A STANDARD SOFTWARE PLATFORM FOR SHARED MEMORY MULTIPROCESSOR SIGNAL PROCESSING SYSTEMS

by

Manish Arya

Memorandum No. UCB/ERL M90/92

5 October 1990

Cover Page

A STANDARD SOFTWARE PLATFORM FOR SHARED MEMORY MULTIPROCESSOR SIGNAL PROCESSING SYSTEMS

by

Manish Arya

Memorandum No. UCB/ERL M90/92

5 October 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Title page

A STANDARD SOFTWARE PLATFORM FOR SHARED MEMORY MULTIPROCESSOR SIGNAL PROCESSING SYSTEMS

by

Manish Arya

Memorandum No. UCB/ERL M90/92

5 October 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

To facilitate rapid prototyping of new signal processing designs we developed a standard hardware platform consisting of a general purpose host processor controlling multiple slave boards on a common bus. Processors on these boards locally control custom application specific circuitry; they communicate with the host through shared memory.

We recently developed system software, which is the subject of this report, to support the common hardware base. It provides a means of controlling the slaves from the host, synchronizing programs on the host with programs on the slaves, and communicating information between the host and slaves.

With this hardware/software platform, one can design, prototype, and debug a custom system supported by this master/slave model of control more quickly than without, concentrating mainly on the application-specific circuitry and software, and investing little effort in the supporting circuitry and software.

Our systems use a Heurikon 68020-based single board computer running the VxWorks operating system as the host on a VME bus and AT&T DSP32C digital signal processors as the slaves. However, since we implemented the software almost entirely in the language C, it should be easy to adapt to other hardware environments provided that they also fit within the multiprocessor, shared memory, single master, multiple slave architecture.

This report describes the design of the common software platform and provides all information necessary for developing applications in this environment.

Contents

List of Figures

List of Tables

Part I

General Description of System Functions

The first part of this document discusses general aspects of the system software and its operation. The second part covers those details of interest to those who use the system, develop software for it, or maintain it.

Chapter 1

Introduction

Today's complex signal processing systems involve a great deal more than just application-specific components. They often need local microprocessors to intelligently control the custom hardware; and, if they form parts of a larger system, they need a master microprocessor to coordinate the activities of all the subsystems. Two systems we are currently designing exemplify this basic structure. The sections that follow describe them.

1.1 An Image Processor

Bill Baringer, here at UC Berkeley, is developing an image processing system for machine vision applications capable of computing the radon transform (a projection operation in image space) in real time (consult [1] for more information). Among other tasks, this system will be capable of tracking the position and orientation of a polygonal object moving in the field of view of a video camera in real time to guide a robot's hand.

1.1.1 Custom Hardware

Bill has designed a custom ASIC (application specific integrated circuit) to perform the projection quickly in hardware. His system will string together multiple ASICs in a video pipeline to compute projections at varying angles all within a single frame time.

1.1.2 Local Support and Control

Although these ASICs form the heart of his system, they cannot perform any useful work in isolation. Some video support circuitry must supply the chips with the pixels in an image. Moreover, the projection vectors a bank of such chips produce are not useful results in themselves – an on-board microprocessor must analyze the projections to compute higher level features, such as the location and orientation of edges in the image. The microprocessor must also continuously update parameters that control the chips' behavior, including the regions of interest in the image and the angles of the projections.

1.1.3 Global Control

For higher accuracy and speed, Bill wishes to use more ASICs than can fit on one circuit board. Then yet another microprocessor must coordinate the boards. Using the individual boards to find different edges on the border of a polygonal object, this master processor must compute the location and orientation of the overall object. This processor must also control some of the other supporting devices, like the video frame grabber. And since the object position information is to guide a robot, this processor must also control the robot controller.

1.2 A Multi-axis Robot Controller

Mani Srivastava, also here at Berkeley, is implementing the six axis robot controller Gautam Doshi designed earlier (see [2] for details). It is similar to the image processor discussed above in its needs for microprocessor intelligence. It fits on a single board and contains one motion control chip per axis, each capable only of servoing a motor to a particular location. The local microprocessor performs the inverse kinematics necessary to take a high level command for motion to a new position (given as the Cartesian position and orientation of the robot hand) and to control the individual axes to produce the desired result.

The image processing and robot control systems are diagrammed in Figure 1.1.

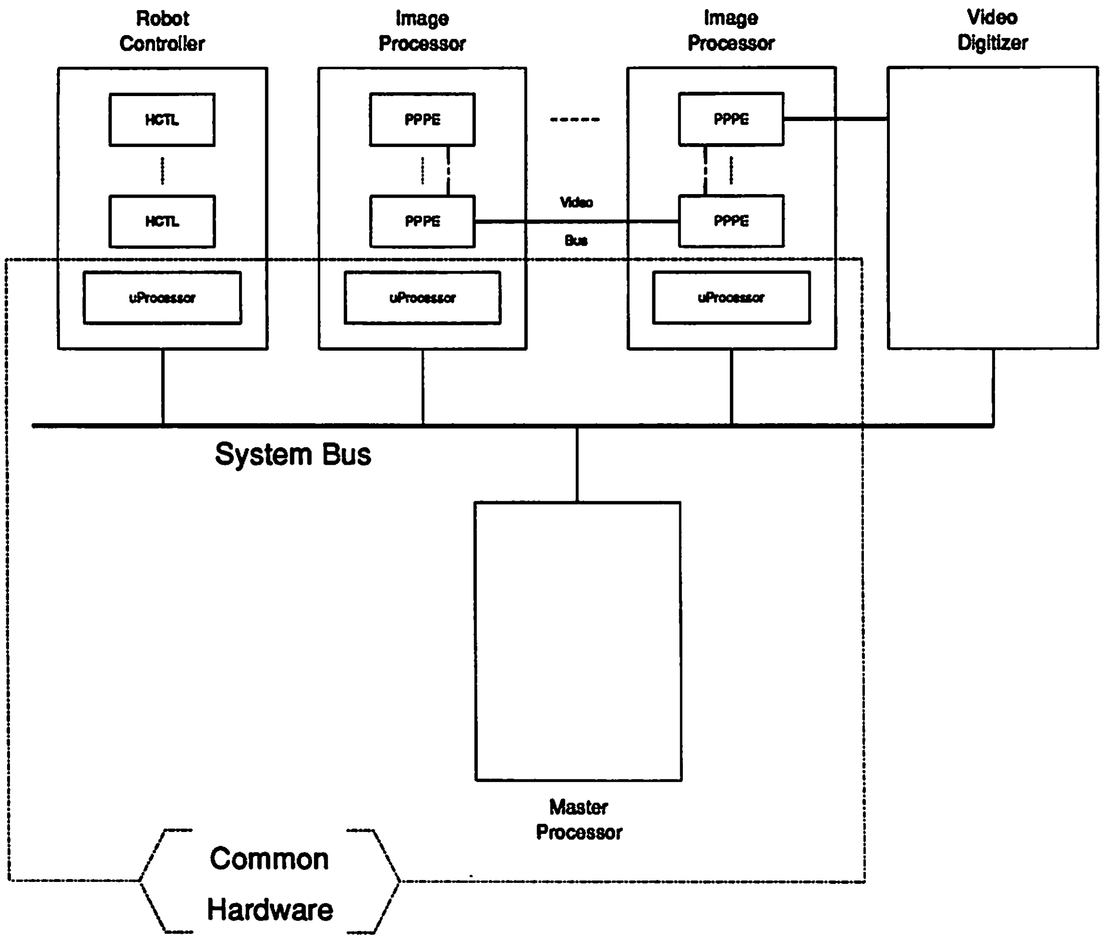

Figure 1.1: An image processing system (being developed by Bill Baringer), and robot control system (being developed by Mani Srivastava) working together in a machine vision application. The *HCTL* chips control the robot's servos, and the *PPPE* chips compute projections in the images that the digitizer captures from a camera.

1.3 Benefits of a Common Platform

We feel that the hierarchy of control evidenced in these machine vision and robot control systems is typical of many signal processing applications. Such systems have a great deal in common: the local slave microprocessors to control ASICs, the master processor to tie together the various subsystems, and the interface between the two.

All too often, we must start from scratch when we design a new system. We could speed up the design process substantially if we could standardize on a flexible, reusable hardware support platform. We could then concentrate mainly on the parts of the system which form the heart of the application. We would have to debug the common platform just once, and debugging the custom circuitry would then proceed more quickly. Furthermore, we would be able to integrate many such systems together more easily than if they were developed independently.

Reducing development time has been one of our research group's goals. The next chapter describes the common development platform we use to accommmplish this.

1.4 Common Software

Since the master and slave controllers are all microprocessors, they require software in order to function; so we can go one step further than just common hardware: common software as well.

This software must support some basic operations, such as initialization, synchronization of the slaves with the host, and communication between the two. It must provide a base on top of which the application specific software components can be written.

This report focusses on the design and implementation of a common system software platform. We aim to use this common software to help us further exploit the benefits of the common support hardware, cutting down on the time and effort required to design, implement, and test new signal processing systems.

Chapter 2

The Common Platform

2.1 Hardware

Figure 2.1 shows the hardware common to our systems. The application-specific boards are illustrated in greater detail in Figure 2.2. The boards are all interconnected via a VME bus. Gautam Doshi designed the board-level circuitry with the microprocessor, support components, and VME bus interface; he describes it in full detail in [2].

2.1.1 Host Processor

A Heurikon HK68/V20 series 20Mhz 68020-based single board computer serves as the master, or *host*, processor. It ties into our local area network through a CMC ENP-10 Ethernet Node Processor. The host runs *VxWorks*, a real-time, multitasking operating system which integrates well with Unix systems and supports most standard Unix network-based interprocess communications protocols. We interact with the host by logging into it remotely through a Sun workstation running SunOs Unix on the LAN. The host has no mass storage devices of its own; it accesses data on the LAN fileservers using the network file system (NFS) protocol.

2.1.2 Local Processors

Each board contains a 50Mhz AT&T DSP32C digital signal processor for performing local computations and also for controlling the application-specific

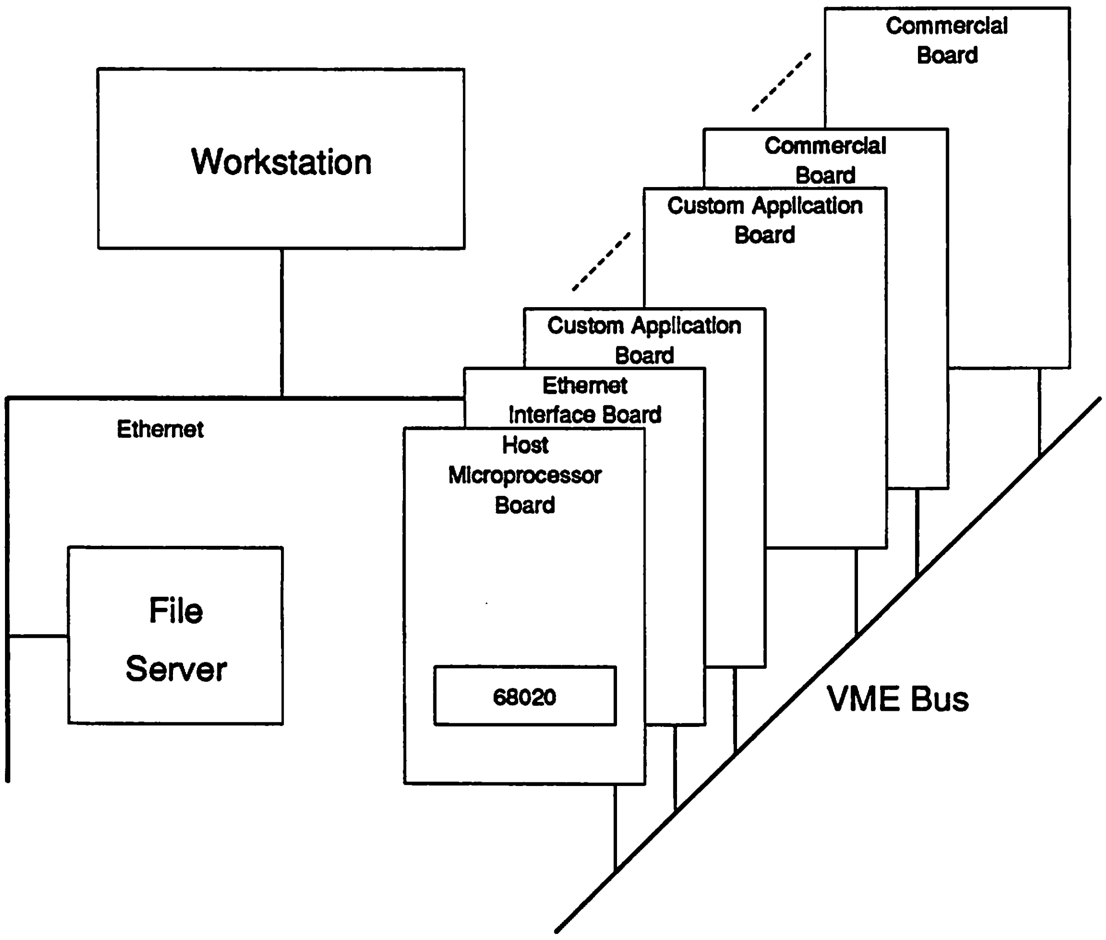

Figure 2.1: The common hardware platform.

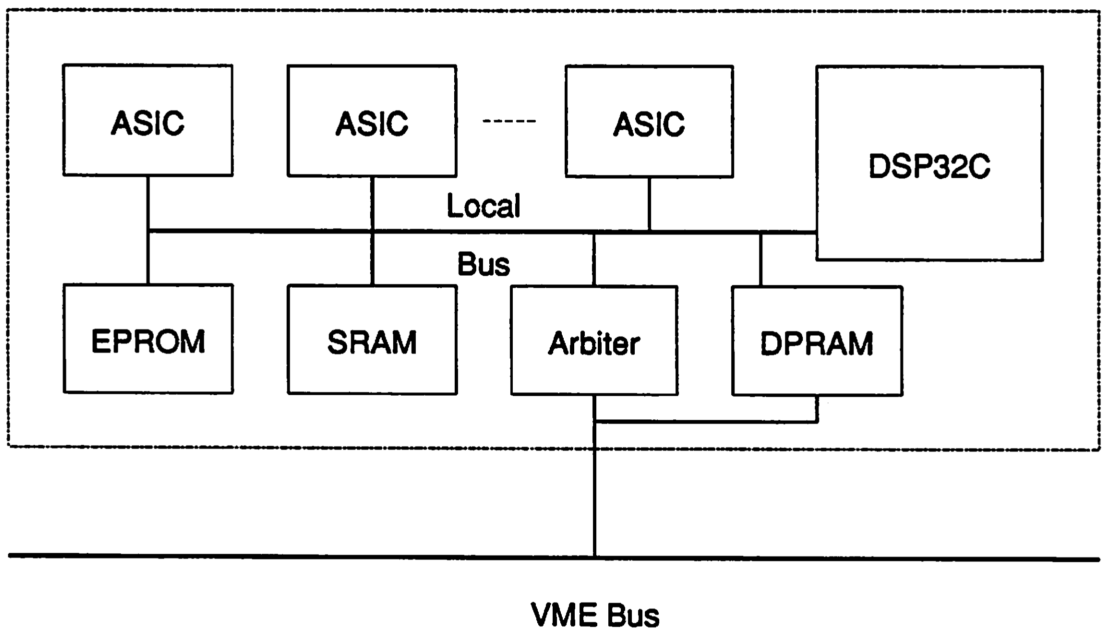

Figure 2.2: Detail of the application-specific boards in the common hardware platform.

components. Each processor has 6K bytes of on-chip static RAM and can access another 64K bytes of high-speed static RAM and 16K bytes of EPROM on the board. We refer to these local processors as *DSPs* throughout this report to distinguish them from the host processor.

2.1.3 Host ⇔ Board Interface

The host communicates with the processors on each board through shared memory: each board contains 16K bytes of 35ns dual-port static RAM (DPRAM) with one port attached to the VME bus and the other attached to the local processor's bus. As discussed in Section 3.2, the arbitration logic these particular DPRAM chips provide is not sufficient to support the synchronization needs of these systems; so each board also contains a separate arbitration circuit.

In addition, the host can access two registers on each board: a write-only control register and a read-only status register; both are mapped into the address space of the VME bus. Through these registers, the host can, for example, reset DSPs or interrupt them.

Similarly, each DSP can access a (different) pair of control and status registers. These registers allow DSPs to interrupt the host processor, for instance.

2.2 Software

The remainder of this report discusses in detail the common software platform, or *system software*, as we call it. The rest of this chapter summarizes some of its major aspects.

2.2.1 Implementation

We have written the common system software almost entirely in the language C. AT&T provides a C cross-compiler and linker which run on our workstations to produce executable binary code for the DSPs.

We have written the code in a manner that should make it easy to port to different hardware platforms, provided that they also feature a master processor controlling multiple slaves with a shared memory interface in between.

Appendix E explains how to convert the software to such new environments.

2.2.2 Data Types

The basic data type which the host and DSPs share is the 32-bit integer word. We refer to this as a *word* throughout this report. Section 10.1 discusses the system data types in more detail; however, the extra information there should not be necessary for following the material in the first part of this report.

2.2.3 Communication Model

The communications model we wish to support dictates the design of the synchronization and communications services. Specifically, we require that:

- Any DSP may communicate with the host.
- Any host task may communicate with any DSP.
- Multiple host tasks may communicate with the same DSP,

Note that DSP's may not communicate with one another directly; if they wish to exchange information, the host must mediate.

Chapter 4 describes the communications support in the system software, and Chapter 3 describes the synchronization services that the communications routines employ.

2.2.4 DSP Monitor Program

The host processor is an off-the-shelf computer for which we can obtain an off-the-shelf operating system: VxWorks. However, since the DSP32C section of the hardware platform is a custom design, we must provide our own small operating system, or monitor for it. Our monitor responds to commands from the host. Running on each DSP in the system, it provides the ability to:

- Examine memory local to the DSP.
- Alter memory local to the DSP.

- Load an application program into the DSP's local memory.
- Run a previously loaded application program.

Chapter 8 discusses the monitor program in depth.

A bootstrap loader, described in Chapter 7, is the piece of code each DSP actually executes first when reset. It is responsible for loading the monitor program into each DSP's local memory and transferring control to it.

2.2.5 Remote DSP I/O Console

Debugging code running on the DSPs can be difficult since we do not have a debugger that can interactively execute programs there under control of the host. This is due in part to the fact that the DSPs have no console for input and output: the DPRAM is their only means of communicating with the rest of the world.

As a first step toward aiding the debugging process, we implemented in software a remote input-ouput console for each DSP. DSP C programs may call special versions of *printf()* and *scanf()* in order to send output to or obtain input from this console. Users interact with this console through their workstations, typically in a separate window. The console is linked to every DSP through the network and the DPRAM. Chapter 5 describes this feature more thoroughly.

2.2.6 Writing Applications

All of the services the system software provides are implemented by a set of functions that are packaged together into libraries to which application programs can link.

Makefiles, header files with common definitions and declarations, lint libraries, and man pages supplement the libraries.

Chapter 9 details the process of writing an application program on this platform.

2.2.7 Some Conventions

We have followed certain guidelines in writing the system software. Understanding them should make reading the pseudocode throughout this report

Prefix	Type of Symbol
vdi...	variable
pVdi...	pointer variable
Vdi...	function or type
VDI...	constant

Table 2.1: Symbol naming conventions.

simpler. First, to prevent a clash between system symbol names and application symbol names, all globally visible symbols include the letters *vdi* in a prefix, which, for lack of a better idea, form an acronym for something like *VxWorks to DSP Interface* or *VME to DSP Interface*. Table 2.1 summarizes the meanings of these prefixes.

Second, most functions indicate success by returning a status code of zero and indicate failure by returning some non-zero error code. For simplicity, however, the pseudocode in this report does not show the function return code checking the system routines actually perform.

Chapter 3

Synchronization of Host Tasks with DSP Programs

In order to establish a protocol for communication between the host and DSPs, we must first provide a means of arbitrating access to the dual port memories. When the host and DSP simultaneously access different memory cells, no problem arises. However, when the two agents access the same cell, they will clash if either access is a *write* operation.

3.1 Locking Regions of DPRAM

We need a primitive atomic locking operation that allows an agent (host or DSP) to temporarily claim exclusive ownership of a region of the DPRAM so that it may write information into it or read information from it without concern that the other agent might simultaneously access a location in the same region.

This in itself forms the basis for a simple form of communication between the two agents. The host can lock a region of DPRAM, write information into it, and unlock it; the corresponding DSP can then lock the region, read the information from it, and unlock it.

However, the system software actually supports a more versatile form of communication using the higher-level synchronization primitive discussed in Section 3.4.

3.2 Insufficient Existing Support for Locking

The DPRAM chips on each board feature some arbitration circuitry to help detect write-write, read-write, and write-read contention. Whenever such a situation arises, the chips make an arbitration decision and honor the transaction on one port while disregarding that on the other port. They also set a *busy* flag for the port whose access they denied; the host and DSPs can read latched versions of these busy flags through their respective status registers.

The DPRAM chips are organized in 32-bit wide banks to allow the host to make 32-bit accesses via the VME bus (indeed the host *must always* make 32-bit accesses to the DPRAM locations). Normally, one of the four 8-bit wide chips acts as a master. It makes the arbitration decisions and communicates them to the other three, which act as slaves. If all chips are masters, some may rule in favor of one port and others in favor of the other port during a simultaneous access.

As described in [2], the DPRAM chips are not operated in the master-slave mode in this hardware interface because, unlike the host, the DSPs are allowed to make 8-, 16-, or 32-bit accesses to the DPRAM. Instead, they are all "masters", and on each side their four busy signals are logically ORed together to form the status register busy signal.

The original plan called for the software (on both the host and DSPs) to perform the following steps when accessing a DPRAM location:

1. Clear the busy bit latch by writing to the control register.
2. Access the DPRAM.
3. Check the busy bit in the status register.
4. If the busy bit indicates contention, repeat the process.

This involves a great deal of overhead, and was to be done only when checking or updating synchronization variables in the DPRAM to control a higher level communication protocol.

However, a detailed look at the data sheet for these particular DPRAM chips and the design of this DPRAM interface shows that no simple, efficient software scheme exists for using this hardware to implement the primitive

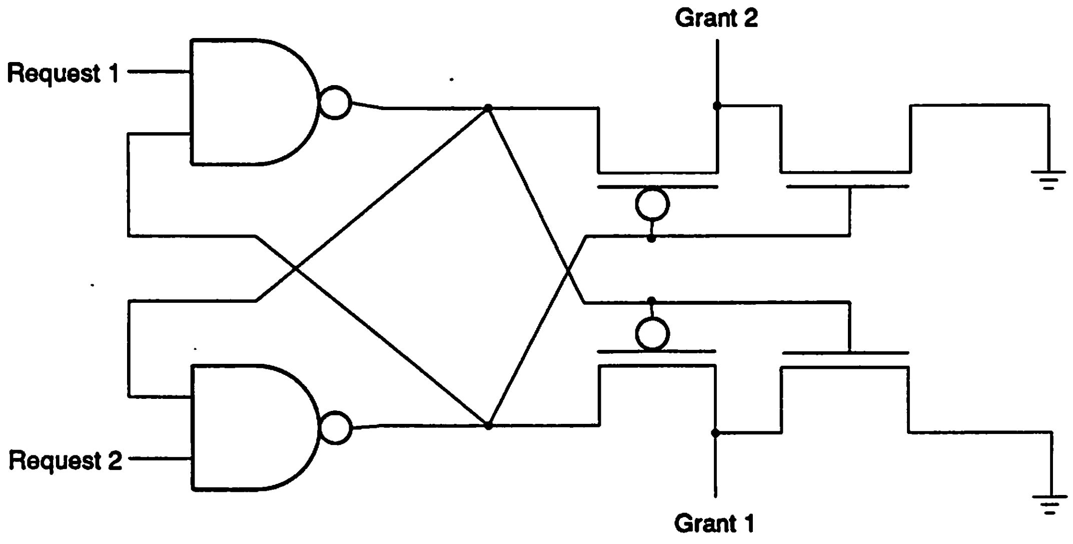

Figure 3.1: An arbitration circuit which supports the locking operation necessary for synchronization.

locking operation necessary for synchronization. The problem relates to the fact that all chips are masters and parts of two simultaneous DPRAM transfers may succeed, with neither side knowing from a "busy" indication whether its access failed completely or just partially.

3.3 Arbiters for Locking Parts of DPRAM

We opted to provide direct hardware locking support by including the circuit in Figure 3.1 on each board in addition to the interface circuitry described in [2]. The request and grant lines are attached to the control and status registers, respectively.

We conceptually associate the arbiter with a region of DPRAM. When an agent wishes to exclusively access that region, it must:

1. Assert its request line.

2. Wait for its grant line to go high.

3. Access the DPRAM.
4. Disassert its request line.

An asserted grant line corresponds to possession of the lock and permission for access to the protected region.

This particular arbiter has the following desirable properties:

- *Mutual exclusion*: it will ***never*** grant both agents permission simultaneously, even if the latch composed of the cross-coupled NAND gates enters a meta-stable state (which may occur if both agents assert their request lines simultaneously).
- *Speed*: the arbiter makes its decision in less than one processor cycle time.
- *Fairness*: if one agent requests permission after the other agent has been granted permission, the first agent will receive permission as soon as the other agent disasserts its request line. Thus two busy agents will take turns if they need frequent access to the locked region.

Note that because of the possibility of metastability, we must modify our procedure for accessing the DPRAM. An agent must:

1. Assert its request line.
2. Wait for its grant line to go high. If the grant line is inactive after a fixed number of attempts, the arbiter may be in a metastable state, so it must disassert its request line and return to step 1.
3. Access the DPRAM.
4. Disassert its request line.

Figures 3.2 and 3.3 show the procedures for locking and unlocking a region with an arbiter. The parameter *pDSP* is a pointer to a structure with information about the DSP to which the arbiter *a* is connected.

The transistors in this circuit mandate a chip-level implementation, which we will pursue later. For the moment, we are using the slightly different circuit of Figure 3.4, implemented in a PLD. Unlike the other circuit, however, this one might violate the mutual exclusion property by activating both grant lines if metastability sets in. This is a rare occurrence which we have not yet witnessed in our experiments.

```
void VdiLock(pDSP, a)
    VdiDSP *pDSP;
    VdiArbiter *a;
{
    int timeOut;

    for ( ; ; ) {
        timeOut = VDI_ATTEMPTS;
        VdiAssert(pDSP, a->request);
        while (!VdiActive(a->grant) && (--timeOut))
            ;
        if (timeOut)
            break;
        VdiDisassert(pDSP, a->request);
    }
}
```

Figure 3.2: The procedure VdiLock() for locking a region with an arbiter.

```
void VdiUnlock(a)
    VdiDSP *pDSP;
    VdiArbiter *a;
{
    VdiDisassert(pDSP, a->request);
}
```

Figure 3.3: The procedure VdiUnlock() for unlocking a region with an arbiter.

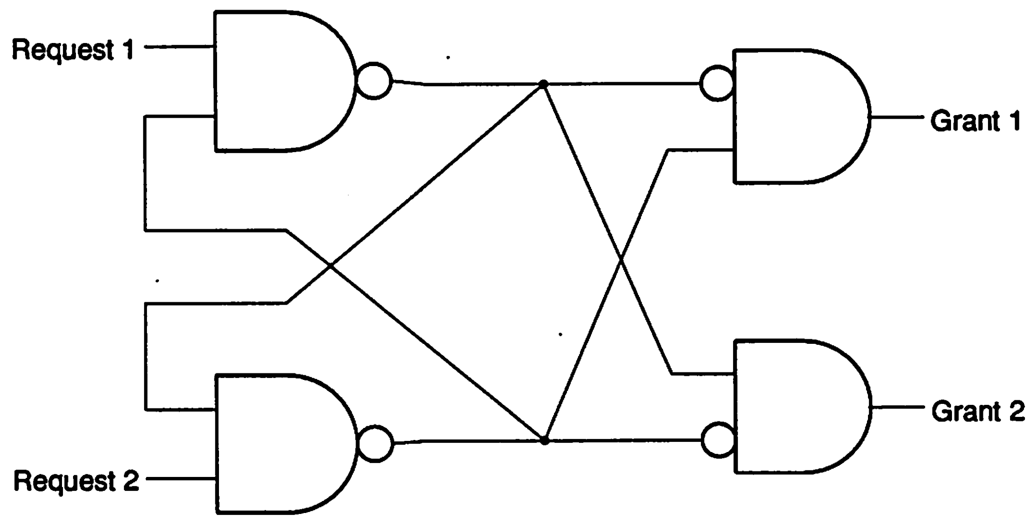

Figure 3.4: A modified arbitration circuit which can be implemented out of simple gates in a PLD.

3.4 Semaphores

Semaphores are more versatile synchronization devices than the simple locking scheme described so far, and they can be implemented easily in software using this primitive locking operation. For a more detailed discussion of semaphores and their implementation than that below, consult [5]. Chapter 4 illustrates the utility of semaphores in the system software.

3.4.1 Interpretation

Physically, a semaphore is just a counter that can take on non-negative integer values. Its value indicates the number of units of some scarce resource that are available. A system may have more agents wishing access to the resource than it has units of that resource, each of which only one agent can use at a time. A computer system may, for example, have a limited number of buffers for holding a certain type of data. The system initializes a semaphore to this number. To provide mutual exclusion as does the locking primitive, the initial value of the semaphore is simply set to one to indicate that only one "key" is available to access the protected region.

3.4.2 Locking

In the case of the system software, we wish to synchronize the host with each DSP, so the semaphores must reside in the DPRAM where both agents can access them. Moreover, because both agents might otherwise access the semaphore variables at any time, they must be protected by the locking mechanism.

3.4.3 Operations

The two basic operations on semaphores are *take()* and *give()*, or, synonymously, *p()* and *v()*. When an agent wants a unit of the resource, it performs a *take()* operation on the associated semaphore; the function *take()* will wait until a unit of the resource is available. When finished, the agent performs a *give()* operation.

More generally, we can allow agents to take or give multiple units of a resource in a single operation by specifying this number in an extra argument to the functions *take()* and *give()*.

In some situations we will want to take a semaphore if enough units are available, and if not, to simply know the fact rather than wait for units to be freed by another agent. This *non-blocking* behavior is useful for implementing some variants of the higher-level communication primitives discussed in Chapter 4.

Taking a Semaphore

More formally, Figure 3.5 defines the *VdiSemaphoreP()* operation for code running on the DSPs: The parameter *pDSP* identifies the DSP in whose DPRAM the semaphore *s* resides. The parameter *n* indicates the number of units desired, and *nonBlocking*, if *true*, indicates that the call must not wait if enough units are not available, but return a non-zero value instead.

For the host side, we must modify this routine slightly because many tasks may run concurrently on the host and try to access a semaphore for a particular DSP simultaneously. Figure 3.6 illustrates the procedure. It

```
int VdiSemaphoreP(pDSP, s, n, nonBlocking)
    VdiDSP *pDSP;
    VdiSemaphore *s;
    int n;
    int nonBlocking;
{
    /* Wait until n units are available. */
        for ( ; ; ) {
            VdiLock(pDSP, s->arbiter);
            if (s->value >= n)
                break;
            VdiUnlock(pDSP, s->arbiter);
            if (nonBlocking)
                return(1);
        }

    /* Decrement the semaphore. */
        s->value -= n;

    /* Unlock the semaphore. */
        VdiUnlock(pDSP, s->arbiter);

    return(0);
}
```

Figure 3.5: The procedure VdiSemaphoreP() for the DSPs.

ensures that only one host task accesses the arbiter for a given DSP at a time. The VxWorks operating system supports semaphores for synchronizing tasks under its control; the procedure uses one VxWorks semaphore (initialized to the value 1) for each DSP in the system to satisfy the constraint. The functions *semTake()* and *semGive()* operate on VxWorks semaphores.

Giving a Semaphore

Figure 3.7 illustrates the procedure a DSP must follow to give a semaphore, while Figure 3.8 shows the procedure for the host. The parameters are similar to those for *VdiSemaphoreV()*, except that *n* indicates how many units to give up.

3.4.4 Conserving Arbiters

The system software makes use of multiple semaphores for each DSP. Upon first thought, this indicates that we need one arbiter for each semaphore. However, note that a semaphore need only be locked for short periods of time within the *VdiSemaphoreP()* and *VdiSemaphoreV()* routines.

If we use a single arbiter, then only the host or DSP (not both), may access one of the semaphores (for that DSP) at any given time. Because the time that a semaphore must be locked is short, having an agent wait in order to lock its semaphore because the other agent is accessing a (potentially) different semaphore does not significantly degrade performance. Furthermore, just as one arbiter suffices for all semaphores for a given DSP, one local VxWorks semaphore suffices as well.

This is the approach we have taken in the system software, necessitating only one arbiter per DSP. Therefore, contrary to what Figures 3.2 through 3.8 show, the routines *VdiLock()* and *VdiUnlock()* do not need a parameter specifying which arbiter to use (since the parameter *pDSP* implies it).

```
int VdiSemaphoreP(pDSP, s, n, nonBlocking)
    VdiDSP *pDSP;
    VdiSemaphore *s;
    int n;
    int nonBlocking;
{
    /* Wait until n units are available. */
        for ( ; ; ) {
            semTake(s->local);
            VdiLock(pDSP, s->arbiter);

            if (s->value >= n)
                break;

            VdiUnlock(pDSP, s->arbiter);
            semGive(s->local);

            if (nonBlocking)
                return(1);
        }

    /* Decrement the semaphore. */
        s->value -= n;

    /* Unlock the semaphore. */
        VdiUnlock(pDSP, s->arbiter);
        semGive(s->local);

    return(0);
}
```

Figure 3.6: The procedure VdiSemaphoreP() for the host.

```
void VdiSemaphoreV(pDSP, s, n)
    VdiDSP *pDSP;
    VdiSemaphore *s;
    int n;
{
    /* Lock the semaphore. */
        VdiLock(pDSP, s->arbiter);

    /* Increment the semaphore. */
        s->value += n;

    /* Unlock the semaphore. */
        VdiUnlock(pDSP, s->arbiter);
}
```

Figure 3.7: The procedure VdiSemaphoreV() for the DSPs.

```
void VdiSemaphoreV(pDSP, s, n)
    VdiDSP *pDSP;
    VdiSemaphore *s;
    int n;
{
    /* Lock the semaphore. */
        semTake(s->local);
        VdiLock(pDSP, s->arbiter);

    /* Increment the semaphore. */
        s->value += n;

    /* Unlock the semaphore. */
        VdiUnlock(pDSP, s->arbiter);
        semGive(s->local);
}
```

Figure 3.8: The procedure VdiSemaphoreV() for the host.

Chapter 4

Communication between Host Tasks and DSP Programs

The system software supports communication between the host and DSPs through the shared memories. It formats sections of these memories into unidirectional first-in-first-out (FIFO) queues. An agent may place information into a queue, from which the recipient may read it when ready. This way, the sender can proceed with other computations without waiting for the recipient to accept the information. (Nevertheless, if the sender requires an acknowledgement, it can wait for a response from the recipient in another queue.)

4.1 Queue Layout

Within each DPRAM, one queue holds host-generated data destined for the corresponding DSP. Another queue holds DSP-generated data destined for the host. The two queues are located at predetermined addresses within the DPRAM.

A few other queues, also located in the DPRAM, help implement the remote DSP input-output console; Chapter 5 discusses them.

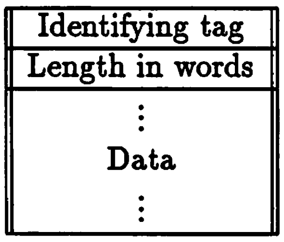

Figure 4.1: The structure of a queue packet.

4.2 Queue Packets

We refer to the basic unit of information that may be placed into a queue as a *packet*. Figure 4.1 shows its structure. Packets are stored contiguously in queues. The *data* field can vary in length, but packets must be small enough to fit inside a queue's buffer.

The *tag* field simply serves to identify the contents of the *data* field, whose format is the responsibility of the communicating agents.

4.3 Queue Structure

The queues consist of a header and circular buffer, as shown in Figure 4.2. The *lock* semaphore guarantees the consistency of the queue: agents must take this semaphore before accessing the queue and give it up when finished.

The *number of packets* semaphore counts the number of packets waiting in the queue. An agent wishing to take a packet from the queue must take one unit of this semaphore. An agent putting a packet into the queue must give up one unit of this semaphore.

The *free room* semaphore counts the number of words of room that remain in the queue. An agent wishing to put a packet into the queue must first take enough units of this semaphore to hold the packet header and contents. An agent removing a packet from the queue must give back the number of units occupied by the packet.

The *head* and *tail* pointers mark the ends of the queue within the circular *buffer* that holds the packets.

Lock (semaphore)
Number of packets (semaphore)
Free room in words (semaphore)
Head pointer
Tail pointer
⋮ Buffer ⋮

Figure 4.2: The structure of a queue.

4.4 Queue Operations

The two standard operations which agents may perform on a queue are *put()* and *get()* to add packets and remove packets from a queue, respectively. Sections 4.4.1 and 4.4.2 describe them.

These operations, however, require that the data fit entirely within a single packet. Often, agents must exchange blocks of data larger than the queues' capacities. For these situations, two similar procedures operate on what we call *messages*, or simply large packets; sections 4.4.3 and 4.4.4 describe these.

4.4.1 Getting a Packet from a Queue

Figure 4.3 illustrates the procedure *VdiQueueGet()*. The parameter *pDSP* identifies the DSP in whose DPRAM the source queue *pQueue* resides. The parameters *pTag* and *pLength* point to variables in which *VdiQueueGet()* is to return the corresponding information about the packet. The parameter *pData* points to a buffer of length *room* to receive the packet contents. The parameter *nonBlocking*, if *true*, indicates that *VdiQueueGet()* should not wait if a packet is unavailable.

It is this non-blocking variation of *VdiQueueGet()* (and *VdiQueuePut()*) that necessitates the non-blocking variation to *VdiSemaphoreP()*. Some applications may not wish to wait when a queue is empty in order to continue

```
int VdiQueueGet(pDSP, pQueue, pTag, pLength, pData, room,
                nonBlocking)
    VdiDSP *pDSP;
    VdiQueue *pQueue;
    int *pTag, *pLength;
    int room;
    void *pData;
    int nonBlocking;
{
    /* Wait for a packet to appear in the queue. */
        if (VdiSemaphoreP(pDSP, pQueue->packets, 1, nonBlocking))
            return(VDI_ERROR_QUEUE_EMPTY);

    /* Lock the queue. */
        VdiSemaphoreP(pDSP, pQueue->lock, 1, FALSE);

    /* Extract the header information from the next packet. */
        *pTag = ... ;
        *pLength = ... ;

    /* Copy the packet data. */
        memcpy(pData, ..., min(*pLength, room));

    /* Update the queue pointer(s). */
        pQueue->head = ... ;

    /* Unlock the queue. */
        VdiSemaphoreV(pDSP, pQueue->lock, 1);

    /* Update the free room semaphore. */
        VdiSemaphoreV(pDSP, pQueue->room,
                      pLength + VDI_PACKET_HEADER_LENGTH);

    return(0);
}
```

Figure 4.3: The procedure VdiQueueGet() for removing packets from a queue.

with other processing.

4.4.2 Putting a Packet into a Queue

Figure 4.4 illustrates the procedure *VdiQueuePut()*. The arguments to *VdiQueuePut()* are analogous to those of *VdiQueueGet()*, except that *tag* and *length* are inputs describing the new packet to whose contents *pData* points. Here *nonBlocking*, if *true*, instructs *VdiQueuePut()* not to wait if the destination queue currently does not have room for the new packet.

4.4.3 Sending a Message

Figure 4.5 shows the procedure *VdiMessageSend()*, which is analogous to and built on top of *VdiQueuePut()*. It allows agents to send arbitrarily large blocks of data by automatically breaking them up into multiple packets small enough to fit within a queue. *VdiMessageSend()* first sends a *start of message* packet that contains the parameters *tag* and *length*. It then sends multiple *message data* packets with the actual contents of the buffer to which *pData* points. Afterwards, it sends an *end of message packet.*

4.4.4 Receiving a Message

Figure 4.6 shows the procedure *VdiMessageReceive()*, which is analogous to *VdiQueueGet()* and complements *VdiMessageSend()*. It expects the *start of message*, *message data*, and *end of message* packets that *VdiMessageSend()* generates.

4.5 Performance

We conducted a few experiments to benchmark the communication time. In the first test, the host writes a packet into a queue, a DSP reads it from that queue and writes it into another queue, and the host then reads it back from the second queue. This involves four queue operations: a *VdiQueueGet()* and *VdiQueuePut()* on each side. Table 4.1 shows the results of this full loopback test for two different packet sizes.

```
int VdiQueuePut(pDSP, pQueue, tag, length, pData, nonBlocking)
    VdiDSP *pDSP;
    VdiQueue *pQueue;
    int tag, length;
    void *pData;
    int nonBlocking;
{
    /* Wait for room to hold the packet. */
        if (VdiSemaphoreP(pDSP, pQueue->room, 1,
                          length + VDI_PACKET_HEADER_LENGTH))
            return(VDI_ERROR_QUEUE_NO_ROOM);

    /* Lock the queue. */
        VdiSemaphoreP(pDSP, pQueue->lock, 1, FALSE);

    /* Write the header information for a new packet. */
        ... = tag;
        ... = length;

    /* Copy the packet data. */
        memcpy(..., pData, length);

    /* Update the queue pointer(s). */
        pQueue->tail = ... ;

    /* Unlock the queue. */
        VdiSemaphoreV(pDSP, pQueue->lock, 1);

    /* Update the packet count semaphore. */
        VdiSemaphoreV(pDSP, pQueue->packets, 1);

    return(0);
}
```

Figure 4.4: The procedure VdiQueuePut() for adding packets to a queue.

```
int VdiMessageSend(pDSP, pQueue, tag, length, pData)
    VdiDSP *pDSP;
    VdiQueue *pQueue;
    int tag, length;
    void *pData;
{
    VdiMessage m;
    int size;

    /* Send the start of message packet. */
        m.tag = tag;
        m.size = length;
        VdiQueuePut(pDSP, pQueue, VDI_TAG_MESSAGE_START,
                    VDI_WORDS(sizeof(VdiMessage)), &m, 0);

    /* Send the message contents. */
        do {
            size = min(length, pQueue->capacity);
            VdiQueuePut(pDSP, pQueue, VDI_TAG_MESSAGE_DATA,
                        size, pData, 0);
            length -= size;
            (VdiWord *)pData += size;
        } while (length > 0);

    /* Send the end of message packet. */
        VdiQueuePut(pDSP, pQueue, VDI_TAG_MESSAGE_END,
                    0, NULL, 0);

    return(0);
}
```

Figure 4.5: The procedure VdiMessageSend() for sending messages via a queue.

```
int VdiMessageReceive(pDSP, pQueue, pTag, pLength, pData, room)
    VdiDSP *pDSP;
    VdiQueue *pQueue;
    int *pTag, *pLength;
    void *pData;
    int room;
{
    int tag, length;
    VdiMessage m;

    /* Get the message tag and size. */
        VdiQueueGet(pDSP, pQueue, &tag, &length, &m,
                    VDI_WORDS(sizeof(VdiMessage)), 0);
        if (tag != VDI_TAG_MESSAGE_START)
            return(VDI_ERROR_MESSAGE_BAD);
        *pTag = m.tag;
        *pLength = m.length;

    /* Get the message contents. */
        do {
            VdiQueueGet(pDSP, pQueue, &tag, &length, pData,
                        room, 0);
            (VdiWord *)pData += length;
            room -= length;
        } while (tag == VDI_TAG_MESSAGE_DATA);

    /* Check for the end of message signal. */
        if (tag != VDI_TAG_MESSAGE_END)
            return(VDI_ERROR_MESSAGE_BAD);

    return(0);
}
```

Figure 4.6: The procedure VdiMessageReceive() for receiving messages via a queue.

Packet Length (words)	Roundtrip Transit Time (microseconds)
1	850
1000	13100

Table 4.1: Time spent by the system in a full loopback communications test. (Results are average times for 5000 iterations).

Packet Length (words)	Transit Time (microseconds)
1	780
1000	4860

Table 4.2: Time spent by the system in a local loopback communications test. (Results are average times for 5000 iterations).

In a second test, the host writes a packet into a queue and reads it back from the *same* queue – it does not involve the DSP at all. Table 4.2 shows the results for this local loopback test.

If we assume that *VdiQueuePut()* and *VdiQueueGet()* require approximately the same amount of time on either the host or a DSP, we can derive a formula for the communication time for an arbitrary size packet from the results of these two tests. The difference between the full and local loopback times give the time the DSP spends in queue operations. The local loopback times give the time the host spends in queue operations. Table 4.3 lists the results of such an analysis.

The time for one complete communication from the host to DSP or vice versa is the sum of the two times shown in Table 4.3. Note that the host contributes most heavily to the overhead, while the DSP contributes most heavily to the per byte time.

Agent	Time
Host	$388\mu s + 2.04\frac{\mu s}{word}$
DSP	$31\mu s + 4.09\frac{\mu s}{word}$
Total	$419\mu s + 6.13\frac{\mu s}{word}$

Table 4.3: Approximate time spent by the host and DSP in either VdiQueueGet() or VdiQueuePut().

Chapter 5

Remote Input-Output Console for DSP Programs

When we debug programs in mature computer environments, source-level, symbolic debuggers usually offer the greatest insight into problems with the software. When such a debugger is not available, we often resort to a more "traditional" techniques — adding *printf()* and *scanf()* statements (in the case of C) into the code to display or alter the values of variables and generate messages indicating progress.

The VxWorks operating system includes interactive source-level symbolic debugging support through a remote version of the tool *dbx*. This is useful for debugging those parts of an application that run on the host. Unfortunately, we do not have even a simple object-level debugger for the code running on the DSPs. AT&T supplies a DSP32C simulator, but having no knowledge of our particular hardware platform, it is mostly useful for checking the computational parts of an application in isolation.[1]

Furthermore, because the DSPs do not have a console for input and output, the *printf()/scanf()* debugging technique would be impossible as well ... unless we provide a virtual console through the system software. This is precisely what we have done.

[1]AT&T does offer an in-circuit emulator that allows one to debug the target system under the control of a personal computer; however, we do not have one at our disposal.

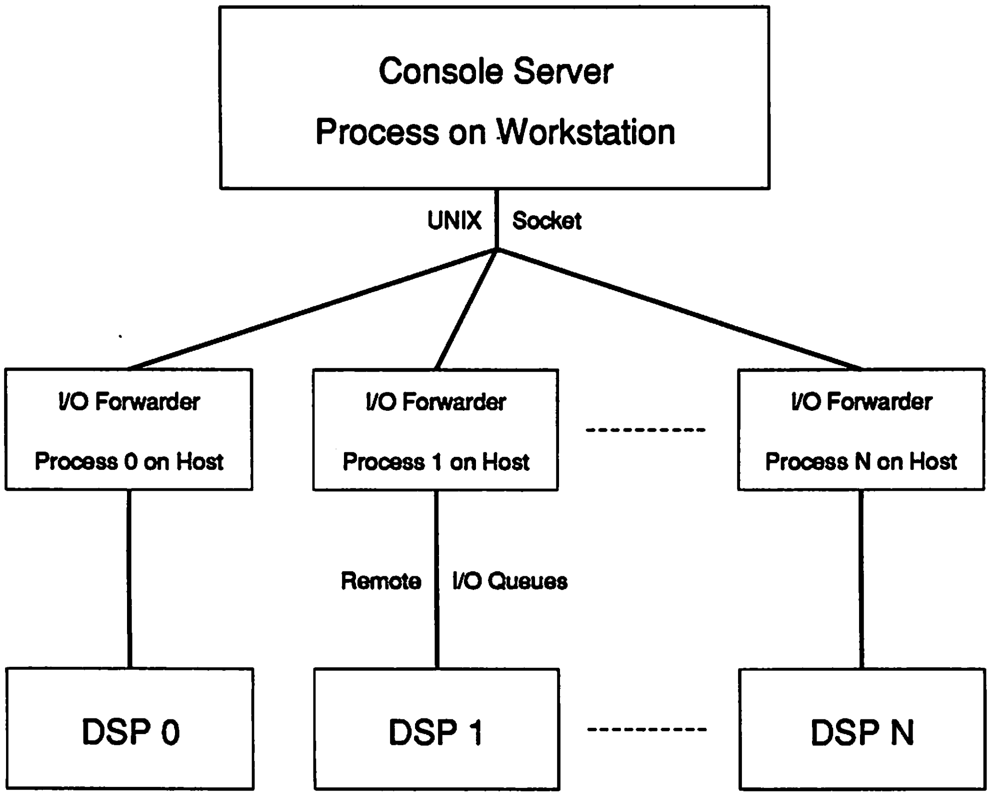

Figure 5.1: The components in the remote DSP input-output console.

5.1 Overview

Figure 5.1 depicts the components in the system software's remote input-output console.

Users interact with a console server process that runs on the workstation, usually in a separate window. The *standard input* and *standard output* of this process are effectively tied to each DSP. The system software provides special versions of *printf()* and *scanf()*: *VdiPrintf()* and *VdiScanf()*. Presently, they provide the only means for a DSP program to access these input and output handles.

All of the DSPs share one console server process. The server names the DSP making each request for input or output by displaying an identifying number in the first several columns of the display. Thus a test program,

```
.
.
.
2 -> Remote I/O Console Test ...
2 <- Enter an integer: 1234
2 -> You entered 1234.
.
.
.
```

Figure 5.2: A sample interaction with a DSP program through the remote input-output console.

running on DSP #2, might yield the interaction of Figure 5.2. Rightward-pointing arrows mark output the DSP program generates with *VdiPrintf()*, and leftward-pointing arrows mark *VdiScanf()*'s prompts for input.

Two extra queues reside in the DPRAM of each DSP in addition to the normal communication queues. One holds requests from the DSP (for both *VdiPrintf()* and *VdiScanf()*); the other holds responses from the console server (for *VdiScanf()* only).

Forwarding tasks run on the host, one per DSP. Each task constantly watches its request queue for request packets. When it finds one, it creates a connection to the console server through a Unix socket and passes the request to the server. It then listens to the socket for an acknowledgement. In the case of *VdiScanf()*, this acknowledgement carries with it the data the user entered at the console; the forwarder places this data into the response queue where the DSP can access it.

5.2 Request and Response Formats

Figure 5.3 shows the contents of the request packet which *VdiPrintf()* and *VdiScanf()* generate. The *DSP id* field, which the forwarding tasks fill in, names the DSP originating the request. The *request type* field indicates whether the request is for input or output. The *data type* field identifies the type of data involved in the transaction: for *VdiPrintf()*, the data is supplied

DSP id
Request type
Data type
Newline flag
Data value
Text

Figure 5.3: The format of the remote input-output request packet.

Data type
Data value

Figure 5.4: The format of the remote input response packet.

in the data field; for *VdiScanf()*, the data field is ignored. The *newline flag* field indicates whether the server should print a newline character after processing the request (for *VdiPrintf()* only). The *text* field contains the string which the console server displays when processing the request.

In the case of *VdiScanf()*, the console server sends back a reply. Figure 5.4 shows the format of the response packet which the forwarding tasks place into their response queues. The *data type* field identifies the type of the *data value* the user supplied.

5.3 Supported Data Types

The remote input-output system supports the C data types listed below:

- *char*.
- *int*. Signed or unsigned, hexadecimal or decimal, and short, normal, or long.
- *float* and *double*.
- *string*. Null-terminated, but with a fixed upper limit on length.

The hex data type variants merely inform the console server to display or read values in hexadecimal for user convenience.

5.4 Data Conversions

Because of differences between our workstation environment, where the console server runs, and the DSP environment, where the applications using the remote services run, the system software must take some extra steps to provide an input-output service which hides these dissimilarities.

5.4.1 Floating Point Numbers

The workstation uses the IEEE format for *float* and *double* data, while the DSPs do not. Moreover, the DSPs treat the two types identically: *double* does not imply any greater precision than *float* and is permitted in C programs solely for compatibility.

The system software therefore explicitly converts DSP 32-bit *float* and *double* variables to and from IEEE format. Since IEEE 32-bit *floats* do not have enough dynamic range to properly cover the range of DSP numbers, the console server utilizes IEEE 64-bit *doubles.*

5.4.2 Integers

The workstation environment defines *int* as a 32-bit quantity, but the DSPs define them as 24-bit quantities. The types *short int* and *long int*, however, are 16- and 32-bits wide in both environments, respectively.

The system software simply extends DSP *ints* into *long ints.*

5.4.3 Character Strings

Although both the workstation and DSP architectures employ 32-bit wide data paths, they place 16-bit and 8-bit data on different parts of their 32-bit data busses. The workstation is what is known as a *high-endian* machine while the DSPs are *low-endian* machines. The result is that characters in strings sent from one to the other are scrambled: within every block of four characters, the characters are in reverse order.

To correct for this effect, the console server pads strings to lengths that are multiples of four characters and reorders the characters within each group of four.

5.5 Details of Operation

This section covers the actual procedures involved in the remote input-output system in more detail. Note that for both the host and DSPs, *pWriteQueue* in the pseudocode refers to the outbound queue and *pReadQueue* to the inbound queue. Therefore, *pWriteQueue* refers to the request queue in *VdiPrintf()* and *VdiScanf()* (which run on the DSPs), but to the response queue in *VdiIOForwarder()* (which runs on the host). The opposite is true of *pReadQueue*.

5.5.1 Request Generators

VdiPrintf()

Figure 5.5 shows the code for the procedure *VdiPrintf()*. The console server first displays the string to which *pText* points; then it prints the value of the variable of type *varType* to which *pVar* points; and if *newline* is *true*, it advances the cursor to the next line.

VdiScanf()

Figure 5.6 illustrates the procedure *VdiScanf()*. The console server first displays the string to which *pText* points; then it reads a value of type *varType* and returns it in the variable to which *pVar* points.

5.5.2 Request Forwarder

Figure 5.7 shows a simplified version of the procedure *VdiIOForwarder()*. A separate process for each DSP executes this function. The parameter *dspId* tells it which DSP to service.

```
void VdiPrintf(pText, pVar, varType, newline)
    char *pText;
    void *pVar;
    int varType;
    int newline;
{
    VdiIOParameters parameters;
    VdiDSP *pDSP;

    /* Locate the information for this DSP. */
        pDSP = ...;

    /* Format the request. */
        parameters.request = VDI_IO_REQUEST_PRINTF;
        parameters.dataType = varType;
        parameters.newline = newline;

    /* Copy the text. */
        strncpy(parameters.text, pText, VDI_IO_TEXT_MAX);

    /* Copy the variable, making any necessary conversions. */
        VdiDataCopy(pVar, parameters.data, varType);

    /* Send the request. */
        VdiQueuePut(pDSP, pDSP->pWriteQueue, VDI_TAG_IO_REQUEST,
                    VDI_WORDS(sizeof(VdiIOParameters)),
                    &parameters, 0);
}
```

Figure 5.5: The procedure VdiPrintf() for remote printing.

```
void VdiScanf(pText, pVar, varType)
    char *pText;
    void *pVar;
    int varType;
{
    VdiIOParameters parameters;
    VdiIOResultScanf result;
    int tag, length;
    VdiDSP *pDSP;

    /* Locate the information for this DSP. */
        pDSP = ...;

    /* Format the request. */
        parameters.request = VDI_IO_REQUEST_SCANF;
        parameters.dataType = varType;

    /* Copy the text. */
        strncpy(parameters.text, pText, VDI_IO_TEXT_MAX);

    /* Send the request. */
        VdiQueuePut(pDSP, pDSP->pWriteQueue, VDI_TAG_IO_REQUEST,
                    VDI_WORDS(sizeof(VdiIOParameters)),
                    &parameters, 0);

    /* Get the reply. */
        VdiQueueGet(pDSP, pDSP->pReadQueue, &tag, &length,
                    &result, VDI_WORDS(sizeof(VdiIOResultScanf)),
                    0);

    /* Copy the user's response into the variable. */
        VdiDataCopy(result.data, pVar, varType);
}
```

Figure 5.6: The procedure VdiScanf() for obtaining remote input.

```
void VdiIOForwarder(dspId)
    int dspId;
{
    VdiIOParameters parameters;
    VdiIOResultScanf result;
    int tag, length;
    VdiDSP *pDSP;

    /* Locate the DSP information in the global table. */
        pDSP = ...;

    for ( ; ; ) {
        /* Get the next request. */
            VdiQueueGet(pDSP, pDSP->pReadQueue, &tag, &length,
                        &parameters,
                        VDI_WORDS(sizeof(VdiIOParameters)), 0);

        /* Fill in the DSP identification number for the console
           server's use.
        */
            parameters.dspId = dspId;

        /* Create an Internet stream socket, forward the request
           to the server.
        */
            ...

        /* Wait for some form of acknowledgement that the request
           was serviced.
        */
            if (parameters.request == VDI_IO_REQUEST_PRINTF) {
                /* Get the acknowledgement from the server. */
                    ...
            } else {
                /* Get the response from the server. */
                    ...

                /* Forward the response to the DSP. */
                    VdiQueuePut(pDSP, pWriteQueue,
                                VDI_TAG_IO_RESULT_SCANF,
                                VDI_WORDS(sizeof(VdiIOResultScanf)),
                                &result, 0);
            }
    }
}
```

Figure 5.7: The procedure VdiIOForwarder() that forwards requests for remote input or output.

5.5.3 Request Server

Figure 5.8 shows a simplified version of the procedure *VdiIOConsole()*.

```
void VdiIOConsole()
{
    VdiIOParameters parameters;
    VdiIOResultScanf result;
    VdiDSP *pDSP;

    /* Setup the socket. */
        ...

    for ( ; ; ) {
        /* Read the request from the socket. */
            ... &parameters ...

        /* Locate the DSP information in the global table. */
            pDSP = ...;

        /* Reorder the characters in the request's text field. */
            ...

        /* Identify and process the request. */
            if (parameters.request == VDI_IO_REQUEST_PRINTF) {
                /* Display the text. */
                    printf("%d -> %s", parameters.dspId, parameters.text);

                /* Convert and display the data. */
                    switch (parameters.dataType) {
                        ... parameters.data ...
                    }

                /* Display a newline if requested. */
                    if (parameters.newline)
                        printf("\n");

                /* Acknowledge the completion of the request */
                    ...
            } else {
                /* Display the prompt. */
                    printf("%d <- %s", parameters.dspId, parameters.text);

                /* Read and convert the data value. */
                    switch (parameters.dataType) {
                        ... result.data ...
                    }

                /* Return the value to the forwarder. */
                    ... &result ...
            }
    }
}
```

Figure 5.8: The procedure VdiIOConsole() that services remote input-output requests.

Chapter 6

DSP Fatal Error Reporting Mechanism

DSP applications and the system software may detect various forms of errors during execution, some of which require immediate attention. For instance, a DSP application may detect a robot malfunction; it must then shutdown power to the servos and notify the host. Another possibility is that the system software may be unable to service a request, such as one initiated by *VdiPrintf()*, because some communication link failed or because a DSP application accidentally destroyed a system data structure.

In either scenario, the DSP must directly notify the host that a problem exists. Using a communication queue is not appropriate for two reasons. First, other packets may already be waiting in the queue; the host will not see the error notification until it processes these packets waiting ahead of it. Second, the queue itself may be in an inconsistent state or corrupted, in which case it cannot be used to transmit any information at all.

For these reasons, the system software provides one alternate communication channel for each DSP solely for the purpose of reporting severe errors.

6.1 Overview

Each DPRAM contains a location reserved for fatal error codes. The location is initialized to zero. A DSP may report an error by writing a (non-zero) error code into the special location in its DPRAM.

```
void VdiErrorReport(errorCode, loop)
    int errorCode;
    int loop;
{
    VdiDSP *pDSP;

    /* Locate the information for this DSP. */
        pDSP = ...;

    /* Write the error code into the DPRAM. */
        *(pDSP->pErrorCode) = errorCode;

    /* Loop forever if requested. */
        if (loop)
            for ( ; ; )
                ;
}
```

Figure 6.1: The procedure VdiErrorReport() for reporting fatal errors to the host.

A special task, *VdiErrorServer()*, runs on the host. It periodically checks the error word in every DSP's DPRAM. When it finds a non-zero value, it displays it in the window through which the user logged into the host, and then resets the word back to zero.

6.2 Error Reporter

Figure 6.1 shows the procedure *VdiErrorReport()* which any DSP function may call to report a severe error. The parameter *errorCode* identifies the error, and the parameter *loop* indicates whether the routine should enter an infinite loop to stop the DSP.

```
void VdiErrorServer()
{
    VdiDSP *pDSP;
    int errorCode;
    int i;

    for ( ; ; ) {
        for ( i = each DSP ) {
            /* Locate the information for the DSP. */
                pDSP = ... i ...;

            /* Get the error code. */
                errorCode = *(pDSP->pErrorCode);

            /* Report it if non-zero. */
                if (errorCode) {
                    printf("Detected error %d from DSP %d.\n",
                           errorCode, i);
                    *(pDSP->pErrorCode) = 0;
                }
        }
    }
}
```

Figure 6.2: The procedure VdiErrorServer() that detects and reports fatal errors from DSPs.

6.3 Error Server

Figure 6.2 shows the procedure *VdiErrorServer()* that the host error server task executes.

Chapter 7

Starting the DSPs

In a production system, the DSP side of the system software will reside in the EPROM on each board. During development of the system software, however, this would pose a major inconvenience – changing the software would require (slowly) burning the new code into an EPROM.

We have designed the system software to allow the host to load the DSP system code into RAM on each board. This ability greatly speeds up the modify-compile-test cycle for system software components.

7.1 Special Memory Address Decoders

We use a special version of the memory decoder PAL chip on each board that allows us to place the system startup code into RAM. The DSPs, when reset, start executing code from location zero. These special PALs are programmed to map the beginning of the DPRAMs to address zero in the memory spaces of the DSPs. By contrast, the production PALs map the beginning of the EPROMs to address zero.

7.2 Bootstrap Code

The bootstrap code is the first piece of code which the DSPs actually execute. Its main purpose it to bring the DSPs into a controlled state and to load the complete monitor program into the DSPs' local static RAMs.

The host does not try to put the monitor program directly into DPRAM for two reasons. Firstly, the monitor is too large to fit. Secondly, even if the monitor were trimmed down to fit within the DPRAM, it would occupy valuable space that communication queues could better utilize.

The bootstrap program occupies only the first half of the DPRAM. The two normal communication queues occupy the second half. After the booststrap program terminates, the host uses the first half of the DPRAM to hold the remote input-output queues.

7.3 DSP Initialization from the Host

For each DSP, the code on the host takes the following actions to start each DSP during system initialization:

1. Forces the DSP into an idle reset state by writing to the control register on the board.
2. Writes the bootstrap code into the DPRAM starting at location zero.
3. Formats the second half of the DPRAM into empty communication queues.
4. Releases the DSP from its reset state, allowing it to start executing the bootstrap code from location zero.
5. Sends the monitor program to the bootstrap loader via a queue.
6. Formats the first half of the DPRAM into empty remote input-output queues, overwriting the bootstrap code.
7. Tests that the monitor is "alive."

Chapter 8

DSP Monitor Program

The monitor program serves as a simple operating system for the DSPs. During initialization, the boostrap program loads it into the local static RAM on each board and transfers control to it.

8.1 Organization

The monitor is essentially a server that responds to commands from the host. The host may send commands to a DSP's monitor by placing an appropriately formatted packet into its communication queue — the tag specifies the command and the contents its arguments. The host may send commands faster than the DSPs can execute them; the DSPs will simply process them in order.

8.2 Functions

8.2.1 Report Version Number

The monitor can report its version number. The initialization code on the host uses this to test that the monitor is actually running and to verify that it is compatible with the support code on the host. If the monitor is burned into EPROM, and the host code is subsequently modified, the version number checking can safeguard against situations in which changes in monitor protocol lead to incorrect interpretation of commands.

8.2.2 Perform Loopback Test

The monitor supports a loopback test in which it simply returns all packets sent to it, until it receives an *end of test* packet. This is useful for checking that the communications routines do not corrupt packets and for benchmarking their speed.

8.2.3 Write Block of Memory

Although the host can access the DPRAM on any board, it cannot access any other memory in a DSP's address space. Instead, the monitor can act on behalf of the host and write host-supplied data into a block of DSP memory. This function is useful both for diagnostic purposes and for loading application programs into a DSP's static RAM.

8.2.4 Read Block of Memory

The monitor can also send the contents of a block of DSP memory to the host for diagnostic purposes.

8.2.5 Call Application Function

Finally, the monitor can jump to a subroutine at a specified address. Typically, the host uses this function to execute an application previously written into the static RAM.

Part II

Details of System Operation

The remaining sections of this document cover the details necessary for developing and running applications and for maintaining the system software.

Chapter 9

Using the System

This chapter describes how to interact with a system built on the common software/hardware base and run applications on it.

9.1 Establishing the Environment

9.1.1 Environment Variables

Set the environment variable *VDI* on the workstation to the name of the directory containing the system software. For example:

```
setenv VDI ~arya/vdi
```

Also, add the *bin* and *util* subdirectories within that directory to your path. You will probably want to modify your *.cshrc* or *.login* file to make the change permanent.

9.1.2 Manual Pages

The simplest way to access the manual pages of Appendix B online is to define an alias on the workstation that calls the Unix *man* command with a pointer to the directory containing the man pages. For instance, after issuing the Unix command:

```
alias vdiman man -M $VDI/man
```

you can access the manual page for *VdiQueuePut()* with the command:

```
vdiman VdiQueuePut
```

Some topics in the manual pages are pertinent to system use, while others concern application development. Refer to Appendix B for more details.

9.1.3 System Parameter File

Very little system dependent information is compiled into the system software. Most of it is contained in a special human-readable ASCII parameter file. The sample parameter file *system.parms* in the system software root directory looks like this:

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

/* Host Information */

"bin/vdiBootCode.d3bin"    /* Bootstrap code file */
"bin/vdiMonitorCode.d3bin" /* Monitor code file */

/***********************************************************/

/* Remote I/O Console Server Information */

"bryce"                   /* Console server machine */
1100                      /* Port number (must exceed 1000) */

/***********************************************************/

/* Task Priorities: from 51 (highest) to 253 (lowest). */

125                       /* I/O server tasks */
100                       /* Fatal error server task */
2.0                       /*
                            Fatal error server sleep time (sec).
                            The error server pauses for this
                            duration of time between checks for
                            errors.
                          */
```

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

/* Application information. */

0x7000                  /* Application load address */

/**************************************************************/

/*
  DSP Information:

  Each "DSP { ... }" structure defines one DSP in the system.
  The DSPs are identified by their positions in this file:
  the first one is DSP #0, the second DSP #1, and so on.  No
  more than 16 DSPs may be defined.

  All queues must lie within the shared dual port memory.
  Since the bootstrap loader makes use of the in and out
  queues, they must not lie within the region occupied by it
  (which starts at the specified boot address and is as long
  as the bootstrap loader itself).  The read and write
  queues, however, may lie in that region.

  The request mask is OR'ed into the control register to
  assert the arbiter request line.  The release mask is
  AND'ed into that register to disassert the request line.
  Similarly, the reset mask is OR'ed and the run mask is
  AND'ed.  The grant mask is AND'ed with the status register
  to test the arbiter grant line.
*/

DSP {                   /* DSP #0 */
    /* Host view */
    0x01980000          /* Control Register */
    0x00000000          /* Initial Control Register Value */
    0x01980400          /* Status Register */
    0x01903000          /* In Queue */
    0x03ef              /*   Length */
    0x01902000          /* Out Queue */
```

```
        0x03ef                  /*   Length */
        0x01901000              /* Read Queue */
        0x03ef                  /*   Length */
        0x01900000              /* Write Queue */
        0x03ef                  /*   Length */
        0x00002000              /* Request Mask */
        0xffffdfff              /* Release Mask */
        0x00002000              /* Grant Mask */
        0x00000100              /* Reset Mask */
        0xfffffeff              /* Run Mask */
        0x01900000              /* Boot Address */

        /* DSP view */
        0xffd400                /* Control Register */
        0x0000                  /* Initial Control Register Value */
        0xffd440                /* Status Register */
        0x002000                /* In Queue */
        0x003000                /* Out Queue */
        0x000000                /* Read Queue */
        0x001000                /* Write Queue */
        0x0020                  /* Request Mask */
        0xffdf                  /* Release Mask */
        0x0020                  /* Grant Mask */
    }
```

The system software reads this file before initializing the hardware. The comments near each entry define the type of information the initialization code expects.

Lexical Format

Lexically, the parameter file looks very similar to C source. White space consisting of spaces, tabs, and newlines may appear anywhere between tokens. Comments begin with /* and end with */. Decimal integers consist of a simple sequence of digits. Hexadecimal integers are similar but begin with *0x*. Floating point numbers follow the usual convention (and may contain exponential notation). Strings are enclosed in double quotation marks.

Syntax

DSP System Code: The first section of the parameter file specifies the names of the files containing the binary executable code for the bootstrap and monitor programs.

Remote I/O Console Server: This section names the machine on which the remote input-output console server is running, and the Unix socket port number on which the server is listening for requests. The machine name must appear exactly as it does in the VxWorks host table. The port number must exceed 1000 and be identical to that you specify when running the console server (see Section 9.3.1).

Host Background Tasks: This section sets the priorities of the host background tasks — the remote input-output forwarders and the error server. VxWorks allocates *less* cpu time to tasks with *high* priority numbers and *more* time to tasks with *low* numbers. The priorities must lie between 51 and 253, inclusive. In addition, this section specifies how long, in seconds, the error server should wait between scans for fatal error reports from DSPs. The error server does not consume cpu time during this delay period. The shorter the period, the quicker the error server will detect and report fatal errors, and the more cpu time it will consume.

Application: This section specifies the address at which the host loads application programs in the static RAM local to a DSP. It must match the number specified in the memory map file passed to the linker (see Section 10.6.4 for details).

DSPs: The final section of the parameter file describes each DSP in the system. It consists of a series of structures, one per DSP, that begin with *DSP* and contain information between a pair of curly braces. The order of the structures defines the DSP identification numbers: the first one is number 0, the next 1, and so on .... The system supports no more than 16 total.

Within each structure, the first set of entries provide information about the DSP from the host's perspective, and the second set from the DSP's own perspective. The comments in the sample file describe each individual entry more thoroughly.

9.2 File Names

Many types of files exist in this system, especially since object code exists for both the host and DSPs. The list below explains the relationship between the role of a file and the extension to its basic name:

.c:	C source code.
.h:	C header.
.s:	DSP assembler source code.
.o:	Host object code.
.d3o:	DSP object code.
.d3bin:	DSP linked executable code.
.d3img:	DSP executable code in a special ASCII format.
.map:	Module memory layout information for input to the DSP linker; or, library/executable file layout information from the host or DSP librarian/linker.
.a:	DSP library.
.ln:	Host or DSP lint library.

All the *.d3xxx* extensions are nonstandard names for files of those types. They serve to differentiate the DSP files from the host files of the same type. This is necessary because some source files may be compiled for both the host and DSPs.

9.3 Running the System

The following sections outline the steps you must follow in order to run the system.

9.3.1 Running the Remote I/O Console Server

The input-output forwarding tasks expect the remote input-output console server to be running before they start. You will probably want to run it in a separate shell window. To bring it up, issue the command:

```
vdiIOConsole <port number>
```

The *port number* argument identifies the port on which the server will listen for requests via a socket. It must be identical to the number specified in the system parameter file, as discussed in Section 9.1.3.

For instance, if you issue the command:

```
vdiIOConsole 1100
```

to use port 1100, you should see the following if all is well:

```
===========================================================
             VxWorks <--> DSP32C Interface Software
                  Remote DSP I/O Console Server
                          Version 1.00
               University of California, Berkeley
===========================================================

Waiting for requests on port #1100 ...
```

To terminate the server for any reason, press *control-c*.

If the console server reports the error "Can't bind socket," use a different port number. This problem sometimes arises when the console server or the forwarding tasks (on the host) terminate abnormally. The port in use at the time of termination can be reused after a short period of time (on the order of fifteen minutes).

9.3.2 Logging into the Host

To connect with the host machine, log into it remotely through another shell window on the workstation with the command:

```
rlogin <host name>
```

where *host name* is the name of the machine. You should then see the standard VxWorks prompt:

```
->
```

If you see an error message stating that the machine refuses your request for a connection, someone else is logged into that host. VxWorks is not a multi-user system, so only one person may use it at a time.

After logging in, execute your VxWorks startup shell script. For example:

```
< vwstartup
```

This script must define the machine on which the console server runs in the VxWorks host table. Consult [6] for more information.

9.3.3 Loading the Host Code

Once you have logged into the host, you must load the host code. To load the system library, issue the command:

```
ld < lib/libvdi.o
```

To load the support code, use the command:

```
ld < bin/vdiHostCode.o
```

These examples assume that the current directory is the one containing the system software; if not, you must prefix the file names with the correct path.

9.3.4 Bringing up the System

With the host code loaded, you are ready to initialize the system. At the VxWorks prompt, the command:

```
VdiInit("<parameter file>")
```

accomplishes this, using the *parameter file* for hardware information. Be sure to enclose the file name in quotation marks. For example:

```
VdiInit("system.parms")
```

The function *VdiInit()* reports its progress in detail. When everything is in order, you should see a display like this:

```
===========================================================
          VxWorks <--> DSP32C Interface Software
                       Version 1.00
            University of California, Berkeley
===========================================================

Initializing System ...

Reading system configuration from 'system.parms' ...
Loading bootstrap code from 'bin/vdiBootCode.d3bin' ...
    Reading: .text .data .parms .mon[skipped] .bss[skipped]
Loading monitor code from 'bin/vdiMonitorCode.d3bin' ...
    Reading: .text .data .parms .bss[skipped]

Initializing DSP #0:
    Initializing control register ...
    Halting DSP ...
    Initializing host task sync semaphore ...
    Initializing communications queues ...
    Writing bootstrap code into DP RAM ...
    Starting DSP ...
    Sending monitor code to bootstrap program ...
    Testing monitor ...
        Monitor version number: 1.00
    Initializing remote I/O queues ...
    Spawning remote DSP I/O forwarder ...

Initializing DSP #1:
    .
    .
    .

Spawning DSP fatal error server ...
```

If *VdiInit()* stops somewhere in the middle of this process, verify that the boards are seated properly in the card cage and that the entries in the parameter file are correct.

9.3.5 Interacting with the Monitor

The host support code includes a set of utility routines for sending commands to the monitor program running on a DSP; they send properly formatted command packets to the monitor and thus serve as a crude front-end to the DSPs.

VdiBlockPut()

The command:

```
VdiBlockPut(<dspId>, <address>, <length>, <pData>)
```

writes to a block of memory in the address space of DSP #*dspId*. The parameters *address* and *length* specify the location of the block and its size in words, and *pData* points to the buffer containing the source data.

For example, to write *0x11* into *0x10* words at address *0xa000* of the memory belonging to DSP #0, issue the commands:

```
mybuffer = malloc(0x10 * 4)
bfill(mybuffer, 0x10*4, 0x11)
VdiBlockPut(0, 0xa000, 0x10, mybuffer)
free(mybuffer)
```

The host implements this command by sending a packet with tag *VDI_TAG_MON_BLOCK_PUT* to the monitor running on the specified DSP followed by a message containing the contents of the buffer.

VdiBlockGet()

The command:

```
VdiBlockGet(<dspId>, <address>, <length>, <pData>)
```

reads from a block of memory in the address space of DSP #*dspId*. The parameters *address* and *length* specify the location of the block and its size in words, and *pData* points to the buffer for holding the data read.

For example, to display the contents of *0x10* words at address *0xa000* of the memory belonging to DSP #0, use the commands:

```
mybuffer = malloc(0x10 * 4)
VdiBlockGet(0, 0xa000, 0x10, mybuffer)
d mybuffer
free(mybuffer)
```

The host implements this command by sending a packet with tag *VDI_TAG_MON_BLOCK_GET* to the monitor running on the specified DSP and writing the contents of the monitor's reply message into the buffer.

VdiAppLoad()

The command:

```
VdiAppLoad(<dspId>, "<file>")
```

loads a compiled application program into the memory local to DSP #*dspId* for execution later. The parameter *file* names the file containing the compiled, linked application program; such files normally have the extension *.d3bin.*

To load the application *test.d3bin* into DSP #0, for example, use the command:

```
VdiAppLoad(0, "test.d3bin")
```

The host implements this command by calling *VdiBlockPut()* for each section of the binary file, starting at the address *application load address* in the parameter file.

VdiAppRun()

The command:

```
VdiAppRun(<dspId>)
```

executes an application program previously loaded into the memory local to DSP #*dspId* by the command *VdiAppLoad().*

To run an application loaded into DSP #0, for example, use the command:

```
VdiAppRun(0)
```

You may repeat this multiple times without reloading the application between iterations.

The host implements this command by sending a packet with tag *VDI_TAG_MON_CALL* to the monitor on the specified DSP with the *application load address* from the parameter file as the target address. The host does *not* wait for the DSP; the DSP will begin execution once it has processed other commands waiting in the queue.

VdiAppLoadAndRun()

The command:

```
VdiAppLoadAndRun(<dspId>, "<file>")
```

loads *and* executes an application program on DSP *#dspId*. It is merely shorthand for the combination of *VdiAppLoad()* and *VdiAppRun()*.

To load and run the application in *test.d3bin* on DSP #0, for example, issue the command:

```
VdiAppLoadAndRun(0, "test.d3bin")
```

The host implements this command simply by calling *VdiAppLoad()* followed by *VdiAppRun()*.

VdiLoopback()

The command:

```
VdiLoopback(<dspId>, <mode>, <iterations>)
```

runs a communication loopback test of DSP *#dspId* and reports the round trip packet transit time. It is useful for both testing that a monitor is "alive" on a DSP and for benchmarking the communications routines. The parameter *mode* identifies one of several test variations and *iterations* specifies the number of times to perform the test. The manual page for this command describes each mode in detail.

To test how long it takes to transmit short, one word packets to DSP #0, for instance, issue the command:

```
VdiLoopback(0, 1, 1000)
```

The host will time 1000 packets and display the average.

The host implements this command by sending a packet with tag *VDI_TAG_MON_LOOPBACK_START* to the monitor on the specified DSP, followed by a series of test packets. The monitor returns these packets to the host unaltered. The host sends a packet with tag *VDI_TAG_MON_LOOPBACK_STOP* to the monitor to terminate the test.

To time packets, the host calls the VxWorks routine *tickGet()* both before and after the entire test. It then divides the elapsed time by the number of iterations and the ticks-per-second conversion factor that *sysClkRateGet()* returns.

9.3.6 Accessing VME Memory

The two routines described below provide a means of accessing *any* memory in the VME address space. They are necessary because the VxWorks shell only permits access to memory *on the host circuit board* through the use of pointers and the *d()* command.

VdiMemDump()

The command:

```
VdiMemDump(<address>, <length>)
```

displays memory words in 32-bit hexadecimal format. The parameters *address* and *length* specify the starting address and number of words to display.

To dump *0x40* words at address *0x1900000*, for example, issue the command:

```
VdiMemDump(0x1900000, 0x40)
```

VdiMemSet()

The command:

```
VdiMemSet(<address>, <value>)
```

writes to a 32-bit memory word. The parameters *address* and *value* specify the word address and value to write.

To write *0x55aa55aa* into the word at address *0x1900000*, for instance, use the command:

```
VdiMemSet(0x1900000, 0x55aa55aa)
```

9.3.7 Viewing Queues

The command:

```
VdiQueueDisplay(<address>)
```

displays the contents of the queue at *address* in DPRAM. It is useful mostly for determining at what point a malfunctioning system stopped.

To view a queue at address *0x1902000*, for instance, use the command:

```
VdiQueueDisplay(0x1902000)
```

9.3.8 Shutting down the System

To stop a running system, issue the command:

```
VdiShutdown()
```

at the VxWorks prompt. The host will then halt every DSP, delete every input-output forwarder task, and delete the error server task.

Like *VdiInit()*, *VdiShutdown()* reports its progress in detail. When everything is in order, you should see a display like this:

```
Bringing down the system ...
    Halting DSPs ...
    Killing remote I/O forwarders ...
    Killing fatal error server ...
```

Before reinitializing a *running* system with *VdiInit()*, you must execute *VdiShutdown()* to delete these background tasks; otherwise the tasks which *VdiInit()* creates the second time will compete with the original ones.

9.3.9 Logging out from the Host

To leave the host, issue the command:

```
logout
```

Do *not* use the command *exit*, as this will prevent you from logging into the host again until it is physically reset.

The system continues to run even after you log out. You need only be logged in to control it.

9.4 Sample Session

Some aspects of the system's operation will be clearer from an extended example. Figure 9.1 is a script of a session with the host. Figure 9.2 is a script of the corresponding interaction with the remote input-output console server. Section 10.8.3 explains the sample application program illustrated here.

```
navajo:arya (~/vdi) 53 > rlogin vw2

-> <arya
hostAdd "navajo", "128.32.139.73"
value = 0 = 0x0
iam "arya"
value = 0 = 0x0
cd "zabriskie:~arya/vdi"
value = 0 = 0x0
-> ld < lib/libvdi.o
value = 0 = 0x0
-> ld < bin/vdiHostCode.o
value = 0 = 0x0
-> VdiInit("system.parms")

==========================================================
          VxWorks <--> DSP32C Interface Software
                       Version 1.00
            University of California, Berkeley
==========================================================

Initializing System ...

Reading system configuration from 'system.parms' ...
Loading bootstrap code from 'bin/vdiBootCode.d3bin' ...
    Reading: .text .data .parms .mon[skipped] .bss[skipped]
Loading monitor code from 'bin/vdiMonitorCode.d3bin' ...
    Reading: .text .data .parms .bss[skipped]

Initializing DSP #0:
    Initializing control register ...
    Halting DSP ...
    Initializing host task sync semaphore ...
    Initializing communications queues ...
    Writing bootstrap code into DP RAM ...
    Starting DSP ...
    Sending monitor code to bootstrap program ...
    Testing monitor ...
        Monitor version number: 1.00
    Initializing remote I/O queues ...
    Spawning remote DSP I/O forwarder ...

Spawning DSP fatal error server ...
value = 0 = 0x0
-> VdiAppLoadAndRun(0,"app/test.d3bin")
    Reading: .text .data .parms .bss[skipped]
value = 0 = 0x0
-> VdiShutdown
Bringing down the system ...
    Halting DSPs ...
    Killing remote I/O forwarders ...
    Killing fatal error server ...
value = 0 = 0x0
-> logout
Connection closed.
navajo:arya (~/vdi) 54 >
```

Figure 9.1: Script of a sample interaction with the host.

```
navajo:arya (~/vdi) 62 > bin/vdiIOConsole 1100

==========================================================
         VxWorks <--> DSP32C Interface Software
              Remote DSP I/O Console Server
                      Version 1.00
            University of California, Berkeley
==========================================================

Waiting for requests on port #1100 ...

0 -> DSP Test Application
0 <- Enter an integer value for X: 10
0 -> X * X = 100
0 <- Enter a string: hello
0 -> You typed:
0 -> hello
```

Figure 9.2: Script of a sample interaction with the console server.

Chapter 10

Developing Applications

This chapter explains some of the more detailed aspects of the common hardware/software platform and describes how to develop applications that run atop it.

10.1 Data Types

Table 10.1 summarizes the representation of the basic C data types as implemented by both the host and DSP C compilers.

The sections that follow describe those system software data types built from these that *applications* may find useful.

Name	Size in bits	
	Host	DSP
char	8	8
short	16	16
long	32	32
int	32	24

Table 10.1: The internal host and DSP representations of C data types.

10.1.1 VdiWord

The most primitive integer quantity which the host may share with a DSP is the *VdiWord*. It is defined as an *unsigned long int* and is 32-bits wide. Wherever this report and the online manual pages refer to *words*, they are speaking of *VdiWords*.

10.1.2 VdiInt

The DSPs do not handle *VdiWords* very efficiently because their *integer* registers are only *24-bits* wide (the *32* in the name *DSP32C* refers to the *32-bit* wide *floating point* registers). The compiler must allocate two registers to hold *VdiWords* and produce many extra instructions to treat the pairs as single units. The host's microprocessor, however, does have 32-bit wide integer registers and thus suffers no performance loss when handling *VdiWords*.

When 24-bits of integer precision are sufficient, use the data type *VdiInt* instead of *VdiWord*. *VdiInt* is defined as an *unsigned int*. To the host, this is *32-bits* wide. To the DSPs, this is *24-bits* wide. Furthermore, whenever DSPs write a *VdiInt* to memory, they actually write a *32-bit* word whose upper 8 bits are zero; and whenever they read a *VdiInt* from memory, they read a 32-bit word and ignore the upper 8 bits.

Thus the host and DSPs can directly share *VdiInts* without any software adjustments. Furthermore, *VdiInts* are the quantities which each agent handles most efficiently in software.

10.1.3 VdiRegister

The type *VdiRegister* defines the control and status register objects. It is defined as *unsigned long int* on the host side since the host sees registers as *32-bit* wide objects. It is defined as *unsigned short int* on the DSP side since the DSP sees them as *16-bit* wide objects.

10.1.4 VdiDSP

The type *VdiDSP* is a large structure which holds physical hardware constants read from the system parameter file. Table 10.2 shows its contents. The second group of fields are only present in the definition of the structure

Field Name	Data Type	Description
pControlReg	VdiRegister *	Address of control register
controlRegS	VdiRegister	Shadow of control register
pStatusReg	VdiRegister *	Address of status register
pInQueue	VdiQueue *	Address of inbound comm. queue
pOutQueue	VdiQueue *	Address of outbound comm. queue
pReadQueue	VdiQueue *	Address of inbound I/O queue
pWriteQueue	VdiQueue *	Address of outbound I/O queue
requestMask	VdiRegister	OR bit-mask for arbiter request
releaseMask	VdiRegister	AND bit-mask for arbiter release
grantedMask	VdiRegister	AND bit-mask for arbiter grant
inQueueLength	unsigned int	Length of inQueue
outQueueLength	unsigned int	Length of outQueue
readQueueLength	unsigned int	Length of readQueue
writeQueueLength	unsigned int	Length of writeQueue
resetMask	VdiRegister	OR bit-mask for resetting DSP
runMask	VdiRegister	AND bit-mask for letting DSP run
bootAddress	char *	VME address of DSP boot location
hostSync	SEM_ID	VxWorks host task semaphore
IOForwarderTask	int	VxWorks id of forwarder task

Table 10.2: The data type VdiDSP. Note that the second group of fields are present only in the structure definition on the host side.

on the host side. Furthermore, those fields which occupy less than 32-bits on the DSP side are padded to 32-bit boundaries with dummy fields; this is necessary to allow the load-time fixup described in Section 10.6.3.

Not all of the fields in the table should be of interest to applications; the various *arbiter mask* fields, for instance, are necessary for accessing the arbiter, but applications should use the higher-level semaphore functions for synchronization instead. On the other hand, applications must access the various *p... Queue* fields to name queues when calling the queue functions. Section 10.2.3 explains the purpose of the *pControlRegS* field.

10.1.5 VdiSemaphore

System semaphore objects are of type *VdiSemaphore*. Internally, they are defined as *VdiInts*. Applications requiring extra synchronization beyond that provided by the queues may define their own semaphores in the DPRAM.

10.1.6 VdiQueue

System queue objects are of type *VdiQueue*. Figure 4.2 describes their contents. Applications need not know the actual internal structure of *VdiQueues*, however, because the system library provides functions for manipulating them: *VdiQueueInit()*, *VdiQueuePut()*, and *VdiQueueGet()*. Applications thus need only use this data type to define the types of arguments to these functions.

10.2 Global Variables

Applications need to access the global system variables that contain the hardware information shown in Table 10.2. The sections that follow describe these variables.

10.2.1 DSP Code

The DSP library defines a global variable called *pVdiDSP*. It is a pointer to a *VdiDSP* containing the hardware information for the DSP on which the code is executing, from that DSP's perspective.

An application wishing to get a packet from the inbound queue, for instance, may use this structure to name the queue as:

```
pVdiDSP->pInQueue
```

in the *VdiQueueGet()* argument list.

10.2.2 Host Code

The host must carry more hardware information than the DSP for two reasons. First, it requires knowledge of *every* DSP from its own perspective.

Field Name	Data Type	Description
total	int	Count of DSPs in system
h	VdiDSP []	Host view of DSP
d	VdiDSP []	DSP view of DSP

Table 10.3: The structure to which pVdiDSPs points on the host.

Second, it requires knowledge of each one from the DSP's perspective as well to implement the load-time fixup described in Section 10.6.3.

Thus the global variable *pVdiDSPs* on the host side is pointer to the structure shown in Table 10.3. The field *total* indicates how many DSPs are installed in the system. The array *h[]* defines the host's view of each DSP, while the array *d[]* defines each DSPs' view of itself.

Thus host code wishing to reference the inbound communication queue from DSP #0 can name it as:

```
pVdiDSPs->h[0].pInQueue
```

in a call to *VdiQueueGet()*.

10.2.3 Control Register Shadow Variables

The control registers on each board are *write-only*, so it is convenient to keep a software copy of the last value written to them. The system software uses the *controlRegS* field of the *VdiDSP* structure for exactly this purpose.

Therfore, an application wishing to modify a control register *must* use this same field in the appropriate (host or DSP) global hardware information variable; otherwise, the system software will not be aware of changes the application makes to the register, and vice versa.

10.3 Byte Swapping and Endian Differences

The first prototype of the robot controller pointed out a slight error in the design of the DSP to host interface section of the common hardware platform. Because of differences in the *endian-ness* of the DSP32C and 68020, the design called for reordering the data lines between the VME bus connector and DSP on each DSP board to compensate. The design exchanged lines for byte 0 with those for byte 3, and lines for byte 2 with those for byte 1.

This, however, is not the correct solution to the problem of endian differences. In fact, the system software must swap bytes to compensate for the design error. Specifically, C preprocessor macros filter accesses to memory in the DPRAM (see Section 10.4). On the host side, the macros swap bytes; on the DSP side, they do nothing. These macros are sensitive to the state of a compile-time flag so that code may be reused directly on newer systems without a byte-swapping problem.

The system queue functions *VdiQueueGet()* and *VdiQueuePut()*, automatically hide this swapping problem from applications that use them. However, applications that access DPRAM directly must use the macros to ensure correct behavior.

Note that these macros simply undo the swapping; they do *not* correct for endian differences. Applications must take these differences into account when exchanging data between the host and DSPs. For a *char* at offset x modulo 4, the recipient must look to offset 3 - x to find the data. For a *short* at offset x modulo 4 (which must be 0 or 2), the recipient must look to offset 2 - x. For an *int* or *long*, the recipient need not do anything special. For a string, the recipient must swap bytes within each group of 4 characters, as discussed in Section 5.4.3.

10.4 Macros and Constants

The system software provides some C preprocessor constants and macros to facilitate software development. The following sections describe them.

10.4.1 VDI_VXWORKS_MODE

VDI_VXWORKS_MODE is defined in modules compiled for the host. Files can test this flag to tailor source code that the host and DSPs share through *#if defined(VDI_VXWORKS_MODE)* ... *#endif* sequences.

10.4.2 VDI_DSP_MODE

VDI_DSP_MODE is defined in modules compiled for the DSPs. Files can test this flag to tailor source code that the host and DSPs share through *#if defined(VDI_DSP_MODE)* ... *#endif* sequences.

10.4.3 VDI_FIX_BYTE_ORDER

VDI_FIX_BYTE_ORDER is defined in modules compiled for the host on systems that require byte ordering correction as described in Section 10.3. Application code should not have to test this flag since the macros discussed in Sections 10.4.7 through 10.4.10 hide this hardware detail.

10.4.4 VDI_WORD_SIZE

VDI_WORD_SIZE is the number of bytes in a *VdiWord* or *VdiInt*: 4.

10.4.5 VDI_WORDS()

VDI_WORDS(b) returns the number of *VdiWords* that *b* bytes occupy.

10.4.6 VDI_BYTES()

VDI_BYTES(w) returns the number of bytes that *w* *VdiWords* occupy.

10.4.7 VDI_WORD_PUT()

VDI_WORD_PUT(w) returns the value of *VdiWord* *w* scrambled to correct for byte ordering problems, if necessary, in preparation for a write to a memory location on a DSP board, whether it is a register or the DPRAM.

For example, an application wishing to write the value of *x* to the DPRAM location to which *p* points must use the statement:

```
    *p = VDI_WORD_PUT(x);
```

10.4.8 VDI_WORD_GET()

VDI_WORD_GET(w) returns the value of *VdiWord w* scrambled to correct for byte ordering problems, if necessary, assuming *w* resulted from a direct read from a memory location on a DSP board, whether it is a register or in DPRAM. The actual definition of *VDI_WORD_GET()* is identical to that of *VDI_WORD_PUT()* because the scrambling operation is its own inverse.

For instance, an application wishing to assign to *x* the value of the DPRAM location to which *p* points must use the statement:

```
    x = VDI_WORD_GET(*p);
```

10.4.9 VDI_INT_PUT()

VDI_INT_PUT() is analogous to *VDI_WORD_PUT()*, but operates on *VdiInts*.

10.4.10 VDI_INT_GET()

VDI_INT_GET() is analogous to *VDI_WORD_GET()*, but operates on *VdiInts*.

10.4.11 VdiWordReorder()

VdiWordReorder(w) returns the value of *VdiWord/VdiInt w* with its bytes swapped: byte 0 is exchanged with byte 3, and byte 1 with byte 2.

VdiWordReorder() is defined only on the host side and scrambles bytes regardless of the state of the compile-time byte swapping flag *VDI_FIX_BYTE_ORDER*. It is most useful for correcting the ordering of characters in strings shared by the host and DSPs.

VdiWordReorder() is either an asm macro or C function, depending on which C compiler is used, to provide for very rapid byte swapping. With the Gnu C compiler, for instance, it translates into only 3 inline assembly language instructions.

Name	Description
d3as	Assembler
d3cc	C compiler
d3ld	Linker/loader
d3ar	Librarian
d3dump	Object file dumper
d3nm	Symbol table dumper
d3sim	Simulator
d3genmemimage	Image file generator

Table 10.4: The DSP32C development tools.

10.4.12 VDI_TAG_APPLICATION

The constant *VDI_TAG_APPLICATION* represents the lowest tag value applications may use; the system software reserves lower numbered tags for its own use.

10.4.13 VDI_ERROR_APPLICATION

The constant *VDI_ERROR_APPLICATION* represents the lowest error code applications may use; the system software reserves lower numbered codes for its own use.

10.5 DSP32C Tools

Table 10.4 lists some of the utilities that support DSP32C cross development on the workstation. Consult [9], [8], and [7] for more information on all but *d3genmemimage*, which is discussed in Section 10.6.5.

10.6 DSP Code Memory Layout

10.6.1 d3cc Object Files

Object files that *d3cc* generates are divided into *sections*. Each section contains a group of related information. Normally, *d3cc* generates three sections in each file: *.text* with the executable code, *.data* with the static data, and *.bss* with an initial word for the stack. Command line options can instruct *d3cc* to use other section names.

These *.d3o* object files are in what is known as the *Common Object File Format (COFF)*, and they contain relocation information that the linker needs.

10.6.2 d3ld Binary Executable Files

The DSP linker, *d3ld*, takes object files that *d3cc* generates and lays them out in physical memory. A DSP can directly execute the code *d3ld* generates when loaded into memory.

The *.d3bin* files it produces are in *COFF* format also, but contain absolute addresses.

10.6.3 Load-time Parameter Section Fixup

During the initialization phase, the DSPs need the hardware information from the system parameter file. If this information is not compiled into the code, then the DSPs must obtain it later. They cannot access it at run time becase they cannot communicate with the host until they know the control register and queue addresses, for example.

The system software circumvents this difficulty by storing the global hardware information structure (to which *pVdiDSP* points) in a special section named *.parms* rather than in *.data*. The *.d3bin* executable files that *d3ld* generates initially contain zeros in the region that *.parms* occupies. Whenever the system software loads a *.d3bin* file, it looks for a *.parms* section. If it finds one, it overwrites it with the information from the appropriate entry in the global array *d[]* of *pVdiDSPs*. Thus the code it sends to the DSP during the initialization phase contains the necessary information.

Address	Length (bytes)	Description
0x000000	0x004000	DPRAM
0x004000	0x010000	Static RAM
0x014000	0x004000	EPROM
⋮	⋮	⋮
0xffe000	0x000800	On-chip RAM
0xffe800	0x000800	Reserved
0xfff000	0x001000	On-chip RAM

Table 10.5: The DSP address space.

The DSP definition of the type *VdiDSP* contains padding to fill out every field to 32-bits to ensure that when the host writes information into the *.parms* section, it falls into the places where the DSP expects it, given its definition of *VdiDSP*.

10.6.4 Linker Memory Map Files

Special *.map* files tell *d3ld* where to place each object file section in memory. A *MEMORY* directive assigns symbolic names to regions of physical memory. A *SECTIONS* directive then names the memory region to which each section (*.text*, *.data*, ...) belongs.

Table 10.5 depicts the DSP address space. The the system operates the DSPs in *mode 6* (see [10] for an explanation of DSP32C memory modes). The locations of the EPROM and DPRAM may be interchanged if desired in order to start the system from code in EPROM.

Accesses to off-chip memory incur wait states while on-chip references do not. The map files described below place the stack in the smaller on-chip memory bank for rapid stack variable access; the other bank is free for application use.

Boot Code

The map file for the boot code contains:

```
MEMORY {
        boot_ram (RWX): origin=0x000000, length=0x02000
        ext_ram (RWX):  origin=0x004000, length=0x10000
        eprom (RX):     origin=0x014000, length=0x04000
        int_ram1 (RWX): origin=0xffe000, length=0x00800
        int_ram2 (RWX): origin=0xfff000, length=0x01000
}

SECTIONS {
        .text 0x000000: { } > boot_ram
        .data:          { } > boot_ram
        .parms:         { } > boot_ram
        .bss:           { } > int_ram2
        .mon:           { } > ext_ram
}
```

This places the boot code in the lower half of DPRAM, leaving the rest free for the communication queues. The startup code uses the section *.mon* to locate the monitor's region so that it may branch there when it has finished loading the monitor.

Monitor Code

The map file for the monitor contains:

```
MEMORY {
        mon_ram (RWX):  origin=0x004000, length=0x03000
        app_ram (RWX):  origin=0x007000, length=0x0d000
        eprom (RX):     origin=0x014000, length=0x04000
        int_ram1 (RWX): origin=0xffe000, length=0x00800
        int_ram2 (RWX): origin=0xfff000, length=0x01000
}

SECTIONS {
        .text 0x004000: { } > mon_ram
        .data:          { } > mon_ram
        .parms:         { } > mon_ram
        .bss:           { } > int_ram2
}
```

This places the monitor code into the lowest locations of the static RAM, reserving the rest for applications.

Application Code

The map file for applications contains:

```
MEMORY {
        app_ram (RWX):  origin=0x007000, length=0x0d000
        int_ram1 (RWX): origin=0xffe000, length=0x00800
        int_ram2 (RWX): origin=0xfff000, length=0x01000
}

SECTIONS {
        .text 0x007000: { } > app_ram
        .data:          { } > app_ram
        .parms:         { } > app_ram
        .bss:           { } > int_ram2
}
```

This places the applications in the region of static RAM just following the monitor.

Applications may alter this map file to define extra object code sections or to access the free bank of fast on-chip RAM.

10.6.5 d3genmemimage Image Files

Mani Srivastava experimented with an AT&T library of routines for accessing COFF files and wrote a set of functions to interface those low-level routines with the system software. As a result, the system can now directly load COFF files that *d3ld* generates.

Prior to discovering this library, however, we captured the output of *d3dump* and generated *.d3img* files, which the system software then read. Mani wrote the current version of *d3genmemimage* using the library to perform this conversion rapidly as a transition step toward direct COFF file loading. The system software no longer needs this tool, but it is sometimes useful for analyzing *.d3bin* files since its *.d3img* are human-readable.

For each object file section, these *.d3img* files include a header line with the section name, address, and length in bytes followed by a series of data lines with the section contents. Each data line represents one 32-bit word. All numbers, both in the header and data lines, are in hexadecimal. For example, a *.d3img* file might contain the following:

```
.text 00000000 00000010
00000000
11111111
22222222
33333333
.data 00000010 00000008
55AA55AA
87654321
```

to define a *.text* section at address *0x0* of length *0x10* bytes and a *.data* section at address *0x10* of length *0x8* bytes.

10.6.6 Libraries

The file *lib/libvdi.o* in the system software directory contains the core synchronization and communication library routines for the host. VxWorks performs linking at load time, so when host code for an application is loaded after this library, it will automatically reference the appropriate routines, and only one copy of them will ever exist in host memory.

The situation is different for DSP code, however. The DSP library resides in *lib/libvdi32c.a*. The boot code, monitor code, and DSP application code all link to it. The linker extracts the referenced routines from the library and packages them together with each independently. Since the boot code, monitor, and applications all typically reference *VdiQueueGet()*, for example, that routine is duplicated in memory three times. Those routines that only the application calls, such as *VdiPrintf()*, exist in only one place.

Actually, since the boot code ceases to exist after DSP initialization, and the memory it occupies is reclaimed for the remote input-output queues, it does not contribute to any memory "waste." Only those routines which both the monitor and DSP access are duplicated in static RAM. The total size of these routines is not very large, so the wastage may not be important in most applications.

10.7 Header Files

The system provides several C header files that define various data types and constants and declare the system functions. The sections that follow describe the contents of those files which applications should find useful.

All applications must include *vdiApplication.h* before any other system include files; that file includes *vdiCommon.h* for basic system definitions and *vdiErrorCodes.h* for function return codes. The manual page for each system function mentions the other files that applications must include before calling it.

10.7.1 vdiAddresses.h

The file *vdiAddresses.h* defines the internals of the type *VdiDSP* and declares the variable *pVdiDSP* or *pVdiDSPs* for the DSPs or host, respectively. Applications wishing to access these global hardware information variables must include this file.

10.7.2 vdiApplication.h

The file *vdiApplication.h* provides a basic set of definitions that all applications need. It includes the files *vdiCommon.h* and *vdiErrorCodes.h.*

10.7.3 vdiCommon.h

The file *vdiCommon.h* declares the most primitive system data types, constants, and macros. Among them are *VdiWord, VdiInt, VDI_WORD_SIZE, VDI_WORDS()*, and VDI_BYTES().

10.7.4 vdiErrorCodes.h

The file *vdiErrorCodes.h* defines symbolic constants that represent the error return codes from all system functions. The manual pages list the error codes that every system function may return.

10.7.5 vdiErrorReport.h

The file *vdiErrorReport.h* declares the function *VdiErrorReport()* for applications that wish to report fatal errors directly to the host.

10.7.6 vdiIO.h

The file *vdiIO.h* provides applications with access to the remote input-output console. It declares *VdiPrintf()* and *VdiScanf()* and defines symbolic constants for the various data types they handle.

10.7.7 vdiMessage.h

The file *vdiMessage.h* declares the functions *VdiMessageSend()* and *VdiMessageReceive()* for applications that need to send data larger than the capacity of a queue.

10.7.8 vdiMonitor.h

The file *vdiMonitor.h* defines symbolic constants for the packet tags for each monitor commmand as well as the structures for parameters to those commands. Applications that wish to speak with a DSP monitor directly, bypassing the host monitor support functions such as *VdiBlockPut()*, must include this file.

10.7.9 vdiQueue.h

The file *vdiQueue.h* defines the internals of the *VdiQueue* structure and declares the routines *VdiQueueInit()*, *VdiQueuePut()*, and *VdiQueueGet()* that operate on queues.

10.7.10 vdiSemaphore.h

The file *vdiSemaphore.h* declares the functions *VdiSemaphoreInit()*, *VdiSemaphoreP()*, and *VdiSemaphoreV()* that operate on semaphores. Applications that wish to create their own semaphores for synchronization schemes that the queues do not support must include this file.

10.7.11 vdiSharedMemory.h

The file *vdiSharedMemory.h* defines the macros *VDI_WORD_PUT(), VDI_WORD_GET(), VDI_INT_PUT()*, and *VDI_INT_GET()*. In order to access the DPRAM directly, applications must include this file and use the macros to automatically correct byte ordering problems, if present.

10.7.12 vdiWordReorder.h

The file *vdiWordReorder.h* defines *VdiWordReorder()* for swapping bytes in a word. Applications that share character strings between the host and DSPs, for instance, must include this file to correct for the endian differences between the two sides.

10.8 Application Program Structure

The following sections explain the requirements that the system hardware and software impose on application code.

10.8.1 Host Code

The VxWorks environment is more mature and complete than the DSP environment that the system software provides, so host code faces few restrictions. Consult [6] for details on VxWorks software development; the system software does not add any further requirements.

Note that application host code must *not* use libraries on the workstation; it must use the header files and libraries supplied with VxWorks.

10.8.2 DSP Code

Header Files

All modules *must* include the header file *vdiApplication.h* to access system software data types and functions, in addition to other header files that specific functions mandate. The *#include* statement for this file must precede that for any other system header file.

Module Layout

Applications execute under control of the monitor on a DSP. Since the monitor does not have access to any code entry point information in *.d3bin* files, it makes the simple assumption that the beginning of the executable portion of the application lies right at the beginning of the application memory space in static RAM.

This means that the top-level application function, which does not necessarily have to be named *main()*, *must* precede all other code and data in the source file. If the application consists of multiple modules that are later linked together, the module with the top-level function must precede all others; naming this module before all others on the *d3ld* command line guarantees this (see Section 10.9.2).

Return Codes

The monitor expects an *int* return code from any application it executes, so the top-level application function must be a function with return type *int*. Also, since the monitor does not pass any parameters to applications, the top level function must not take any arguments.

After calling an application, the monitor checks the return code; if it is nonzero, it interprets it as an error code and passes it to the host via the fatal error reporting routine *VdiErrorReport()* and enters an infinite loop. Thus applications must return 0 when successful and an appropriate status code under fatal error conditions.

Limited Stack Space

To provide rapid access to stack-based data, the system software keeps the DSP stack in on-chip RAM, as discussed in Section 10.6.4. This region is limited to 2K bytes or 512 words in size, so applications must be careful not to overflow it — highly recursive routines may not work.

10.8.3 Sample

Figure 10.1 shows the source code for the sample application corresponding to the script of Section 9.4. This application consists of just one module, and the top-level function *main()* is the first piece of code or data in the file. It

```
#include "vdiApplication.h"
#include "vdiIO.h"

int main()
{
    long x;
    char string[VDI_IO_TEXT_MAX];

    VdiPrintf("DSP Test Application", 0, VDI_NONE, 1);
    VdiScanf("Enter an integer value for X: ", &x, VDI_LONG);
    x = x * x;
    VdiPrintf("X * X = ", &x, VDI_LONG, 1);

    VdiScanf("Enter a string: ", string, VDI_STRING);
    VdiPrintf("You typed: ", 0, VDI_STRING, 1);
    VdiPrintf(string, 0, VDI_STRING, 0);

    return(0);
}
```

Figure 10.1: A sample DSP application.

includes *vdiIO.h* in order to use the remote input/output console functions. It returns 0 when finished to signal successful completion to the monitor.

Refer to Appendix C for a detailed look at a more involved application.

10.9 Building Applications

The system software provides some support for the utility *make* to simplify application building. The sections that follow cover the steps involved.

10.9.1 Environment

In addition to following the steps outlined in Section 9.1.1, you must set the environment variable *DSP32SL* to name the directory containing the AT&T DSP32C Support Software Library, and the variable *VXWORKS* to name the directory containing the VXWORKS software. For example:

```
setenv DSP32SL /usr/cadtools/dsp/dsp32sl
setenv VXWORKS /usr/cadtools/vw
```

You must also add the *bin* directory inside the DSP support software directory to your path to provide access to the utilities in Table 10.4.

10.9.2 Default Make Inference Rules

The file *vdiRules.make* in the top-level system software directory provides the vast majority of the makefile definitions applications require. Application makefiles should include it with the *make* command:

```
include $(VDI)/vdiRules.make
```

Read the comments at the top of the file for the most up-to-date information.

Directories

Just prior to the include command, application makefiles must set the *make* variables *DIR_SRC, DIR_OBJ*, and *DIR_BIN* to name the directories for holding the application source, object, and executable binary code, respectively. For a simple application with few modules, one directory suffices for all of these, so the statements:

```
DIR_SRC = .
DIR_OBJ = .
DIR_BIN = .
```

suffice, naming the application directory with the makefile for all three purposes.

DSP Tools

The file *vdiRules.make* defines the following *make* variables to access the host and DSP tools:

```
CC      = gcc
DSP_AR  = d3ar
DSP_AS  = d3as
DSP_CC  = d3cc
DSP_LD  = d3ld
DSP_IMG = d3genmemimage
```

Assembling DSP Code

The rule:

```
$(DIR_OBJ)/%.d3o: $(DIR_SRC)/%.s
    echo "***** Assembling '$<' for the DSP ..."
    $(DSP_AS) -Q -o $@ $<
```

in *vdiRules.make* automatically generates *.d3o* files from *.s* DSP assembler source files.

Compiling DSP Code

The rule:

```
$(DIR_OBJ)/%.d3o: $(DIR_SRC)/%.c
    echo "***** Compiling '$<' for the DSP ..."
    $(DSP_CC) -Q -c $(C_FLAGS_DSP) -o $@ $<
```

in *vdiRules.make* automatically generates *.d3o* files from *.c* DSP C source files. Application makefiles may append extra flags to *C_FLAGS_DSP* prior to including *vdiRules.make*.

Linking DSP Code

The rule:

```
$(DIR_BIN)/%.d3bin: $(DIR_OBJ)/%.d3o
        echo "***** Linking '$<' for the DSP ..."
        $(DSP_LD) $(LD_FLAGS_DSP) -o $@ $< $(LIB_DSP) \
            $(LD_SUFFIX_DSP) $(MAP_APP) | tee $(@:.d3bin=).map
```

in *vdiRules.make* automatically generates *.d3bin* files from *.d3o* DSP object files. It also generates a *.map* file in the application *bin* directory; this map file shows exactly where the linker placed each object module section in physical memory (do not confuse this *.map* file with the *.map* files supplied as input to the linker to specify the memory layout).

Application makefiles may append extra flags to *LD_FLAGS_DSP* prior to including *vdiRules.make* to alter the linker's behavior.

LD_SUFFIX_DSP is initialized to include references to the libraries *libvdi32c.a, libm32c.a, libap32c.a*, and *libc32c.a* in that order (see [7] for more information on the AT&T supplied libraries). To search other libraries or object files before or after these, applications may append extra *-l* flags or object file names to *LIB_DSP* or *LD_SUFFIX_DSP*, respectively.

To use a memory layout file different from the default (discussed in Section 10.6.4), application makefiles may set *MAP_APP* to the name of the file to use.

If the application consists of several modules, the names of all but the main module should be appended to *LIB_DSP* as well, and the rule applied to the main module only.

For instance, assume the directory *mylib* contains application libraries named *libtools32c.a* and *librobot32c.a*, that the file *util.d3o* also contains some library functions but is in *.d3o* format, that the main application module is in *main.d3o*, and that the remaining modules are in *init.d3o* and *move.d3o*. Then the following sequence of *make* commands:

```
LD_FLAGS_DSP += -Lmylib
LIB_DSP += init.d3o move.d3o util.d3o -ltools32c -lrobot32c
```

will set up the correct environment so that a command to build the target *main.d3bin* will properly combine all necessary modules. The *-L* flag tells *d3ld* to also search *mylib* for libraries.

Generating DSP Image Files

The rule:

```
%.d3img: %.d3bin
	echo "***** Creating the memory image for '$<' ..."
	$(DSP_IMG) $< > $@
```

in *vdiRules.make* builds *.d3img* files from *.d3bin* files, as discussed in Section 10.6.5.

Checking DSP Code with lint

The file *vdiRules.make* does not provide an inference rule for running *lint* on DSP code. Instead, application makefiles should define a rule on their own. For instance, the rule:

```
dsp.lint:
	echo "***** Checking DSP code ..."
	$(LINT) $(LINT_FLAGS_DSP) $(DSP_SRC) $(LINT_VDI_DSP)
```

checks the application source files named in *DSP_SRC* against the *lint* library *LINT_VDI_DSP* for the system functions. The file *vdiRules.make* defines *LINT_FLAGS_DSP*, but application makefiles may append extra flags to it prior to including *vdiRules.make*.

Compiling Host Code

The rule:

```
$(DIR_OBJ)/%.o: $(DIR_SRC)/%.c
	echo "***** Compiling '$<' for VxWorks ..."
	$(CC) -c $(C_FLAGS_HOST) -o $@ $<
```

in *vdiRules.make* automatically generates *.d3o* files from *.c* host C source files. Application makefiles may append extra flags to *C_FLAGS_HOST* prior to including *vdiRules.make*.

Linking Host Code

The file *vdiRules.make* does not provide an inference rule for linking host code. Instead, application makefiles should define a rule on their own. For instance, the rule:

```
host.o: $(HOST_OBJS)
    echo "***** Linking VxWorks code ..."
    $(LD) $(LD_FLAGS_HOST) -o $@ $(HOST_OBJS) \
        $(LD_SUFFIX_HOST) | tee $(@:.o=).map
```

combines all modules named in *HOST_OBJS* with any necessary functions from the libraries mentioned in *LD_SUFFIX_HOST* to produce *host.o.* The file *vdiRules.make* defines *LD_FLAGS_HOST* and *LD_SUFFIX_HOST*, but application makefiles may append to these variables prior to including *vdiRules.make.*

Note that one of the flags in *LD_FLAGS_HOST* is *-r* to instruct the host linker to combine the modules into another *relocatable .o* file since VxWorks dynamically links binary files when it loads them.

Checking Host Code with lint

The file *vdiRules.make* does not provide an inference rule for running *lint* on host code. Instead, application makefiles should define a rule on their own. For instance, the rule:

```
host.lint:
    echo "***** Checking host code ..."
    $(LINT) $(LINT_FLAGS_HOST) $(HOST_SRC) $(LINT_VDI_HOST)
```

checks the application source files named in *HOST_SRC* against the *lint* library *LINT_VDI_HOST* for the system functions. The file *vdiRules.make* defines *LINT_FLAGS_HOST*, but application makefiles may append extra flags to it prior to including *vdiRules.make.*

10.9.3 Sample Makefiles

The simplest possible application makefile contains only:

```
.SILENT:

DIR_SRC = .
DIR_OBJ = .
DIR_BIN = .
include $(VDI)/vdiRules.make

test.d3bin:
```

The statement *.SILENT:* tells *make* not to echo the commands it executes; the inference rules explicitly display progress messages that do not clutter the screen.

In order to build *test.d3bin*, *make* looks for *test.d3o*. To build *test.d3o*, it searches for *test.c*. The inference rules in *vdiRules.make* tell it how to build each file.

A slightly more complicated makefile might look like this:

```
.SILENT:

DIR_SRC = src
DIR_OBJ = obj
DIR_BIN = bin
C_FLAGS_DSP += -l
LD_FLAGS_DSP += -Llib
LD_FLAGS_HOST += -Llib

DSP_OBJ = $(DIR_OBJ)/init.d3o $(DIR_OBJ)/move.d3o
DSP_SRC = $(DIR_SRC)/main.c $(DIR_SRC)/init.c \
          $(DIR_SRC)/move.c
LIB_DSP += $(DSP_OBJ) lib/utils.d3o -lrobot32c -ltools32c

HOST_SRC = $(DIR_SRC)/control.c $(DIR_SRC)/report.c
HOST_OBJ = $(DIR_OBJ)/control.o $(DIR_OBJ)/report.o

include $(VDI)/vdiRules.make

all:
    main.d3bin
    host.o
```

```
main.d3bin: $(DSP_OBJ)

host.o: $(HOST_OBJ)
    echo "***** Linking VxWorks code ..."
    $(LD) $(LD_FLAGS_HOST) -o $@ $(HOST_OBJ) \
        $(LD_SUFFIX_HOST) | tee $(@:.o=).map

dsp.lint:
    echo "***** Checking dsp code ..."
    $(LINT) $(LINT_FLAGS_DSP) $(DSP_SRC) $(LINT_VDI_DSP)

host.lint:
    echo "***** Checking host code ..."
    $(LINT) $(LINT_FLAGS_HOST) $(HOST_SRC) $(LINT_VDI_HOST)
```

The definitions of *DIR_SRC, DIR_OBJ*, and *DIR_BIN* name special directories for holding various types of files. The addition to *C_FLAGS_DSP* tells *d3cc* to generate listing files showing the assembler source into which it compiles C source. The additions to *LD_FLAGS_DSP* and *LD_FLAGS_HOST* tell the linkers to search the directory *lib* for libraries. The additions to *LIB_DSP* name the secondary DSP application code modules and the application libraries.

The variables *DSP_SRC, DSP_OBJ, HOST_SRC*, and *HOST_OBJ* name the various source and object modules for the DSP and host. The target *all* forces *make* to generate both the DSP and host binaries when invoked with no parameters.

The targets *dsp.lint* and *host.lint* force *make* to run *lint* on the source files when invoked with:

```
make dsp.lint host.lint
```

Chapter 11

System Software Internals

The preceeding sections of this report cover most of the internal details of the system software. The miscellaneous extra details included in this section, combined with the numerous comments throughout the code, should provide sufficient information for understanding and maintaining the code.

11.1 System Software Directory

Table 11.1 summarizes the contents of the system software directory and the subdirectories within it.

11.2 Module Overview

The following list briefly describes each source code module:

vdiAddresses.c: Defines the global hardware information structures and pointer variables *pVdiDSP* for the DSPs and *pVdiDSPs* for the host.

vdiBoot.c: Defines the function *main()* for the DSP startup code.

vdiBootStartup.s: Contains a modified version of the standard C startup code in *crt0_32c.s.* It is linked together with only the module *vdiBoot.* It is the first code a DSP actually executes (even before the function *main()* of *vdiBoot.c*) and is responsible for initializing the stack and

Name	Description
app/	Sample DSP application
bin/	Binary files: vdiIOConsole vdiHostCode.o
coff/	COFF library
include/	Header files
lib/	Libraries: libvdi.o libvdi32c.a
lint/	Lint libraries: llib-lvdi.ln llib-lvdi32c.ln
man/	Online manual pages
map/	Memory layout files: vdiBoot.map vdiMonitor.map vdiApplication.map
obj/	Object code: .d3o files .o files
src/	Source code: .c files .lex files .s files
clean-lex.sed	Sed script to clean lex output
Makefile	System makefile
system.parms	Sample parameter file
vdiDependencies.make	C header file dependencies
vdiRules.make	Common make rules

Table 11.1: The contents of the system software directory. Subdirectories are marked with a trailing slash.

number of memory wait states. After it calls the boot code, it branches to the monitor that the boot code just loaded.

vdiDataCopy.c: Defines *VdiDataCopy()* for copying a variable of any supported remote input-output data type from one place to another.

vdiErrorReport.c: Defines *VdiErrorReport()* to allow DSP programs to report severe errors to the host.

vdiErrorServer.c: Defines *VdiErrorServer()* which the host error server task executes to detect DSP errors.

vdiHost.c: Defines *VdiInit()* for initializing the entire system and the support functions *VdiLoadBootCode()*, *VdiLoadMonitorCode()*, and *VdiTest()*. It also contains *VdiShutdown()* for bringing down the system and *VdiCheckInitialized()* that many other routines call to verify that the system is up before attempting to communicate with a DSP.

vdiIEEE2DSP.c: Defines *IEEE2DSP()* and *DSP2IEEE()* to convert between the DSP and IEEE floating point formats. The DSP library does contain functions for just this purpose, but they do not always work correctly.

vdiIOConsole.c: Defines the routine *main()* containing the remote I/O console server code. For more information on the techniques it employs for accessing Unix *sockets*, consult [3].

vdiIORequester.c: Defines *VdiPrintf()* and *VdiScanf()* to allow applications to request remote console services.

vdiIOForwarder.c: Defines the function *VdiIOForwarder()* that the host remote input-output forwarder tasks execute. Consult [3] for more information on the techniques it uses to operate on Unix *sockets*.

vdiImage.c: Defines the routines for operating on COFF format binary files. *VdiImageLoad()* loads files into host data structures, *VdiImageParametersStore()* writes hardware information from a DSP's perspective into the *.parms* section of the structures, and *VdiImageFree()* deallocates the structures. These routines call functions in the library *libcoff.a*. It includes low level COFF access routines from an AT&T library and some higher-level interface functions that Mani Srivastava wrote.

vdiMessageReceive.c: Defines *VdiMessageReceive()* to receive messages through queues.

vdiMessageSend.c: Defines *VdiMessageSend()* to send messages through queues.

vdiMonitor.c: Contains the routine *main()* for the DSP monitor and the supporting routines *VdiMonBlockGet()*, *VdiMonBlockPut()*, and *VdiMonLoopback()* for servicing monitor commands.

vdiNullStartup.s: Contains an almost empty copy of the standard C start-up code in *crt0_32c.s.* It is linked together with all DSP code except *vdiBoot.* It simply defines the variable *errno* for C library functions and contains no executable code.

vdiParameters.c: Defines the function *VdiParametersGet()* for reading the system hardware parameter file and the supporting routine *VdiDSPGet()* for reading each DSP structure within the file. These functions use the *lex* tokenizer in *vdiTokenize.lex* to scan the file.

vdiQueueDisplay.c: Defines *VdiQueueDisplay()* for viewing the contents of a queue.

vdiQueueGet.c: Defines *VdiQueueGet()* for reading packets in queues.

vdiQueueInit.c: Defines *VdiQueueInit()* for formatting parts of DPRAM into empty queues.

vdiQueuePut.c: Defines *VdiQueuePut()* for adding packets to queues.

vdiSemaphore.c: Defines *VdiSemaphoreInit()* for creating semaphores in DPRAM and *VdiSemaphoreP()* and *VdiSemaphoreV()* for operating on them.

vdiTokenize.c: Contains the output that the tool *lex* generates from *vdiTokenize.lex.* The *lex* output is filtered with the *sed* script *clean-lex.sed* to prefix all *lex yy...* symbols with *vdi_* to ensure that they do not clash with the *lex* and *yacc* symbols of the VxWorks shell.

vdiTokenize.lex: Contains the *lex* source file describing the parameter file tokenizer.

vdiTools.c: Contains *VdiLoopback()*, *VdiMemDump()*, *VdiMemSet()*, *VdiBlockGet()*, *VdiBlockPut()*, *VdiAppLoad()*, *VdiAppRun()*, and *VdiAppLoadAndRun()* to interact with DSP monitors and examine or modify VME memory.

11.3 Building the System Software

The file *Makefile* contains all the information necessary for building the system software. The following list describes the targets it defines:

vdi: Builds all system software binary files, including *bin/vdiIOConsole*, *bin/vdiHostCode.o*, *lib/libvdi.o*, and *lib/libvdi32c.a*. This is the default target, so *make* will build these files if invoked without any arguments.

dep: Scans all C source files for *#include* directives and builds the file *vdiDependencies.make* to reflect the header file dependencies. Note that *vdiDependencies.make* must exist in order to use the *Makefile*; if it doesn't exist, create an empty version first with *touch* and then build the proper dependencies with this target.

vdi.dsp.lint: Checks all DSP system software library modules.

vdi.host.lint: Checks all host system software library modules.

boot.lint: Checks the module *vdiBoot*.

monitor.lint: Checks the module *vdiMonitor*.

host.lint: Checks all host support modules.

console.lint: Checks the console server modules.

Chapter 12

Conclusion

Our initial experiences with the common hardware/software design platform have shown it to be extremely worthwhile. The time and effort we invested in designing, prototyping, and debugging the system software and the first custom DSP board dramatically reduced the time and effort we needed to create a second working system.

Furthermore, because both systems, one for controlling a robot and another for processing video images, share the interface the platform defines, the two may easily work together to produce a "seeing" robot that can track moving objects in real time.

In restrospect, however, two changes to the hardware platform would significantly improve the efficiency of the systems built upon it. First, using dual port memory chips that provide hardware semaphores would provide cleaner, faster synchronization. Second, using a local processor that is a true 32-bit integer machine, unlike the DSP32C, and which comes with a more intelligent compiler and other development tools, would speed up the communication routines. In fact, having a better compiler would have simplified the development of the system software; we spent a great deal of time hand optimizing C source code to lead the AT&T compiler to produce more efficient assembly code.

Appendix A

Acknowledgements

Professor Bob Brodersen has been the guiding force in our efforts to reduce the time we spend prototyping custom systems.

Mani Srivastava and Bill Baringer conceived many of the ideas I implemented in the system software. Mani and Bill also tested and helped debug the software by putting it to actual use in developing their own robot control and image processing systems, respectively.

Dragutin Petkovic, of the *IBM Almaden Research Center* in San Jose, California, provided inspiration and feedback on the design.

Wayne Niblack, also from *IBM Almaden*, aided me in porting the high accuracy line measurement software to this new platform by explaining how the algorithms work and what results I should expect.

I am deeply grateful to these people for the time and effort they invested in this project. Hopefully, the common platform we developed together will simplify and expedite future custom design efforts.

Appendix B

Online Manual Pages for System Software Functions

The following pages contain copies of the system software manual pages that are also accessible online. The topics *support* and *vdi* in *section 1* summarize the host support code and system service library, respectively. *Section 2* contains pages describing each individual function in greater detail.

These manual pages are *not* substitutes for this report. They contain specific details about the functions but assume familiarity with the structure and operation of the system software, its terminology and data types, and procedures for developing applications atop it.

NAME

vdi -- VDI synchronization and communication library

SYNOPSIS

VdiErrorReport()	Report a fatal error to the host
VdiMessageSend()	Send a message through a queue
VdiMessageReceive()	Receive a message from a queue
VdiPrintf()	Send output to the remote I/O console
VdiQueueGet()	Get a packet from a queue
VdiQueueInit()	Initialize a queue
VdiQueuePut()	Put a packet into a queue
VdiScanf()	Receive input from the remote I/O console
VdiSemaphoreInit()	Initialize a semaphore
VdiSemaphoreP()	Take a semaphore
VdiSemaphoreV()	Release a semaphore

DESCRIPTION

This library provides a set of routines for synchronizing tasks running on the host with programs running on the DSP and for allowing them to communicate through the shared memory.

These routines are available to both host and DSP programs, except for VdiErrorReport(), VdiPrintf(), and VdiScanf(), which may be called only by DSP programs.

INCLUDE FILE

vdiCommon.h

NAME

support -- VDI system support routines

SYNOPSIS

vdiIOConsole	Remote DSP I/O console server program
VdiAppLoad()	Load an application into DSP memory
VdiAppLoadAndRun()	Load and run a DSP application
VdiAppRun()	Run a loaded DSP application
VdiBlockGet()	Read a block of DSP memory
VdiBlockPut()	Write a block of DSP memory
VdiInit()	Initialize the system
VdiLoopback()	Run a loopback test routine
VdiMemDump()	Display a block of host memory
VdiMemSet()	Write to a word of host memory
VdiQueueDisplay()	Display the contents of a queue
VdiShutdown()	Shutdown the system

DESCRIPTION

These routines provide a means of starting & stopping the system, manipulating DSP memory, running DSP applications, servicing remote I/O requests from DSP programs, and testing & benchmarking the communication channels.

vdiIOConsole runs on a workstation. The other routines all run on the host machine.

NAME

VdiAppLoad -- Load an application into DSP memory

SYNOPSIS

```
int VdiAppLoad(dspId, imageFile)
    int dspId;                  /* DSP identification number */
    char *imageFile;            /* Name of application code file */
```

DESCRIPTION

VdiAppLoad() loads the compiled, linked DSP application in imageFile into DSP #dspId's memory for later execution.

RETURN VALUE

0 if successful. VDI_ERROR_INIT_SYS_NOT_UP if the system has not been initialized. VDI_ERROR_IMAGE_FILE if the application code cannot be loaded from the file.

SEE ALSO

VdiAppRun()

NAME
    VdiAppLoadAndRun -- Load and run a DSP application

SYNOPSIS
```
int VdiAppLoadAndRun(dspId, imageFile)
    int dspId;                    /* DSP identification number */
    char *imageFile;              /* Name of application code file */
```

DESCRIPTION
    VdiAppLoadAndRun() loads the compiled, linked DSP application in imageFile into DSP #dspId's memory and then executes it. It does not wait for the application to finish before returning.

RETURN VALUE
    0 if successful. VDI_ERROR_INIT_SYS_NOT_UP if the system has not been initialized. VDI_ERROR_IMAGE_FILE if the application code cannot be loaded from the file.

SEE ALSO
    VdiAppLoad()
    VdiAppRun()

NAME

VdiAppRun -- Run a loaded DSP application

SYNOPSIS

```
int VdiAppRun(dspId)
    int dspId;                     /* DSP identification number */
```

DESCRIPTION

VdiAppRun() executes the application previously loaded into DSP #dspId's memory. It does not wait for the application to finish before returning.

RETURN VALUE

0 if successful. VDI_ERROR_INIT_SYS_NOT_UP if the system has not been initialized.

SEE ALSO

VdiAppLoad()

NAME

VdiBlockGet -- Read a block of DSP memory

SYNOPSIS

```
int VdiBlockGet(dspId, address, nWords, pData)
    int dspId;                  /* DSP identification number */
    VdiInt address;             /* Address of block in DSP memory space */
    unsigned int nWords;        /* Length of block in words */
    void *pData;                /* Pointer to buffer to receive data */
```

DESCRIPTION

<u>VdiBlockGet</u>() returns the contents of a block of DSP #<u>dspId</u>'s memory <u>nWords</u> long beginning at <u>address</u> in the buffer to which <u>pData</u> points.

RETURN VALUE

0 if successful. VDI_ERROR_INIT_SYS_NOT_UP if the system has not been initialized. VDI_ERROR_MESSAGE_BAD if the message from the monitor program containing the block is not properly formatted.

SEE ALSO

VdiBlockPut()

NAME

VdiBlockPut -- Write a block of DSP memory

SYNOPSIS

```
int VdiBlockPut(dspId, address, nWords, pData)
    int dspId;                   /* DSP identification number */
    VdiInt address;              /* Address of block in DSP memory space */
    unsigned int nWords;         /* Length of block in words */
    void *pData;                 /* Pointer to buffer with data */
```

DESCRIPTION

VdiBlockPut() writes the nWords of data to which pData points into DSP #dspId's memory space beginning at address.

RETURN VALUE

0 if successful. VDI_ERROR_INIT_SYS_NOT_UP if the system has not been initialized.

SEE ALSO

VdiBlockGet()

NAME

VdiErrorReport -- Report a fatal error to the host

SYNOPSIS

```
#include "vdiErrorReport.h"

void VdiErrorReport(errorCode, loop)
    int errorCode;              /* Code identifying error */
    int loop;                   /*
                                  Flag indicating that VdiErrorReport()
                                  should enter an infinite loop
                                */
```

DESCRIPTION

VdiErrorReport() communicates a fatal error, given by errorCode, from a DSP program to the host. It does not use the communications queues or the remote I/O console and can thus be trusted even when those lines of communication fail.

If loop is non-zero, VdiErrorReport() will enter an infinite loop after posting the error.

The error message is detected and displayed by the fatal error server task on the host.

VdiErrorReport() may be called only from a DSP program.

RETURN VALUE

None.

NAME

VdiInit -- Initialize the system

SYNOPSIS

```
int VdiInit(parameterFileName)
    char *parameterFileName;    /* Name of file with system parameters */
```

DESCRIPTION

VdiInit() initializes the system, performing all of the steps necessary to bring up the monitor program on each DSP in the system. It must be called exactly once before performing any other operations.

VdiInit() looks in the file parameterFileName for descriptions of all of the components in the system, both from the host's perspective and each DSP's perspective. Consult the manual or the comments in the sample system parameter file for more details.

RETURN VALUE

0 if successful. VDI_ERROR_PARAMETERS_BAD if the parameter file is not correctly formatted. VDI_ERROR_IMAGE_FILE if the bootstrap or monitor code cannot be loaded from the corresponding files. VDI_ERROR_INIT_SPAWN_FAILED if a background task cannot be spawned.

SEE ALSO

VdiShutdown()

NAME
vdiIOConsole -- Remote DSP I/O console server program

SYNOPSIS
vdiIOConsole [port number]

DESCRIPTION
vdiIOConsole services remote I/O requests from DSP applications, as generated by the functions _VdiPrintf_() and _VdiScanf_().

It listens on a socket with the specified _port number_, which must match that supplied to _VdiInit_() in the system parameter file. The number must be greater than 1000.

The first column of each line the server displays shows the identification number of the DSP which made the corresponding request. The next several characters are -> or <- if the request is for output from the DSP or input to the DSP, respectively.

Press _control-C_ to stop the program.

RETURN VALUE
1 if the command line syntax is not correct. VDI_ERROR_IO_SOCKET_FAILED if the socket cannot be created. VDI_ERROR_IO_BIND_FAILED if the socket cannot be bound. VDI_ERROR_IO_LISTEN_FAILED if the server cannot listen on the socket. VDI_ERROR_IO_ACCEPT_FAILED if the server cannot accept a connection from a client. VDI_ERROR_IO_READ_FAILED if the server cannot read a request from the socket. VDI_ERROR_IO_WRITE_FAILED if the server cannot send a response through the socket.

SEE ALSO
VdiPrintf()
VdiScanf()

NAME

VdiLoopback -- Run a loopback test routine

SYNOPSIS

```
int VdiLoopback(dspId, mode, n)
    int dspId;              /* DSP identification number */
    int mode;               /* Test mode number */
    int n;                  /* Number of samples (modes 1 to 4) */
```

DESCRIPTION

VdiLoopback() performs a loopback test of the monitor program running on DSP #dspId. It sends packets to the DSP's monitor through a queue and waits for the monitor to return them through another queue.

The parameter mode sets the test format.

In mode 0, which is interactive, data is read from standard input and sent to the DSP; the receieved data is then displayed. The routine exits when a null line is read. The parameter n is ignored.

In mode 1, a packet with one data word is sent n times.

In mode 2, the largest packet the outgoing queue can hold is sent n times.

Modes 3 and 4 are like 1 and 2, respectively, except that the data is read from and written to the same queue to test the communications routines locally (i.e. the DSP's monitor program is not involved).

In the non-interactive modes 1 through 4, the average round-trip packet transit time is computed and displayed to benchmark the communications routines.

RETURN VALUE

0 if successful. VDI_ERROR_INIT_SYS_NOT_UP if the system has not been initialized.

SEE ALSO

VdiQueuePut()
VdiQueueGet()

NAME

VdiMemDump -- Display a block of host memory

SYNOPSIS

```
void VdiMemDump(address, nWords)
    VdiWord *address;           /* Address of block */
    int nWords;                 /* Length of block */
```

DESCRIPTION

VdiMemDump() displays in hex the contents of nWords of host memory beginning at address. It is useful for displaying the contents of off-board memory, such as the shared communications memory, which the VxWorks memory dump command cannot access.

RETURN VALUE

None.

SEE ALSO

VdiMemSet()

NAME

VdiMemSet -- Write to a word of host memory

SYNOPSIS

```
void VdiMemSet(address, value)
    VdiWord *address;           /* Address of word */
    VdiWord value;              /* Value to write */
```

DESCRIPTION

VdiMemSet() writes value into the word at address in the host's memory space. It is useful for setting the contents of off-board memory, such as the shared communications memory, which the VxWorks memory dump command cannot access.

RETURN VALUE

None.

SEE ALSO

VdiMemDump()

NAME

VdiMessageReceive -- Receive a message from a queue

SYNOPSIS

```
#include "vdiMessage.h"

int VdiMessageReceive(pDSP, pQueue, pTag, pNWords, pData, room)
    VdiDSP *pDSP;              /* Pointer to DSP information */
    VdiQueue *pQueue;          /* Pointer to queue */
    unsigned int *pTag;        /* Pointer to variable to receive tag */
    unsigned int *pNWords;     /* Pointer to variable to receive size */
    void *pData;               /* Pointer to buffer to receive data */
    unsigned int room;         /* Length of buffer in words */
```

DESCRIPTION

VdiMessageReceive() receives a message from the queue. It is analogous to VdiQueueGet() but does not impose restrictions on the length of the data. The message must have been sent by VdiMessageSend().

The message tag is returned in *pTag, the length in *pNWords (in words), and the data in *pData.

No more than room words will be written into *pData; if the message length exceeds room, only the first room words of the message data will be written into the buffer (*pNWords will still be set equal to the actual message length).

RETURN VALUE

0 indicates success.

SEE ALSO

VdiMessageSend()

NAME

VdiMessageSend -- Send a message through a queue

SYNOPSIS

```
#include "vdiMessage.h"

int VdiMessageSend(pDSP, pQueue, tag, nWords, pData)
    VdiDSP *pDSP;               /* Pointer to DSP information */
    VdiQueue *pQueue;           /* Pointer to queue */
    unsigned int tag;           /* Message tag */
    unsigned int nWords;        /* Message length */
    void *pData;                /* Pointer to buffer with message data */
```

DESCRIPTION

<u>VdiMessageSend</u>() sends a long packet, or message, through the queue. It is analogous to <u>VdiQueuePut</u>(), but does not restrict the length of the data to the capacity of the queue.

The parameter <u>tag</u> specifies the message tag, <u>nWords</u> its length (in words), and *<u>pData</u> its contents.

<u>VdiMessageSend</u>() sends a start of message packet, followed by as many packets as necessary to transfer the data, followed by an end of message packet.

RETURN VALUE

0 indicates success.

SEE ALSO

VdiMessageReceive()

NAME

VdiPrintf -- Send output to the remote I/O console

SYNOPSIS

```
#include "vdiIO.h"

void VdiPrintf(pText, pVar, varType, newline)
    char *pText;                /* Identification string */
    void *pVar;                 /* Pointer to variable */
    int varType;                /* Data type of variable */
    int newline;                /*
                                   Flag indicating that VdiPrintf()
                                   should print a newline character
                                */
```

DESCRIPTION

VdiPrintf() displays the message, *pText, followed by the value of the variable, *pVar, on the remote I/O console. If newline is non-zero, then it will also print a newline character following the variable.

The parameter varType identifies the object to which pVar points. It must have one of the following values:

```
VDI_NONE
VDI_CHAR
VDI_SHORT
VDI_INT
VDI_LONG
VDI_FLOAT
VDI_DOUBLE
VDI_U_SHORT
VDI_U_INT
VDI_U_LONG
VDI_HEX_SHORT
VDI_HEX_INT
VDI_HEX_LONG
VDI_HEX_U_SHORT
VDI_HEX_U_INT
VDI_HEX_U_LONG
VDI_STRING
```

When the type is VDI_STRING or VDI_NONE, VdiPrintf() just prints the string and ignores pVar.

VdiPrintf() displays only the first VDI_IO_TEXT_MAX characters of the identification string.

VdiPrintf() may be called only from a DSP program.

RETURN VALUE

None.

SEE ALSO

VdiScanf()

NAME

VdiQueueDisplay -- Display the contents of a queue

SYNOPSIS

```
void VdiQueueDisplay(pQueue)
    VdiQueue *pQueue;          /* Pointer to queue */
```

DESCRIPTION

VdiQueueDisplay() displays the contents of the queue and its header.

It shows only the first 6 words of long packets waiting in the queue.

RETURN VALUE

None.

SEE ALSO

VdiQueuePut()
VdiQueueGet()

NAME

VdiQueueGet -- Get a packet from a queue

SYNOPSIS

```
#include "vdiQueue.h"

int VdiQueueGet(pDSP, pQueue, pTag, pNWords, pData, room, nonBlocking)
    VdiDSP *pDSP;               /* Pointer to DSP information */
    VdiQueue *pQueue;           /* Pointer to queue */
    unsigned int *pTag;         /* Pointer to variable to receive tag */
    unsigned int *pNWords;      /* Pointer to variable to receive size */
    void *pData;                /* Pointer to buffer to receive data */
    unsigned int room;          /* Length of buffer in words */
    int nonBlocking;            /*
                                   Flag indicating that VdiQueueGet()
                                   should not wait if queue is empty
                                */
```

DESCRIPTION

VdiQueueGet() returns the packet at the head of the queue. If the queue is empty, it waits until a packet is available, unless nonBlocking is non-zero, in which case it returns VDI_ERROR_QUEUE_EMPTY instead.

The packet tag is returned in *pTag, the length in *pNWords (in words), and the data in *pData.

No more than room words will be written into *pData; if the packet length exceeds room, only the first room words of the packet data will be written into the buffer (*pNWords will still be set equal to the actual number of words in the packet).

RETURN VALUE

0 indicates success. VDI_ERROR_QUEUE_EMPTY indicates that the queue is empty (non-blocking mode only).

SEE ALSO

VdiQueuePut()

NAME

VdiQueueInit -- Initialize a queue

SYNOPSIS

```
#include "vdiQueue.h"

int VdiQueueInit(pDSP, pQueue, length)
    VdiDSP *pDSP;                /* Pointer to DSP information */
    VdiQueue *pQueue;            /* Pointer to queue */
    unsigned int length;         /* Total length of queue with header */
```

DESCRIPTION

VdiQueueInit() formats the block of memory to which *_pQueue_ points into an empty queue. The capacity is set such that the queue header and contents occupy _length_ _words_.

RETURN VALUE

0 indicates success.

SEE ALSO

VdiQueuePut()
VdiQueueGet()

NAME

VdiQueuePut -- Put a packet into a queue

SYNOPSIS

```
#include "vdiQueue.h"

int VdiQueuePut(pDSP, pQueue, tag, nWords, pData, nonBlocking)
    VdiDSP *pDSP;                 /* Pointer to DSP information */
    VdiQueue *pQueue;             /* Pointer to queue */
    unsigned int tag;             /* Packet tag */
    unsigned int nWords;          /* Packet length */
    void *pData;                  /* Pointer to buffer with packet data */
    int nonBlocking;              /*
                                     Flag indicating that VdiQueuePut()
                                     should not wait if queue does not
                                     have room
                                  */
```

DESCRIPTION

VdiQueuePut() adds a packet to the end of the queue. It waits until the queue contains enough room to hold the new packet, unless nonBlocking is non-zero, in which case it returns VDI ERROR QUEUE NO ROOM instead.

The packet must be small enough to fit entirely into the queue. Thus nWords must not exceed pQueue->capacity - VDI PACKET HEADER SIZE; if it does, VdiQueuePut() will never return.

RETURN VALUE

0 indicates success. VDI_ERROR_QUEUE_NO_ROOM indicates that the queue does not have enough space to hold the new packet (non blocking mode only).

SEE ALSO

VdiQueueGet()

NAME
    VdiScanf -- Receive input from the remote I/O console

SYNOPSIS

```
#include "vdiIO.h"

void VdiScanf(pText, pVar, varType)
    char *pText;              /* Prompt string */
    void *pVar;               /* Pointer to variable */
    int varType;              /* Data type of variable */
```

DESCRIPTION

VdiScanf() first displays the prompt, *pText, on the remote I/O console. It then waits for the user to enter a value of type varType at the console, and it returns that value in *pVar.

The parameter varType indicates the type of variable expected and must have one of the following values:

```
VDI_NONE
VDI_CHAR
VDI_SHORT
VDI_INT
VDI_LONG
VDI_FLOAT
VDI_DOUBLE
VDI_U_SHORT
VDI_U_INT
VDI_U_LONG
VDI_HEX_SHORT
VDI_HEX_INT
VDI_HEX_LONG
VDI_HEX_U_SHORT
VDI_HEX_U_INT
VDI_HEX_U_LONG
VDI_STRING
```

When the type is VDI NONE, VdiScanf() just prints the string and does not read a value from the console. When the type is VDI STRING, VdiScanf() truncates the string to VDI IO TEXT MAX characters, including the terminating null. If there is room, the null character will be preceeded by a newline. The buffer to which pVar points must have room for up to VDI IO TEXT MAX characters.

VdiScanf() displays only the first VDI IO TEXT MAX characters of the prompt string.

VdiPrintf() may be called only from a DSP program.

RETURN VALUE
    None.

SEE ALSO

VdiPrintf()

NAME

VdiSemaphoreInit -- Initialize a semaphore

SYNOPSIS

```
#include "vdiSemaphore.h"

void VdiSemaphoreInit(pDSP, pSemaphore, n)
    VdiDSP *pDSP;               /* Pointer to DSP information */
    VdiSemaphore *pSemaphore;   /* Pointer to semaphore */
    VdiSemaphore n;             /* Initial value for semaphore */
```

DESCRIPTION

VdiSemaphoreInit() initializes a region of memory for later use as a semaphore.

Such semaphores are used for synchronizing host processes with DSP programs; they may exist only in the dual port memory.

RETURN VALUE

None.

SEE ALSO

VdiSemaphoreP()
VdiSemaphoreV()

NAME

VdiSemaphoreP -- Take a semaphore

SYNOPSIS

```
#include "vdiSemaphore.h"

int VdiSemaphoreP(pDSP, pSemaphore, n, nonBlocking)
    VdiDSP *pDSP;               /* Pointer to DSP information */
    VdiSemaphore *pSemaphore;   /* Pointer to semaphore */
    VdiSemaphore n;             /* Number of "units" to take */
    int nonBlocking;            /*
                                   Flag indicating that VdiSemaphoreP()
                                   should not wait if semaphore is
                                   unavailable
                                */
```

DESCRIPTION

VdiSemaphoreP() performs the atomic P() operation on the semaphore: it first waits for the semaphore's value to equal or exceed n and then decrements it by n.

However, if nonBlocking is non-zero, VdiSemaphoreP() will not wait: if the semaphore has a value greater than or equal to n, it will decrement it by n and return zero to indicate success; otherwise it will return non-zero to indicate failure.

RETURN VALUE

0 indicates success. Non-zero indicates failure (non-blocking mode only).

SEE ALSO

VdiSemaphoreV()

NAME
VdiSemaphoreV -- Release a semaphore

SYNOPSIS
```
#include "vdiSemaphore.h"

void VdiSemaphore(pDSP, pSemaphore, n)
    VdiDSP *pDSP;                 /* Pointer to DSP information */
    VdiSemaphore *pSemaphore;     /* Pointer to semaphore */
    VdiSemaphore n;               /* Number of "units" to release */
```

DESCRIPTION
VdiSemaphoreV() performs the atomic V() operation on the semaphore: it increments the value of the semaphore by n.

RETURN VALUE
None.

SEE ALSO
VdiSemaphoreP()

NAME

VdiShutdown -- Shutdown the system

SYNOPSIS

```
int VdiShutdown()
```

DESCRIPTION

VdiShutdown() shuts down the entire system: it halts each DSP and deletes every background task started by VdiInit().

RETURN VALUE

0 if successful.

SEE ALSO

VdiInit()

Appendix C

An Application: High Accuracy Edge Measurement

As a first step toward implementing the real-time motion-tracking system, we ported a high accuracy straight-line edge measurement software package to the common platform with Bill Baringer's projection-based image processing board. Wayne Niblack and Dragutin Petkovic originally developed this software at the IBM Almaden Research Center in San Jose. They describe the algorithm in detail in [4].

We only ported the core routines which locate edges. Furthermore, because the projection ASICs are not yet fully operational, we simulate the projections in software. When the ASICs are ready, we will replace the simulation routines with functions that utilize chips.

This appendix briefly describes how the ported software implements the algorithm on the common platform. For details about the algorithm, consult [4]. For details about the software that we ported, consult Wayne or Dragutin. This appendix assumes you are somewhat familiar with the operation of the original software.

C.1 Processing

The program takes as input a grey-scale image and two points which *roughly* mark the ends of an edge in the image. It then computes the *exact* location of the edge: an angle θ and distance ρ (from the origin). In addition, the program accepts several parameters that control the details of the algorithm.

The algorithm computes projections at various angles in a range centered around the angle of the line specified by the input points. It then analyzes the results to determine the actual exact angle of the edge and its location. Ideally, the image processing boards should perform the projections (on the ASICs) and local analyses (on the DSPs) in parallel, while the central host processor should perform the global analysis. Presently, we have only one image processing board, so it must sequentially do the local computing for each angle.

We have distributed the various processing steps such that for each angle, the DSP, lacking a projection ASIC, asks the host to perform the projections since the host has access to the images stored in files. The DSP then computes the gradient of the projection vector, finds the peak, computes the ρ corresponding to that particular angle, and sends the results to the host. The host collects these results and computes the final θ and ρ.

We operate the DSP as a "server". It waits for requests for local computation from the host, executes them, and returns the results. Thus the host maintains control over the system.

C.2 Communication

The DSP and host must exchange several pieces of data. Initially, the host must provide the DSP with the region of interest for the projection (derived from the initial points) and the parameters that control the local analysis. These are sent in a message with tag *TAG_ANALYZE* containing the following structure:

```
typedef struct {
    unsigned int XUpperLeft, YUpperLeft, XRectSize, YRectSize;
    float theta;
    int rhoCentroidK;
    int rhoThresholdPercent;
} AnalysisParms;
```

(Naturally, the host must convert the *float* field *theta* to the DSP's floating point format before sending it.)

Next, when the DSP sends a request to the host for a projection, it sends a message with tag *TAG_SIMULATE_PROJECTION* and no body (because the host already has the information necessary to do the projection). The host responds with the results of the projection in a message with tag *TAG_SIMULATION_RESULTS* containing the following structure:

```
typedef struct {
    unsigned int sums[MAX_POINTS];
    unsigned int counts[MAX_POINTS];
    long rhoOffset;
} SimulationResults;
```

Finally, when the DSP completes a local analysis, it reports its results to the host in a message with tag *TAG_ANALYSIS_RESULTS* containing the structure:

```
typedef struct {
    float maxGradient;
    float rhoOfTheta;
} AnalysisResults;
```

(The host must of course convert the *float* fields from the DSP's floating point format back to the IEEE format after receiving the message.)

C.3 Files

The list below describes the files that comprise the software:

comm.h: Defines the tags and structures for the messages that the DSP and host exchange.

host.o: Contains the linked code for the host.

image.dat: Contains the image that the host reads when simulating projections. For the moment, the software reads only this image file. It consists of 80 lines of text with 79 characters on each line (a mixture of exclamation marks and tildes to yield high contrast). The Y coordinate runs from 0 to 79 from top to bottom and the X coordinate runs from 0 to 78 from left to right.

main.c: Contains the server loop that executes on the DSP.

main.d3bin:	Contains the linked code for the DSP.
Makefile:	Describes how to build the application (both the DSP and host pieces).
measure.c:	Contains the top-level function *measure()* that starts the program on the host.
measure.h:	Contains the definitions common to all the code.
measure.pro:	Defines the parameters that control the analysis; the host reads it at run-time.
param.c:	Contains the code for reading measure.pro.
param.h:	Contains the function prototypes for code in param.c.
project.c:	Contains the code for computing projections in software.
project.h:	Contains the function prototypes for code in project.c.
projrec.c:	Contains the local analysis code that runs on the DSP.
projrec.h:	Contains the function prototypes for code in projrec.c
projutil.c:	Contains utility routines that projrec.c uses.
projutil.h:	Contains the function prototypes for code in projutil.c.
slctmeas.c:	Contains the global analysis code that runs on the host.
slctmeas.h:	Contains the function prototypes for code in slctmeas.c.
slctutil.c:	Contains utility routines that slctmeas.c uses.
slctutil.h:	Contains the function prototypes for code in slctutil.c.

With few exceptions, these file names correspond to those in the original software. Likewise, the names of the functions and variables in the ported code almost exactly correspond to those in the original code.

C.4 Operation

To run the software:

1. Set the parameters in *measure.pro*. The sample file contains typical values as well as comments describing the meaning of each parameter.

2. Initialize the system.

3. Load and start the DSP code with the command:

   ```
   VdiAppLoadAndRun(0,"main.d3bin")
   ```

4. Load and start the host code with the commands:

   ```
   ld < host.o
   measure
   ```

5. Supply the information the host requests. If you respond with the quit command, the host will tell the server on the DSP to terminate (by sending the tag *TAG_END* instead of *TAG_ANALYZE*) and return you to the VxWorks prompt.

Sample output from the program is shown below:

```
====== High-Accuracy Projection-Based Edge Locator ======

Reading system parameters from file 'measure.pro' . . .

Reading image from image.dat . . .

Enter the two approximate endpoints (x1, y1, x2, y2) of the
line you wish to find or Q to quit: 22 42 15 49
Region of interest: (10, 37) to (26, 53)
+++++ Sending analysis request to DSP (theta =  43.00)... done
rhoRectUpperLeft:  32.54, rhoOffset: 0, rounded: 2165798
+++++ Sending analysis request to DSP (theta =  43.25)... done
rhoRectUpperLeft:  32.63, rhoOffset: 0, rounded: 2171565
+++++ Sending analysis request to DSP (theta =  43.50)... done
rhoRectUpperLeft:  32.72, rhoOffset: 0, rounded: 2177293
+++++ Sending analysis request to DSP (theta =  43.75)... done
```

```
rhoRectUpperLeft:  32.80, rhoOffset: 0, rounded: 2182978
+++++ Sending analysis request to DSP (theta =  44.00)... done
rhoRectUpperLeft:  32.89, rhoOffset: 0, rounded: 2188623
+++++ Sending analysis request to DSP (theta =  44.25)... done
rhoRectUpperLeft:  32.98, rhoOffset: 0, rounded: 2194227
+++++ Sending analysis request to DSP (theta =  44.50)... done
rhoRectUpperLeft:  33.06, rhoOffset: 0, rounded: 2199790
+++++ Sending analysis request to DSP (theta =  44.75)... done
rhoRectUpperLeft:  33.15, rhoOffset: 0, rounded: 2205311
+++++ Sending analysis request to DSP (theta =  45.00)... done
rhoRectUpperLeft:  33.23, rhoOffset: 0, rounded: 2210792
+++++ Sending analysis request to DSP (theta =  45.25)... done
rhoRectUpperLeft:  33.31, rhoOffset: 0, rounded: 2216230
+++++ Sending analysis request to DSP (theta =  45.50)... done
rhoRectUpperLeft:  33.39, rhoOffset: 0, rounded: 2221627
+++++ Sending analysis request to DSP (theta =  45.75)... done
rhoRectUpperLeft:  33.48, rhoOffset: 0, rounded: 2226983
+++++ Sending analysis request to DSP (theta =  46.00)... done
rhoRectUpperLeft:  33.56, rhoOffset: 0, rounded: 2232296
+++++ Sending analysis request to DSP (theta =  46.25)... done
rhoRectUpperLeft:  33.64, rhoOffset: 0, rounded: 2237568
+++++ Sending analysis request to DSP (theta =  46.50)... done
rhoRectUpperLeft:  33.72, rhoOffset: 0, rounded: 2242798
+++++ Sending analysis request to DSP (theta =  46.75)... done
rhoRectUpperLeft:  33.80, rhoOffset: 0, rounded: 2247985
+++++ Sending analysis request to DSP (theta =  47.00)... done
rhoRectUpperLeft:  33.88, rhoOffset: 0, rounded: 2253131
 theta   rho   maxGradient
  43.00  45.20   44.4782
  43.25  45.30   38.7500
  43.50  45.44   29.7599
  43.75  45.63   32.1923
  44.00  44.99   39.8571
  44.25  45.00   43.4000
  44.50  44.99   52.0800
  44.75  45.00   76.5882
  45.00  45.00   76.5882
  45.25  45.00   76.5882
  45.50  44.99   65.0999
  45.75   45.00   50.0769
  46.00   45.00   44.8965
  46.25   44.99   41.3333
  46.50   46.37   33.4800
  46.75   46.51   28.3043
  47.00   46.66   36.3913

(First) absolute peak: Rho =  45.00, i =  44.75
rho/theta:  44.75, gradient/weight:  76.5882
rho/theta:  45.00, gradient/weight  76.5882
rho/theta:  45.25, gradient/weight  76.5882

Final rho taken from rho of theta peak.
Final edge parameters: rho =  45.00, i =  45.00
 rho  45.00
XL1 XL2 DeltaX: 64 0 64   YL1 YL2 DeltaY: 0 64 -64
Rho = 45.000000, theta = 45.000000

Enter the two approximate endpoints (x1, y1, x2, y2) of the
line you wish to find or Q to quit: q
```

A parameter in *measure.pro* controls how much information the program displays as it runs. The script above corresponds to a detail level of 2.

Appendix D

Optimizing DSP C Code

The following sections suggest techniques for writing C source code for the *d3cc* compiler to speed up and shorten the object code it produces. Consult [8] for additional suggestions from AT&T.

We learned these "tricks" during our experiences writing the system software. Until AT&T adds an intelligent optimizer to *d3cc*, the use of such techniques makes a significant difference.

D.1 Use ints Wherever Feasible

The DSP32C compiler deals with the data type *int* more efficiently than any other. Being 24-bits wide, it exactly matches the size of the machine's integer registers. To handle *longs*, which are 32-bits wide, the compiler must separately manage the lower and upper 16-bits. To handle *shorts*, which are 16-bits wide, the compiler must often generate extra instructions to mask off the upper 8-bits of registers.

This is the prime reason for the existence of the system data type *VdiInt*. For variables that the host and DSPs must share, it allows each machine to work with the data type it handles most efficiently, provided that 24-bits of precision are sufficient.

D.2 Use the Keyword register

One of *d3cc*'s shortcomings is that its register allocation algorithm *underutilizes* the DSP32C register set. You must explicitly label often used function parameters and local variables with the keyword *register* to instruct *d3cc* to keep them in registers for rapid access.

Appendix E

Porting the Software to Other Hardware Platforms

The main assumption which the system software makes about the underlying hardware is that it consists of a master host processor controlling multiple slave microprocessors through shared memory. It should be easily portable to other hardware platforms that are based on this model of control.

The most significant changes must be made to the semaphore module since the arbitration logic is likely to vary greatly across different hardware designs.

If the new hardware represents C data types in a different manner, or if it follows different *high/low-endian* coventions, some extra changes to the communications and remote input-output code will be necessary. For instance, our implementation shuffles the characters in strings and converts between AT&T's DSP floating point representation and the IEEE scheme.

The memory maps and system parameter files and associated routines will probably require some changes to account for different memory organizations.

Some other changes will be necessary to optimize the software to the new architecture. For instance, our implementation uses 24-bit integers throughout the DSP code because that processor manipulates them more efficiently than any other data type.

Finally, some changes may be necessary to the DSP code file loading routines if the cross-compiler and linker for the new architecture produces files in a different format.

Appendix F

Manufacturers

Contact the companies named below for more information on the commercially-available hardware and software discussed in this report:

- AT&T Microelectronics
  Department 50AL330240
  555 Union Boulevard
  Allentown, PA 18103
  800-372-2447

- Communication Machinery Corporation
  125 Cremona Drive
  Santa Barbara, CA 93117
  805-968-4262

- Heurikon Corporation
  3201 Latham Drive
  Madison, WI 53713

- Wind River Systems, Incorporated
  1316 Sixty-Seventh Street
  Emeryville, CA 94608
  415-428-2623
  UUCP: sun!wrs!inquiries

Bibliography

[1] William B. Baringer, Robert W. Brodersen, Dragutin Petkovic, and Jorge Sanz. *ASICs and Machine Vision Applications of the Parallel Pipeline Projection Engine.* 1988 IEEE VLSI Conference, Monterey, California. November 1988.

[2] Gautam B. Doshi. *Design and Implementation of a Six Axis Robot Controller.* Electronics Research Laboratory, University of California, Berkeley. February 7, 1989.

[3] R. Nigel Horspool. *C Programming in the Berkeley UNIX Environment.* Prentice-Hall Canada, Incorporated. Scarborough, Ontario. ©1986.

[4] Dragutin Petkovic, Wayne Niblack, Myron Flickner. *Projection-Based High Accuracy Measurement of Straight-Line Edges.* Machine Vision and Applications, Volume 1, Number 3. 1988. Pages 183-199.

[5] Abraham Silberschatz and James L. Peterson. *Operating System Concepts, Alternate Edition.* Addison-Wesley Publishing Company. ©1988.

[6] *VxWorks Version 4.0 Reference Manual, Volumes 1 & 2.* Wind River Systems. 1988.

[7] *WE DSP32 and DSP32C C Language Compiler, Library Reference Manual.* AT&T Document Management Organization. Publication #MN88-12DMOS. June 1988.

[8] *WE DSP32 and DSP32C C Language Compiler, User Manual.* AT&T Document Management Organization. Publication #MN88-03DMOS. August 1988.

[9] *WE DSP32 and DSP32C Support Software Library, User Manual.* Publication #MN88-04DMOS. AT&T Document Management Organization. August 1988.

[10] *WE DSP32C Digital Signal Processor, Information Manual.* AT&T Document Management Organization. Publication #MN88-06DMOS. December 1988.