Contents

Figures and Tables

1 The Pixel Machine System Architecture

Introduction

DEVtools is the software development toolkit for the Pixel Machine. DEVtools differs from the other Pixel Machine libraries (PIClib, RAYlib, etc.) in that DEVtools users program both the host and the processors in the Pixel Machine. Users of the other libraries, however, program only the host system—the Pixel Machine functions are performed implicitly by the libraries supplied with the Pixel Machine.

DEVtools enables users to implement a wide variety of applications that take advantage of the graphics and compute power of the Pixel Machine. Consequently, DEVtools users require a deeper understanding of the Pixel Machine architecture and the DSP32 processor than do users of the other libraries.

DEVtools is designed to provide a high level programming model for the Pixel Machine. This model enables users to quickly develop Pixel Machine applications without having to know or understand the details of the inner workings of the Pixel Machine. However, DEVtools does supply the detailed information for those users with special needs that may require lower level access to the Pixel Machine hardware.

DEVtools comprises the following:

- DSP32 C compiler, assembler, linker, library and miscellaneous other utilities
- a library of host functions that control and communicate with the Pixel Machine
- a library of Pixel Machine functions that are used to program the pipe and pixel nodes

Components of a Typical DEVtools Application

A typical DEVtools application consists of a host program, pipe node programs and a pixel node program.

The host program serves as the controller or master of the application. The application is initiated by invoking the host program in the same manner as any other host program. The host program, through the use of DEVtools function calls, loads the Pixel Machine executable files into the pipe and pixel nodes and initiates execution. Once execution has begun the host is responsible for sending data and commands to the Pixel Machine, and for servicing message requests for the Pixel Machine to perform operations such as input/output.

Pipe node programs are used to perform transformations on the data produced by the host before the data is sent to the pixel nodes. Many DEVtools applications do not require use of the pipe. The Pixel Machine can be configured without a pipe for users with no need for pipeline processing. When an application does not use the pipe but is run on a system equipped with a pipe, a program must be loaded into the pipe that passes the data through the pipeline to the pixel nodes. DEVtools includes a pipe program that performs this function. Applications that do make use of the pipe can load a different pipe program into each pipe node or they can load the same program into every node. DEVtools includes functions to read and write command information, send messages to the host system, control access to the pixel broadcast bus, and to send data to the host feedback FIFO.

Pixel node programs are typically the core of the application. The same program is usually loaded into all pixel nodes, although this does not have to be the case. Pixel node programs read commands from either the host or pipe, process the command, and produce results in the distributed frame buffer. Applications that produce non-graphical results can send the data back to the host for storage or output. Applications that use data distributed among the pixel nodes can use the serial I/O communications facility for interprocessor communication. DEVtools includes functions to read commands, send messages to the host, perform frame buffer I/O, serial I/O, memory management, processor synchronization, etc.

Pipe and pixel node programs are created in much the same manner as would be used to create host executables, with the exception that the command **devcc** is used in place of cc.

The Pixel Machine

Introduction

Generating a realistic image from complex two and three dimensional data in real time demands a lot of computational power. Graphics and image processing algorithms, particularly rendering algorithms, often perform a set of operations to generate each pixel, with little or no interaction between pixels. These algorithms are candidates for mapping to a parallel architecture, with performance increasing nearly linearly with the number of processors.

In many display systems, a single custom processor handles the typical frame buffer operations. This approach is adequate for rendering simple two-dimensional images. However, when realistically shaded images must be displayed in real time, a single processor cannot provide the necessary computational power.

Pipelines or arrays of special purpose processors provide high performance at the expense of flexibility. Their performance improvements are limited to the narrow range of algorithms that they were designed to solve.

While the use of parallelism and pipelining gives a system the power needed to render high quality images in real time, the use of programmable processors provides the flexibility to attain high performance for a wide range of graphics and image procession algorithms. It is much easier to change a program from Gouraud shading to Phong shading, for example, than to redesign a customized processor.

The AT&T Pixel Machine combines the strengths of both coarse grain pipelining and multiple instruction/multiple datapath (MIMD) computing arrays. A pipeline of computing elements processes the serial tasks that precede pixel-level processing while a processor array provides high-bandwidth access to an integrated frame buffer and computes individual pixel values. The processors in both the pipeline and the array are programmable, with hardware floating point operations.

The programmability of the processors allows all algorithms to be implemented in software. A set of mapping functions transfer frame buffer algorithms written for conventional serial computers to algorithms that execute in the pixel nodes and access the distributed frame buffer. The ability to use floating point computations in frame buffer operations such as antialiasing, ray tracing, and cascaded filtering, allows high quality image generation.

The Pixel Machine provides up to 820 megaflops of processing power and 48 megabytes of memory for data visualization applications, including three-dimensional rendering and animation, image processing, and display of multi-dimensional data.

Pixel Machine Architecture

Design

The Pixel Machine combines the strengths of both coarse grain pipelining and MIMD computing arrays to provide the performance of a supercomputer on image synthesis and image analysis applications. Synthesis applications include the generation and display of two and three dimensional scenes as well as the visualization of scientific and engineering computations. Analysis applications include the processing and interpretation of image data from, say, a nuclear magnetic resonance machine or a satellite. The design philosophy is:

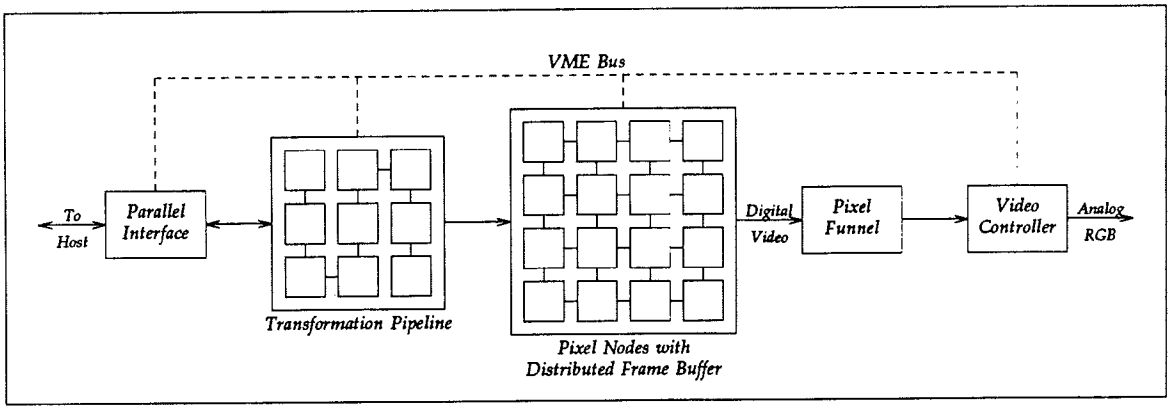
- Use floating point computation and large image memories, which are useful for image processing.
- Design simple, modular processors that can be repeated a number of times to build a system.
- Implement all algorithms in software.

The modular approach enabled the Pixel Machine to be designed, built, and programmed by a small group of people in a short period of time. The decision to implement algorithms in software rather than special purpose VLSI chips gives wide functionality, faster implementation of new algorithms, and easier modification of existing ones.

The architecture has the following features:

- AT&T DSP32 processors
- 0, 9 or 18 pipe nodes, configurable as zero, one or two pipelines
- 16, 20, 32, 40, or 64 pixel nodes
- 32-bit pixel and z-buffer data
- floating-point computation for pixel generation
- a frame buffer with pixel-interleaved parallel architecture
- 1280x1024 or 1024x1024 high-resolution 60 Hz non-interlaced display
- NTSC and PAL display modes
- a large image memory that allows single, double, or quadruple buffering
- software that transparently handles the different frame buffer sizes

Figure 1-1: Pixel Machine block diagram



Both the pipe nodes and the pixel nodes include an AT&T DSP32 Digital Signal Processor, a 32–bit, high speed, programmable device whose features include:

- 20 MHz, 5 MIPS, 10 MFLOPS
- 4K bytes of on-chip memory
- 32–bit floating point arithmetic
- four 40–bit floating point registers
- twenty-one 16–bit integer and address registers
- an interface to off-chip expansion memory
- parallel and serial I/O ports with DMA

The DSP32 can be programmed in assembler language or in C. The software development environment includes a compiler, an assembler, a linking loader, and a simulator. All arithmetic operations on data are floating point operations. Only memory address generation and program control calculations use integer arithmetic. Software is developed on a host computer, typically a SUN or SGI workstation. The Pixel Machine is connected to the host computer via the VMEbus.

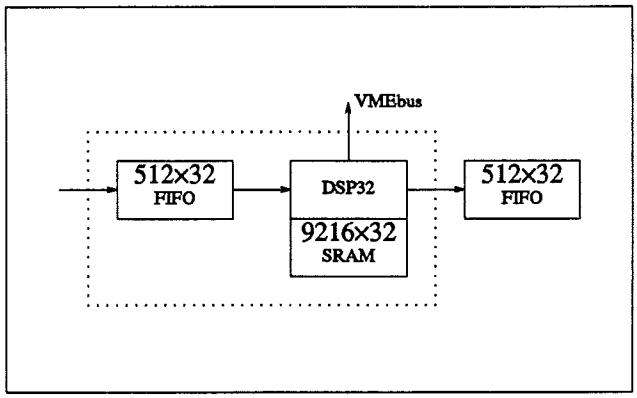
Inside the Pixel Machine, there are 0, 9 or 18 pipe nodes configured as zero, one or two pipelines, a broadcast bus that transfers data from the end of the pipes to the pixel nodes, an array of 16, 20, 32, 40, or 64 pixel nodes that form a distributed frame buffer, and a *pixel funnel* that transfers digital video data from the frame buffer to the video processor, which controls the display monitor.

Each pipe and pixel node can be viewed as a small independent computer that executes its instructions and operates on data asynchronously with all the other nodes. Programs are loaded into the nodes by the host, using unique, software-defined node numbers to distinquish between them.

Pipe Nodes

Figure 1-2 shows a block diagram for a pipe node. Each pipe node has a DSP32 processor that executes five million instructions or ten million floating point operations per second. The parallel DMA interface of each processor is connected to the VMEbus. The pipe nodes have 9K×32 bits of memory for instructions and data, a 512×32 bit input FIFO containing data written by the previous pipe node, a 512×32 bit output FIFO where all output is written, to be read by the next node in the pipeline.

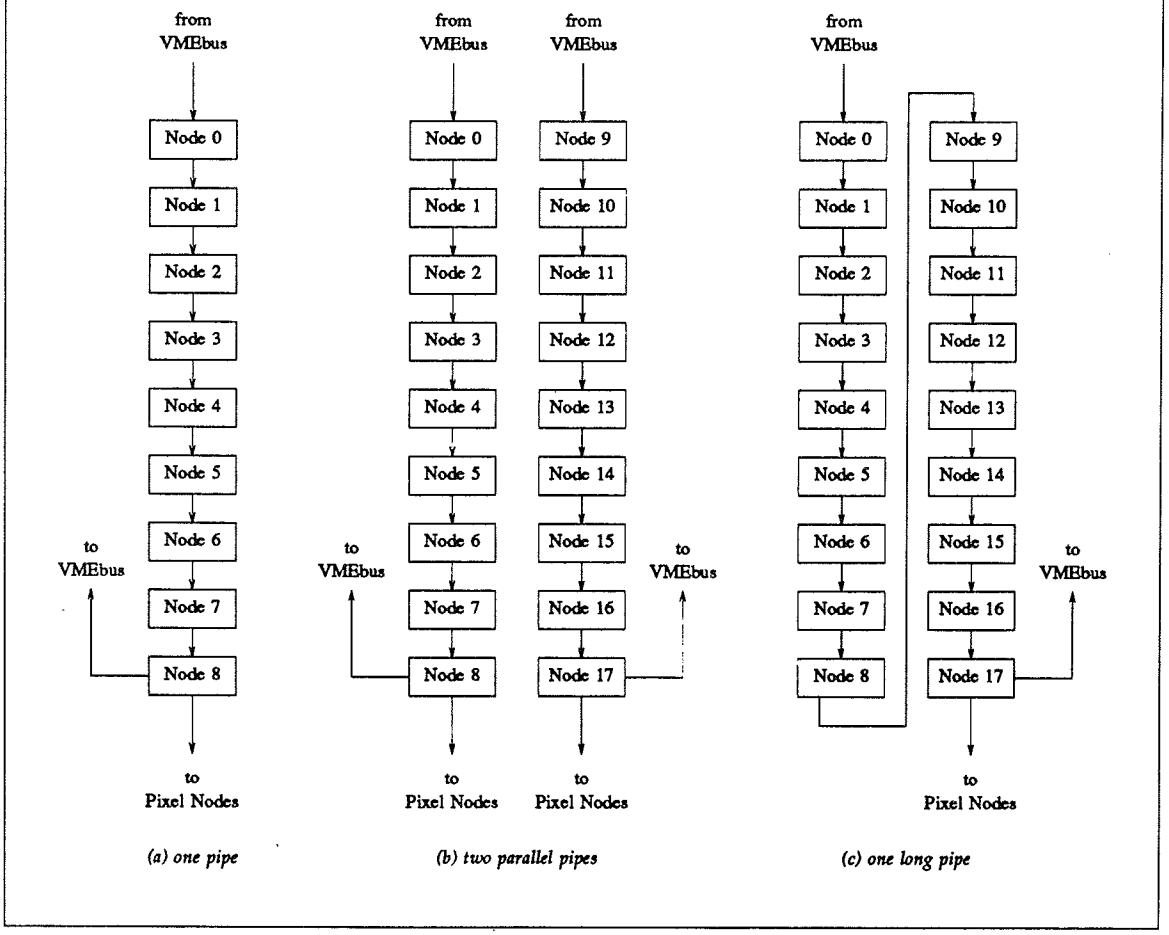
Figure 1-2: Pipe node block diagram



The host computer provides input to the first pipe node via the VMEbus. The output from the last pipe node is broadcast to all of the pixel nodes. In addition, the last pipe node has a second output FIFO that can be read by the host, again via the VMEbus.

A system can have 0, 9 or 18 pipe nodes. The 18-node systems are software-configurable as either two nine-node parallel pipelines or one 18-node pipeline (see Figure 1-3). In a two pipeline system, the node in each pipeline has the ability to request, acquire, and release the broadcast bus. In the one pipeline system, the last node has continuous access to the broadcast bus.

Figure 1-3: Pipeline configurations



The pipe nodes perform those parts of the algorithms that are serial in nature and can be pipelined. These include 3D transformations, clipping, projections, shading, and image filtering. The pipeline can also be used as a hardware subroutine by processes running in the host computer, which can send data to the first node and read results from the last one.

Pipe Node Memory Areas

This section describes the memory areas and their use within pipe node programs. Direct use of these memory areas and flags is discouraged because the addresses of the areas and other dependencies, such as timing requirements, are considered to be implementation defined and may be different in future systems. When it is necessary to access these memory areas, the symbolic names given below and defined in the header file **pipe.h** should be used.

Table 1-1: Memory Map of a Pipe Node's Address Space

Address	Name	Mode	Description
0000 – 0060		R/W	crt0 (startup code)
0060 – 7fff		R/W	static RAM for program and data storage
c000 – c1ff		R	Input FIFO
c000	PM_FIFOIN	R	Input FIFO
c000	PM_FIFOIN_L	R	Input FIFO – low word
c002	PM_FIFOIN_H	R	Input FIFO – high word
c200 – c3ff		W	Output FIFO
c200	PM_FIFOOUT	W	Output FIFO
c200	PM_FIFOOUT_L	W	Output FIFO – low word
c202	PM_FIFOOUT_H	W	Output FIFO – high word
c400	PM_EMPTY_IN	W*	Input FIFO empty flag
c600	PM_HALF_IN	W*	Input FIFO half-full flag
c800	PM_HALF_OUT	W*	Output FIFO half-full flag
ca00	PM_FULL_OUT	W*	Output FIFO full flag
e000 – efff			on-chip ROM (unusable)
f000 – ffff		R/W	static RAM for program and data storage
Last Node on a Board			
b000	PM_BUS_RELEASE	W	Broadcast bus release
ce00 – cfff			Feedback FIFO
ce00	PM_FIFOFB	W	Feedback FIFO
ce00	PM_FIFOFB_L	W	Feedback FIFO – low word
ce02	PM_FIFOFB_H	W	Feedback FIFO – high word
d400	PM_HALF_FB	W*	Feedback FIFO half-full flag
d600	PM_FULL_FB	W*	Feedback FIFO full flag
d800	PM_BUS_GRANT	W*	Broadcast bus grant
da00	PM_BUS_REQUEST	W	Broadcast bus request
dc00	PM_PIXEL_ALLRDY	W*	Pixel node vsync flag
de00	PM_PIXEL_XFLAG	W*	Pixel node psync flag

The mode field in the memory map defines whether the address can be read (R), written (W), or both (R/W). Memory areas that are not defined must not be referenced.

Sync Signal Selectors

The flags in the memory map designated with an asterisk (''*'') are boolean values whose state may be sensed by the DSP32 conditions *sys* and *syc* (*sync set* and *sync clear*). The values of each of these flags is routed through a multiplexer. The value to be passed to the DSP32 is selected by writing any value into the address associated with the flag to be sensed. For example, to check the input FIFO half-full flag, you would write a value to the address c600 (PM_HALF_IN), then check the sync signal. If the signal is set, then the condition is true; the FIFO is half full.

The sync signal must not be tested immediately after setting one of the signal selector flags because a short delay is required for the hardware to switch the signals. The minimum delay is three instructions for the last node on a board and two instruction for all of the other nodes.

Static RAM

The static RAM area totals 36k bytes of memory for general purpose program and data storage. The standard memory definition file (*ifile*) designates 0060 through 7fff for program storage and f000 through ffff for data storage. This can be changed by supplying an *ifile* to the linker that distributes the memory in the manner desired.

Input FIFO

The input FIFO contains up to 2048 bytes of data, organized as 512 units of four bytes each. The input FIFO may be read as one four-byte word, two 2-byte words or as four bytes, however, all four bytes of each FIFO entry must always be read in order for the contents of each byte of the FIFO to remain synchronized with the others. The status of the input FIFO is checked by writing a value to *input empty* or *input half-full*, then checking the sync flags (sys or syc). The FIFO must not be read when it is empty.

The FIFO can be read using any of the 32 bit 4-byte words that are mapped to the output port of the FIFO. This allows a program to use a standard block move routine to read to the FIFO as long as no more than 128 4-byte words are moved at one time.

Output FIFO

The output FIFO is the input FIFO of the next pipe node. For the last node in the pipeline, the output FIFO is the broadcast bus to the input FIFO's of the pixel nodes. As with the other FIFOs, all four words must always be written in order to maintain synchronization. The status of the output FIFO is checked by writing a value to *output full* or *output half-full*, then checking the sync flags (*sys* or *syc*). The FIFO must not be written when it is full. The FIFO of the last node (the broadcast bus) must not be written unless the bus is granted to the board that wishes to do the write operation.

The FIFO can be written using any of the 32 bit 4-byte words that are mapped to the output port of the FIFO. This allows a program to use a standard block move routine to write to the FIFO as long as no more than 128 4-byte words are moved at one time.

Feedback FIFO

Each pipe board contains a feedback FIFO that can be read by the host system. Both feedback FIFOs can be used regardless of whether the pipes are operating in serial or parallel mode. As with the other FIFOs, all four words must always be written in order to maintain synchronization. The status of the feedback FIFO is checked by writing a value to *feedback full* or *feedback half-full*, then checking the sync flags (sys or syc). The FIFO must not be written when it is full.

The FIFO can be written using any of the 32 bit 4-byte words that are mapped to the output port of the FIFO. This allows a program to use a standard block move routine to write to the FIFO as long as no more than 128 4-byte words are moved at one time.

Accessing the Broadcast Bus

In a dual-pipe system with the pipes operating in parallel mode only one pipe has access to the broadcast bus at any point in time. When operating in parallel, each pipe must release access to the broadcast bus to the other pipe on a regular basis, because both pipes must be able to write to the broadcast bus in order to achieve optimal performance.

Access to the broadcast bus is controlled by three memory locations: PM_BUS_REQUEST, PM_BUS_RELEASE, and PM_BUS_GRANT. PM_BUS_REQUEST is used to request access to the broadcast bus. PM_BUS_RELEASE is used to relinquish control of the broadcast bus to the other pipe. PM_BUS_GRANT is used to connect the bus grant signal to the sync signal of the DSP32 so that the software can sense whether access to the bus has been granted.

```
*PM_BUS_REQUEST = r1        /* Can be any register-the contents don't matter */
*PM_BUS_GRANT = r1
2*nop              /* Delay needed before signal can read */
loop:
if (syc) goto loop /* Wait for grant signal to be true */
nop
```

To gain access to the bus

```
*PM_BUS_RELEASE = r1        /* Can be any register - the contents don't matter */
```

To release the broadcast bus

Single pipe configurations and dual pipe configurations operating in a series do not need to check the bus grant flag because the bus will always be granted to the node that has access to the broadcast bus.

Pixel Node Flags

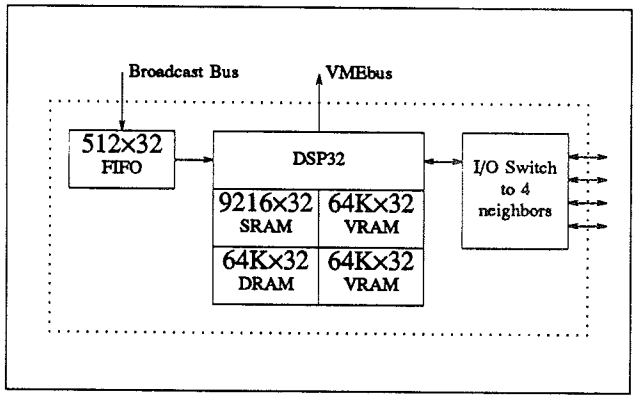
Pipe activity can be synchronized with pixel node activity by using the PM_PIXEL_ALLRDY and PM_PIXEL_XFLAG signals. The PM_PIXEL_ALLRDY flag is true when all of the pixel nodes have set their *vsync* signals. PM_PIXEL_XFLAG is true when all of the pixel nodes have set their *psync* signals. These signals are available to both pipes of a dual pipe system in both serial and parallel modes.

Pixel Nodes

The pixel nodes form an $n \times m$ array with a distributed frame buffer. They receive their data from the broadcast bus of the pipe nodes and store their output into the frame buffer or return it to the host computer. Mapping registers provide uniform access to the frame buffer across different configurations of pixel nodes, and a four–way multiplexed I/O switch and channel allows two-way communication with the four neighboring pixel nodes.

A DSP32 is the computing element in each pixel node. Just as in the pipe nodes, there is an 8192×32 bit static RAM in the node, in addition to the 1024×32 bits of on-chip storage. Figure 1-4 shows a block diagram.

Figure 1-4: Pixel node block diagram



Two banks of 64K×32 bit VRAMs form the pixel node's piece of the the distributed frame buffer. The video RAMs store the red, green, blue, and alpha settings for the pixels. These memories can be displayed or used as off-screen storage for images.

Pixel nodes also contain a 64K×32 bit dynamic RAM that can be used to hold floating point z-buffer values or any data in byte or word format, pixels in floating point representation, sections of display list, or code segments. The processor can execute instructions from this memory, although it is slower than executing from either the on-chip or SRAM memory.

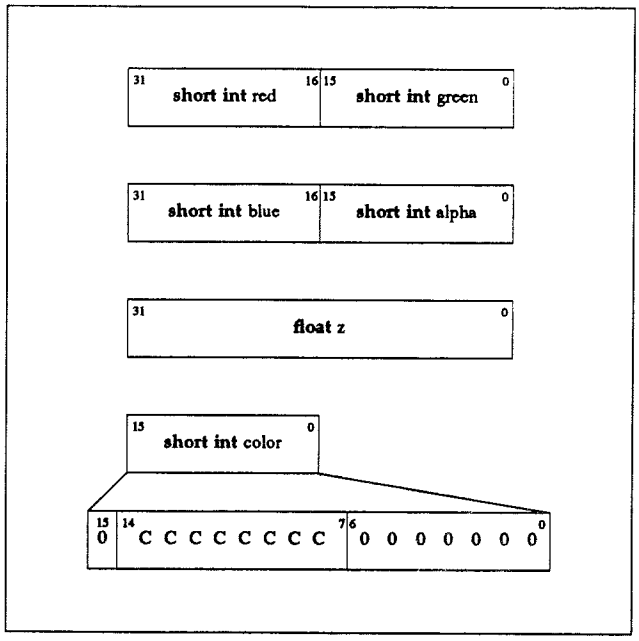
Table 1-2: Pixel array configurations

Pixel Nodes	Pixel Node Array physical	virtual	Display Resolution	Subscreens Size	# pixels	per node
16	4x4	8x8	1024x1024	128x128	65536	4
20	5x4	10x8	1280x1024	128x128	65536	4
32	8x4	8x8	1024x1024	128x128	32768	2
40	10x4	10x8	1280x1024	128x128	32768	2
64	8x8	8x8	1024x1024	128x128	16384	1
64	8x8	8x8	1280x1024	160x128	20480	1

A Pixel Machine can be configured with 16, 20, 32, 40, or 64 pixel nodes. Table 1-2 summarizes these five configurations. The two video RAMs and the dynamic RAM (when used to store pixel data) are organized as three blocks of 256x256 32-bit pixels. Each bank can be logically divided further into smaller blocks, called *subscreens* (see Chapter 3 for a discussion on subscreens).

In order to allow configuration-independent software, the concept of *virtual* pixel nodes that reside inside physical nodes is introduced. Each virtual node accesses a single subscreen. All systems have either 64 or 80 virtual nodes, depending on the resolution of the display screen. In a 64-node system, each physical pixel node contains a single virtual node, while a 16- or 20-node system has four virtual nodes per physical node.

Figure 1-5: Pixel format



Pixel data is stored in the frame buffer as 16–bit signed integers. The four components of a pixel, (red, green, blue, α) form two 32–bit words, as shown in Figure 1-5. Only eight of the 16 bits in a pixel component are populated with memory.

Each pixel node has a serial input/output (SIO) channel that provides a communication path to its four nearest neighbors, allowing the Pixel Machine to function as a computing mesh. Node placement follows pixel interleaving conventions, as shown in Figure 1-6. Thus, in a 4x4 array of pixel nodes, node 5's neighbors are nodes 1, 4, 6, and 9. The edges of the mesh wrap around to form a torus, so node 0's neighbors are 1, 3, 4, and 12, for example.

The SIO capability at each node consists of one input and one output serial port that operates at peak rates of 16*Mbits/sec*. Pixel data can be moved from node to node at a sustained rate of 5.25*Mbits/sec*, including the time spent buffering pixel data to and from the display memory. In practice, however, processor cycles will be shared between an application program and SIO, and the data transfer rate will be proportionately slower.

Pixel Node Memory Areas

This section describes the memory areas and their use within pixel node programs. Direct use of these memory areas and flags is discouraged because the addresses of the areas and other dependencies, such as timing requirements, are considered to be implementation defined and may be different for future systems. When it is necessary to access these memory areas, the symbolic names given above and defined in the header file **pixel.h** should be used.

Address	Name	Mode	Description
0000 – 0060		R/W	crt0 (startup code)
0060 – 7fff		R/W	static RAM for program and data storage
8000 – bfff	PM_PIXEL_MEM	R/W	reserved for VRAM/ZRAM access via page registers
8000 – 83ff		R/W	memory reference via page register 0
8400 – 87ff		R/W	memory reference via page register 1
8800 – 8bff		R/W	memory reference via page register 2
			.
			.
			.
bc00 – bfff		R/W	memory reference via page register 15
c800 – c840	PM_MAP_ADDR	R/W	page register storage
c800 – c803		R/W	page register 0
c804 – c807		R/W	page register 1
c808 – c80b		R/W	page register 2
			.
			.
			.
c83c – c840		R/W	page register 15
d002 – d003	PM_FLAG_REG		drawing mode register
d800 – dfff			Input FIFO
d800 – d803	PM_FIFO_32	R	Input FIFO
d800 – d801	PM_FIFO_16L	R	Input FIFO – low word
d802 – d803	PM_FIFO_16H	R	Input FIFO – high word
e000 – efff			on-chip ROM (unusable)
f000 – ffff		R/W	static RAM for program and data storage

The mode field in the memory map defines whether the address can be read (R), written (W), or both (R/W). Memory areas that are not defined must not be referenced.

Static RAM

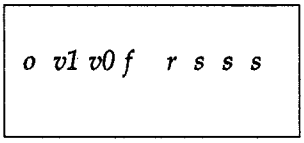
The static RAM area totals 36k bytes of memory for general purpose program and data storage. The standard memory definition file (*ifile*) designates 0060 through 7fff for program storage and f000 through ffff for data storage. This can be changed by supplying an *ifile* to the linker that distributes the memory in the manner desired.

Flag Register

Each pixel node has a flag register that contains:

- the sync signal section flags
- the node's psync and vsync flags
- the flags that select the video buffer to be displayed
- the overlay flag for the processor

The flag register is 16–bits and is accessed through the address d002 (PM_FLAG_REG). The following describes the structure of the register:



where:

- *o* is the overlay flag. For overlaying to be enabled, the overlay flag of all the drawing nodes must be set, and the overlay flag for all the pixel board mode registers (see below) must also be set.
- *v0* and *v1* designate which area of the video memory should be displayed. *v0* controls whether the top buffer or bottom buffer is displayed (0 displays the top buffer). If *v1* is false, the image is displayed starting at the first pixel of image memory. If *v1* is true, an offset of 128 pixels is used.
- *f* is the processor's psync flag.
- *r* is the processor's vsync flag.

- *sss* is the sync signal selection flag. The sync signal selection flag is used to designate which of several flags is to be connected to the DSP32 sync signal. Once selected these signal may be sensed by the DSP32 conditions, sys and syc (sync set and syc clear). The value of *sss* must be one of the following:
  - 000 (PM_EMPTY_LOWER): input FIFO 16–bits (not usually used)
  - 001 (PM_HALF_FULL): lower 16–bits (not usually used)
  - 010 (PM_DRAW_EMPTY): input FIFO empty flag
  - 011 (PM_DRAW_HALF): input FIFO half-full flag
  - 100 (PM_VERTBLNK): vertical blanking flag
  - 101 (PM_HORZBLNK): horizontal blanking flag
  - 110 (PM_XFLAG): all processors vsync flags are set
  - 111 (PM_ALLRDY): all processors psync flags are set

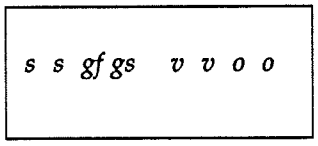
There must be a delay between setting the flag register and testing the sync signal in order for the hardware to have time to switch the signals. A minimum of two instructions must be executed between setting the flag register and checking the sync signal.

Pixel Array Board Mode Register

Each pixel array board contains a mode register that contains:

- the board level overlay mode flag
- the video shift flag
- gate flags that are used to disable pixel boards
- serial I/O direction flags

The mode register is a 16–bit register that can only be accessed by the host. The structure of the register is:



where:

- *ss* is the serial I/O direction to be used.
- *gf* is the DEV_GATES_FIFO flag. If true the FIFO flags are included when determining the FIFO flags that are passed to the pipe board(s) to determine whether they can broadcast to the pixel nodes.
- *gs* is the DEV_GATES_SYNC flag. If true, the psync and vsync flags from the processors on this board are included when determining the value of the PM_ALLRDY and PM_XFLAG signals.
- *vv* is the video shift mode, and must be one of the following;
  - 00 (DEV_SHIFT_NOT964): used for all models except the 964
  - 01 (DEV_SHIFT_TOP964): upper 4 lines of each 8 scan lines of the 964
  - 11 (DEV_SHIFT_BOT964): bottom 4 lines of each 8 scan lines of the 964
- *oo* is the overlay mode. This indicates whether or not overlaying is enabled, and if enabled which of several modes is to be used. The value must be one of the following:
  - 00 (DEV_OVERLAY_OFF): Overlaying is disabled.
  - 01 (DEV_OVERLAY_ON): Overlaying is enabled. If the overlay value is zero, the RGB value is used. If the overlay value is 255, the inverse of RGB is used. If the overlay value is 1-254, the overlay value is used.
  - 10 (DEV_OVERLAY_FORCE): The overlay value is always used.
  - 11 (DEV_OVERLAY_MASK): If the high order bit (80 hex) of the overlay value is true, the overlay value is used, otherwise RGB is used.
  - The value displayed for each pixel is determined by:
    - the value of the pixel memory (RGB and overlay)
    - the overlay mode in the pixel mode register of each pixel array processor board
    - the overlay flag in each of the pixel node flag registers

If all of the overlay flags are on, overlay mode is determined by the overlay mode in the pixel mode register. If all of the overlay flags are off, then DEV_OVERLAY_OFF mode is used.

The three components described above are used to make two decisions:

1. which values should be sent to the video controller for RGB. The video controller accepts 24-bits of color information, 8-bits each of red, green, and blue. The 24-bits can contain either the red, green, and blue pixel data, or the overlay data can be copied into the red, green, and blue fields (the same 8-bits is copied into each of the three colors).

2. which lookup tables on the video controller should be used to produce the final color value. Normally the primary lookup table is used. When the overlay data is being displayed, the overlay table is used for those pixels.

Table 1-3: Memory Map of a Pixel Node's Address Space

Overlay Flag	Overlay Mode	Overlay Value	RGB	Lookup Table
OFF	(any)	(any)	RGB	Primary
ON	OFF	(any)	RGB	Primary
ON	ON	0	RGB	Primary
		1-254	ooo	Overlay
		255	~RGB	Primary
ON	FORCE	(any)	RGB	Overlay
ON	MASK	0-127		RGB
		128-255	ooo	Overlay

When the overlay flag in the pixel nodes is off (false), the RGB data is displayed using the primary lookup table in all cases. When the overlay flag in the pixel nodes is true, the displayed value depends on the overlay mode and the contents of each overlay pixel.

DEV_OVERLAY_ON: If the overlay value is zero, the red, green, and blue data is displayed using the primary lookup table. If the overlay value is in the range 1-254, the overlay value is used for red, green, and blue, and the overlay lookup table is used. If the overlay value is 255, the bitwise complement of red, green, and blue is displayed using the primary lookup table.

DEV_OVERLAY_FORCE: The red, green, and blue data is displayed using the overlay lookup table.

DEV_OVERLAY_MASK: If the overlay value is in the range 0-127, the red, green, and blue data is displayed using the primary lookup table. If the overlay value is in the range 128-255, the overlay value is used for red, green, and blue, and the overlay lookup table is used.

Input FIFO

The input FIFO contains up to 2048 bytes of data, organized as 512 units of four bytes each. The input FIFO may be read as one four-byte word, two 2-byte words or as four bytes, however, all four bytes of each FIFO entry must always be read in order for the contents of each byte of the FIFO to remain synchronized with the others. The status of the input FIFO is checked by setting the PM_DRAW_EMPTY or PM_DRAW_HALF bit in the mode register then checking the sync flags (sys or syc). The FIFO must not be read when it is empty.

Z Memory

The Z memory (also referred to as DRAM or ZRAM) consists of 256k bytes of dynamic RAM memory for each pixel node. The Z memory can be used for floating point values, integers, or bytes, and it is accessed through the use of page registers, which are described below.

Video Memory

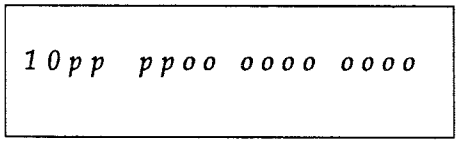
Each node contains 512k bytes of video memory. The video memory is also accessed through the use of page registers, and it is divided into two sections, VRAM0 and VRAM1. Each of these areas is subdivided into two sections: one containing the red and green pixel components (RG0 and RG1) and the other containing the blue and overlay pixel components (BO0 and BO1).

As described above, each color component of a pixel consists of 8 bits of data stored in the high order bits (the bits after the sign bit) of a short integer. In the red/green section the red pixel data is stored in the low order word of the 32 bit value for a pixel, and the green data is in the high order word. Since the byte ordering on the DSP32 is least significant byte first, the red pixel data is stored at byte location *N*, and the green information is at byte location *N*+2. In the blue/overlay region the blue data is in the low order word (memory address *N*) and the overlay data is in the high order word (memory location *N*+2).

Page Registers

Page registers allow the pixel nodes to access 256k bytes of ZRAM and 512k bytes of video RAM even though the DSP32 only has a 16 bit address space.

Memory addresses in the range 8000 through bfff are reserved for paged memory access. Put another way, all addresses that begin with the bit sequence 10 are reserved for paged memory access. Paged memory addresses have the form:



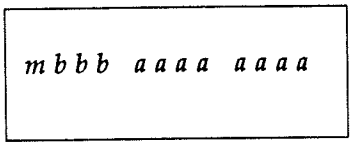
where:

10 designates this as a paged address

pppp is the page register number (0 to 15)

oooooooooo is the ten bit offset from the address contained in the page register.

The page registers are accessed as four byte values at memory locations starting at c800 for page register 0, c804 for page register 1, and so on, up to c83c for page register 15. Only the 12 low order bits of the page register are used. The structure of the page register is:



where *m* is the mode selection bit. The two addressing modes are:

- 0: Fixed row addressing. The address field contains the row number to be accessed. This mode allows the access of a processor's pixels on a given scan line.
- 1: Fixed column addressing. The address field contains the column number to be accessed. This mode allows the access of a processor's pixels on a given screen column.

bbb is the bank selection field. The defined values are:

- 001: PM_ZMEM – Z memory
- 100: PM_RG0 – the red/green components stored in VRAM bank 0
- 101: PM_BO0 – the blue/overlay components stored in VRAM bank 0
- 110: PM_RG1 – the red/green components stored in VRAM bank 1
- 111: PM_BO1 – the blue/overlay components stored in VRAM bank 1

aaaaaaaa is the 8–bit extended address value.

Macros are provided to build and use page registers. They hide the internal structure of the page register and the physical addresses that are used.

PMdesc is used to build a value to be stored in a page register. The format of the macro is:

PMdesc(*mode,bank*)+*extended_address*

mode must have the value PM_FIX_ROW or PM_FIX_COL

bank must have the value PM_ZMEM, PM_RG0, PM_BO0, PM_RG1, or PM_BO1

PMpagereg is used to access the locations in which the page registers are stored. The format of the macro is:

PMpagereg(*reg_number*)

reg_number must have a value in the range of 0 to 15

PMxlate is used to generate an address that uses a page register. The format of the macro is:

PMxlate(*reg_number*)

reg_number must have a value in the range of 0 to 15

The following is an example of the macros used in assembly code. r2 holds the row index and r3 contains the column index:

```
r1 = PMdesc(PM_FIX_ROW, PM_ZMEM) + r2 /* Access the Z memory row
                                         designated by the value in r2 */
*PMpagereg(4) = r1            /* Move the descriptor into
                                 page register 4 */
r3 = r3 * 2                   /* convert the column index in r3 to a byte    */
r3 = r3 * 2                   /* offset by multiplying by 4 (since each
                                 float takes up 4 bytes)                     */

r4 = PMxlate(4) + r3          /* Get the address of the value
                                 designated by the row
                                 number in page register 4,
                                 plus the offset in register r3              */
a0 = *r4                      /* Move the desired value into register a0 */
```

Internode Communications

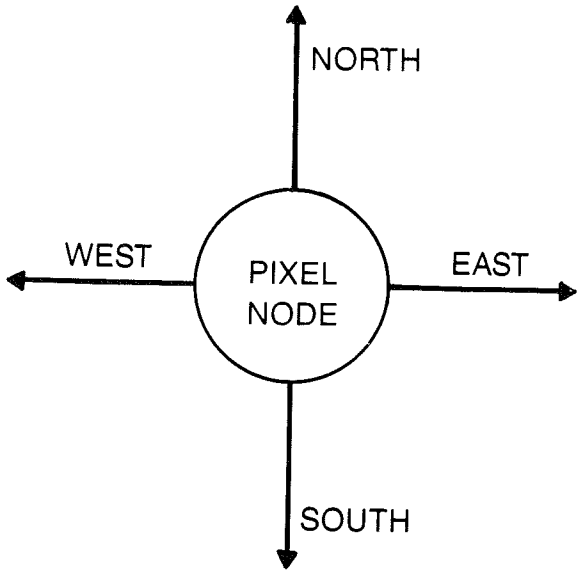
In some applications, it may be necessary for the pixel nodes to exchange data. For example, to scroll the image memory by one pixel requires that each processor send the entire contents of its pixel memory to a neighboring processor, replacing the pixels with those received from another neighboring processor. Another use is communicating intermediate results of a computation that is distributed over the processor array, for example, multiplying matrices that are distributed among the nodes.

To accomplish this exchange of data, the Pixel Machine supports nearest-neighbor communications among the pixel nodes. Because this communication is implemented using the serial I/O port of the DSP chip, it is sometimes referred to as SIO (Serial I/O) to distinguish it from the node-host communication implemented with the parallel I/O port.

Topology

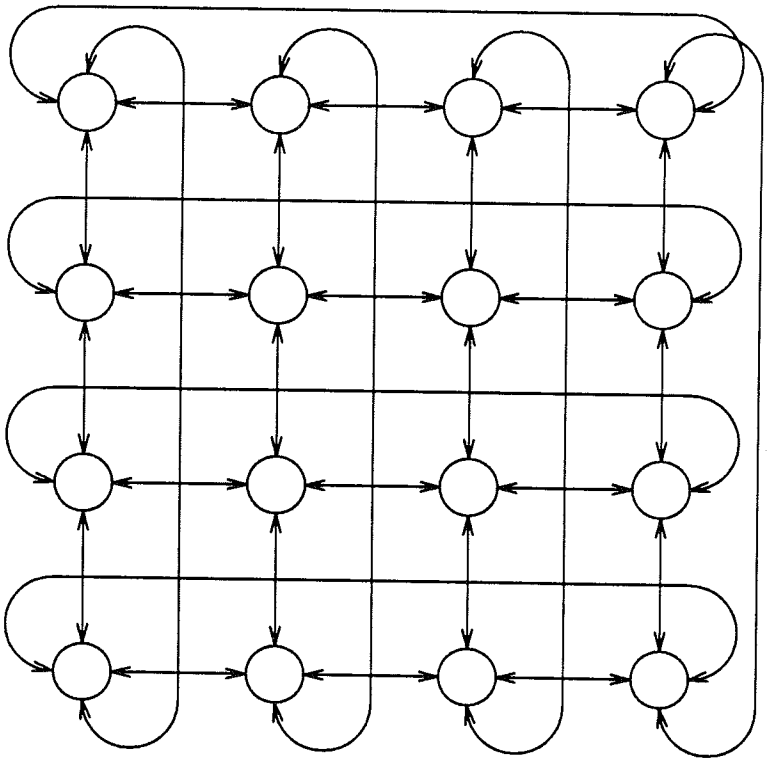
Each node can communicate with one of four neighboring nodes over the communications links; the neighbors of any node, and the links connecting to those neighbors, are referred to as North, South, East, and West.

Figure 1-6: Link directions from a pixel node



The communications links between the mesh of pixel nodes form a torus: a mesh connected at the edges (Figure 1-7). This allows every node to connect to four neighbors. The neighboring nodes are arranged in the same pattern as pixels are interleaved among pixel nodes; complete topologies for all Pixel Machine models are given at the end of this section.

Figure 1-7: Torus topology for a 4x4 processor mesh

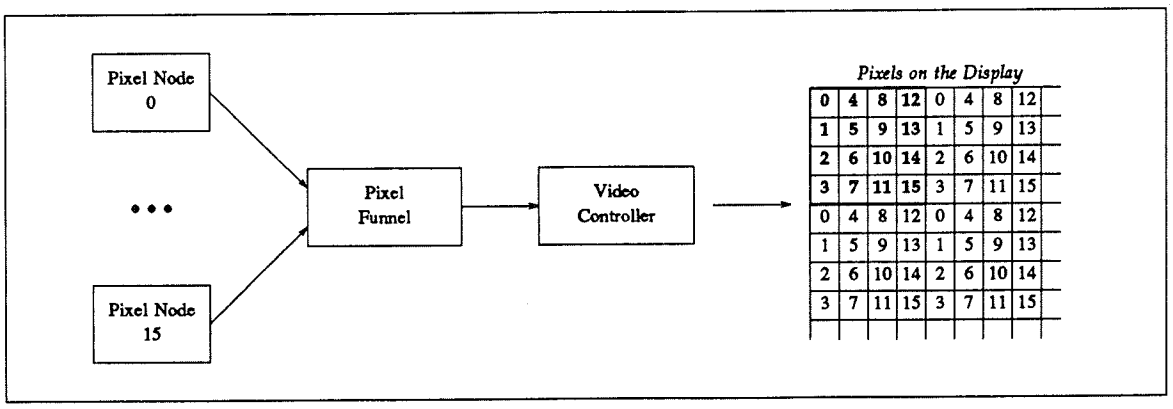


While each node has links to four neighboring nodes, only one of the four links may be active at a given time. Furthermore, all nodes communicate in the **same** direction. Setting the active link (also referred to as setting the link *direction*) is done by the host using the **DEVserial_direction** call. This restriction means that all pixel nodes must agree on the order in which data is sent over the links, and to which destination nodes it is sent. The link direction is set to North, South, East, or West, and sends data to the neighboring node in that direction, while receiving data from the neighboring node in the opposite direction. For example, if the link is set East, a node will send data to its East neighbor while receiving data from its West neighbor.

Video Display

The frame buffer is distributed throughout the array of pixel nodes. An example is shown in Figure 1-8. The *pixel funnel* rearranges pixels from the frame buffer into a properly ordered raster scan sequence. Both the video controller and the pixel funnel are software configurable for the five different pixel arrays.

Figure 1-8: Pixel mapping in the distributed frame buffer for a PXM 916



The frame buffer stores (red, green, blue, α) values for each pixel; the video processor may substitute the α value for the red, green, or blue value, based on the display mode:

$$RGB_{out} = rgb_{in}$$

OVERLAY_OFF

$$RGB_{out} = \begin{cases} rgb_{in} & \text{if } \alpha=0 \\ rgb_{in} & \text{if } \alpha=255 \\ \alpha\alpha\alpha & \text{if } 0<\alpha<255 \end{cases}$$

OVERLAY_ON

$$RGB_{out} = \begin{cases} rgb_{in} & \text{if } 0\leq\alpha\leq128 \\ \alpha\alpha\alpha & \text{if } 128<\alpha\leq255 \end{cases}$$

OVERLAY_FORCE

$$RGB_{out} = rgb_{in}$$

OVERLAY_MASK

The video processor uses six 256×10 lookup tables, or color maps, to translate 8-bit pixel color values to 10-bit video data. Three of the tables map red, green, and blue. The other three map α to red, green, and blue values.

There are two sets of color maps. One set contains high-speed *video* tables that are used to convert video data. The other set are *shadow* tables that can be read and written via the VMEbus. The contents of the shadow tables are automatically copied to the video tables during a vertical retrace period, with copying enabled and disabled in software. The shadow tables prevent two problems common to many video systems from arising: snowy and sheared video because color maps are modified during active video periods, and distracting flashes on the screen because of partially modified color maps.

In high-resolution mode, the video system displays 1024 lines of either 1024 (in systems with 16, 32 or 64 pixel nodes) or 1280 (in systems with 20, 40, or 64 nodes) pixels, at 60 Hz non-interlaced. In NTSC mode, the video system uses the RS-170A format to display 485 of 720 pixels in all pixel node configurations. In PAL mode, 575 lines of 720 pixels are displayed.

Video Format

DEVtools supports the PAL and NTSC video format, PAL is the video format used in Europe which corresponds to the NTSC format used in North America. PAL enables the Pixel Machine to produce a PAL signal for European customers to use with their video equipment. The screen resolution for PAL is 720x576; NTSC resolution is 720x485.

Using the Pixel Machine in PAL mode is similar to using it normally. All the user is required to do is set the appropriate model, which should be 964p for a single pipe 964 and 964pd for a dual pipe system, and issue a **hypinit** command to actually switch to PAL video format. To switch back to standard hi-res video, change the model to 964 or 964X and issue a **hypinit** command.

For NTSC mode, set the model as you would normally but append an **n** to the end of it. For example, if you have a 964 with a dual pipe, set the model to a 964dn.

For high resolution the subscreens are:

Model	Subscreens	Size
964X	1	160x128
964	1	128x128
940	2	128x128
932	2	128x128
920	4	128x128
916	4	128x128

For PAL the subscreens are:

Model	Subscreens	Size
964p	1	90x72
940p	2	72x72
932p	2	90x72
920p	4	72x72
916p	4	90x72

For NTSC the subscreens are:

Model	Subscreens	Size
964n	1	90x61
940n	1	72x122
932n	1	90x122
920n	1	144x122
916n	1	180x122

The DEVtools variables **PMimax** and **PMjmax** are set to these limits minus one. Therefore, on a model 940n (NTSC) **PMimax** would be equal to 71 and **PMjmax** would be 121.

System Configurations

As described above, the Pixel Machine can be configured with five different pixel array sizes and three different pipeline sizes. The ten models are described in Table 1-4. Models **964** and **964d** can be programmed to display either 1024x1024 or 1280x1024 pixels.

In high-resolution display mode, Models **916** and **920** have two $rgb\alpha$ frame buffers and one z-buffer, with enough memory to render full-screen 32-bit images in double-buffered mode with a floating point depth buffer. Models **932** and **940** have two off-screen buffers in addition to the two displayable buffers, plus two z-buffers. In addition to the two displayable buffers and one z-buffer, the model **964** has an additional six video buffers and three z-buffers in 1024x1024 mode. In 1024x1280 mode there are two additional video buffers and one additional z-buffer.

In NTSC display mode, each pixel node has a single subscreen, regardless of configuration. The subscreen size in Model **916** is 180x122 = 21960 pixels, while in the **964**, it is one fourth as big. This means that a **916** can render full-screen NTSC images about as fast as a **964** can in high-resolution mode.

Table 1-4: Pixel Machine configurations

Model Number	Nodes		Peak Performance		Memory	Buffers		Bytes
	pipe	pixel	MIPS	MFLOPS	(Mbytes)	*rgb* α	z	per pixel
916z	0	16	80	160	12	2	1	12
916	9	16	125	250	12	2	1	12
916d	18	16	170	340	12	2	1	12
920z	0	20	100	200	15	2	1	12
920	9	20	145	290	15	2	1	12
920d	18	20	190	380	15	2	1	12
932z	0	32	160	320	24	4	2	24
932	9	32	205	410	24	4	2	24
932d	18	32	250	500	24	4	2	24
940z	0	40	200	400	20	4	2	24
940	9	40	245	490	30	4	2	24
940d	18	40	290	580	30	4	2	24
964z	0	64	320	640	48	8	4	48
964	9	64	365	730	48	8	4	48
964d	18	64	410	820	48	8	4	48
964X*						4**	2**	24

* 964 programmed to display 1280X1024 pixels.
** 18 Mbytes in additional partial buffers are available.

Pixel Machine Software

The distinct architectural components of the Pixel Machine are the host computer, the pipe nodes, and the pixel nodes. The host computer allows an application program to access the power and functionality of the Pixel Machine, the pipe nodes are responsible for the serial parts of algorithms, and the pixel nodes execute parallel algorithms. The following sections describe the software that supports each architectural component.

Host Software

A host-resident, C-callable library is responsible for command creation and transmission, invocation of subprocesses that monitor external events, and machine initialization and control.

Commands are the packets of data that the host sends to the Pixel Machine to request actions or to serve as data. Commands are discussed more completely in Chapter 2, "Writing Programs for the Host". Commands should not be confused with messages, which are requests that originate in the Pixel Machine and are directed to the host. Messages are explained in Chapter 3, "The DEVtools Message Service Protocol".

The primary functions provided by the host are:

- translating high-level function calls and macros into commands
- transmitting commands over the VMEbus to the Pixel Machine
- down-loading code to the pipe and pixel nodes and initializing them
- handling interactive functions (e.g., mouse/cursor interface)
- processing message requests received via the parallel I/O from the Pixel Machine processors

All commands are sent to the first node in the pipeline. Commands proceed serially down the pipe until the last node broadcasts them simultaneously to all of the pixel nodes. In systems without a pipeline, the host sends commands directly to the pixel node broadcast bus.

Pipe Node Software

The pipe nodes are typically used to implement a set of algorithms that act serially on a set of data. For example, a rendering and modeling application might use the pipeline to generate objects, apply modeling and viewing transformation, cull and shade the objects, apply projection transformations, do x, y, and z clipping, and finally, map the image to a viewport on the screen.

A useful analogy is to think of the pipe nodes as UNIX® system filters. Each node, like a UNIX system filter, reads some input, transforms the input, and writes some output.

The pipe program reads commands from the input FIFO. Each command consists of an opcode, the number of parameters, and the parameter list. When a command arrives at a pipe node, four actions can be triggered. The node can:

- forward the command to the next node in the pipeline,
- modify the parameter list and send it down the pipe
- process the command, possibly generating new commands,
- consume the command.

Each pipe node stores and executes routines that are invoked by command opcodes. In a typical polygonal rendering and modeling application, the pipe nodes perform geometric processing algorithms. For instance, three nodes could be assigned to clip the polygons in the x, y, and z planes, and one node to shade the polygon. In order to optimize the shading, however, the system could easily be re-configured to have three shading nodes and only one clipping node. Thus the pipeline can be optimized for any application through experimentation and new functions can be added as needed.

A Pixel Machine can contain no pipe, a single 9–node pipe, or 18 pipe nodes that be configured as either two parallel 9–node pipes or a single 18–node pipeline. In parallel mode, shown in a single pipeline are duplicated in both pipes.

Pixel Node Software

Pixel nodes implement algorithms that can be done in parallel, like the raster-scan conversion of points, lines, and polygons, image compositing, and ray tracing. Because the frame buffer memory is distributed through the pixel node array [Figure 1-6], all routines that access the frame buffer are implemented here as well. In the pixel node array, identical functions are usually replicated in each node.

The Distributed Frame Buffer

Programming the pixel nodes to access the interleaved frame buffer requires an understanding of two concepts:

1. an algebraic *domain* transformation that maps from a screen space coordinate system to a *processor* space coordinate system, and
2. techniques for rendering images in a *subscreen*, the small, contiguous frame buffer that is attached to each pixel node.

The domain transformation maps point from Cartesian (x,y) screen space to (i,j) processor space as follows:

$$i = \frac{1}{N_x}(x - O_x) \qquad j = \frac{1}{N_Y}(y - O_y)$$

where N_x and N_y are the number of processors per row and column, respectively, in the pixel node array, and are fixed for any given model of the Pixel Machine. O_x and O_y select a particular processor in the array, with O_x in the range $[0,N_x-1]$ and O_y varying between 0 and N_y-1.

The transformations from processor to screen space are:

$$x = i\ N_x + O_x \qquad y = j\ N_y + O_y$$

The effort required to parallelize an existing algorithm involves the restructuring of the algorithm so that it operates in the (i,j) processor space rather than screen space. Any algorithm that processes each pixel independently, such as fractal generation or ray tracing, requires very little modification, because no coherence is required from one pixel to the next. The number and complexity of modifications required increases with the degree of coherency between one pixel and the next or one scan line and the next. Writing a program so that it adheres to the domain transformation guarantees portability to single processor systems, where $N_x = N_y = 1$ and $O_x = O_y = 0$.

The pixel interleaving scheme presents an obstacle to applications that require a single pixel node to process and display a contiguous set of pixels. The serial I/O (SIO) capability of the pixel nodes provides a way to circumvent the problems created by interleaving. The set of pixels can be created in undisplayed memory and then routed, using SIO, to the pixel nodes that will display them.

The pixel nodes are arranged in an $n \times m$ array, and the processor in the ith row, jth column handles every nth pixel on every mth scan line (see Figure 1-6). Each processor addresses a portion of the frame buffer, which it sees as a contiguous subscreen. The coordinate system of the subscreen is called the *processor space*. DEVtools provides mapping functions from *(x,y)* screen space to *(i,j)* processor space:

i = PMilo(scrn, x) returns the smallest integer $i \geq \frac{x-O_x}{N_x}$

i = PMihi(scrn, x) returns the largest integer $i \leq \frac{x-O_x}{N_x}$

j = PMjlo(scrn, y) returns the smallest integer $j \geq \frac{y-O_y}{N_y}$

j = PMjhi(scrn, y) returns the largest integer $j \leq \frac{y-O_y}{N_y}$

Where O_x and O_y are the processor offsets in the x and y direction, respectively, and N_x and N_y are the numbers of processors in the x and y direction, respectively.

Because there are more pixels in screen space than in processor space, the mapping is not one–to–one. To ensure that the processor space pixel (i_1,j_1) is actually screen space pixel (x_1,y_1), the following condition must hold:

```
(PMilo(scrn,x1)==PMihi(scrn,x1)) && (PMjlo(scrn,y1)==PMjhi(scrn.y1))
```

Here is a simple example. The code segment shown in Example 1-1 draws a set of vertical and horizontal lines in a screen space viewport defined by xmin , xmax , ymin , and ymax .

```
for (x=xmin; x<xmax; x+=delta)
for (y=ymin, y<ymax; y++)
        PMputpix(scrn, x, y, RED);

for (y=ymin, y<ymax; y+=delta)
for (x=xmin; x<xmax; x++)
        PMputpix(scrn, x, y, GREEN);
```

Example 1-1. Line drawing in screen space.

This code segment can be converted into code that will run on a 964 by adding a conditional statement to test for the pixel's presence in the processor space of this node, as shown below (Example 1-2).

```
for (x=xmin; x<xmax; x+=delta)
for (y=ymin, y<ymax; y++)
if ((i=PMilo(scrn, x))==PMihi(scrn, x)&&(j=PMjlo(scrn, y))==PMjhi(scrn, y))
        PMputpix(scrn[0],i, j, RED);

for (y=ymin, y<ymax; y+=delta)
for (x=xmin; x<xmax; x++)
if ((i=PMilo(scrn, x))==PMihi(scrn, x)&&(j=PMjlo(scrn, y))==PMjhi(scrn, y))
        PMputpix(scrn[0],i, j, GREEN);
```

Example 1-2. Line drawing in processor space.

The pixel node code shown above is straightforward but inefficient. It iterates across screen space, and does the processor space mapping and testing for each pixel. A better method is to iterate over processor space, as shown in Example 1-3.

```
imin = PMilo(scrn, xmin);
imax = PMihi(scrn, xmax);
jmin = PMjlo(scrn, ymin);
jmax = PMjhi(scrn, ymax);

for (i=imin; i<=imax; i+=delta)
for (j=jmin, j<=jmax; j++)
        PMputpix(PMscrns[0],i, j, RED);

for (j=jmin, j<=jmax; j+=delta)
for (i=xmin; i<=imax; i++)
        PMputpix(PMscrns[0],i, j, GREEN);
```

Example 1-3. Efficient line drawing in processor space.

In the next section, a more complicated example of an algorithm that might be implemented in the Pixel Machine nodes is presented.

Visualizing Complex Functions

Fractal geometry is a branch of mathematics used to describe self-similar structures such as those observed in nature. The Julia set is a class of fractals in the complex plane. A generating function is evaluated at discrete points in a complex range until the function diverges or a maximum number of iterations is reached.

The generating function is a squaring function of the form:

$$Z_{n+1} = Z_n^2 + C = (X_n + Y_n \mathbf{i})^2 + (P + Q\mathbf{i})$$
$$= X_n^2 - Y_n^2 + 2X_n Y_n \mathbf{i} + P + Q\mathbf{i}$$
$$= X_n + 1 + Y_n + 1\mathbf{i}$$

where $X_{n+1} = X_n^2 - Y_n^2 + P \qquad Y_{n+1} = 2X_n Y_n + Q$

Different values of P and Q define different Julia sets. If $z = X_n^2 + Y_n^2$ is greater than a pre-specified limit, the function has diverged.

The Standard Implementation

The Julia set is displayed by mapping a rectangular region of the complex plane onto a raster display. The complex region is described by the real and imaginary ranges, *relo* to *rehi*, and *imlo* to *imhi*. The real axis is plotted in the x direction and the imaginary axis in the y direction.

The generating function is evaluated at discrete complex coordinates corresponding to each pixel in the viewport. The complex coordinates are defined by a linear mapping from screen space to complex space:

$$re = a_1 * x + b_1 \qquad im = a_2 * y + b_1$$

The algorithm loops over all pixels in the range [*xmin*,*xmax*], [*ymin*,*ymax*]. The generating function is iterated at each pixel, with initial values $X_0 = re$ and $Y_0 = im$. The iteration continues until the square of the magnitude of the generating function, z, diverges from the origin ($z \geq zmax$) or a pre-set number of iterations is reached ($n = nmax$).

If the function does not diverge within a limited number of iterations, the pixel color is based on the final z value. Otherwise, the intensity is based on the number of iterations performed.

Pixel Machine Implementation

The transformation from *(i,j)* processor space to *(x,y)* screen space is given by the equations:

$$x = i * N_x + O_x \qquad y = j * N_y + O_y$$

where N_x and N_y are the number of processors in x and y, respectively, and O_x and O_y are the offsets into the two dimensional processor array. Each processor loops over pixels in the range *imin* to *imax*, *jmin* to *jmax*. The library functions **PMilo(), PMihi(), PMjlo()**, and **PMjhi()** are used to map the given *(x,y)* limits into boundaries in *(i,j)* space for each processor.

Equations 1 and 2 provide the mapping from a given *(x,y)* screen coordinate to its corresponding *(re, im)* complex coordinates. Library functions **PMfxtoi()** and **PMfytoj()** transform a screen space equation into a processor space equation by modifying its coefficients. Equations 1 and 2 become:

$$re = a_1' * i + b_1' \qquad im = a_2' * j + b_2'$$

Once these initial transformations have been performed, the algorithm proceeds exactly as the sequential one does, except that each processor loops from *imin* to *imax* and *jmin* to *jmax*, as opposed to *xmin* to *xmax* and *ymin* to *ymax*. When a pixel value has been determined, its color is set using the function **PMputpix()**.

Example 1-4 shows the two implementations. Example 1-4(a) is the sequential algorithm, operating in screen space. Example 1-4(b) is the parallel algorithm, operating in processor space. The lines that differ are shown in boldface, illustrating the minimal changes which are required to adapt existing algorithms to the Pixel Machine.

```
a1 = (rehi - relo) / (xmax - xmin);
b1 = relo - a1*xmin;
a2 = (imhi - imlo) / (ymax - ymin);
b2 = imlo - a2*ymin;

for (y = ymin; y<=ymax; y++)
        for (x=xmin; x<=xmax; x++) {
            re = a1*x + b1;
            im = a2*y + b2;

            done = FALSE;
            for  n=0 ; n<nmax && !done ; n++) {
                 if ((z = re*re + im*im) <= zmax) {
                     temp_im = 2*re*im + Q;
                     re = re*re - im*im + P;
                     im = temp_im;
                 }
                 else done = TRUE;
            }

            if (done) write_pixel(x, y, value_based_on_n);
            else write_pixel(x, y, value_based_on_z);
        }
```

(a) The standard implementation.

```
a1 = (rehi - relo) / (xmax - xmin);
b1 = relo - a1*xmin;
a2 = (imhi - imlo) / (ymax - ymin);
b2 = imlo - a2*ymin;

imin = PMilo( scrn, xmin );
jmin = PMjlo( scrn, ymin );
imax = PMihi( scrn, xmax );
jmax = PMjhi( scrn, ymax );

PMfxtoi(scrn,a1, b1);
PMfytoj(scrn,a2, b2);

for (j=jmin; j<=jmax; j++)
         for (i=imin; i<=imax; i++) {
             re = a1*i + b1;
             im = a2*j + b2;

             done = FALSE;
             for (n = 0; n<maxn && !done; n++) {
                  if ((z = re*re + im*im) <= zmax) {
                      temp_im = 2*re*im + Q;
                      re = re*re - im*im + P;
                      im = temp_im;
                  }
                  else done = TRUE;
             }

             if (done) PMputpix(scrn, i, j, value_based_on_n);
            else PMputpix(scrn[0],i, j, value_based_on_z);
         }
```

(b) the Pixel Machine implementation.

Example 1-4. Fractal functions: Sequential and parallel implementations of the Julia set.

2 DEVtools Libraries

Header Files and Subroutine Libraries

Introduction

An application program for the Pixel Machine can be written to run in the host, the pipe nodes, the pixel nodes, or a combination of the three. The header files and libraries that provide useful definitions and functions for all three programming environments are discussed in this chapter.

Following is a list of header files that are used to compile programs for the host and for the Pixel Machine:

Table 2-1: Host and Pixel Machine Header Files

Host	Pixel Machine
devtools.h	pxm.h
devcommand.h	libmath.h
devimage.h	syscmd.h
deverror.h	pageregs.h
msgserve.h	model.h
sysmsg.h	pipe.h
crt0.h	pixel.h
pipe.h	sysmsg.h
pixel.h	

The first section describes writing host programs that can control and communicate with the Pixel Machine. The second section describes the library that contains Pixel Machine functions for the pipe and pixel nodes for the DSP32. The final section describes the libraries that contain DSP32 routines. These are general purpose routines, not limited to use on a Pixel Machine.

Documentation

All the functions for the host and Pixel Machine are described in the *DEVtools Reference Manual* and are included in the on-line manual pages. The DSP32 libraries are described in the *WE® DSP32* and *DSP32C C Language Compiler: Library Reference Manual.*

Getting Started with DEVtools

Introduction

To compile and link programs that use DEVtools you need to know where the header files, executable files, and libraries reside on your system. devtools, the DEVtools directory, can usually be found in the directory **hyper**. This directory usually resides in **/usr**, but it can be located elsewhere on your system; you may need to check with your system administrator. The examples in this section assume that the DEVtools directory is called **/usr/hyper/devtools.**

Setting up your Environment

The PXMtools software provides files in the **/usr/hyper** directory that can be used to initialize the execution search path and environment variables that you need to use the DEVtools software. The files are:

Name	Function
.hyper_profile	Profile file for Bourne shell and Korn shell
.hyper_env	Environment definition file for Korn shell
.hyper_login	Login initialization file for C shell
.hyper_cshrc	C shell startup file

To use the DEVtools software, your executable program search path list must include the required **hyper** directories, and must also contain:

Name	Function
/usr/hyper/devtools/bin	Contains DEVtools executables such as **devprint**
/usr/hyper/devtools/dsp32/bin	Contains the executables for the DSP32 Support Software Library

The environment variable **DSP32SL** must contain the pathname of the directory that contains the DSP32 Support Software Library, usually **/usr/hyper/devtools/dsp32.**

To access the DEVtools online manual pages, your MANPATH environment variable must include **/usr/hyper/devtools/man.**

Compiling DEVtools Programs for the Pixel Machine

Pixel Machine programs are compiled using the **devcc** command. **devcc** is similar to the **d3cc** command (described in the *DSP32 C Language Compiler User's Manual*), but it knows about the special files needed for compiling and linking Pixel Machine programs. These special files are the DEVtools library, include files, startup code, and the loader directive file. A typical command line used to compile a Pixel Machine program is:

```
devcc -c ctest.c
```

Additional information about **devcc** can be found on the manual page in the *DEVtools Reference Manual*, and more information on the DSP32 Support Software Library can be found in the DSP32 Support Software manuals:

DSP32 Software Support Library User's Manual
DSP32 C Language Compiler User's Manual
DSP32 C Language Compiler Library Reference Manual

Linking DEVtools Programs for the Pixel Machine

Pixel Machine programs may be linked using **devcc** or **d3ld**. After the Pixel Machine library has been specified, any of the DSP32 Support Software Library libraries can be specified. libc, libm, and libap may be specified using the **-lc**, **-lm** and **-lap** compiler or linker options.

devcc supplies the linker with the appropriate options to successfully link programs that run on the Pixel Machine. Following is a typical command line used to link a Pixel Machine program:

```
devcc -o ctest.dsp ctest.o
```

In the few cases where the loader must be called explicitly, the link command must also provide the following information:

- **/usr/hyper/devtools/lib/crt0_pixel.o** or **crt0_pipe.o**: the startup (crt0) file
- **/usr/hyper/devtools/include/pixel_ifile** or **/usr/hyper/devtools/include/pipe_ifile**: the memory usage definition file
- **/usr/hyper/devtools/lib/libpm.a**

Following is an example of how to use **d3ld** to link a Pixel machine program:

```
d3ld /usr/hyper/devtools/include/pixel_ifile\
/usr/hyper/devtools/lib/crt0_pixel.o\
ctest.o\
/usr/hyper/devtools/lib/libpm.a\
-lc\
-o ctest.dsp
```

The above command links a program that runs in a Pixel node. For Pipe node programs the startup file would be:

/usr/hyper/devtools/lib/crt0_pipe.o

and the loader directive file would be:

/usr/hyper/devtools/include/pipe_ifile

When you use **devcc**, pipe or pixel programs can be specified with the **-pipe** or **-pixel** command line options (**-pixel** is the default). With these options **devcc** specifies the appropriate files to **d3ld**. See the manual page for **devcc** in the *DEVtools Reference Manual* for more information.

Stack Configuration

The default action of the loader is to load the stack segment immediately after the text segment. Because the DSP C compiler grows the stack from low memory to high memory, the stack is allowed to grow to fill all available memory in bank0.

The stack section is set up differently in the pipe and pixel nodes as explained below.

Pipe Node Stack

In a pipe node, the stack is set to be a minimum of 320 bytes long. The loader exits with an error if there is not enough room for the minimum stack. It is possible to change the default minimum to something less than 320 bytes, but it is the user's responsibility to make sure that the new minimum is sufficient.

To change the minimum stack size it is necessary to assemble and link a new **stack.o** with your program to replace the one that is automatically loaded from **libpm.a**. A sample **stack.s** that should be used can be found in **/usr/hyper/devtools/lib/stack.s**. To change the size of the stack, copy this file to another directory and assemble defining PM_STACK_SZ to the desired stack size, and PIPE. For example, to change the stack size to 2048 bytes use:

```
d3as -DPM_STACK_SZ=1024 -DPIPE stack.s
```

The **stack.o** that is produced should then be linked with the user program. Be sure to include it before **libpm.a**. The stack size should be a multiple of 4.

Pixel Node Stack

In the pixel nodes the stack is set up differently. Because there is a relatively large initialization function (**_initpixel()**) that is called once and only once before **main()**, it is loaded above the stack in a section called **.init**, so that the stack can grow into it and reuse the space. The stack itself is initially given only 30 bytes, but the initialization code affords approximately another 1.5 kilobytes. Again, the loader will produce an error if this minimum stack does not fit.

The user is still able to enlarge the stack, but should not reduce it any further. The same technique that is used for the pipe nodes is used here except that **PIXEL** needs to be defined. For example:

```
d3as -DPM_STACK_SZ=1024 -DPIXEL stack.s
```

Compiling DEVtools Programs for the Host System

Host programs are compiled using the cc command. To locate the header files, the directories that contain the Pixel Machine header files and the DEVtools header files must be specified on the cc command line. The cc command line should include the following options:

```
-I/usr/hyper/devtools/include
```

A typical command line used to compile a Pixel Machine program is:

```
cc -c -I/usr/hyper/devtools/include host.c
```

Linking DEVtools Programs for the Host System

Host programs should be linked using the cc command. The name of the Pixel Machine DEVtools host library (**devlib.a**) must be included on the cc command line.

A typical command line used to link a host program is:

```
cc -o host host.o /usr/hyper/devtools/lib/devlib.a
```

Versions of **devlib.a** are provided that support the Sun floating point accelerator (fpa), or support profiling, or support both fpa and profiling. Fpa support is only provided for the Sun 3 libraries. The names of the libraries are:

Library	Supports
devlib.a	Any host, without profiling
devlib_p.a	Any host, with profiling
devlib_ffpa.a	Uses fpa, does not include profiling code (Sun 3 only)
devlib_ffpa_p.a	Uses fpa, includes profiling code (Sun 3 only)

Sample Programs

The DEVtools package includes a set of sample programs that illustrate the use of DEVtools for a variety of applications. The sample programs are located in **/usr/hyper/devtools/sample/misc**. The sample directory contains the directories:

Directory	Contains
bin	host executable programs
boot	Pixel Machine executable programs and shell scripts to run the sample programs
host	host source files
include	host and Pixel Machine include files for sample programs
pipe	pipe node source files
pixel	pixel node source files

The host, pipe, and pixel directories each contain source code and a **makefile**. These files provide a good place to look for efficient usage of many Pixel Machine functions. The **makefile** can be used to generate the executable versions of the sample programs, and is a good guide for constructing **makefiles** for your own programs.

The following lists the shell scripts that can be invoked to run the sample programs, and describes the functions illustrated by the program.

- **Circle**: A simple program to draw a large circle on the screen. A simple example of the use of subscreen information.
- **Colors**: A very simple program. Clears the screen red then to grey. Uses **PMapply()** to replicate a function for each subscreen.
- **Copies**: Make multiple, overlapping copies of upper the left hand corner using the VRAM and ZRAM copy routines.

- **Dataflow**: Generates commands in one of the pipe nodes. These commands are passed down the pipe into the pixel nodes. A host program (**devprint**) is used to output print commands from the pixel nodes. Shows the use of command processing functions and the use of **printf**.
- **Fastpixels**: Demonstrate an efficient way to fill contiguous pixels.
- **Hello**: Clears the screen and uses **printf**. Prints the node ID.
- **Julia**: Displays and animates julia set fractals. Uses subscreens and the **PMilo()/PMihi()** and **PMjlo()/PMjhi()** macros, as well as double buffering. Shows the processing power of the Pixel Machine.
- **Led**: Turns off the **vsync** and **psync** LEDs on the pixel processor boards. Only interesting if the cover of the machine is removed.
- **Lights**: Flashes **vsync** and **psync** LEDs on the pixel processor boards. Only interesting if the cover of the machine is removed.
- **Mand**: Mandelbrot set; another fractal.
- **Math**: Uses a number of math library functions.
- **NTSC**: Displays colored bars on the screen. Can be used with any type of display.
- **Pipes**: Shows how to pass data from one pipe node to the next.
- **Pong**: A sample animation of a bouncing ball.
- **Pxmclear**: Clear the front and back buffers to black.
- **Qcopies**: Use fast ZRAM copy to replicate image in front buffer.
- **Send**: A host program and associated pixel node programs. Implements a user-message handling routine on the host to route messages from one node to any other node.
- **Shift**: Moves pixels around the screen using serial I/O.
- **Texture**: Generates a random texture.
- **Zstuff**: Uses a host program to set the contents of the Z memory and a pixel node program to read the contents from Z memory.
- **Ztest**: Sample use ZRAM allocation routines.

Setting-up the Pixel Machine for DEVtools

Introduction

The operation of the Pixel Machine is controlled by the host system to which it is attached. Running a program on the Pixel Machine requires the use of a group of system control commands that perform functions such as resetting the Pixel Machine processors, loading programs into the memory of the processors, displaying status information, etc. More detailed information about the system control commands can be found in the PXMtools manual pages.

How to Run a Program on the Pixel Machine

The execution of Pixel Machine programs is typically controlled by a program executing on the host system. Some simple programs that require no interaction with the host can be run without a host program through the use of the **hypload** and **hyprun** commands.

Host programs provide:

- a simple mechanism to ensure that the proper programs are loaded into the pipe and pixel nodes
- a convenient method of controlling the Pixel Machine
- a message passing protocol that allows a user program running on the Pixel Machine to signal the host program. This feature can be used to send data to the host, request data from the host, and to perform any other tasks that the nodes cannot perform by themselves.
- the ability for the Pixel Machine to output information on the host using the DEVtools **printf** routine
- other control functions that must be performed by the host such as selecting the serial I/O direction.

The host program should begin by calling **DEVinit**. This opens the Pixel Machine and resets all of the processors in the Pixel Machine. Before exiting the host program, **DEVexit** should be called to release the Pixel Machine so that it can be accessed by other users.

Programs that require no communication with the host can be loaded and started by the **hypload** and **hyprun** commands.

Following is an example of the commands used to run a program that does not require any host communication on all of the pixel nodes:

```
hyplock
hypload -dall prog.dsp
hyprun -dall
hypfree
```

hyplock should be used to lock the Pixel Machine before running programs that are not controlled by a host program to prevent other users from accessing the machine while the current program is running. After the program has finished, executing **hypfree** makes the machine available for other users. When the Pixel Machine is controlled by a host program, **hyplock** and **hypfree** are not needed

Refer to the PXMtools manual pages for more information about these programs.

Writing Programs for the Host

Host programs are written in C and compiled, linked, and loaded with the standard compiler, libraries, and header files. In addition, there are libraries and header files of special functions and definitions used to control and communicate with the Pixel Machine.

The devlib Library

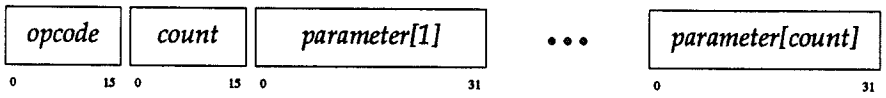
devlib includes functions for sending commands to the pipe and pixel nodes and for writing a message server to respond to messages from the nodes. **host/devtools.h** should be included in all programs that use devlib. **host/msgserve.h** is required if the message polling functions of DEVtools are used. **host/devcommand.h**, supplies macros for sending properly formatted messages to the pixel and pipe nodes.

Host programs tend to focus on sending and receiving commands from the Pixel Machine. This section will describe the format of a command and the routines for reading and writing them, and then discuss a message server program on the host and how it might be extended to handle user-defined messages.

Commands

Commands are made up of an opcode, a parameter count, and a parameter list, as shown in Figure 2-1.

Figure 2-1: Command format



Commands are generated on the host and written to the pipe node FIFOs using the **DEVcwrite** macros. **DEVcommand** is used to encode an opcode and parameter count into a properly formatted 32-bit value that can be passed to the **DEVcwrite** macros. There are twelve **DEVcwrite** macros, each handling a different number of parameters.

DEVcwrite0(DEVcommand(*opcode***, 0));**
DEVcwrite1(DEVcommand(*opcode***, 1),** *type, arg1*);
DEVcwrite2(DEVcommand(*opcode***, 2),** *type, arg1...arg2*);
DEVcwrite3(DEVcommand(*opcode***, 3),** *type, arg1...arg3*);
DEVcwrite4(DEVcommand(*opcode***, 4),** *type, arg1...arg4*);
DEVcwrite5(DEVcommand(*opcode***, 5),** *type, arg1...arg5*);
DEVcwrite6(DEVcommand(*opcode***, 6),** *type, arg1...arg6*);
DEVcwrite7(DEVcommand(*opcode***, 7),** *type, arg1...arg7*);
DEVcwrite8(DEVcommand(*opcode***, 8),** *type, arg1...arg8*);
DEVcwrite9(DEVcommand(*opcode***, 9),** *type, arg1...arg9*);
DEVcwrite10(DEVcommand(*opcode***, 10),** *type, arg1...arg10*);

DEVcwriten(DEVcommand(*opcode, count***),** *type, arg_array, count*);

Commands with ten or fewer arguments are assembled more efficiently by the count-specific **DEVcwrite** macro. Arguments can be either integer floating point or another type that represents a properly aligned 32-bit value, but all arguments to a command must have the same type. The *type* parameter to the **DEVcwrite** macros is a type name, either **int**, **float** or other type name. **DEVwrite** macros are similar to **DEVcwrite** macros but they do not include command arguments.

There is another set of twelve macros for writing commands to the second pipeline whenever a system with eighteen pipe nodes is configured with two parallel pipelines. They are identical in form and function to the macros presented above, except that **_alt** is appended to the macro name (*e.g.*, **DEVcwrite0_alt**).

Four macros for reading commands are also defined in **devcommand.h**: **DEVcread()** and **DEVcread_alt()** read a command and parameter count, and **DEVread()** and **DEVread_alt()** read the arguments. These are used for reading the feedback FIFOs.

High Level Functions

Table 2-2 shows the routines that make up the high level devlib routines that are most commonly used by host programs. **DEVinit()** and **DEVexit()** are the recommended ways to start and finish host programs. The other routines are part of the message passing support, and many of them will be used in the host program that is described in the next few paragraphs.

Table 2-2: High Level Functions

Name	Function
DEVexit()	halt processors and close Pixel Machine device
DEVget_scan_line()	upload an image or a portion of an image to a Pixel Machine
DEVinit()	open and initialize Pixel Machine device
DEVpipe_boot()	load a DSP executable into the specified pipe nodes
DEVpixel_boot()	load a DSP executable into the specified pixel nodes
DEVpoll_nodes()	polls DSP processors for messages
DEVput_scan_line()	download an image or a portion of an image to a Pixel Machine
DEVrun()	begins execution of the current Pixel Machine program
DEVswap_pipe()	reverses the rolls of the primary and secondary pipes
DEVuser_msg_enable()	define a message code and associated functions
DEVwait_exit()	wait for pixel nodes to signal completion, then call **DEVexit()**

The host program performs certain functions on behalf of the Pixel Machine. These functions include initializing the system, loading programs into the nodes, beginning execution, and servicing message requests from the Pixel Machine. Message requests are used to perform other actions such as input/output (I/O) operations, controlling serial I/O, etc. Following is a sample host program:

```
#include <stdio.h>
#include <host/devtools.h>
main()
{
DEVpixel_system *pixel_system;

if ((pixel_system = DEVinit()) == NULL) {
        fprintf(stderr, "Open of Pixel Machine failed.");
        exit(1);
}

/* Load all of the pipes with "pipe.dsp". */
DEVpipe_boot(pixel_system, "pipe.dsp", 0,
        DEVlast_pipe(pixel_system), NULL, DEV_BOOT_CHECK_TIME);

/* Load all of the pixels with "pixel.dsp". */
DEVpixel_boot(pixel_system, "pixel.dsp", 0,
        DEVlast_pixel(pixel_system), NULL, DEV_BOOT_CHECK_TIME);

/* Begin execution */
DEVrun(pixel_system);

/* Poll the nodes for message requests.  DEVpoll_nodes returns when a
          "host exit" message is received from a node. */

DEVpoll_nodes(pixel_system, 0, DEVlast_pipe(pixel_system),
        0, DEVlast_pixel(pixel_system), DEV_FOREVER, DEV_NONE);

/* Close the Pixel Machine. */
DEVexit();
}
```

Example 2-1. Sample Host Program

To customize the host program to receive application-specific messages, calls to **DEVuser_msg_enable()** can be inserted after **DEVinit()** and before the polling loop. Each user message has a unique opcode in the range (0,DEV_HIGHEST_USER_MESSAGE), defined in **host/msgserve.h**, and specifies two functions. The first routine is called if the message is received from a pipe node, and the second one is used when a pixel node sends the message. See the manual page for **DEVuser_msg_enable()** for more details.

Low Level Functions

Table 2-3 contains low level system control and I/O functions.

Table 2-3: Control and I/O Functions

Name	Function
DEVclose()	disconnect the host program from the Pixel Machine device
DEVfifo_parallel()	configure 18 pipe nodes as two parallel pipes
DEVfifo_read()	read 4 bytes from a pipe node FIFO
DEVfifo_reset()	reset all FIFOs on a board
DEVfifo_serial()	configure 18 pipe nodes as one long serial pipeline
DEVfifo_write()	write 4 bytes to a pipe node FIFO
DEVget_color_map()	fetch the contents of the color tables
DEVget_pixel()	read a pixel from the frame buffer
DEVload_color_tables()	load the color tables using a gamma correction table
DEVload_linear_ramp()	load the color tables with a linear table (no gamma correction)
DEVlock()	manage Pixel Machine locks
DEVopen()	connect a user program to a Pixel Machine
DEVopen_system()	allow system to be opened without resetting config information
DEVpipe_enable_error_halt()	set DSP to halt on hardware errors
DEVpipe_get()	read data from a pipe node
DEVpipe_get_msg()	read data from a pipe node
DEVpipe_get_pir()	read the PIR register in a pipe node
DEVpipe_halt()	halt a pipe node
DEVpipe_id_check()	verify a pipe node's identification block
DEVpipe_id_print()	print a pipe node's identification block on **stdout**
DEVpipe_id_write()	write a pipe node identification block into memory
DEVpipe_put()	send a block of data to a pipe node
DEVpipe_read()	read data from a pipe node
DEVpipe_run()	initialize and start a pipe node
DEVpipe_write()	DMA a buffer of data to a pipe node
DEVpixel_buffer()	select one of the two frame buffers for display
DEVpixel_enable_error_halt()	set DSP to halt on hardware errors
DEVpixel_get()	read data from a pixel node
DEVpixel_get_msg()	read data from a pixel node
DEVpixel_get_pir()	read the PIR register in a pixel node
DEVpixel_halt()	halt a pixel node
DEVpixel_id_check()	verify a pixel node's identification block
DEVpixel_id_print()	print a pixel node's identification block on **stdout**
DEVpixel_id_write()	write a pixel node identification block into memory
DEVpixel_mode_init()	initialize pixel mode register
DEVpixel_mode_overlay()	set overlay mode in the pixel mode register

Table 2-3: Control and I/O Functions (continued)

DEVpixel_mode_serial()	set serial I/O direction in the drawing mode register
DEVpixel_overlay()	set overlay mode in the pixel nodes
DEVpixel_put()	send a block of data to a pixel node
DEVpixel_read()	read data from a pixel node
DEVpixel_run()	initialize and start a pixel node
DEVpixel_start()	start a program running in a pixel node
DEVpixel_write()	DMA a buffer of data to a pixel node
DEVput_color_map()	update contents of the color tables
DEVput_pixel()	write a pixel value into the frame buffer
DEVread_z()	read from the z-buffer memory of a pixel node
DEVserial_direction()	update serial I/O link direction
DEVsservershadow_off()	turn off shadow palate update to allow color tables to be updated
DEVshadow_on()	turn on shadow palate update after color tables have been updated
DEVunit()	return the value of HYPER_UNIT environment variable
DEVwrite_z()	write to the z-buffer memory of a pixel node

System Status Tracking

Pixel Machines are frequently used by a number of people, each of whom can be running a different application, and possibly even using different libraries. For example, a single system may be used for applications written using PIClib, RAYlib, and DEVtools. Furthermore, several DEVtools users may each require different DSP code to be loaded in the Pixel Machine.

Even if all of the users of a Pixel Machine are running a single application, there still can be differences in configurations based on pipe modes (parallel vs. serial), video format (hi-res vs. NTSC) and other configuration parameters.

All Pixel Machine software maintains a file that reflects the current status of the Pixel Machine. This status information includes the:

- number of pipe nodes
- number of pixel nodes
- current pipe mode (serial, parallel)
- current video format (hi-res, NTSC, PAL)

- current gamma correction mode
- current video options (sync source, etc.)
- pathname and modification time of the executable file loaded into each pipe and pixel node

When a program is invoked that uses the Pixel Machine, the current state of the machine is compared with the configuration parameters specified by the user's environment (HYPER_MODEL, HYPER_PIPE, etc.) The DSP executables required for the user's application (as specified by **DEVpipe_boot** and **DEVpixel_boot**, or as implicitly specified by library functions such as **PICinit**) are compared with those currently loaded. If the appropriate software is not already present in the machine, it will automatically be loaded.

Users that are developing DSP software can request that the file modification times of the executable files be compared with those of the files currently loaded in the machine. This allows new versions of files to be loaded automatically.

When a file is loaded into a Pixel Machine node, a checksum value is computed based on the pathname of the file and the process ID of the process performing the load operation. Subsequently, when another program checks whether the correct files are loaded, it first compares the pathname of the desired file with the pathname of the loaded file (relative pathnames are converted to absolute pathnames by prepending the current directory name). If the file names match, the modification times are compared (if this option has been selected). Finally, the checksum value stored in the node's memory is compared with the value in the status file. If the checksums match, the program is not reloaded. The checksum is a safeguard to ensure that the system can not be fooled by a corrupted status file or by turning off the Pixel Machine.

The status file is read by **DEVinit()** and written by **DEVexit()**. The status information is maintained in memory during the execution of the host program. During execution, the disk copy of the status file is marked as invalid. As a result, executing a command that checks the status file (**hypid**, for example) will result in a `node checksum does not match` message.

Writing Programs for the Pipe and Pixel Nodes

libpm

libpm is the library that supplies subroutines for use in both the pipe and pixel nodes. All programs that use libpm must include the header file **pxm.h**

Both pipe and pixel nodes use the command data structure as defined in Figure 2-2 and **pxm.h**. Pipe nodes read commands from their input FIFO in two stages, first the opcode and argument count (using **PMgetop()**) and then the arguments themselves (via **PMgetdata()**). Pixel nodes read all three command components at once by calling **PMgetcmd()**.

Figure 2-2: PMcommand() data structure

```
#include "pxm.h"

typedef struct {
        short opcode;
        short count;
        float *data_ptr;
} PMcmdtype

extern PMcmdtype PMcommand;
```

Pipe nodes write commands to their output FIFO by calling either **PMputcmd()**, which writes an entire command, or **PMputop()** followed by **PMputdata()** if the argument count is greater than zero. Pixel nodes cannot send commands.

Functions for Pipe and Pixel Nodes

This section describes routines that are useful in both pipe and pixel node programs.

A global variable, **PMsem**, is used as a semaphore by the host and node to synchronize DMA accesses. **PMsetsem()** and **PMwaitsem()** are the synchronizing primitives that set and test the semaphore.

Recall that each node has a PIR register, written by the node and read by the host. **PMoutpir()** is the function that writes a value into the register. If necessary, it waits until the previous value has been read by the host. The PIR register is also written by the **PMusermsg()** routine, which sends a user-defined message to the host.

Table 2-4: Pipe and Pixel Node Functions

Name	Function
PMcolor_float()	converts internal color value to floating point number
PMcolor_int()	converts internal color value to an integer
PMdelay()	do nothing for a specified time
PMenable()	enable processing of selected system commands
PMfloat_color()	converts floating point value to internal color value
PMhost_exit()	signal **DEVpoll_nodes** to return to caller
PMint_color()	converts an integer to an internal color value
PMoutpir()	output a value to the PIR register
PMsetsem()	set the semaphore
PMusermsg()	send a user defined message to the host
PMwaitsem()	wait for semaphore to clear
printf()	formatted output conversion on host

Pipe Node Functions

This section describes routines that can only be used in pipe nodes. Most of them concern reading commands from and writing commands to the FIFOs.

Table 2-5: Pipe Functions

Name	Function
PMbus_wait()	waits until control of the broadcast bus is granted
PMcopycmd()	copy opcode, parameter count, and data from input to output FIFO of a pipe node
PMfb_off()	direct output commands to the regular output FIFO
PMfb_on()	direct output commands to the feedback FIFO
PMgetdata()	get data from a pipe node FIFO
PMgetop()	get opcode and parameter count from input FIFO of a pipe node
PMputcmd()	write opcode, parameter count, and parameters to the output FIFO of a pipe node
PMputdata()	write parameters to the output FIFO of a pipe node
PMputop()	write opcode and parameter count to the output FIFO of a pipe node
PMswap_pipe()	release the broadcast bus and request it again

Pixel Node Functions

This section describes functions that are only useful in pixel nodes. Most of them are used to read and write values in the various pixel nodes memories.

pxm.h includes macro definitions that convert (x,y) screen space coordinates to (i,j) processor space coordinates.

- **PMilo**(*subscreen,x*) returns the smallest processor space integer that, when mapped to screen space, will be $\geq x$.
- **PMihi**(*subscreen,x*) returns the largest processor space integer that, when mapped to screen space, will be $\leq x$.
- **PMjlo**(*subscreen,y*) returns the smallest processor space integer that, when mapped to screen space, will be $\geq y$.
- **PMjhi**(*subscreen,y*) returns the largest processor space integer that, when mapped to screen space, will be $\leq y$.

Figure 2-3 shows some examples of these mappings. Figure 2-3(a) shows an 8×8 corner of the screen, with some pixels turned on. Figure 2-3(b) shows the pixel to processor mapping for a 4×4 pixel node mesh. Each pixel location is tagged with the number of the processor which keeps that pixel in its subscreen memory. Figure 2-3(c) shows the *(i,j)* values for the pixels and Figure 2-3(d) the individual subscreens. Table 2-6 shows the values associated with calls on the four macros used to map *(x,y)* values into *(i,j)* values. Whenever **PMilo(subscreen,x)==PMihi(subscreen,x)** and **PMjlo(subscreen,y)==PMjhi(subscreen,y)**, then the pixel (x,y) is part of the issuing processor's subscreen. (Macros **PMmyx** and **PMmyy** can be used to abbreviate the equality tests.) The table shows pixel ownership by using boldface for the values that satisfy the condition.

Figure 2-3: Screen to processor space mapping functions

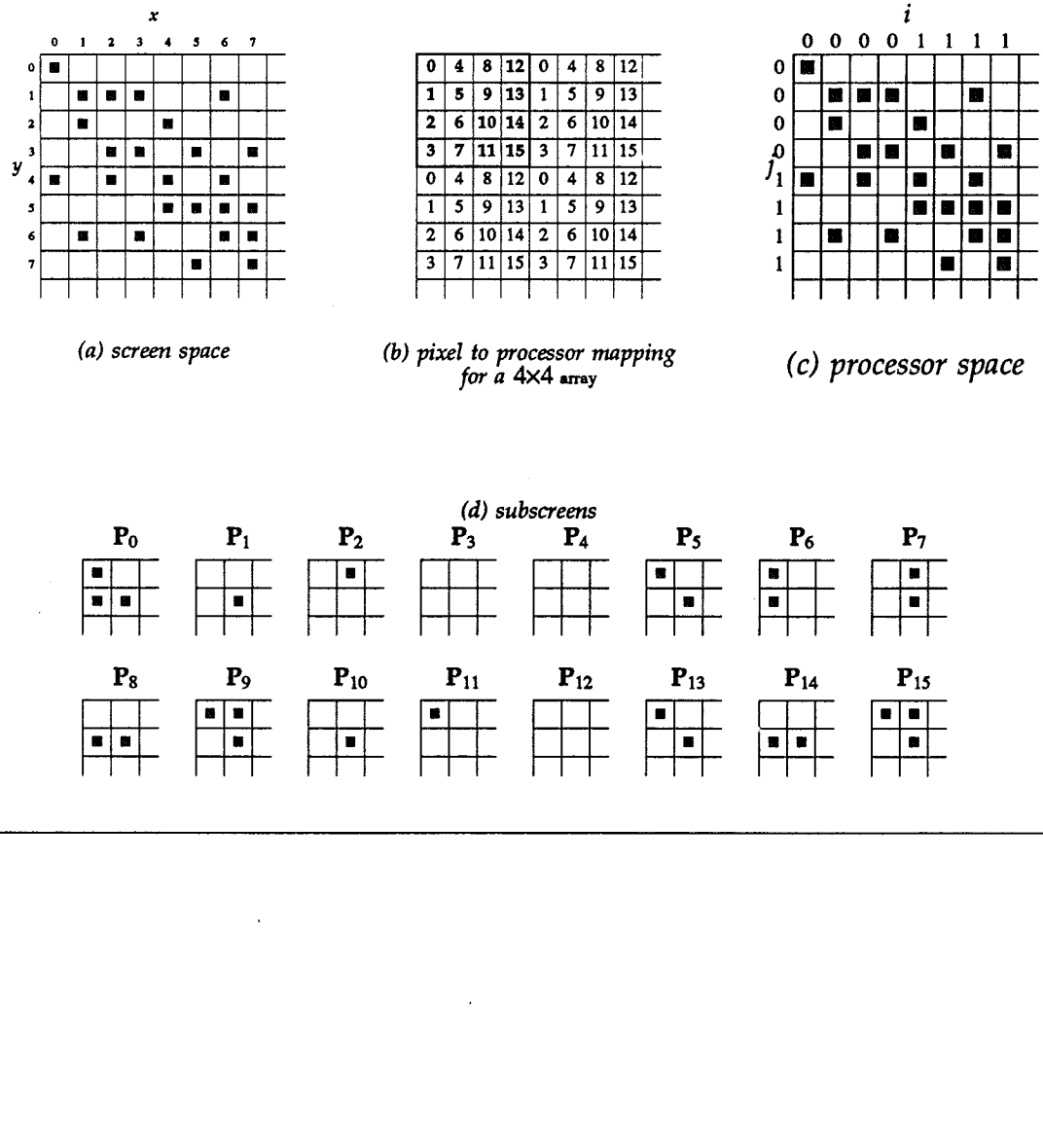


Table 2-6: Converting screen space coordinates into processor space

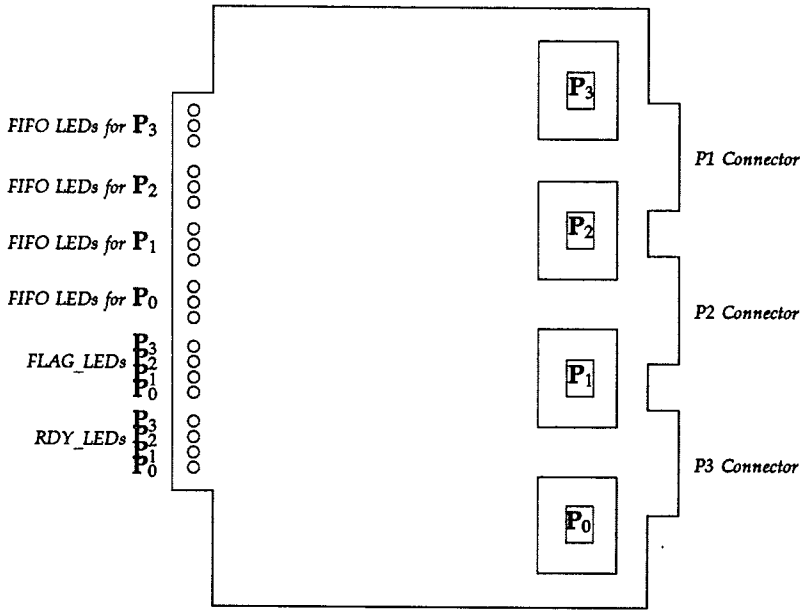
screen pixel	*mapping macro*	*processor doing the mapping* 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(3,0)	ILO(3)	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
	IHI(3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	JLO(0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	JHI(0)	0	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1	0	-1	-1	-1
(4,2)	ILO(4)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	IHI(4)	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	JLO(2)	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
	JHI(2)	0	0	0	-1	0	0	0	-1	0	0	0	-1	0	0	0	-1
(1,5)	ILO(1)	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
	IHI(1)	0	0	0	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1
	JLO(5)	2	1	1	1	2	1	1	1	2	1	1	1	2	1	1	1
	JHI(5)	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0
(6,6)	ILO(6)	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1
	IHI(6)	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
	JLO(6)	2	2	1	1	2	2	1	1	2	2	1	1	2	2	1	1
	JHI(6)	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0

It is also possible to reverse the mapping. That is, the **PMxat** and **PMyat** macros can be used in a pixel node program to convert *(i,j)* into *(x,y)* coordinates.

There are three macros that map the coefficients in linear expressions from screen space to processor space.

- **PMfxtoi(subscreen,A,B)** converts an expression of the form $Ax+B$ to one of the form $A'i+B'$. The macro modifies the values of A and B.
- **PMfytofj(subscreen,A,B)** converts an expression of the form $Ay+B$ to one of the form $A'j+B'$. The macro modifies the values of A and B.
- **PMfxytofij(subscreen,A,B,C)** converts an expression of the form $Ax+By+C$ to one of the form $A'i+B'j+C'$. The macro modifies the values of A, B, and C.

Figure 2-4: LEDs on the pixel node boards



Each pixel machine has FLAG and READY signals that are connected to LEDs on the pixel node processor board (see Figure 2-5). FLAG is used by **PMpsync()** and READY is used by **PMvsync()** and **PMrdyoff()** . They can also be set by the user (whenever the sync routines are not required by the program) with the **PMrdy_led()** and **PMflagled()** routines in libpm. The signals and LED displays can be useful for debugging to identify states in the program.

Note that the LEDs are inverted logic. That is, when the signal is off, the light is on. When the signal is on, the light is off.

Table 2-7: Pixel Node Functions

Name	Function
PMapply	apply a function to all subscreens
PMclear	fill a rectangular region of the screen
PMcopy_f	fast but dangerous 32 bit D/VRAM copy
PMcopy_s	safe 32 bit DRAM or VRAM copy
PMcopy_v	32 bit copy with variable increments
PMcopyftob	copy front to back
PMcopyvtov	copy blocks of VRAM
PMcopyvtoz	copy video RAM to DRAM
PMcopyztov	copy DRAM to video RAM
PMcopyztoz	copy from one section of DRAM to another
PMdblbuff	enable double buffering mode
PMflagled	turn the DEV_FLAG LED on or off
PMfreezaddr	decrement references to a page register
PMfxtoi	map a linear function of x from screen space to processor space i
PMfxytoij	map a linear function of x and y from screen space to processor space i and j
PMfytoj	map a linear function of y from screen space to processor space j
PMgetcmd	read command from a pixel node FIFO
PMgetpix	read a pixel from the current buffer
PMgetrow, PMgetcol	read a scanline from pixel memory without subscreens
PMgetscan	read a scanline from pixel memory
PMgetzaddr	assign address to a section of DRAM
PMgetzbuf	read a float value from the Z buffer
PMgetzdesc	allocate DRAM
PMihi	map from screen space(xmax) to processor space (ihi)
PMilo	map from screen space(xmin) to processor space (ilo)
PMinterleave	interleave or deinterleave a block
PMjhi	map from screen space(ymax) to processor space (jhi)
PMjlo	map from screen space(ymin) to processor space (jlo)
PMmsg_exchange	send and receive data packet over serial links
PMmsg_setup	set serial DMA input pointer
PMmyx	test if a given screen space coordinate is in processor space
PMmyy	test if a given screen space coordinate is in processor space
PMpagereg	macros to manipulate page registers used to access video and Z memory
PMpixaddr	generate a pointer to a specific pixel
PMpsync	wait for all pixel processors to synchronize
PMputpix	output a pixel to the current buffer
PMputrow, PMputcol	read a scanline from pixel memory without subscreens
PMputscan	write a scanline to pixel memory
PMputzbuf	write a float value to the z-buffer
PMqcopyztoz	copy from one section of DRAM to another

Table 2-7: Pixel Node Functions (continued)

PMqget	quick read of a pixel from the current buffer
PMqput	quick write of a pixel to the current buffer
PMqzget	quick read of Z value from the Z buffer
PMqzput	quick write of Z value to the Z buffer
PMrdyled	turn the DEV_RDY_LED on or off
PMrdyoff	turn the ready signal off
PMsiodir	set serial I/O link direction
PMsioinit	initialize serial I/O
PMsnglbuff	disable double buffering mode
PMswapbuff	swap visible and pixel buffers
PMv0get	read a pixel from buffer 0
PMv0put	write a pixel to buffer 0
PMv1get	read a pixel from buffer 1
PMv1put	write a pixel to buffer 1
PMvsync	synchronize and wait for vertical retrace
PMxat	map subscreen coordinates to screen space
PMyat	map subscreen coordinates to screen space
PMzaddr	generate a ZRAM pointer to a row
PMzaddrcol	generate a ZRAM pointer to a column
PMzbrk	initialize DRAM for allocation
PMzget	read a float from the z-buffer
PMzput	write a float to the z-buffer

Pixel Machine Math Functions

This section describes the hand-optimized assembly language versions of a few mathematical subroutines that are included in libpm (see Table 2-8). These routines have been implemented for the architecture and requirements of the Pixel Machine and will run more efficiently than similar routines in the other DSP32 libraries. Some of these routines, however, are more restrictive than the other DSP32 libraries; please see the manual pages in the *DEVtools Reference Manual* for further information.

Table 2-8: Math Functions

Name	Function
PMcos	cosine
PMieee_dsp	convert IEEE floating point number to DSP format
PMldot	specialized dot product for light sources
PMlong_dsp	convert **long integer** to **float**
PMnorm	normalize a vector and return its length
PMpow	power function
PMsin	sine
PMsqrt	square root function
PMx_exp_n	integer power function

Writing Programs for the DSP32

The *DSP32 C Language Compiler Library Reference Manual* describes three libraries that contain routines that can be included in Pixel Machine programs.

The libc Library

libc is a subset of the standard UNIX system C library and includes functions that support error handling and debugging. Table 2-8 lists the routines and gives a brief synopsis.

libc is needed even if none of its functions are called explicitly because the compiler needs it for some operations, for example, cases, mod and integer divide. Some of the routines require header files, including math.h, memory.h, stdio.h and string.h. Refer to the individual manual pages in the *DSP32 C Language Compiler Library Reference Manual* for more information.

Table 2-9: DSP32 libc

Name	Function
ecvt	convert a floating point number to a string
isalnum, isalpha, isascii, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit	classify characters
frexp	separate mantissa and exponent in a floating point number
ldexp	combine mantissa and exponent into a floating point number
memchr	find first occurrence of a character in a block of memory
memcpy	copy a block of memory
modf	separate mantissa and exponent in a floating point number
perror	print system error messages (only for use with the d3sim simulator)
printf	print formatted output (only for use with the d3sim simulator) (the libpm version of printf should be used for Pixel Machine programs)
strlen	return the length of a string

The libm Library

libm provides all of the functions found in the standard UNIX system math library. Table 2-10 lists and briefly describes them. Note that these libraries do error checking and are *not* optimized for the DSP32; therefore they are rather slow. Whenever possible, use an alternative function from libap or libpm.

To use the libm routines, include **math.h** and load the program with **-lm**.

Table 2-10: DSP32 libm

Name	Function
acos	arccosine
asin	arcsine
atan, atan2	arctangent
ceil	ceiling function
cos, qcos	cosine
erf, erfc	error functions
exp	exponential
fabs	absolute value
floor	floor function
fmod	remainder function
gamma	Gamma function
hypot	Euclidian distance
j0, j1, jn	Bessel functions
log, log10	logarithms
matherr	error handling
pow	power function
sin, sinh	sine
sqrt	square root
tan, tanh	tangent
y0, y1, yn	Bessel functions

The libap Library

libap is a library of routines that have been written and optimized for the DSP32 processor, and includes functions for mathematics, matrix manipulation, filtering, and imaging. The routines are listed in Table 2-10.

To use the libap routines, include libap.h and load the program with **-lap**. Some of the mathematical functions appear in both the math library and the applications library. The routines in libap have been hand-optimized for the DSP32 and will run faster than the libm version. The *DSP32C Language Compiler Library Reference Manual* contains a section describing how to use both libraries in the same program.

Table 2-11: DSP32 libap

Name	Function
acos, qacos	arccosine
alog10, alog2, aloge	anti-logarithms
asin, qasin	arcsine
atan, qatan	arctangent
cos, qcos	cosine
div, divf	quotient
dsp32	convert from IEEE to DSP floating point format
ieee32	convert from DSP to IEEE floating point format
inv, invf	inverse
invsqr	inverse of the square root
log10, qlog10, log2, loge	logarithms
ran, gran	random number generators
sin, qsin	sine
sqrt, qsqrt, sqrtf, sqrtq	square root
tan, qtan	tangent
xtoy	x^y
matin2, matinf, maninv	matrix inversion
matmul, mat2x2, mat3x3, mat4x1, mat4x1f, mat4x4, mat5x5	matrix multiplication
fir, fir5, firc	finite impulse response filters
iir, iir2, iir3, iir4, iird, iirt, iirt1, iirt2, iirt3, iirt4	infinite impulse response filters
lms, lmsc, lmsl	real adaptive FIR filters using least-mean-square algorithm
fft	fast fourier transform
hamm, hamm0, hamm1, chamm0	multiply by Hamming Window
hann, hann0, hann1, chann0	multiply by Hanning Window

3 Using DEVtools

Host/Node Communication

Introduction

Most applications that run on the Pixel Machine must communicate with the host system in order to receive the data to be processed, to return results, or to perform operations, such as I/O, that the Pixel Machine processors cannot perform on their own.

This section describes the ways in which host programs and Pixel Machine programs communicate. It is divided into sections that describe:

- communication to the Pixel Machine using the DEVtools command protocol
- communication from the Pixel Machine to the host using the message passing protocol
- other communication using a user-defined protocol

The DEVtools Command Protocol

A complete Pixel Machine program is one that uses all the architectural components of the Pixel Machine, and consists of:

- a controlling host program
- DSP programs running in the pipe nodes
- DSP programs running in the pixel nodes

Host programs usually command the Pixel Machine by sending data using the DEVtools command protocol, which is a convention for passing data from the host to the pipe(s) in the Pixel Machine. The data is sent from the host to the first pipe node in the Pixel Machine in units called *commands*. The pipe nodes can modify, delete, or pass on the command packets unmodified to the next node. They can also generate new packets to be broadcast to the pixel nodes.

Each *command* consists of opcode, an operand count, and the operands. The opcode and operand count are encoded into a single 32-bit value. The operands are 32-bit quantities that can be integers, host floating point values, or Pixel Machine floating point values.

The format of *commands* on the host is:

OPCODE COUNT $PARAM_{[1]}$ ... $PARAM_{[count]}$

Macros are provided to simplify the generation and processing of commands on the host. These macros are used to write commands to the Pixel Machine pipelines and read commands back from the feedback FIFO. These macros are defined in **devcommand.h** (see the DEVtools *Reference Manual*). The following is an example of the code required to generate a command:

```
DEVcwrite2(DEVcommand(opcode, 2), int, some_data, more_data);
```

DEVcommand is used to encode an opcode and parameter count into a 32-bit command code. The *command* argument of the **DEVcwrite** macros is usually a call to **DEVcommand**. *opcode* is a user defined positive value. It is only important that the host and Pixel Machine routines agree on the meaning of the opcodes and the format of the operands that follow each opcode. The 2 in the **DEVcommand** macro is the number of operands that follow this command. This is frequently the same as the last character of the macro name, but it is not always the same, because multiple *write* macro invocations are required for commands that contain operands of more than one data type. *int* is the type of the operand to be passed to the Pixel Machine. *some_data* and *more_data* are expressions that are used as the values of the operands.

Following is an example of a command that contains both integer and floating point operands:

```
DEVcwrite2(DEVcommand(opcode, 4), float, x, y);
DEVwrite2(int, i, j);
```

To send an opcode *cmd* with no parameters use:

```
DEVcwrite0(DEVcommand(cmd, 0));
```

To send an opcode with one integer argument, *argi*, use:

```
DEVcwrite1(DEVcommand(cmd,1), int, argi);
```

To send an opcode with one float argument, *argf*, use:

```
DEVcwrite1(DEVcommand(cmd,1), float, argf);
```

There are separate macros to write from 0 to 9 parameters. If the number of parameters will not be known until run-time, use **DEVcwriten**. For example:

```
DEVcwriten(DEVcommand(cmd, length), float, flt_array, length);
```

If possible, it is best to use the individual macros because they are more efficient than **DEVcwriten**.

The **DEVcwrite0** through **DEVcwrite9** macros are used to write commands and a number of operands that match the last character of the macro name. The **DEVwrite0** through **DEVwrite9** macros only write operands; they do not output a command code.

DEVcommand_opcode and **DEVcommand_length** are used with the **DEVreadn** macros to extract the opcode and length from the encoded value when reading from the feedback FIFO.

On systems with multiple pipes configured in parallel, the macros write to whichever of the pipes is the *current* pipe. Commands can be written to the alternate pipe by using the macros ending with the string *_alt*. The *_alt* macros must not be used on single pipe systems or on multi-pipe systems whose pipes are configured in series.

A few more details about the command formats (these are all taken care of by the macros): The **DEVcommand** macro turns the count into a negative byte count and packs it into one word together with the opcode. The byte ordering on the Sun and Pixel Machine are also different, so when sending bytes or 16-bit ints packed into a 32-bit parameter, it is necessary to do some swapping. The floating point format is also different, and the conversion must be done explicitly either on the host or in the nodes. It is usually more efficient to do the float conversion in the pipe nodes.

From the point of view of the pipe nodes, the command packets are read one 32-bit word at a time from the input FIFO and possibly written to the output FIFO. In the nodes, a set of functions (**PMgetcmd, PMgetdata, PMgetop, PMputcmd, PMputdata, PMputop** (described below)) is provided for efficient reading and writing of the hardware FIFOs. All the FIFO routines use a data structure called **PMcommand** that holds the command packets.

The **PMcommand** structure is defined in the **pxm.h** file, and is as follows:

```
#include <pxm.h>

typedef struct
{
     short     opcode;
     short     count;
     float     *data_ptr;
} PMcmdtype;

extern PMcmdtype PMcommand;
```

The global data structure, **PMcommand**, defined in both the pipe and pixel node libraries, reflects this packet structure. The members of this structure have the following functionality:

- PMcommand.opcode: contains the user-defined opcode.
- PMcommand.count: contains the negated byte count of the parameters pointed to by the next field.
- PMcommand.data_ptr: points to a static buffer containing the parameters. It is initialized by the system, although this can be changed to point to a user-defined buffer. The location of this buffer is specified to optimize the DSP32 data move instruction.

Reading Commands from the Input FIFO

Pipe node programs read a command in two steps:

1. call **PMgetop()** to load an opcode and count from the input FIFO into the **PMcommand** structure.
2. if the parameter count is nonzero, call **PMgetdata()** to load parameters from the input FIFO into the **PMcommand** structure.

Pixel nodes read a command by calling **PMgetcmd()**, which loads all three components of the command into **PMcommand**.

Writing Commands to the Output FIFO

Pipe node programs can write a command in two ways:

1. by calling **PMputop()** followed (if *count* is nonzero) by **PMputdata()**.
2. by calling **PMputcmd()**, which combines the functionality of **PMputop()** and **PMputdata()**.

By changing members of the **PMcommand** structure, a pipe node program can modify the command stream as needed. Pixel node programs read commands from the pipe nodes but cannot write

commands.

Data flows through the system in the following manner: the program on the host assembles command packets which consist of an opcode, count, and data and sends them to the first pipe node via the first input FIFO. The pipe node (and subsequently the rest of the pipe nodes) reads the opcode and decides what to do with it. There are three scenarios:

1. it could simply pass the command packet to the next node via the output FIFO
2. it could do some processing and consume the command packet without passing it on
3. it could do some processing of the data and then pass it on to the next node. It can use the same opcode or change it to another one. It could alter the data (e.g., convert IEEE format floats to DSP format) or change it entirely, even passing on several new command packets.

After the last pipe node processes its commands, it writes the packets to its output FIFO which is broadcast to the input FIFOs of all the pixel nodes. Each pixel node can then read the packets and process the opcode and parameters according to its algorithm.

Image Upload and Download

During the course of processing images on the Pixel Machine, it is often necessary to download an image from the host to the Pixel Machine (e.g., to display or perform some processing) or upload an image from the Pixel Machine to the host (e.g., to save a result). Because upload/download of a full image requires moving over 4Mbytes of information, it is desirable to accomplish this task as quickly as possible.

Two DEVtools routines, **DEVget_scan_line** and **DEVput_scan_line**, have been designed to provide users with fast and flexible image upload/download processing. Both routines take care of the pixel interleaving/de-interleaving. To process images quickly, both **DEVget_scan_line** and **DEVput_scan_line** require cooperating code to be executed on the Pixel Machine. To perform the upload/download, the host routines send system commands to the Pixel Machine. When a program executing on a pipe/pixel node receives a system command via **PMgetop** or **PMgetcmd**, it checks to see if the command was "enabled" by **PMenable**. If the command was not enabled, the node takes no action and passes the command on to the next node (in the case of pipe nodes). If the command was enabled, the appropriate DEVtools routine is called to process the command. After the system command is processed, control is passed back to **PMgetop** or **PMgetcmd** to receive the next user command.

Both **DEVget_scan_line** and **DEVput_scan_line** take a *mode* argument that specifies the format of an individual pixel and which portion of Pixel Machine memory the pixels should be uploaded/downloaded from/to. The following pixel formats are supported:

- DEV_RGBA_PACKED_PIXELS – on the host each pixel is 4 bytes long and the red pixel component is stored in the first byte (the byte at the lowest memory address).
- DEV_RGB_PACKED_PIXELS – on the host each pixel is 3 bytes long and the red pixel component is stored in the first byte (the byte at the lowest memory address). When using DEV_RGB_PACKED_PIXELS in ZRAM, it is assumed that pixels are stored in RGBA format, therefore upload uploads 3 bytes, skips one, uploads the next 3 bytes, etc.
- DEV_MONO_PIXELS – on the host each pixel is one byte long. When downloaded to VRAM, the pixel component is placed in each of the red, green, blue and alpha components. When uploaded/downloaded from/to ZRAM, pixels occupy consecutive bytes.
- DEV_MONO_R_PIXELS – on the host each pixel is one byte long. When uploaded/downloaded from/to VRAM, the pixel component is placed in the *red* component of a pixel. Other pixel components are left untouched. ZRAM upload/download is not supported.
- DEV_MONO_G_PIXELS – on the host each pixel is one byte long. When uploaded/downloaded from/to VRAM, the pixel component is placed in the *green* component of a pixel. Other pixel components are left untouched. ZRAM upload/download is not supported.

- DEV_MONO_B_PIXELS – on the host each pixel is one byte long. When uploaded/downloaded from/to VRAM, the pixel component is placed in the *blue* component of a pixel. Other pixel components are left untouched. ZRAM upload/download is not supported.
- DEV_MONO_A_PIXELS – on the host each pixel is one byte long. When uploaded/downloaded from/to VRAM, the pixel component is placed in the *alpha* component of a pixel. Other pixel components are left untouched. ZRAM upload/download is not supported.
- DEV_MONO_16_PIXELS – on the host each pixel is two bytes long. When uploaded/downloaded from/to ZRAM, each pixel occupies successive ints. VRAM upload/download is not supported.
- DEV_DSP_FLOAT_PIXELS – on the host each pixel is four bytes long. When uploaded/downloaded from/to ZRAM, each pixel occupies successive floats. VRAM upload/download is not supported.
- DEV_IEEE_FLOAT_PIXELS – on the host each pixel is four bytes long. When uploaded/downloaded from/to ZRAM, each pixel occupies successive floats. During the download operation, each pixel (float) is treated as an IEEE floating point number and converted to the DSP internal floating point format. During the upload operation, each pixel (float) is treated as a DSP floating point number and converted to the IEEE floating point format. VRAM upload/download is not supported.

In addition to the above pixel formats, the *mode* argument also specifies the area in Pixel Machine memory to upload/download from/to:

- DEV_FRONT_BUFFER – the currently visible portion of VRAM. Typically used to display an image on the monitor.
- DEV_BACK_BUFFER – the currently non-visible portion of VRAM. Typically used to upload/download an image while another image is being displayed on the monitor.
- DEV_VRAM0_BUFFER – the VRAM0 portion of VRAM (only available on models 932 and above). Typically used to store an image that is larger than the size of the screen. Note that VRAM0 is the union of the FRONT and BACK buffers.
- DEV_VRAM1_BUFFER – the VRAM1 portion of VRAM (only available on models 932 and above). Typically used to store an image that is larger than the size of the screen. Note that VRAM1 is not directly visible.
- DEV_ZRAM_BUFFER – non-displayable dynamic RAM used typically for storing Z buffer values (ZRAM). Typically used to perform numerical calculations on image data.

Note that for all forms of VRAM, the subscreen concept is used, but for ZRAM upload/download the subscreen concept is *not* used. This allows for more efficient use of ZRAM when performing image processing.

The following table gives the size (in pixels) of the largest image that can be stored in each of the above buffers:

Model	FRONT	BACK	VRAM0	VRAM1	ZRAM
916	1024x1024	1024x1024	–	–	1024x1024
920	1280x1024	1280x1024	–	–	1280x1024
932	1024x1024	1024x1024	1024x2048	1024x2048	1024x2048
940	1280x1024	1280x1024	1280x2048	1280x2048	1280x2048
964	2048x1024	2048x1024	2048x2048	2048x2048	2048x2048
964X	2048x1024	2048x1024	2048x2048	2048x2048	2048x2048

Note that for ZRAM the numbers given in the above table should be multiplied by 2 for DEV_MONO_16_PIXELS and by 4 for DEV_MONO_PIXELS.

When enabling reception of system commands using **PMenable**, the user has the choice of enabling upload/download for all memories, just VRAM or just ZRAM. Because program size on the pixel nodes is minimal, it is recommended that users enable the smallest piece that they need. If users run out of program space and still wish to perform image upload/download, the **DEVget_pixel** and **DEVput_pixel** routines can be used (albeit more slowly).

The DEVtools Message Service Protocol

It is often necessary for the processors in the Pixel Machine to initiate communication back to the host system. This is required to perform tasks that can only be done on the host, such as input and output operations.

The protocol used to send messages to the host has the following steps:

- the node checks its semaphore to see if any previous operation has completed
- the node loads a message code into the PIR; the semaphore is set
- meanwhile, the host is polling the PIR registers of a designated set of pipe and/or pixel nodes. When a message code is found in a node's PIR, it is used as an index into an array of function pointers initialized by calling **DEVuser_msg_enable()**
- message specific code is executed on the host to perform any other communication related to this message
- if the message operation must complete before execution continues, then the node process must wait for the semaphore to be cleared by the host

The message protocol is used for communications operations that are internal to DEVtools (such as host communication required by the **printf** routine) as well as to implement user message routines. Message codes that are used internally by DEVtools are known as *system messages*. Message codes that are used for user defined functions are known as *user messages*.

Functions are provided to send a message code to the host and to check to see if the semaphore has been cleared. The **PMusermsg** function checks to see if the semaphore is clear, and then sets the semaphore and loads the PIR. It has one argument, the message code (a positive integer from 1 to 256) to be sent to the host.

Upon receiving a message code, the host will perform the action requested and then clear the semaphore. The Pixel Machine program can continue execution after sending the message code, or it may wait for the semaphore if its processing requires that the host action be completed before execution can continue.

In order for the host to serve the message requests, a process on the host must poll the Pixel Machine processors for pending messages. The polling processing is designed to be incorporated into a user's program that runs on the host. In this way, the message serving functions can be combined with other host processing that may include other operations such as generating pipe commands using the command protocol described previously.

DEVpoll_nodes is the function that polls the Pixel Machine processors. The user program may poll a single node, all nodes, or a range of nodes. Both pipe nodes and pixel nodes may be polled. The number of times that the processors are to be polled and the delay time between polls are arguments to **DEVpoll_nodes**. To poll continuously, the value DEV_FOREVER can be used. The delay time is used as a argument to the **usleep()** system call. If no delay is wanted, DEV_NONE should be used. DEV_NONE should only be used for host serving applications that need to be able to serve Pixel Machine requests very quickly, because the host process using DEV_NONE will consume as much CPU time as it can get.

For **DEVpoll_nodes** to recognize a user message code, that specific message code must be enabled. This is done by calling **DEVuser_msg_enable**. For example:

```
DEVuser_msg_enable(1, pipe_function, pixel_function);
DEVuser_msg_enable(2, NULL, pixel_func_2);
```

In the first line of the example, user message code 1 is enabled. If message code 1 is received from a pipe node, a call to pipe_function is made. If message code 1 is received from a pixel node, a call to pixel_function is made. In the second line, if message code 2 is received from a pipe node, an error will be generated. If a message code is received for a code that has not been enabled, an error is also generated. pipe_function, pixel_function, and pixel_func_2 represent user written routines that are called when the specified message code is received. The following is a sample of the declarations for a user-written message handler:

```
int pipe_function(opcode, pixel_system, node)
int      opcode;
DEVpixel_system *pixel_system;
int      node;

int pixel_function(opcode, pixel_system, node)
int      opcode;
DEVpixel_system *pixel_system;
int      node;
```

opcode is the user message code that caused the message handler to be called. This allows the message handler to know which code was received so that one function can be used to handle several message codes. *pixel_system* is a pointer to a system descriptor and is returned by **DEVinit**. *node* is the number of the node that sent the message code. It is possible to have a single function serve both the pipe and pixel nodes.

Once a message code has been received, it is often necessary for the server routine to communicate with a processor to send or receive other information. Other communication may be performed using the low-level Pixel Machine control library functions. These functions provide routines that perform DMA I/O, read from the PIR, write using the PDR, and provide other monitoring and control functions.

Message serving routines may use any of the DEVtools routines to transfer data to and from the processor. The message server routine and the message sending routine must agree on how data is transferred, and how much data is transferred. If the sender and receiver get out of sync, it is possible for the Pixel Machine or the host system to get caught in a loop waiting for more data, or for the host to attempt to interpret data from the Pixel Machine as message codes.

Message serving routines should not access the software semaphore since this is used by the message handling routines to indicate that a message operation has been completed. The semaphore is reset by **DEVpoll_nodes** after the return from the user message handler.

Node Global Variables

libpm contains a set of global data that is initialized by the startup code and is available for users. These variables should be treated as read-only by the user. Corrupting their values would have destructive effects on many of the library commands.

The following variables are defined for both pipe and pixel node programs. They are all declared "extern" in **pxm.h**. Their values are set by **hypload, DEVpixel_boot** or **DEVpipe_boot.**

```
int       PMnode;          /* node identification number [0-63]          */
int       PMnx;            /* number of drawing nodes in x [4,8,10]      */
int       PMny;            /* number of drawing nodes in y [4,8]         */
int       PMox;            /* drawing node's offset in x [0-7]           */
int       PMoy;            /* drawing node's offset in y [0-7]           */
char      PMsid[10];       /* software name (10 chars)                   */
int       PMsem;           /* software semaphore                         */
int       PMmodel;         /* coded pixel machine model                  */
int       PMvideo;         /* video format code                          */
int       PMpipe;          /* pipe mode code                             */
int       PMxmax;          /* maximum x value in screen space            */
int       PMymax;          /* maximum y value in screen space            */

PMcmdtype PMcommand;       /* PMcommand struct with Opcode,Count and
                            * DataPtr for reading and writing FIFO's
                            */
```

The **PMcommand** structure is used by all the FIFO input and output routines in both pipe and pixel nodes.

The following variables are defined only for pixel node programs. They should not be referenced from pipe node programs. If they are, their values will be undefined. Their values are set in the startup code and depend on the configuration of the machine.

```
int     PMimax;          /* max pixels in I direction in processor space */
int     PMjmax;          /* max pixels in J direction in processor space */
int     PMmx;            /* more processing in x direction? boolean      */
int     PMmy;            /* more processing in y direction? boolean      */

        /*
         *       Table of Values for PMmx and PMmy
         *
         *
         *                |model    |PMmx     |PMmy  |
         *                |-------  |-------  |----  |
         *                |964      |  0      |  0   |
         *                |940      |  1      |  0   |
         *                |932      |  1      |  0   |
         *                |920      |  1      |  1   |
         *                |916      |  1      |  1   |
         *
         */

int     PMnindex;        /* total number of subscreens (2*PMmy+PMmx+1)   */

PMsubscrn *PMscrns[4];   /* initialized array of subscreen pointers */
```

Although there are four **PMscrns**, only the appropriate ones are initialized for a given model. **PMscrns** should only be used to pass it to an appropriate screen function.

Frame Buffer Memory and Subscreens

Introduction to Subscreens

As described in Chapter 1, each processor contains a portion of the frame buffer memory. For example, on a 64 processor system displaying a 1024x1024 image, each processor contains 128x128 (16384) pixels of the frame buffer. On systems with fewer than 64 processors, each processor is responsible for a larger area of the frame buffer. A 32 processor system, for example, is responsible for 128x256 (32768) pixels, while a 16 processor system is responsible for 256x256 (65536) pixels.

To provide a simple and uniform interface to the hardware that works for all system configurations, the concept of virtual nodes or subscreens was developed. Through the use of subscreens, each processor repeats a set of operations from one to four times, operating on a different "subscreen" or portion of the frame buffer each time. In essence, subscreens perform the function of several processors of a larger system.

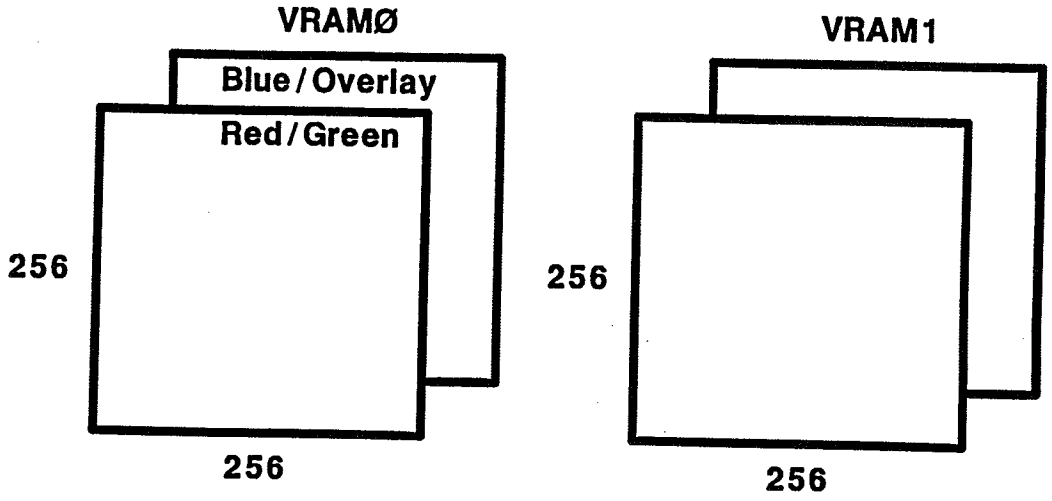
On a 64 processor system each processor contains a single subscreen. On smaller configurations each processor contains a number of subscreens such that the total number of subscreens is either 64 or 80; 80 for 20 and 40 processor systems, and 64 for 16 and 32 processor systems. In other words, in 32 and 40 processor systems, each processor contains two subscreens; in 16 and 20 processor systems, each processor contains four subscreens.

Where the Frame Buffers are Stored in VRAM

For many DEVtools applications it is helpful to understand where pixels are located in memory, which memory areas are used by the frame buffer and which memory areas are available, and so on. The figures that follow illustrate which memory is used for frame buffer storage on each system configuration. All examples are for 1024x1024 images on 16 and 32 processor systems, and for 1280x1024 for 20 and 40 processor systems. Examples of both 1280x1024 and 1024x1024 are provided for 64 processor systems.

Figure 3-1 illustrates the two 256k banks of VRAM found on each pixel node. Each bank consists of two planes, and each plane consists of a 256x256 array. Each element of the array contains two color components: the first plane contains the red and green values, while the second plane contains blue and overlay.

Figure 3-1: Pixel Nodes: Video Memory Organization



Figures 3-2–3-5 designate the portion of VRAM that is used to contain the frame buffer. The Pixel Machine supports double–buffering on all configurations. The buffer shown is the memory used in single buffer mode, or what is called the "top buffer" in double–buffered mode. The second buffer, or "bottom buffer", is stored in the lower portion of the boxes shown in the figures and always begins at row number 128. The names "top" and "bottom" refer to the location of the buffer within VRAM, and should not be confused with "front" and "back" which refer to the buffer currently being displayed and updated, respectively.

Figure 3-2 shows the top buffer of a 964 operating in 1024x1024 mode. Each buffer consists of a single subscreen containing 128x128 pixels.

Figure 3-2: Frame Buffer Organization on a Model 964

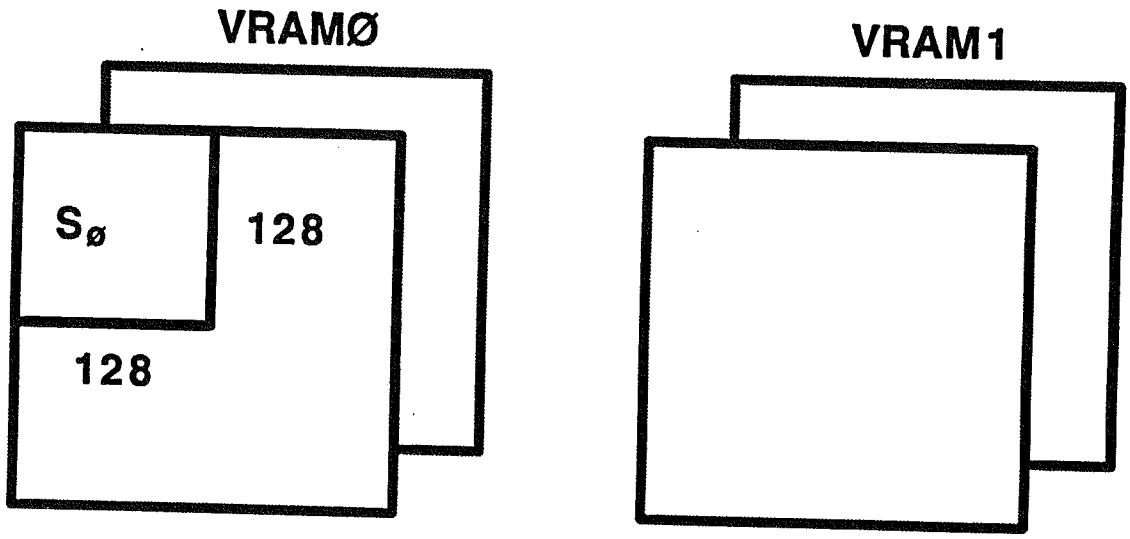


Figure 3-3 shows the top buffer of a 964 operating in 1280x1024 mode. Each buffer consists of a single subscreen containing 160x128 pixels.

Figure 3-3: Frame Buffer Organization on a Model 964X

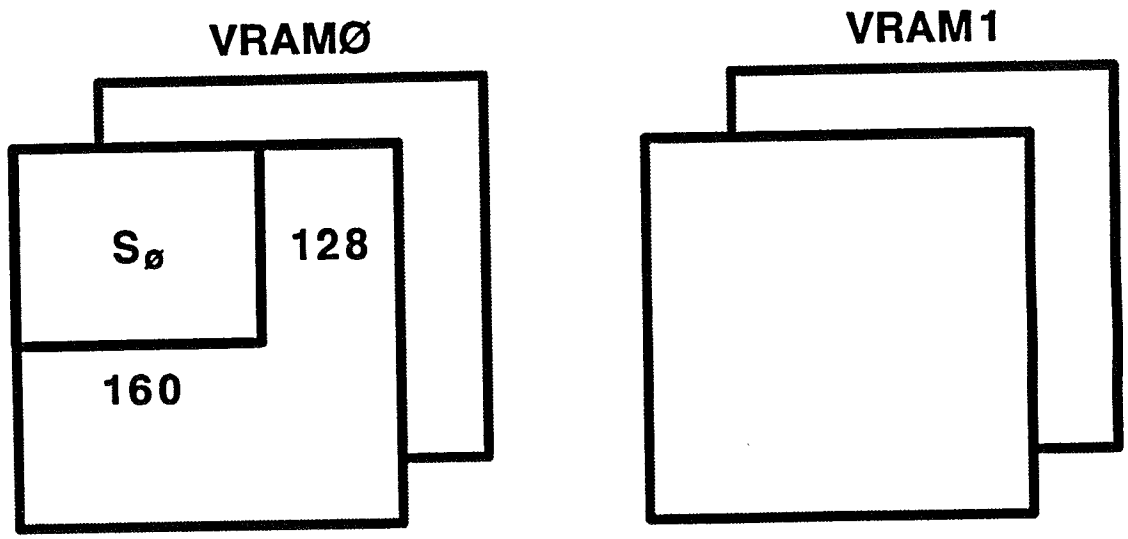


Figure 3-4 shows the top buffer of a 940 operating in 1280x1024 mode, or a 932 operating in 1024x1024 mode. Each buffer consists of two subscreens, each containing 128x128 pixels.

Figure 3-4: Frame Buffer Organization on a Model 940/32

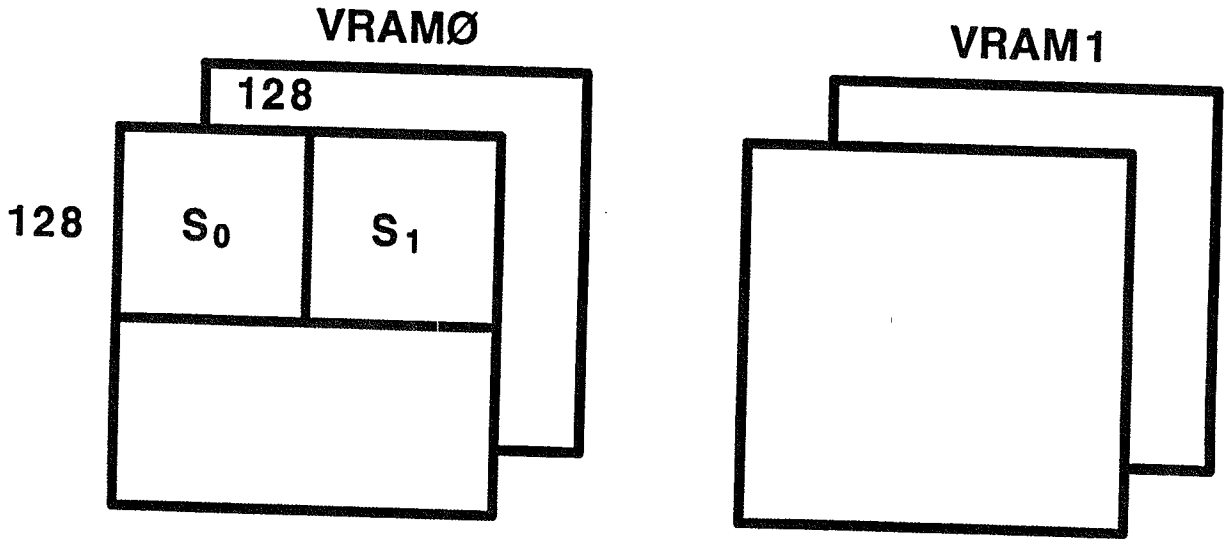
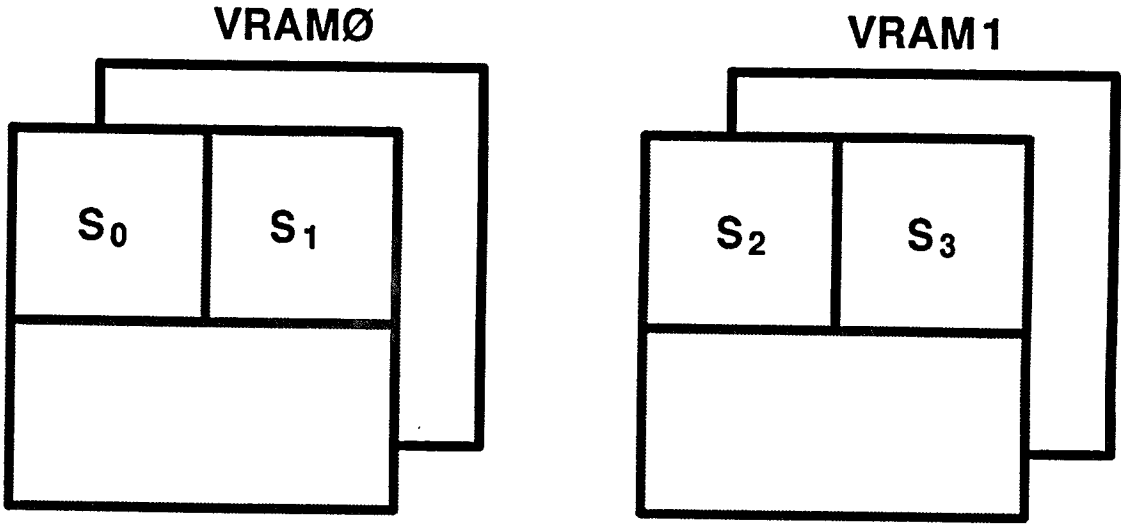


Figure 3-5 shows the top buffer of a 920 operating in 1280x1024 mode, or a 916 operating in 1024x1024 mode. Each buffer consists of a four subscreens, each containing 128x128 pixels.

Figure 3-5: Frame Buffer Organization on a Model 920/16

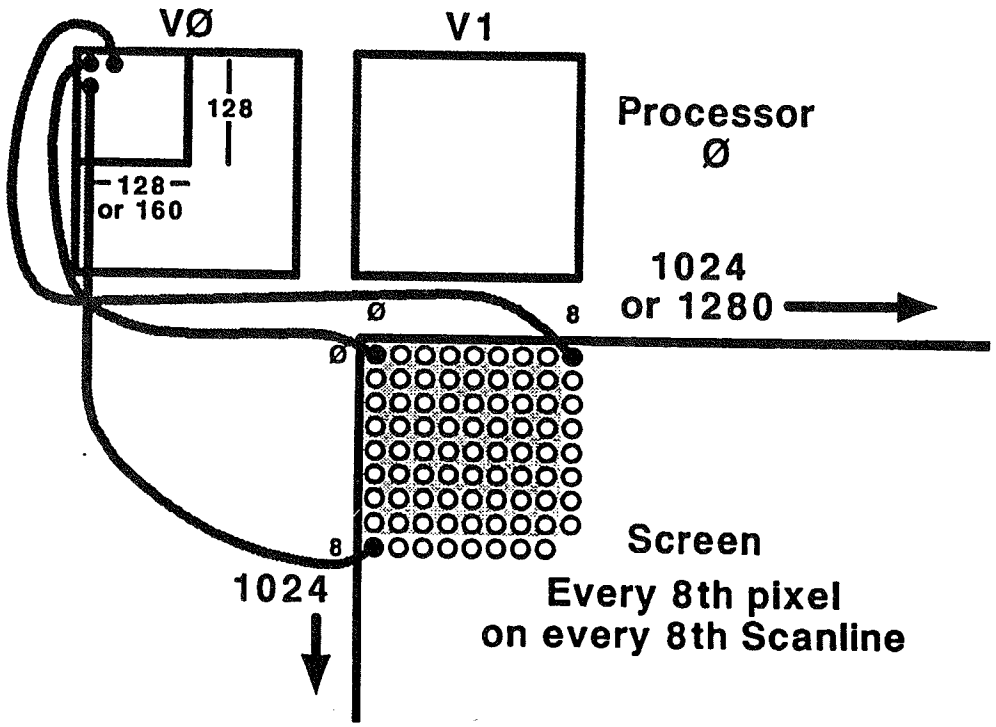


Subscreen to Screen Mapping

Once you know where the subscreens are in memory, you must understand how the pixels in a given subscreen correspond to the pixels on the screen. Figures 3-6–3-10 show, for each configuration, where the pixels for a given subscreen are displayed.

Figure 3-6 shows the mapping for a 964. With only a single subscreen, the 964 is the simplest case. Each processor displays every 8th pixel of every 8th scanline.

Figure 3-6: Processor to Screen Mapping on a Model 964



Figures 3-7 and 3-8 show the mappings for the 940 and 932, respectively. The 940 contains a 10x4 array of processors, the 932 an 8x4 array. Each processor performs the function of two processors in the Y dimension, resulting in a 10x8 or 8x8 array of subscreens.

Figure 3-7: Processor to Screen Mapping on a Model 940

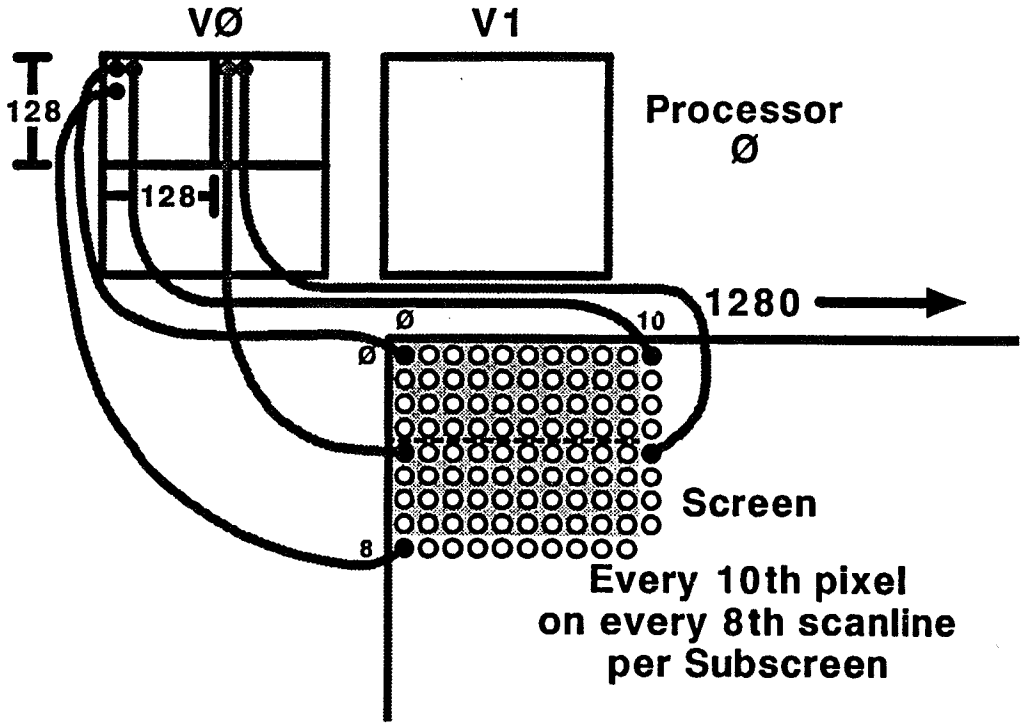
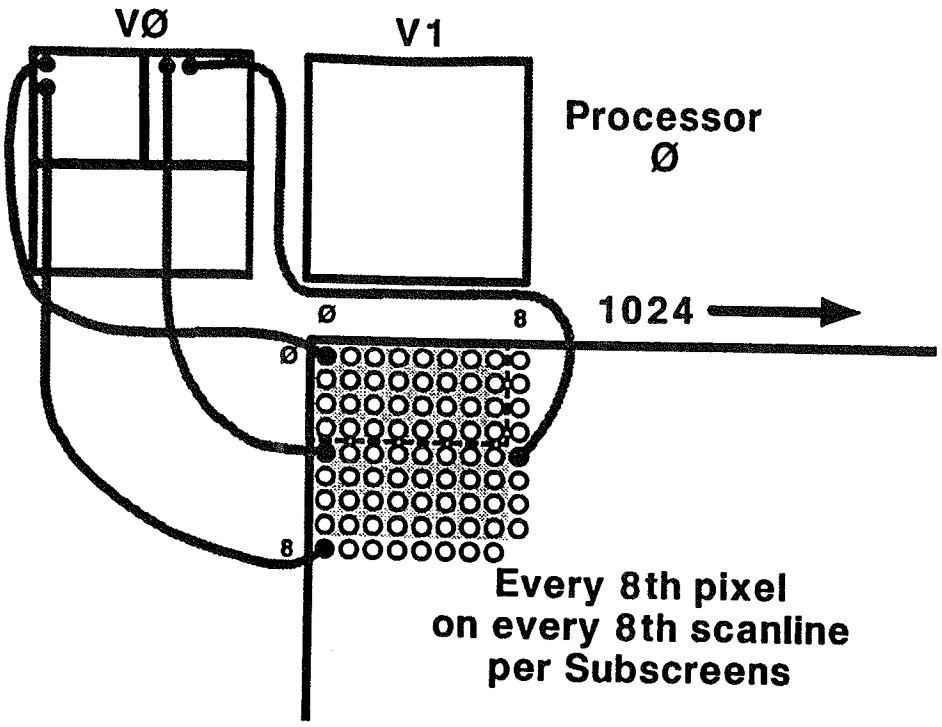


Figure 3-8: Processor to Screen Mapping on a Model 932



On a 940, each processor displays:

Subscreen 0 every 10th pixel of every 8th scanline, beginning with scanline **PMoy**

Subscreen 1 and every 10th pixel of every 8th scanline, beginning with scanline **PMoy+4**

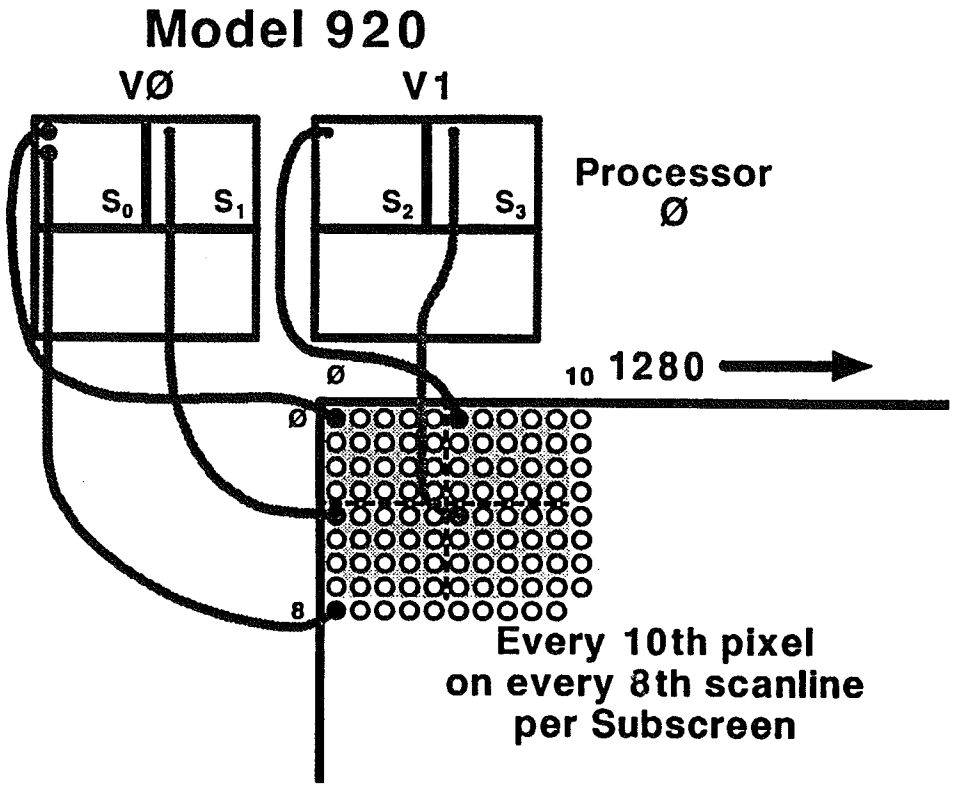
On a 932, each processor displays:

Subscreen 0 every 8th pixel of every 8th scanline, beginning with scanline **PMoy**

Subscreen 1 and every 8th pixel of every 8th scanline, beginning with scanline **PMoy+4**

Figures 3-10 and 3-11 show the mappings for the 920 and 916, respectively. The 920 contains a 5x4 array of processors, the 916 a 4x4 array. Each processor performs the function of two processors in the X dimension and two processors in the Y dimension, for a total of four. The result is an array of 10x8 or 8x8 subscreens.

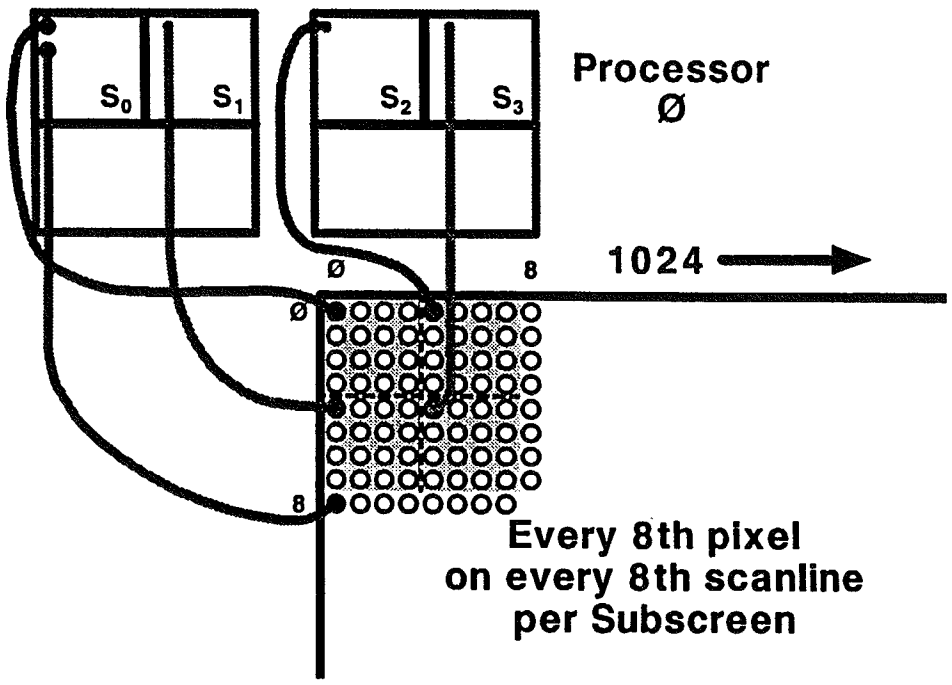
Figure 3-9: Processor to Screen Mapping on a Model 920



On a 920, each processor displays:

Subscreen 0	every 10th pixel of every 8th scanline, beginning with pixel **PMox** of scanline **PMoy**
Subscreen 1	and every 10th pixel of every 8th scanline, beginning with pixel **PMox** of scanline **PMoy+4**
Subscreen 2	every 10th pixel of every 8th scanline, beginning with pixel **PMox+5** of scanline **PMoy**
Subscreen 3	and every 10th pixel of every 8th scanline, beginning with pixel **PMox+5** of scanline **PMoy+4**

Figure 3-10: Processor to Screen Mapping on a Model 916



On a 916, each processor displays:

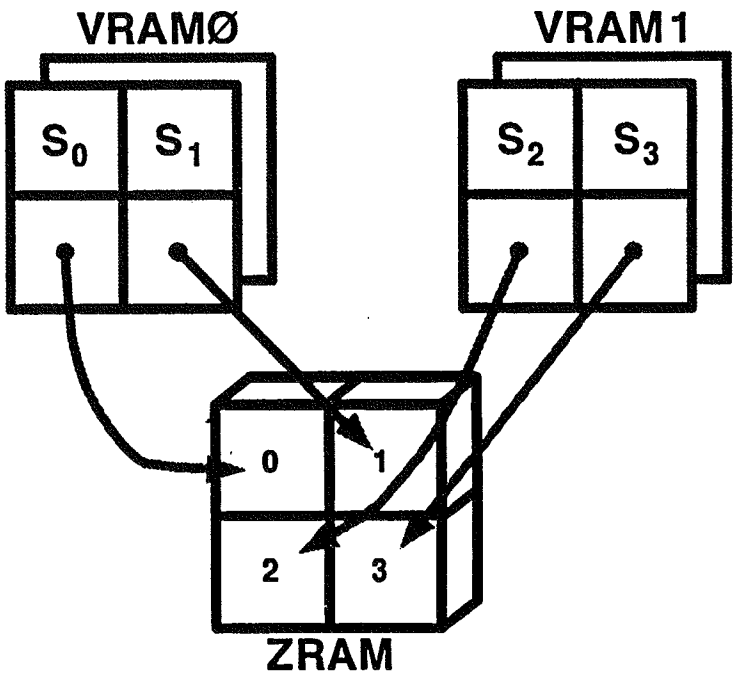
Subscreen 0	every 8th pixel of every 8th scanline, beginning with pixel **PMox** of scanline **PMoy**
Subscreen 1	and every 8th pixel of every 8th scanline, beginning with pixel **PMox** of scanline **PMoy+4**

Subscreen 2	every 8th pixel of every 8th scanline, beginning with pixel **PMox+4** of scanline **PMoy**
Subscreen 3	and every 8th pixel of every 8th scanline, beginning with pixel **PMox+4** of scanline **PMoy+4**

Subscreens in Z Memory

When Z memory is being used to store pixel-related data, such as Z-buffer values, it is necessary to divide the Z memory into subscreens. Figure 3-12 shows the Z memory subscreen mapping used by such DEVtools functions as **PMputzbuf()**.

Figure 3-11: Z-Buffer Mapping on a Model 916/920



Programming with Subscreens

Program code can be broken into three classes:

- subscreen independent code – code that must be executed only once
- subscreen dependent code – code that must be executed once for each subscreen
- code that functions correctly whether executed once or once for each subscreen

Subscreen independent code includes functions such as reading data from an input FIFO, swapping buffers with **PMswapbuff()**, and initializing Z memory using **PMzbrk()**. These are functions that should not be repeated for each subscreen.

Subscreen dependent code includes computing the processor space coordinates using the **PMilo()** and **PMihi()** macros, and updating the frame buffer using functions such as**PMclear()**, and **PMputpix()**. All functions that take a subscreen argument (e.g., **PMilo(), PMihi()**) are subscreen dependent.

Programs typically consist of a subscreen independent function that reads data from the input FIFO, performs some processing and then calls a subscreen dependent function N times, once for each subscreen. The subscreen dependent function receives as an argument a pointer to a structure that describes the subscreen to be processed.

Code that functions correctly as either subscreen dependent or independent could include part of an application that performs calculations on data not directly related to drawing pixels on the screen.

PMsubscrn is the type name of the structure used to describe a subscreen. The fields contained in each subscreen structure are:

Nx	number of subscreens or virtual nodes in the X dimension
Ny	number of subscreens or virtual nodes in the Y dimension
Ox	offset of this subscreen in the X dimension
Oy	offset of this subscreen in the Y dimension
ifix	byte offset from the beginning of the VRAM row of this subscreen
jfix	specifies whether the subscreen is in VRAM 0 or VRAM 1 and whether it is in the top buffer or bottom buffer

All of these values are stored as floating point values.

The *ifix* and *jfix* values can be useful when determining the location of a subscreen if page registers are being used to update the frame buffer. The subscreen structure also contains values that are used to compute the **PMilo()**, **PMihi()**, **PMjlo()**, and **PMjhi()** functions.

DEVtools includes a simple facility that can be used to call a subscreen dependent function the appropriate number of times with the proper subscreen structure pointer. This is done by using the **PMapply()** function. **PMapply()** calls a specified function and passes, as the first argument, the subscreen pointer. For example the statement:

```
PMapply(PMclear, 0, 0, PMimax, PMjmax, &color);
```

is functionally equivalent to:

```
for (i = 0; i < PMnindex; i++) {
        PMclear(PMscrns[i], 0, 0, PMimax, PMjmax, &color);
}
```

If code is written to run on a specific model, the **PMscrns** array can be used to access the pointer for a specified subscreen. Code written exclusively for a 964, for example, could be written as:

```
PMclear(PMscrns[0], 0, 0, PMimax, PMjmax, &color);
```

The global variables **PMmx** and **PMmy** can be used to determine the number of subscreens in each dimension. When **PMmx** is true, it means that there is more than one subscreen for each scanline X. **PMmx is true in high-resolution mode for all systems except the 964. When PMmy** is true it means that there is more than one subscreen for each scan column Y. **PMmy** is true in high-resolution mode for the 916 and 920.

Subscreens and Video Formats

NTSC uses only a single subscreen on all configurations. PAL uses the same number of subscreens as high-resolution, however, each subscreen is smaller.

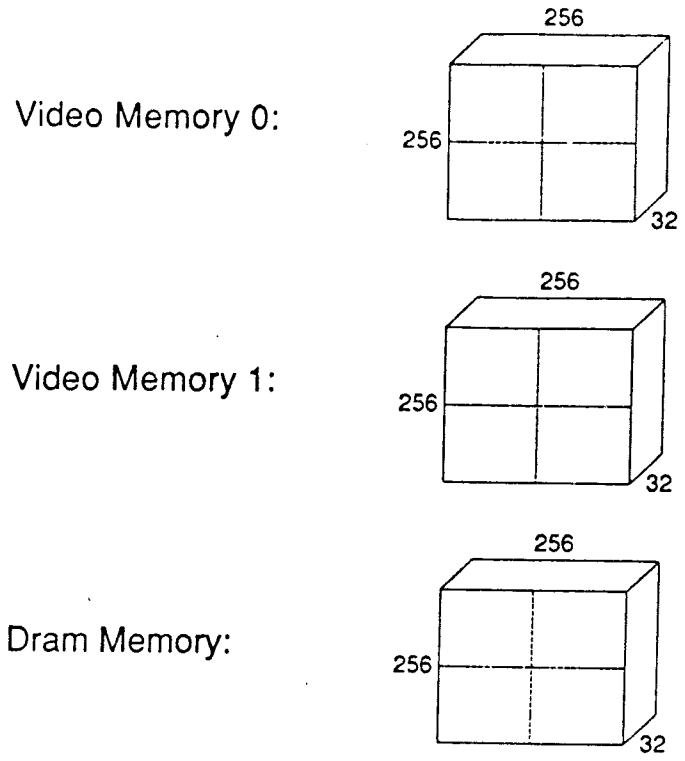
Accessing Memory Without Subscreens

Images are sometimes stored in memory without using subscreens. This can simplify some image manipulation tasks when the data is not required to be in a displayable format while being processed. For example, an image copied into Z memory using the **PMcopyvtoz()** function is not stored in subscreen format in Z memory. A special subscreen structure is provided to allow functions such as **PMilo()** and**PMihi()** to be used with these images. The global variable **PMrealscrn** can be used to access this structure. In the **PMrealscrn** structure, the physical node counts and offsets are always equal to the virtual values.

Memory Access

There are several different types of memory associated with each pixel node. These can be divided into three groups: Static Memory (SRAM), Video Memory (VRAM), and Dynamic Memory (DRAM). SRAM includes 1kx32-bit of on chip memory and 8kx32-bit off chip memory, which are both available for program storage. Additionally, there is memory for the frame buffer and pixel data (usually for z-depth information). This additional memory consists of two banks of 256x256x32-bit VRAM, referred to as VRAM0 and VRAM1, and one bank of 256x256x32-bit DRAM used for the z-buffer, which is also referred to as ZRAM.

Figure 3-12: Pixel organization of the rgbα (video) and z (dynamic) memories



VRAM and ZRAM Access

Because of the way that non-program memory is organized, special low level functions and macros are needed to access it. These functions are needed because access to this memory involves using page registers. In addition, each bank of pixel memory (VRAM) is actually composed of two planes, an RG (red/green) plane and BO (blue/overlay) plane that are accessed separately. Also, only 8-bits out of each 16 are actually used. For additional information, see the section "Pixel Nodes" in Chapter 1 of this guide.

Following is a list of library routines to help access memory correctly:

Table 3-1: VRAM Access

Routine	Function
PMgetscan()	read a scanline from video memory
PMputscan()	write a scanline to video memory
PMpixaddr()	generate a pointer to a specific pixel
PMgetpix()	read a pixel from the specified subscreen
PMputpix()	write a pixel to the specified subscreen
PMv0get()	read a pixel from video buffer 0
PMv0put()	write a pixel to video buffer 0
PMv1get()	read a pixel from video buffer 1
PMv1put()	write a pixel to video buffer 1
PMqget()	quick read of a pixel from the current buffer
PMqput()	quick write of a pixel to the current buffer

Table 3-2: ZRAM Access

Routine	Function
PMgetzbuf()	read a float value from the Z-buffer
PMputzbuf()	write a float value to the Z-buffer
PMzget()	read a float from the Z-buffer
PMzput()	write a float to the Z-buffer
PMqzget()	quick read of Z value from the Z buffer
PMqzput()	quick write of Z value to the Z buffer

There are two types of functions that access pixels: those that use subscreens and those that do not use subscreens. For example, when you are rendering, you would want to access pixels with reference to where they map to the screen. In this case you would want to use subscreens, but if you did not care about the mapping, you could use the direct functions.

When accessing data row by row, you can substantially increase the efficiency by careful use of the "quick" routines, after setting up the pointer and page register by a call to the appropriate function. For example, to update an entire scanline, use **PMpixaddr()** to generate a pointer to the first pixel on the line:

```
dptr = PMpixaddr(scrn, 0, j);
```

Besides returning the pointer **PMpixaddr()** and the other memory functions, all of these routines also have the effect of setting up the appropriate mapping registers. Next use **dptr** as an argument to **PMqput**:

```
dptr = PMqput(color, dptr);
```

PMqput will also return a pointer to the next element. It is safe to use the q routines up to **PMimax** times before reaching the end of the subscreen boundary, at which time a new pointer needs to be generated.

The same applies for ZRAM. However, instead of using **PMqzget** or **PMqzput** the pointer can be used directly because it points to real 32-bit memory (except that the access is slower than SRAM). The Z pointer can be incremented up to **PMimax** times, or, if you are not using subscreens, up to 256 times before the page registers have to be updated by a call to one of the other z routines.

Using Z Memory As General Purpose Memory

Z memory can be used for both Z values used in displaying images and as general purpose memory for data storage. A number of functions have been provided to facilitate the use of Z memory as general purpose memory. These functions correspond, somewhat, to the **malloc** function available on most UNIX systems, but their use is more complex. The additional complexity is due to the limitations imposed by page registers. These functions hide most of the details of page register usage, but still impose some responsibility on the user. Thus programs that used **malloc** on a UNIX system based processor will require some modifications.

The functions used to manage Z memory are:

- **PMzbrk()** – used to reserve the general purpose Z memory pool.
- **PMgetzdesc()** – gets a block of Z memory of the requested size, and returns addressing information. The memory is still not accessible by the program.

- **PMgetzaddr()** – makes the memory accessible to the program and returns the address at which it can be accessed.
- **PMfreezaddr()** – frees the address space to be used for another block of memory.

PMzbrk() is called first to initialize the values that will be used by the other functions. Its only argument is the number of ZRAM rows to devote to allocation by these functions. The memory is allocated from high numbers down, so that, if viewed as rows, **PMzbrk(1)** would set aside row 255. The memory not allocated by **PMzbrk()** is still available for other purposes.

After a large chunk of memory is reserved by **PMzbrk()**, the memory is subdivided and allocated with **PMgetzdesc()**. **PMgetzdesc()** is called with the number of bytes desired. It rounds up to the nearest 4 byte boundary to make sure that the next block of memory is properly aligned for floats or other such data. It updates a private data structure to give the location of the next unallocated block of memory, and returns a *PMzdesc* structure that contains the information on the location of this block of memory. This return should be checked and its value retained. If a number of memory blocks are to be allocated in a program, that is, if **PMgetzdesc()** is called repeatedly, it is probably a good idea to create an array of type *PMzdesc* and keep the returns in that array. Once a block of memory has been allocated it cannot be returned to the memory pool. In most cases, the best way to use *PMzdesc* is to set up a set of buffers, that are re-used, as opposed to the usual way that **malloc** is used, allocating, freeing, and reallocating.

PMgetzdesc() locates available Z memory but does not make it accessible by a program. This requires the setting of page registers and the determination of the correct pointer of the memory block given the page register used. This is the purpose of **PMgetzaddr()**. **PMgetzaddr()** searches the list of page registers reserved for its use (as explained below) to find one which is not in use for some other purpose. Once one is found, it is loaded with the information from **PMgetzdesc()**, so that the processor can address the DRAM. The address is then calculated, so that the program can address that memory. The return from **PMgetzaddr()** should also be checked, because, even if there is enough memory, there may not be any available page registers.

Page registers are a limited resource, and some functions require the use of certain ones. The following table shows the allocation of page registers to functions in **libpm.a**:

Table 3-3: Page Register Assignments

Page Register	Function
0	**PMgetscan(), PMputscan(), PMclear(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
1	**PMgetscan(), PMputscan(), PMclear(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
2	**PMv0get(), PMgetpix(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
3	**PMv0get(), PMgetpix(), PMgetcol(), PMputcol(), PMgetrow(), PMputrow()**
4	**PMv0put(), PMputpix()**
5	**PMv0put(), PMputpix()**
6	**PMv1get()**
7	**PMv1get()**
8	**PMv1put()**
9	**PMv1put()**
10	**PMpixaddr()**
11	**PMpixaddr()**
12	**PMzget(), PMgetzbuf(), PMzaddr()**
13	**PMzput(), PMputzbuf(), PMzaddrcol()**
14	Reserved for host use
15	Reserved for host use

It is unlikely that all of these functions will be used in a given program. It is likely that a set of these functions will be used with the Z memory allocation functions. The Z memory allocation routines have been designed to be flexible in the use of page registers. There is an array that maintains the busy status of each page register. Page registers can be made available for Z memory allocation, or set aside for use by the routines listed in the table, by the use of macros. The macro **PMblock_reg()** sets aside a page register, so it will not be used by the Z memory allocation routines. It takes as its argument the number of the page register. **PMblock_reg()** puts a non-zero value into the busy status array element for that page register. The opposite function is served by the macro **PMavail_reg()**, which puts a zero into that element. Because external memory is not always cleared when a program is restarted, it is a good idea to explicitly set the status of every page register before the Z memory allocation functions are used. The macro **PMset_lowreg()** is provided to reduce unnecessary searching through reserved page registers. Normally, the page register status array is searched from 0 to 13. If, say, only 12 and 13 are available, this is extra work for the machine. **PMset_lowreg()** can be used to restrict the search by setting the low register to 12.

After **PMgetzaddr()** returns, the segment of Z memory is in the address space of the program. The pointer can be used as any pointer would be used. When this memory is not in use, other Z memory may be used and **PMgetzaddr()** called, the page register should be freed. This is done with **PMfreezaddr()**. **PMfreezaddr()** does not free Z memory or modify the contents of Z memory. It only makes a page register available for use elsewhere. When memory needs to be reaccessed, the function **PMgetzaddr()** should be called again using the Z descriptor for that memory. The address may be different, but the contents will not be changed.

The procedures can be illustrated by the following code fragment:

```
#include "pxm.h"
#include "pageregs.h"

#define NULL (char *)0
#define MAXDESC             21

main()
{
        PMzdesc desc[MAXDESC];
        int i, j;
        char *ptrval1;
        char *tptr1;
        char *tptr4;
        char *msgptr = "Got memory on pass";

        PMzbrk(5); /* make 5k usable for general purpose */

        /* set limits on page registers */
        PMset_lowreg(10);
        PMset_hireg(13);

        for ( j = 0 ; 1 ; j++) { /* loop until error */
                desc[j] = PMgetzdesc(256);
                if ( !PMzdesc_valid(desc[j]) ) { /* no memory left */
                        printf("No DRAM memory available, pass %d0, j);
                        break;
                }
                if ( (ptrval1 = PMgetzaddr(desc[j])) == NULL ) {
                        /* no pointers (i.e. page registers) left */
                        printf("No Page Registers available, pass %d0, j);
                        break;
                }

                /* copy string to to the allocated mem in ZRAM */
                /* (this is generally not an efficient copy on DSP's) */
                for ( i = 0, tptr1 = ptrval1 , tptr4 = msgptr; i < 19 ; i++ ) {
                        *tptr1++ = *tptr4++;
                }
                /* print string from ZRAM */
                printf("%s %d0, ptrval1, j);
                /* block till printf is done so we don't change page reg before
                devprint needs to read the string */

                PMwaitsem();
                /* comment out next call and reuse desc's to get no page registers avail */
                PMfreezaddr(ptrval1);                /* make pointers available */
        }

        PMhost_exit();

}
```

Serial I/O Protocols

A Pixel Machine program using SIO progresses through three stages: initialization, setting link direction, and exchanging data. Each stage is explained in detail below, and a short example program demonstrating SIO is provided at the end of this section.

Initialization

Every pixel node program using SIO must call the DEVtools routine **PMsioinit** before any other SIO routines are called. **PMsioinit** initializes the SIO hardware and must be called only once in each program.

Setting Link Direction

To change the SIO link direction, call the DEVtools routine **PMsiodir**. This routine takes one parameter specifying the direction, which must be one of:

```
PM_MSG_SERIAL_NORTH
PM_MSG_SERIAL_SOUTH
PM_MSG_SERIAL_EAST
PM_MSG_SERIAL_WEST
```

These constants are defined in the header file **sysmsg.h**, and must be **#included** before **PMsiodir** is called.

PMsiodir requires that a host server process such as **devprint** (described earlier in this document), or a user program calling the host devlib library, be running on the host machine. **PMsiodir** sends a message to the server process requesting that a call to the host library routine **DEVserial_direction** be made to actually change the link direction.

All nodes must call **PMsiodir**.

Exchanging Data

After SIO is initialized and the link direction set, data packets may be exchanged with neighboring nodes. All nodes must transmit simultaneously and send exactly the same amount of data as all other nodes. The sequence of DEVtools calls to exchange a data packet is:

```
float *inbuf, *outbuf;
short size;

PMmsg_setup(inbuf);
PMpsync();
PMmsg_exchange(inbuf, outbuf, size);
```

The variables *outbuf* and *inbuf* are pointers to an output buffer, whose contents are sent in the link direction, and an input buffer which receives data from the opposite direction.

The call to **PMmsg_setup** sets up the SIO hardware to do DMA input to *inbuf*. Next, the **PMpsync** call ensures that all processors are synchronized and have set up their input buffers. Finally, **PMmsg_exchange** is called to exchange data packets. It sends *size* floats from *outbuf* and then waits until *size* floats have been received into *inbuf*.

Example Program

The short program below uses SIO to send a data packet from each node to its west neighbor, then sends the received data packet back to its starting point.

Serial I/O Protocols

A Pixel Machine program using SIO progresses through three stages: initialization, setting link direction, and exchanging data. Each stage is explained in detail below, and a short example program demonstrating SIO is provided at the end of this section.

Initialization

Every pixel node program using SIO must call the DEVtools routine **PMsioinit** before any other SIO routines are called. **PMsioinit** initializes the SIO hardware and must be called only once in each program.

Setting Link Direction

To change the SIO link direction, call the DEVtools routine **PMsiodir**. This routine takes one parameter specifying the direction, which must be one of:

```
PM_MSG_SERIAL_NORTH
PM_MSG_SERIAL_SOUTH
PM_MSG_SERIAL_EAST
PM_MSG_SERIAL_WEST
```

These constants are defined in the header file **sysmsg.h**, and must be **#included** before **PMsiodir** is called.

PMsiodir requires that a host server process such as **devprint** (described earlier in this document), or a user program calling the host devlib library, be running on the host machine. **PMsiodir** sends a message to the server process requesting that a call to the host library routine **DEVserial_direction** be made to actually change the link direction.

All nodes must call **PMsiodir**.

Exchanging Data

After SIO is initialized and the link direction set, data packets may be exchanged with neighboring nodes. All nodes must transmit simultaneously and send exactly the same amount of data as all other nodes. The sequence of DEVtools calls to exchange a data packet is:

```
float *inbuf, *outbuf;
short size;

PMmsg_setup(inbuf);
PMpsync();
PMmsg_exchange(inbuf, outbuf, size);
```

The variables *outbuf* and *inbuf* are pointers to an output buffer, whose contents are sent in the link direction, and an input buffer which receives data from the opposite direction.

The call to **PMmsg_setup** sets up the SIO hardware to do DMA input to *inbuf*. Next, the **PMpsync** call ensures that all processors are synchronized and have set up their input buffers. Finally, **PMmsg_exchange** is called to exchange data packets. It sends *size* floats from *outbuf* and then waits until *size* floats have been received into *inbuf*.

Example Program

The short program below uses SIO to send a data packet from each node to its west neighbor, then sends the received data packet back to its starting point.

Figure 3-13: SIO sample program

```
#include <pxm.h>
#include <sysmsg.h>

#define SIZE 100

main() {
        float inbuf[SIZE], outbuf[SIZE];
        short i;

        /* Fill the data packet with a sequence of numbers */
        for (i = 0; i < SIZE; i++)
                outbuf[i] = PMnode * i;

        PMsioinit();  /* Initialize sio */

        /* Set link direction */
        PMsiodir(PM_MSG_SERIAL_WEST);

        /* Send outbuf to the West, receive inbuf from the East. */
        PMmsg_setup(inbuf);
        PMpsync();
        PMmsg_exchange(inbuf, outbuf, SIZE);

        /* Reverse link direction */
        PMsiodir(PM_MSG_SERIAL_EAST);

        /* Send inbuf to the East, receive outbuf from the West. */
        PMmsg_setup(outbuf);
        PMpsync();
        PMmsg_exchange(outbuf, inbuf, SIZE);
}
```

Pixel Node Synchronization

Introduction

It is often necessary for the pixel nodes to synchronize themselves to ensure they have all reached the same stage of a computation before continuing to compute. An example is making sure that all processors have finished rendering their portion of a frame into the back buffer before switching it to become the visible buffer.

Hardware features of the Pixel Machine support this synchronization; alternatively, synchronization may be done in software with the aid of the host workstation. Both approaches are described in this section.

Hardware Synchronization

The basic DEVtools synchronization routine is named **PMpsync**. When called, this routine does not return until *all* pixel nodes have called **PMpsync**. This form of synchronization has very low overhead, returning within twelve instruction cycles of synchronization. Because **PMpsync** is so efficient, it is heavily used in internode communications routines.

The **PMvsync** DEVtools routine is similar to **PMpsync**. **PMvsync** is used to accomplish synchronization for tasks that change the displayed image; for example, before switching buffers in double-buffer mode. Like **PMpsync**, **PMvsync** waits until all processors have called it. It then waits for the beginning of a **vertical blanking interval** - when the electron beam of the monitor has reached the beginning of a field. **PMvsync** returns within twelve instruction cycles of the interval. It may take up to 1/60th second to return, depending on the position of the electron beam when the routine is called.

Since there is very little time between the start of the blanking interval and the time pixels in a frame begin to be displayed, **PMvsync** returns as soon as the interval is detected. The hardware signal used for synchronization must be turned off before another call to **PMvsync** is made, using another DEVtools routine **PMrdyoff**.

In summary, **PMvsync** is used as follows:

```
PMvsync();   /* Return after start of blanking interval */

(swap buffers or similar task)

PMrdyoff();  /* Disable VSYNC signal for the next call */
```

Note that in the normal case, **PMswapbuff**() should be used if the you intend to swap the visible buffer. **PMvsync**() and **PMrdyoff**() should only be used when synchronization with the vertical retrace is desired for some other purpose.

Software Synchronization

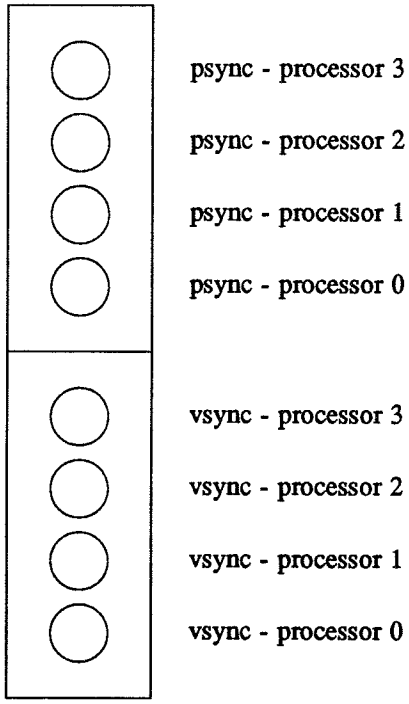
Another method of synchronizing processors requires the aid of the host workstation. The host polls the nodes, waiting for a software condition to be established. An example would be waiting for nodes to set their software semaphores to a specific value. After this condition is established and the work associated with it accomplished, the host establishes a different software condition in the nodes enabling them to proceed. This form of synchronization requires that the host and nodes agree upon the software protocol to be used. Many protocols may be used; below is a simple example using the software semaphore:

Host action	Node action
Wait for semaphore == 1 in all nodes	
	Set semaphore = 1
Take action upon synchronization	
	Wait for semaphore == 0
Set semaphore = 0 in all nodes	

Synchronization Signals and LEDs

The hardware mechanism used by **PMpsync** and **PMvsync** is visible in the form of LEDs on the Pixel Array Processor boards. The strip of 8 red LEDs on each board contains 2 LEDs for each pixel node. The upper 4 LEDs show the state of the **PMpsync** signal in each node. The lower 4 LEDs show the state of the **PMvsync** signal in each node:

Figure 3-14: LED layout on pixel node boards



These LEDs may be used for other purposes (such as a debugging aid) if the pixel node program in question makes no use of the corresponding synchronization calls (including calls to other DEVtools routines that require synchronization).

If no use is made of **PMpsync**, the upper LEDs may be turned on and off explicitly with the call **PMflagled**. This takes one integer argument. If nonzero, the LED is turned on, otherwise it is turned off.

Similarly, if no use is made of **PMvsync**, the lower LEDs may be turned on and off explicitly with the call **PMrdyled**, which takes the same on/off parameter as **PMflagled**.

Software Synchronization

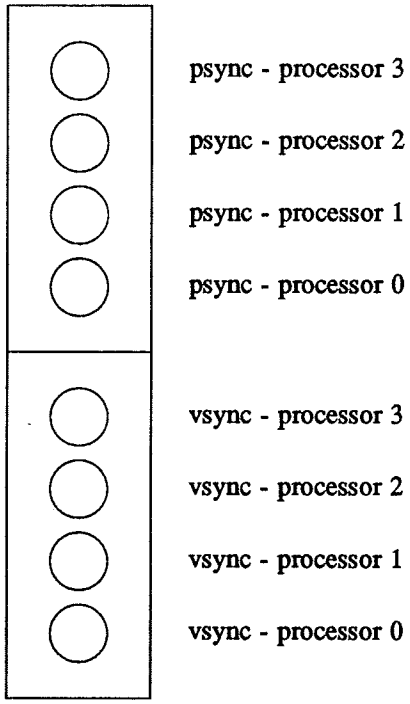
Another method of synchronizing processors requires the aid of the host workstation. The host polls the nodes, waiting for a software condition to be established. An example would be waiting for nodes to set their software semaphores to a specific value. After this condition is established and the work associated with it accomplished, the host establishes a different software condition in the nodes enabling them to proceed. This form of synchronization requires that the host and nodes agree upon the software protocol to be used. Many protocols may be used; below is a simple example using the software semaphore:

Host action	Node action
Wait for semaphore == 1 in all nodes	
	Set semaphore = 1
Take action upon synchronization	
	Wait for semaphore == 0
Set semaphore = 0 in all nodes	

Synchronization Signals and LEDs

The hardware mechanism used by **PMpsync** and **PMvsync** is visible in the form of LEDs on the Pixel Array Processor boards. The strip of 8 red LEDs on each board contains 2 LEDs for each pixel node. The upper 4 LEDs show the state of the **PMpsync** signal in each node. The lower 4 LEDs show the state of the **PMvsync** signal in each node:

Figure 3-14: LED layout on pixel node boards



These LEDs may be used for other purposes (such as a debugging aid) if the pixel node program in question makes no use of the corresponding synchronization calls (including calls to other DEVtools routines that require synchronization).

If no use is made of **PMpsync**, the upper LEDs may be turned on and off explicitly with the call **PMflagled**. This takes one integer argument. If nonzero, the LED is turned on, otherwise it is turned off.

Similarly, if no use is made of **PMvsync**, the lower LEDs may be turned on and off explicitly with the call **PMrdyled**, which takes the same on/off parameter as **PMflagled**.

Figure 3-15: Model 916

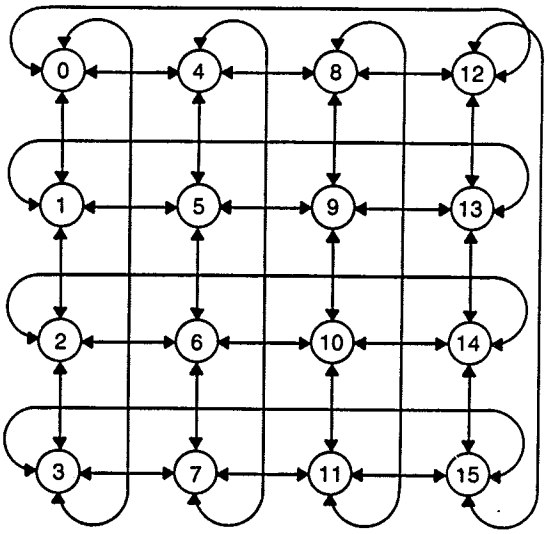


Figure 3-16: Model 920

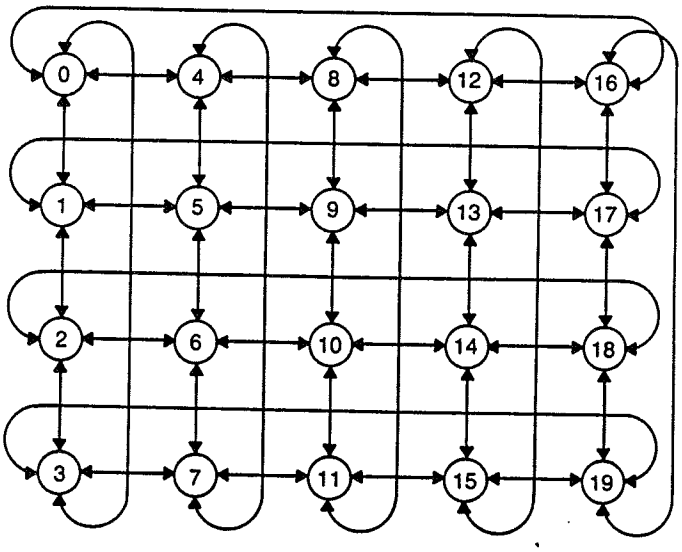


Figure 3-17: Model 932

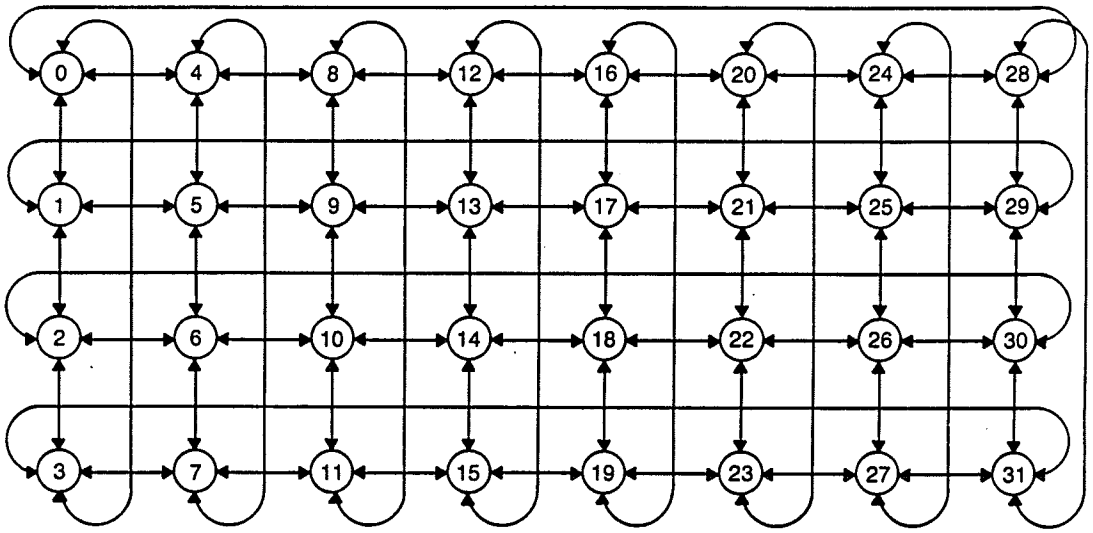


Figure 3-17: Model 932

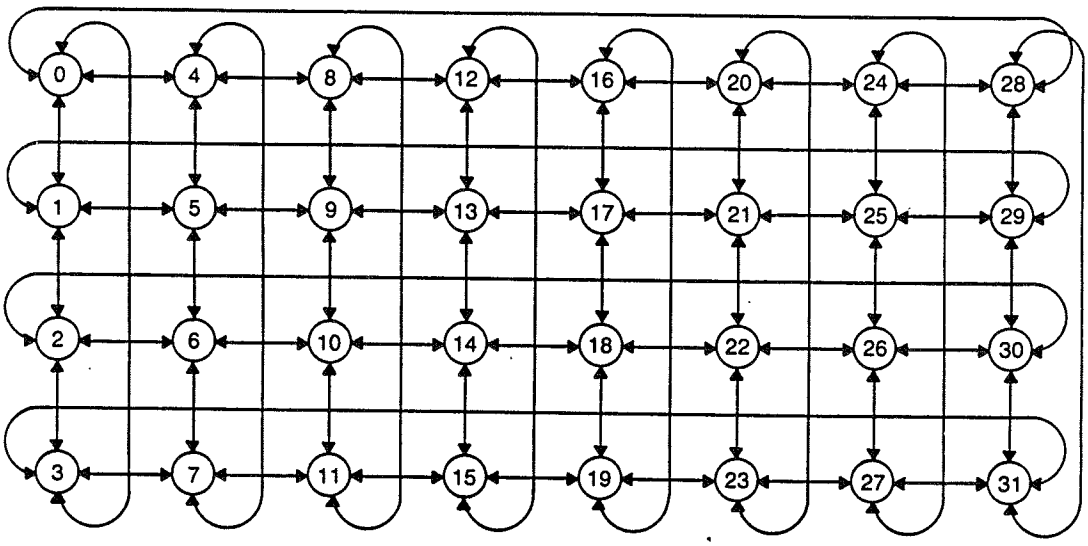


Figure 3-18: Model 940

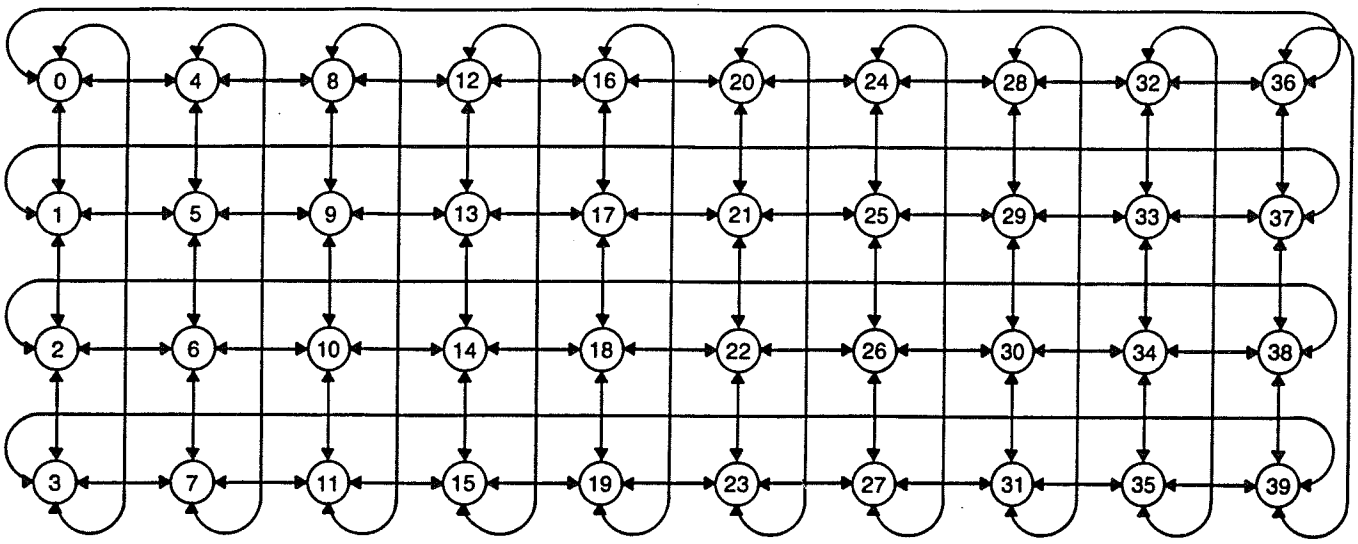
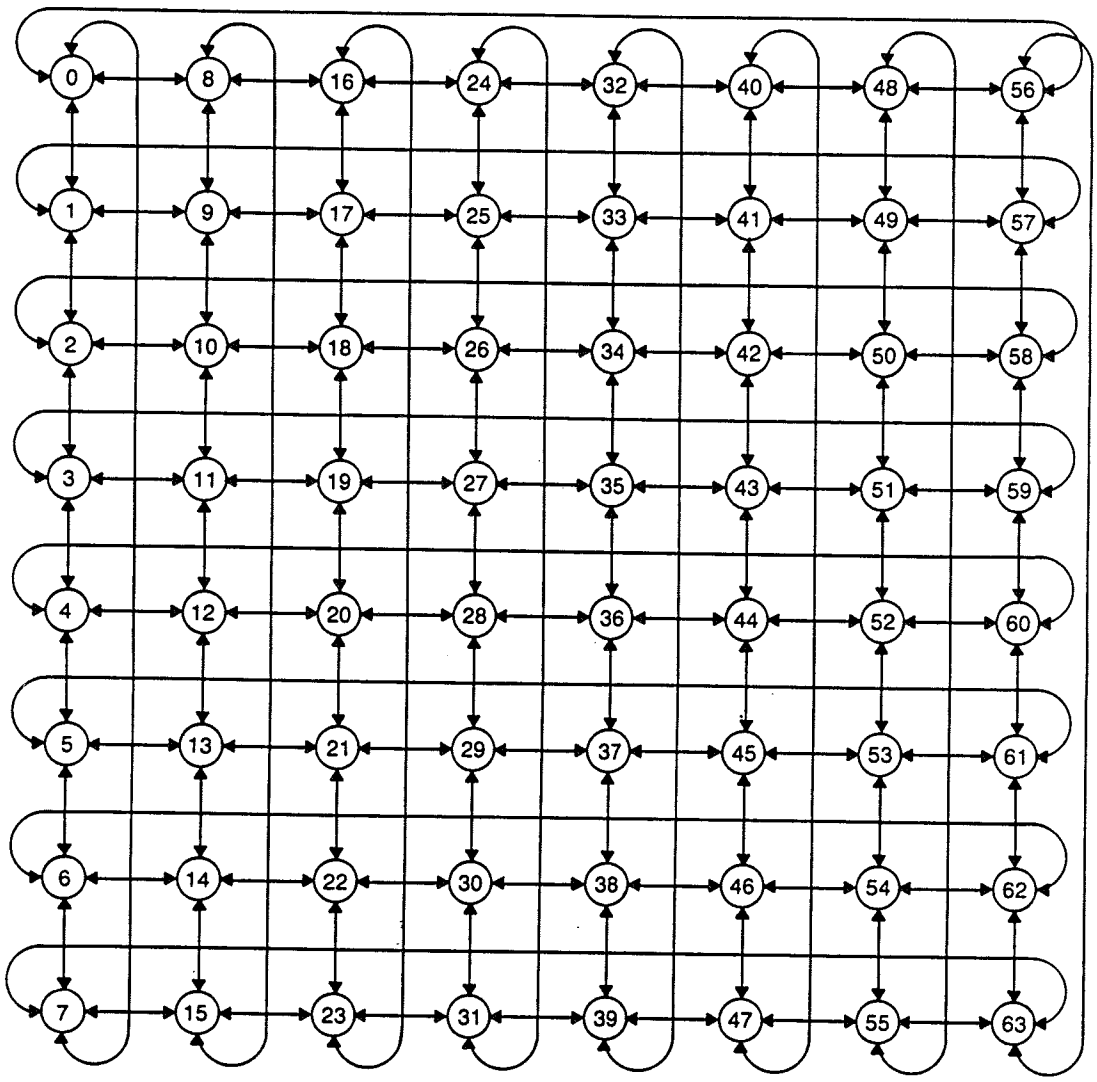


Figure 3-19: Model 964



Runtime Skeleton

The skeleton directory in **/usr/hyper/devtools/sample/skeleton** contains a sample of a complete Pixel Machine program, which means that all the architectural components of the Pixel Machine are used.

Sample Skeleton Program

The skeleton program is a sample of how to use command passing through the system. The main program boots the pipe and pixel nodes with their corresponding programs and starts the Pixel Machine running, and then enters the main loop. In the first part of the loop the host sends down commands to alternately clear the screen to red and then blue while flashing the FLAG LEDs. A delay command is also sent down between colors; the delay is shorter as the loop progresses towards the end.

In the next part of the loop, the host loops while sending down a random rgb color value and then a command to draw a random rectangle of that color. At the end of the main loop, the host sends down a single command, the DEV_GENERATE opcode. This opcode instructs a pipe node to generate many random rectangles on its own.

Finally, after the main loop exits, the host sends the pixel nodes a command to clean up and exit. It then calls **DEVpoll_nodes()** to wait for an exit code from the pixel nodes before exiting.

Debugging Code on the Pixel Machine

Introduction

Debugging programs that run on parallel computers is a task that strikes fear into the hearts of most programmers. Debugging code on the Pixel Machine, however, is much simpler than may be expected, and, in fact, is not much different than debugging on an ordinary computer. This is true because, in most cases, it is possible to debug a program on a single processor without worrying about what all of the other processors are doing. On the Pixel Machine this is possible because the processors do not share any of their memory with other processors. With the exceptions of serial DMA I/O with neighboring processors and parallel DMA I/O with the host, a single processor is in complete control of its environment.

Tools for General Debugging

Following are the tools available for general debugging:

- **printf:** the libpm library provides a version of **printf** that can be used to display data during the execution of a program on the Pixel Machine. The data is directed to the standard output of the controlling program running on the host. Print statements can then be used to display information as you would do on a conventional system.
- User Messages: using the message handling routines provided by devlib, a Pixel Machine program can send messages to the host indicating what the Pixel Machine program is doing, providing values of variables, etc. The host program can than check the sequence of the events, the values of the variables, etc.
- **hypeek** and **hypoke:** these commands allow you to display and modify data in a node's memory. They are useful for examining the data of a running process.
- **d3sim:** is the general purpose DSP simulator provided with the DSP Tools to simulate and debug programs written in DSP32. It can be used to debug Pixel Machine programs, except that it does not model any of the Pixel machine specific components such as FIFOs and frame buffers. For more information, see Chapter 6 of the *WE®DSP32 and DSP32C Support Software Manual.*

Tools for Debugging Pipe Routines

Programs that run in the pipe nodes usually perform operations similar to a UNIX system filter. They read input from the FIFO, perform some transformations, and output to the FIFO of the next processor. It is often useful to display the data that is the input to a given processor, and then to display the output of that processor (that is the input to the next processor). The program **/usr/hyper/boot/pipe_fb.dsp** can be loaded into a pipe node. It reads input from its FIFO and transfers the data to a host program through the PIR. A host program, called **hypfb**, can then read

the PIR data and display it.

For example, if you wanted to display the data that pipe node 0 is receiving as input, you would load **pipe_fb.dsp** into pipe node 0, run the host program that sends commands to the pipe, and run the program **hypfb** on the host. This would display all of the commands from the host as received by node 0.

If you wanted to display the output of node 0, you would load **pipe_fb.dsp** into pipe node 1, run the host program that sends the commands and run **hypfb**. This would display the output of node 0, that is also the input to node 1.

If the program you are debugging accepts commands in the format supported by the devlib command macros and functions, the **hypcmd** command can be used to translate the output of **hypfb** into a more readable format. To use **hypcmd**, simply pipe the output of **hypfb** into **hypcmd**. The commands:

```
hypload -g0 $HYPER_PATH/boot/pipe_fb.dsp
hyprun -g0
host_program
hypfb -g0 | hypcmd
```

will read the feedback information from pipe node 0 and translate it into command format. *host_program* must be run in the background or from a different window, because it and **hypfb** need to run at the same time.