Concordia Cum Vaxen

Porting Harmony to the VAX-11/750

by

Wai Victoria Wong

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 1986.

Author's Declaration

Abstract

Harmony is a message-based multiprocessor multitasking operating system developed at the National Research Council for real-time applications. It has been used for applications in computer graphics on a Motorola 68000-based computer in an Ikonas frame buffer. This document describes how it was ported to a DIGITAL VAX-11/750 minicomputer. Graphics support for the Ikonas frame buffer as an output device is also discussed. As a test, the Waterloo Paint program was modified to run on the VAX version of Harmony. Its performance is evaluated and the prospects for porting Harmony to future VAX architectures are examined.

Acknowledgements

I would like to express my gratitude to all those who helped me in completing this project.

The research has been supervised by Dr. John Beatty and Dr. Kellogg Booth. Financial support was provided by the University of Waterloo and an Ontario Graduate Scholarship.

Dr. Kellogg Booth, Dr. Morven Gentleman, and Dr. Douglas Dyment were the readers for this thesis. Shawn Neely was the student reader. Shawn also worked on a Harmony host-based tool for calculating stack sizes.

Dr. Morven Gentleman, Peter Tanner, Dave Forsey, Ian! Allen, Darlene Stewart, Marc Riese, and Peter Baker supplied information about Harmony, Alan Paeth helped explain the configuration of the Ikonas, and Dave Martindale presented some glimpses of the world of UNIX wizardry. Ken Lalonde, Kim Martin, and the MFCF crew helped me continuously with the hardware and UNIX guts. The Shoshin group supplied references on the VAX architecture.

In preparing this document, Ian! Allen's Erb Street typesetting macros, Steve Hayman's music preprocessor, and Vic Klassen's help with the previewer are appreciated.

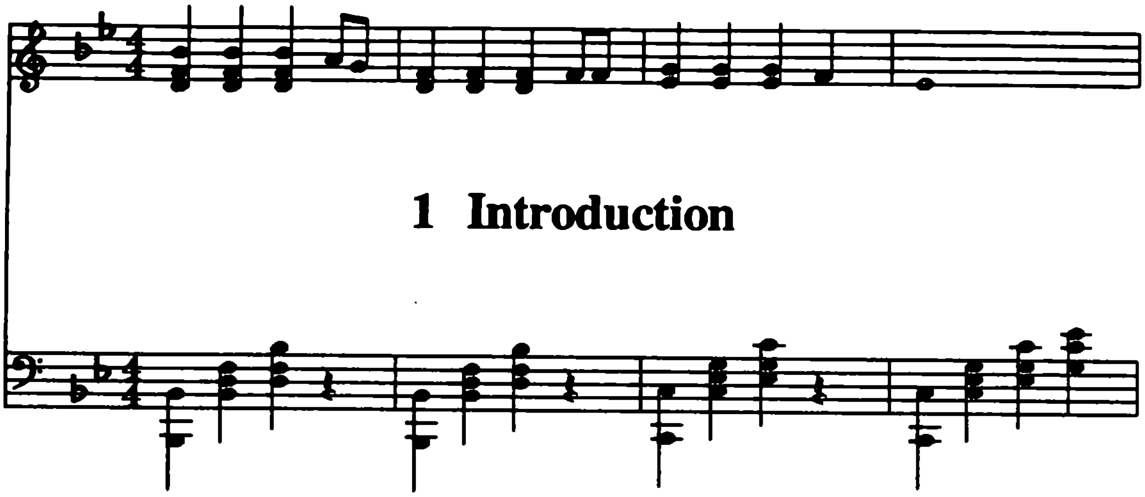
It has been a pleasure to work in the Computer Graphics Lab; the new ideas, friendship, and encouragement I received from the group will be remembered.

Finally, I must thank my family for their constant support from the opposite side of the Earth.

W. Victoria Wong
University of Waterloo
December 1986

Table of Contents

List of Illustrations



1 Introduction

Harmony ‡ is a message-based multiprocessor multitasking operating system developed at the National Research Council for real-time applications. It was originally implemented on the Chorus multiprocessor, a Motorola 68000-based system.

At the University of Waterloo, Harmony was ported by Dave Forsey to a Motorola 68000-based MPC computer in an Ikonas frame buffer. It has been used for Paint programs and robotics simulations. It will also be used for supporting cognitive psychology experiments on user interface design.

The University of Waterloo Computer Graphics Laboratory will have a number of DIGITAL MicroVAX workstations and later more advanced workstations having multiple processors based on the VAX architecture. These will be an ideal test bed for a multiprocessor implementation of Harmony. Harmony has been ported to other 68000 family processors, but there was no implementation for the DIGITAL VAX architecture prior to this project. As a first step in this research, Harmony has been ported to a VAX-11/750 minicomputer to be evaluated. The version of Harmony that runs on the VAX-11/750 will be referred to as VAX-Harmony in this document. It is ported with reference to the version of Harmony running on the MPC, which will be referred to as MPC-Harmony for clarity. The source development of VAX-Harmony is done on a VAX 8600 under a UNIX † environment. The version of source code used is Harmony Release 1.0.

The Waterloo Paint program, the first application program to run under MPC-Harmony, has been modified to run under VAX-Harmony as a test. It also serves as a comparison with the MPC version in terms of speed and performance.

This first chapter describes the Harmony operating system, the VAX-11/750 computer, the Ikonas frame buffer, and the Paint program. Chapter 2 concentrates on how Harmony was ported to the VAX-11/750. Chapter 3 describes graphics support for the Ikonas and modifications to the Paint program. The final chapter summarizes Harmony's performance on the VAX, limitations on the current implementation, and the prospects for porting Harmony to MicroVAX-based workstations.

‡ Mark reserved for the exclusive use of Her Majesty the Queen in right of Canada by Canadian Patents and Development Ltd./Société canadienne des brevets et d'exploitation Ltée.

† UNIX is a Trademark of AT&T.

1.1 Harmony

Harmony is a multiprocessor, multitasking operating system for real-time control. It was developed at the National Research Council by a team led by Dr. Morven Gentleman. A detailed description of Harmony can be found in the NRC technical report "Using the Harmony Operating System" [Gentleman85].

Harmony is an open and portable system, designed to be used on many different hardware configurations and to allow the integration of new peripherals. It is written in the C programming language and a small amount of assembler code. Application programs can be written in either C or FORTRAN, but Harmony does not preclude the use of other languages such as Pascal if the host system supports linking across languages.

Harmony is a system of tightly-coupled multiprocessors. It devotes all of the resources of the machine to running a single multitask application and is bound in with the program at link time. Harmony does not directly support program development or multiuser time sharing. A host computer is used for source development and downloading of executable images into target processors.

1.1.1 Tasks

A Harmony program is organized into many tasks. A *task*, or a *process*, is an instance of a program or subroutine; it executes sequentially and in parallel with other tasks. Each task is independent of other tasks, but tasks can communicate and synchronize with each other through message passing.

Tasks are created and destroyed dynamically. Harmony is a multiprocessor system, and the processor on which a task runs is specified in the *task template* describing it. A task competes for scheduling only with other tasks running on the same processor. As a real-time system, Harmony uses "natural break" priority scheduling to control task execution on each processor: a task is assigned a priority level; each level has a separate FIFO queue for ready tasks. Execution of a task will be preempted when a task of higher priority becomes ready, otherwise a task will execute sequentially until it blocks waiting for external events or messages from other tasks. Urgent tasks can be assigned a high priority to ensure a short response time.

A task template defines parameters for tasks created from it:

• GLOBAL_INDEX	A positive integer (maximum 231 in VAX-Harmony) identifying the template, which must be unique over all templates on all processors.
• ROOT	The function that the task will execute. The function has no parameters or return value. If the function returns, the task is destroyed.
• STACKSIZE	Size of the run-time stack for the task, which may be calculated by the *bound* tool.
• PRIORITY	The priority level of the task. Smaller numbers imply higher priority.
• LOCAL_TASK_MANAGER	Supplied by the system, this identifies the local task manager that creates and destroys the task, and hence effectively determines on which processor the task runs.

Tasks are managed by create and destroy primitives:

```
id = _Create( global_index );

_Destroy( id );

_Suicide();
```

_Create returns a unique id for identifying the task. Destroying a task includes returning all its resources to the system, closing all its connections to servers, and destroying all its descendants.

A task can obtain its own id and the id of the task that created it by using the following primitives:

```
_My_id();

_Father_id();
```

The global indices of 1 to 3 are reserved. Index 1 is for the first *user task*, which normally has the root function main(). Index 2 is for the *directory task*, which handles the translation of symbolic names into task ids. Index 3 is for the *gossip task*, which provides a general logging and error reporting mechanism for other tasks. Harmony will create one instance of each of these tasks on processor 0 when starting up.

1.1.2 Communication and Synchronization

Communication and synchronization between tasks are accomplished by message passing primitives. The choice of message passing versus other concurrency control techniques is discussed in another paper [Gentleman81]. All message passing is implemented with an interprocessor interrupt.

A message is a variable length contiguous block of storage, the first two bytes of which are a short unsigned integer specifying the length. Messages are queued in FIFO order.

The message passing primitives are as follows:

```
id = _Send( request, reply, id );
id = _Receive( request, id );
id = _Try_receive( request, id );
id = _Reply( reply, id );
```

A task wanting to send a message will set up the message in memory space pointed to by the request argument, allocate space pointed to by the reply argument for receiving a message in reply, and call the _Send primitive specifying the id of the correspondent task. A task wanting to receive a message must set up space pointed to by the request argument into which the contents of the sender's message will be copied. It can either specify the id of a particular task or it may receive from any task by using an id of zero. The _Receive primitive will block the task until a message is received, but the _Try_receive primitive always returns immediately. Once a message is received, a task can set up a message pointed to by the reply argument and call _Reply with the id of the sender task whenever it chooses.

Since tasks may be destroyed, these message passing primitives may fail. If a call is successful, the id of the correspondent task is returned. Otherwise, the return value will be zero. If a task calls _Try_receive but no messages are pending, the call returns a zero.

The _Send and _Receive primitives are blocking, while the other two are not. _Send and _Reply both transfer data to another task – by reversing the usual roles of these primitives, a non-blocking communication is possible in any single direction between two tasks. This feature is used in the *courier* abstraction to obtain bidirectional non-blocking communication [Gentleman81].

1.1.3 Memory

Current Harmony implementations use a single, contiguous, shared address space. Due to the real-time constraint, no algorithms with unbounded execution time can be used, and resources such as stacks are preallocated.

Memory resources are handled with storage pool management primitives:

```
pointer = _Getvec( size );
_Freevec( pointer );
_Trimvec( pointer, size );
size = _Sizeof( pointer );
```

and with a special function to check for stack overflow:

```
boolean = _Stackoverflow();
```

Memory is allocated using a first fit algorithm; the pool is searched sequentially from the start, and interrupt windows are provided within the search. A function is provided for optimizing the memory pool search time:

```
_Tune_Getvec( size );
```

When called, this function modifies the searching algorithm to skip the blocks at the beginning of the pool, which are allocated for task creation or which are known to be too small due to the statistical effect of *grading* that occurs with first fit techniques.

1.1.4 Stream I/O

For I/O devices, a stream I/O library and await-interrupt primitive are provided. A stream is an infinite sequence of bytes in which a current position can be defined, much as for a file in UNIX. A stream, like all Harmony connections, is identified by a user connection block (ucb). The stream I/O primitives include:

```
ucb = _Open( pathname, mode );
_Close( ucb );
previous_ucb = _Selectinput( ucb );
previous_ucb = _Selectoutput( ucb );
byte = _Get();
byte = _Put( byte );
_Flush();
_Unget();
_Seek( ucb, positive, relative );
```

Data in a stream are buffered so that the _Get and _Put primitives usually reference the respective local buffers. The output buffer is sent to the server to be output when it is full; if desired, the _Flush primitive can be used to force output without waiting for a full buffer. The _Unget primitive returns the last byte read from the input stream, so it will be read again by the next _Get. However, this only works for the most recently read byte of the stream.

On top of the stream I/O model, some high level I/O functions are provided. They are similar to the standard I/O functions in C:

```
number = _Getnum();
_Putdec( number );
_Puthex( number );
_Putstr( string );
```

which can be combined by using

```
_Printf( format, item1, item2, ... );
```

There are busywait counterparts to each of the above functions (e.g., _BWGetnum()). The busywait functions use polling instead of interrupt-driven I/O. They are useful when Harmony stream I/O cannot be relied upon, such as when debugging the kernel of the system. Busywait I/O is seldom employed by user programs.

1.1.5 Interrupts

A Harmony program can disable or enable interrupts with:

```
_Disable();
_Enable();
```

It can also wait for an external interrupt event to happen by using

```
_Await_interrupt( interrupt_id, reply_message );
```

At most one task can be waiting for any specific interrupt. A task can only be waiting for one interrupt. It must run on the processor physically connected to the interrupt and at the priority level corresponding to the hardware level of the interrupt. Volatile data received during the interrupt are passed to the task in the reply message. With these primitives and second level interrupt handlers, the user can implement I/O devices that do not fit the stream I/O model.

A hardware interrupt will invoke a first level interrupt handler, which saves the registers on the stack of the interrupted task and transfers to the corresponding second level interrupt handler. The second level handler is written in assembler and no stack is readily available for its use. The second level handler must save the stack pointer of the interrupted task, demultiplex the hardware interrupt if necessary, and clear the interrupt while saving any volatile data. The second level interrupt handler will then either activate the task waiting for the interrupt or, if the interrupt is spurious, reactivate the interrupted task. Any interrupts caused by noise or configuration error and any interrupts that occur without a waiting task are considered to be spurious. Spurious interrupts and most exceptions are handled by logging a message with the gossip task and invoking the debugger.

The distinction between a first level interrupt handler and a second level interrupt handler is one of convention. A first level handler is generic, whereas a second level handler usually contains instructions specific to the interrupt or device being handled. A second level handler is written like user code, but it is executed during the interrupt servicing to permit processing that cannot wait for normal task dispatching.

After its time-critical processing, the second level interrupt handler adds the task that has an _Await_interrupt pending for the device to the ready queue. It then exits by dispatching the highest priority task. Under the preemptive priority scheduling scheme, the interrupt, and hence the task waiting for it, must have higher priority than the interrupted task; all lower priority interrupts would have been held off. Therefore the newly readied task, if any, will be the one to be dispatched.

The VAX hardware, like that of many other machines, implements preemptive priority dispatching. Levels of processor priority are defined, and at any time the processor is running at a certain priority. The only interrupts that are allowed to affect the processor are those at a higher priority than the current processor level – all others are held off by the hardware until the processor priority drops to a level where they will be accepted. If the hardware has allowed an interrupt to occur, then the processor must have been executing at a lower priority level, therefore the handler for the incoming interrupt can preempt whatever was running. Disabling interrupts on such a processor is often done by temporarily raising the priority level of the processor above the level that incoming interrupts could produce.

Harmony software has exactly the same abstraction. When a task is running at a certain priority, another task at the same or lower level of priority becoming ready is immaterial – the original task keeps running. However, if a task of higher priority becomes ready, it preempts the original task, and runs instead. The fact that the original task was of a certain priority guarantees that there were no ready tasks of higher priority, so a task becoming ready need only have its priority compared with that of the active task – the whole ready queue does not need to be searched.

A proper port of Harmony to a new machine integrates these two priority systems so that they are, in fact, one and the same. This implies that when a task is running, the hardware priority at which it runs is such as to hold off interrupts that would activate tasks that run at the same or lower software priority. Consequently, the very fact that the interrupt happened ensures that the ready queue for the waiting task must be empty and the proper task to dispatch is the one that the interrupt readies.

To allow the user to perform I/O instructions directly and have user code for interrupt handlers, Harmony and the application program always run in the most privileged mode available on the hardware.

1.1.6 Servers

Servers are used to implement "smart peripherals" and resource abstractions. A Harmony *server* is a task that owns and manages a *resource*. It is analogous to a library function. *Client tasks* request the use of a resource by sending messages to its server. The message passing code requires knowledge of a task's id; this information is maintained by a directory task. The primitives used by a server for reporting and managing its connections are as follows:

```
_Report_for_service( name, message_type );
table = _Alloc_connection_table( init_num_entries, scb_size );
scb = _Get_connection( table, client, new_connection );
scb = _Lookup_connection( table, client, connection );
connection = _Free_connection( table, connection );
```

A server task must report to the directory task and must be prepared to handle open and close messages. The other types of messages it handles depend on the I/O model and the resource it controls.

1.1.7 Programs

A simple Harmony program is shown in Figure 1.1. The executable image for each processor is linked and downloaded separately. For each processor image, the user must identify the processor by _Pnumber, supply the task templates for all tasks that can be created on that processor, and provide the second level interrupt handlers, each specified by an interrupt id and the handler code to be executed. The example program in Figure 1.1 demonstrates how tasks are created and how they communicate with each other.

```
#define CHILD     10

extern main();
extern _Directory();
extern _Gossip();
extern Child();

extern _Ptm_int();
extern _Serial_int();

uint_32 _Pnumber = 0;

struct TASK_TEMPLATE _Template_list[] =
  {
    { MAIN, main, 400, 7, 0 },
    { DIRECTORY, _Directory, 436, 7, 0 },
    { GOSSIP, _Gossip, 308, 5, 0 },
    { CHILD, Child, 300, 6, 0 },
    { 0, 0, 0, 0, 0 }
  };

struct INT_PAIR _Interrupt_list[] =
  {
    { 4, _Ptm_int },
    { 5, _Serial_int },
    { 0, 0 }
  };

main()
  {
    uint_32 child;
    struct STD_RQST request;
    struct STD_RPLY reply;

    child = _Create( CHILD );

    for(;;)
      {
        request.MSG_SIZE = sizeof( request );
        reply.MSG_SIZE = sizeof( reply );
        _Send( (char *)&request, (char *)&reply, child );
      }
  };

Child()
  {
    uint_32 counter, requestor;
    struct STD_RQST request;
    struct STD_RPLY reply;

    counter = 0;

    for(;;)
      {
        request.MSG_SIZE = sizeof( request );
        requestor = _Receive( (char *)&request, 0 );

        reply.MSG_SIZE = sizeof( reply );
        reply.RESULT = counter++;
        _Reply( (char *)&reply, requestor );
      }
  };
```

Figure 1.1 Sample Harmony Program

1.2 VAX-11/750

The DIGITAL VAX-11/750 is a 32-bit minicomputer. The target machine used in this project has three million bytes of physical memory, a floating point accelerator, and two UNIBUS adapters. A DMZ32 multiplexor and an Adage RDS-3000 frame buffer are on one UNIBUS.

1.2.1 Architecture Summary

The basic features of the VAX-11/750 architecture are discussed below [Digital81, Digital82].

The VAX-11/750 supports six primary data types:

- Bits
- Integers – 8, 16, 32 or 64 bits
- Floating point reals – 32, 64 (two precisions), or 128 bits
- Packed decimal strings – 0 to 16 bytes, two digits per byte
- Character Strings – 0 to 65535 bytes
- Numeric Strings – 0 to 31 bytes

and it also supports a queue data type which is a circular doubly-linked list; each queue entry is accompanied by two longword pointers.

There are instructions for manipulating all the data types. A floating point accelerator (an independent coprocessor) can be added to execute the floating point instructions for improvement in speed.

In data representation, the lower order byte precedes higher order bytes.

There are sixteen 32-bit general registers (R0 to R15), of which the last four have special significance:

- R15 or PC – Program Counter, contains the address of the next instruction to be executed.
- R14 or SP – Stack Pointer, contains the address of the top of a stack.
- R13 or FP – Frame Pointer, contains the address of the base of the function call frame data structure on the stack.
- R12 or AP – Argument Pointer, contains the address of the base of the argument list data structure on the stack.

The registers R12 to R14 may be used for general purposes if they are not required for stack management.

The registers R0 to R5 are used in character string and polynomial evaluation instructions, and registers R0 and R1 are used to hold function return values; precaution should be taken when using these registers.

The VAX family processors allow 21 addressing modes of nine basic types:

- Literal
- Register
- Register Deferred
- Autoincrement
- Autodecrement
- Autoincrement Deferred
- Absolute
- Displacement
- Displacement Deferred

The last seven types can be indexed by a value in a general register.

A processor register called the Processor Status Longword (PSL) determines the execution state of the processor at any time. The lower-order 16 bits of the PSL are the Processor Status Word (PSW). It contains unprivileged information and is available and controllable by any program. The higher-order bits provide privileged control of the system, including access mode, instruction set information, interrupt priority level, and the selection of an interrupt stack.

The processor has four hierarchically ordered access modes: Kernel, Executive, Supervisor, and User. There is a separate stack pointer for each mode. Memory is protected by specifying read or write permissions for each mode, which will also be inherited by the more privileged modes. Privileged registers can only be accessed in kernel mode.

Harmony always runs in kernel mode, which has the maximum privileges, so that it can access all the processor status for implementing preemptive priority dispatching.

Exception and interrupt handling are vectored. The vectors are stored in a structure called the System Control Block, the dedicated pages of physical memory locations shown in Figure 1.2. Each vector contains the virtual address of the interrupt service routine (in Harmony, the first level interrupt handler), which must be aligned on a longword boundary. The lowest two bits indicate whether the interrupt stack or the user control store containing customized microcode are used.

There are 31 interrupt priority levels. The current level can be changed through the IPL privileged register.

1.2.2 Memory Management

The VAX (Virtual Address eXtension) family computers have a memory management system. Its operation is discussed below.

The physical addresses of the VAX-11/750 are 24 bits long. The *physical address space* is divided into the *memory space* and *I/O space*. The target machine used in this project has three million bytes installed memory and two UNIBUS subsystems. The corresponding physical memory organization is shown in Figure 1.3.

The region from F20000 to F40000 is divided into sixteen 8K-byte adapter register address spaces; this arrangement resembles that for the NEXUSs of the VAX-11/780. The first address space (number 0) is for the memory controller. It contains three Control and Status Registers and four 256-byte read-only memories for bootstrapping from devices. UNIBUS 0 and 1 adapters use address space numbers 8 and 9 respectively.

The *virtual address space* uses 32-bit addresses. The basic unit for relocation and protection is a 512-byte page. The entire virtual address space is divided into four regions as shown in Figure 1.4. Standard conventions for using the four regions are used by operating systems such as VMS. The P0 region is designed for program image and data. The P1 region for stack and control information is maintained by the operating system for each process. The system region is used by the operating system. The last region is not used.

The memory management unit translates virtual addresses to physical addresses using information in the page table for each region. A page table is a vector of page table entries, which are described in Figure 1.5. The bits 0 to 20 in the page table entry hold the page frame number. Because the VAX-11/750 has 24-bit physical addresses, only the lower 15 bits determine the page frame number. The unused bits are zero.

Vector(hex)	Interrupt or Exception	IPL
00	(Reserved)	
04	Machine Check exception	1F
08	Kernel Stack Not Valid exception	1F
0C	Power Fail interrupt	1E
10	Reserved/Privileged Instruction exception	
14	Customer Reserved Instruction exception	
18	Reserved Operand exception	
1C	Reserved Addressing Mode exception	
20	Access Control Violation exception	
24	Translation Not Valid exception	
28	Trace Pending exception	
2C	Breakpoint Instruction exception	
30	Compatibility exception	
34	Arithmetic exception	
38-3C	(Reserved)	
40	CHMK exception	
44	CHME exception	
48	CHMS exception	
4C	CHMU exception	
50	(Reserved)	
54	Corrected Memory Read Data interrupt	1A
58-5C	(Reserved)	
60	Memory Write Timeout interrupt	1D
64-80	(Reserved)	
84-BC	Software interrupts	1-F
C0	Interval Timer interrupt	18
C4-DC	(Reserved)	
E0-EC	(Unused)	
F0	Console Storage Device Receive interrupt	17
F4	Console Storage Device Transmit interrupt	17
F8	Console Terminal Receive interrupt	14
FC	Console Terminal Transmit interrupt	14
100-1FC	(Reserved)	
200-3FC	UNIBUS 0 device interrupt vectors	14-17
400-5FC	UNIBUS 1 device interrupt vectors	14-17

Figure 1.2 System Control Block

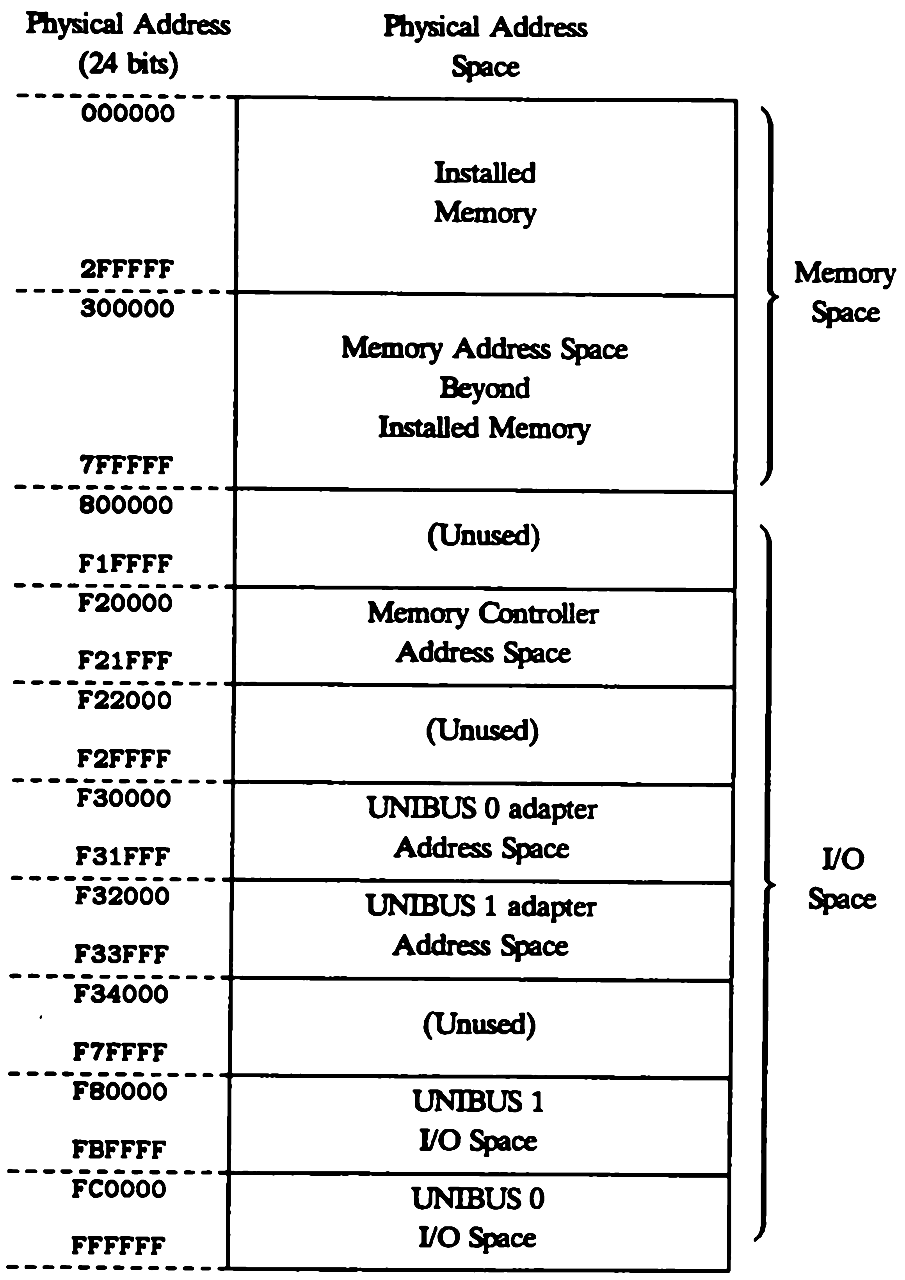


Figure 1.3 Physical Address Space

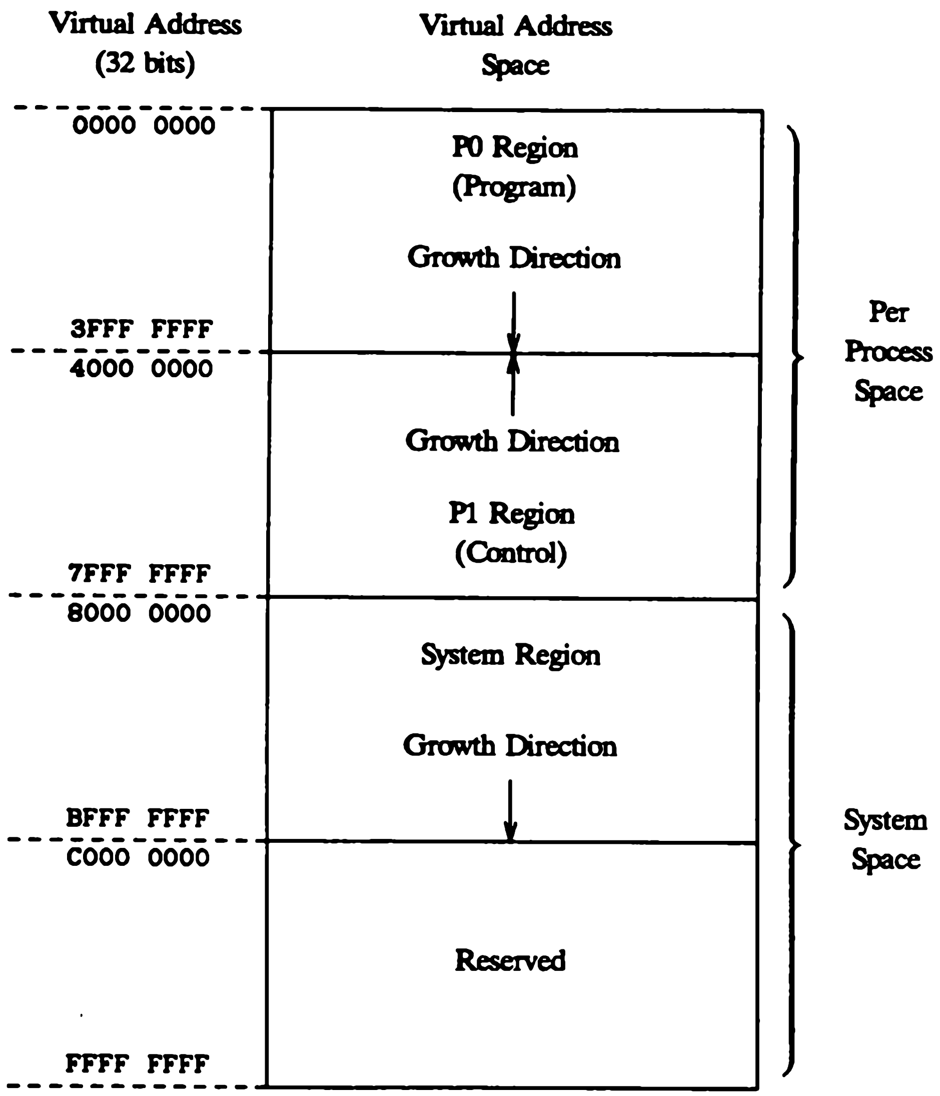


Figure 1.4 Virtual Address Space

A page table is specified by a base register and a length register for its region; these registers hold the base address of the table and the number of entries (or the number of entries unused for P1) respectively. The memory management unit will check the access privileges and map virtual pages to the physical pages. If the page is marked invalid, a page fault exception will occur and the processor may retrieve the page from other storage devices if the operating system supports this feature.

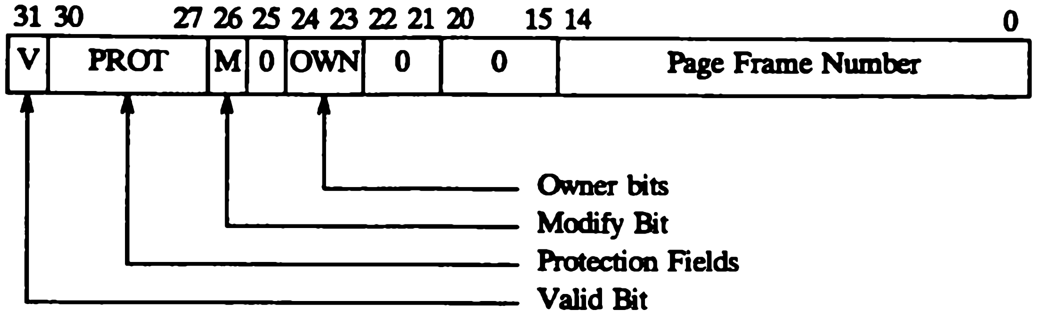


Figure 1.5 Page Table Entry

Each process has its own page tables for the P0 and P1 regions. All page tables for the process space must be stored in the system region. The system region is not context switched. The system page table must reside in main memory.

The memory management unit is enabled using the MAPEN privileged register. When it is disabled, all addresses are treated as physical, with the highest 8 bits ignored. The unit also uses a 4K-byte cache, an 8-byte prefetch instruction buffer, and a 512-entry address translation buffer for speed.

1.2.3 UNIBUS Subsystem

The VAX-11/750 can have one or two UNIBUS subsystems for peripheral devices. The UNIBUS is a communication path that links I/O devices to the UNIBUS adapter, which is part of the CPU. Conceptually, the UNIBUS is designed around memory elements with ascending addresses starting from zero, while registers storing data or device status information have addresses in the highest 8K bytes of the addressing space. Communication between any two devices on the bus is in a master/slave relationship. The UNIBUS uses 18-bit addresses and 16-bit data.

The UNIBUS adapter allows the processor to access 16-bit control and status registers on the UNIBUS. It also allows devices to perform DMA transfers to the VAX-11/750 memory and to interrupt the processor. It performs priority level arbitration among the devices according to the four priority levels BR4 to BR7, which are equivalent to levels 14 to 17 on the processor. There are three buffered data paths and one direct data path. Because UNIBUS data are 16 bits and the internal data path of the processor is 32 bits, buffered data paths act as a very small cache and allow only one memory transfer for every two UNIBUS transfers. With the use of buffered data paths, transfer rates can be optimized from 1M bytes per second to 1.5M bytes per second.

1.3 Paint

1.3.1 Ikonas

The VAX-11/750 in the Computer Graphics Laboratory is connected to an Adage/Ikonas RDS-3000 frame buffer through the UNIBUS. The Ikonas is a high performance frame buffer consisting of image memory, display hardware, auxiliary processors, and interface boards. The architecture of the Ikonas system is shown in Figure 1.6.

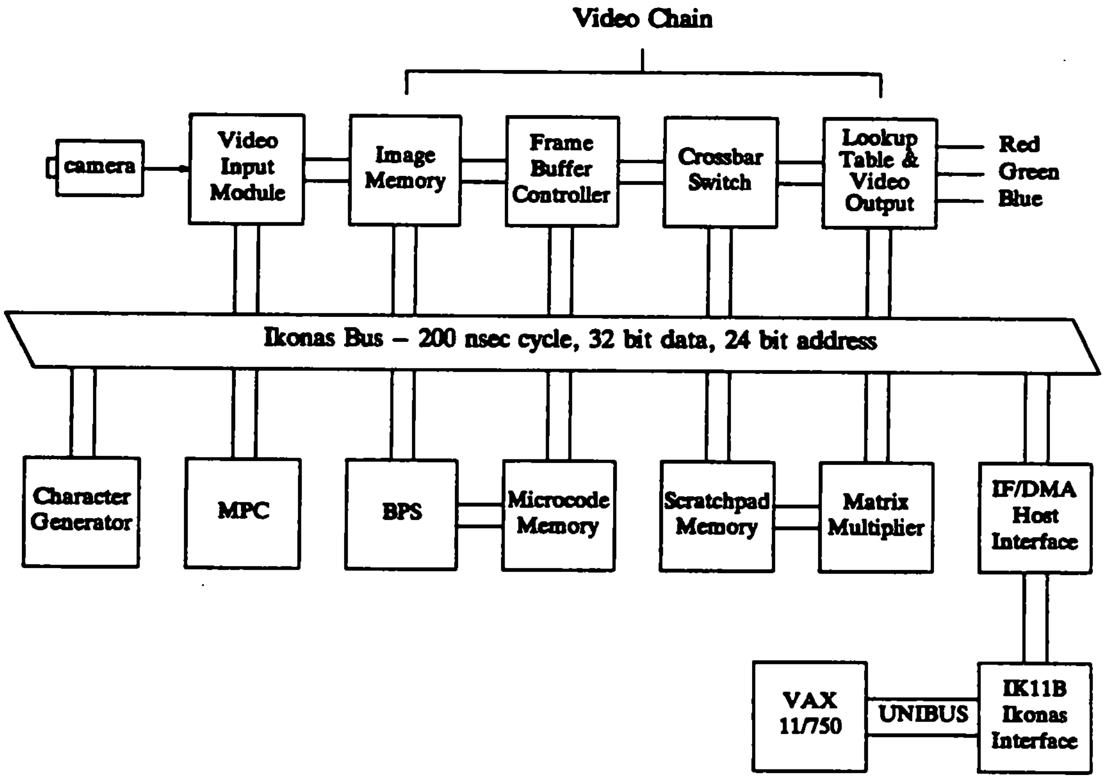


Figure 1.6 Ikonas System Architecture

Each hardware module is addressable on the Ikonas bus. Some parts of the hardware are discussed briefly in this section. Fuller details can be found in the hardware manuals [Adage82]. The arrangement of the hardware in the Ikonas address space is shown in Figure 1.7.

The image memory consists of GM 256 type boards, providing a 1024x1024x32bit memory. In low resolution, only 512x512 pixels are used for display, the rest are used as off-screen memory. There are eight write masks to allow bit fields in a pixel to be selected for updating. Eight shade registers are used in mask mode writing, where 32 consecutive pixels can be written with the same value. Protection by write mask applies to pixels instead of bits in this case.

The video chain is the collection of hardware that processes the pixel data before they are displayed on the monitor. The frame buffer controller controls the cursor and the order and speed at which the pixels are displayed. It also adds the video sync and blanking signals to the RGB output. The crossbar switch allows the rearrangement of bits from input to output channels. The color lookup table maps each pixel value with one of four color maps and sends the result to the digital-to-analog converters to produce video output.

The IF/DMA host interface supports three transfer modes:

- Word mode – two host 16-bit words correspond to one Ikonas 32-bit word.
- Half-word mode – one host 16-bit word corresponds to the lower half of an Ikonas word, the higher half is zero-extended.
- Byte mode – each 8-bit byte of host data corresponds to one 32-bit Ikonas word. The byte number within the word is specified.

The IK11B board provides three classes of interrupts: I/O completion, video field, and processor interrupt. These may be enabled separately and each has its own interrupt vector on the UNIBUS.

The BPS (Bipolar Graphics Processor) is a microprogrammable bit-slice microprocessor that can execute 5 million instructions per second.

The MPC (Multifunction Peripheral Controller) is a Motorola 68000-based processor. Harmony has been ported to run on the MPC by Dave Forsey [Forsey85].

1.3.2 Paint

Typically, paint programs read from an input device such as a mouse or tablet and trace a cursor on an output device (usually a frame buffer). When a button is depressed on the input device, the program modifies image pixels according to a brush pattern.

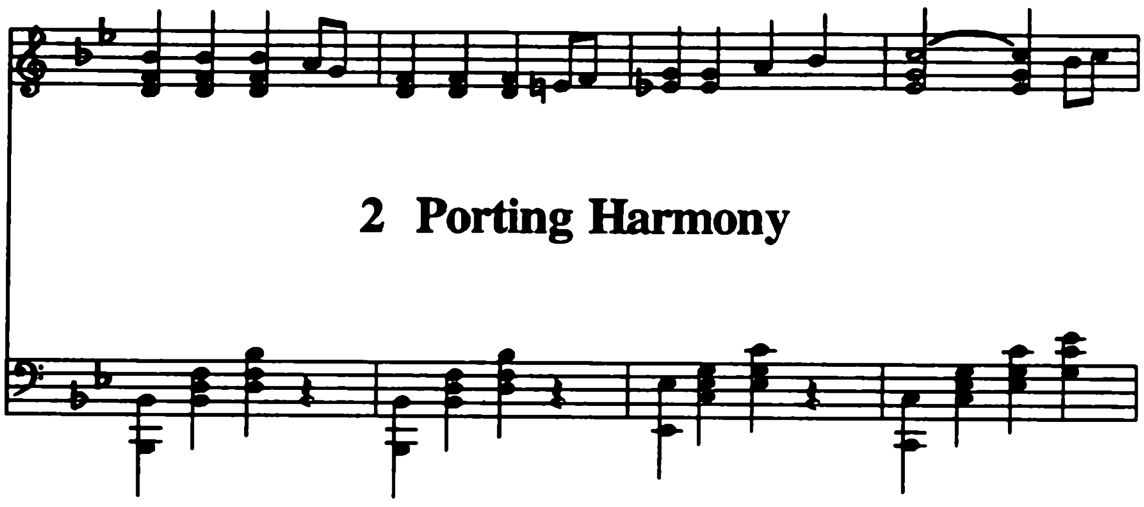
Hex Address	Device	Y$X Address
FF03FF$_{16}$ ... F00000$_{16}$	(Reserved for multiple MPC's)	37700$1777$_8$ ... 35000$0000$_8$
E7FFFF$_{16}$ ... E00000$_{16}$	MPC Data and I/O Space (Lower 16 bits only)	34777$1777$_8$ ... 34000$0000$_8$
DF03FF$_{16}$ ... C30000$_{16}$	(Reserved)	33700$1777$_8$ ... 30300$0000$_8$
C203FF$_{16}$ ... C20000$_{16}$	Crossbar Switch	30200$1777$_8$ ... 30200$0000$_8$
C103FF$_{16}$ ... C10000$_{16}$	Video Input Module	30100$1777$_8$ 30100$0000$_8$
C003FF$_{16}$ ... C00000$_{16}$	Frame Buffer Controller	30000$1777$_8$ ... 30000$0000$_8$
BFFFFF$_{16}$ ... 870000$_{16}$	(Reserved)	27777$1777$_8$ ... 20700$0000$_8$
86FFFF$_{16}$ ... 860000$_{16}$	Character Generator	20677$1777$_8$ ... 20600$0000$_8$
85FFFF$_{16}$ ... 850000$_{16}$	Bipolar Graphics Processor (BPS)	20577$1777$_8$ ... 20500$0000$_8$
84FFFF$_{16}$ ... 840000$_{16}$	Matrix Multiplier (MA1024)	20477$1777$_8$ ... 20400$0000$_8$
83FFFF$_{16}$ ... 830000$_{16}$	Color Lookup Tables (LUVO)	20377$1777$_8$ ... 20300$0000$_8$
82FFFF$_{16}$ ... 820000$_{16}$	Scratchpad Memory (SR8s)	20277$1777$_8$ ... 20200$0000$_8$
81FFFF$_{16}$ ... 810000$_{16}$	(Reserved)	20177$1777$_8$ ... 20100$0000$_8$
80FFFF$_{16}$ ... 800000$_{16}$	Microcode Store (MCM4s)	20077$1777$_8$ ... 20000$0000$_8$
7FFFFF$_{16}$ ... 000000$_{16}$	Frame Buffer Memory	17777$1777$_8$ ... 00000$0000$_8$

Y$X is the notation of a 24-bit address
where the upper 14 bits are Y and the lower 10 bits are X.

Figure 1.7 Ikonas Memory Map

Waterloo Paint was originally written by Eugene Fiume in 1981 and was later modified by Rick Beach and Darlene Plebon [Beach82]. It used the Thoth operating system [Cheriton79], a predecessor of Harmony, which is also a message-based real-time multitasking operating system.

Paint originally ran on a Honeywell Level 6 minicomputer. When Harmony was ported to the MPC, Paint was also modified to run under MPC-Harmony [Forsey85]. It has since been tested by local artists and students and improved to provide a better user interface and real-time response. To test the correctness of Harmony on the VAX-11/750, and to develop graphics library support for future applications, Paint was ported to run under VAX-Harmony as part of this project.



2 Porting Harmony

2.1 The Kernel

Harmony was already running on the MPC and the software maintained on a VAX 8600 host computer. Harmony source is organized into a directory structure, a style of source management evolved from Thoth [Cargill79]. Because Harmony is a portable and open system, code specific to a device, processor, assembler, or compiler is put in a corresponding subdirectory under each directory. For compilation, an inclusion file containing only **#include** statements for all of the required source files is used. The directories used by the VAX-11/750 implementation are shown in Figure 2.1. The examples and tools each contain subdirectories for source, inclusion files, and documentation. They are not listed in detail. Subdirectories under those servers not used by the VAX-Harmony have been omitted. This chapter describes the implementation of the Harmony kernel on the VAX-11/750.

2.1.1 Booting

The target machine normally runs UNIX. The first step in porting Harmony was to get the hardware to run an alternative operating system. One way to reboot the VAX-11/750 is by using the BOOT command in the Console Command Language for the VAX-11/750 Console Subsystem. This command will deposit a boot control flag in a register, which can be referenced by software. The second level booting program for UNIX examines the boot control flag and accepts an alternate file name under the root directory for the operating system, if requested [Lalonde85]. Harmony is compiled as a C program; the binary image (in *a.out* format) is copied to the root directory on UNIX and rebooted on the console after UNIX is shut down.

This approach is convenient for target machines running UNIX. Program development can be done on the target machine and no other host computer is required. The development of this version of Harmony is done on a VAX 8600 instead of the VAX-11/750, but only to take advantage of the 8600's relative speed and stability (the VAX-11/750 was being used concurrently to test 4.3BSD UNIX and was not always available for program development).

The console terminal is controlled by four privileged registers. Its action is similar to a serial port. It has separate receive and transmit interrupt vectors in the System Control Block. For preliminary test programs, this terminal was used.

```
           |-doc
           |          |-aio
           |          |-clock
           |          |-ehvt
           |          |-fsys
           |          |-locator
           |-example--|-null
           |          |-srtest
           |          |-tabtest
           |          |-teach
           |          |-timing
           |
           |          |-doc
           |          |          |-boot
           |          |          |
           |          |          |-connect--|-contab
           |          |          |
           |          |          |-debug----|-busywait-|-vax750
           |          |          |          |-regular--|-vax750
           |          |          |
           |          |          |-gossip---|-vax750
           |          |          |
           |          |          |-kernel---|-vax750
           |          |          |          |-asa
           |          |          |-lib
           |          |          |          |-aio
           |          |-src------|          |-arm
           |          |          |          |-clock----|-vax750---|-asa
           |          |          |          |          |-userlib
           |          |          |          |-exsched
           |          |          |          |-fdev
           |          |          |          |-fsys
|-Harmony--|          |          |-servers--|-ikfb-----|-vax750---|-asa
           |          |          |          |-locator--|-vax750---|-asa
           |-sys------|          |          |-screen---|-vax750
           |          |          |          |-tablet---|-bitpad---|-vax750---|-asa
           |          |          |          |          |-userlib
           |          |          |          |
           |          |          |          |-tty------|-vax750---|-asa
           |          |          |          |-video
           |          |          |-streamio
           |          |
           |          |          |-connect--|-contab
           |          |          |-debug----|-busywait
           |          |          |          |-regular
           |          |          |-gossip
           |          |          |
           |          |          |-kernel
           |          |          |
           |          |-vaxinc---|-lib
           |                     |          |-clock
           |                     |          |-ikfb
           |                     |-servers--|-locator
           |                     |          |-screen
           |                     |          |-tablet
           |                     |          |-tty
           |                     |-streamio
           |          |-bound
           |          |-download
           |          |-examine
           |          |-fixexe
           |-tools----|-listing--|-doc
                      |          |-src------|-vax750
                      |          |          |-unix
                      |          |-vaxinc
                      |-makemsr
                      |-transfer
```

Figure 2.1 Directory Tree for VAX-Harmony

2.1.2 Interrupt Vectors

Interrupts and exceptions are vectored on the VAX-11/750. The vectors are stored in the System Control Block, which is described in the previous chapter. The physical address of the base of the System Control Block must be page-aligned. The current version of "as", the VAX-11 assembler on UNIX, does not support alignment larger than 4 bytes [Reiser83]. This alignment problem is solved by using "ld", the UNIX linker, which is capable of forcing the text segment origin to an arbitrary boundary. Assembling the interrupt vectors as a separate load module, putting this as the first load module to be linked, and giving the origin at a page boundary will ensure page alignment. If the kernel is debugged, the vectors can also be built at run time.

2.1.3 Interprocessor Interrupt

In a multiprocessor Harmony implementation, processors need to interrupt each other (and themselves) for message passing, task creation, and task destruction. This is usually done by custom hardware. When bringing up Harmony on a uniprocessor, this interrupt can be substituted by an interrupt local to the processor. If no appropriate interrupt can be used, it can even be simulated in software through function calls. This interrupt is only triggered by two system routines _Block_signal_processor() and _Signal_processor, which will be generalized as interprocessor communication code.

The interprocessor interrupt used by Harmony must be an interrupt and not an exception, because it must be maskable so that it will be held off from the time it is raised until the mask drops. This is to ensure that the interrupt does not occur in a critical section within the Harmony kernel.

The interprocessor interrupt must also be of a priority higher than the device interrupts. The software interrupts for the VAX-11/750 are of low priority; the interrupts of higher priority than the UNIBUS and timer interrupts are used for processor, memory or bus errors. Dr. Morven Gentleman suggested that Memory Write Timeout was the reasonable choice for substituting the interprocessor interrupt on the VAX-11/750.

The Memory Write Timeout interrupt is easily generated by writing to the address of a location beyond installed memory. However, since Harmony runs in kernel mode, any piece of code can write to non-existent memory causing the same interrupt, and thus it is hard to guarantee that an interrupt of this type is only generated by the interprocessor communication code – it could result from an error in the program. It would be nice to devise a method for detecting such spurious interrupts. One way to do this would be to require that only Memory Write Timeout interrupts generated by specific instructions associated with interprocessor communication are to be interpreted specially.

If the origin of the interrupt can be tracked by the address of the code which generated it, the system can check if it is indeed for interprocessor communication. The PC saved during the interrupt is the next instruction to be executed after the interrupt is serviced, thus one would expect that the saved PC would readily provide the address of the instruction that caused the Memory Write Timeout interrupt. Unfortunately, this is not the case. The reason is because the Memory Write Timeout is an interrupt and not an exception (which is necessary for it to work, but also provides a problem).

When the interprocessor communication code generates the interprocessor interrupt, it may not be acknowledged immediately if interrupts are masked for the critical section of mailbox access in the general multiprocessor case (this is always the case in the current VAX-Harmony implementation). This means that the interrupt will be delayed until the next task is dispatched. The dispatched task could either be the current task continuing to run or a new task. The interrupt is acknowledged when the active task has enabled interrupts. The PC at that time will be the next instruction to be executed by the dispatched task. Thus the PC cannot reliably specify where the interrupt was generated.

A second possibility for detecting special Memory Write Timeout interrupts would be to check for particular target addresses.

At the time of interrupt, the page being written to can be obtained from the Control and Status Registers of the memory controller, but the exact location within the page is not provided by the hardware and must be decoded from the instruction being executed. Decoding the target address would require the PC at the time of the interrupt, which we know is not available (as explained above). Thus it is not possible to require that only Memory Write Timeout interrupts to particular locations be valid for interprocessor communication, but we can demand that only certain pages be valid.

To increase protection against memory errors caused by user code, only one page in the validated virtual address space is mapped to non-existent memory. This page is "reserved" for generating Memory Write Timeout interrupts associated with interprocessor communication. Any interrupt caused by an attempt to write to this page is interpreted as a valid interprocessor communication signal. This still admits the possibility of spurious interrupts generated by erroneous user code, but the margin for error has been significantly decreased.

We could go one step further than this. It is possible to write protect the special page used for generating the Memory Write Timeout interrupt at all times other than during interprocessor communication by having the interprocessor communication code change the protection code in the memory map before and after generating the interrupt. If this is done, the translation buffer must be updated using the Translation Buffer Invalidate Single Register (TBIS) whenever the protection code changes. Updating the translation buffer slows interprocess communication and thus this extra precaution was not implemented in VAX-Harmony.

Using an actual interrupt is preferable to a simulation because it allows the interprocessor communication code to be more readily extended to a multiprocessor configuration. Although the Memory Write Timeout interrupt is not perfectly safe to use as the interprocessor interrupt, in practice it has not caused any problems. No matter what scheme is chosen for triggering interprocessor communication, an incorrect program can always generate a spurious interrupt, so our efforts can only reduce the chances of this happening.

2.1.4 Interrupt Stack

VAX-Harmony runs in kernel mode, so it uses the kernel stack.

The VAX-11/750 has an interrupt stack. The Machine Check and Kernel Stack Not Valid exception handlers can only run on this stack, because if these errors occur, even the kernel stack cannot be relied on. The PSL has a flag to indicate whether the interrupt stack is being used. The previous stack (either the kernel stack or the interrupt stack for VAX-Harmony) is switched back to when returning from an interrupt, depending upon the stacked PSL. The VAX architecture does not allow a switch from one of the normal stacks to the interrupt stack, which would be convenient for VAX-Harmony when handling the Machine Check and Kernel Stack Not Valid exceptions.

If the handling of these faults involves message passing or other code that causes a return from interrupt, stacks will be changed and data may be clobbered. It is preferable to use busywait I/O and no message passing in these special handlers. Since an application program cannot recover from these faults in most cases, this inconvenience should be acceptable.

If the flexibility in handling these errors is important, Harmony can run exclusively on the interrupt stack by setting the interrupt stack flag in the PSL. Dr. Morven Gentleman suggested that this is preferable because the stack switching can be avoided. VAX-Harmony will be modified to run on the interrupt stack in the future.

2.1.5 Stack Frames

A task is always dispatched by building an *interrupt stack frame* on the stack and invoking the return from interrupt (REI) instruction.

The PSL and the PC are pushed onto the stack when an interrupt is acknowledged and the hardware builds the stack frame. Depending on the type of interrupt or exception, zero to eleven parameters and possibly a type code may be pushed onto the stack before control is transferred to the interrupt handler. An interrupt stack frame is shown in Figure 2.2.

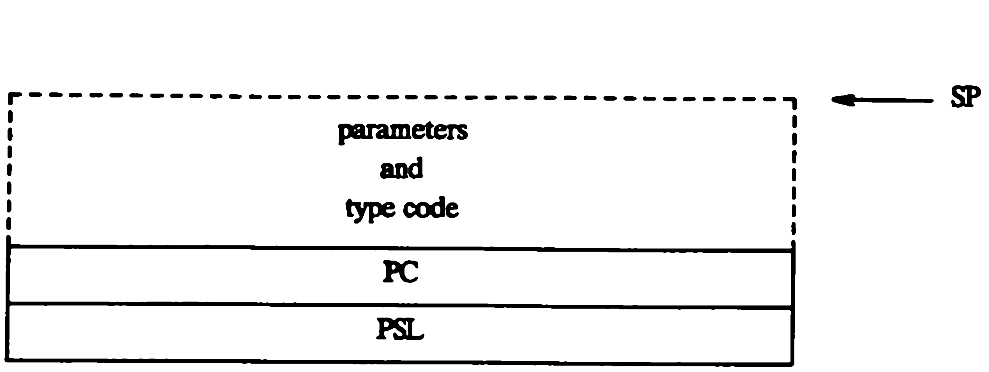


Figure 2.2 Interrupt Stack Frame

There is nothing in the stack frame to indicate the number of parameters pushed onto the stack (except for the special case of Machine Check). Thus the return from interrupt instruction always adjusts the stack under the assumption that there are no parameters or type code in the stack frame. Because of this the parameters must be removed explicitly from the stack frame by the interrupt handler prior to issuing the return from interrupt instruction. A fake interrupt stack frame is constructed containing only the PC and PSL which replaces the stack frame created by the hardware.

When a task calls a primitive such as _Send, _Receive, or _Await_interrupt where it blocks, it need not return from the function call normally; instead it can be removed from the ready queue until it is redispatched; at that time it will execute the instruction following the primitive. This optimization is straightforward on the MC68000 processor, because the only difference between an interrupt stack frame and a *function call stack frame* is the saving of the status register. When the task calls the primitive, the status register is pushed onto its stack to complete the interrupt stack frame. This is less true for the other processors in the MC68000 family. On the VAX, the function call stack frame is much different from the interrupt stack frame.

The VAX-11 instruction set provides two instructions for function calls: the CALLG instruction for functions whose arguments can be anywhere in memory, and the CALLS instruction that passes the argument list on the stack. The C compiler compiles all function calls into the CALLS instruction. The stack after executing this instruction is shown in Figure 2.3. The stack frame for the CALLG instruction is the same, but it has no argument count or list.

Assuming the PSW options are standard for the primitive functions in question, it takes about five instructions to reconstruct an interrupt stack frame in the place of the function call stack frame.

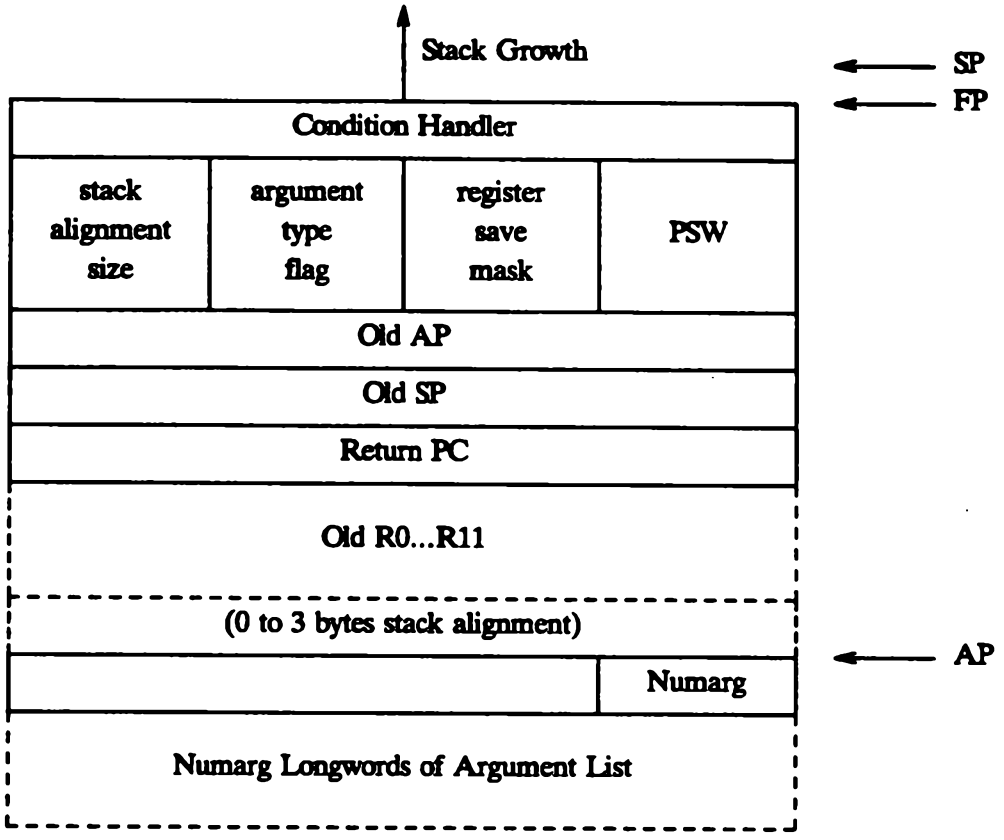


Figure 2.3 Function Call Stack Frame and Argument List

To let the task return normally from the primitives without optimization, an interrupt stack frame can be built on top of the current stack, with the return PC set to a return from function call (RET) instruction. This cleans up the function call stack frame. This is simpler, but it was found to be several microseconds slower than reconstructing stack frames after both approaches were tried. The faster approach of substituting a fake interrupt stack frame is used in the current implementation.

This complexity of the function call stack frame also comes into play when initializing the stack of a task. Harmony allows a task to destroy itself by letting it return to a _Suicide() function. Instead of just supplying the entry point as in MPC-Harmony, a complete stack frame has to be constructed for this purpose. To optimize the speed and space, a CALLG stack frame is constructed.

2.1.6 Function Entry Point

A function in VAX-11 assembler has an entry mask at the beginning. The entry mask is a 16-bit word specifying the function's register usage and overflow enables. It determines which of the general registers R0 to R11 are saved in the stack frame and restored when returning from the function. The actual entry point is the word immediately after the location defined by the function name. This is the entry address to put on the stack when creating a task or forcing it to destroy its children when it dies, because in these cases the function is entered through task dispatching instead of the function call instructions which recognize the entry mask.

Second level interrupt handlers are not invoked by the function call instructions but by jumping to the entry point from the first level handlers. For this reason, the handlers do not have entry masks.

2.1.7 Memory Management

The VAX-11/750 has a memory management unit that translates virtual addresses to physical addresses. It controls memory access by page protection. After the VAX-Harmony kernel ran in physical memory, the memory management capability was added. Although the translation of virtual to physical addresses is usually the identity transformation, the memory protection is useful. For example, it can be used to restrict the region of non-existent memory in the interprocessor interrupt generation mentioned above, or to write protect the interrupt vectors and the text (program memory) region from accidental modification.

The virtual address space of the VAX-11/750 is arranged so that the system space intended for the operating system is in the higher-numbered half. The lowest region is for a user process; its memory map is reloaded for each process when the context switching instructions are used. This design is intended for a multitasking environment where each task can use a large virtual address space that may be mapped onto external storage devices and retrieved by demand paging.

In a real-time operating system such as Harmony, only physical memory is used to ensure response time. If the system ran with memory management disabled, all addresses would be directly interpreted as physical memory addresses. However, the memory mapping facility is still useful to provide memory protection. This requires that memory management be enabled and thus that a decision be made as to which of the virtual memory regions would be used in VAX-Harmony.

With appropriate settings of the memory management unit, any of the three regions (system, P0 and P1) could be used to map directly to physical memory by translating virtual addresses directly to the corresponding physical addresses.

To allow a greater flexibility in terms of selective memory protection, the P0 region in process space is used instead of the system region, whose memory mapping is not updated during a context switch. Since Harmony uses a contiguous address space, the P1 region is ignored because it runs "backwards" from the P0 region.

For reasons of speed, only one page table is used for all the tasks running on VAX-Harmony. Potentially, each task could have its own page table with different write protection patterns. This would prevent a task from accessing memory sections reserved for other tasks and would make stack overflow easier to detect within a task.

Even though VAX-Harmony does not use the system space directly, the process space page tables must be located in system space, therefore the system space must also be defined. A simple solution is to double-map the process space onto the system space, so that both refer to the same physical region. One page table is built, and both the System Base Register (SBR) and the P0 Base Register (P0BR) point to it. The P1 Base Register (P1BR) is initialized to an arbitrary address in the system space and the P1 Length Register (P1LR) is set to 2 million (because it holds the number of non-existent pages), so all accesses to P1 region are invalid.

The translation of virtual addresses for the process space involves two memory references. With the cache and the translation buffer, the cost should be acceptable. The details of address translation are given in Figures 2.4 and 2.5.

For the Paint application, 256K bytes of memory is sufficient. The UNIBUS adapter registers only occupy the lower 4K bytes of the UNIBUS adapter address space; for the UNIBUS I/O space, the upper 8K bytes are mapped to device control and status registers while the rest are mapped to physical memory for DMA use. One page is mapped for the three memory controller registers for error checking. Finally, a single page beyond installed memory is mapped for generating the interprocessor interrupt. The mapping of virtual addresses is illustrated in Figure 2.6.

Due to the double mapping, the contents of memory in the system region are the same as in the P0 space.

Current implementations of Harmony assume that each processor uses a different region of a large contiguous address space in multiprocessor implementations, but that all processors can access each other's memory. Harmony currently allows a maximum of 16 processors. Because the MC68000 can only address 16 million bytes, each processor is defined with memory starting on a megabyte boundary. This is integrated into the current kernel software for Harmony.

The standard Harmony task id is 32 bits long, the lower order 24 bits of which are a pointer to the data structure describing the task. Bits 20 to 23 naturally identify which processor the task is running on. The higher order 8 bits are used to hold an integer

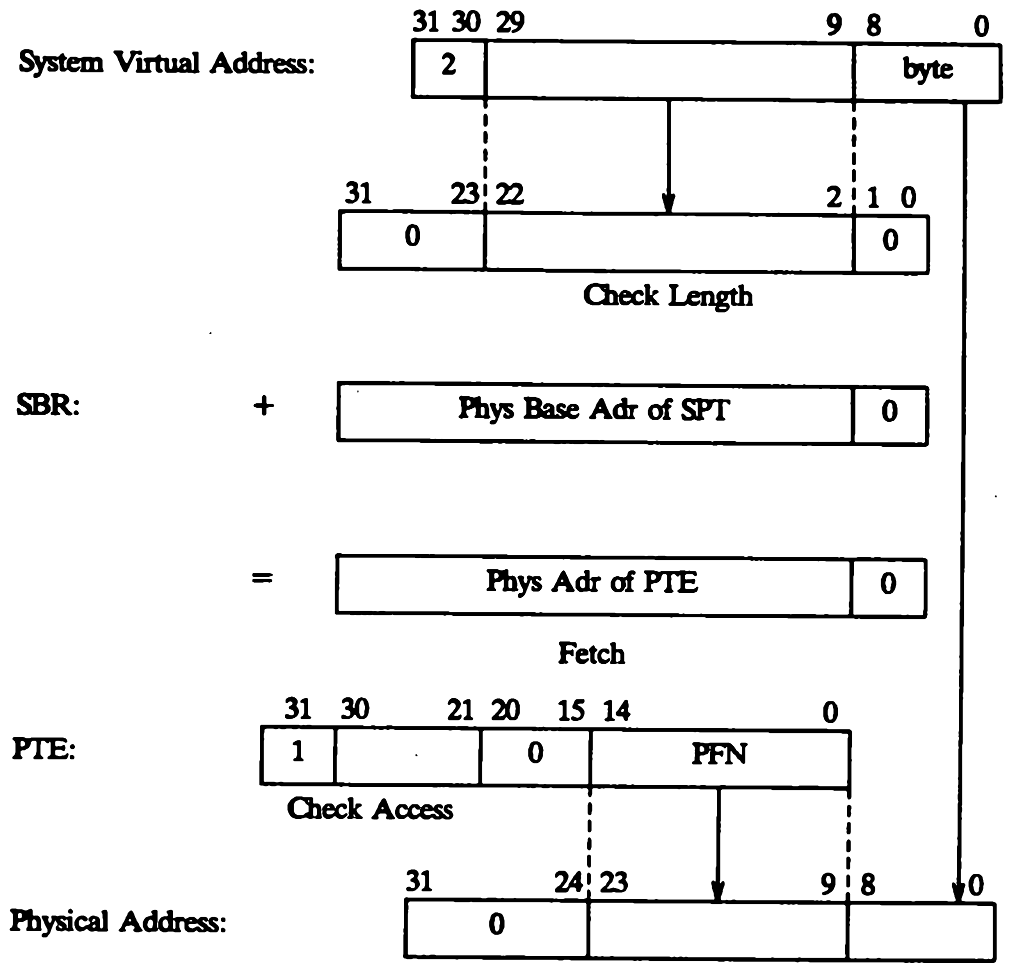


Figure 2.4 Virtual to Physical Address Translation in System Space

identifying the task. With this convention, no task can use more than 1 million bytes of local memory. These limits are parameterized and localized in several functions. They can be changed if necessary. For this application, the limits cause no problems, though it may be desirable to use the full 32-bit address space and more processors for larger applications.

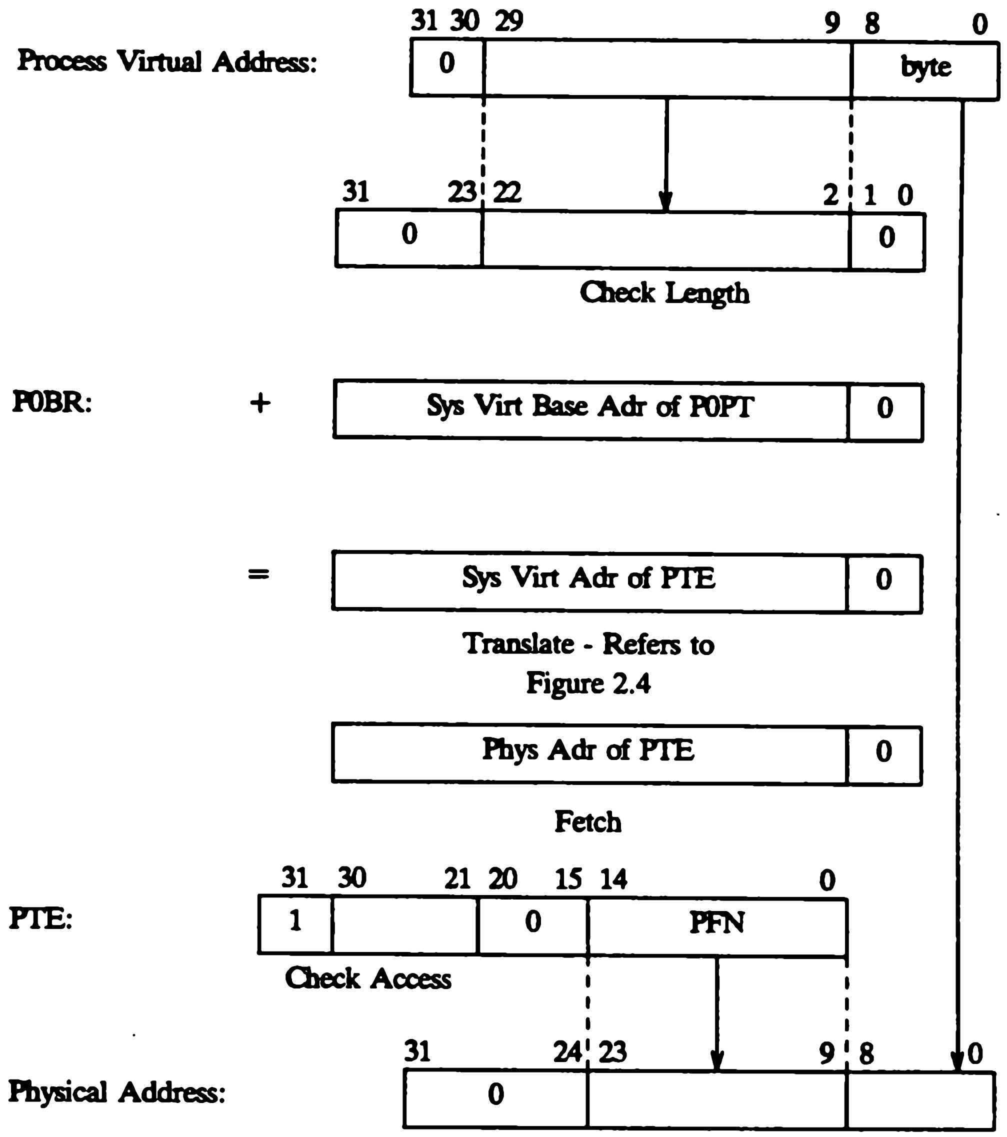


Figure 2.5 Virtual to Physical Address Translation in P0 Space

Virtual Address		Physical Address
0000 0000$_{16}$ ... 0003 FFFF$_{16}$	Program and Data	000000$_{16}$ ... 03FFFF$_{16}$
0004 0000$_{16}$ ... 0004 0FFF$_{16}$	UNIBUS 0 Adapter Registers	F30000$_{16}$ ... F30FFF$_{16}$
0004 1000$_{16}$ ... 0004 2FFF$_{16}$	UNIBUS 0 I/O Space	FFE000$_{16}$ ... FFFFFF$_{16}$
0004 3000$_{16}$ ... 0004 3FFF$_{16}$	UNIBUS 1 Adapter Registers	F32000$_{16}$ ... F32FFF$_{16}$
0004 4000$_{16}$ ... 0004 5FFF$_{16}$	UNIBUS 1 I/O Space	FBE000$_{16}$ ... FBFFFF$_{16}$
0004 6000$_{16}$ ... 0004 61FF$_{16}$	Memory Controller Registers	F20000$_{16}$ ... F201FF$_{16}$
0004 6200$_{16}$ ... 0004 63FF$_{16}$	For Interprocessor Interrupt	300000$_{16}$ ... 3001FF$_{16}$

Figure 2.6 VAX-Harmony Memory Map

2.2 Servers

The Harmony source contains many servers. Only some are applicable to the VAX-11/750. The *aio server* is for analog-digital conversions on an Analog Devices RT1-732-V board. The *video server* is used to control a Datacube Video Graphics Module. The *arm server* controls a robot arm. None of this hardware is available on the VAX-11/750.

The *file device server* under fdev and the *file system server* under fsys together implement a file system for Harmony. The VAX-11/750 has only one disk, which cannot be shared by UNIX and Harmony. These two servers were not ported, but they are useful for target machines that are dedicated to running Harmony.

The *exsched server* is an explicit scheduler. It uses a repeated event such as the video field interrupt for the Ikonas frame buffer to fake other interrupts for an alternative scheduling scheme. The code for the exsched server in the standard version of Harmony does not depend on the hardware generating the repeated event since it measures times in units of the repeated event interrupts. No changes were needed for the VAX-11/750.

2.2.1 Tty server

Due to the temperature and noise level of the machine room where initial debugging took place, the first server to be ported was the *terminal* or *tty server*.

The VAX-11/750 has a DMZ32 24-line asynchronous multiplexor for some of the I/O devices. The lines are divided into three octets. Each octet has its own receive and transmit interrupts and a receive silo for buffering character input. The DMZ32 can also perform DMA transfers from main memory. The DMZ32 control and status registers are shown in Figure 2.7.

Each line has four Indirect Registers: Indirect Register 0 is both for writing to the transmit silo and reading silo status, Indirect Register 1 is a command/status register for the line, Indirect Register 2 is for the DMA buffer address, and Indirect Register 3 contains the DMA character count. Each of these indirect registers can be accessed by writing the register number and line number to the corresponding fields in the Octet Control and Status Register.

The terminals are connected to the DMZ32 on the UNIBUS instead of normal serial ports. Though the multiplexing overhead is handled by the DMZ32, it causes other problems. Each of the eight lines in an octet has its own transmit silo, but there is only one window (Indirect Register 0) to access all eight silos. More than one device in the same octet will cause competition for the window. The selection and access of the indirect register

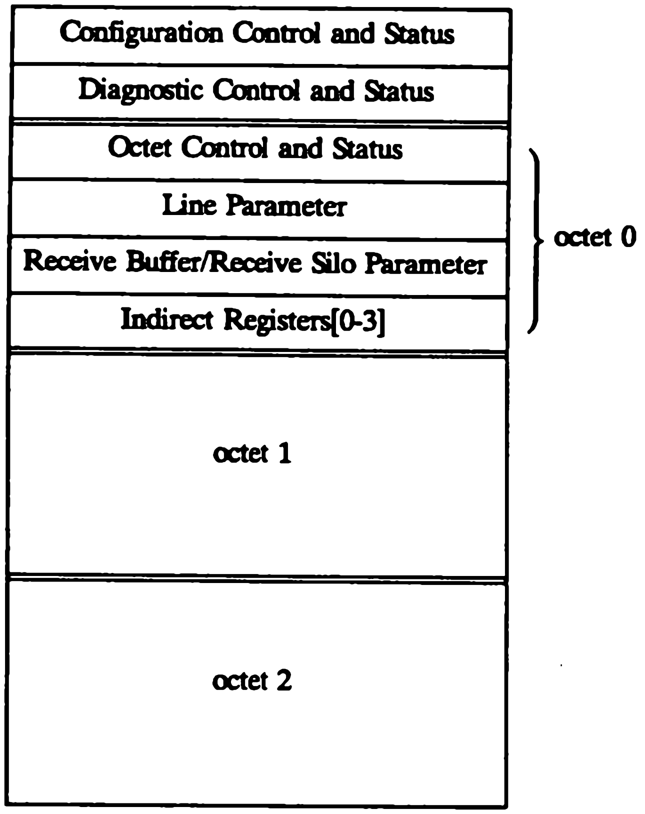

Configuration Control and Status
Diagnostic Control and Status
Octet Control and Status
Line Parameter
Receive Buffer/Receive Silo Parameter
Indirect Registers[0-3]
octet 0
octet 1
octet 2

Figure 2.7 DMZ32 Control and Status Registers

must be done with interrupts disabled to avoid misdirected data. Alternatively, all output in one octet could be handled by a octet driver task, instead of using one output task for each device.

2.2.2 Screen server

The *screen server* is an output control server; it controls the cursor for the terminal so that different tasks can use the same terminal for output. The screen server uses Harmony stream I/O. A new terminal type used on the VAX-11/750 was added to this server.

2.2.3 Tablet server

A Summagraphics Bitpad One tablet is connected to the DMZ32. The tablet has two different modes of operation: in the ASCII mode, the tablet coordinates are readable as Arabic numerals and each coordinate is composed of 12 bytes; in the binary mode, each coordinate is encoded in 5 bytes.

The *tablet server* is similar to the terminal server in organization. It receives input from a DMZ line and occasionally writes commands to the same line. An overseer task forces input if the tablet has not been moved in a specified period of time. The tablet server allows the user to define *windows* which are regions on the tablet. Different tasks can use different regions on the tablet. The rate of sampling can be varied. To reduce overhead and jitter, the coordinates are filtered by the tablet server.

The standard tablet server handles coordinates in ASCII mode, which is not efficient enough for most real-time applications. The server was modified to use binary mode on the VAX-11/750.

2.2.4 Locator server

The locator package is a general input package written by Dave Forsey. It provides a reasonably consistent interface to all the graphics input devices available in the Computer Graphics Laboratory, including tablets, mice, and joysticks. It is implemented both as a UNIX library and as a server on MPC-Harmony. The *locator server* allows the user to provide a *viewport*, which defines the region of the physical device to be used, and a *window*, which defines a range of values desired from the device. The current position of the locator is mapped to a *virtual cursor* location in the window. Unfortunately this convention conflicts with that used by the tablet server. The locator server was ported to run on VAX-Harmony.

2.2.5 Clock server

The *clock server* allows a task to get and set the time and to delay the execution of the task itself for a specified period of time.

The MPC uses a M6840 timer module. The M6840 has a 16-bit count register, into which the user writes an unsigned integer. The timer decrements the count every microsecond until it underflows and generates an interrupt. The interval between interrupts is called the *alarm resolution* and must be less than 64 ms because of the limitation imposed by the size of the count register.

A programmable timer is part of the VAX-11/750 hardware. Access is only through privileged registers. The clock rate is 1MHz, the same as the M6840 module, but the count register is 32 bits instead of 16, allowing a larger alarm resolution and thus less overhead. There are differences in the way the timer is operated in the VAX-11/750. In particular, the count register must be loaded with the negative alarm resolution since the counter goes up instead of down.

2.3 Software Development Tools

Harmony supplies three basic host-based tools: listing, bound, and examine. The debug tool is embedded in a Harmony program and can be considered as part of Harmony. The other available tools – download, fixexe, makemsr, and transfer – are specific to other hardware and not applicable to the VAX-11/750.

2.3.1 Listing

Harmony source is organized in a directory structure with many small files. C compilation is done with include files that reference the appropriate source files. The *listing* tool generates a readable program listing that has a cover, a table of contents, and numbered pages with headings. It can operate on the include files or it can print a list of files directly as text files. This tool was modified at the beginning of the project to run on 4.2BSD UNIX.

2.3.2 Bound

Bound helps determine the stack size required by each task. It constructs the call graph of the root function and all the functions called by that function, then calculates its stack size including the maximum stack requirement of all its subgraphs. *Bound* cannot be fully automatic because of some exceptions. If there are indirect function calls through pointer variables, the possible functions must be supplied interactively. If the call graph involves a cycle (implying possible recursion), the maximum number of levels of recursion must be supplied by the user. If the stack pointer is loaded with an unknown value, *bound* will also ask the user for a stack size.

This tool operates on an a.out format executable image for the VAX-11/750. For each root function the user specifies, *bound* will produce a minimum stack requirement and a larger one that provides for Harmony exception handling. The latter should be chosen during program development to allow for errors. Shawn Neely ported *bound* for the VAX architecture.

2.3.3 Examine

Examine is a tool used on an executable image; it can be used to print external symbols and initialized data, disassemble instructions, and patch values in the image. This tool is not yet implemented for the VAX-11/750.

2.3.4 Debug

Debug is actually a library of functions rather than a debugger task or server. The user must compile calls to the _Breakpoint function into the source code. Once invoked, debug runs as the active process without affecting other tasks, allowing the user to inspect and change memory locations, display status information and system data structures, examine the structure of the memory pool, and display the call stack of a task.

Debug uses either regular I/O or busywait I/O, so the debugging of the kernel can be done without the stream I/O facility. There are two subdirectories – busywait and regular – under the debug directory; the user can link in either one of the two libraries with the program.

Debug has been modified to run on VAX-Harmony. As mentioned earlier, the stack frames for various interrupts and exceptions are different. Each must be considered separately in the _Put_traceback() function for displaying the call stack.

3 A Prototype Graphics Application

3.1 Ikonas

MPC-Harmony has a well-defined Ikonas library. The same interface is retained for VAX-Harmony, but access to the Ikonas is a bit different.

The MPC is on the Ikonas bus, providing it easy and fast access to the other hardware on the Ikonas. The MPC address space has 512 8192-byte windows, each mapped onto a contiguous block of 2048 Ikonas words on a 2048-word boundary in the Ikonas address space. Each window has a *translation control block* that supplies mapping and access information. With this mapping, the MPC can access the Ikonas address space as memory.

In the VAX-11/750, the Ikonas is connected to the UNIBUS through the Ikonas IF/DMA host interface. Its control and status registers in the UNIBUS I/O space are shown in Figure 3.1.

UNIBUS Word Count Register
UNIBUS Address Register
UNIBUS Status/Command Register
Data I/O Register
Lower Ikonas Address Register
Upper Ikonas Address Register
Ikonas Status/Command Register

Figure 3.1 UNIBUS Control and Status Registers for the Ikonas

Programming data transfers between the Ikonas and its host, in this case the VAX-11/750, is discussed in detail in Ikonas documentation [Adage82a], a summary of which is given below.

The VAX-11/750 can perform direct programmed I/O on the Ikonas. The procedure is as follows:

- Set Ikonas Command Register (clear DMA bit)
- Set Lower Ikonas Address Register
- Set Upper Ikonas Address Register
- Set UNIBUS Command Register
- For write operations, write Data I/O Register; for read operations, either wait for interrupt (enabled in UNIBUS Command Register) or busywait on the READY bit in the UNIBUS Status/Command Register, then read Data I/O Register.

The VAX-11/750 can also initiate DMA transfers to and from the Ikonas. This is handled by the IK11B board and does not take up the VAX processor's time. The procedure is as follows:

- Map UNIBUS I/O space to VAX main memory
- Set Ikonas Command Register (include DMA bit)
- Set UNIBUS Word Count Register
- Set UNIBUS Address Register
- Set Lower Ikonas Address Register
- Set Upper Ikonas Address Register
- Set UNIBUS Command Register
- Wait for interrupt or busywait on the READY bit in the UNIBUS Status/Command Register

The address given in the UNIBUS Address Register is an 18-bit address in the UNIBUS I/O space. The first 248K bytes of this space are mapped to an area in physical memory. The map information is in the UNIBUS adapter registers, which are shown in Figure 3.2. The UNIBUS map entries are described in Figure 3.3. They are very similar to the page table entries for the memory management unit. The address translation is summarized in Figure 3.4. The page number in the UNIBUS address is extracted and the corresponding map entry is looked up. If the valid bit is set, the page frame number will be extracted and concatenated with the byte number to form the physical address.

The data path number in a UNIBUS map entry specifies which data path is used. Data path 0 is the direct data path, while paths 1 to 3 are buffered data paths, where data are grouped into 32-bit longwords to minimize transfers. If a buffered data path is used and the number of bytes transferred is not a multiple of 4, the data path must be flushed using the corresponding Data Path Register in the UNIBUS adapter registers.

Offset(hex)	
000	Configuration Register
004	Control Register
008	Status Register
00C	Diagnostic Control Register
010	Failed Map Entry Register
014	Failed UNIBUS Address Register
018	Reserved
040	Data Path Registers[0-4]
050	Reserved
800	Map Registers[0-495]
EC0	Reserved

Offset relative to base of UNIBUS Adapter Address Space
Registers unused in the VAX-11/750 are not shown

Figure 3.2 UNIBUS Adapter Registers

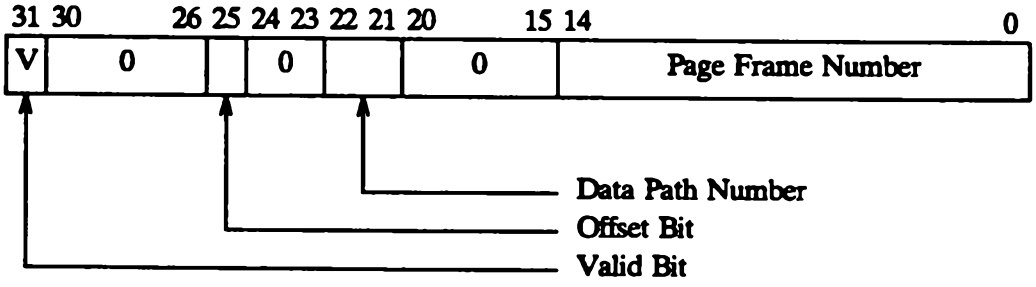


Figure 3.3 UNIBUS Map Entry

Since the UNIBUS space is mapped onto physical memory and bypasses the memory management unit, no memory access protection is provided. When developing a driver for a UNIBUS device, the following safety measure is useful: map one more page than necessary and explicitly mark the last page invalid. If an error causes the driver to transfer too large a block of data, it will result in a fault instead of writing over the memory [Martindale86].

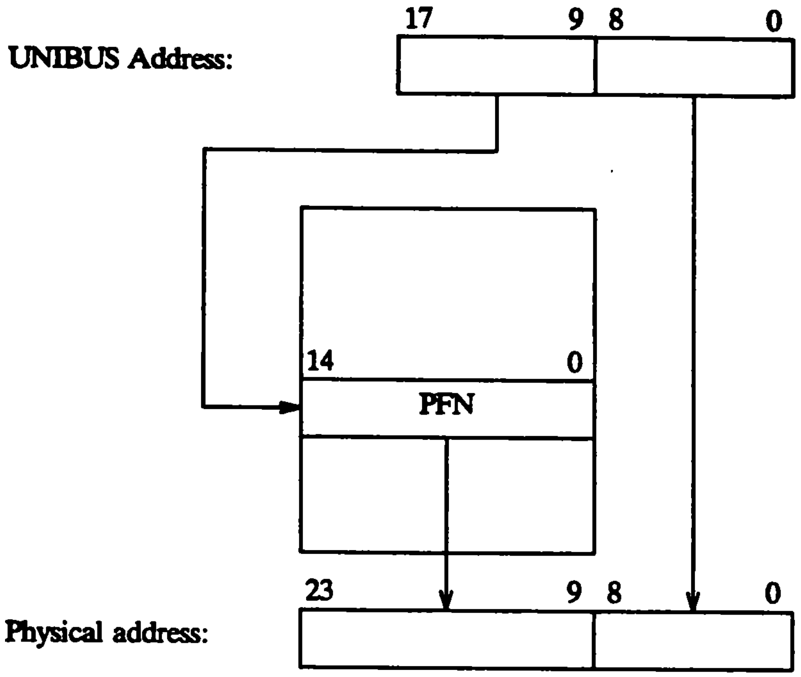


Figure 3.4 UNIBUS to Physical Address Translation

Four functions are implemented for access to the Ikonas: _IkDioRead, _IkDioWrite, _IkDmaRead, and _IkDmaWrite. Their usage is similar to the functions of those names in the Computer Graphics Laboratory Ikonas library on UNIX. All other functions in the VAX-Harmony Ikonas library use these four as the lowest level interface to the Ikonas. Because function calls are expensive on the VAX-11/750, many apparent functions are converted directly to these primitive function calls using the **#define** feature of the C preprocessor.

The MPC maps Ikonas memory to windows in its address space using Translation Control Blocks. The Paint program accesses Ikonas memory mostly through direct I/O or DMA of a short length. Both are slow compared to the MPC's access times. Even though DMA of a longer length is more efficient, it will take a longer time overall and require a lot of space. Therefore access to a large area in Ikonas memory is usually done by the bit-slice microprocessor.

There is as yet no formal server to exclusively handle all access to the Ikonas. Any process can call the functions provided in the library. The current implementation assumes that no two processes in the application program will use the Ikonas at the same time or the results will be unpredictable. In more complicated applications, where this assumption will not hold, a simple high priority proprietor task can be added to guarantee sequential access and mutual exclusion [Gentleman81]. The proprietor task waits for the Ikonas to be ready, then receives from any clients asking for permission to use the Ikonas, replies to grant the permission, and waits for the next I/O completion.

3.2 Paint

The Paint program was designed using the methodology of anthropomorphic programming [Booth84]. The program consists of a number of tasks that communicate with each other in a fashion analogous to humans in an organization. The organization of the tasks in Paint is summarized in Figure 3.5.

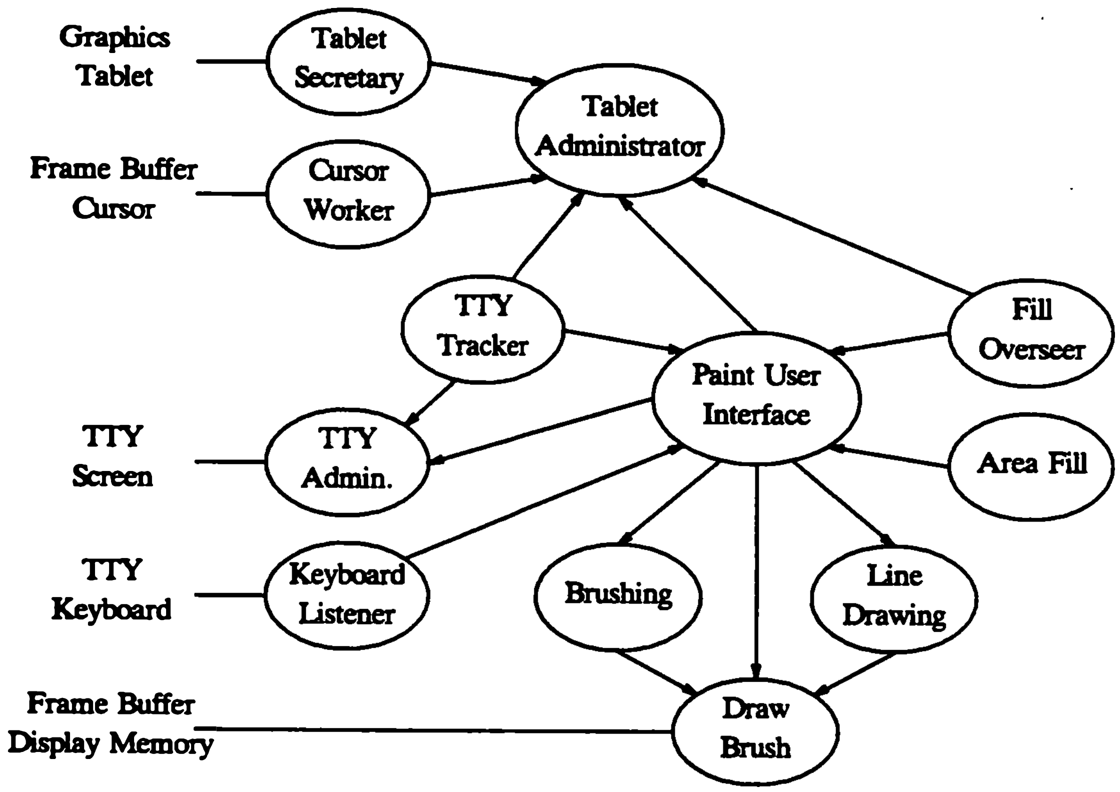


Figure 3.5 Organization of Tasks in Paint

In the diagram, ellipses represent tasks and arrows indicate the direction of sending in message passing. More details on the design of the Paint program are found in another paper [Beach82]. The tablet secretary actually talks to the Harmony tablet server instead of directly using the tablet hardware. The TTY administrator and the keyboard listener access the alphanumeric terminal through the use of Harmony stream I/O.

The Paint program had already been translated to C by Dave Forsey, so the changes required to run under VAX-Harmony were mainly the device dependent code, including the Ikonas interface and the tablet and terminal servers.

The source directory tree for Paint is shown in Figure 3.6.

```
                  |-BRUSHES
                  |
                  |-CURSOR----------|-CURSOR_DEFNS
                  |                 |-OUTLINES
                  |
                  |-FB--------------|-MICRO
                  |
                  |-LEVEL1----------|-TEXT
|-PAINT-----------|
                  |-TABLET
                  |
                  |-TEXT
                  |
                  |                 |-CONSTRAINTS
                  |                 |-FILE
                  |-USER_INTERFACE--|-FILL
                                    |-PAINTING
                                    |-TEXT
```

Figure 3.6 Paint Source Directory Tree

The Paint program has its own set of functions for the Ikonas (maintained under the FB directory), the lowest level of which comprises four functions: Ikprd1, Ikpwr1, Ikprd, and Ikpwr. These handle the reading and writing of a single pixel and the reading and writing of multiple pixels, respectively. The four functions parallel the low level Ikonas library functions in VAX-Harmony. The Paint program was modified to call the corresponding VAX-Harmony Ikonas functions with appropriate changes to their parameters.

Some other functions were slightly modified to conform to the VAX-Harmony Ikonas library. Data structures that assumed the MPC byte ordering (higher order in the first byte) were reorganized for the VAX.

The tablet secretary uses the locator server package. The current locator package has enough capability so that the functions of the tablet administrator can be substituted by locator interface and utility functions, but the saving of message passing time is only significant in the uniprocessor case when the processes are not sampling tablet coordinates more frequently than they are received, so the original organization of the tasks is unchanged.

The TTY_Administrator is similar to the screen server used by Harmony. It was only modified to handle a different type of terminal.

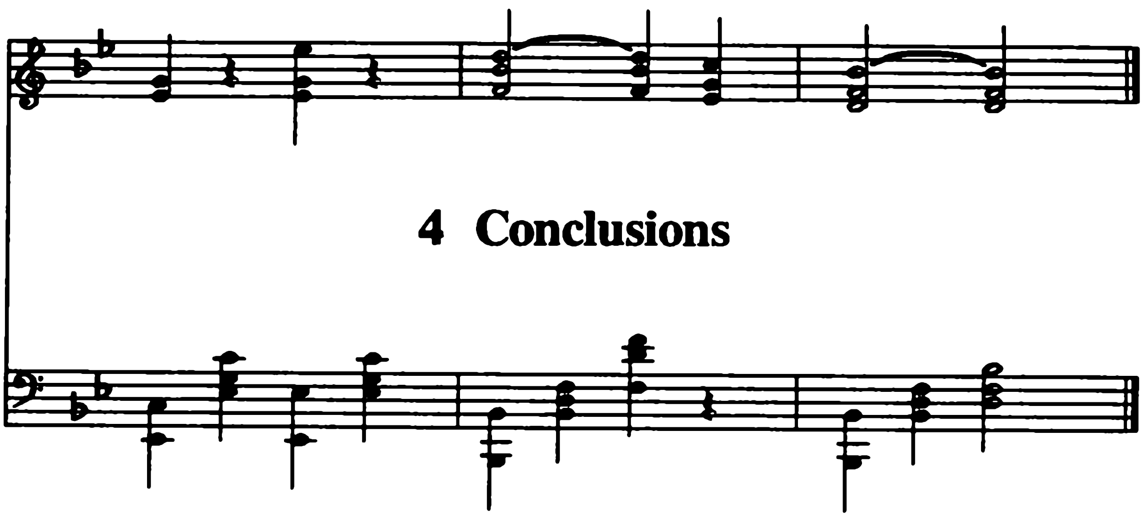
The VAX-11/750 has only one disk, which cannot be shared by UNIX and Harmony. The file system servers were not ported, so the saving of image data in disk files is not implemented. The microcode for the bit-slice processor and the encoded menu area data must be downloaded by UNIX before booting Harmony. This would be easily accomplished with a Harmony file system.

MPC-Harmony uses the VAX connected to the Ikonas as a file server because the Ikonas had no file system. The VAX runs UNIX, so all of its file facilities are available to MPC-Harmony with appropriate server code to transfer data to and from the Ikonas using the standard UNIX library for the Ikonas.

The Ikonas connected to the VAX-11/750 cannot use this same scheme because UNIX is not running on the VAX-11/750 when VAX-Harmony is running. If a second VAX (such as the VAX 8600) were simultaneously connected to the Ikonas, the existing file server code from the MPC-Harmony version of Paint could be used without modification. This was in fact the intention when the Ikonas was originally configured with two IF/DMA host interface boards. Unfortunately, an upgrade from the Ikonas RDS-2000 to the RDS-3000 incorporated larger boards.

On the RDS-3000, the IF/DMA host interface board is part of the frame buffer controller board. Therefore having two host interfaces means having two frame buffer controllers for the same frame buffer. Normally this would cause problems. However, the Computer Graphics Laboratory succeeded in disabling the frame buffer controller on one board, thus successfully connecting two hosts to the same Ikonas frame buffer. This configuration ran successfully using a Honeywell Level 6 (the original vehicle for Paint) as the first host and a PDP-11/45 and then a VAX-11/780 as the second host.

The hardware boards were subsequently modified by Adage when the RDS-3000 was "upgraded" to meet Adage's standards for maintenance contracts. Adage now informs the Computer Graphics Laboratory that it is impossible to configure two hosts on the Ikonas bus. If this were done, however, it would possible to take advantage of a second UNIX system as a file server for Paint on VAX-Harmony.



4 Conclusions

The porting of Harmony to the VAX-11/750 has been completed. One advantage that facilitated this porting was that the VAX architecture is designed for a multiuser time-sharing environment and various types of applications. This is reflected by the range of data types and the hierarchical control schemes available. Harmony requires only a subset of the many capabilities that VAX computers offer. Porting Harmony to the VAX thus did not involve any hardware incompatibilities. The only drawbacks were more time spent on studying the system and the cost of initializing some of the unused hardware features offered by the VAX architecture (such as initializing unused registers in case of error).

Many of these features were not used because it was feared that their use might degrade the real-time performance of VAX-Harmony. If a faster VAX-based computer is used in the future, these features may be added to Harmony to extend its power while maintaining its real-time performance.

An example of a hardware feature that is used in VAX-Harmony is the memory management unit. The Harmony kernel was first brought up with the memory management unit disabled. Harmony does not use demand paging, so a Harmony application can use physical memory directly. The reason for adding memory management is that Harmony runs in the kernel mode and the C language offers little error checking, so the memory management unit is set up to provide memory protection. The cost of this checking is the overhead of address translation, which is not very significant.

VAX-Harmony uses one page table for all its processes because of efficiency considerations – context switching that required new page tables might have a significant effect on timing. If this turns out not to be significant, it would be possible to further utilize the memory management by having a different set of page tables with selective protection for each process.

Harmony itself is a well-designed and portable system. It compares favorably to other real-time operating systems [Parr86]. There were no problems porting the device independent code for the Harmony kernel. The integration of I/O devices was flexible enough for all the devices used in VAX-Harmony.

The source code organization of Harmony facilitates the transition from one hardware environment to another. It isolates the parts requiring change and makes managing the source code for an increasing number of architectures easier. If the device dependent code for the different architectures were kept in the same file for conditional compilation, it would have been incomprehensible.

Harmony is a relatively new system, so little documentation is available. The technical report on using Harmony covers the user's view of Harmony, but the internals of Harmony are not discussed in detail. The best way to understand Harmony is still by studying the source code, which is the reason Harmony was described as a "hacker's system" [Forsey85].

This was probably the most difficult and time consuming part of porting Harmony. For example, a working knowledge of the MC68000 is needed when translating the low level assembler code, in addition to the knowledge of Harmony and the VAX. The new documentation in Release 2.0 is helpful for understanding Harmony. For the internals, Marc Riese's Master's Essay contains a description of Harmony code that supplements the in-line documentation [Riese86].

The debugging tool is still primitive. Since it is part of Harmony, it needs to be debugged itself. However, further development of the debugger is under way. In addition to the improvements from the new release, local enhancements are also being added [Brossard86].

This chapter contains discussions about the results of the VAX-Harmony port. The first section provides some timing data and performance evaluation. The second section discusses some limitations and drawbacks. The last section describes some recent extensions to Harmony and directions for future research.

4.1 Evaluation

The following data were obtained using the timing example in Harmony. Each instruction was executed in a loop of 10000 iterations and the time taken after the loop.

Time in microseconds:

Operation	VAX Raw	VAX Corrected	MPC Corrected
null loop	5.1	0.0	0.0
32bit shift(<<16)	9.9	4.8	6.48
16bit shift(>>16)	12.84	7.74	7.4
16bit add(reg)	12.68	7.58	4.32
16bit add(RAM)	12.83	7.73	6.95
16bit mult	17.24	12.14	5.48
16bit div	20.3	15.2	82.47
32bit add	8.28	3.18	2.4
32bit mult	11.4	6.3	58.35
32bit div	15.78	10.68	79.98
floating add by 0	13.08	7.98	304.82
floating mult by 0	13.36	8.26	382.73
floating mult by 1	13.9	8.8	418.95
floating div by 1	17.96	12.86	1005.74
null function call	25.85	20.75	16.7
single arg function	27.59	22.49	20.83
Send/Rcv/Rply	2212.3	2207.2	1668.54
Destroy(Create)	18678.0	18672.9	13220.5
single pixel read	96.99	91.89	11.55
single pixel write	107.44	102.34	4.77
DMA read(1 pixel)	485.95	480.85	
DMA write(1 pixel)	483.24	478.14	
DMA read(512 pixels)	3818.04	7.45	
DMA write(512 pixels)	3267.95	6.37	

The *corrected* timings column have the looping overhead subtracted. For 512 pixel DMA operations, the corrected timings are per pixel.

The minimum instruction time for the VAX-11/750 is given as 0.32 microseconds. The MPC is an MC68000 board, with an 8MHz clock rate and a minimum instruction time of 0.05 microseconds. The 32-bit multiplication and division are slow on the MPC because it does not have these instructions in its instruction set [King83]. The 16-bit divide is slow because the compiler used (MIT's *mc*) always does a 32-bit division to maintain precision [Forsey85]. The floating arithmetic figures are better on the VAX-11/750 because of its floating point accelerator.

_Create and _Destroy are slow because of the many function calls and extra initialization. _Create takes about 12 ms and _Destroy about 6 ms. If the _Tune_Getvec function is called first to optimize the memory pool search algorithm, task creation time can be cut down to about 5 ms.

Access to the Ikonas is much slower on the VAX. The Paint program was thus also expected to be slower than the original MPC implementation. However, when the Paint program is executed, the response is comparable to the MPC version – the tablet input is handled in real time with no perceivable delay. This proved the strength of Harmony as a real-time system, making up for the loss of processor power.

4.2 Limitations

The VAX-11/750 is designed for a uniprocessor environment. It cannot be used in a multiprocessor Harmony implementation without external hardware support.

On a uniprocessor system, there is always a processing limit above which real-time response will fail. Currently the message passing cycle on VAX-Harmony takes about 2.2 ms, therefore the limit of message passing cycles per second is about 450 on the VAX-11/750. In the Paint program, the tablet is sampled at 1200 baud, or about 30 coordinates per second, allowing the equivalent of 15 message passing cycles for processing each new tablet position. A tablet coordinate is passed through a chain of tasks and translated to some action on the frame buffer. This involves about 10 message passes in the current implementation, so the limit is not reached. If the tablet is sampled at 2400 baud, real-time response will not be possible.

If the uniprocessor implementation cannot meet the demand, a natural solution would be adding more processors. Multiprocessors based on the VAX architecture have been developed; they will be considered for a future multiprocessor Harmony implementation.

Many features of the VAX-11/750 are useful for supporting multiuser time-sharing but not real-time systems. Some examples include the detailed function call stack frame, the memory management unit with four access modes, hardware-supplied data for demand paging and per-process memory maps, and the fifteen software interrupt levels for priority interrupt scheduling.

The VAX instruction set provides two instructions for loading and saving the *process context*, which includes the stack pointer for each access mode, all the general registers, the PSL, and the mapping registers for the process space. These instructions are to support the operating system in context switching. However, the save context instruction also invalidates the translation buffer, and switches to the interrupt stack. Using these instructions requires manipulating unused registers and increases memory access time, therefore they are too expensive to use in VAX-Harmony. Instead, the stack manipulation instructions are used to load and save the general registers, as in MPC-Harmony.

Although useful for large time-sharing systems such as VMS or UNIX, these features may actually hinder the performance of a simpler real-time system like Harmony when the processor power is limited.

The UNIX-based arrangement of UNIBUS devices places all I/O devices at level 15. This may cause more urgent interrupts (like those of the tablet) to be missed. This is not using the priority scheduling system in Harmony to best advantage and should be changed.

4.3 Extensions

Harmony is a relatively new system. It still has room for improvement.

With the new Release 2.0, Harmony adopted a standard for software maintenance. Directories are set up for recording bugs, new features, and details about changes made. When maintaining Harmony for several different architectures simultaneously, changes or bug fixes to device independent code must be observed by all versions. During the VAX-Harmony port, one such change caused an unexpected and hard to find error. Therefore this new maintenance scheme will be a useful addition to Harmony. On-line documentation has also increased, both as feature documentation and Harmony application notes. These are useful for understanding the system.

The debugger in Harmony was very limited. Every change of breakpoints requires a recompilation. Embedding the debugger in the code changes the timing and the behavior of a program. The debugger in Release 2.0 allows breakpoints to be planted dynamically, but only on a per-process basis. It also provides more detailed processor status information and *use bits* for the user to keep track of events on a processor. The new debugger is planned to provide new features in three areas: traditional debugging tools for non-real-time programs, tools designed specifically for real-time multitasking multiprocessor programs, and a better user interface.

Rob Parr implemented an Ethernet server for Harmony, allowing it to communicate with and use the resources of other machines, potentially expanding its power [Parr86a].

The message passing time in Harmony is still slow. The possibility of implementing message passing in hardware has been examined in the Sylvan project [Riese86] and on the Dy4 hardware, but no research in this direction has been done on the VAX architecture.

Harmony has been demonstrated to run successfully on the VAX architecture. It should be ported to MicroVAX II-based workstations as the next step toward a multiprocessor VAX implementation.

The MicroVAX II/GPX workstations are prospective target machines for Harmony [Digital86]. The architecture is compatible with the VAX minicomputers, except for the following differences:

- The MicroVAX architecture uses a subset of the data types, instructions, and privileged registers in the VAX architecture. The missing instructions are supported by emulation. Harmony itself does not use any of the missing types or registers, so their unavailability is not a problem.

- The MicroVAX has 30-bit physical addresses, though the physical address space is still split into memory space and I/O space like the VAX-11/750.
- The interrupt vectors are slightly different. The MicroVAX has an interprocessor doorbell interrupt at level 14, but because it is lower than the I/O interrupts, it cannot be used for the message passing interprocessor interrupt. External hardware support is thus still required for running a multiprocessor configuration of Harmony.
- The timer priority is lowered to level 16. Hopefully I/O devices of higher priority will not cause the timer to lose any clock ticks.
- The MicroVAX uses a Q22 bus instead of the UNIBUS. The operation is very similar to the UNIBUS.

The GPX workstation has a VCB02 video subsystem with a high resolution monitor. A screen management system would be a desirable feature for the graphics monitor. It will be useful for handling multiple input devices, as in the Adagio switchboard [Tanner86]. The video subsystem provides scrolling and clipping support for rectangular viewports [Digital86a]. It will be useful for screen management in a workstation environment.

The MicroVAX II processor is over 50% faster than the VAX-11/750. In addition, it has a VLSI coprocessor for graphics operations. This could replace much of the functionality now supplied by the Ikonas. It is an attractive target machine for real-time graphics applications. VAX-Harmony will be ported to the MicroVAX II processor. A four month effort is expected to be sufficient for porting the kernel and the basic servers.

References

Adage82. Adage Inc., *RDS 3000 User's Guide*. April 1982.

Adage82a. Adage Inc., *RDS 3000 Programming Reference Manual*. June 1982.

Beach82. R. J. Beach, J. C. Beatty, K. S. Booth, D. A. Plebon, and E. L. Fiume, The Message is the Medium: Multiprocess Structuring of and Interactive Paint Program, *Computer Graphics* 16(3) pp. 277-287 (July 1982).

Booth84. K. S. Booth, W. M. Gentleman, and J. Schaeffer, Anthropomorphic Programming, (Tech. Rep. CS-82-47), Department of Computer Science, University of Waterloo, Waterloo, Ontario (February 1984).

Brossard86. A. Brossard, *personal communication*. 1986.

Cargill79. T. A. Cargill, A View of Source Text for Diversely Configurable Software, PhD Thesis, University of Waterloo, Waterloo, Ontario (1979).

Cheriton79. D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager, Thoth, a Portable Real-time Operating System, *CACM* 22(2) pp. 105-115 (1979).

Digital81. Digital Equipment Corporation, *VAX Architecture Handbook*. 1981.

Digital82. Digital Equipment Corporation, *VAX Hardware Handbook*. 1982.

Digital86. Digital Equipment Corporation, *VAXstation II/GPX Technical Manual, BA123 Enclosure*. January 1986.

Digital86a. Digital Equipment Corporation, *VCB02 Video Subsystem Technical Manual*. February 1986.

Forsey85. D. R. Forsey, Harmony in Transposition: A Toccata for Vax and Motorola 68000, Master's Thesis, University of Waterloo, Waterloo, Ontario (March 1985).

Gentleman81. W. M. Gentleman, Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept, *Software - Practice and Experience* **11** pp. 435-466 (1981).

Gentleman85. W. M. Gentleman, Using the Harmony Operating System, (Tech. Rep. NRC/ERB-966), Division of Electrical Engineering, National Research Council of Canada, Ottawa, Ontario (December 1983, revised May 1985).

King83. T. King and B. Knight, *Programming the M68000*, Addison-Wesley (1983).

Lalonde85. K. W. Lalonde, *personal communication*. June 1985.

Martindale86. D. M. Martindale, *personal communication*. July 1986.

Parr86. R. K. Parr, Realtime Multitasking Multiprocessing Operating Systems - A Detailed Comparison of VRTX, MTOS, pSOS, and Harmony, National Research Council of Canada, Ottawa, Ontario (1986).

Parr86a. R. K. Parr, TCP/IP Ethernet Support for the Harmony Operating System, Master's Thesis, University of Waterloo, Waterloo, Ontario (1986).

Reiser83. J. F. Reiser and R. R. Henry, *Berkeley VAX/UNIX Assembler Reference Manual*, Bell Laboratories, Holmdel, New Jersey (November 79, revised February 1983).

Riese86. H. M. Riese, Towards Harmony on Sylvan, Master's Essay, University of Waterloo, Waterloo, Ontario (1986).

Tanner86. P. P. Tanner, S. A. MacKay, D. A. Stewart, and M. Wein, A Multitasking Switchboard Approach to User Interface Management, *Computer Graphics* **20**(4) pp. 241-248 (August 1986).