Kennedy, Timothy James

A MULTIPROCESSOR KERNEL AND MONITOR FOR IMAGE PROCESSING APPLICATIONS USING 286/10 SINGLE BOARD COMPUTERS

The University of Arizona M.S. 1986

A MULTIPROCESSOR KERNEL AND MONITOR FOR IMAGE PROCESSING APPLICATIONS USING 286/10 SINGLE BOARD COMPUTERS

by

Timothy James Kennedy

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 8 6

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements of an advanced degree at The Univerity of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Timothy J. Kennedy

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Ralph Martinez

Ralph Martinez
Associate Professor of
Electrical and Computer Engineering

24 October 86
DATE

ACKNOWLEDGMENTS

The author wishes to express his appreciation to Mr. Perry Lindberg of General Dynamics Corporation, Convair Division, for support of this work. Further thanks are due to General Dynamics Corporation for financial support which made this work possible (contract P.O. 46-96344-02). Special thanks are due to Mr. Michael Patrick and Mr. Bruce Nicholson of General Dynamics for their valuable help. Further gratitude is extended to the principal investigators of the project, Dr. Ralph Martinez and Dr. Robin N. Strickland, for their guidance. The author would also like to thank Dr. Donald L. Shirer for his assistance. Thanks are also due to the other Computer Engineering Research Laboratory (CERL) students who participated on the project.

TABLE OF CONTENTS

TABLE OF CONTENTS--Continued

TABLE OF CONTENTS--Continued

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The development of a monitor and multiprocessor kernel, designed to execute concurrent image processing algorithms on tightly-coupled Intel 286/10 single-board computer systems, is described. In this thesis the structure, development, and execution of the monitor are discussed. The monitor is organized into five functional layers. The innermost layer consists of a synchronizing kernel, initialization code, an exception handler, and low-level I/O. The next layer contains level 1 I/O routines. After this is a standard I/O library for C application code support. The application code and commands for executing and debugging the application code are contained in the fourth layer. Finally, an interactive command interpreter is provided for running the monitor. The monitor is configured as a firmware component, residing in EPROM on the 286/10 board. The kernel is optimized for use with concurrent image processing algorithms. Experiments describing the execution of the kernel, including a statistical edge operator, are presented.

CHAPTER 1

INTRODUCTION

1.1 Statement of Problem

The new generation of 16 bit microprocessors, such as the Intel 80286 and Motorola 68000, contain features which allow multiprocessor synchronization, which were not supported in the previous 8 bit families [1]. Popular microprocessor system busses (Multibus, VMEbus) contain mechanisms for multiple bus master priority resolution and bus arbitration, which allow a multiprocessor system to be carried out. However, operating system components which take advantage of multiprocessing are scarce, because interprocessor communication hardware mechanisms and protocols are not standardized, and because multiple processor systems are application dependent, and cannot be generalized efficiently. An example of this is the Intel MMX 800 message exchange software package [2], which supports the Multibus. This product allows the user to code a message passing protocol, which is written as a task. The task is then configured as a resident part of the RMX86 real-time operating system [3], [4]. This may be a good solution for some systems, but this requires the RMX86 operating system to be resident, which occupies a large amount of system read only memory (RAM), typically 256k bytes, which subtracts usable RAM from the application code. Further, RMX86 cannot be configured entirely for read

only memory (ROM), some parts must be read in from an external device at boot up [5].

Image processing algorithms can become large and require excessive execution times. Consider the task of developing temporal processing algorithms. Given a "real time" image rate of 30 frames/second, even a few minutes video footage could take hours to process. Multiprocessing is a known method to increase processing throughput. For a system consisting of N homogeneous processors, an ideal speed up factor approaching N can occur, depending upon how much operating system overhead and interprocessor communication is required.

1.1 Objective of Thesis

This thesis addresses the design and development of a specialized multiprocessor kernel system and monitor, called "M286". This will be referred to as the "system monitor" or "M286 monitor". The purpose of this monitor is to support a homogeneous processor environment, consisting of Intel 286/10 single board computers (SBCs). These SBCs are tightly coupled and communicate through the Multibus system bus, through dual-port memory. The monitor is designed for executing image processing algorithms, using Imaging Technology IP-512 family image processing Multibus modules.

Image processing algorithms were developed on an Intel System 286/380, running the RMX86 operating system, written in the C programming language [5]. The algorithms were then partitioned to run as concurrent tasks, with each task executing on a separate SBC. The

application code tasks was compiled and the object code was downloaded to target SBCs, which were configured with the M286 monitor in ROM.

The M286 monitor approach has several advantages. Since it resides entirely in ROM (while using a small amount of system RAM), it frees up the system RAM, leaving the RAM for the application code to use. Also, the monitor supports the C programming environment, and contains the C standard I/O library, further increasing memory efficiency. The monitor also provides an efficient synchronizing kernel and a set of commands to execute and test the application code. Further, the monitor itself was written in C, except for several hardware dependent routines, contained in the first layer, which were written in 80286 assembly language. The monitor should be portable, requiring only that the assembly routines be rewritten. This feature is useful if later system requirements necessitate integration of other processor boards.

Not all systems containing several processors are considered multiprocessor systems. Classifying a multiple processor system into a multiprocessor system can sometimes be difficult [7]. The proposed system, consisting of two or more single board computers, can be considered as a multiprocessor system for the following reasons. First, the processors boards on the system are identical, or homogeneous. The processor boards will communicate through dual-port shared memory, with each processor accessing system programs through private local memory. The local program memory is used to increase processor efficiency. Every processor will have equal access to the bus resources, which consist of the Itex processing boards. Finally, the processors are tied together

with an identical operating system component, namely the M286 monitor itself.

1.3 Approach

This thesis discusses the requirements, structure, development, and execution of the monitor. The hardware environment which the M286 monitor is to support is described in Chapter 2. This discusses the Multibus, the Intel 286/10 architecture, and multiprocessor synchronization implementation. Chapter 3 describes the logical organization and services provided by the system monitor and kernel. Chapter 4 concentrates on the monitor development, including the development environment, ROM code development, and executing the monitor. Applications of the monitor and test results are described in Chapter 5. Finally, Chapter 6 summarizes the current work and gives suggestions for future work.

CHAPTER 2

MULTIPROCESSOR HARDWARE ENVIRONMENT

The algorithm development system (ADS) is based on the Intel 286/380 microprocessor development system [8]. This is an "open system", based on the IEEE standard 795 Multibus system bus. The 286/380 system consists of two separate chassis, which are a processor chassis and a peripheral chassis. The basic processor chassis hardware configuration consists of the Intel 286/10 single board computer (SBC), a 512 kilo-byte memory board, and a disk controller board. The peripheral chassis contains a 35 mega-byte Winchester disk and a 1 mega-byte floppy diskette. Added to the processor chassis are six Imaging Technology Multibus family boards, which are used as used for high performance image processing. A Pioneer LD-V6000 laser disk player and Pulnix Corporation TM-540 charged-coupled device (CCD) camera are used for video sources, and a color video monitor is used to view output images. A block diagram of the ADS is shown in Figure 2.1.

This section describes the multiprocessor hardware configuration of the ADS, which will execute the M286 monitor. The architecture of the 286/10 SBC is discussed, along with the Multibus interface and peripheral chips. The Imaging Technology IP-512 family boards and the ADS configuration are described. Finally, the multiprocessor synchronization mechanism which will be implemented by the monitor is

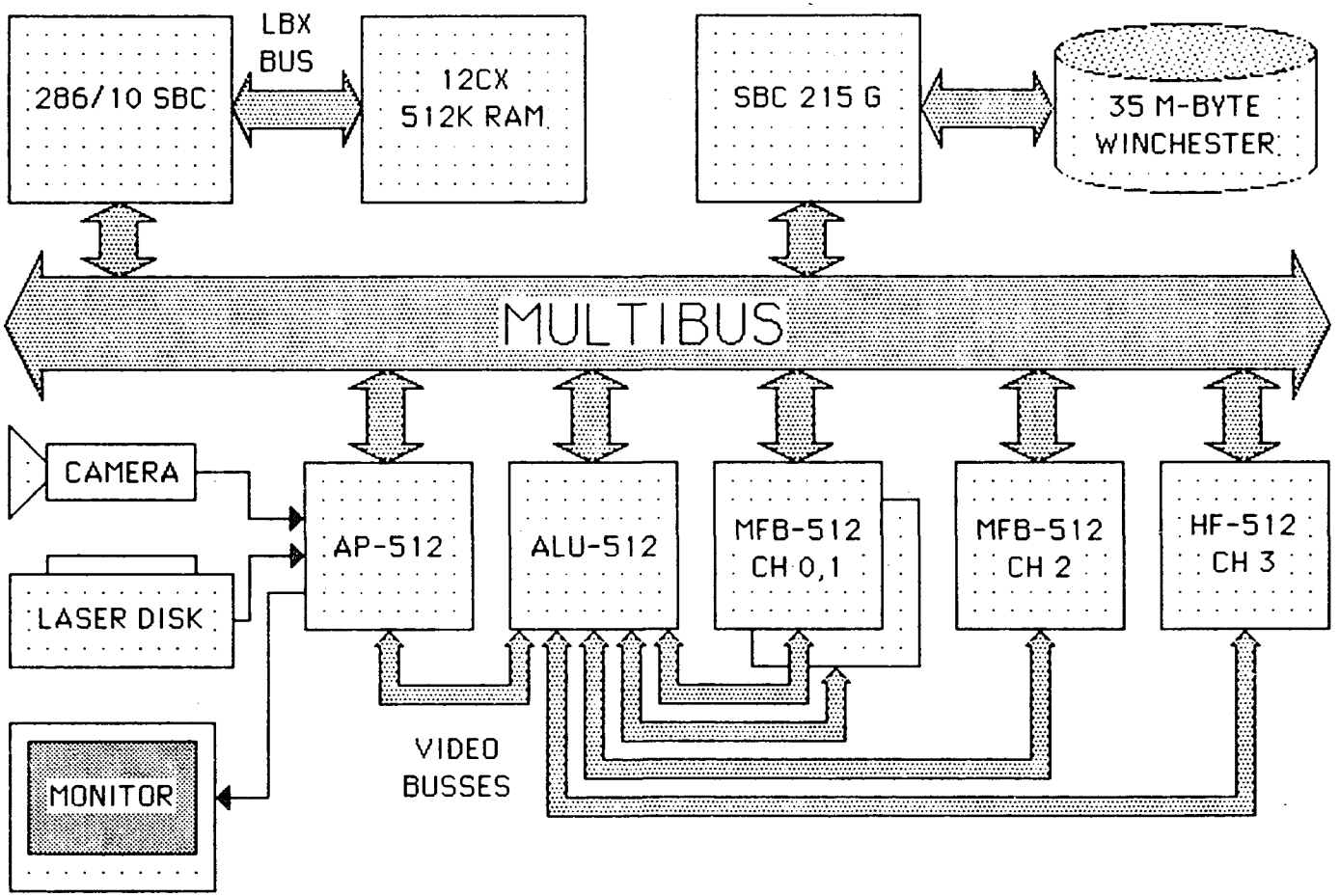


Figure 2 1 Algorithm Development System Block Diagram

presented.

2.1 Target SBC Architecture

The Intel 286/10 single board computer [9] is selected a the target processor. This is a Multibus based board. It uses Intel 80286 CPU, which can operate in the real or protected mode. In real mode, the CPU can access up to one megabyte of memory; in protected mode, it can access up to 16 megabytes of real memory and one gigabyte of virtual memory. The board also features a local bus extension for dedicated program memory (LBX bus), configurable on-board memory, dual-port memory, parallel and serial I/O, and optional numeric processor extension capability. A block diagram of the 286/10 SBC is shown in Figure 2.2.

2.1.1 Multibus Interface

The Multibus system bus is made up of five groups of busses and control signals [10]. These consists of 24 address lines (16 Mega bytes of address space), and supports 8 and 16 bit data transfers. It also supports 64 kilo bytes of I/O addressing through an 8 or 16 bit I/O bus. The Multibus also contains command, acknowledge, and interrupt signals.

The Multibus was adopted by IEEE Standard Committee as IEEE-796 (1980). IEEE notation will be used to refer to Multibus signal names. A slash ("/") appended to a signal name designates the signal as asserted low. Signal lines are numbered in decimal.

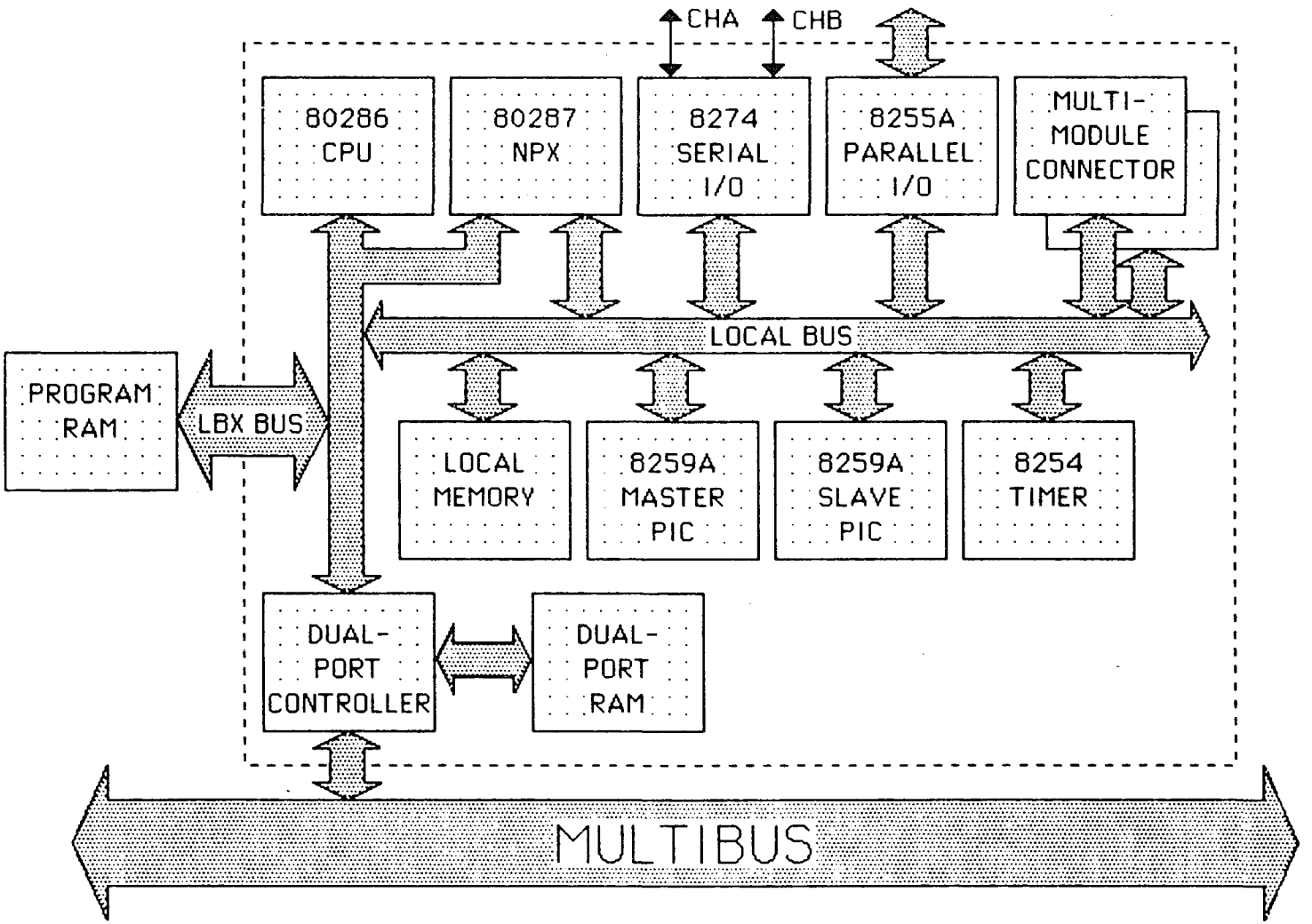


Figure 2.2 286/10 Single Board Computer Block Diagram

2.1.1.1 Multibus Overview

The Multibus contains a 24 line address bus, designated ADR0/-ADR23/, and a 16 line data bus, DAT0/-DAT15/. The Multibus is capable of making both byte (8 bit) and word (16 bit) data transfers. Word transfers must occur at even addresses, while byte transfers may occur at even or odd addresses. The byte high enable line, BHEN/, controls the high/low byte data access.

The Multibus supports three types of bus devices, which are bus masters, bus slaves, and hybrid modules. A bus master is any module that can control the bus and initiate bus transfers. A bus master can make a bus transfer by first requesting control of the bus through exchange logic. After the bus is granted, it can perform data transfers by driving the command and data lines. The Multibus can support up to 16 bus masters. A bus slave is any module that can respond to bus commands generated by the bus master. The command protocol is asynchronous and interlocked; all bus cycles require a positive acknowledgement from a bus slave before a bus master can continue. A hybrid module has all the attributes of a bus master and some of the attributes of a bus slave. In other words, it looks like a bus slave to other masters on the bus, but may also initiate bus transfers itself.

The the control line LOCK/ signal is used by a bus master to provide for mutual exclusion of the bus. This signal will guarantee that no other bus device can access a resource until the bus master has finished using it. In systems with multiple processors, this establishes a method for microprocessors to communicate with each other (usually,

through shared memory).

The 286/10 board complies with the following Multibus specifications. It supports master operation on the Multibus interface. It supports an 8 and 16 bit data path, with a 24 bit address. It has an 8 or 16 bit I/O address (or can access I/O locally on the board). It also supports two-cycle bus vector interrupt requests, with either level or edge-level triggering.

2.1.1.2 Multibus Arbitration Schemes

Multiple microprocessors must be able to share global resources. Only one processor must be allowed to control the bus at any given time. The Multibus system bus supports multiple bus masters with a hardware arbitration scheme. The Multibus supports two types of bus arbitration methods - serial and parallel. The bus arbitration schemes are carried out through four signals. These are Bus Busy (BUSY/), Bus Priority In (BPRN/), Bus Priority Out (BPRO/), and Common Bus Request (CBRQ/).

The serial and parallel bus arbitration schemes are implemented through the BPRN/ and BPRO/ signals. BPRN/ indicates to a particular bus master that it has the highest priority request for the bus. The bus master may access the bus only if it has BPRN/ asserted. Bus priority out (BPRO/) is used for serial or daisy chain arbitration. In the serial arbitration scheme, the BPRO/ of the highest priority master is passed to the BPRN/ of the next priority master, and so on. Serial arbitration is fast, since the arbitration logic is simple. However, since arbitration must be carried out in one Multibus clock cycle, the

propagation delay through the gates limits the number of masters to 3. Also, if one board in the arbitration chain fails, successive processors will be removed from the arbitration path.

Parallel arbitration is carried out by sending all BPRN/ signals to an arbitration logic unit, which can detect bus requests simultaneously, and arbitrate and then grant the processors access to the bus through the BPRQ/ signal. Parallel arbitration solves some of problems with serial arbitration, although logic to carry out this is more complex and costly. The BPRN/, BPRO/ and BPRQ/ signals on the Multibus backplane are not bussed, but are wired according to the arbitration scheme selected.

The Bus Busy (BUSY/) signal indicates whether the bus is being used by another master. The common bus request (CBRQ/) indicates to the bus master in control of the bus that no other masters are requesting the bus. This allows a master to hold the bus and eliminate the bus exchange overhead.

2.1.2 LBX Bus

Program memory is provided to the 286/10 SBC through a dedicated bus, called the local bus extension (LBX). This is interfaced through the P2 connector on the Multibus. Through this bus the 80286 CPU can access up to 16 megabytes of memory. The advantage of providing the SBC with its own dedicated memory allows bus traffic to be removed from the Multibus system bus. This allows more processing bandwidth on the system bus, which is required for a multiple-processor, real-time system.

2.1.3 On-Board Local Memory

The 286/10 board contains two independent address areas for adding local memory. These are socket pair U40/U75, which must contain an EPROM set to hold boot code. Socket pair U41/U76 is configurable, and may contain EPROM or static RAM.

2.1.4 Dual Port Memory

The 286/10 SBC contains dual port memory, accessible independently from the on-board processor and by other bus masters on the Multibus. The on-board CPU has priority in simultaneous access to the dual port RAM. The dual port memory is user configurable, which consists of the dual port local starting address, block size, and the Multibus starting address. The dual port RAM consists of static 8-bit wide family devices, which are configured in pairs, for 16 bit read/write transfers. The 286/10 board contains socket pairs U54/U87 and U53/U86 to accept the dual port memory.

2.1.5 Local Bus and Peripheral Chips

The 286/10 SBC architecture contains a local bus for accessing processor resources. Contained of the local bus are several peripherals. These include the 8274 multiple protocol serial controller (MPSC), which provides two channels of serial I/O, the 8255A programmable peripheral interface (PPI), which provides parallel I/O, and can be configured for Centronics-compatible parallel interface. The local bus also contains two cascaded programmable interrupt controllers (PICs), which process

the local and Multibus bus-vectored interrupts. The 8254A programmable interrupt timer (PIT) provides three independent timer channel sources. An optional 80287 numeric processor is available, which provides high speed floating point processing. In addition, the 286/10 SBC provides two additional connectors on the local bus (SBX connectors), to accept additional hardware modules.

2.2 12CX RAM Board

The 12CX random access memory board [11] is used by the ADS to provide program memory for the 286/10 processor boards. The 12CX board is a Multibus compatible board, and provides up to 512-K bytes of memory. The 12CX contains an LBX bus interface, and provides for dual-port access, from both the Multibus and the LBX bus. The 12CX board incorporates error correction and checking circuitry (ECC) to detect memory errors and optionally correct the errors. The board uses 64 K-byte dynamic RAM devices.

2.2.1 ECC Operation

The purpose of the ECC circuit is to correct any data errors which may occur during the read operation. ECC operations are carried out with the Intel 8206 error correction and detection unit. This chip performs error correction by using a modified Hamming code to generate and store a 6-bit check word for each 16-bit data word. The Hamming code allows correction of one erroneous bit, which can be located and corrected in correcting mode, and detection of multiple bit errors. When the word is read back, a new check word is computed and compared with

the stored check word. If the two are not identical, at least one error has occurred, and the syndrome of the data and check word can be computed, which reveals if one or more bit errors occurred, and the position of the offending bit, to allow the word to be corrected.

The ECC circuitry contains several modes of operation. This consists of a correcting mode, where a single erroneous bit is located and fixed, and a non-correcting mode, where one or more bit errors are detected only. In the event of either a non-correctable error or any error, the 12CX board can be programmed to interrupt the host processor through the Multibus. The ECC also contains a diagnostic mode to test the ECC circuitry. Control and programming of the ECC functions are accomplished through two 8 bit registers, which are the control status register (CSR) and the error status register (ESR). These registers are accessed through one 8-bit I/O port from the Multibus.

2.3 Imaging Technology IP-512 Family Description

To support high speed image processing, the 286/310 ADS is configured with Imaging Technology IP-512 image processing modules [12]. These include one AP-512 analog processor for digitizing video input, one ALU-512 arithmetic logic unit which performs real-time arithmetic operations, three MFB-512 memory mapped frame buffers for storing image frames, and one HF-512 histogram/feature extraction board. This section gives an overview of the IP-512 family boards, and the configuration used in the ADS.

2.3.1 IP-512 Family Multibus Boards

The IP-512 image processing modules are versatile "building block" modules that can be configured for a wide range of image processing applications. The IP-512 system is centered around the ALU-512 board. This contains a custom high-speed video bus, composed of five independent 8-bit busses, which the ALU uses to read and write to the analog processor, frame buffers or histogram board. The IP-512 system boards are programmed and controlled by the application programs through a group of registers which are memory mapped or I/O mapped on the Multibus.

The purpose of the AP-512 analog processor is to convert input images, such as those from a video camera, into 8-bit digital data. The AP-512 also converts digital data, back to analog, for display on a video monitor. The IP-512 supports the EIA RS-170 analog video. The analog board also contains groups of look up tables (LUTs), for performing linear pixel input and output transformations. Each LUT contains four software selectable subtables, consisting of 256 8-bit addresses each. One set of LUTs are provided on the input channel, and three LUTs lut are provided on the output channel, one each for driving the red, green and blue drivers of a color monitor. This allows the AP-512 to be programmed for doing pseudo-color displays.

The ALU-512 is a five stage pipeline processor, which operates on live video input or full frame buffers. One ALU-512 operation is done in one frame time (1/30 second). ALU-512 stages consist of input operand selection, consisting of video channels or constants, an 8x8 multiplier,

an ALU stage consisting of addition, subtraction, and logical bitwise operations. The ALU stage is followed by a 16-bit barrel shifter, which finally goes to output selection. Each ALU-512 operation produces a delay, which equates to 11 pixel times. This results in a displacement of frame buffer pixels by 11 positions from its original position, before a ALU-512 operation. The frame buffers contain a pan and scroll register, which are adjusted after each ALU operation, which compensate for this offset. The AP board displays frame buffer data, adjusted by its pan and scroll registers.

The MFB-512 board is used to store high-resolution images, consisting of 512 rows by 512 columns of 8-bit pixels per board. The MFB-512 is a memory mapped frame buffer, which allows a block of pixels to be accessed through a fixed window on the Multibus. The pixel window shape and size can be varied, with hardware jumpers. If the pixel window is less than the size of the frame buffer, a set of x and y offset registers are provided to allow the pixel block to the access the entire 512x512 frame. Frame buffers are accessed from the Multibus or video bus as 8-bit pixels, or may be combined into high/low pairs, accessed as 16-bit pixels, for high precision arithmetic operations.

The HF-512 module provides feature extraction and histogram computation of an image, in real-time. This is done by counting image features as an image is being acquired through the video bus. Results are stored as x, y coordinates in a 4096 x 18-bit output buffer. The output buffer can be read through Multibus registers or can be directly

memory mapped on the Multibus. The HF-512 also contains an input LUT, composed of a 16-bit input and output data bus.

2.3.2 ADS IP-512 Configuration

The current ADS configuration consists of an AP-512 analog board, an ALU-512, three MFB-512 frame buffers, and a HF-512 histogram/feature processor (as shown in Figure 2.1). Video bus channels 0-2 are connected to the frame buffers, and channel 3 is used by the histogram board. In addition, frame buffers channels 0 and 1 are configured as a hi/lo pair, accessible as two 8-bit frame buffers, or as one 16-bit deep frame buffer from the Multibus.

The IP-512 group is memory mapped on the Multibus for access by the 286/10 SBC. The 286/10 board contains 48 k-bytes of Multibus address space, beginning at address E0000H. The first 16k-bytes of Multibus space is reserved for dual-port RAM for 286/10 SBC use. The IP-512 group registers were then located at E4000H through E4060H. The MFB-512 pixel memory blocks were fixed at 512 pixel in the x dimension and 16 pixel in the y dimension (8 k-bytes total). The pixel block for MFB-512 hi/lo pair is set at E8000H, and the pixel block for the third MFB begins at EA000H. A memory map of the IP-512 group, showing the register locations, is shown in Figure 2.3.

2.4 Multiprocessor Synchronization

Since bus masters on the system will require communication with each other, a mechanism for interprocessor communication is needed. Further, this interprocessor communication must be asynchronous. A

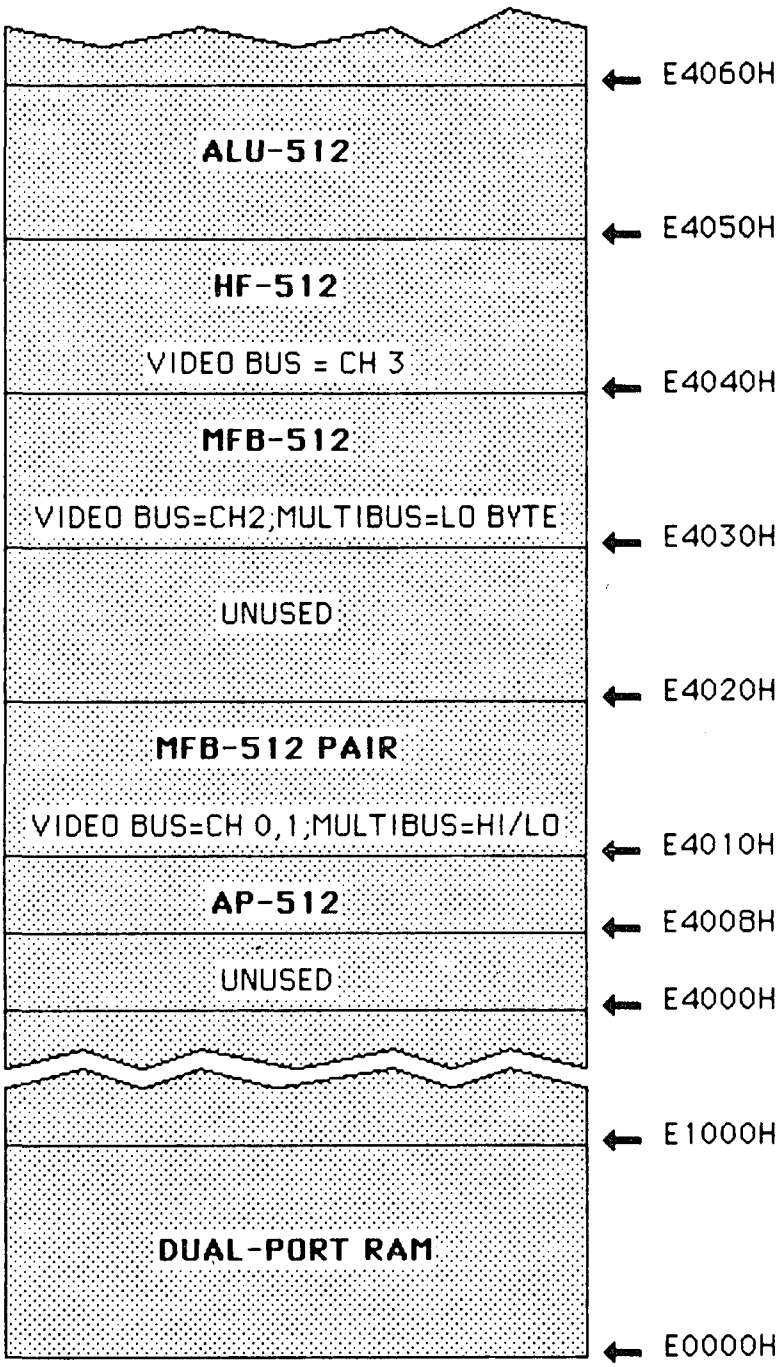


Figure 2.3 IP-512 Control Register Interface Memory Map

binary form of the semaphore wait and signal routines can be carried out through shared memory [13]. This is done by having a processor wait by setting a shared memory location with a non-zero value, and reading it back. This is called the test-and-set operation. This process continues until the synchronizing processor writes a zero value back to the shared location, to signal that the second processor is ready.

The test-and-set operation, which consists of a separate read and write to the semaphore memory location, must be indivisible to the second processor. This indivisible read and write operation is required because of the following disastrous scenario. Suppose a resource is being shared by three masters on the Multibus. The first processor has access to a resource, then writes a zero to the semaphore, indicating that the shared resource is available. The second master could then read the semaphore as free, but before it can claim the resource (by writing back a 1), a third master could grab the bus and read the resource as free also. Both masters would try to access the resource.

The 80286 microprocessor provides a mechanism to carry out an indivisible test-and-set operation, by asserting the Multibus LOCK/ line. This is done in assembler by prefixing an instruction with the "lock" keyword. The test-and-set operation is implemented using the "xchg" instruction, as follows:

```
lds     di, dword ptr [bp+06h]  ;Get semaphore pointer off stack
mov     ax, 1                   ;Set ax to 1 (implies locked)
lock    xchg byte ptr [di], al  ;Test and set lock
```

The result is stored in the AX return register. A return value of 1 implies the resource is busy. A return value of 0 means the calling processor owns the resource, and another processor cannot access it. It is then the responsibility of the calling processor to release the resource when it is finished, by writing a 0 to the semaphore address.

CHAPTER 3

M286 MONITOR STRUCTURE

The M286 monitor is organized into functional layers. Each layer provides a class of functions. The layers are organized hierarchically; only a higher layer may make a call to a lower layer, but a lower layer may not call any higher layers. A higher layer typically makes calls only to its next adjacent (lower) layer, but may skip layers, for efficiency. All calls will ultimately reach the 286/10 SBC hardware interface. A block diagram of the monitors functional layers are shown in Figure 3.1

In this section the five functional layers of the monitor are described. These include the innermost layer, which consists of a synchronizing kernel, initialization, low-level I/O drivers, and other hardware dependent drivers. The next layer contains level 1 I/O routines. After this is a standard I/O library for C application code support. The application code and monitor commands for executing and testing the application code reside in layer four. Finally, an interactive command interpreter is provided for running the monitor.

3.1 Inner Layer

The innermost monitor layer is made up a of several hardware dependent drivers, for communicating to or from the 286/10 SBC interface. Many of these routines are written in 80286 assembly

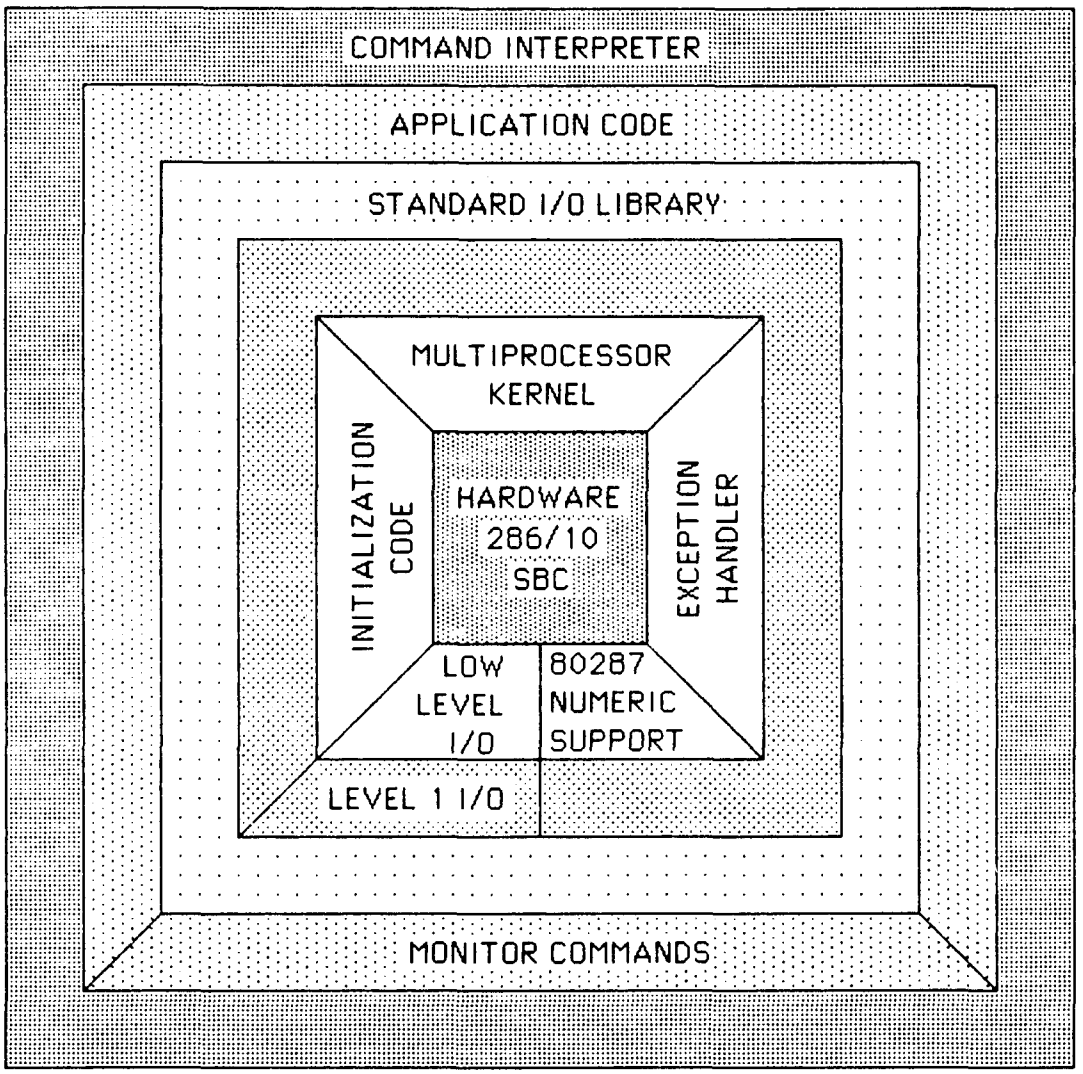


Figure 3.1 Structure of the M286 Monitor

language [14]. These routines provide several services. Included at this level are a synchronizing kernel, 286/10 board initialization code, the system exception handler, 80287 numeric processor support, and low level serial I/O drivers.

3.1.1 Synchronizing Kernel

The 286/10 SBC provides for interprocessor communication through dual-port memory, accessed locally on the 286/10 board and through the Multibus. The kernel provides system calls for interprocessor synchronization, through test-and-set semaphores. Also provided are routines for wait and signal operations. These routines are "tas", for performing a test and set operation, and "mp_signal" and "mp_wait" for the wait and signal operations, respectively. See Appendix A for a description of these routines.

The mp_signal and mp_wait routines can be used to carry out several multiprocessor functions, including sharing of processor resources, processor synchronization, and inter-processor data transfer. The following sections describe these applications.

3.1.1.1 Sharing Resources

Semaphores can be used to arbitrate the use of a resource among several processors. This is done by assigning one semaphore to a resource. A shared resource is usually another Multibus module, such as a disk controller or an Itex board. The semaphore must be initially cleared to zero, to indicate that the resource is free. Whenever the

processor needs the resource, it makes an mp_wait system call. If the resource is being used by another processor, the first processor will have to wait until the semaphore becomes ready. Next, when the processor is done with the resource, it must to an mp_signal call to release to other processors. A block diagram showing the use of semaphores to share resources is shown in Figure 3.2.

3.1.1.2 Processor Synchronization

Occasionally, processors need to synchronize, to allow them to work in parallel and share data. A method to allow two processors to synchronize can be done using two semaphores, as shown in Figure 3.3. Here, two processors, designated P1 and P2, use two semaphores, designated SEM0 and SEM1. Initially, processor P1 owns SEM0 and processor P2 owns SEM1. In order to synchronize, processor P1 signals (releases) SEM0, then waits for SEM1 to become ready, while processor P2 signals for SEM1 and then waits for SEM0. This scheme forces the two processors to wait until each has done the signal operation before they can continue.

3.1.1.3 Data Transfer Via Mailbox

Data can be transferred between two processors, by using a method similar to the synchronization scheme, described in the previous section. As shown in Figure 3.4, two processors, designated P1 and P2, use two semaphores, designated SEM0 and SEM1. Initially, processor P1, the sender, owns SEM0 and processor P2, the receiver, owns SEM1. The first processor, P1, writes data to a shared memory location, which is

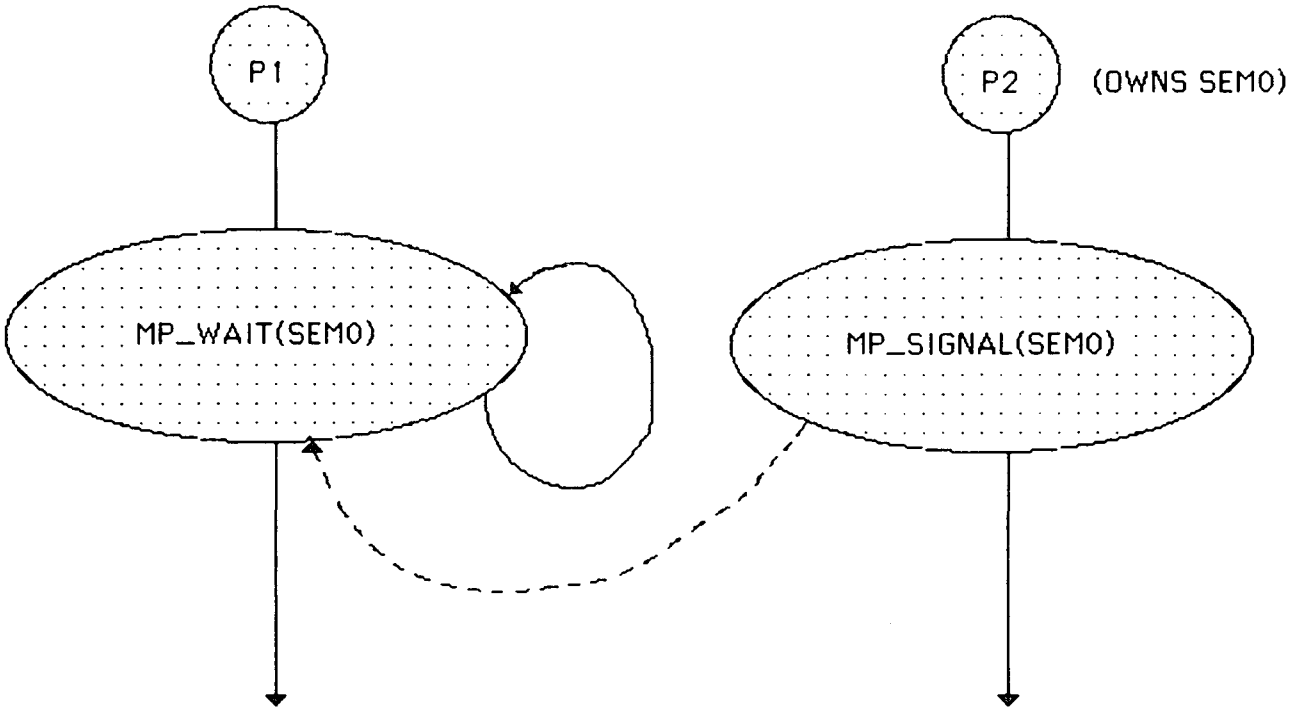


Figure 3.2 Sharing Resources

used as a mailbox. Processor P1 then signals that the mailbox is valid, through SEM0. The second processor, P2, which has been waiting for SEM0, will now be allowed to read the mailbox. After this, processor P2 signals back to processor P1 that it has finished reading the mail, through SEM1. Processor P1 is now free to use the mailbox again to send more data.

3.1.2 Initialization Code

Upon reset of the 286/10 SBC, the processor code segment (CS) and instruction pointer (IP) are set to CS:IP = FFFF:0000. Here, a long jump is performed to the 286/10 SBC initialization code. The initialization sequence begins by clearing the interrupt enable flag. Following this, the MPSC, PIC, and PIT are initialized. A full system RAM test is performed, which will be described. The system exception handler and interrupt vectors are loaded next. The 80287 status of the numeric processor checked, and initialized if resident, or numeric processor emulation if not resident. The C run time stack is set up, followed by enabling of interrupts. Last, a call is made to the M286 command interpreter, to begin accepting user commands. A flow chart of the initialization process is shown in Figure 3.4.

All peripheral chips are programmed by reading and writing through the I/O bus. An important consideration when programming the peripheral chips is to allow for proper set up/hold time between successive writes to the same device. To insure this set up time is observed, a delay loop follows every write to a peripheral chip. A

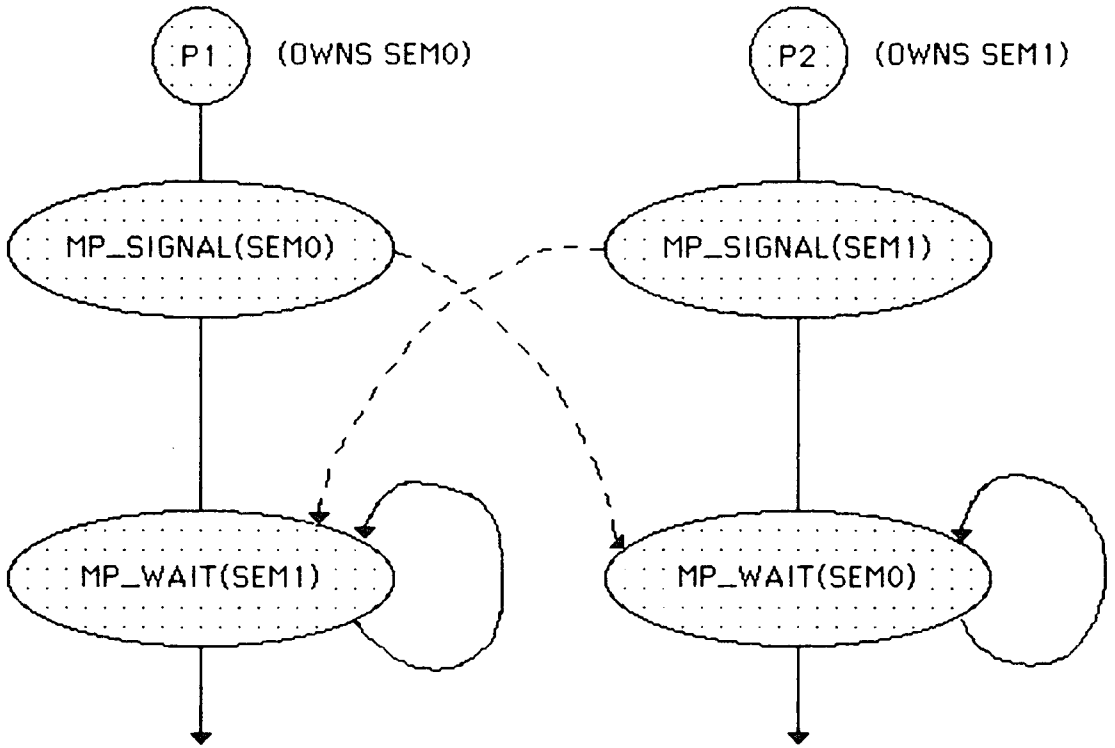


Figure 3.3 Processor Synchronization

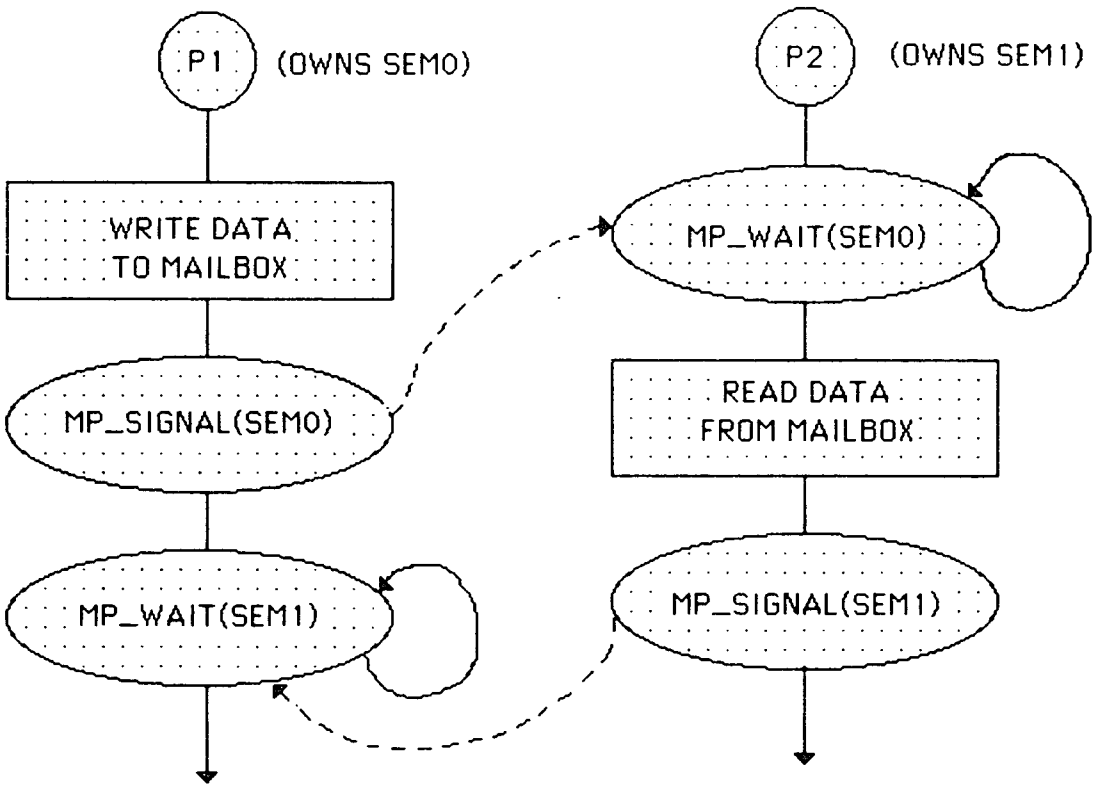


Figure 3.4 Data Transfer Via Mailbox

conservative delay, lasting 7.67 microseconds is used.

3.1.2.1 MPSC Initialization

The 8274 multiple protocol serial controller (MPSC) provides two channels of serial I/O, channels A and B. The MPSC is initialized to operate in asynchronous mode, for interfacing to RS-232 equipment. Channel B is designated for console I/O with the monitor, while channel A is used to interface with the host system for uploading code. Both channels are reset at 9600 baud, 8 bits per character, parity disabled.

3.1.2.2 PIT Initialization

The 8259A PIT consists of three independent counters, CNT 0-2. As the SBC is configured, timer channels 1 and 2 are used to generate clocks for channels A and B of the 8274 MPSC, respectively. These are both driven by a common 1.23 MHz frequency reference. Channels 1 and 2 are set in PIT mode 3, which produces a square wave for driving channels A and B of the MPSC, respectively.

3.1.2.3 PIC Initialization

The 286/10 SBC contains a master and a slave programmable interrupt controller to detect and resolve priority of interrupt sources. The 8259A consists of 8 interrupt input lines, IR0-IR7, and can pass interrupt vector numbers to the 80826 processor through an 8 bit bus. The 286/10 SBC allows versatile interrupt configurations through an interrupt source jumper matrix.

As jumpered, CNT0 from channel 0 of the PIT is connected to IR0

Table 3.1 8259A Master and Slave PIC Initialization

MASTER PIC
8086 edge mode 4 bytes per interrupt level Interrupt vector for 8259A begins at location 80H Fully nested Edge-triggered Buffered operation (jumper) Normal end-of-interrupt Slave PIC on master level 7 (jumper) Level 0 has highest priority IR0 interrupt level enabled (timer interrupt)
SLAVE PIC
8086 edge mode 4 bytes per interrupt level Interrupt vector for 8259A begins at location 81H Fully nested Edge-triggered Buffered operation (jumper) Normal end-of-interrupt Slave PIC on master level 7 (jumper) Level 0 has highest priority All interrupt levels are masked

of the master PIC, IR1-IR5 are connected to INT1-INT5 Multibus interrupt lines, respectively, 8274 serial interrupt is connected to IR6, and the slave PIC is connected to IR7. Currently, only CNT0 interrupt is used, and the remaining interrupts are masked off. Table 3.1 gives a list of the master and slave initialization parameters.

3.1.2.4 Memory Test

Testing the system RAM is a challenging problem, due to several reasons. First, it is time consuming to exhaustively test the RAM, so a group of shorter confidence tests must be developed [15]. Another consideration is that the memory test itself be implemented using no RAM itself, using only registers. This allows the memory test to run in the absence of failure of RAM. Next, the amount of RAM installed on the system may vary, but is usually contiguous. Finally, the problem remains of what action to take in the event of a RAM test failure.

The approach taken by the M286 monitor is to test the first 64k byte block RAM. RAM is tested by words, consisting of three separate tests. First, stuck bits are tested. This is done by writing a zero to the location (which is read back for verification), then a pattern of all ones are written and then read back, for verification. Next, addressing faults are tested. This is done by writing the lower 16 bits of the memory address to the word, and reading this back. This test helps verify that address lines are decoded properly for both the read and write operations. Finally, a pattern is read and written to the word. This is done partly to initialize RAM to a fixed pattern. If a

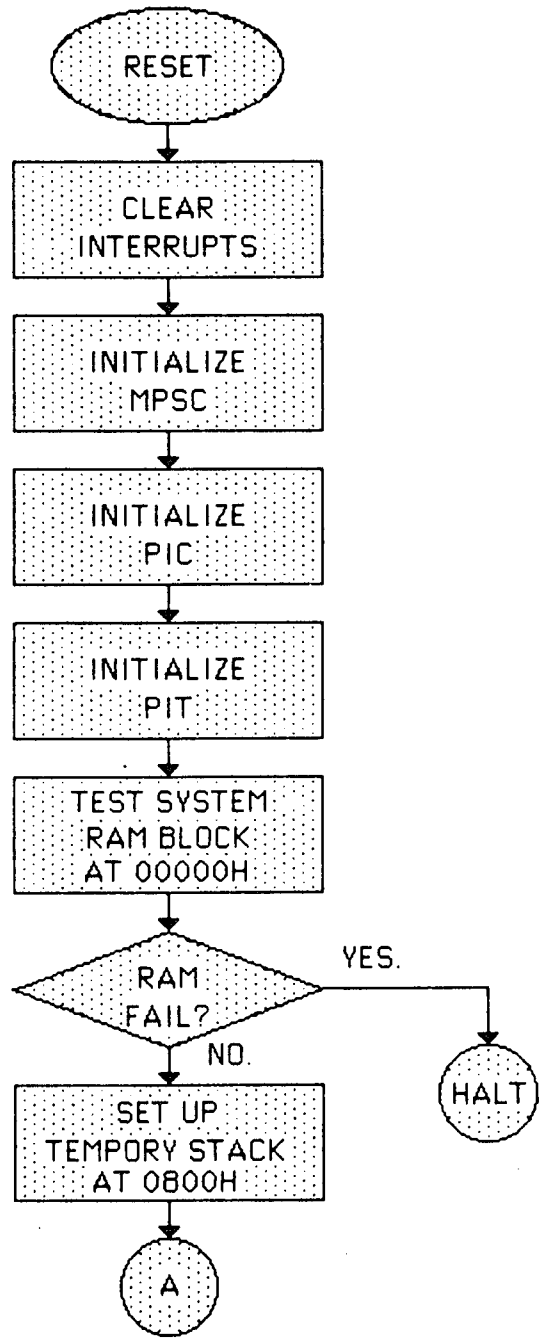


Figure 3.5 M286 Initialization Process

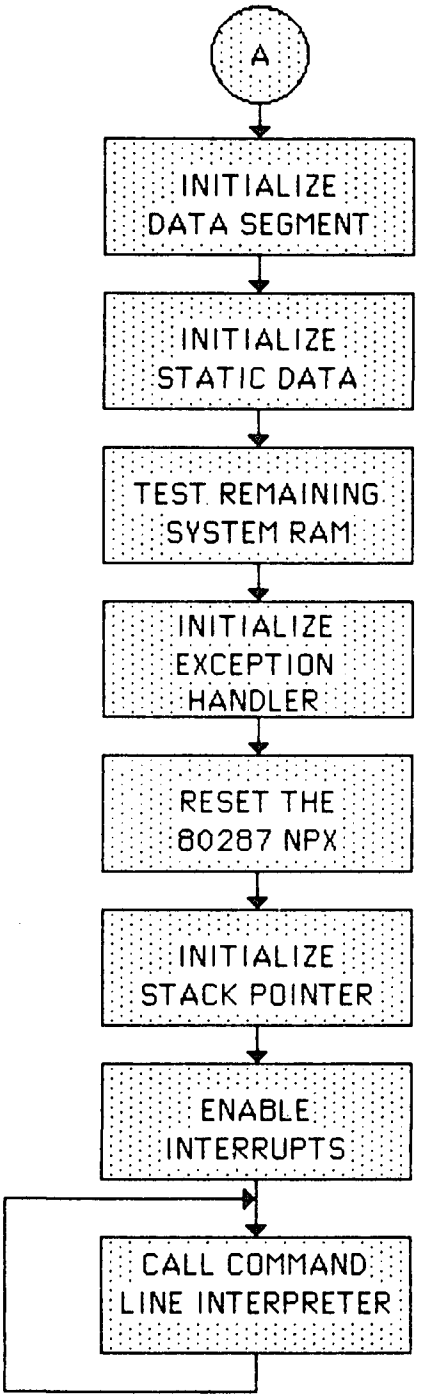


Figure 3.5-- Continued

test fails within this first RAM block, the processor is halted. The memory test is implemented using registers only.

After successfully testing the first 64k block of RAM, a temporary stack is set up within this area. Monitor data is initialized, so to allow execution of the C I/O library routines. The remaining system RAM is tested, and status of the tests are echoed to the console. This continues until a failure occurs, since the upper limit of RAM is not known. The user will then have confidence that the system memory works and is configured properly.

3.1.3 Exception Handler

Since internal errors may occur on a microprocessor while executing a program, the 80286 CPU provides dedicated interrupt vectors to handle many exceptional conditions. The kernel provides an exception handler which will respond to these exceptional runtime conditions. When an exception occurs, the handler will print first the address where the exception was raised, followed by a diagnostic message to the console, then returns to the command interpreter. In addition to the dedicated system interrupt vectors (0-16), all reserved interrupt vectors (17-31) and unused user vectors (32-255) are assigned to this handler, to provide protection if a spurious exception occurs. Table 3.2 gives a list of the interrupt types and their corresponding handler message.

3.1.4 80287 Numeric Processor Support

The 286/10 single board computer provides for an optional 80287 floating point processor. Alternately, an 80287 software emulation

Table 3.2 Exception Types and Error Messages

EXCEPTION TYPE	HANDLER MESSAGE
0	<XXXX:XXXX> DIVIDE BY ZERO.
1	<XXXX:XXXX> SINGLE STEP.
2	<XXXX:XXXX> NMI INTERRUPT.
3	<XXXX:XXXX> BREAKPOINT.
4	<XXXX:XXXX> OVERFLOW EXCEPTION.
5	<XXXX:XXXX> BOUND RANGE EXCEEDED.
6	<XXXX:XXXX> INVALID OP-CODE.
7	<XXXX:XXXX> PROCESSOR EXTENSION NOT AVAILABLE.
8	<XXXX:XXXX> INTERRUPT TABLE LIMIT TOO SMALL.
9	<XXXX:XXXX> PROCESSOR SEGMENT OVERRUN EXCEPTION.
13	<XXXX:XXXX> SEGMENT OVERRUN EXCEPTION.
16	<XXXX:XXXX> PROCESSOR EXTENSION ERROR.
10-12,14,15,17-31	<XXXX:XXXX> UNKNOWN INTERRUPT.

NOTE: XXXX:XXXX will be replaced with the code segment and instruction pointer (CS:IP) where the exception was raised.

library may be used if the 80287 is not available [16]. The monitor provides initialization routine for 80287. To do this, it first determines if an 80287 processor is present on the board. This is done by reading the 80287 status word. If it contains all zeros, the 80287 is not present. If the 80287 is present, the math present (MP) flag on the 80286 machines status word (MSW) is set, which allows numeric instructions to be executed on the 80287 NPX processor. Otherwise, the emulate mode (EM) bit on the 80286 MSW is set to permit emulation of 80287 instructions, through the processor exception not present exceptions. An 80287 emulation library is provided by the RMX system.

3.1.5 Low Level Serial I/O

Low level routines are provided to read and write to the serial I/O channels. These drivers use a polling method to transmit and receive characters through user supplied buffers. The drivers use the polling method to send and receive characters. Polling is done by reading, in a loop, the status/control words of the MPSC, until a character is received or can be sent. The polling drivers were used, instead of interrupt driven I/O, because it allows debugging of interrupt routines easier. However, the polling drivers are not ideal for interfacing to RS-232 equipment, since they cannot buffer input if the processor is running other operations.

In addition to the 8274 serial drivers, a mechanism to send and receive data from one SBC to another through the Multibus was developed. This was accomplished by using two semaphores and a one byte dual-port

RAM address as a mailbox, as described in Section 3.1.3.3. Drivers were written to send and receive data through a one byte mailbox, which use the mp_signal and mp_wait routines.

The low level I/O drivers are "tread" (for reading a port), "twrite" (for writing to a port), and "mread" and "mwrite", for reading and writing to a Multibus mailbox, respectively. All other serial I/O is built from these routines. A description and C calling sequence for these routines is given in Appendix A.

3.2 Level 1 I/O

The next monitor layer, are the level 1 I/O routines. These provide ASCII or binary interpretation the input and output streams. The modes supported are the ASCII mode, where carriage returns sent to the output stream out are expanded to carriage return (CR), linefeed (LF) pairs, and output is echoed back to the input stream, to support full duplex terminals. The other mode is the binary mode, which performs no special processing of the input or output stream.

The level 1 drivers consist of four routines: "tmode", which sets the port mode, "read", which reads into a user supplied buffer, and "write", which writes from a user supplied buffer. Finally, the routine "_editline" provides editing of the input buffer (supports delete key). Accessing these routines are described in Appendix A.

The M286 monitor supports seven pre-defined I/O channels. These are standard input (stdin), which reads the console port, standard output (stdout), and standard error (stderr), which writes to the

Table 3.3 M286 Standard I/O Library Routines

Routine Name	M286 Library Source
1) Character Classification	
isalnum	ctype.c, m86ctype.h*
isalpha	"
isascii	"
iscntrl	"
isdigit	"
islower	"
isprint	"
ispunct	"
isspace	"
isupper	"
isxdigit	"
2) String Manipulation	
strcat	strcat.c
strncat	strncat.c
strcmp	strcmp.c
strncmp	strncmp.c
strlen	strlen.c
strcpy	strcpy.c

Table 3.3--Continued

Routine Name	M286 Library Source
strncpy	strncpy.c
index	index.c
rindex	rindex.c
4) Byte-by-Byte I/O	
fgetc	fgetc.c
fputc	fputc.c
getchar	m86stdio.h*
getc	m86stdio.h*
putchar	m86stdio.h*
putc	m86stdio.h*
3) String I/O	
fgets	fgets.c
gets	gets.c
fputs	fputs.c
puts	m86stdio.h*
4) Formatted I/O	
printf	printf.c
fprintf	printf.c
sprintf	printf.c

Table 3.3--Continued

Routine Name	M286 Library Source
5) String Conversions	
atoi	atoi.c
atol	atoi.c
utoi	utoi.c
utol	utoi.c
xtoi	xtoi.c
xtol	xtoi.c
6) Odds and Ends	
abs	abs.c
setjump	m86lclib.lib
longjmp	m86lclib.lib

NOTE: * designates the routine is implemented as a macro.

console (MPSC channel B). Also supported are auxiliary input (auxin) and auxiliary output (auxout) channels, for reading and writing to the auxiliary serial port (MPSC channel A), respectively. Finally, two Multibus channels are declared, (mbox0 and mbox1) to allow SBCs to communicate through the Multibus. All channel designators (stdin, stdout, stderr, auxin, auxout, mbox0 and mbox1) are declared in C as type "FILE *". The application code developer must include the file special M286 header file "m86stdio.h", in place of the standard I/O header, to access the I/O streams.

3.3 Standard I/O Library

The application code is written in the C programming language. For support of the application code, a standard I/O library was written, as described in Chapter 7 of iC-86 Compiler User's Guide [17]. The library is included in ROM for use by both the monitor and application code. The library calls were written to be compatible with the library supplied with the Intel C compiler, although disk I/O routines are not supported, because they are not used. This high level I/O layer is built from the set of level 1 I/O routines. Several additional string manipulation utilities are also provided. A table of the standard I/O routines are listed in Table 3.3.

3.4 Application Layer and Monitor Commands

Executable program modules reside on the fourth layer. This includes several monitor commands, which reside in ROM, and the

application code, which is loaded in through the serial port into RAM. Several user commands are provided to execute and help debug the application program. Commands are included to load an application program into RAM from a host system, view and change memory, and execute the application program. Here is a brief description of the monitor commands:

1) boot - Reboot type system. Re-executes the initialization code.
2) display - Display an area of memory. Hex and ASCII field of the data are shown in two display fields.
3) fill - Fill a block of memory with data.
4) help - Give a brief command summary and their usage.
5) input - Input data from an I/O port.
6) load - Load and optionally execute an application program in Intel hex format.
7) mode - Set the serial port mode parameters.
8) memtest - Test one or more 64K byte blocks of RAM.
9) output - Output data to an I/O port.
10) register - Display 80286 registers and their contents.
11) run - Run the application program.
12) set - Set memory locations with data.

The monitor commands and their usage are given in Appendix B. In specifying command parameters, all addresses are given as an Intel segment and offset, both separated by a colon, and may contain up to

four hex digits, i.e. XXXX:XXXX, where X is a legal hex digit (0-9, A-F, or a-f). The command description contains the following notation. Optional parts of commands are enclosed in brackets ("[","]"). If one and only one parameter is required, it will be enclosed in braces ("{","}"). Options for the bracket or braces specifications will be separated by vertical bars ("|"). Two dots ("..") designate parameters may be repeated. All command entries are followed by a carriage return.

3.5 Command Line Interpreter

The outermost layer of the system monitor is the command interpreter. This provides a user interface to allow monitor commands to be invoked. The operation of the command interpreter is to parse the command line parameters into a vector of parameter strings, and an argument count. Parameters consist of ASCII characters, which are delineated by white space (space, tab, or carriage return). The argument count, and argument vector, are then passed to each command upon invocation. The first argument vector is always the name of the command. The command then can parse the input arguments as appropriate. The command interpreter allows commands to be abbreviated, although enough characters must be typed to specify the command uniquely. The command interpreter will respond with "ambiguous command" if this is not the case. If the command is not in the command table, the command interpreter will respond with "unknown command".

CHAPTER 4

SYSTEM MONITOR DEVELOPMENT

The RMX 86 microcomputer development environment consists of a group of tools for generating object code for the 80286 target microprocessor. This chapter describes the RMX microprocessor development environment, including the C compiler implementation and procedures for generation ROM based code. The details used to develop the M286 monitor, and the procedure for developing application code, are presented.

4.1 RMX Development Environment

Software is developed using the RMX microprocessor development environment. This consists of the RMX 86 real-time operating system, which includes an assembler, several compilers, and utilities for generating 8086/8087/80186 family object code. The following is a list of these utilities and a brief description:

IC86 - Intel C language compiler, generates 8086 object modules. Supports a separate large and small model of segmentation.

ASM86 - The 8086/8087/80186 family assembler. Generates relocatable object modules.

AEDIT - Full screen text editor. Supports many terminals and provides many features for efficient text file editing.

LINK86 - Combines a list of 80286 object modules into a single object module. The output object module can be relocatable, or can be specified to be load-time locatable, for loading and executing by the RMX operating system.

LOC86 - Converts relocatable object modules to absolute object modules. This allows the absolute segment address and module order to be specified.

LIB86 - Utility to create and maintain object module libraries. Library files can be used for input to link86.

OH86 - Converts a located object module and converts it to an ASCII format, called Intel Hex Format. This format is usually used for downloading to a target processor or an EPROM burner.

4.2 IC86 C Compiler Implementation

The monitor and application code are developed in the C programming language, using the Intel IC86 compiler, under the LARGE programming model. During the monitor development, some special considerations are needed to configure the code for ROM based execution. This includes memory access, handling ROM initializers and constants,

and the C calling procedure to allow interfacing assembly routines to C code.

4.2.1 C Object Types

The C object sizes may be hardware dependent. The object types and sizes for the IC86 compiler are shown in Table 4.1. In addition, all object types listed may be qualified with the keyword "unsigned", to designate the data will operate under unsigned arithmetic.

4.2.2 IC86 Pointer Usage

Memory addressing on the 80286 processor uses a segmented addressing scheme. A memory address is specified with two parts, consisting of a 16-bit segment and a 16-bit offset. The processor forms the absolute address by shifting the segment left by four bits and adding to it the offset. To accommodate memory access with the IC86 compiler, a 32-bit pointer is formed by concatenating the segment as the upper 16 bits, with the offset as the lower 16 bits. This scheme allows pointers can access the entire 80286 address space. This is true for function pointers, also. The programmer must take caution when incrementing pointers beyond the 64-k byte offset range, since this will cause the segment to be incremented, resulting in a non-linear address span. Also, the programmer should not perform pointer arithmetic.

4.2.3 C Large Model Function Call Conventions

Under the LARGE program model, all pointers must be described using a segment and offset register pair. The IC86 register usage is

Table 4.1 IC86 Object Sizes

TYPE	BIT SIZE
char	8
int	16
short	16
long	32
float	32
double	64

Table 4.2 IC86 Register Usage (LARGE Model)

REGISTER OR REGISTER PAIR	FUNCTION
CS:IP	Program instruction pointer.
SS:SP	Stack pointer.
DS:DI	Data segment pointer.
SI, DI	"register" variables.
BP	Frame pointer (uses SS segment).
ES, CX, BX	Compiler usage.
DX:AX	Function return quantities.

given in Table 4.2. A function call in C is made by building a stack frame, as shown in Figure 4.1. For this discussion, assume a function call is made to function f1. The following steps describe a C function call:

(1) The caller pushes the function arguments on the stack, from right to left. This convention is backwards from other Intel compilers, but it allows C to support a variable number of arguments. This is because the leftmost arguments will always appear at the bottom of the stack, regardless of any optional arguments. In C, it becomes the responsibility of the caller to restore the stack pointer (SP) to the position before the function call was made. When pushing arguments, characters are widened to 16-bit words, to maintain even stack alignment.

(2) After pushing arguments, a call is made to function f1. The far CALL instruction first pushes the return address code segment (CS) and instruction pointer (IP) on the stack. The program address pointer CS:IP is then loaded with the address specified by the CALL opcode.

(3) The IC86 compiler incorporates usage of two 16-bit registers for variables declared as "register". The compiler uses the SI and DI registers for this. The current contents of SI an DI are first saved by pushing them on the stack.

(4) The base pointer (BP) register is used to keep track of the

stack frame. The next step of the function f1 is to push the previous frame pointer, which will be pointing to the previous stack frame, for use when returning from f1. The BP is then set to the stack pointer. BP provides access the stack parameters, through relative addressing. The 80286 supports 8-bit, 16-bit, 32-bit, and 64-bit access.

(5) Next, the stack pointer is decremented to make room for any auto class variables declared by function f1. Auto variables are also accessed by relative addressing from the BP. The function can now execute its code, or call another function. The stack frame allows recursion, so it may make a call to itself.

(6) After the function executes, if it has a return value, it is copied into the return registers. If the function returns a char, it is copied into AL. All 16-bit objects are returned in the AX register, and 32-bit objects are return in the DX:AX pair. Double and float quantities are returned on the 80287 stack.

(7) The function returns by reversing the previous steps. First, the stack pointer is set to the frame pointer. Next, the previous frame pointer is restored, by popping it off the stack, followed by the DI and SI registers. A far RET instruction is executed which loads the return CS:IP address, which was saved on the stack, and increments the stack pointer by 4.

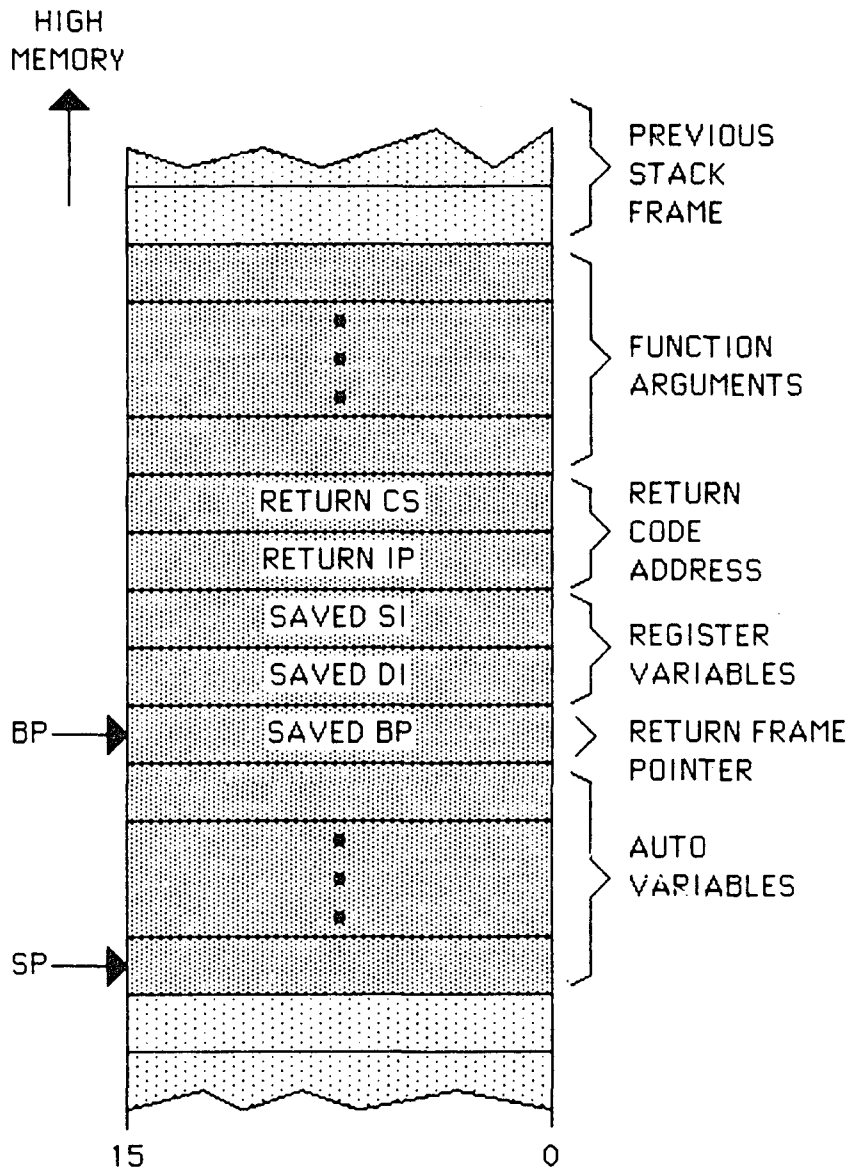


Figure 4.1 C Stack Frame

(8) The caller then adds an offset back to the SP, to remove any arguments which were pushed for the function call. Lastly, the function assignment is made from the return registers.

4.2.4 Initializers and Constants

By default, all C data are put in the class 'data' area. This is acceptable for RAM based applications, because the loader takes care of initializing the data area before running the program, but for a ROM resident code, this is not possible. C code data may follow under one of 3 categories: string constants, static pure data, or static impure data. Considerations must be made for ordinary static read-write data, to ensure it is initialized properly.

4.2.4.1 String Constants

String constants include the function argument message strings used by library routines such as printf or fputs. These are always only read, but by default, located in the data area, and initialized by the loader. The Intel C compiler addresses this problem, however, with the compiler ROM option. This control causes these strings to be moved from the class 'data' area to the class 'code' area of the module where the string is defined.

4.2.4.2 Static Pure Data

Static pure data are variables that are initialized, but read only. These are usually placed in the data area by the C compiler, but again, this is unacceptable for ROM based code. This can be overcome

with another feature of the Intel C compiler, which is the non-standard keyword, "readonly". Prepending any initialized variable with this keyword will tell the compiler to locate the variable in the code segment. For example, the string

```
char *str = "This string is read only\n";
```

would, by default, be located in the data area, since the compiler would assume that the character buffer is read and writable. Changing this to

```
readonly char *str = "This string is read only\n";
```

would move it to the code area. This is true for other object types, such as integers, structures, etc.

4.2.4.3 Static Impure Data

All data that are read and written to falls into the static impure data category. When generating ROM based code, the only additional consideration is whether any of these variables are initialized. If this is the case, the only option is to initialize them explicitly, before these are needed. For the case of M286 monitor modules, any data which falls into this class are handled with a procedure inside the module, and the initialization procedure is made global to a special initialization command, which calls these initialization routines on start up of the monitor. Using this approach saves from having to make the initialized variables global to the initialization command.

4.3 Memory Organization

The Intel IC86 C compiler, under the LARGE model of segmentation, generates object modules containing two class names, which are the 'code' class and the 'data' class. The 'code' class is used to contain the machine code, and the 'data' class is used to contain the static data used by the code segment. Since the LARGE memory model of the C compiler is used, the code and data areas may exceed the 64K-bytes per segment boundary.

The kernel sets up a C runtime environment, by initializing the stack, data, code, and heap memory areas. The unused RAM between the data area and the code area is used as the memory heap, for dynamic memory allocation. The stack area is set at the top of the RAM area, and grows down, as stack space is needed. A memory map of the monitor is shown in Figure 4.2.

The stack, data, and code areas cannot be changed dynamically, but are determined at link time of the monitor. This reduces flexibility of the RAM usage, but simplifies system initialization. Absolute addresses are assigned using the LOC86 utility.

4.4 M286 Development Procedure

The M286 monitor was developed in three phases. Initially, higher-level monitor functions, such as the command line interpreter, and most of the monitor commands themselves, were written in C and executed directly under RMX, using all RMX I/O libraries. This provided

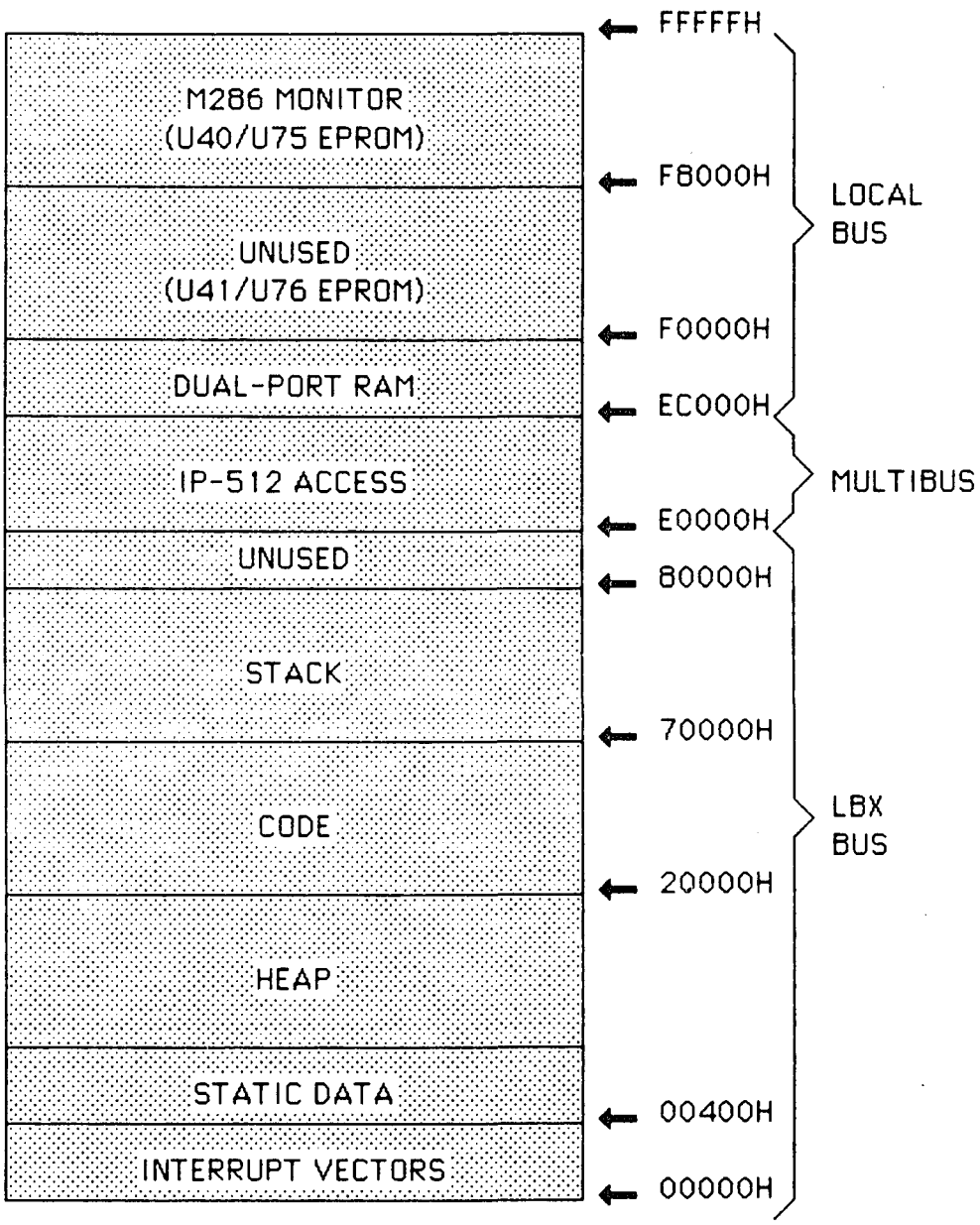


Figure 4.2 Memory Map of the M286 Monitor

an efficient method to test C code. Next, the M286 standard I/O library routines were written tested under RMX. The next step in monitor development involved writing all 286/10 SBC initialization code, and the low-level hardware drivers. This code was linked in place of the RMX libraries. This code is invoked using the standard RMX loader, which starts execution at the SBC initialization code. This disables all RMX control, and runs exclusively M286 libraries. For this approach, monitor code will execute in RAM, almost identical to the final ROM based implementation. After testing at this phase, the system can be rebooted to RMX to make software updates. The final development step involves burning the monitor into EPROM. Slight configuration changes are made to the RAM implementation, resulting from differences in the memory map from of M286 RAM version. The reconfigured code is then located to an absolute address, and then converted to Intel hex object file format, for downloading to an EPROM burner. An even/odd EPROM pair is burned, and installed on a target SBC, for testing. Conditional compilation and assembly directives were used to differentiate the three monitor software configurations. The three object file versions were stored in separate subdirectories.

4.5 Generation of ROM Code

The M286 monitor, after initial testing and debugging in RAM, was configured for ROM execution. This involved first compiling all M286 C source files, under the "ROM" option. Precautions for initialized variables was taken, as outlined in Section 4.2.1. The 80286 assembly

source files were assembled, making sure all read only constants and tables are put in the 'code' class. The resulting object files are then all linked, using LINK86, producing the relocatable file "M86ROM.LNK". This file is then located, using the LOC86 utility, resulting in the absolute file "M86ROM". The 'data' class is located at 0x400, just above the interrupt descriptor table, and the 'code' class is put at 0xF8000, the start of the EPROM area. The reset boot segment, named "BOOTSTRAP", is located at 0xFFFF0, which performs a long jump to the monitor initialization code. The next step for ROM generation is to convert the located file to Intel hex object format, using OH86. The resulting file, "M86ROM.HEX", is then converted to two hex files, containing the even and odd hex data records, named "EVN.HEX" and "ODD.HEX", respectively. This is accomplished using a user-supplied utility, named HEX80, which produces Intel 8080 hex file records, only. The resulting hex files are then downloaded to the Intel UP-20 programmer, and an even/odd EPROM set are burned.

To provide an independent interface for system calls between the application code and the monitor, a jump table was created and located at low ROM area. Calls to the M286 monitor are made to the jump table, which then jumps to the actual CS:IP address of the M286 routine. The advantage to this method is whenever monitor code is recompiled, routine addresses can change, which would cause the application code to be relinked. The jump table prevents this problem. One special consideration is the CTYPE character classification routine. This function is implemented as a look up table, and uses macros (defined in

m86ctype.h) to mask appropriate classification bits. Since the CTYPE table is accessed directly, this is located at the start of the jump table, to keep its address at a constant location. An assembly language interface was written for linking with the application code, named M86CODE.A86, for linking application code with the monitor. The rom code generation procedure is outlined in Figure 4.3.

4.6 Generation of Application Code

It is the job of the programmer to identify and extract the parallelism of the algorithm. Parallelism may be achieved by two approaches. The first is to divide the algorithm into equal parts, which may be executed independently. An example of this method is an image processing algorithm, where an image may be grouped into blocks of pixels, which may then be processed in parallel. This approach is most efficient when a sharing of data between tasks is kept to a minimum. A second method to implement parallelism is through pipelining. This involves dividing the algorithm up into sequential tasks, where the results of one task are sent to another task for further processing. This first task can then start processing the next group of data, thus achieving parallelism. Pipelining produces an output delay to fill the pipeline, but the throughput is increased to that of the slowest task. This suggests that tasks be partitioned to execute at approximately equal times. Other considerations when implementing tasks is to keep bus access and synchronization steps to a minimum. One good method is to use buffers to store intertask data, and transfer the buffer in one step.

This reduces bus accesses.

Tasks are configured to run under the monitor through the following steps. First, the tasks are compiled under the IC86 C compiler. The resulting object files are then linked with the location table, "M86CODE.OBJ", described in Section 4.5 and with a library, "M86APP.LIB", which contains floating point routines, which were not included in the monitor because of the size of this code. This produces a relocatable link file, which located, and then converted to Intel hex format, for downloading to M286. Hex code may be downloaded from the host RMX system through the serial port or through the Multibus. The application code must contain the "main" routine, where execution begins. The application code generation procedure is shown in Figure 4.4.

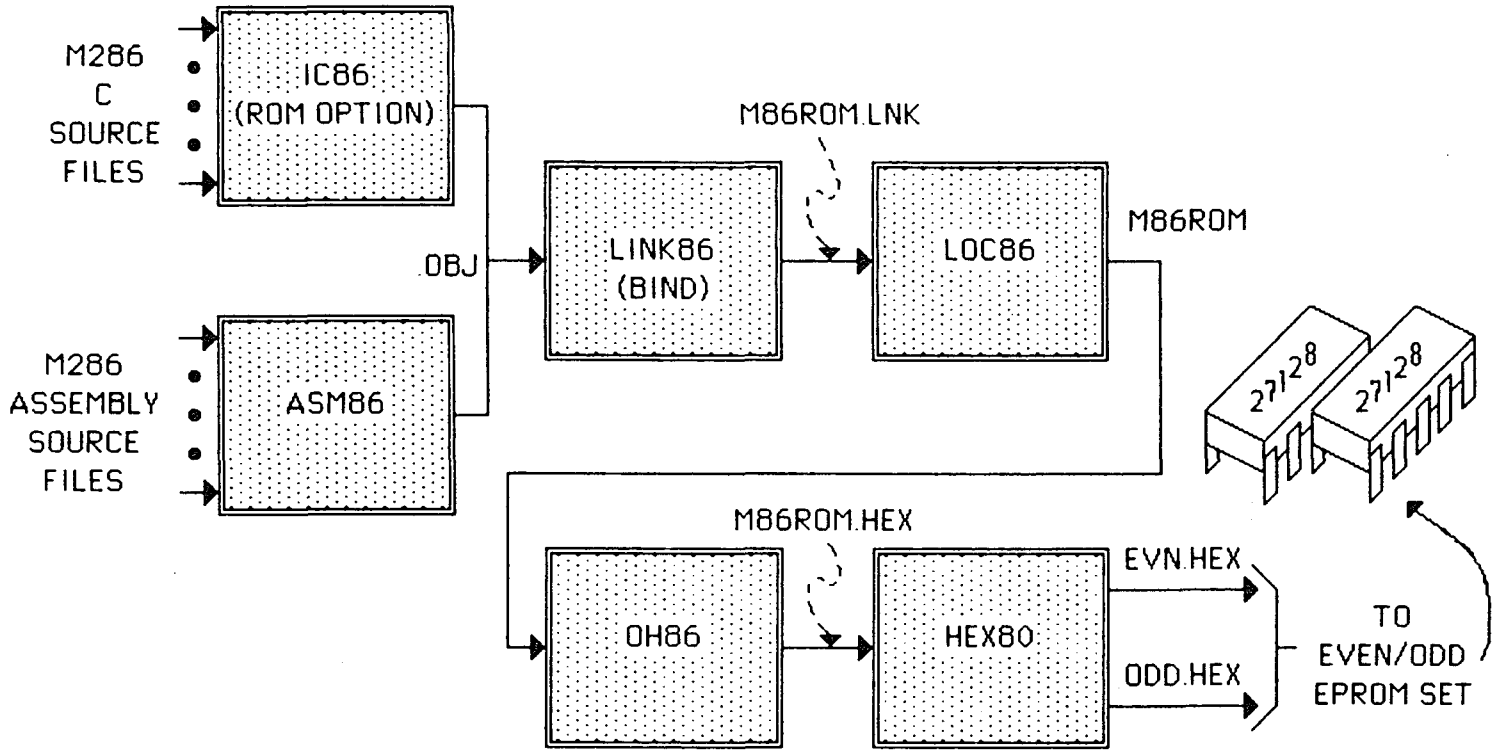


Figure 4.3 M286 Monitor Code Generation

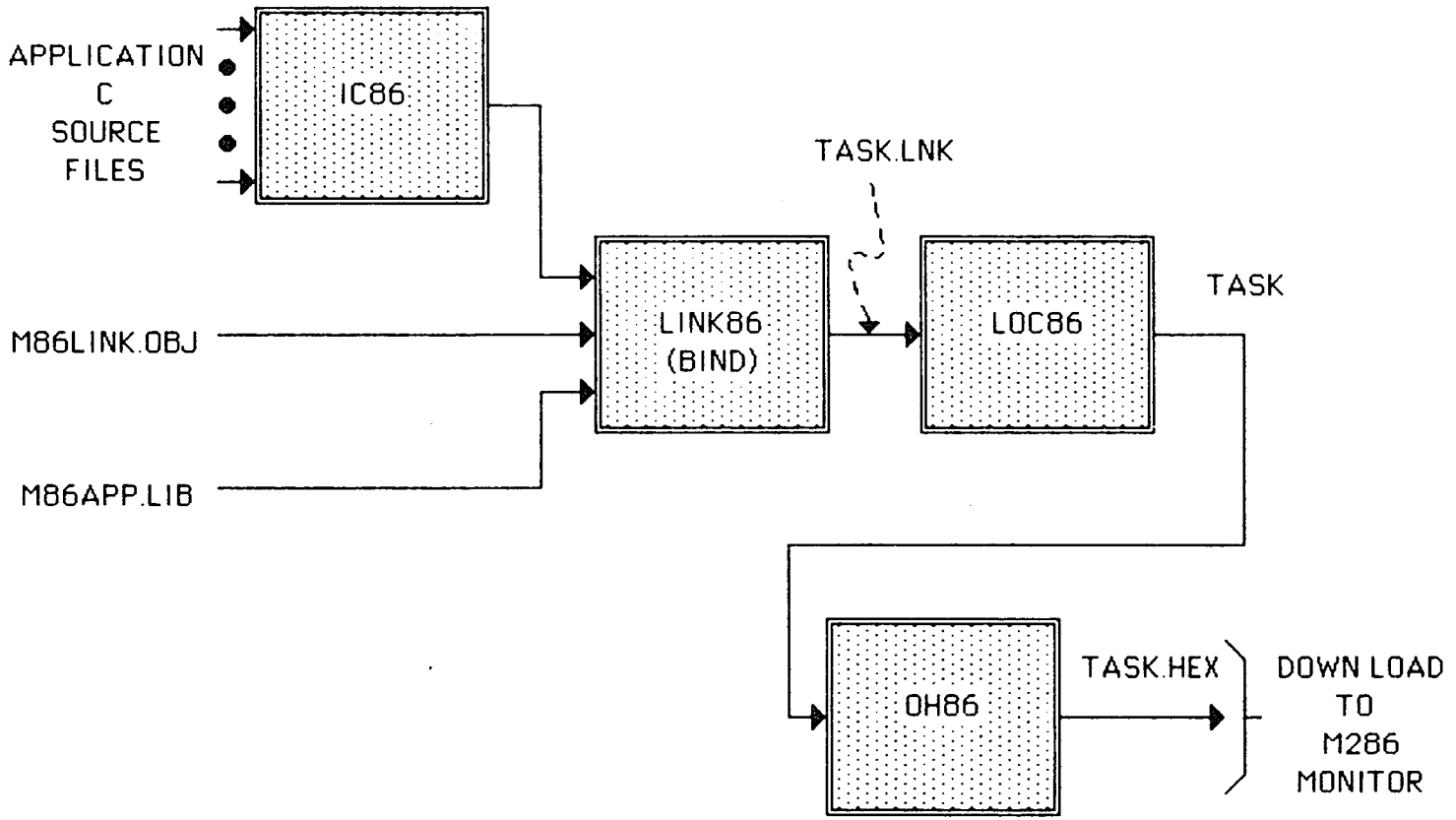


Figure 4.4 Application Code Generation

CHAPTER 5

EXPERIMENTS AND DEMONSTRATIONS

A multiprocessor test bed, consisting of two 286/10 SBCs, was set up to demonstrate the use of the M286 monitor and applications to image processing. The hardware test bed and monitor execution procedure is described, along with the demonstration of a statistical edge operator.

5.1 Multiprocessor Test Bed for Kernel Demonstration

A multiprocessor test bed was configured by adding a second 286/10 single board computer to the to the 380 system. This allows application code to easily be developed and tested under the host RMX system, then downloaded to the target processor for multiprocessor execution. The hardware and tests carried out are described in the following sections.

5.1.1 Hardware Configuration

The test bed was configured for two SBCs. Local memory bus (LBX) P2 connector was added on the System 380 Multibus backplane to provide each SBC with dedicated program memory. A 12CX LBX memory board was added to the second SBC, providing 512 k-bytes of RAM. The Multibus memory access was deselected, providing access only from the LBX bus.

The 286/380 system supports parallel priority arbitration scheme, with up to 14 bus masters. The slot priority is set up using

jumpers on the Multibus backplane. As configured, the disk controller board (slot 1) is given priority 1 and the main 286/10 processor (slot 2) is assigned priority 2. The second processor (slot 7) is given priority 8. The 286/10 SBC supports common bus request signal (CBRQ/), and is enabled on both processor boards.

Dual-port RAM was added to the second processor board. This was configured with two 6116 static RAMs, for a total of 4k-bytes of memory. Local memory jumpers were set for a starting address at 0xEC000, with an on-board block size of 4k-bytes. The Multibus access address was set to 0xE0000, for use by the other processor boards as semaphores. The memory was installed in socket pair U54/U87.

Local memory was configured for four 8k x 8 EPROMs, with U40/U75 socket pair set at base address 0xF8000, and U40/U75 socket pair at 0xF0000. The M286 monitor code was burned into two 27128 EPROM devices, and installed in local memory sockets U40/U75. Local memory U41/U76 sockets are left empty, and may be used for future expansion.

5.2 Multiprocessor T-Edge

In order to test the M286 monitor and investigate parallel processing, an image processing algorithm was selected and executed on the test bed. The algorithm used is the t-edge, which is a statistical edge operator. This algorithm was selected because it requires considerable execution time, and it allows partitioning among multiple processors. Two different partitions of the algorithm were selected, to test different features of the kernel.

5.2.1 Description of the T-Edge Algorithm

The t-edge is a two-dimensional operator designed to enhance edges in images with noisy backgrounds, such as noise present in forward-looking infrared (FLIR) imagery [18]. The operator is based on the student-t test, which quantifies the difference between the means of two local populations (pixel samples). The t-edge equation is:

$$t^2 = \frac{(m_1 - m_2)^2}{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \tag{5.1}$$

where n_1 and n_2 are the number of pixels in the two windows. These values are usually equal in practice. The mean of the windows, m_1 and m_2, are given by

$$m = \frac{1}{n} \sum_{i=1}^{n} P_i \tag{5.2}$$

where P_i is the value of the i-th pixel in the window, and the variances, s_1 and s_2, are defined by

$$s^2 = \frac{1}{(n-1)} \left(\sum_{i=1}^{n} P_i^2 - m \right). \tag{5.3}$$

The t-edge algorithm is edge directional, dependent upon the tessellation of the two statistics windows. For example, windows oriented such that the first is above the second will detect horizontal

edges. Windows oriented side by side will detect vertical edges.

5.2.2 Horizontal T-Edge

The first experiment investigated the execution of the t-edge by partitioning the image into two separate windows, with one processor computing the image top half and the second processor computing the lower half. One semaphore was used to allocate access to the Itex boards, including the input frame buffer, and the output frame buffer. Processors gain access to the resources through the "mp_wait" system call and release the resource through the "mp_signal" call, as described in section 3.1.1.1. A 256x256 pixel t-edge was carried out, using 2x5 pixel windows.

The t-edge routines were coded to minimize the number of access needed to the Itex boards. The algorithm proceeded in the following steps. First the input frame buffer was allocated, using a semaphore, and the two adjacent 2x5 pixel windows were read into buffers. The frame buffer was then released, to let other processor use the frame buffer. The statistics of the two windows was computed, followed by the actual t^2 value. Floating point math was used. This value was then scaled, converted to a byte, and stored in a 256 byte row buffer. Every pixel in the row was computed in this way. At the end of the row, the output frame buffer was allocated, and the entire row was written to it.

5.2.2.1 Horizontal T-Edge Results

The horizontal t-edge routine was benchmarked. First, a single processor t-edge was implemented, using no system arbitration calls, to

Table 5.1 Execution Times for 256x256 Horizontal T-Edge

CASE	EXECUTION TIME (minutes:seconds)	% OVERHEAD
(1) 1 processor, reference	8:30.3	-
(2) 1 processor, with kernel overhead	8:32.5	0.43
(3) 2 processors	4:18.5	1.31

Table 5.2 Execution Times for 256x256 Combined T-Edge

CASE	EXECUTION TIME (minutes:seconds)	% OVERHEAD
(1) 1 processor, reference	16:33.1	-
(2) 2 processors	8:30.5	2.81

determine the minimum processing time, which is used as a reference. Second, a single processor t-edge was timed, which incorporated the allocation calls, to determine the overhead of these routines. Finally, two processors were used, partitioning the upper half of the image to one processor, and the lower half to the other. The results are shown in Table 5.1. These results show a low overhead of 1.31% when compared to the single processor t-edge. This high efficiency can be attributed to the cycling which will be established after the two processors start up, resulting in few bus collisions.

5.2.3 Horizontal and Vertical T-Edge

A second experiment was performed, to investigate both resource sharing and data sharing, through a mailbox. A separate horizontal and a vertical t-edge computation was made for each point, and the two results were averaged, to form a combined edge map. The multiprocessor algorithm was carried out as follows. Processor 1 computes the horizontal t-edge for one pixel, while processor 2 computes the corresponding vertical t-edge for the same pixel. The processors compute the t-edge values for one entire row, and save the result in a 256 byte buffer. At the end of the row, the second processor then sends the vertical result which is saved in the buffer back to the first, through a 256 byte mailbox. The first processor then averages the horizontal and vertical values, and writes the result to the output frame buffer. One semaphore allocates bus resources, while two others are used to transfer data through the mailbox.

5.2.3.1 Horizontal and Vertical T-Edge Results

The combined horizontal and vertical t-edge was benchmarked, similar to the previous experiment. A 256x256 t-edge was carried out, using 2x5 pixel windows. The one and two processor execution times are shown in Table 5.2, resulting in an overhead of 2.81% for the two processor implementation. The additional overhead, as compared to the first experiment, can be attributed to the mailbox transfer at the end of each row calculation. For increased processing speed, three processors could be used, allocating the third processor to read the results from the horizontal and vertical processors, and combining the results, in pipeline fashion.

CHAPTER 6

SUMMARY AND CONCLUSIONS

Integration of off-the-shelf processor boards and image processing hardware into a multiprocessor environment through the M286 system monitor have been accomplished. The system described allows investigation of executing image processing algorithms in parallel. The results of this project fall into two categories, including design and implementation of the M286 monitor, and the investigation of multiprocessing and applications to image processing.

6.1 Conclusion

The design and implementation of a multiprocessor kernel and monitor have been described. The design of the M286 monitor into functional layers has proven to be a reliable method to carry out a large software project. The five functional layers allow the monitor to be understandable and developed efficiently. The C programming language has proven to be an outstanding development language, since it allows low level access to hardware, and compiles into efficient object code, resulting in 17K bytes of EPROM usage. The test-and-set mechanism has proven to be a simple and reliable method to achieve mutual exclusion. The kernel allows 286/10 single board computers to synchronize and share data and bus resources through semaphores.

The M286 monitor and RMX operating system create an efficient

environment to develop multiprocessor algorithms. The monitor has been applied to image processing algorithms, which allow partitioning algorithms into tasks, and executed on 286/10 SBCs. The monitor was demonstrated, using a statistical edge operator, with two processor boards. The M286 kernel was shown to require little overhead, and processing throughput was improved.

6.2 Further Research Work

This section outlines improvements and extensions to the M286 monitor and further multiple processor research.

6.2.1 Multiprocessor Profiling

In order to further investigate parallel processing, a method of profiling a multiple processor system is needed. This would give the algorithm developer a method to study the effects of different task partitioning, and help uncover processing bottlenecks. A multiprocessor profiler could be implemented by periodically interrogating all processors using an interrupt line and then reading the execution status through shared memory mailboxes. The data could then be collected and time/activity plots could be displayed.

6.2.2 Kernel Extensions

The test-and-set mechanism used to achieve mutual exclusion is a simple and trustworthy method to implement a multiprocessor system on a shared bus. However, the test-and-set protocol can become a bottleneck for a large multiprocessor system. If many processors try to use the

test-and-set operation, which asserts the Multibus LOCK, they will prevent other processors from accessing the Multibus, thus slowing system performance.

To carry out large multiprocessor systems efficiently, other solutions would be needed. One suggestion is to use a semaphore with interrupts to allocate resources. This scheme would put one processor in charge of allocation, which would be configured with dual-port RAM to implement a semaphore. Each processor would be given a dedicated Multibus interrupt line. The protocol would begin with a processor accessing the semaphore, through test-and-set operation. When the requesting processor gets the semaphore, it would then write an operation code to a mailbox location in the dual-port RAM, and then assert an interrupt line to signal to the allocating processor that a request is made. The requester would then go to a wait state, until a dedicated interrupt line responds back. The allocating processor would examine the request, and determine if the resource is busy. If it is not busy, the allocator would respond back with an interrupt signal. If it is busy, the allocator would put the requester at the end of a queue. For a processor to release a resource, it would access to semaphore, then write a code signifying the resource if finished to the requester's mailbox, and interrupt the allocator, to let it know a request is made. The allocator would then check the request queues and grant any waiting processors through the interrupt lines. The advantage to this method is that the processors that would be forced to wait for a resource would not have to perform a continuous test-and-set operation; instead waiting

is done off the bus, through interrupt lines. However, a test-and-set operation is needed to access the allocator, and speed is important for the allocator to process the request. Also, this scheme requires each processor to have a dedicated interrupt line, which would limit the system to eight processors on the Multibus.

APPENDIX A: M286 System Calls

Routine Summary

ROUTINE	FUNCTION PERFORMED
input	Input data from I/O port.
mkptr	Make a LARGE model C pointer.
mp_signal	Multiprocessor semaphore signal.
mp_wait	Multiprocessor semaphore wait.
mtest64k	Test a 64k-byte segment of memory.
output	Output data to I/O port.
read	Read characters from channel to buffer.
reg286	Read 80286 registers.
tas	Perform test and set operation.
tmode	Set the mode of a serial port.
tread	Performs low-level input from a serial port.
twrite	Performs low-level output to a serial port.
write	Write characters from channel to buffer.
_editline	Gets a line from standard input and supports editing.

NAME

input - Input data from I/O port.

SYNOPSIS

```
char input(port)
  int port;                    - Address of port.
```

DESCRIPTION

This routine inputs a data byte from the 80286 processor I/O bus.

SEE ALSO

output

NAME

mkptr - Make a LARGE model C pointer.

SYNOPSIS

```
unsigned long mkptr( seg, off )
  unsigned int seg;            - Address segment.
  unsigned int off;            - Address offset.
```

DESCRIPTION

This routine forms a segmented LARGE model C pointer, by concatenating the segment as the upper 16 bits, and the offset as the lower 16 bits. The absolute address is computed by shifting the segment left by four bits and adding this to the offset.

This operation may be done in C, but is optimized here in assembly. The equivalent code fragment in C would be:

```
ptr = ((unsigned long)seg) << 16 | off;
```

RETURNS

Segmented pointer format segment:offset (32 bits).

NAME

mp_signal - Multiprocessor semaphore signal.

SYNOPSIS

```
mp_signal(semaphore)
  char *semaphore;          - Pointer to shared memory semaphore.
```

DESCRIPTION

Causes the shared memory semaphore to be cleared, thus signalling to a waiting processor that the shared resource is ready. This is used to achieve synchronization between processors.

SEE ALSO

mp_wait, tas.

NAME

mp_wait - Multiprocessor semaphore wait.

SYNOPSIS

```
mp_wait(semaphore)
  char *semaphore;          - Pointer to shared memory semaphore
```

DESCRIPTION

Causes the calling routine to poll the semaphore until it becomes available. The semaphore is made available through the mp_signal system call. This is used as a synchronizing mechanism for multiple processors.

SEE ALSO

mp_signal, tas.

NAME

mtest64k - Test a 64k byte segment of memory.

SYNOPSIS

```
unsigned int mtest64k(segment, mloc)
  unsigned int segment;     - Segment to test.
  unsigned int *mloc;       - Offset of failure.
```

DESCRIPTION

This routine tests the specified 64k segment. The routine is implemented using registers only. Three separate, independent tests are performed:

1) Stuck at zero faults. This is done by writing a 0x0000 to each address, reading back the result, then writing/reading to the same location, with 0xffff.

2) Stuck or shorted address lines are tested by reading/writing the address offset to the location.

3) A pattern consisting of ones and zeros is then written and read back and compared.

RETURNS

If the segment is OK, 0 is returned. If the segment contains an error, 1 is returned. Also, the offset where the test failed will be returned in "mloc".

NAME

output - Output data to an I/O port.

SYNOPSIS

```
output(port, data)
  char *port;          - Address of port.
  char data;           - Byte to send to port.
```

DESCRIPTION

This routine outputs data to the 80286 processor I/O bus.

SEE ALSO

input

NAME

read - Read characters from serial port to buffer.

SYNOPSIS

```
read(fp, buffer, n)
  FILE *fp;                 - I/O stream pointer.
  char *buffer;             - Buffer to store data.
  int n;                    - Number of characters to read.
```

DESCRIPTION

Reads "n" characters from serial channel to buffer. If the file stream is opened in ASCII mode, characters are echoed back, and newlines are expanded to CR, LF pairs.

NAME

reg286 - Read 80286 registers.

SYNOPSIS

```
#include ":m86:include/m86kernel.h"
reg286( reg )
  REG286 *reg;
```

DESCRIPTION

Returns current registers in REG286 structure. The REG286 structure is defined (in m86kernel.h) as:

```
typedef struct
{
   unsigned int AX, BX, CX, DX, CS, SS, DS,
                ES, IP, SP, SI, DI, BP, FL;
} REG286;
```

NAME

tas - Performs test and set semaphore operation.

SYNOPSIS

```
int tas(sem);
  int *sem;                    - Pointer to semaphore.
```

DESCRIPTION

Does an indivisible test and set for multiprocessor synchronization. A call to "tas" will return a 1 if it is busy; otherwise, will return a 0 and will lock the semaphore from other tasks attempting to access it.

This is implemented in assembly using indivisible XCHG instruction, while asserting the Multibus bus lock.

SEE ALSO

mp_signal, mp_wait.

NAME

tmode - Set the mode of a serial port.

SYNOPSIS

```
int tmode(fp, mode)
  FILE *fp;                    - The file pointer.
  char *mode;                  - The file mode.
```

DESCRIPTION

Legal modes are:

"b" - Set to binary mode. Data is unbuffered. No processing of input data. Data is returned as read. Data is not echoed.

"a" - Set to full-duplex, ASCII mode. Special processing of input data. Data is only read upon a carriage return. The delete character removes character from the input buffer and send a backspace to the terminal. Data are echoed to back to the terminal.

RETURNS

0 upon error, 1 otherwise.

NAME

tread - Performs low-level input from a serial port.

SYNOPSIS

```
tread(fd, buf, n);
  int fd;          - Port id, 0 = aux port, 1 = console port.
  char *buf;       - Pointer to I/O buffer.
  int n;           - Number of chars to write.
```

DESCRIPTION

This is a driver for reading the 8274 MPSC. Reads "n" characters from a serial port into "buf". Implemented using polling.

SEE ALSO

read.

NAME

twrite - Performs low-level output to a serial port.

SYNOPSIS

```
twrite(fd, buf, n);
  int fd;          - Port id, 0 = aux port, 1 = console port.
  char *buf;       - Pointer to I/O buffer.
  int n;           - Number of chars to write.
```

DESCRIPTION

This is a driver to write to the 8274 MPSC. Writes a character buffer of length "n" to the serial ports, by polling.

SEE ALSO

write.

NAME

write - Write characters from buffer to channel.

SYNOPSIS

```
write(fp, buffer, n)
  FILE *fp;                    - I/O stream pointer.
  char *buffer;                - Buffer to store data.
  int n;                       - Number of characters to read.
```

DESCRIPTION

Writes "n" characters from channel to buffer. If the file stream is opened in ASCII mode, newlines are expanded to CR, LF pairs.

NAME

_editline - Gets a line from standard input and supports editing.

SYNOPSIS

```
char *_editline(buf, maxbuf)
  char *buf;            - Buffer to store the line
  int  maxbuf;          - Maximum number of characters to store
```

DESCRIPTION

This routine inputs a line from standard input and supports the delete key to remove the last character typed from the buffer. The bell will sound if the buffer is deleted past the beginning of the buffer or if more than "maxbuf" characters are typed into the buffer.

Reading is terminated upon a newline.

RETURNS

A pointer to the start if "buf". If end of file is read, NULL is returned.

APPENDIX B: M286 Monitor Commands

Command Summary

COMMAND	FUNCTION PERFORMED
boot	Execute 286/10 board monitor boot code.
display	Display an area of memory.
fill	Fill a block of memory.
help	Give command summary.
input	Input data from an I/O port.
load	Load Intel hex format object file.
memtest	Memory test.
mode	Set serial port parameters.
output	Output data to an I/O port.
register	List 80286 registers.
run	Execute loaded program.
set	Set memory bytes.

NAME

boot - Execute 286/10 board ROM boot code.

SYNOPSIS

boot

DESCRIPTION

Forces a call to the ROM boot address located at 0xFFFF0. This runs the monitor boot code.

NAME

display - display an area of memory.

SYNOPSIS

display [OPTIONS] [Segment:Offset]

DESCRIPTION

The display command allows the user to view an area of memory. The memory is displayed in two separate fields. The first field is the absolute hex bytes, with up to 16 bytes per line, followed by an ASCII display field. The previous address is saved for displaying the next group of memory, if no address is specified. The address is incremented linearly; offsets will not wrap around. The segment will increment by 0x1000 to the next contiguous memory location, when the offset wraps to 0.

OPTIONS

-n COUNT

Display COUNT bytes. This value is saved for successive displays. The value is read in hex. The default count is 100 hex bytes (16 bytes by 16 rows).

[Segment:Offset]

This specifies the memory address to display the data. The address is given as the segment, followed the offset, which are delimited by a colon, ':'. Both segment and offset are read as hex.

NAME

fill - fill a block of memory

SYNOPSIS

fill segment:offset number_of_bytes value

DESCRIPTION

The fill command is used to set a contiguous block of memory with a value. Data is filled by bytes. The block to be filled consists of the starting address, followed by the length of the block (in hex), and the datum value (in hex) to be filled.

SEE ALSO

set

NAME

help - Give command summary

SYNOPSIS

help

DESCRIPTION

Gives a quick listing of all M286 monitor commands and their usage.

NAME

input - Input data from an I/O port.

SYNOPSIS

input [OPTIONS] PORT_ADDRESS

DESCRIPTION

The input command inputs and displays a byte from the specified I/O port. The port address is read as hex, in the range 0000 - FFFF. The port datum is displayed in hex.

OPTIONS

-n COUNT

Output COUNT bytes. If no count is specified, one byte is the default.

SEE ALSO

output

NAME

load - Load Intel hex format object file.

SYNOPSIS

load [options]

DESCRIPTION

Reads Intel hexadecimal object file format data records through the serial data and loads them into memory.

OPTIONS

-p {a|b}

Specify serial port to load object file. Legal port channels are "a" (aux port) or "b" (console port). The console port is the default.

-m

Upload hex object file through Multibus channel mbox0.

-r

Run the hex file at the code segment and instruction pointer specified by the start address record. The code will execute only after receipt of end of file record. The default case returns to the monitor.

-v

Verbose. Echo the file, in ASCII format, as it is being downloaded.

APPENDIX

Intel 8086 hex object format is made up of four record types. These are the data record, end of file record, extended address record (used to specify the base address), and the execution start address record (specifies where program execution begins). An absolute data byte is represented in hex format, as two (7 bit) ASCII characters, '0' through '9', or 'A' through 'F'.

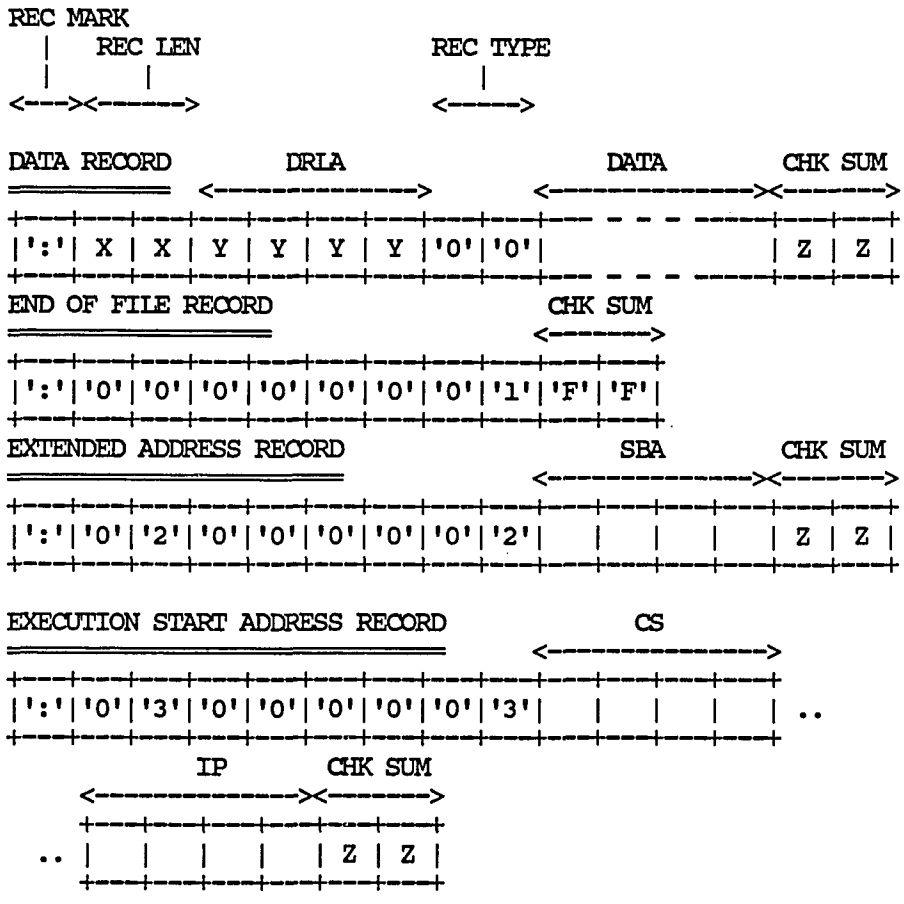
The check sum field is the two's compliment of all the sum of all previous data bytes (formed by combining pairs of ASCII data, and converting to integer form). This does not include the REC MARK.

The record fields and sizes are:

REC MARK - Beginning of record mark, ":" (byte).
REC LEN - Record length, in bytes (byte).
REC TYPE - Record identifier, 0-3 (byte).
DRLA - Data record load address offset (word).
CHK SUM - Check sum (byte).

SBA - Segment base address (word).
CS - Code segment (word).
IP - Instruction pointer (word).

The Intel 8086 hex record types, showing data fields are:



SEE ALSO

loc86 [RMX 86], oh86 [RMX 86], run

NAME

memtest - Memory test.

SYNOPSIS

memtest [OPTIONS] Segment:Offset

DESCRIPTION

This routine is used to test system RAM. RAM is tested until a read/write failure occurs, or RAM upper limits are reached. Blocks of 64k bytes are tested.

OPTIONS

-n COUNT

Number of 64k blocks of RAM to test. The default number is 1 block. This number is read in hex format.

Segment:Offset

This specifies the memory address to display the data. The address is given as the segment, followed the offset, which are delimited by a colon, ':'. Both segment and offset are read as hex.

NAME

mode - Sets serial port parameters.

SYNOPSIS

mode [OPTIONS]

DESCRIPTION

This command is used the change the parameters on the console and auxiliary serial ports. This command with no options will display the current port parameters.

OPTIONS

-p {a|b}

Specify the serial port to modify. Legal port channels are "a" (aux port) or "b" (console port). The console port is the default.

-b BAUD_RATE

Set the baud rate. Legal baud rates are 150, 300, 600, 1200, 1800, 2400, 4800, 9600, and 19200 bits/sec.

-x {e|o|n}

Set the transmit and receive parity sense. Legal values are: e = even parity, o = odd parity, and n = no parity.

-s NO_BITS

Set the number of stop bits. Legal number of bits are 0, 1 and 2.

-l NO_BITS

Set the transmit and receive character bit lengths. Legal lengths are 5, 6, 7 and 8 bits. Note that if 8 bits are selected, parity is not transmitted to the processor.

NAME

output - Output data to an I/O port.

SYNOPSIS

output [OPTIONS] PORT_ADDRESS DATA

DESCRIPTION

The output command outputs and a byte to the specified I/O port. The port address is read as hex, in the range 0000 - FFFF. The port datum is read in hex.

OPTIONS

-n COUNT

Output COUNT bytes. If no count is specified, one byte is the default.

SEE ALSO

input

NAME

register - List 80286 registers

SYNOPSIS

register

DESCRIPTION

Displays the contents of all 80286 registers.

NAME

run - Execute loaded program.

SYNOPSIS

run

DESCRIPTION

Checks if ESA record was given during download (see load). If so, execution is begun at the CS:IP specification.

SEE ALSO

load

NAME

set - set memory bytes

SYNOPSIS

set Segment:Offset

DESCRIPTION

The set command is use to modify one or more bytes of memory. The command is invoked by specifying the starting address. The command then prompts with the address, and its current contents, in hex. A byte new value, in hex, may be entered, which will be stored at the current address. The command will then prompt for the next address. More data can be entered. A data value can be skipped over (left unchanged) by typing a carriage return. Also, addresses may be decremented by entering a '-'. When all memory has been modified, typing a 'q' or 'Q' will return to the monitor.

SEE ALSO

fill

REFERENCES

[1] Bowen, B. A. and Buhr, R. J. A. The Logical Design of Multiple Microprocessor Systems, Prentice-Hall, 1980, pp. 105-107.

[2] iMMX 800 Multibus Message Exchange Reference Manual, Intel Corporation, 1982.

[3] iRMX 86 Programmer's Reference Manual for Release 6.0 Part I, Intel Corporation, 1984.

[4] iRMX 86 Programmer's Reference Manual for Release 6.0 Part II, Intel Corporation, 1984.

[5] iRMX 86 Installation and Configuration Guide, Intel Corporation, 1984, pp. 15-1 - 15-10.

[6] Kernighan, B. W., and Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

[7] Johnson, J. B. and Kassel, S., The Multibus Design Guidebook, McGraw Hill, 1984, pp. 244-245.

[8] Introduction to the System 286/380 Microcomputer, Intel Corporation, 1984.

[9] Guide to Using the iSBC 286/10 Single Board Computer, Intel Corporation, 1983.

[10] Intel Multibus Specification, Intel Corporation, 1982.

[11] iSBC 028CX/056CX/012CX/010CX/020CX (CX-Series) RAM Boards Hardware Reference Manual, Intel Corporation, 1984.

[12] IP-512 Technical Manual Multibus Version, Imaging Technology Incorporated, 1985.

[13] Bowen, B. A. and Buhr, R. J. A. The Logical Design of Multiple Microprocessor Systems, Prentice-Hall, 1980, pp. 103-104.

[14] ASM86 Language Reference Manual, Intel Corporation, 1983.

[15] Savitzky, S., Real-Time Microprocessor Systems, Van Nostrand Reinhold, 1985, pp. 118-121.

[16] iAPX 286 Programmer's Reference Manual including the iAPX 286 Numeric Supplement, Intel Corporation, 1985, Numeric Supplement, pp. 3-3 - 3-7.

[17] iC-86 Compiler User's Guide, Intel Corporation, 1984.

[18] Strickland, R. N., and Gerber, M. R. "Estimation of Ship Profiles from a Time Sequence of Forward-Looking Infrared Images", Optical Engineering, Vol. 25, No. 2, pp. 995-1000, August 1986.