aware, however, that in doing so it strips off any leading pathname. Thus, a file "/usr/pixar/lib/chad.ucode" would appear in the symbol table as "chad.ucode."

ChapSymLookup, the symbol table lookup routine, has the following calling convention:

```
ChapLoadSym **ChapSymLookup(chap, symname)
CHAP *chap;
char *symname;
```

The C typedefs **ChapLoadSym** and **CHAP** are defined in the standard include file, [line 1]. Note that a pointer to a pointer is returned; on [line 20] we check to see if the symbol is present by testing "***sp**."

ChapSetArchives [line 23]

Next we notify the dynamic loader that a set of libraries should be searched whenever we request code to be loaded. This set of archives is given as string of colon (:) separated pathnames. Our call to **ChapSetArchives** is equivalent to our use of the **:a** command in Charm. The companion routine **ChapAppendArchives** is equivalent to the Charm **:A** command.

ChapLoad [lines 25-26]

Finally, once we know the file is not presently loaded in the Chap and the libraries have been set up, we request that the dynamic loader load our file. **ChapLoad**'s calling convention is

```
int ChapLoad(chap, file, plocs)
CHAP *chap;
char *file;
ChapLoadLocs *plocs;
```

where, as before, the C typedefs are defined in [line 1].

The only parameter that deserves some attention is the "**&locs**" supplied for **plocs**[line 25]. Normally, the dynamic loader allocates space in the Chap for the code (**.text**) and data (**.data** and **.bss**) segments in the files it has to load. To have this information returned to you, a pointer to a **ChapLoadLocs** structure must be supplied. It is also possible to provide the loader with pre-allocated resources and/or specify that a file should be loaded at a specific location in the Chap. This information is passed to the loader by specifying non-zero values in the **ChapLoadLocs** parameter; in this case, the parameter becomes a value-result parameter.

ChapLoad returns either a **-1** in the case of an error, or a non-zero positive value, which is the number of undefined symbols encountered during the load. Specific errors may be deciphered from the global value **ChapErrno**; consult the manual pages for more information.

With the file loaded we can figure out the appropriate program counter we need by looking at the **cl_entry** entry in the **locs** structure [line 27]. This is the value of the entry point for our code after the dynamic loader has performed relocation. In the case where the file was already loaded, we can simply take the program counter from the base of the loaded **.text** segment [line 29].

ChapRunAsync [line 31]

With our code loaded into the Chap, the only thing left to do is start the processor executing our code at the proper instruction. To do this we use **ChapRunAsync** [line 31]. This routine resets the state of the Chap to a *default* state (stack pointer set to zero and runflags set to 0xf), forces the program counter to the specified value, and starts the machine running, returning immediately to its caller. The routine **ChapRun** does the same things but does not return until the Chap program has hit a breakpoint or the user types an interrupt on the keyboard. Related routines **ChapCont** and **ChapStep** are also available (see *Chaprun(3H)* in the manual pages).

The dynamic loader invokes a set of routines to perform memory management for it. These same routines are available to the host programmer who has the need to allocate resources independently of the dynamic loader. These routines are the subject of the next section.

11.3. Memory Management

Figure 11-4 is a diagram of the several modules involved in the allocation of Chap resources. It shows that there are resource allocation tables independent of the device driver and the Chap symbol table. These tables are managed by Unix utilities described in *ioctl*(2). The interested reader is referred to the Unix manual and *mman*(3C) for details of the implementation. For our purposes here, it is enough to describe the routines available for directly allocating Chap resources.

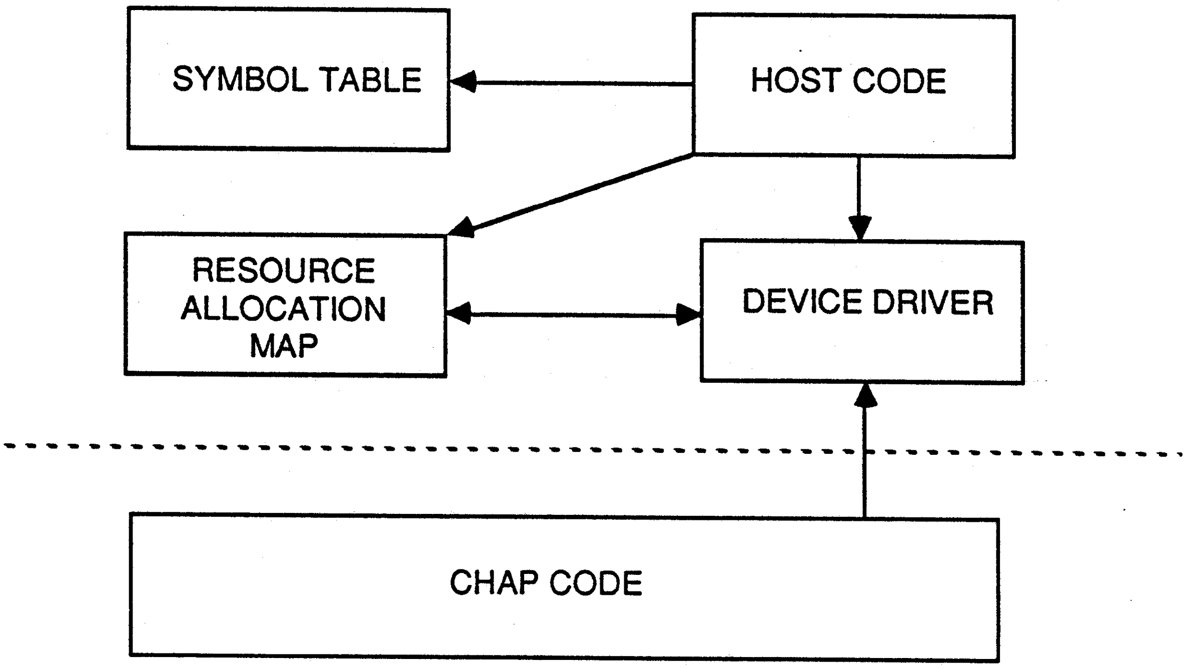


Figure 11-4: Memory Management Diagram

There are three types of Chap resources to be separately managed: instruction RAM, scratchpad memory, and frame-buffer memory. The units of allocation are instructions (96 bits), pixels (64 bits), and tile blocks (32 by 32 pixels), respectively.

`ChapGetConfig`

The amount of hardware area for each of these resources may be gotten from the shell level command *chconfig*(8). From inside a host program the same information may be gotten by a call to **`ChapGetConfig()`**. The calling convention is:

ChapGetConfig(chap, conf)
CHAP *chap;
struct *config* *conf;

The definition of the data structure involved may be found in the standard include file <*pixardev/chapioctl.h*>. This call would be useful in writing portable code that adjusted automatically to the resources available on a given Pixar Image Computer.

Allocating and Freeing Chap Resources

The resource allocation routines themselves are straightforward. For example, the code

```
u_short addr;
addr = ChapAllocSpad(chap, 100);
           ...
ChapFreeSpad(chap, addr, 100);
```

allocates a block of scratchpad memory of 100 pixels, then frees that block. Alternatively, the code

```
u_short addr;
addr = ChapGetSpad(chap, 1000, 100);
           ...
ChapFreeSpad(chap, addr, 100);
```

allocates a block of the same size, but forces the allocation to take place at pixel location 1000. Unsuccessful completion is signaled by a return value of -1.

The analogous routines for the framebuffer are **ChapAllocFB**, **ChapGetFB**, and **ChapFreeFB**. Those for instruction RAM are **ChapAllocRam**, **ChapGetRam**, and **ChapFreeRam**. Consult manual pages for *chapmman(3H)* for details.

Resetting

In the event of irreversible corruption to the symbol table or resource allocation maps, these may be flushed and reset by a call to **ChapReset**. Use ChapReset with care, as it destroys all information concerning the contents of the Chap instruction and scratchpad memory. The routine **ChapResetMap** described in *chapmman(3H)* is less appropriate, since it may leave the resource allocation tables inconsistent with the symbol table.

11.4. Hardware Interrupts

Recall from Section 6.1 that there are hardware interrupt facilities on the Chap to signal the host. We used the routine **ChapWaitForInterrupt** to implement a host-to-Chap signalling process. We can now explain why we had to include the call

```
ChapMMan(chap, 0)
```

for the interrupts to work.

There is a set of allocation routines available for Chap programs similar to those for host programs described in the previous section. See the manual page *mman*(3C) for their description. These Chap routines use hardware interrupts to communicate their requests to the central allocation tables on the host. If the second argument to **ChapMMan** is non-zero, interrupts are intercepted by the device driver and interpreted as memory management requests. Hence, by setting this argument to zero, we ensure that the interrupts sent by the Chap program will be passed along to the host process without interference from the memory manager.

12. Using Chad

This section gives you the information you need to smoothly integrate your Chas programs into host programs that use **Chad**. Of course, we assume the familiarity with **Chad** provided by *Programming with Chad.*

12.1. General Remarks

Once **ChadBegin()** has been called, there is no need to use **ChapOpen()** to open the Chap devices that will be used, since the **Chad** environment includes a `CHAP *` for each Chap. You can obtain this pointer via the macro **ChadChap()**, defined in */usr/pixar/include/chad.h*, giving it a `ChapID`:

CHAP ***ChadChap(** chapid)
ChapID chapid;

The monitor that **Chad** runs on the Chap immediately does `jsr initstack`, so that all of the library routines which use `pushr, popr, pushb, popb, pushi` and `popi` will work properly. You needn't include this `jsr initstack` statement in any Chap program you write, although doing so will cause no harm, since **Chad**'s monitor itself does not use the register stacks.

12.2. The Cost of Using *Chad*

The following registers are used for special purposes by **Chad**, and therefore cannot be used in host programs operating at the **Chad** level:

```
sysbus<12>
sysbus<13>
sysbus<14>
sysbus<15>
acc
Virtual Data Registers 170 through 255
```

`sysbus<14>` and `sysbus<15>` are used for the virtual data registers and are therefore offlimits anyway. `sysbus<12>` and `sysbus<13>` are special to **Chad**; the former is used for addressing and the latter for the host-Chap synchronization flag. `acc` is **Chad**'s operating register. It is in constant use by the monitor, and therefore cannot be set usefully or read reliably. Finally, the **VDR**s are mapped to various destinations in the Chap for I/O operations.

These restrictions *do not*, however, restrict host programs which send data directly to Chap programs while they are running (i.e., without returning to the monitor level). These transactions are wholly independent of **Chad**, and therefore unrestricted. This is the subject of the next section.

12.3. Data Transfer to Running Chap Programs

Chad is heavily dependent on the routines in *libpixar*. In fact, it is fair to say that the former is a simplification of the latter's functionality. In one way, however, they differ radically: where *libpixar* uses the Chap's diagnostic registers to transfer data between host and Chap, **Chad** uses its monitor running on the Chap to move data, in cooperation with its hostside subroutines, via the `sysbus` registers and VDRs. The reason for this difference is speed. Whatever the justification, however, it is important to realize that *the* **Chad** *monitor must be the program currently running in the Chap in order to use* **Chad** *to move data onto the Chap.*

Under normal circumstances, in which execution of a Chap program is a matter of loading a set of parameters into registers or scratchpad locations and branching to the appropriate routine, this difference means nothing; the monitor is used to do the transfer, and it is executing whenever other routines are not (before they are called and after they return). However, if the *routine itself* needs to have data passed to it while it is running, then **Chad** cannot be used: its monitor is not running. If **ChadWrite**() was used to pass data, it would wind up waiting for the Chap to return to the monitor -- for the executing Chap routine to complete, in other words. If the routine is just idling, awaiting data, deadlock ensues.

Therefore, whenever routines running in cooperation between the host and the Chap must pass data back and forth, it is up to the programmer to establish a protocol for 1) which registers are used for what data, and 2) how the data movement is to be synchronized.

It is beyond the scope of this document to discuss software synchronization, except to remind the reader of the sample programs we have already seen. However, a few observations are in order and will make life easier.

Almost all the Chap routines in the Pixar Software Release get along by passing parameters, either in registers or in scratchpad buffers. Before writing data-passing code, you should convince yourself that it really can't be done using repeated subroutine calls.

If your program absolutely must have intermittent injections of data, it is usually possible to write your code so that it returns to the **Chad** monitor (where it receives the data it needs), then picks up where it left off upon the next invocation. In this case you must be careful to save the state of the routine (register values, that is; scratchpad locations are generally private) and restore it upon return, *unless you are sure that no other routines will be called by the monitor before returning to your routine.* Since **Chad** promises not to affect any registers but the last four **sysbus** registers and **acc**, **Chad** is no threat, but multiprocess control over the Chap can send execution off to routines which are dedicated to trashing your registers.

While it is possible to use the routines in *libpixar* to access any of the registers or scratchpad locations in the Chap, doing so is not recommended for one reason: speed. It is **much** faster to move data using the **sysbus** registers and VDRs. The earlier example programs illustrate this process.

As in Sample 6, you have the option of using the Chap's hardware interrupt as a synchronization mechanism. **Chad** does not interfere with this mechanism, or use it in any way.

Be aware of the value of the **sysrel** bit as a synchronization mechanism: After VDR access, the host stops automatically, waiting for **sysrel** to be asserted. Thus the host is prevented from proceeding until the Chap program deems it appropriate. Of course, the danger here is a bus timeout if the Chap takes too long doing so. Consequently, the Chap program has a responsibility not to tarry too long.

12.4. *Chad* and the Chap Libraries

The tutorial sample programs appearing in earlier sections required their Chap programs to be loaded explicitly by the user. One of the most important services of **Chad** is to simplify this process and to automate it in host programs.

Recall from *Programming with Chad* that **Chad** will load any Chap routine under host control with a call to **ChadAlloc()**, giving a character string identifying the routine. There are three prerequisites for this:

- The routine (the location of the code) must have been declared with the Chas `.globl` directive, making it known outside the file in which it is defined.
- After compilation, the routine must be included in a library.
- The library must be among the set of libraries **Chad** searches to find routines.

You should already be prepared for the first step by the example programs. Chap code can be included in an archive file by the following steps:

- Compile the '.s' module using *chc* with the '-c' flag. Thus, to compile the module *foo.s*, type

  ```
  chc -c foo.s
  ```

  to the UNIX Shell. Of course, you are responsible for using the '-I' option to use the appropriate header directories.
- Use the standard Unix command 'ar' to create/modify the library file.
- Use the Pixar program *chranlib* to prepare the archive for use. This is analogous to the Unix shell command *ranlib*.

Figure 12-1 is a makefile for creating an archive file *foo.a* from a Chap program file *foo.s*. This is a useful general template for compiling Chap microcode libraries, which you are encouraged to modify and use for your own programs.

```
#
#	Makefile template illustrating Pixar Chapcode archives
#
PIXAR= /usr/pixar
CHC=	${PIXAR}/host/bin/chc
CHRANLIB=${PIXAR}/host/bin/chranlib
INCLUDE=${PIXAR}/include
IPATH= -I. -I${INCLUDE} -I${PIXAR}/include/pixar
CHCFLAGS=${IPATH}

SRCS=	foo.s
OBJS=	foo.o

LIB=	foo.a

.s.o:
	${CHC} ${CHCFLAGS} -c $*.s

all:	${LIB}

$(LIB):${OBJS}
	ar uc $(LIB) ${OBJS}
	${CHRANLIB} ${LIB}
```

Figure 12-1: Making an Archive File (*foo.make*)

Once the library file exists, **Chad** must be told to look in it when asked for routines. This is done using **ChadLibs()**, which is documented in *Programming with Chad.*

Figure 12-2 illustrates these steps. The program is equivalent to *sample6*, but uses **Chad** instead of the *libpixar* routines to load the Chapside routine and set it running. It is instructive to compare this program with *sample6s.c* for simplicity and intuitiveness.

```
#include <pixar/pixar.h>
#include <chad.h>
#include <stdio.h>

#define UCODE "foo.a"
#define ROUTINE "sample4"
main()
{
        ...
        ChadPC *routine;

        ASSERT( ChadBegin( CHAP0, 0 ));
        chap = Chaps[CHAP0];
        ChadLibs( UCODE, NIX );
        ASSERT ( ChadAlloc( CHAP0,
                        RAM, &routine, ROUTINE,
                        NIX));
        ASSERT ( ChadGo( routine ) );
        for (;;) {
                ...
        }
        ChadEnd( CHAP0 );
}
```

Figure 12-2: Replacing *sample6* with **Chad** (*sample7s.c*)

Exercise: Modify *foo.make* from Figure 12-1 (don't forget to rename it *Makefile* or *makefile*) to create an archive from the sample programs in earlier sections of this tutorial. Modify the sample programs themselves, so that rather than allocating their own pixel windows, a pixel window is passed as a parameter in a scratchpad base register. Then write a **Chad** program (use *sample7s.c* as a departure point) to allocate a pixel window with **ChadAlloc()**, put its address (**pw->addr**) in the appropriate base register with **ChadWrite()**, and *then* call the Chap routine to draw the rectangles. You will find the source files in the Pixar Software Release (*/usr/pixar/host/src/lib/lib** for host routines, */usr/pixar/chap/src/lib/lib** and */usr/pixar/chap/src/bin* for Chap code) helpful if you get stuck.

12.5. Discussion: the *Chad* Development Process.

The Pixar Image Computer in general and the Chap in particular, like many other fast parallel-processing devices, is best at inner loops: those repetitive tasks, like the pixel-level operations that are performed on every pixel in an image, which account for the preponderance of the work done by a computer. The outer loops, which do the semi-intelligent work of setting up the environment and determining what the inner loops will do, are usually best left to a host. They account for most of the *development* time of a major application, so this development cycle is best performed on a host computer with a good environment. The line between outer and inner loops is indistinct; determining the appropriate level for the host-Chap interface for a given application is the process **Chad** is designed to support.

Chad gives you easy, reasonably efficient access to the important parts of the Chap, making it possible for host programs to "emulate" Chap programs which consist mostly of setting up parameters and calling other Chap programs. This means two things. First, it gives you immediate access to the tiniest (globally defined) Chap routines in the Pixar Software Release -- and there are a **lot** of those. More important, it allows you to develop programs and debug algorithms *on the host*, introducing speed improvements by moving work down to the Chap in tiny pieces, beginning with the innermost loops (which hopefully already exist). As performance warrants, you can move more and more of your program incrementally from the inner loops outward, until the speed of the application is satisfactory. This layered approach also tends to produce routines which are useful in other contexts, and can profitably be added to a site's libraries.

Chad, then, attempts to provide both an interface and a convention encouraging the development of re-usable code. This code will be the source of Pixar Contributed Software. We look forward to hearing of your efforts.

13. Video Routines

In addition to a Chap, each Pixar Image Computer has a programmable video board to control the display of the framebuffer memory. The reader who would like to become familiar with this board is encouraged to experiment with the shell-level command, *tool*(1), an interactive program that exercises most of its features (see manual entry or type the shell command `tool -`). This discussion focuses on the set of routines available to the user who wants to control the video board from a host program. We attempt to illustrate as many as possible through short example programs. There are three main topics:

- Display parameters
- Colormap
- Hardware cursor

The discussion is simplified in comparison to that for the Chap routines since there are no companion programs running on the video board requiring synchronization. In fact, Chap programs cannot directly access the video board.

General Remarks

As with the Chap, any user program must first open the video board for use via a call to `VideoOpen`. The calling convention is:

VIDEO *
VideoOpen (videoname, width, height, shared)
char *videoname;
int width, height, shared;

videoname is typically "/dev/video*n*" where *n* is the number of the video board, usually 0 if it is the only one. *width* and *height* are dimensions in pixels of the window into the framebuffer. If you do not use the hardware cursor (see below), then these parameters are not used. As before, if *shared* is non-zero, then the device is open with shared access.

13.1. Display Parameters

Figure 13-1 shows the listing of a sample program that opens the video board and manipulates the display parameters before closing the video board and exiting. The only file that needs to be included, `<pixar/pixar.h>`, is the same as for the Chap routines.

`VideoSetDisplay`

After opening the video board, we make a series of calls to `VideoSetDisplay`. This routine controls the location and size of the displayed data. Its calling convention is

VideoSetDisplay(video, base, width, height, x, y, mode)
VIDEO *video;
int base, width, height, x, y, mode;

base, *width*, and *height* are in tile blocks, and determine the format and size of the

```
#include <pixar/pixar.h>

main()          {

    int i, j;
    VIDEO *v;

    v = VideoOpen("/dev/video0", 1024, 768, 1);
    if (v == (VIDEO *) 0) {
        printf("video: can't access video controller. \n");
        exit(-2);
        }
    VideoSetDisplay(v, 0, 32, 24, 0, 0, VMODE_RED);
    sleep(1);
    VideoSetDisplay(v, 0, 32, 24, 0, 0, VMODE_GREEN);
    sleep(1);
    VideoSetDisplay(v, 0, 32, 24, 0, 0, VMODE_BLUE);
    sleep(1);
    VideoSetDisplay(v, 0, 32, 24, 0, 0, VMODE_RGB);
    sleep(1);

    for (i=0;i<16; ++i)      {
        VideoSetDisplay(v, 0, 32, 24, 64*i, 48*i, VMODE_RGB);
        sleep(1);
        }
    VideoSetDisplay(v, 0, 32, 24, 0, 0, VMODE_RGB);
    for (i=ZOOM_MIN ; i<= ZOOM_MAX ; ++i)      {
        VideoZoom(v, i);
        sleep(1);
        }
    VideoZoom(v, 1);
    VideoClose(v);
}
```

Figure 13-1: Video Display Parameters Demo (*videmo.c*)

framebuffer memory. Recall that the framebuffer memory is essentially a linear array of tile blocks and must be organized into a two dimensional rectangle by the user. *base* sets the beginning tile block, *width* determines how many tile blocks span one row of the rectangular area, and *height* gives an upper bound on how many such rows are to be displayed.

The next two parameters, *x* and *y*, are in pixels, and specify an offset into the rectangular array of tile blocks defined above. This is the point where the video controller begins to scan out the framebuffer memory; it will be at the upper left corner of the visible display. In the coordinate system of the display, *y* increases in downward direction. Due to hardware limitations, *x* and *y* are truncated to be multiples of 4.

Channel Crossbar

Finally, *mode* determines how the four channels of data in the framebuffer are to be interpreted in the display. A **channel crossbar** switch routes the data from the framebuffer into the colormaps. In the usual, or ''full-color'' mode, the red, green, and blue channels are routed through the red, green, and blue colormaps, respectively. Each channel can be used as the source for pseudo-color, too. For example, in red pseudo-color, the data in the red channel of the pixel is fed separately to all three colormaps, putting a "black and white" image on the screen. The green, blue, and alpha channels are ignored. There are also pseudo-color modes using the green, blue and alpha channels. To use this mode, the user usually will want to load a customized colormap (see the next sample program).

The sample program illustrating these features is shown in Figure 13-1. As with other sample programs, you can use the UNIX utility *make* to create an executable version of this program. Simply type the shell command **`make videmo`**. For best results, there should be a full-color image in the display. Make sure the video board is set to its defaults by the command '**`video -init`**'. Then run the program with the command **`videmo`**. Notice the changes to the display first.

Commentary on Figure 13-1

The calls to **`VideoSetDisplay`** [lines 13,15,17,19] illustrate the various settings of the channel crossbar discussed above. As these calls are executed, the monitor should display a grey scale image of the red, green, and blue channels, in order, before returning to full color mode.

The loop [lines 22-25] moves the upper left corner of the display, controlled by parameters *x* and *y* in the calling description above. Then the corner is set back to the default (0,0).

`VideoZoom`

The next loop [lines 27-30] plays with the hardware zoom factor. It increases this from its minimum (1) to its maximum (16). These values are included as macros, **`ZOOM_MIN`** and **`ZOOM_MAX`**, in the include file. The hardware zoom is implemented by pixel replication on the video scan-out. For example, a zoom factor of four causes each pixel of framebuffer memory to cover a block of four by four pixels on the display. Hence, 1/16 as many pixels are displayed. If the upper left corner of the display is black, then at some point as the zoom increases, the display will go black.

Finally, the zoom is set back to its default state [line 31], and the video is closed with a call to **`VideoClose`** [line 32].

13.2. Colormap Demo

All the data coming out of the video output port goes through the colormap. There are three colormaps: one for each of the output channels. The channel crossbar, described above, determines the input channels. Figure 13-2 presents a simple demonstration of the colormap routines available to the programmer. The program, called *contour*, accepts a single command line integer argument, *n*. It then creates *n* linear ramps within the colormap array and sends this colormap to the video board. The program gets its name from the fact that at the boundary of the individual ramps the displayed color changes from black to white; hence, sharp contour lines appear. Use the command **`make contour`** to create an executable version.

Computing the Colormap [lines 21-36]

Recall from Chapter 4 that the framebuffer stores pixel values in the range [–.5, 1.5), by interpreting the top two bits of the intensity value according to Table 13-1.

Upper 2 Bits	Range
00	[0, .5)
01	[.5, 1.0)
10	[1.0, 1.5)
11	[–.5, 0)

Table 13-1: Pixel Value Conversion

Each value coming from the framebuffer is adjusted (by hardware) to be in the 10-bit "address space" of the colormap. Thus, values from a 12-bit frame buffer are automatically shifted right by two bits; values from an eight-bit frame buffer are shifted left by two bits. This value is used as an index into the colormap. For example, in a 12-bit frame buffer, the pixel values 16, 17, 18, and 19 all map to colormap entry 4. The values in the colormap are scaled integers with 12 bits of fraction. There are no integer bits. In this system, then, maximum intensity corresponds to the value 4095.

The conversion of Table 13-1 is implemented in hardware in the Chap when transfers take place between scratchpad and framebuffer. This does not happen in the video board. Instead, the colormaps determine this conversion. To emulate Table 13-1 in a linear ramp, use the colormap described in Table 13-2.

```
 1   #include <pixar/pixar.h>
 2   #define        ONE        0xfff
 3
 4   main(argc, argv)       char **argv;            {
 5
 6        VIDEO *v;
 7        int num = 8;
 8        short cm[3][1024];
 9
10        if (argc > 2)             {
11            printf("Usage: contour number_of_ramps.\n");
12            exit(-2);
13        }
14        if (argc == 2) num = atoi(*++argv); /* default: 8 */
15
16        v = VideoOpen("/dev/video0", 1024, 768, 1);
17        if (v == (VIDEO *) 0) {
18            printf("video: can't access video controller. \n");
19            exit(-2);
20        }
21        { register int i, j, k, m, number;
22        double dc, c;
23        number = 512.0/num;
24
25        dc = num/512.0;
26        for (i=0,k=0;k<num;++k)
27            for (c = 0, m=0; m<number; c += dc, ++m, ++i)      {
28                  if (i > 511) break;
29                  for (j=0;j<3; ++j)
30                    cm[j][i] = c * ONE;
31            }
32        for (;i<768;++i)
33              for (j=0; j< 3; ++j) cm[j][i] = ONE;
34        for (;i<1024;++i)
35              for (j=0; j< 3; ++j) cm[j][i] = 0;
36        }
37
38        VideoSetColormap(v, cm[0], cm[1], cm[2]);
39
40        VideoClose(v);
41   }
```

Figure 13-2: Colormap Sample Program (*contour.c*)

Top two bits	Colormap indices	Values	Interpretation
11	768-1023	0	(Clamp to 0)
0*x*	0-511	0-4095	(Linear ramp)
10	512-767	4095	(Clamp to 1.0)

Table 13-2: Simple Linear Colormap

Sample Program *contour* (Figure 13-2)

The colormaps contructed in the sample program [lines 21-36] respect this convention. The sequence of ramps is placed into the bottom half of the array, while the top half is used for clamping, as in Table 13-2. The ramps are calculated in [lines 25-31]; the clamp areas are filled in [lines 32-35]. Once the array has been computed, it is sent to the Pixar Image Computer via a call to `VideoSetColormap` [line 38]. Its calling convention is:

VideoSetColormap (v, red, green, blue)
VIDEO *v;
short red[1024], green[1024], blue[1024];

The current colormap may be retrieved via a call to `VideoGetColormap`, which has the identical calling convention.

One common use of colormaps is to implement **gamma** correction for the aberrations in the response of the display elements. This is available with the shell level utility, *video*(1); consult the manual page, or type '`video -`'.

13.3. Hardware Cursor

One final focus of this discussion is the hardware cursor. This is a programmable bit array of 128 by 128 pixels, overlaid on the video signal without affecting the contents of the framebuffer. Bits not on are not affected; bits turned on in the cursor array appear as "superwhite," so they are distinguishable from any framebuffer color. A cursor which was all 0's would be totally transparent and hence invisible. Up to four cursors may be loaded at any one time, so that the user may switch from one to another.

Figure 13-3 contains a listing of a sample program that demonstrates the use of the hardware cursor. Execute the command `make cursor` to create an executable version. Run this program before and after reading over the program listing to acquaint yourself with the working of the hardware cursor.

The main data structure required is `CURSOR`. There is a set of externally defined cursors for use by programs. We have included the crosshair and clef cursors in this demo [line 4]. See the Pixar manual pages (*videocursor(3H)*) for other available cursors. For the curious reader, the `CURSOR` data structure is defined in the file *<pixar/video.h>*.

Once a cursor data structure has been defined, as in the case of *xhair_cursor*, the next step is to load the cursor:

VideoLoadCursor(v, n, c)
VIDEO *v;
CURSOR *c;

The parameter *n* is an integer from 0 to 3 and identifies in which of four possible slots to load the cursor. We load the two externally defined cursors into the first two slots [lines 20-21].

Manipulating the Active Cursor

The loop [lines 23-31] illustrates two more routines. First, we can move the cursor to an arbitrary screen location with `VideoSetCursor`.

```
#include <pixar/pixar.h>

      VIDEO *v;
      extern CURSOR xhair_cursor, clef_cursor;
      static int loc[10][2] =
           {100, 100, 100, 300, 100, 500, 100, 700,
            300, 700, 500, 700, 700, 700, 500, 500,
            300, 300, 100, 100 };

main()           {

      int i;

      v = VideoOpen("/dev/video0", 1024, 768, 1);
      if (v == (VIDEO *) 0) {
           printf("video: can't access video controller.\n");
           exit(-2);
           }

      VideoLoadCursor(v, 0, &xhair_cursor);
      VideoLoadCursor(v, 1, &clef_cursor);

      for (i=0; i< 10; ++i)      {
           VideoSetCursor(v, loc[i][0], loc[i][1]);

           VideoCursorOn(v, 0);
           sleep(1);

           VideoCursorOn(v, 1);
           sleep(1);
           }

      VideoCursorOff(v);

      VideoClose(v);

}
```

Figure 13-3: Hardware Cursor Sample Program (*cursor.c*)

VideoSetCursor(v, x, y)
VIDEO *v;
int x, y;

This will position the cursor at the pixel location (*x,y*) within the display window, as described in the first sample program.

Only one cursor can be *active* — that is, displayed — at a time. To change the active cursor, use a call to `VideoCursorOn`:

VideoCursorOn(v, n)
VIDEO *v;
int n;

This will make cursor *n* the active cursor, and will display it according to the most recent call to `VideoSetCursor`. **Warning:** *Calling* `VideoSetCursor` *before*

VideoLoadCursor *will not achieve the desired result.* At least one cursor must be loaded in to set the position of the cursor.

The loop moves the cursor through a predefined sequence of locations on the screen, alternating between the two loaded cursors. After this journey, the demo is completed and we terminate with a call to **VideoClose**.

Conclusion

This concludes our discussion of video routines available to the host programmer. Further details are available in the Pixar manual pages *videodisplay*, *videocmap*, *videocursor*, and *videopen* in section 3H. The reader is also referred to the section 1 entries for *tool*, *loop*, and *video* for shell-level commands that utilize the video board.

14. Miscellaneous Routines

Most host routines for the Pixar Image Computer have been covered in the previous two sections. However, there are a few miscellaneous ones that concern neither the Chap nor the video board. For most applications, these routines are not called explicitly. These routines fall into three categories:

- Dumi routines
- Memory controller routines
- Disk buffer routines

Dumi is an acronym for "Dumb Interface." It is the interface card between the host and the Pixar Image Computer. All communication of the host and either the Chap or the video board goes through this card. Each Dumi can control up to eight Chaps, four video boards, a memory controller (see below), and a disk buffer (see below). For a more complete description of the hardware configuration, consult the manual entry Dumi (4). There are two routines available, for opening and closing the Dumi:

DUMI ***DumiOpen**(device)
char *device;

DumiClose(dumi)
DUMI *dumi;

Under normal circumstances, the user does not need to know that the Dumi exists; however, in certain circumstances he or she may want to examine the diagnostic registers contained there. For an example, consult the shell-level command *dumi*, documented in Section 1 of the Pixar manual pages, with source in */usr/pixar/host/src/bin/dumi.c.*

14.1. The memory controller routines

Each Dumi has an associated minor device, known as a memory controller. As with the Dumi itself, under normal circumstances, the user does not need to directly access this device. However, for the user who wishes to examine the device's diagnostic registers, there are routines:

MCTRL ***MctrlOpen**(device)
char *device;

MctrlClose(Mctrl)
MCTRL *Mctrl;

Interested users are referred to the maintenance command *mctrl* (see section 8 of the Pixar manual pages), and the associated source code in */usr/pixar/host/src/bin/mctrl.c.*

14.2. Disk buffer routines

The remaining routines support access to a Dumi's disk buffer, which is a special hardware device for speeding up disk transfers to and from the Dumi. It can be thought of as a more powerful version of the virtual data registers (see Section 12.1.1 above). We have included a sample program illustrating its use. Figures 14-1 and 14-2 show the source listings of a sample host and Chap program pair that use this feature to transfer the contents of a disk file on the host into the scratchpad memory of the Chap.

```
 1      #include <pixar/pixar.h>
 2      #include <stdio.h>
 3      #include <sys/file.h>
 4      #define        DWSIZE            32*1024
 5
 6      main()              {
 7
 8            DUMI *dp;
 9            DB *db;
10            CHAP *chap;
11            int fd;
12
13            chap = ChapOpen("/dev/chap0", 1);
14            ChapLoadGo(chap, "chdbdemo.o", "dbdemo");
15
16            if((dp = DumiOpen("/dev/dumi0")) == NULL) {
17                  printf("Unable to open dumi.\n");
18                  exit(-2);
19                  }
20
21            if((db = DbOpen("/dev/diskw0", DWSIZE)) == NULL) {
22                  printf("Unable to open diskwindow.\n");
23                  exit(-2);
24                  }
25
26            if ((fd = open("primes", O_RDONLY) < 0)       {
27                  printf("Unable to open data file\n");
28                  exit(-2);
29                  }
30
31            dp->dumi->dr_addr1 = 1;      /* set dumi register */
32            read(fd, db->db_bp, 128 * sizeof(short));
33
34            DumiClose(dp);
35            DbClose(db);
36            ChapClose(chap);
37      }
```

Figure 14-1: Disk Buffer Usage, Host Side (*dbdemo.c*)

Notice that, as in previous sample programs, we open the Chap and use `ChapLoadGo` to load and run the companion Chap program (shown in Figure 14-2).

Setting Up the Dumi [lines 16-19]

To use the disk buffer, the host program must first open the Dumi. This is so it can later set one of the diagnostic registers with a "magic" value ([line 31]) which is required in order that the disk buffer function correctly.

```
#define  WAIT_NOT_BUSY                          $\
      push;                                     $\
            dowhile !sysbus busy;  1 ticks;     $\
            1 ticks

      .bss
primes:      .space 128
      .text
dbdemo:            .globl dbdemo

      b0 = primes
      i0 = 1
      loop 128 do
            WAIT_NOT_BUSY
            @b0 = sysbus<15>;      b0 = b0 + i0
            sysrel = 1;
      done
      bpt
```

Figure 14-2: Disk Buffer Usage, Chap Side (*chdbdemo.s*)

Opening the Disk Buffer [lines 21-24]

The host then opens the disk buffer with a call to **DbOpen**. Its calling convention is similar to previous open routines:

DB ***DbOpen**(dbname, size)
char *dbname;
int size;

The generic name for a disk buffer device is "/dev/diskw*n*," where *n* is a number identifying the disk buffer. *size* is an upper limit on the number of bytes to be transferred at one time. The value in this sample program, 32 kilobytes, is the maximum allowed. The buffer area is memory mapped. The starting address of the area is stored as a **char *** in the field **db_bp** of the DB structure. It is this address which we use in the **read** statement [line 32].

The sample program reads a small disk file named *primes* [lines 26-29]. *primes* contains the first 128 primes in packed binary format, with two bytes per prime. To examine this file, use the UNIX utility **od** to get an octal dump of the contents of the file.

NOTE: If this file is not present, use the shell command **make mkprimes**, then execute it by typing **mkprimes**. This will create a new copy of the file *primes*.

After setting the magic Dumi register [line 31], the sample program reads the contents of *primes* into the disk buffer [line 32]. **This is all that the host program has to do to transfer the data.**

Now look at the Chap program in Figure 14-2. The code resembles that of *vdr.c* (Figure 11-1). This is because both the virtual data registers and the disk buffer use the same mechanism to transfer single words of information: the value to transfer is poked into **sysbus<15>**, the Chap detects the poke via the **sysbus** condition code, the Chap stores away the value contained in **sysbus<15>**, and finally releases the Sysbus for another transfer. The main difference is that in the case of the disk buffer, the host program does not need to make individual transfers.

Each two consecutive bytes of the disk file is transferred as one 16-bit Chap word. If the amount of data to be transferred is larger than 32 kilobytes, then the host program will have to make multiple reads to the disk buffer, each time reading at most that much information.

Cleaning Up

After the transfer is completed, the various devices that have been opened have to be closed by calls to the appropriate routines [lines 34-36].

To verify that this mechanism works, create an executable version of the sample program by the command **`make dbdemo`**. After running this sample program, use Charm to examine the Chap scratchpad to verify that the prime numbers contained in the file *primes* have been transferred there.

15. Conclusions

This concludes the discussion of host routines for interactive programming of the Pixar Image Computer. The discussion has been intended as a companion document to the Pixar manual pages for these routines, contained in the 3H section. This discussion has been successful if it provides a foundation for creating customized programs to fit your needs. It has been designed so that users whose requirements go beyond the information provided here will now be able to find the information in the manual pages, by following up the references to more sophisticated source files, or by consulting the include files which define the data structures such as **`CHAP`** and **`VIDEO`**. The interested user can find the source files for the routines under discussion in subdirectories of */usr/pixar/host/src/lib*, principally *libpixar* and *libchad*. Most of the include files are contained in */usr/pixar/include*, or its subdirectory *pixar*.

Chap Assembler Reference Manual
CHAS
Copyright 1986, Pixar

1. Introduction

This document describes the usage and syntax of the Pixar Channel Processor Assembler *chas*. Readers of this document should be familiar with the Chap architecture, no description is presented here.

2. Usage

Chas is invoked as follows

chas [−**wsS**] [−**o** *output*] [$file_1, file_2, ..., file_n$]

The −**w** flag suppresses the generation of warning messages.

The −**s** flag causes messages to be printed regarding multiple instructions assemblies which result from the ''special bit'' (see §8.10 and §9).

The −**S** flag causes *chas* to print the contents of the symbol table on the standard output after all input files have been assembled.

The −**o** flag causes the output to be placed in the file *output.* By default, the output of the assembler is placed in the file *a.out* in the current directory.

The input to the assembler is taken sequentially from $file_1$, $file_2$, ..., $file_n$. Files are not assembled separately, all input files are concatenated. If no files are supplied as arguments on the command line, *chas* reads from the standard input.

3. Lexical Conventions

3.1. Scalar Constants

All scalar (integer) constants are treated as 32 bit quantities. Constants are specified as in C.

The set of *digits* consists of ''0123456789abcdefABCDEF'' with the obvious values. An octal constant consists of a sequence of digits with a leading zero. A decimal constant consists of a sequence of digits without a leading zero. A hexadecimal constant consists of the characters ''0x'' (or ''0X'') followed by a sequence of digits.

3.2. Fixed Point Constants

Fixed point constants consist of an integer part, a decimal point, a fractional part, and an optional one character ''fractional precision specification'': **e** or **E** for eleven bit alpha and component values, **f** or **F** for fourteen bit coefficient values. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fractional part (not both) may be missing. By default fixed point constants are considered to have eleven bits of fraction (i.e. they are treated as *alpha/component* values).

3.3. Operators

There are several single and multi-character operators; see §6.1.

3.4. Blanks

Blank and tab characters may be interspersed freely between tokens, but may not be used within tokens. A blank or tab is required to separate adjacent identifiers or constants not otherwise separated.

3.5. Comments

"C style" comments, introduced with a "/*" prologue and ended with a "*/" epilogue, are supported.

4. Segments and Location Counters

Assembled code is placed in the *text* segment, assembled data is placed in either the *data* segment or the *bss* segment. The *bss* segment contains uninitialized memory locations (zero filled by the run-time loader when the module is loaded); the *data* segment contains initialized memory locations*. Only instructions may be assembled into the *text* segment; the assembler makes no assumptions about data placed in the *data* and *bss* segments. Associated with each segment is a location counter which begins at zero. For each 16-bit word assembled into the *data* or *bss* segments, the *data* or *bss* location counter is incremented by one. For each 96-bit instruction assembled into the *text* segment the *text* location counter is incremented by one. There is no way to reference a specific location counter. The current segment's location counter may be referenced by the special symbol ".". The *text* segment of a program is mapped to the Chap instruction RAM; the *data* and *bss* segments of a program are mapped to the Chap scratchpad RAM.

The Chap link editor, *chld*, and the Chap dynamic loader, *chload*, align each input files' *data* segment on a 16 word boundary and each *bss* segment on a 4 word boundary. The *data* segment alignment allows *chas* to tessellate initialized data.

5. Statements

A source program is composed of a sequence of *statements*. Statements are separated by semicolons. There are two kinds of statements: null statements and keyword statements. Either kind of statement may be preceded by one or more labels. All statements on a single line assemble into one Chap microinstruction. To force statements on multiple lines to be assembled into a single microinstruction, the statements must be surrounded by braces, "{}". When multiple statements are grouped with braces, the compound statement need not be followed by a semicolon.

5.1. Named Global Labels

A global label consists of a name followed by a colon. The effect of a name label is to assign the current value and type of the location counter to the name. An error is indicated in pass 1 of the assembler if the name is already defined.

A global label is referenced by its name.

5.2. Numeric Local Labels

A numeric label consists of a digit 0 to 9 followed by a colon. Such a label serves to define temporary symbols of the form "*n*b" and "*n*f", where *n* is the digit of the label. As in the case of name labels, a numeric label assigns the current value and type of the location counter to the temporary symbol. However, several numeric labels with the same digit may be used within the same assembly. References to symbols of the form "*n*b" refer to the first numeric label "*n*:" backwards from the reference; "*n*:" symbols refer to the first numeric label "*n*:" forwards from the reference.

5.3. Null Statements

A null statement is an empty statement and is ignored by the assembler. A null statement may, however, be labeled.

* *bss* stands for "block starting with symbol". By using *bss* one may minimize the size of load modules and optimize the speed at which modules are loaded into Chap scratchpad.

5.4. Keyword Statements

A keyword statement begins with one of the many predefined keywords known to *chas*; the syntax of the remainder of the statement depends on the keyword. The remaining keywords are assembler pseudo-operations, also called *directives*. The pseudo-operations are listed in §7 together with the syntax they require.

6. Expressions

An expression is a sequence of symbols representing a value. Its constituents are identifiers, constants, operators, and backslash-parentheses, ("\(" and "\)"). Each expression has a type.

All operators in expressions are fundamentally binary in nature. Arithmetic is two's complement and has 32 bits of precision. *Chas* supports only limited arithmetic with fixed point numbers. There are four levels of precedence, listed here from highest precedence level to lowest:

precedence	operators
unary	–, ~
binary	*, /, %
binary	+, –
binary	<<, >>
binary	| , &, ^, nand, nor, xnor

All operators of the same precedence are evaluated strictly left to right, except where the evaluation order is enforced by parenthesis.

6.1. Expression Operators

The operators are:

operator	meaning
+	addition
–	(binary) subtraction
*	multiplication
/	division
%	modulo
–	(unary) 2's complement
&	bitwise and
|	bitwise or
^	bitwise exclusive or
~	bitwise 1's complement
>>	logical right shift
<<	logical left shift
nand	bitwise not and
nor	bitwise not or
xnor	bitwise not exclusive or

Expressions may be grouped by use of backslash-parentheses, "\(" and "\)". Only the – (unary and binary), +, *, and / operators may be used with fixed point numbers.

6.2. Data Types

The assembler manipulates several different types of expressions. The types which may be met are:

undefined

Upon first encounter, each symbol is undefined. It will remain undefined if it is assigned an undefined expression. It is an error to attempt to assemble an undefined expression in pass 2; in pass 1 it is not (except that certain keywords require operands which are not undefined).

undefined external

A symbol which is declared **.globl** but not defined in the current assembly is an undefined external. If such a symbol is declared, the runtime monitor/loader must be used to combine the assembler's output with another routine that defines the undefined reference.

absolute integer

An absolute integer symbol is defined ultimately from a constant which has no fractional portion. Its value is unaffected by any possible future applications of the link-editor to the output file.

absolute fixed point

An absolute fixed point symbol is defined ultimately from a constant which has a fractional portion. Its value is unaffected by any possible future applications of the link-editor to the output file.

text

The value of a text symbol is measured with respect to the beginning of the text segment of the program. If the assembler output is link-edited, its text symbols may change in value since the program need not be the first in the link editor's output. Most text symbols are defined by appearing as labels. At the start of an assembly, the value of "**.**" is "text".

data, bss

The value of a data (bss) symbol is measured with respect to the origin of the data (bss) segment of a program. Like text symbols, the value of a data (bss) symbol may change during a subsequent link-editor run since previously loaded programs may have data (bss) segments. After the first **.data** (**.bss**) statement, the value of "**.**" is "data" ("bss").

external absolute, text, data, or bss

Symbols declared **.globl** but defined within an assembly as absolute, text or data symbols may be used exactly as if they were not declared **.globl**; however, their value and type are available to the link editor so that the program may be loaded with others that reference these symbols.

6.3. Type Propagation in Expressions

When operands are combined by expression operators, the result has a type which depends on the types of the operands and on the operator. For purposes of expression evaluation the important types are

undefined
undefined external
absolute (either integer or fixed point)
text
data
bss
external absolute, text, data, or bss

The combination rules are:

1) If one of the operands is undefined, the result is undefined.

2) If both operands are absolute, the result is absolute.

3) If an absolute is combined with one of the external types, the result has the type of the external. If an external type is combined with a type other than external, the external is treated as an absolute.

Further rules applying to particular operators are:

+ If one operand is text- data-, or bss- segment relocatable, or is an undefined external, the result has the postulated type and the other operand must be absolute.

– If the first operand is a relocatable text-, data-, or bss- segment symbol, the second operand may be absolute (in which case the result has the type of the first operand); or the second operand may have the same type as the first (in which case the result is absolute). If the first operand is external undefined, the second must be absolute. All other combinations are illegal.

others It is illegal to apply these operators to any but absolute symbols.

The following rules apply to fixed point arithmetic with two absolute expressions:

+, − It is illegal to combine an 11-bit fixed point value with a 14-bit fixed point value. The type of the resultant expression is either an 11-bit or 14-bit fixed point value depending on the type of the operands.

*, / One of the operands must be a 14-bit fixed point value. If one of the operands is an integer constant, the resultant type is integer. If one of the operands is an 11-bit fixed point constant, the resultant type is an 11-bit fixed point constant.

7. Pseudo-operations

The keywords listed below introduce directives or instructions, and influence the later behavior of the assembler. The metanotation ''[stuff]'' means that 0 or more instances of the given ''stuff'' may appear. **Boldface** tokens must appear literally; words in *italics* are substitutable. The pseudo-operations listed below are grouped into functional categories.

7.1. Interface to Previous Pass

# *number* [*file-name*]

This directive (normally produced by the C preprocessor) causes the assembler to believe it is on line *number*. The second argument, if included, causes the assembler to believe it is in file *file-name*, otherwise the current file name does not change. The ''#'' *must* be in the first column.

7.2. Segment Control

.data [*expr*]
.bss [*expr*]
.text [*expr*]

These three pseudo-operations cause the assembler to begin assembling into the text, data, or bss segment. If specified, the expression indicates a new value for the text or data location counter. Any memory locations skipped as a result of moving the location counter are zero filled. The expression must be defined and absolute; an omitted expression implies use of the current location counter value. The locations in the data and bss segments are mapped into the Chap scratchpad RAM; locations in the text segment are mapped into the Chap instruction RAM.

Each instruction in the text segment increments the text segment location counter by one. Each word of data allocated in the data (bss) segment increments the data (bss) segment location counter by one.

.comm *symbol*, *expr*

Define a common block with size *expr* words. The **.comm** directive implicitly declares *symbol* as an external identifer. The symbol may not previously have been defined as other than a common block.

chas does not allocate storage for common symbols; this task is left to the link editor, *chld*. The link editor computes the maximum declared size of each common symbol (which may appear in several files), allocates storage for it in the final *bss* section, and resolves linkages.

.space *expr*

Add *expr* to the current location counter and zero fill the resultant space. The value of *expr* must be defined and absolute.

.align *expr*

Round the current location counter to the next multiple of *expr*. If the location counter is incremented, the resultant space is zero filled. The value of *expr* must be defined and absolute.

7.3. Initialized Data

Scratchpad memory locations may be initialized at assembly time with two directives. Initialized memory locations must be specified as tessellated or untessellated. Data and bss segments generated by *chld* are guaranteed to start on a 16 word boundary in scratchpad memory. When tessellated and untessellated initialized memory locations are mixed in a single segment, *chas* automatically zero pads to a 4 word boundary between each region.

7.3.1. Untessellated Data

.word *expr* [, *expr*]

> The expressions in the comma-separated list are truncated to 16-bit values and assembled in successive locations without tessellation. Expressions supplied in a **.word** may be integer or fixed point.
>
> If the previous initialization contained tessellated data, the location counter is aligned to a four word boundary prior to processing the untessellated data.

7.3.2. Tessellated Data

.pixel *expr* [, *expr*]

> The expressions in the comma-separated list are truncated to 16-bit values and assembled in a *tessellated* manner in successive logical scatchpad memory locations. Expressions may be integer or fixed point. Fixed point constants are stored in the appropriate fractional format (see §3.2).
>
> If the previous initialization contained untessellated data, the location counter is aligned to a four word boundary prior to processing the tessellated data.

7.4. Symbol Definitions

7.4.1. External Symbols

.globl *name*

> This statement makes the *name* external. If it is otherwise defined by appearance as a label, it acts within the assembly exactly as if the **.globl** statement were not given; however, the link editor may be used to combine this object module with other modules referring to this symbol.
>
> Conversely, if the given symbol is not defined within the current assembly, the link editor can combine the output of this assembly with that of others which define the symbol. The assembler makes all otherwise undefined symbols external.

7.4.2. Absolute Symbols

.set *name value*

> This statement defines an absolute symbol *name* with value *value*. The symbol may not previously have been defined with type other than absolute.

7.5. Default Instruction Duration

.clock *n*

> This statement causes the instruction duration assigned each Chap microinstruction to be *n* clock cycles (unless specified explicitly in the instruction, see §8.10). By default the instruction duration is calculated according to a set of rules which are intended to provide a minimal value. If *n* is 0, *chas* resumes automatic calculation of the instruction duration.

7.6. Multiplier Shift Control

.shift *qualifier n*

> This statement defines a multiplier input shift qualifier named *qualifier*. When used in qualifying a multiplier input the quantity is shifted left *n* places. Three qualifiers are predefined as shown below.

Qualifier	Shift	Value Type
comp	2	component
alpha	3	alpha channel
coeff	0	coefficient

Shift qualifiers share scope with ''absolute'' symbols.

8. Chap Directives

Directives for controlling the channel processor fall roughly into six categories:

- controlling the 29116 ALU,
- controlling the 29517 multiplier,
- scratchpad memory address calculation,
- managing movement of data through the datapaths,
- sequencing, and
- controlling the runner.

In addition to the above, *chas* directives exist for directly defining many of the microinstruction fields.

Where data movement is involved, the assembler eliminates the need to give explicit directions regarding control signals. Instead, the architecture is presented at a register transfer level with the assembler translating register transfer requests into assertion of the appropriate control signals. The assembler, however, will not translate data transfer requests which would result in multiple Chap microinstructions; these are flagged as errors and reported to the user.

8.1. ALU Control

Most ALU operations are represented by assignment statements of the form

data-location = expression;

The possible ALU data locations are shown in Table 1.

Assembler	Interpretation
r0, r1, ..., r31	internal RAM location
acc	accumulator
latch	latched ALU data input
ybus	ALU data output
link	ALU status word link bit
flag1	ALU status word flag1 bit
flag2	ALU status word flag2 bit
flag3	ALU status word flag3 bit
overflow	ALU status word overflow bit
negative	ALU status word negative bit
carry	ALU status word carry bit
zero	ALU status word zero bit

Table 1. ALU Data Locations.

Expressions on the right hand side of an ALU assignment statement translate into 29116 operations. Thus, the expressions may be simple values (resulting, for example, in a ''move'' operation), or more complex values which utilize the 29116's arithmetic capability. Table 2 shows the correspondence between assembler expressions and 29116 operations; *op* means an operand (either an ALU location or an expression), *loc* means an ALU location, and *expr* means an absolute constant expression.

Commutative operators may have their *op* and *loc* parameters swapped. For example, *op* + *loc* is accepted

	Expression	Instruction Type	Opcode
	op	single operand	MOVE, COMP, INC, NEG
+	loc – op [**with carry**]	double operand	SUBx [SUBxC]
+	loc + op [**with carry**]	double operand	ADD [ADDC]
+	loc & op	double operand	AND
+	loc **nand** op	double operand	NAND
+	loc ^ op	double operand	EXOR
+	loc **nor** op	double operand	NOR
+	loc | op	double operand	OR
	loc **xnor** op	double operand	EXNOR
	loc << expr [**fill with** expr]	single bit shift	SHUPx
	loc >> expr [**fill with** expr]	single bit shift	SHDNxx
	[~] **pow2**(expr)	bit oriented	LD2Nx [LDC2N]
	loc + **pow2**(expr)	bit oriented	A2Nx
	loc – **pow2**(expr)	bit oriented	S2Nx
+	loc **rotate** expr	rotate by n bits	ROTRx
	loc **rotmerge** expr, mask	rotate and merge	ROTM
	prior(loc [, mask])	prioritize	PRTx
	[**reverse**] **crc**	crc	CRCx

Table 2. ALU Expression to Opcode Mapping.

by the assembler. A few ALU operations are not expressible as "standard" assignments. Table 3 shows how these operations are represented to the assembler. Bit operations involving the status word use expressions of the sort "**overflow**" or "**zero**". Bit operations on a single bit use an expression "**bit**(*n*)", to effect the n^{th} bit in the word or byte. To request the overflow, carry, negative, and zero bits as an operand use "**overflow, carry, negative, zero**".

Operation	Assembler syntax
bit set	loc |= **bit**(expr);
bit set status	loc |= expr;
bit clear	loc &= **bit**(expr);
bit clear status	loc &= expr;
bit test	loc == **bit**(expr);
bit test status	loc == expr;
rotate and compare	loc == op **rotmerge** expr , mask;
no-operation	(*an empty statement*)

Table 3. Miscellaneous ALU Operations.

Finally, the ALU status flags may be disabled by compounding an ALU operation with "**not sce**", i.e.

{ ALU-operation; **not** sce; }

(**sce** stands for "status control enable", as defined in the Chap instruction description).
If an ALU operation is not specified, a no-operation is supplied by the assembler.

8.2. Multiplier Control

The 29517 multiplier inputs and outputs are referenced in much the same way as the 29116 ALU.

8.2.1. Input Control

To supply an input to the multiplier a statement of the following form is used.

multiplier-input = input-specification;

The "multiplier-input" is either **multx** or **multy**. The "input-specification" identifies a source in the Chap from which the input value is to be obtained, as well as optional qualifiers to control and modify the data value. Input data is normally treated as a ~~signed~~ unsigned quantity; this may be overridden on a per instruction basis with an explicit specification,

(signed) input-source
(unsigned) input-source

In addition, the multiplier input data may be shifted one to three places before presentation to the 29517. Coefficient quantities are normally not shifted, while component values are shifted two places and alpha values are shifted three places. Shifting must be explicitly specified with an additional input qualification,

(signed *qualifier***)** input-source
(unsigned *qualifier***)** input-source

Three qualifiers, *coeff*, *comp*, and *alpha*, are predefined by *chas*, others may be added with the **.shift** directive (see §7.6).

8.2.2. Output Control

Products generated by the 29517 can be accessed only 16-bits at a time. The least significant and most significant portion of the product are referenced with

lsp
msp

and may appear only on the right hand side of an assignment statement.

The multiplier output is, by default, adjusted for 32-bit precision and not rounded. To affect either or both of these, the following statements must be specified **at the time the last operand is supplied to the multiplier.**

round;	(round product)
adjust;	(produce 31-bit product for multi-precision arithmetic)

Finally, updating of the multiplier output register may be disabled by specifying

not mce;

(**mce** stands for "multipler clock enable").

The default values for **round, adjust,** and **mce** may be explicitly specified:

not round;
not adjust;
mce;

8.3. Datapath Control

Datapath control signals are implicitly specified through higher level assembler constructs which provide the programmer with a register transfer-like interface to the Chap. All data transfers in the machine are represented as assignment statements of the form

data-destination = data-source;

where data destinations and sources are either Chap "registers" (crossbar, scalar devices, runflag, etc.), multiplier inputs and outputs, or scratchpad memory locations. *Chas* translates statements of this type into assertions of the signals necessary to generate the data transfer. If multiple destinations are to be enabled on a data transfer, multiple assignment statements may be grouped into a single compound statement (using braces), or multiple assignment statements may be cascaded. For example, to transfer data from a single source to multiple destinations, a statement of the form

data-destination$_1$ = data-destination$_2$ = data-source;

might be used. *Chas* will not translate an expression which would result in multiple Chap instructions; these are flagged as errors.

8.4. Processor Register Mnemonics

Table 4 lists the non-memory locations in the Chap where data can reside; memory references are discussed in the next section.

Mnemonic	Chap Location
alu	ALU input data latch/output Y-bus
b0, ..., b15	base registers
i0, ..., i15	index registers
lsp	least significant part of multiplier output
msp	most significant part of multiplier output
pbus	Pbus data register
pbus a0	Pbus address register 0
pbus a1	Pbus address register 1
pbus csr	Pbus control status register
sp	stack pointer register
spad	scratchpad address register
status	Chap status register (read-only)
sysbus<*n*>	Sysbus registers
tos(pc)	top of program counter return stack
tos(lc)	top of loopcounter stack
tos(runflag)	top of runflag stack
wrxbar	write crossbar
rdxbar	read crossbar
yapbus	Yapbus data buffer
yapbus csr	Yapbus control status register

Table 4. Chap Non-memory Data Locations.

Certain locations refer to different Chap registers depending on whether they appear on the left or right hand side of an assignment statement. For example, the **alu** keyword refers to the output Ybus register when used on the right hand side of an assignment, and the input data latch register when it appears on the left hand side.

The Sysbus interface is presented as an array of 16 registers. To reference Sysbus register *n*, an expression of the form

sysbus<*n*>

is used.

8.5. Scratchpad Memory

References to scratchpad memory may be used as source or destination operands in an assignment statement. There are four modes in which scratchpad memory may be accessed: component, pixel, broadcast, and indexed. Each access mode is represented to the assembler with a different syntax.

8.5.1. Scratchpad Access Modes

Pixel access is indicated by a parenthesized expression,

(expr)

Using pixel access each processor reads and writes one component of a four component pixel value in the tesselated scratchpad memory.

Component access is indicated by an expression of the form,

<expr>

Using component access all processors read and write the same component of four adjacent pixels.

Broadcast access is indicated by an expression of the form,

@expr

Using broadcast access, all processors read from one location, while one processor writes to one memory location.

Indexed accesses are represented by expressions of the form,

[expr]

In indexed mode each processor offers an index address to read four values; for writing, the processors write to four components of a scratchpad pixel pointed to with the base and index register mechanism described below. **Index mode accesses to scratchpad memory are untessellated.**

8.5.2. Scratchpad Address Calculation

The ''expression'' supplied in a scratchpad memory reference is composed of a scratchpad address and, optionally, a component override directive. Scratchpad addresses may be setup ahead of time in the scratchpad address register **spad** (to take advantage of pipeline overlaps), or calculated ''on the fly'' in 3 or 4 clock tick instructions.

Addresses for component, pixel and broadcast mode accesses are calculated from a base register and an index register. Either the value of the base register or the sum of base and index registers may be used. In addition, in the same instruction, the resultant address calculation may be stored back into the base register; this allows for pre- and post- auto-increment and auto-decrement access modes.

An address is specified as the contents of a base register, or the sum of a base and index register,

base-reg
base-reg + index-reg

Base registers are denoted by **b0, b1, ..., b15** while index registers are denoted by **i0, i1, ..., i15**.

To force an address to be stored into the base register involved in the calculation, an assignment statement should be specified in the same instruction indicating the value to be stored. The following examples illustrate the four possibilities.

```
acc = (b0);
acc = (b0+i3);
{ acc = (b0); b0 = b0 + i3; }
{ acc = (b0 + i3); b0 = b0 + i3; }
```

As a convenient shorthand, the following are treated identically

b*n* = b*n* + i*n*; <=> **b*n*++;**

Note that if an index register is loaded with a (2's complement) negative value, the base register will be decremented.

Addresses for index mode accesses must come from the Abus. This means the address may still be stored in a base register, but it can not be computed from the sum of a base and index register, unless carried out ahead of time in an alu. To use an Abus value, or other register, for an index mode address, one simply references it. For example,

multx = [**r0** + **acc**];

computes the index mode address in each alu.

8.5.3. Scratchpad Address Component Selection

The base and index mechanism for scratchpad memory access provides a pointer to a four component scratchpad location. To complete the scratchpad memory reference, a ''component select'' field must be specified. In pixel access this field specifies whether the processors receive RGBA, GBAR, BARG, or ARGB. In component access this field specifies whether the processors receive red, green, blue, or alpha. In broadcast mode, this field is used **only when writing** and indicates the processor from which the value is to be taken. In indexed mode this field is unused.

In pixel access, the default component select is RGBA. To override this value an expression of the form

(expr, **gbar**)

may be supplied (where the symbol **gbar** is defined to be 1, **barg** is 2, and **argb** is 3).

In component access the default component select is red. To override the default value an expression of the form

<expr, **green**>

may be used (where the symbol **green** is defined to be 1, **blue** is 2, and **alpha** is 3).

In a broadcast mode write, the default component select is for processor 0 to perform the write. To override the default value an expression of the form

@expr, **3**

may be used.

In indexed mode any component select is ignored (the assembler always sets the field to 0).

8.6. Sequencing

8.6.1. Low-level Sequencing

Flow control in the Chap is expressed with a set of commands to the sequencer and runner. Each command has its own syntax:

jsr *destination*
jsr [*cc-spec*] **to** *destination*
push [*loopcounter*]
pop [*pop-count*]
ifdo [*cc-spec*] **otherwise** *destination*
ifelse [*cc-spec*] **otherwise** *destination*
continue [*cc-spec*]
elsedo [*n*] **otherwise** *destination*
force *processors*
dowhile [*cc-spec*]
whiledo [*cc-spec*] **otherwise** *destination*
break [*n*] [*cc-spec*] [**to** *destination*]
return [*n*] [*cc-spec*]
goto *destination*

The commands are as described in the ''Chap Instruction Description''. A condition code specification, *cc-spec*, is of the form [**any** | **all**] [**!** | **not**] *cc*

where the '**!** or '**not**' inverts the local condition code polarity (taking the "not" of the logical test to be true) and the **any** and **all** qualifiers are used to check a condition over any or all processors. The **return**, **break**, and **elsedo** commands take an optional pop count expression, *n*, which defaults to one. The **push** command takes an optional *loopcounter* argument which defaults to zero. If no sequencing request is specified for an instruction, *chas* supplies a command which sequences to the next instruction. The *destination*, *loopcounter*, and *processors* arguments may be any expression which is accessible on the scalar bus.

A condition code, *cc*, is one of the following†

true	
false	
stackover	(stack overflow) (sequencer stack)
videosync	(video sync active) (Ø/1 depending on which half of interlace is displayed)
lc zero	(loopcounter zero) (separate register called 'loopcount' is checked)
yapbus busy	
pbus busy	
sysbus busy	
sysbus read	
sysbus write	
negative	(ALU status N set)
positive	(not ALU status N or Z set)
zero	(ALU status Z set)
overflow	(ALU status N ^ OVR set)
carry	(ALU status C set)

Conditions codes related to the ALU may also insert '**alu**' before the condition code specification; e.g. '**alu carry**'.

The full condition code specification may limit the list of source processors by appending an expression of the form

[*processor-list*]

where a "processor list" specifies a set of processors by number or channel assignment. For example, to check the carry flag in the ALU of processors 0 and 3 only, the following expression could be used,

carry[0, 3]

For sequencer commands which have two corresponding sequencer instructions, the instruction chosen is dependent on the condition code specification. If the condition code selected is independent on the state of the runflags (**true** through **sysbus write**), then the runflag-independent sequencer instruction will be used. Otherwise, the use of **any** or **all** in the condition code specification forces a runflag-independent instruction to be used. For example, consider the following sequencer commands:

ifdo lc zero otherwise .+1
ifdo alu zero otherwise .+1
ifdo any alu zero otherwise .+1

In the first command *chas* would use "ifdo4" because the condition code was independent of the runflags. In the second and third commands the condition code is runflag-dependent, and so "ifdo" is used for the second command, while "ifdo4" used in the third (since **any** is specified).

† For compatibility with previous versions of *chas*, the following condition codes are also accepted: **lc** <=, <, and <=.

8.6.2. High-level sequencing

Several high-level control constructs are available for program flow control. These constructs expand into multiple Chap instructions which use the low-level sequencing instructions previously described. In the following table *cc-spec* refers to a condition code specification (as above) and *block* refers to a set of one or more statements. The *cc-check* is an optional *chas* statement which [is expected to] generate the condition code status for the sequencer instruction.

> **if** *cc-spec* [(*cc-check*)] **then** *block* **fi**
> **if** *cc-spec* [(*cc-check*)] **then** *block* **else** *block* **fi**
> **do** *block* **while** *cc-spec* [(*cc-check*)] **done**
> **while** *cc-spec* [(*cc-check*)] **do** *block* **done**
> **loop** *expr* **do** *block* **done**

The "if", "do", and "while" constructs have obvious meaning. The "loop" construct is used to perform the block of code *expr* times, using the loopcounter.

The "do" and "while" constructs expand to include a '**push** 0' instruction at the top of the loop. Thus they are unusable with an '**lc zero**' condition code; instead the "loop" construct should be used.

8.7. Yapbus-Pbus Auto-increment

To request the Yapbus or Pbus pointer auto-increment facility either, or both, of the following may be specified:

> **yapbus++;**
> **pbus++;**

8.8. Runner Control

To restrict the default actions of the runner for a single instruction, an immediate runflag may be specified with statements of the form,

> **runflag** = *expr*;

1 1 1 1
3 2 1 0
a b g r

expr is 4-bit number specifying a runflag value. Bit *i* corresponds to processor *i*, (i=0,1,2,3). A 1 indicates that the processor is active; a zero, that it is inactive. The runflag is the logical **and** of this value and the default runflag.

8.9. Breakpoint Control

To request the Chap to breakpoint on an instruction (saving state and stopping the clock), a statement of the form

> **bpt;**

may be specified.

8.10. Special Bit

Chas attempts to recognize situations where the "special bit" is required and generate the flag automatically. To guarantee the value of the special bit, it may be explicitly specified.

> **special;**
> **not special;**

8.11. Instruction Duration

Chas attempts to automatically generate the minimum instruction duration for each instruction assembled. In case this value must be explicitly set, the **.clock** directive described in section §7.5 may be used. Alternatively, to override the default calculation on a per instruction basis, the duration may be specified explicitly,

n **ticks;**
n **tick;**

where *n* is between 1 and 4.

8.12. Abus Component Selection

When moving data from the Abus to a scalar device on the Sbus it is necessary to specify one of the four possible Abus arithmetic units as the actual source of data. By default, *chas* selects arithmetic unit 0, but this may be changed by appending an explicit specification of the form

[*processor*]

where *processor* is a number, 0, 1, 2, or 3.* Thus, for example, to assign the lower 16 bits of the multiplier output from processor 3 to base address register 5, the following expressions would be used:

b5 = **lsp**[3];

* The processor number used is actually the modulus of the indicated value. Thus, processor 4 translates to processor 0, processor 5 to processor 1, etc.

9. Caveats and Notes

- The algorithm used in calculating instruction durations is the following. For each *chas* assignment statement the minimal instruction duration is calculated as described in Appendix 1 of "The Chap Instruction Description". If multiple assignment statements are grouped in a single Chap instruction with "{}", the duration of the overall instruction is the **minimum of the durations for each expression.** This implies that a construct such as

  { multx = rdxbar; rdxbar = (b0+i0); }

  will be assigned an instruction duration of two clock ticks and result in the x input to the multiplier receiving whatever value is currently being fetched from scratchpad memory. Contrast this with the same instruction, but with a 3 clock tick duration:

  { multx = rdxbar; rdxbar = (b0+i0); 3 ticks }

  This construct is equivalent to

  multx = (b0+i0);

  In general, whenever pipeline overlaps are being utilized in programs, the statements which assume the pipeline overlaps should provide explicit instruction durations.
- While *chas* will not assemble complex expressions which would require multiple Chap instructions, it may be forced to generate multiple instructions when presented with a data transfer which requires the "special bit" (described in the Chap Instruction Description). To force *chas* to print warnings about automatic generation of the special bit, the –s flag should be specified on the command line.
- The runtime monitor, *charm*, and the dynamic loader, *chload*, which load assembled modules into a Chap, merge **.global** bss and data segments (when possible) to allow data structures to be shared between modules.

Appendix: Sequencer Instructions

This appendix describes the low-level control commands mentioned in 8.6.1.

The chap instruction word contains many fields of information; the sequencer instruction occupies a small subset of these fields. Each sequencer instruction is executed in parallel with any other bus, ALU, or multiplication operations specified in the instruction word.

The sequencer's job is to oversee chap instruction execution. Toward this end the sequencer maintains the the program counter (PC), and the runflag (RF), that specifies which of the four processors may potentially execute the next instruction. Just as the computed PC is the PC for use in the next instruction, the RF is the RF for use in the next instruction. LP A loop counter (LC) is also provided as part of the sequencer's control facility, relieving the ALU of some decrement and test duties, but more importantly, allowing nested loops. For nested control structures, the sequencer maintains a stack to save the state of PC, RF, and LC.

The following is a description of each sequencer instruction and how it is used. Boldface indicates an assembler reserved word. Italics indicates a user-specified expression or condition code. Items in brackets are optional. *cc-spec* defaults to **true**.

jsr *destination --- jump to subroutine*

jsr *[cc-spec]* **to** *destination*

The first form of **jsr** pushes PC+1 and sets PC to *destination*, thus performing a standard subroutine call. The second form specifies an optional condition code. If *cc-spec* is true for any processor, a normal jsr is performed as just described. If *cc-spec* is false for all processors, no special sequencer action is taken and PC is simply incremented. Note that since *cc-spec* defaults to **true**, **jsr to** *destination* means the same as **jsr** *destination.*

push *[loopcounter] --- define the top of a dowhile loop*

The **push** instruction stores PC+1, the optional **loopcounter**, and RF on the stack and then increments the stack pointer. **Push** is normally used to define the top of a loop for **dowhile.** Loopcounter is optional, often used to specify the number of iterations for **dowhile not lc zero. Loopcounter** may be an ALU expression.

ifdo *[cc-spec]* **otherwise** *destination --- simple conditional*

If *cc-spec* is true for any processor, PC+1 and a new value of RF are pushed and execution continues at PC+1. Each processor for whom *cc-spec* was false has its RF bit turned off. If *cc-spec* is false for all processors, execution continues at *destination.* Normally, the statement label *destination* is preceded by a **pop.** Be sure that the stack is popped exactly once for each executed **ifdo.**

ifelse *[cc-spec]* **otherwise** *destination --- two part conditional*

When used in conjunction with **elsedo, Ifelse** allows the programmer to specify that each currently running processor will execute either of two blocks of code. If *cc-spec* is true for any processor a new RF is pushed and execution continues at PC+1. Each processor for whom *cc-spec* was false has its bit turned off in the new RF. If *cc-spec* is false for *all* processors, the *current* RF is pushed and execution continues at *destination.* The first block of code is normally followed by an **elsedo** preceding the statement at *destination.* The second block of code must be followed by a **pop** or other instruction that pops the stack.

ifdo4 and **ifelse4**

These instructions are never used explicitly, but are generated by chas(1) when the condition code specification "any" or "all" is used, and when the condition code is not processor dependent. **Ifdo4** and **ifelse4** are identical to **ifdo** and **ifelse**, respectively, except that they do not alter RF. This means that *all* currently running processors will execute or not execute a given block of code.

continue *[cc-spec] --- goto top of loop*

If *cc-spec* is true for any processor, **continue** loads PC from the top of the stack. **continue** most often appears at the bottom of a **whiledo** loop. **Continue** is also commonly used from within a **dowhile** or **whiledo** loop to skip the remaining instructions in a loop.

force *processors --- set runflag*

force sets RF to the designated value and pushes the stack. Be sure to pop the stack before leaving the affected block of code.

dowhile *[cc-spec] --- loop until*

If *cc-spec* is true for any processor, **dowhile** loads PC form the top of the stack and decrements LC. Each processor for whom *cc-spec* is false has its RF bit turned off. If *cc-spec* is false for all processors, the stack is popped and control passes to PC+1. Normally, a **Push** instruction precedes the top of a **dowhile** loop.

whiledo *[cc-spec]* **otherwise** *destination --- loop while*

If *cc-spec* is true for any processor, LC is decremented and control passes to PC+1. Each processor for whom *cc-spec* was false has its RF bit turned off. If *cc-spec* is false for all processors, control passes to *destination.* Normally, **whiledo** is preceded by a **push.** The bottom of a **whiledo** loop is normally a **continue.**

dowhile4 and **whiledo4**

These instructions are never used explicitly, but are generated by chas(1) when the condition code specification "any" or "all" is used, and when the condition code is not processor dependent. **dowhile4** and **whiledo4** are identical to **dowhile** and **whiledo,** respectively, except that they do not alter RF. This means that *all* currently running processors will execute or not execute a given block of code.

elsedo *[n]* **true otherwise** *destination --- else portion of ifelse*

Elsedo is normally used following an **ifelse.** The current RF is complimented and AND'ed with the RF stored on the stack, turning *on* each processor whose *cc-spec* was false when the **ifelse** was executed. The new RF is then tested and if it is nonzero, execution continues at PC+1. If the new RF is zero, the stack is popped and execution continues at *destination.* A **pop** should precede the statement at *destination. N* is the number of pops to perform and may equal 1, 2 or 3; the default is one pop. This cleans up the stack when transferring control, for example, from within a loop.

return *[n] [cc-spec] --- return from subroutine*

If *cc-spec* is true, PC, RF, and LC are loaded from the stack and the stack is popped. *N* is the number of pops to perform and may equal 1, 2 or 3; the default is one pop. This cleans up the stack when returning, for example, from within a loop. No-op if *cc-spec* is false.

break *[n] [cc-spec] [***to** *destination] --- exit a loop or conditional*

If *cc-spec* is true for any processor, control passes to *destination* and the stack is popped. *N* is the number of pops to perform and may equal 1, 2 or 3; the default is one pop. This cleans up the stack, for example, when breaking out of a nested loop. If *cc-spec* is false, control passes to PC+1. Normally, **break to** is used to "break out of" an **ifdo, ifelse, dowhile,**or **whiledo. break** with no *destination* causes the chap to halt and interrupt the host. **Break** is the same sequencer instruction as **return,** but with tos(pc) set to *destination.*

endfBpop *[n]* --- pop the stack

Chas(1) generates a **dowhile** (false) from either of these two instructions. **Push** is usually used to define the top of a **whiledo** or **dowhile** loop and to end an **ifdo** or **ifelse.** *N* is the number of pops to perform and may equal 1, 2 or 3; the default is one pop.

goto *destination*

goto passes control to *destination.* Chas(1) generates an **ifdo** (false) for this instruction.

Appendix: Sequencer Instruction Table

The following table gives a quick summary of each sequencer instruction. PC, LC, and RF have the same meaning as in the previous appendix. Runner condition (RC) is used in this table to calculate the next value of RF - it has a bit set for each processor for whom **cc-spec** is true.

In general, "Sbus" means the immediate field of the instruction. However, for many instructions, including **push**, chas(1) will generate the appropriate multi-word instruction to load "Sbus" from any source, including the ALU.

		Sequencer Condition TRUE				Sequencer Condition FALSE			
n	name	PC	LC	RF	stack	PC	LC	RF	stack
0	jsr	Sbus	n/c	n/c	push	PC+1	n/c	n/c	n/c
1	push	PC+1	Sbus	n/c	push	PC+1	Sbus	n/c	push
2	ifdo	PC+1	n/c	RF&RC	push	Sbus	n/c	n/c	n/c
3	ifelse	PC+1	n/c	RF&RC	push	Sbus	n/c	RF&!RC	push
* 4	ifdo4	PC+1	n/c	n/c	push	Sbus	n/c	n/c	n/c
* 5	ifelse4	PC+1	n/c	n/c	push	Sbus	n/c	n/c	push
6									
7	force	PC+1	n/c	Sbus	push	PC+1	n/c	Sbus	push
8	dowhile	stack	LC-1	RF&RC	n/c	PC+1	stack	stack	pop
9	whiledo	PC+1	LC-1	RF&RC	n/c	Sbus	stack	stack	pop
10									
11	continue	stack	n/c	n/c	n/c	PC+1	n/c	n/c	n/c
* 12	dowhile4	stack	LC-1	n/c	n/c	PC+1	stack	stack	pop
* 13	whiledo4	PC+1	LC-1	n/c	n/c	Sbus	stack	stack	pop
14	elsedo	PC+1	n/c	!RF&stack	n/c	Sbus	stack	stack	pop
15	return/break	stack	stack	stack	pop	PC+1	n/c	n/c	n/c

Moving to the next instruction is accomplished by *continue(FALSE)* or *jsr(FALSE)*. An *else4* command is missing because it is equivalent to *whiledo4(FALSE)*.

* Automatically generated by the assembler if "any" or "all" condition qualifiers are used. "4" may be replaced by "*" by the disassembler.

Chap Runtime Monitor Reference Manual
— CHARM —
Copyright 1986, Pixar

This document describes the usage and syntax of the Pixar Channel Processor Runtime Monitor *charm*. Readers of this document should be familiar with the Chap architecture; no description is presented here.

The runtime monitor serves two functions. It primarily acts as an interactive debugger, similar to adb, for programs executing in a Chap. Second, it may be used to dynamically link-edit and relocate object modules produced by the Chap Assembler, *chas*. *Charm* executes on a host machine, communicating with a Chap through a memory-mapped bank of diagnostic registers on the Pixar-Host Interface Board (PHIB). Certain asynchronous events in the Chap are communicated to *charm* through UNIX† signals.

1. Usage

Charm is invoked as follows

charm [**–x**] [**–I***dir*] [*chap-device*]

The **–x** flag instructs *charm* to open the specified Chap device with exclusive access; this overrides the normal shared access mode.

If a specific Chap is to be used, its associated file may be specified on the command line; *charm* uses "/dev/chap0" by default.

The **–I** flag may be used to specify directories containing command scripts. Normally, *charm* searches only in the directory "/u1/gfx/pixar/chap/lib/charm". Multiple directories may be specified using **-I** several times.

2. Charm Facilities

2.1. Linking and Loading

Charm allows the user to load individual files containing relocatable object modules. If a file contains *references* to symbols defined in code already resident in the Chap, these references are patched to reflect the resident code. Similarly if a file *defines* new symbols, unresolved in code already resident, *charm* will patch the references in the resident code. *Charm* will permit modules to be loaded with undefined references, but will not load a module if it redefines symbols already resident in the Chap.

Loading programs into the Chap requires several tasks other than resolving undefined references:

1. Load instruction and scratchpad RAM.
2. Bind relocatable references based on locations assigned to load modules.
3. Record the load module and global symbol locations in the symbol table maintained on the host.

Charm performs the first two tasks during the linking-loading phase. The symbol table is updated upon exiting *charm*.

† UNIX is a trademark of Bell Laboratories.

2.2. Segment Mapping

In accordance with the Harvard architecture of the chap, *Charm* maps the *text* segment of relocatable object modules into the Chap instruction RAM and the *data* segment into the scratchpad RAM. Space is allocated to segments using tables maintained by the operating system. In addition to the kernel-based allocation tables, a symbol table is maintained for each Chap connected to a host. Symbol table maintenance is 'by convention'—no operating system support (other than synchronizing access to the symbol table file) is utilized.

2.3. User Control

With *charm,* a user may:

- examine or modify the contents of scratchpad, instruction RAM, or processor registers
- set breakpoints
- single step a Chap program
- trace the contents of scratchpad and processor registers
- load and bind programs

Charm uses the diagnostic interface to the Chap to provide all of these facilities.

3. Command Language Interface

Charm's user interface is very similar to that of the UNIX debugger *adb*(1)*. Charm uses a symbol called ".", or "*dot*" to refer to the current address (i.e., the address of the last item printed).

When *charm* is ready to accept commands from the keyboard, it prompts with "> " and waits for input. In general, requests to *charm* are of the form

[*address*] [, *count*] [*command*] [;]

If *address* is present, *dot* is set to *address*. Initially *dot* is set to 0. For most commands *count* specifies how many times the command should be executed. The default *count* is 1. *Address* and *count* may be expressions.

3.1. Expressions

Charm processes two types of expressions: those involving scalar quantities, and those involving vectors (of length 4). Where two scalar expressions are combined, the obvious arithmetic is performed. Combining two vector expressions results in a component by component application of the appropriate operator. When a vector and a scalar are combined, the scalar is combined with each element of the vector to generate a vector result. Constants are considered scalars. 4-way registers (e.g. the ALU accumulator) are treated as vector expressions.

. The value of *dot*.

+ The value of *dot* incremented by the current increment.

^ The value of *dot* decremented by the current increment.

" The last *address* typed.

integer A number. The prefixes "0x" and "0X" force interpretation in hexadecimal radix; the prefix "0" forces interpretation in octal radix; "0t" and "0T" force interpretation in decimal radix. If no prefix appears, the *default* radix is used; see the $d command. The default radix is initially decimal. The hexadecimal digits are 0123456789abcdefABCDEF, with the obvious values.

integer.fraction
A 16-bit Pixar fixed-point number. If the fraction is followed by an e or E, the number is

* Adb is one of the debuggers associated with Unix. A possible source of answers to questions about Charm is "A Tutorial introduction to ADB", in the "Supplementary Documents" part of the Unix programmer's manuals.

treated as a component value (eleven bits of fraction). By default, fixed-point numbers are treated as component values. The *integer* portion of a fixed-point number must be specified in base ten, either explicitly with a "0t" prefix, or implicitly by setting the input radix to 10; see the **$d** command. If the fraction is followed by an **f** or **F**, the number is treated as a coefficient value (fourteen bits of fraction).

<name The value of *name*, which is either a variable name or a register name. *Charm* maintains 36 variables: **a-z** and **0-9**. The register names are the same as those used by the Chap assembler; §3.4 provides a complete list.

symbol A *symbol* is a sequence of upper or lower case letters, underscores, or digits, not starting with a digit. The backslash character \ may be used to escape other characters. The value of the *symbol* is found by first checking the list of known registers, then, failing there, looking in the symbol table.

(*exp*) The value of the expression *exp*.

Monadic Operators

**exp* The contents of the tessellated scratchpad location addressed by *exp*.

@exp The contents of the untessellated scratchpad location addressed by *exp*.

–exp Integer negation.

~exp Bitwise complement.

!exp Logical negation.

Dyadic operators

Dyadic operators are left-associative and less binding than monadic operators.

e1+e2 Integer addition.

e1–e2 Integer subtraction.

*e1*e2* Integer multiplication.

e1%e2 Integer division.

e1&e2 Bitwise conjunction.

e1|e2 Bitwise disjunction.

e1#e2 Round *e1* up to the next multiple of *e2*.

3.2. Commands

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available.

?*f* Locations starting at *address* in instruction RAM are printed according to the format *f*. *Dot* is incremented by the sum of the increments for each format letter.

/*f* Locations starting at *address* in scratchpad RAM are printed according to the format *f* and *dot* is incremented as for "?".

=*f* The value of *address* itself is printed in the styles indicated by the format *f*. (This may not be used with the **i** format.).

A format, *f*, consists of one or more characters that specify a style of printing. Each format may be preceded by a decimal integer that is a repeat count for the format letter. While stepping through a format, *dot* is incremented by the amount given for each format letter. If no format is given, the last format is used.

Lower-case letter formats used with the / operator force *charm* to interpret the address as a tessellated address; upper-case letters cause the address to be interpreted as un-tessellated.

The format letters available are as follows:

o	1	Print the value in octal (O for untessellated).
d	1	Print the value in decimal (D for untessellated).
x	1	Print the value in hexadecimal (X for untessellated).
u	1	Print the value as an unsigned decimal number (U for untessellated).
e	1	Print the value as an 11-bit fixed point number (E for untessellated).
f	1	Print the value as a 14-bit fixed point number (F for untessellated).
i	1	Print the value as a machine instruction.
a	0	Print the value of *dot* in symbolic form. Symbols are checked to ensure that they have an appropriate type as indicated below:
	/	local or global data symbol
	?	local or global text symbol
	=	local or global absolute symbol (A for untessellated).
p	1	Print the addressed value in symbolic form using the same rules for symbol lookup as a (P for untessellated).
b	0	Print the value of *dot* in the form *pixel.component*, where the specified *component* is one of "RGBA" (B for untessellated).
z	1	Print the addressed value in the form *pixel.component*, as for the **b** format (Z for untessellated).
t	0	When preceded by an integer, tabs to the next appropriate tab stop. For example, **8t** moves to the next 8-space tab stop.
r	0	Print a space.
n	0	Print a newline.
"..."	0	Print the enclosed string.
^		*Dot* is decremented by the current increment; nothing is printed.
+		*Dot* is incremented by 1; nothing is printed.
–		*Dot* is decremented by 1; nothing is printed.
c	1	Print the value as an ASCII character. Control characters are printed as ^X and the delete character is printed as ^?.
s	*n*	Print a string of characters (terminated by a null byte).

newline

Repeat the previous command with a *count* of 1.

[?/]w *value* ...

Write a 1-word *value* into the addressed locations. If the command is W, the address is treated as untessellated. If the address expression is 4-way, the value is written to each of the four components. Multiple values are written into consecutive locations. If a *count* is specified, the write command is repeated *count* times with *dot* incremented each time (useful for clearing a block of scratchpad).

>*name*

Dot is assigned to the variable or register named. If a 4-way register is specified and *dot* is a scalar expression, its value is assigned to each component of the register. Assigning a vector expression to a variable causes it to be treated later as a vector expression.

$*modifier*

Miscellaneous printing commands. The available *modifiers* are:

<*f* Read commands from the file *f*. If this command is executed in a file, further commands in the file are not seen. If *f* is omitted, the current input stream is terminated. If a *count* is given, and is zero, the command will be ignored. The value of count will be placed in variable *9* before the first command in *f* is executed.

<<*f* Similar to < except it can be used in a file of commands without causing the file to be closed. Variable *9* is saved during the execution of this command and restored when it

completes. There is a (small) finite limit to the number of << files that can be open at once.

>f Append output to the file *f*, which is created if it does not exist. If *f* is omitted, output is returned to the terminal.

a Print the scratchpad address registers. If *count* is specified, only the first *count* registers are displayed.

b Print all breakpoints and their associated counts and commands.

c Print a stack backtrace. The backtrace shows the value of the **pc**, **lc**, and **runflag** at each level in the stack. If *count* is given, only the first *count* frames are printed. If *address* is specified, the backtrace commences at that stack level.

d Set the default radix to *address* and report the new value. Note that *address* is interpreted in the (old) current radix. If no radix is specified, *charm* reports the current radix.

e Print the names and values of external symbols. If an *address* is specified, it is interpreted as a symbol table type; the possible values are: 2 (*absolute* symbols), 4 (*text* symbols), 6 (*data* symbols), and 8 (*bss* symbols).

l Print the names and values of local symbols. Any *address* specified is interpreted as for **e**.

m Print the segment (load) map.

p Print the contents of the Pbus registers and the Pbus data buffer. If *count* is given, only the first *count* entries in the Pbus data buffer are displayed.

q Exit from *charm* ($Q and ^D work as well).

r Print the registers of each ALU, the loop counter, the stack pointer, and the instruction addressed by the pc. *Dot* is set to **pc**. If *address* is specified, it is interpreted as a bitmask of processors for which ALU registers should be displayed. If *count* is given, only the first *count* ALU registers are displayed.

u Print the name of each unresolved symbol and the modules in which the symbols are referenced.

v Print all non-zero variables in hexadecimal.

x Print the contents of the crossbar.

y Print the contents of the Yapbus registers.

S Set the limit for symbol matches to *address* (default 255).

W Set the page width for output to *address* (default 80).

:*modifier*

Manage the execution of the Chap. The available *modifiers* are:

b*c* Set a breakpoint at *address*. The breakpoint is executed *count*–1 times before causing a stop. Each time the breakpoint is encountered, the command *c* is executed. If this command is omitted or sets *dot* to zero, the breakpoint causes a stop.

d Delete the breakpoint at *address*.

c The Chap is continued. If *address* is given, then the processor is continued at this address. Breakpoint skipping is the same as for **r**.

f*c* Specify a set of commands *c* to be executed each time the Chap is stopped by a **:** command. More explicitly, the "format" commands are executed after each single-step, next, run, continue, or halt command. If a command string is not specified, the current one is displayed.

h Halt the Chap.

l*f* Load and bind a *chas* output file *f*. *Charm* will try to resolve any undefined external references in *f* from code currently resident in the Chap. Failure to resolve references is reported on the standard output. The file is searched for in the list of directories shown with the **:p** command. If a file is not specified, the last file specified in a **:u** or **:l** command is used. If no "last file" is available, *charm* tries to load the file "chap.out", or, failing, "chas.out".

n As for **:s** except that if the current instruction contains a jump to subroutine sequencer instruction, the subroutine is run at full speed with the Chap halted at the instruction immediately following the return. If the Chap had not previously been started with an **r** command, the **n** command will do this.

p*p* Set the "load searchpath" to *p*. The path is a list of directories to search for loadable files. Searchpaths must be separated by colons. If no path is specified, the current load path is displayed. The default load path is ".:/u1/gfx/lib:/u1/gfx/pixar/chap/lib".

r Begin execution of the Chap. If *address* is given explicitly, then the program is entered at this point; otherwise the program is entered at its standard entry point. *Count* specifies how many breakpoints are to be ignored before stopping.

s As for **:c** except that the Chap is single stepped *count* instructions. If the Chap had not been previously started with an **r** command, the **s** command will do this.

u*f* Unload the file *f*. That is, reclaim the instruction and scratchpad memory associated with file and remove the related allocation information from the symbol table. When no file is specified, *charm* searches for a file as described under the **:l** command.

3.3. Variables

Charm provides a number of variables. Certain named variables are set initially by *charm* and used in the print commands (see below). Numbered variables are used to communicate various dynamically-changing values.

0 The last value printed.
1 The last immediate field of an instruction.
2 The previous value of variable 1.
9 The count on the last **$<** or **$<<** command.
a Number of registers to print with the **$a** command.
f "Runflag" to use in limiting printing with the **$r** command.
p Number of data buffer entries to print with the **$p** command.
r Number of registers to print with the **$r** command.
s Number of registers to print with the **$s** command.

3.4. Registers

Charm allows Chap data registers to be referenced symbolically. Register names are identical to those used by the Chap assembler *chas* wherever possible. A component of a vector register may be specified with [*exp*], where *exp* is an expression as described in §3.1. A sysbus register is specified, as in *chas*, ssysbus<*exp*> where, once again, *exp* is an expression. The following list shows the names of registers as understood by *charm.*

a0, a1	Pbus address registers	
acc	ALU accumulator	4-way
admux	address portion of the crossbar	4-way
b0, ..., b15	Scratchpad base address registers	
i0,..., i15	Scratchpad index registers	
lc	Loop counter	
lsp	Least signifcant part of multiplier output	4-way
msp	Most significant part of multiplier output	4-way
multx	Multiplier X-input	4-way
multy	Multiplier Y-input	4-way

pc	Program counter	
pcsr	Pbus control status register	
r0, ..., r31	ALU internal registers	4-way
rdmux	read portion of the crossbar	4-way
rf	Runflag	
sp	Stack pointer	
sysbus	Sysbus shared data register	
status	Chap status register	
wrmux	write portion of the crossbar	4-way
ycsr	Yapbus control status register	

The Format of Stored Pictures

ABSTRACT

This memo presents the Pixar picture storage standard. Pictures can be of arbitrary resolution, with either dumping or run length encoding of any subset of RGBA channels. Pictures are tiled, allowing for efficient recall of subwindows. 8 and 12 bits per channel are handled.

1. The Picture File

Note: *This paper describes the file format for storing pictures used by routines in Pixar Software Release 1.0. Routines like* gt, sv, and gtinfo use this format, hiding the storage details from the user. This material is recommended only for those who wish to write their own storage and retrieval routines or device drivers. — Pixar Documentation

The standard for picture storage at Pixar is intended to accommodate several different picture formats: multiple channels; different numbers of bits per channel; encoded and dumped format; arbitrary picture size. Large pictures can be handled by breaking the picture into smaller uniform rectangular pieces called *tiles.* Thus, a 1024 x 512 picture could be broken up into eight 256 x 256 tiles. All pictures are stored with a 512 byte picture header describing the picture format, followed by pointers to the start of each encoded tile, followed by the pixel data of each tile.

We normally save encoded, one-tile, 1024 x 768 pictures, stored in 8K (8192) byte blocks. The pixel aspect ratio 1:1.

1.1. Picture Headers

byte number	# bytes	name	description
000	4	magic number	0x0000E880
004	2	version number	0
006	246	label	Ascii description
252	4	labelptr	ptr to label continuation
416	2	picture height	pixel height of full picture
418	2	picture width	pixel width of full picture
420	2	tile height	pixel height of each tile
422	2	tile width	pixel width of each tile
424	2	picture format	four bits designating RGBA
426	2	picture storage	encoding and number of bits
428	2	blocking factor	
430	2	alpha mode	matted-to-black:0 ; unassociated:1
432	2	x offset	horizontal offset for picture
434	2	y offset	vertical offset for picture
448	4	unused	
452	28	unused	
512	8*n	tile pointer table	4-byte pointer, 4-byte length

Numbers preceded by Ox... are hexadecimal; all other numbers are decimal. Multiple-byte data are stored with the least significant 8 bits in the first byte. The first 4 bytes of each picture are (0x80, OxE8, 0x00, 0x00).

1.2. Label

The picture label is read as an ASCII description by various picture handling programs. There are no rules about what can be stored in the picture label, except that null and control characters may interfere with the printing of the label. The first 246 characters are stored in the picture header, but labels can be arbitrarily long. A label pointer stored in the header points to any continuation. Further label information may be allocated in chunks the size of the blocking factor, with the last four bytes of that chunk reserved for a link to further blocks.

Picture height and picture width must be positive numbers. Tile height and tile width must be positive numbers not greater than picture height and width, respectively. Normally, a picture has one tile, so that the picture and tile dimensions are the same. It is often useful to choose tile dimensions that are evenly divisible into picture dimensions. It becomes much easier, for example, to retrieve a 512x512 window from a 4096x4096 encoded picture.

Tile dimensions do not have to evenly divide picture dimensions. Tile 0 is always understood to be in the upper left corner of the picture; pixel [0,0] of tile 0 is pixel [0,0] of the picture. Tiles can extend down or to the right beyond the boundaries of the picture. Pixels beyond the picture boundaries yet inside border tiles must be properly encoded in the tiles, yet are undefined with regard to the picture.

1.3. Picture Format

Pictures can include any subset of RGBA channels. Full RGBA pictures are the most common; RGB backgrounds are also popular. Single channel R pictures are currently recovered as black-and-white (single channel RGB) pictures. RGBA channels correspond to bits 3210, so that an RGB picture has a format of 1110 binary.

1.4. Picture Storage

Four picture storage modes are supported: 8 bit encoded (0); 12 bit encoded (1); 8 bit dumped (2); 12 bit dumped (3). 12 bit data is simply stored in two bytes. 11 bit data is understood to cover the interval (-.5, 1.5), where 3072 is -0.5, 0 is 0.0, 2048 is 1.0, and 3071 is almost 1.5.

1.4.1. Encoded Tiles

If the tiles are to be encoded (see byte 426 of the header), the pixel information is broken into packets, each headed by a flag and count. No packet may span multiple scan lines. **However, each scan line of an encoded picture may consist of any combination of the four types of packets listed in the table below.** The flags are listed with corresponding data for four, three, and one channel files:

flag	count	RGBA	RGB	R	comment
0					end of disk block
1	c	RGBA RGBA ...	RGB RGB ...	R R ...	full channel dump
2	c	λRGBA λRGBA ...	λRGB λRGB ...	λR λR ...	full channel run
3	c	A RGB RGB ...	n/a	n/a	partial channel dump
4	c	A λRGB λRGB ...	n/a	n/a	partial channel run

The flag and count are packed into 16 bits as follows:

first byte	count <0:7>	
second byte	flag <0:4>	count <8:11>

allowing 4 bits for flag and 12 bits for count. *This seems to have been an unfortunate choice. An 8-bit count would have been perfectly sufficient and more efficient.*

When flag equals 1 or 3, the 12-bit count c is one less than the number of dumped pixels in the data. A count c of 0 indicates 1 instance and no repetition. This allows dumps of length 4096 pixels.

When flag equals 2 or 4, the 12-bit count c is one less than the number of run lengths. Each run length is started with an 8-bit λ, which indicates the number of repeated pixels. Once again, a length λ of 0 indicates 1 instance and no repetition.

Blocking the data speeds disk access. No data packet spans multiple disk blocks, and zeroes should fill out the block. The number of bytes per block is set in byte 428 of the picture header. Furthermore, no packet spans multiple scanlines.

1.4.2. Dumped Tiles

If byte 426 of the picture header lists this as a dumped picture, no excess bytes are used for encoding the data. The tile data is listed as RGBRGBRGB... for RGB pictures and RRRRR... for single channel pictures.

1.5. Blocking Factor

The blocking factor indicates the optimum disk transfer chunk. The only side effect of the blocking factor is that encoded packets do not span adjacent disk blocks.

1.6. Picture Offsets

Pictures are understood to have an xy translation, so that a saved frame buffer window can be restored to the same spot.

1.7. Tile Pointer Table

Each tile has a 4-byte pointer and 4-byte length. Tiles are numbered across from 0 to (numberxtiles*numberytiles–1), where numberxtiles is (1 + (picturewidth–1)/tilewidth), and numberytiles is similar. A pointer of 0 indicates a null tile; a positive pointer and a count of –1 indicates an incomplete tile; otherwise the tile is complete. Using the tile pointers, it is possible to overwrite individual tiles by appending tile pixels to the file and changing the tile pointer information, leaving the old tile data as garbage in the file.

2. Recommendations for simple use

Set up the picture header with the tile size equal to the picture size. This eliminates one level of complexity. Set up the tile pointer table with one 4-byte pointer (at byte 512) to file location 8192. The 4-byte length is not crucial. Use dumped format for the pixel data for further simplification. Set up the header as indicated in the table above, and start dumping the RGBA pixel data at location 8192.

The simple recommended picture header is listed below.

byte number	# bytes	name	description
000	4	magic number	0x0000E880
004	2	version number	0
006	246	label	0
252	4	labelptr	0
416	2	picture height	768
418	2	picture width	1024
420	2	tile height	768
422	2	tile width	1024
424	2	picture format	15
426	2	picture storage	2
428	2	blocking factor	8192
430	2	alpha mode	0
432	2	x offset	0
434	2	y offset	0
448	4	colormap pointer	unused
452	28	colormap filename	unused
512	8*n	tile pointer table	8192,0
8192	?	dumped pixel information	

Chap Pbus Programming

1. Introduction

The Pbus is the Pixar bus between a Chap and picture memory. The library routines

```
FxSCopy(pw,scanlineptr,count,x,y)
FySCopy(pw,scanlineptr,count,x,y)
```

are provided for reading from picture memory to a scratchpad scanline buffer; the routines

```
SFxCopy(pw,scanlineptr,count,x,y)
SFyCopy(pw,scanlineptr,count,x,y)
```

are provided for writing from a scratchpad scanline buffer to picture memory. The above routines are optimized for scanline access to the memory. This memo explains how to access the Pbus directly.

NOTE: *The numbers published here are based on an 85 ns clock cycle. Use these numbers for comparison only, as the clock cycle time on your Pixar may be different.*

The picture memory is a linear array of 32x32 pixel *tiles*.. The standard Pixar has 4096 tiles, sufficient for storing 4 million pixels. By software convention a *tile block* is a contiguous linear array of tiles understood to be arranged in a rectangle (see *TB*(3C)†). Again by software convention, a *pixel window* is a subrectangle of the tile block, understood to be the clipping window through which all picture memory accesses must go (see *PW*(3C)). The library routines for Pbus access all utilize this notion of pixel windows in referring to picture memory.

2. Pbus Accesses

Pbus accesses may not straddle tiles, but five different access methods may be used inside a tile (see Figure 1).

Transfer	Read (μs)	Write (μs)
32 in x	2.7 to 4.6	2.1 to 3.9
32 in y	2.7 to 4.6	2.1 to 3.9
16 in x	2.3 to 4.2	1.7 to 3.5
16 in y	2.3 to 4.2	1.7 to 3.5
16 square	2.3 to 4.2	1.7 to 3.5

Figure 1. Picture Memory Access Times.

Naturally, a 32-pixel access spans the tile from pixel 0 through 31. The linear 16-pixel accesses can be made either to pixels 0 through 15 or 16 through 31 of a horizontal or vertical span. The 16-pixel square access can be made only to 64 distinct 4x4 squares of the tile: the x and y tile coordinates of the upper left corner must be on a 4-pixel boundary.

The range in the times of Figure 1 reflect contention with the video for the picture memory. For example, a 32-pixel Pbus read takes as little as 2.7μs with the video off (e.g. during vertical retrace) and as much as 4.6μs with the video on. In general, reads take 600ns longer than writes, and 32-pixel accesses take 400ns longer than 16-pixel accesses.

The timing of the library routines for moving entire scanlines from picture memory to Chap scratchpad is as follows: Reads (`FxSCopy`, `FySCopy`) take approximately 20μs plus 4.5μs per tile access. Writes (`SFxCopy`, `SFyCopy`) take approximately 20μs plus 4.2μs per tile access. Each tile access moves 32 pixels, so the library routines read at a rate of 7.1 million pixels per second and write at a rate of 7.6 million pixels per second.

3. Pbus Buffers

The Pbus interface is double-buffered, allowing for Chap access to one set of 32 pixels, the *Chapside*, while another set of 32 pixels, the *Memoryside*, is read from (written to) memory.

The Chapside buffer is accessible through a

† References of the form X(Y) mean the manual page named X in the section Y of the Pixar Programmer's Manual pages.

single register, **pbus**, on the Chap Mbus. The bottom 5 bits of **pbus csr** (the **PBUSCSR_ADDR** field below) indicate which pixel in the Chapside buffer may be accessed through the **pbus** register. One bit of the Chap instruction word is dedicated to indicate possible incrementing of the 5-bit pointer. 32-pixel transfers access Pbus buffers in the natural order; 16-pixel transfers access the first 16 pixels of the Pbus buffer; 16-pixel square transfers fill the first 16 pixels of the buffer by rows.

The pixels in the Chapside buffer are written only by an explicit write into the **pbus** register; the pixels in the Memoryside buffer are written only by an explicit memory read initiated by setting **pbus csr** appropriately; otherwise, the data in the buffer is stable.

4. Pbus Registers

The **pbus csr**, **pbus a0**, and **pbus a1** registers are all accessible on the Chap scalar bus. As described in the Chap Instruction Description, the **pbus csr** contains the fields shown in Figure 2.

Bits	Name	Meaning
0-4	PBUSCSR_ADDR	Buffer address
5	PBUSCSR_PP	Ping/Pong
8	PBUSCSR_RED	Red enable
9	PBUSCSR_GREEN	Green enable
10	PBUSCSR_BLUE	Blue enable
11	PBUSCSR_ALPHA	Alpha enable
14	PBUSCSR_RW	Write/Read
15	PBUSCSR_GO	Go/Done

Figure 2. Pbus Control Status Register.

(Bits 6-7 and 12-13 are reserved.) All bits are high-true. The Go/Done bit is set to 1 to initiate an operation (Go) and cleared by the hardware when the operation completes (Done). The Write/Read bit must be set to 1 for a write operation and 0 for a read operation (beware of this, it is backwards to the Read/Write bit in the **pbus a0** register). The field names listed above are defined in <pixar/pbusreg.h>.

The **pbus csr** register is accessible at any time; **pbus a0** and **pbus a1** should only be accessed when a transfer is not taking place. Transfers are not taking place when the PBUSCSR_GO bit is off. The **pbus busy** sequencer condition may be used to check for a transfer in progress.

The **pbus a0** register contains the fields shown in Figure 3.

Bits	Name	Meaning
0-4	PBUSA0_XADDR	X address
5-9	PBUSA0_YADDR	Y address
10	PBUSA0_XACC	Access in X
11	PBUSA0_YACC	Access in Y
12	PBUSA0_RW	Read/Write
14	PBUSA0_1632	Access 16/32

Figure 3. Pbus Address Register 0.

(Bits 13 and 15 are reserved.) All bits are high-true. The Read/Write bit must be set to 1 for a read access and 0 for a write.* The field names listed above are defined in <pixar/pbusreg.h>.

The **pbus a1** register holds the 16-bit tile number of the tile in picture memory to be accessed. Notice that the 16-bit width of the register limits the system to 64K tiles, or 64M (8K x 8K) pixels.

4.1. Waiting for Completion of Pbus Transfer

The program must wait for the completion of one transfer before initiating another. The following code (referred to below as **PBUS_WAIT**) handles the problem:

```
push;
    dowhile pbus busy;
```

4.2. Toggling the Pbus Buffers

Reading from memory leaves the pixels in the Memoryside buffer. The program must toggle the buffers before the Chap can access the pixels. Similarly, writing to memory takes pixels from the Memoryside buffer, so the program must have toggled the buffers to move the pixels there. The following simple code, **PBUS_TOGGLE**, handles the toggling:

```
acc = pbus csr;
pbus csr = acc ^ PBUSCSR_PP;
```

Note that you should never toggle the buffers while a Pbus transfer is in progress.

* Note that **pbus csr** has a Write/Read bit and **pbus a0** has a Read/Write bit. These must be in agreement: both must indicate read or both must indicate write.

4.3. Initiating a Single Read From Memory

The following steps are necessary:

1) **PBUS_WAIT.**
2) Set **pbus a1** to the proper tile.
3) Set **pbus a0** to address the upper left corner of the desired pixels within the tile. Also set **pbus a0** to indicate a 16 or 32 pixel access, read or write, and X, Y or square access. The latter is indicated by setting both the **PBUSA0_XACC** and **PBUSA0_YACC** bits. For example, to read the first 16 pixels of horizontal scanline 7 of a tile,

```
pbus a0 = PBUSA0_1632|PBUSA0_RW|
     PBUSA0_XACC|7<<5;
```

4) Set **pbus csr** according to

```
acc = pbus csr;
acc = acc & ~PBUSCSR_RW;
pbus csr = acc | PBUSCSR_GO;
```

in order to turn off the Write bit and turn on the Go bit. This code sequence is referred to below as **PBUS_READ**.

5) **PBUS_WAIT.**
6) **PBUS_TOGGLE.**
7) The pixels are now available in the Chapside Pbus buffer. The **pbus csr** can be used to set the Pbus buffer pointer to something other than the first pixel.

4.4. Initiating a Single Write To Memory

Assume that the pixels to be written are now settled into the Chapside Pbus buffer.

1) **PBUS_WAIT.**
2) **PBUS_TOGGLE.**
3) Set **pbus a1** to the proper tile.
4) Set **pbus a0** to address the upper left corner of the desired pixels within the tile. Also set **pbus a0** to 16 or 32 pixel access, read or write, and X, Y or square access. For example, to write the lower right 16 pixel square,

```
pbus a0 = PBUSA0_1632|PBUSA0_XACC|
     PBUSA0_YACC|28<<5|28;
```

5) Set **pbus csr** according to

```
acc = pbus csr;
pbus csr = PBUSCSR_RW|PBUSCSR_GO|acc;
```

in order to turn on the Write bit and the Go bit. This code sequence is referred to below as **PBUS_WRITE**.

6) **PBUS_WAIT**. (This extra step is not needed, but the transfer is not officially complete until the **PBUSCSR_GO** bit is turned off.)

4.5. Handling Overlapping Reads From Memory

1) **PBUS_WAIT**. Make sure that the Pbus is available.
2) Set up **pbus a0** and **pbus a1** for the first read.
3) **PBUS_READ**. Initiate the first read from memory.
4) **PBUS_WAIT**. Wait for the first read to finish.
5) Update **pbus a0** and **pbus a1** for a second read.
6) **PBUS_TOGGLE**. Bring the first buffer of pixels onto the Chapside.
7) **PBUS_READ**. Initiate the second read from memory.
8) While the second access takes place, the Chap can process the first buffer of pixels sitting in the Chapside Pbus buffer.
9) **PBUS_WAIT**. Wait for the second read to finish.
10) Update **pbus a0** and **pbus a1** for a third read.
11) **PBUS_TOGGLE**. Bring the second buffer of pixels onto the Chapside.
12) **PBUS_READ**. Initiate a third read from memory.

This process may continue forever. Note that it takes approximately as long to move pixels from Chapside Pbus buffer to scratchpad (32 ticks = 32 x 85ns = 2.72μs) as it does to move pixels from Picture Memory to Memoryside Pbus buffer (2.7 to 4.6μs).

For optimum speed, the toggle and read may be done in the same instruction by xor-ing **PBUSCSR_GO** and **PBUSCSR_PP** in one operation.

4.6. Handling Overlapping Writes To Memory

1) The Chap can load the first buffer of pixels into the Chapside Pbus buffer.
2) **PBUS_WAIT**. Make sure that the Pbus is available.

3) Set up **pbus a0** and **pbus a1** for the first write.
4) **PBUS_TOGGLE**. Move the first buffer of pixels over to the Memoryside.
5) **PBUS_WRITE**. Initiate the first write to memory.
6) The Chap can load the second buffer of pixels into the Chapside Pbus buffer.
7) **PBUS_WAIT**. Wait for that first transfer to complete.
8) Update **pbus a0** and **pbus a1** for the second write.
9) **PBUS_TOGGLE**. Move the second buffer of pixels over to the Memoryside.
10) **PBUS_WRITE**. Initiate the second write to memory.

This process may continue forever. Note that it takes approximately as long to move pixels from scratchpad to Chapside Pbus buffer (32 ticks = 32 x 85ns = 2.72μs) as it does to move pixels from Memoryside Pbus buffer to Picture Memory (2.3 to 4.2 μs).

For optimum speed, the toggle and write may be done in the same instruction by xor-ing **PBUSCSR_GO** and **PBUSCSR_PP** in one operation.

5. Concluding Comments

The macros **PBUS_WAIT**, **PBUS_TOGGLE**, **PBUS_READ**, and **PBUS_WRITE** are defined in the <pixar/pbus.h> which automatically includes the file <pixar/pbusreg.h>.

Compositing Digital Images

PIXAR

ABSTRACT

Most computer graphics pictures have been computed all at once, so that the rendering program takes care of all computations relating to the overlap of objects. There are several applications, however, in which elements must be rendered separately, relying on compositing techniques for the anti-aliased accumulation of the full image. This paper presents the case for four-channel pictures, demonstrating that a matte component can be computed along with the color channels. The paper discusses guidelines for the generation of elements and the arithmetic for arbitrary composition.

1. Introduction

Increasingly, we find that a complex three dimensional scene cannot be fully rendered by a single program. The wealth of literature on rendering polygons and curved surfaces, handling the special cases of fractals, spheres, quadrics and triangles, implementing refinements for texture mapping and bump mapping, noting speed-ups on the basis of coherence or depth complexity in the scene, suggests that multiple programs will become more and more common.

In fact, reliance on a single program for rendering an entire scene is a poor strategy for minimizing the cost of small modeling errors. Experience has taught us to break down large bodies of source code into separate modules in order to save compilation time. An error in one routine forces only the recompilation of its module and the relatively quick reloading of the entire program. Similarly, small errors in coloration or design in one object should not force the ''recompilation'' of an entire image.

Separating the image into *elements* that can be independently rendered saves enormous time. Each element has an associated *matte,* providing coverage information for each pixel. The *compositing* of those elements makes use of the mattes to accumulate the final image.

The compositing methodology must not induce aliasing; soft edges of the elements must be honored in computing the final image. Features should be provided to exploit the full associativity of the compositing process; this affords flexibility, for example, for the accumulation of several foreground elements into an aggregate foreground which can be examined over different backgrounds. The compositor should provide facilities for arbitrary dissolves and fades of elements during an animated sequence.

Several highly successful rendering algorithms have worked by reducing their environments to pieces that can be combined in a 2 1/2 dimensional manner, and then overlaying them either front-to-back or back-to-front [3]. Whitted and Weimar's graphics test-bed [6] and Crow's image generation environment [2] are both designed to deal with heterogenously rendered elements. Whitted and Weimar's system reduces all objects to horizontal spans which are composited using a Warnock-like algorithm. In Crow's system a supervisory process decides the order in which to combine images created by independent special-purpose rendering processes. The imaging system of

Warnock and Wyatt [5] incorporates 1-bit mattes. The Hanna-Barbera cartoon animation system [4] incorporates soft-edge mattes, representing the opacity information in a less convenient manner than that proposed here. The present paper presents guidelines for rendering elements and introduces the algebra for compositing.

2. The Alpha Channel

A separate component is needed to retain the matte information, the extent of coverage of an element at a pixel. In a full color rendering of an element, the RGB components retain only the color. To place the element over an arbitrary background, a mixing factor is required at every pixel to control the linear interpolation of foreground and background colors. In general, there is no way to encode this component as part of the color information. For anti-aliasing purposes, this mixing factor needs to be of comparable resolution to the color channels. Let us call this an *alpha* channel, and let us treat an alpha of 0 to indicate no coverage, 1 to mean full coverage, with fractions corresponding to partial coverage.

In an environment where the compositing of elements is required, we see the need for an alpha channel as an integral part of all pictures. Because mattes are naturally computed along with the picture, a separate alpha component in the frame buffer is appropriate. Off-line storage of alpha information along with color works conveniently into run-length encoding schemes because the alpha information tends to abide by the same runs.

What is the meaning of the quadruple (r,g,b,α) at a pixel? How do we express that a pixel is half covered by a full red object? One obvious suggestion is to assign (1,0,0,.5) to that pixel: the .5 indicates the coverage and the (1,0,0) is the color. There are a few reasons to dismiss this proposal, the most severe being that all compositing operations will involve multiplying the 1 in the red channel by the .5 in the alpha channel to compute the red contribution of this object at this pixel. The desire to avoid this multiplication points up a better solution, storing the *pre-multiplied* value in the color component, so that (.5,0,0,.5) will indicate a full red object half covering a pixel.

The quadruple (r,g,b,α) indicates that the pixel is α covered by the color (r/α, g/α, b/α). A quadruple where the alpha component is less than a color component indicates a color outside the [0,1] interval, which is somewhat unusual. We will see later that luminescent objects can be usefully represented in this way. For the representation of normal objects, an alpha of 0 at a pixel generally forces the color components to be 0. Thus the RGB channels record the true colors where alpha is 1, linearly darkened colors for fractional alphas along edges, and black where alpha is 0. Silhouette edges of RGBA elements thus exhibit their anti-aliased nature when viewed on an RGB monitor.

It is important to distinguish between two key pixel representations:

black = (0,0,0,1);
clear = (0,0,0,0).

The former pixel is an opaque black; the latter pixel is transparent.

3. RGBA Pictures

If we survey the variety of elements which contribute to a complex animation, we find many complete background images with an alpha of 1 everywhere. Among foreground elements, we find that the color components roll off in step with the alpha channel, leaving large areas of transparency. Mattes, colorless stencils used for controlling the compositing of other elements, have 0 in their RGB components. Off-line storage of RGBA pictures should therefore provide the natural data compression for handling the RGB pixels of backgrounds, RGBA pixels of foregrounds, and A pixels of mattes.

There are some objections to computing with these RGBA pictures. Storage of the color components pre-multiplied by the alpha would seem to unduly quantize the color resolution, especially as alpha approaches 0. However, because any compositing of the picture will require that multiplication anyway, storage of the product forces only a very minor loss of precision in this regard. Color extraction, to compute in a different color space for example, becomes more difficult. We must recover (r/α, g/α, b/α), and once again, as alpha approaches 0, the precision falls off sharply. For our applications, this has yet to affect us.

4. The Algebra of Compositing

Given this standard of RGBA pictures, let us examine how compositing works. We shall do this by enumerating the complete set of binary compositing operations. For each of these, we shall present a formula for computing the contribution of each of two input pictures to the output composite at each pixel. We shall pay particular attention to the output pixels, to see that they remain pre-multiplied by their alphas.

4.1. Assumptions

When blending pictures together, we do not have information about overlap of coverage information within a pixel; all we have is an alpha value. When we consider the mixing of two pictures at a pixel, we must make some assumption about the interplay of the two alpha values. In order to examine that interplay, let us first consider the overlap of two semi-transparent elements like haze, then consider the overlap of two opaque, hard-edged elements.

If α_A and α_B represent the opaqueness of semi-transparent objects which fully cover the pixel, the computation is well known. Each object lets $(1-\alpha)$ of the background through, so that the background shows through only $(1-\alpha_A)(1-\alpha_B)$ of the pixel. $\alpha_A(1-\alpha_B)$ of the background is blocked by object A and passed by object B; $(1-\alpha_A)\alpha_B$ of the background is passed by A and blocked by B. This leaves $\alpha_A \alpha_B$ of the pixel, which we can consider to be blocked by both.

If α_A and α_B represent subpixel areas covered by opaque geometric objects, the overlap of objects within the pixel is quite arbitrary. We know that object A divides the pixel into two subpixel areas of ratio $\alpha_A : 1-\alpha_A$. We know that object B divides the pixel into two subpixel areas of ratio $\alpha_B : 1-\alpha_B$. Lacking further information, we make the following assumption: *there is nothing special about the shape of the pixel; we expect that object B will divide each of the subpixel areas inside and outside of object A into the same ratio* $\alpha_B : 1-\alpha_B$. The result of the assumption is the same arithmetic as with semi-transparent objects and is summarized in the following table:

description	*area*
$\bar{A} \cap \bar{B}$	$(1-\alpha_A)(1-\alpha_B)$
$A \cap \bar{B}$	$\alpha_A(1-\alpha_B)$
$\bar{A} \cap B$	$(1-\alpha_A)\alpha_B$
$A \cap B$	$\alpha_A \alpha_B$

The assumption is quite good for most mattes, though it can be improved if we know that the coverage seldom overlaps (adjacent segments of a continuous line) or always overlaps (repeated application of a picture). For ease in presentation throughout this paper, let us make this assumption and consider the alpha values as representing subpixel coverage of opaque objects.

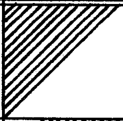
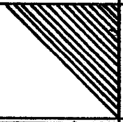
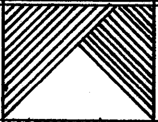
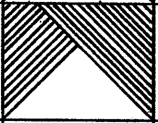
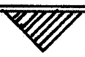
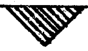
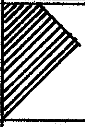
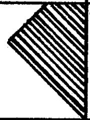
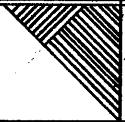
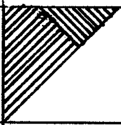
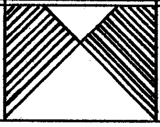
4.2. Compositing Operators

Consider two pictures A and B. They divide each pixel into the 4 subpixel areas

B	A	*name*	*description*	*choices*
0	0	0	$\bar{A} \cap \bar{B}$	0
0	1	A	$A \cap \bar{B}$	0, A
1	0	B	$\bar{A} \cap B$	0, B
1	1	AB	$A \cap B$	0, A, B

listed in this table along with the choices in each area for contributing to the composite. In the last area, for example, because both input pictures exist there, either could survive to the composite. Alternatively, the composite could be clear in that area.

A particular binary compositing operation can be identified as a quadruple indicating the input picture that contributes to the composite in each of the four subpixel areas 0, A, B, AB of the table above. With three choices where the pictures intersect, two where only one picture exists and one outside the two pictures, there are 3×2×2×1=12 distinct compositing operations listed in the table below. Note that pictures A and B are diagrammed as covering the pixel with triangular wedges whose overlap conforms to the assumption above. Useful operators include *A* **over** *B*, *A* **in** *B*, and *A* **held out by** *B*. *A* **over** *B* is the placement of foreground A in front of background B. *A* **in** *B* refers only to that part of A inside picture B. *A* **held out by** *B*, normally shortened to *A* **out** *B*, refers only to that part of A outside picture B. For completeness, we include the less useful operators *A* **atop** *B* and *A* **xor** *B*. *A* **atop** *B* is the union of *A* **in** *B* and *B* **out** *A*. Thus, *paper* **atop** *table* includes *paper* where it is on top of *table*, and *table* otherwise; area beyond the edge of the table is out of the picture. *A* **xor** *B* is the union of *A* **out** *B* and *B* **out** *A*.

operation	quadruple	diagram	F_A	F_B
clear	(0,0,0,0)		0	0
A	(0,A,0,A)		1	0
B	(0,0,B,B)		0	1
A **over** *B*	(0,A,B,A)		1	$1-\alpha_A$
B **over** *A*	(0,A,B,B)		$1-\alpha_B$	1
A **in** *B*	(0,0,0,A)		α_B	0
B **in** *A*	(0,0,0,B)		0	α_A
A **out** *B*	(0,A,0,0)		$1-\alpha_B$	0
B **out** *A*	(0,0,B,0)		0	$1-\alpha_A$
A **atop** *B*	(0,0,B,A)		α_B	$1-\alpha_A$
B **atop** *A*	(0,A,0,B)		$1-\alpha_B$	α_A
A **xor** *B*	(0,A,B,0)		$1-\alpha_B$	$1-\alpha_A$

4.3. Compositing Arithmetic

For each of the compositing operations, we would like to compute the contribution of each input picture at each pixel. This is quite easily solved by recognizing that each input picture survives in the composite pixel only within its own matte. For each input picture, we are looking for that fraction of its own matte that prevails in the output. By definition then, the alpha value of the composite, the total area of the pixel covered, can be computed by adding α_A times its fraction F_A to α_B times its fraction F_B.

The color of the composite can be computed on a component basis by adding the color of the picture A times its fraction to the color of picture B times its fraction. To see this, let c_A, c_B, and c_O be some color component of pictures A, B and the composite, and let C_A, C_B, and C_O be the true color component before pre-multiplication by alpha. Then we have

$$c_O = \alpha_O C_O$$

Now C_O can be computed by averaging contributions made by C_A and C_B, so

$$c_O = \alpha_O \frac{\alpha_A F_A C_A + \alpha_B F_B C_B}{\alpha_A F_A + \alpha_B F_B}$$

but the denominator is just α_O, so

$$\begin{aligned} c_O &= \alpha_A F_A C_A + \alpha_B F_B C_B \\ &= \alpha_A F_A \frac{c_A}{\alpha_A} + \alpha_B F_B \frac{c_B}{\alpha_B} \\ &= c_A F_A + c_B F_B \qquad (1) \end{aligned}$$

Because each of the input colors is pre-multiplied by its alpha, and we are adding contributions from non-overlapping areas, the sum will be effectively pre-multiplied by the alpha value of the composite just computed. The pleasant result that the color channels are handled with the same computation as alpha can be traced back to our decision to store pre-multiplied RGBA quadruples. Thus the problem is reduced to finding a table of fractions F_A and F_B indicating the extent of contribution of A and B, plugging these values into equation 1 for both the color and the alpha components.

By our assumptions above, the fractions are quickly determined by examining the pixel diagram included in the table of operations. Those fractions are listed in the F_A and F_B columns of the table. For example, in the *A* **over** *B* case, picture A survives everywhere while picture B survives only outside picture A, so the corresponding fractions are 1 and $(1-\alpha_A)$. Substituting into equation 1, we find

$$c_O = c_A \times 1 + c_B \times (1-\alpha_A).$$

This is almost the well used linear interpolation of foreground F with background B

$$B' = F \times \alpha + B \times (1-\alpha),$$

except that our foreground is pre-multiplied by alpha.

4.4. Unary operators

To assist us in dissolving and in balancing color brightness of elements contributing to a composite, it is useful to introduce a darken factor, ϕ, and a dissolve factor, δ:

$$\textbf{darken}(A,\phi) \equiv (\phi r_A, \phi g_A, \phi b_A, \alpha_A)$$

$$\mathbf{dissolve}(A,\delta)\equiv(\delta r_A,\delta g_A,\delta b_A,\delta\alpha_A)\ .$$

Normally, $0\le\phi,\delta\le1$, although none of the theory requires it.

As ϕ varies from 1 to 0, the element will change from normal to complete blackness. If $\phi>1$, the element will be brightened. As δ goes from 1 to 0, the element will gradually fade from view.

Luminescent objects, which add color information without obscuring the background, can be handled with the introduction of an opaqueness factor ω, $0\le\omega\le1$:

$$\mathbf{opaque}(A,\omega)\equiv(r_A,g_A,b_A,\omega\alpha_A)\ .$$

As ω varies from 1 to 0, the element will change from normal coverage over the background to no obscuration. This scaling of the alpha channel alone will cause pixel quadruples where α is less than a color component, indicating a representation of a color outside of the normal range. This possibility forces us to clip the output composite to the [0,1] range.

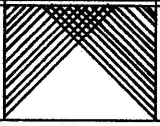
An ω of 0 will produce quadruples $(r,g,b,0)$ which do have meaning. The color channels, pre-multiplied by the original alpha, can be plugged into equation 1 as always. The alpha channel of 0 indicates that this pixel will obscure nothing. In terms of our methodology for examining subpixel areas, we should understand that using the **opaque** operator corresponds to shrinking the matte coverage with regard to the color coverage.

4.5. The PLUS operator

We find it useful to include one further binary compositing operator, **plus** . The expression *A* **plus** *B* holds no notion of precedence in any area covered by both pictures; the components are simply added. This allows us to dissolve from one picture to another by specifying

$$\mathbf{dissolve}(A,\alpha)\ \mathbf{plus}\ \mathbf{dissolve}(B,1-\alpha).$$

In terms of the binary operators above, **plus** allows both pictures to survive in the subpixel area AB. The operator table above should be appended:

operation	quadruple	diagram	F_A	F_B
A **plus** *B*	(0,A,B,AB)		1	1

5. Examples

The operations on one and two pictures are presented as a basis for handling compositing expressions involving several pictures. A normal case involving three pictures is the compositing of a foreground picture A over a background picture B, with regard to an independent matte C. The expression for this compositing operation is

$$(A\ \mathbf{in}\ C)\ \mathbf{over}\ B.$$

Using equation 1 twice, we find that the composite in this case is computed at each pixel by

$$c_O = c_A \alpha_C + c_B (1 - \alpha_A \alpha_C).$$

As an example of a complex compositing expression, let us consider a subwindow of Rob Cook's picture *Road to Point Reyes* [1]. This still frame was assembled from many elements according to the following rules:

Foreground = *FrgdGrass* **over** *Rock* **over** *Fence*
over *Shadow* **over** *BkgdGrass*;

GlossyRoad = *Puddle* **over** (*PostReflection* **atop**
(*PlantReflection* **atop** *Road*));
Hillside = *Plant* **over** *GlossyRoad* **over** *Hill*;

Background = *Rainbow* **plus** *Darkbow* **over**
Mountains **over** *Sky*;
Pt.Reyes = *Foreground* **over** *Hillside* **over** *Background*.

Figure 1 shows three intermediate composites and the final picture.

A further example demonstrates the problem of *correlated mattes*. In Figure 2, we have a star field background, a planet element, fiery particles behind the planet, and fiery particles in front of the planet. We wish to add the luminous fires, obscure the planet, darkened for proper balance, with the aggregate fire matte, and place that over the star field. An expression for this compositing is

(*FFire* **plus** (*BFire* **out** *Planet*))
over **darken**(*Planet*,.8) **over** *Stars* .

We must remember that our basic assumption about the division of subpixel areas by geometric objects breaks down in the face of input pictures with correlated mattes. When one picture appears twice in a compositing expression, we must take care with our computations of F_A and F_B. Those listed in the table are correct only for uncorrelated pictures.

To solve the problem of correlated mattes, we must extend our methodology to handle n pictures: we must examine all 2^n subareas of the pixel, deciding which of the pictures survives in each area, and adding up all contributions. Multiple instances of a single picture or pictures with correlated mattes are resolved by aligning their pixel coverage. Example 2 can be computed by building a table of survivors (shown below) to accumulate the extent to which each input picture survives in the composite.

6. Conclusion

We have pointed out the need for matte channels in synthetic pictures, suggesting that frame buffer hardware should offer this facility. We have seen the convenience of the RGBA scheme for integrating the matte channel. A language of operators has been presented for conveying a full range of compositing expressions. We have discussed a methodology for deciding compositing questions at the subpixel level, deriving a

FFire	BFire	Planet	Stars	Survivor
			●	Stars
		●		Planet
		●	●	Planet
	●			BFire
	●		●	BFire
	●	●		Planet
	●	●	●	Planet
●				FFire
●			●	FFire
●		●		FFire
●		●	●	FFire
●	●			FFire,BFire
●	●		●	FFire,BFire
●	●	●		FFire
●	●	●	●	FFire

simple equation for handling all composites of two pictures. The methodology is extended to multiple pictures, and the language is embellished to handle darkening, attenuation, and opaqueness.

There are several problems to be resolved in related areas, which are open for future research. We are interested in methods for breaking arbitrary three dimensional scenes into elements separated in depth. Such elements are equivalent to clusters, which have been a subject of discussion since the earliest attempts at hidden surface elimination. We are interested in applying the compositing notions to Z-buffer algorithms, where depth information is retained at each pixel.

7. References

1. Cook, R. Road to Point Reyes. *Computer Graphics* Vol 17, No. 3 (1983), Title Page Picture.

2. Crow, F. C. A More Flexible Image Generation Environment. *Computer Graphics* Vol. 16, No. 3 (1982), pp. 9-18.

3. Newell, M. G., Newell, R. G., and Sancha, T. L.. A Solution to the Hidden Surface Problem, pp. 443-448. *Proceedings of the 1972 ACM National Conference.*

4. Wallace, Bruce. Merging and Transformation of Raster Images for Cartoon Animation. *Computer Graphics* Vol. 15, No. 3 (1981), pp. 253-262.

5. Warnock, John, and Wyatt, Douglas. A Device Independent Graphics Imaging Model for Use with Raster Devices. *Computer Graphics* Vol. 16, No. 3 (1982), pp. 313-319.

6. Whitted, Turner, and Weimer, David. A Software Test-Bed for the Development of 3-D Raster Graphics Systems. *Computer Graphics* Vol. 15, No. 3 (1981), pp. 271-277.

8. Acknowledgments

The use of mattes to control the compositing of pictures is not new. The graphics group at the New York Institute of Technology has been using this for years. NYIT color maps were designed to encode both color and matte information; that idea was extended in the Ampex AVA system for storing mattes with pictures. Credit should be given to Ed Catmull, Alvy Ray Smith, and Ikonas Graphics Systems for the existence of an alpha channel as an integral part of a frame buffer, which has paved the way for the developments presented in this paper.

The graphics group at Pixar should be credited with providing a fine test bed for working out these ideas. Furthermore, certain ideas incorporated as part of this work have their origins as idle comments within this group. Thanks are also given to Rodney Stock for comments on an early draft which forced the authors to clarify the major assumptions.

PIXAR
CUSTOMER FEEDBACK FORM

For Pixar use only:

Action ______________

____

____

Date ______________

Pixar serial number ____________________

Comment on:

Hardware __________

Software __________

Documentation ______

Company ______________________________

Address ______________________________

____

____

Person to contact ______________________

Phone number ________________________

Date _______________

Please describe problem:

____

____

____

____

____

____

____

Use this form to report bugs, documentation errors or suggestions. Mail to:

PIXAR
Box 13719
San Rafael
CA 94913