11-7-1995

Flash memory boot block architecture for safe firmware updates

Hsiang Chi
Florida International University

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

Flash Memory Boot Block Architecture for Safe Firmware Updates

A thesis submitted in partial satisfaction of the

requirements for the degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

by

Hsiang Chi

1995

Thesis Committee Approval Sheet

To: Dean Gordon R. Hopkins
College of Engineering and Design

This thesis, written by HSIANG CHI, and entitled, FLASH MEMORY BOOT BLOCK ARCHITECTURE FOR SAFE FIRMWARE UPDATES, having been approved in respect to style and intellectual content, is referred to you for judgement.

We have read this thesis and recommend that it be approved.

Jean Andrian

Subbarao V. Wunnava

Armando Barreto, Major Professor

Date of Defense: November 7, 1995

The thesis of Hsiang Chi is approved.

Dean Gordon R. Hopkins
College of Engineering and Design

Dr. Richard L. Campbell
Dean of Graduate Studies

Florida International University, 1995

I dedicate this thesis to my family. Without their support and most of all love, the completion of this work would not have been possible.

ACKNOWLEDGMENTS

I would like to thank all those who helped me in one way or another to make this work possible.

I would like to acknowledge the members of my committee for their helpful comments and patience. I am thankful to Drs. Jean Andrian, Armando Barreto, and Wunnava Subbarao for their participation in my supervisory committee.

Special thanks must go to my major professor, Dr. Armando Barreto, for his support and encouraging comments, and especially for having the confidence in me to give me the chance to complete this thesis.

I cannot express in words my thanks to Drs. James Story, Jean Andrian, Pierre Schmidt, and Gustavo Roig, for all they have done in support of my graduate studies.

I am infinitely indebted to my parents and family for their incredible support and encouragement throughout my graduate school career.

Last and most importantly, I would like to thank my best friend and wife, Jianping, for her constant love and support.

ABSTRACT OF THE THESIS

Flash Memory Boot Block Architecture
for Safe Firmware Updates

by

Hsiang Chi

The most significant risk of updating embedded system code is the possible loss of system firmware during the update process. If the firmware is lost, the system will cease to operate, which can be very costly to the end user. This thesis is concerned with exploring alternate architectures which exploit the integration of flash memory technology in order to overcome this problem. Three design models and associated software techniques will be presented. These design models are described in detail in terms of the strategies they employ in order to prevent system lockup and the loss of firmware. The most important objective, which is addressed in the third model, is to ensure that the system can continue to process interrupts during the update. In addition, a portion of this research was aimed at providing the capability to perform updates remotely, and at maximizing system code memory space and available system RAM.

TABLE OF CONTENTS

APPENDICES PAGE

TABLE OF FIGURE

FIGURE PAGE

Chapter One
Introduction

1.1 Introduction

Over the last five years, Flash memory products have revolutionized how designers think about storing control code in computers, peripherals, communication devices, and a wealth of other applications. In fact, Flash memory is seen as an enabling technology in many new microprocessor-based designs. The features of Flash memory have made it the fastest growing IC memory type in the 1990s.

1.2 What is Flash Memory

Introduced in 1988, ETOX (EPROM Tunnel Oxide) flash memory is a high-density, nonvolatile, high performance, read-write memory solution. It is characterized by low power consumption, extreme ruggedness, and high reliability. The cost of flash memory components continues to decline sharply due to: (a) manufacturing economies inherent in ETOX, (b) increases in memory density, and (c) rapid growth in production volume.

Flash memory is a new memory technology, that enables high density storage which requires complete erase before re-writings.

Flash offers a design solution with distinct advantages over other solid-state memories. It will be vital to future product differentiation in numerous applications that require firmware updates, or compact mass storage (See Table 1) [1].

Flash memory enables system firmware updates. Thus, it allows manufacturers to enhance product features cost-effectively, fix product bugs, and keep up with changing standards, even after the sale of the product. Boot block flash memory is the architecture of choice for providing safe firmware updates to a wide range of applications that include: cellular phones, modems, medical instruments, printers, PC BIOS, etc.

Memory	Inherently Non-Volatile	High Density	Low Power	One Transistor Cell	In-System Re-Writable	Code and Data Storage	Byte Alterable	Blocking	Hands off Updates
Flash	X	X	X	X	X	X		X	X
SRAM + Battery					X	X	X	X	X
DRAM + Disk		X			X	X	X	X	X
EEPROM	X		X		X	X	X	X	X
OTP/ EPROM	X	X	X	X					
Masked ROM	X	X	X	X					

Table 1. Comparison among firmware implementation alternatives.

Flash memory solutions should be evaluated based upon a checklist of key criteria for updateable firmware. The following paragraphs summarize of the shortcomings of the other storage technologies and how flash addresses them.

- ROM (Read Only Memory) is a mature, high density, nonvolatile, reliable and low cost memory technology, that is widely used in PC and embedded applications. Once it is manufactured, however, the contents in the ROM area can never be altered.

Additionally, initial ROM programming involves a time-consuming mask development process which requires stable code. It is most cost-effective in high volumes.

Easy updatability makes flash memory clearly more flexible than ROM in most applications.

- SRAM (Static Random-Access Memory) is a high-speed, reprogrammable memory technology that is limited by its volatility and relatively low density. As a volatile memory technology, SRAM requires constant power to retain its contents. Built-in battery backup is therefore required when the main power source is turned off. Since battery failure is an inevitable fact of life, SRAM requires four to six transistors to store one bit of information. This becomes a significant limitation when developing higher densities, and thus keeps SRAM costs relatively high [1].

In contrast, flash memory is inherently nonvolatile, and the single transistor cell design of the ETOX (flash memory) manufacturing process is extremely scaleable. This allows for the development of continuously higher densities, and steady cost improvement over SRAM.

- EPROM (Electrically Programmable Read-Only Memory) is a mature, high-density, nonvolatile technology that provides a degree of updatability not found in ROM. An OEM (Original Equipment Manufacturer) can program EPROM as needed to accommodate code changes or varying manufacturing unit quantities. Once

programmed, however, the EPROM can only be erased by removing it from the system, and then exposing the memory component to ultraviolet light. This is an impractical and time-consuming procedure for many OEMs and a virtually impossible task for end-users [1].

Unlike EPROM, flash memory is electrically re-writable within the host system, making it a much more flexible and easier to use alternative. Flash memory offers OEMs not only high density and nonvolatility, but higher functionality and the ability to use the same BIOS system for several product versions.

- EEPROM (Electrically Erasable Programmable Read-Only Memory) is nonvolatile and electrically byte-erasable. Such byte-alterability is needed in certain applications, but involves a more complex cell structure, and significant trade-offs in terms of limited density, lower reliability and higher cost, this makes it clearly unsuitable as a mainstream memory [1].

Unlike EEPROM, flash memory technology utilizes a one-transistor cell, allowing higher densities, scalability, lower cost, and higher reliability, while taking advantage of in-system, electrical erasability.

- DRAM (Dynamic Random Access Memory) is a volatile memory favored for its density and low cost. Because of its volatility, however, it requires not only a constant

power supply to retain data, but also an archival storage technology, such as disk, to back it up [1].

Partnered with hard disk for permanent mass storage, DRAM technology has provided a low-cost, yet space and power-hungry solution for today's PCs.

With ETOX process technology, most flash memory cells are 30% smaller than equivalent DRAM cells. Flash memory's scalability offers a price advantage as well, keeping price parity with DRAM, and also becoming more attractive as a hard disk replacement in portable systems, as densities grow and cost decline.

1.3 Updatable Code Storage

Code and data storage comprise the updatable nonvolatile memory applications that require high performance, high density, and easy update capability. Because these applications are not updated as frequently as solid-state mass storage applications, erase/write cycles are not as critical as integration and performance requirements. This application segment is served effectively with full chip-erase or Boot Block products.

Intel's 28F001BX 1 Mbit Boot Block flash component and AMD's (Advanced Micro Devices) AM29F010 1 Mbit device, featuring a sectored architecture, have been widely accepted in embedded code storage applications, particularly in PC BIOS and cellular communications. By adopting Boot Block for their products, more than 20 PC manufacturers have gained added flexibility and the ability to lead a highly competitive

market [1][5]. End-users also benefit from the ability to upgrade BIOS software quickly and securely. The blocked architecture allows the OEM to store critical system code securely in the lockable "boot block" of the device that can minimally bring up the device to initialize the system. The hardware boot locking feature guarantees that even if the power is disrupted during a BIOS update, the system shall be able to recover immediately.

Recently, Intel introduced the 2 Mbit 28F200BX , 4 Mbit 28F400BX , and 8 Mbit 28F800BV (SmartVoltage) Boot Block products that provided fast speed, high density, low power, surface-mount options and an industry-standard upgrade path for portable computing and telecommunications [7].

These products offer 60 ns performance; two surface mount packages: 40-lead TSOP (Thin Small Outline Package) and 44-lead PSOP (Plastic Small Outline Package); and a proprietary Boot Block architecture similar to that of the 1 Mbit Boot Block device. The Boot Block stores the code necessary to initialize the system, while the parameter block can be used to store manufacturing product code, setup parameters, and frequently updated code, such as system diagnostics. The main operating code is stored in the main blocks.

1.4 Solid-State Mass Storage

This major application type requires very high density memory, automated programming and high-performance erase/write capability at a very low cost per bit.

Erasing and writing portions of the code or data is much more frequent in solid-state mass storage than in the updatable firmware applications.

The symmetrically blocked 8 Mbit 28F008SA Flash-File memory is one of the highest density nonvolatile read/write solutions for solid-state mass storage. Additionally, it is the first flash memory device optimized for solid-state storage of software and data files [1].

The 28F008SA is packaged in an advanced 40-lead TSOP (Thin, Small Outline Package) of 44-lead PSOP (Plastic SOP) to provide the extremely small form factor required for today's hand-held, pen-based and subnotebook portable computers. The compactness of an 8 Mbit device in a TSOP package allows for high-density flash arrays to be included both on a system motherboard, for direct execution of user programs or operating systems, as well as memory cards for transportable program and file storage.

1.5 The Problem with Flash Memory Firmware

Flash memory is block erasable only and can not fulfill normal RAM functionality. The flash erasure process sets all memory bits within a block to a '1'. So, a single flash address can not be updated individually. However, this constraint may not be decisive in some applications.

There are some concerns when using flash memory as system firmware, particularly regarding the ability safely perform updates, as the firmware can be lost during

the update process. Because flash memory must first be erased before it can be reprogrammed, the system can be rendered inoperable if something disrupts the firmware update process before reprogramming is complete. (e.g., a power glitch occurs, the communication link providing the updated code is lost, the system software interrupt vectors are replaced by flash memory programming status, or an asynchronous reset occurs.) Without the firmware, the system will not operate.

1.6 Solution to the Problem

Given the importance of avoiding firmware loss during the update process, I have focused my research on developing a solution which can prevent it. This is crucial in instruments designed for totally remote updates. In order to complete the firmware update processing without the loss of firmware and system lockup, both a good system hardware architecture, and a highly reliable system software design are required.

Herewith are presented three different schemes for memory mapping, and the software to be used as the boot code. The boot code is resident in the flash memory boot block, which performs the firmware update processing. The three memory mapping schemes are (in the order in which they are presented):

a) ***Data Block***;

b) ***Mode Page***;

c) ***Flash Window***.

The boot software in each case performs power-up, system hardware and software initialization, RAM test, internal register test, flash memory test, updates to system firmware (as needed), and handles all maskable and non-maskable interrupts (including power glitch, software trap, asynchronous reset, etc.). Because the software design technique preserves all necessary interrupt vectors at all times, the system is always ready to respond to any kind of maskable or non-maskable interrupt. This technique will be discussed in detail in pursuant paragraphs.

The ***Data Block*** architecture was designed for straight memory mapping. In other words, there is no memory block swap between RAM and flash memory. The ***Data Block*** architecture does not provides extra memory to the processor. This prospect prompted the development of a new design called ***Mode Page***.

The ***Mode Page*** design can provide extra memory to the processor, because the system has more than one memory mode. ***Mode Page*** needs only a very simple hardware modification. However, it was difficult to develop the boot software, and this could not usually handle a non-maskable interrupt.

The final design leap came with the ***Flash Window*** architecture, which not only provides extra memory to the processor, but also prevents non-maskable interrupts from causing system lockup.

The ***Data Block*** is a simple way of utilizing flash memory to store system software or data parameters. The ***Mode Page*** architecture is an economical hardware design that provides extra memory to the system, beyond what is directly addressable by the processor. ***Data Block*** and ***Mode Page*** are simple to implement and easy to load with firmware code. However, neither design can completely prevent the loss of system firmware or system lockup during an update. The Flash Window architecture is the most complete solution of the three and therefore will be treated here in greater detail.

The ***Flash Window*** architecture and Boot Software are subsystems which enable system recovery, even if the firmware upgrade is disrupted. The hardware-protected boot block guarantees that code stored in the boot block cannot be erased. The boot block is designed to store key kernel code that initiates the system and, if necessary, starts a recovery routine to restore firmware. Although firmware updates may be infrequent, and the likelihood of disrupting the firmware update process is remote, the consequences are severe. The alternative to using hardware and software-protected boot block flash memory is to risk losing firmware without the chance of recovery. If the firmware is completely lost, then the product must be disassembled and the memory replaced. Obviously, this can be very costly to both the end-user and the manufacturer.

The ***Flash Window*** architecture and Boot Software not only address normal enable / disable interrupt conditions, they also handle non-maskable interrupts. In other words, under no condition can the system lose the firmware and become disabled. Preventing system lockup is a primary concern in the design of system hardware and

software, and most systems are not protected against in the case of a remote update. It is very important in the case of some systems to be able to update the firmware via remote access. During a remote update, the loss of system firmware can be catastrophic. The ***Flash Window*** architecture and Boot Software protect against such catastrophes by keeping necessary interrupt vectors active during the update process, when the system is most vulnerable.

1.7 Hardware Consideration

For my research I chose the Intel 80KC196 processor, which has been successfully used in many subsystems. The 80KC196 processor is a 16-bit Embedded Controller, whose memory address lines can handle only 64K of memory. Because of the ***Flash Window*** memory architecture, the CPU can, in effect, access much more then 64K memory locations with very simple hardware modifications in a way that is transparent to the system application software developer.

The flash device that I used is the Intel 28F001BX-B. This is a boot block flash memory with a total 128K (byte) memory. This flash memory requires a +12v (Vpp) source for memory erasing and programming, and a +5v (Vcc) is needed for supply voltage. For the memory standby, the typical current is 30μA and the maximum is 100μA (Vcc=5v). For the memory programming and erasing, the typical current is 6mA and the maximum is 30mA (Vpp=12v). The 28F001BX has a 100,000 erase/program life cycles per block [1]. At the time of this writing, the cost of this flash memory is no more than an EPROM which has the same size of memory.

Today's system designer, software developer, and manufacturer are faced with a number of options when choosing a flash memory solution for updateable firmware. The design and implementation of flash memory system architectures must meet the specific requirements of different applications. These three designs could serve as models.

1.8 Organization of the Thesis

Chapter Two and Three are devoted to the ***Data Block*** and ***Mode Page*** models, respectively. Chapter Four focuses on the ***Flash Window*** architecture and its associated Boot Software subsystems. The Boot Software is discussed in Chapter Five, and Chapter Six presents the conclusions of this thesis and some suggestions for future work.

In Appendix A, I have included the computer program used in Chapter Five to perform hardware initialization, flash testing, and system firmware updating.

Chapter Two
Data Block

2.1 Goal of the Design

When it comes to designing better products, flash memory offers exceptional performance and the lowest cost per bit. Boot block flash memories provide updateable code and data storage for a wide range of applications including cellular phones, modems, PC BIOS, automobile engine control and many others. System designers reduce system cost and improve reliability by using flash memory parameter blocks to replace EEPROM for parameter data storage.

Using software techniques described in this chapter, designers can replace EEPROM with flash memory parameter blocks in many applications that previously used EEPROM for parameter data storage and subsystem software. For example, cellular phone designs use flash memory parameter blocks to store data such as telephone numbers, time of use and user identification information. Automobile manufacturers use flash parameter blocks in engine control applications to store fault codes and engine optimization parameters. In each of these cases, manufacturers save both EEPROM component and inventory costs by using boot block flash memory for parameter storage in addition to storing application code. Additionally, improved reliability is achieved with lower system device and pin counts. Finally, the amount and frequency of parameter storage are improved.

This chapter describes the first subsystem, in which I used a flash boot block for storing parameters and system software. I used the 28F001BX-B flash device and the 80C196KC microprocessor (see Figure 2.1).

Flash technology brings unique attributes to system memory. Like ROM, flash is nonvolatile, retaining data after power is removed. Like RAM, flash memory is electronically modifiable in-system, and is readable and writable on a byte-by-byte basis. However, unlike RAM, flash cannot be rewritten on a byte-by-byte basis. It must be completely erased before it can be rewritten.

2.2 *Data Block* Architecture

In this first subsystem, Boot Block flash memory is used to store the updateable system software, namely the kernel code. The hardware-lockable boot block is protected by both hardware and software. The kernel code is necessary to initialize the system and invoke a recovery routine if the application code is lost. The boot block also stores the code necessary to program and erase the flash memory. Because this hardware architecture allows no memory swapping and no code reallocation, we called this approach ***Data Block.*** Some texts refer to this design as ***FlashFile*** [3].

The partitioning of the memory mapping depicted in Figure 2.1 as follows: 0000h-1FFFh is RAM, 2000h-3FFFh is the flash boot block, 4000h-4FFFh is the flash parameter block 1, 5000h-5FFFh is the flash parameter block 2, 6000h-FDFFh the flash code block, and FE00h-FFFFh is the I/O port.

In this hardware design there is slightly under 64K bytes of fixed processor memory. Addresses 0000h-1FFFh are microprocessor internal registers and system RAM. 2000h-3FFFh contains the resident system kernel code and flash program code. 4000h-4FFFh and 5000h-5FFFh contain stored system data. 6000h-FDFFh is used for system application code. As a consequence, available remaining code space actually amounts to much less than 64k bytes. Due to this limitation, software developers have to design system code which requires a minimum of RAM and code space while still fulfilling system requirements.

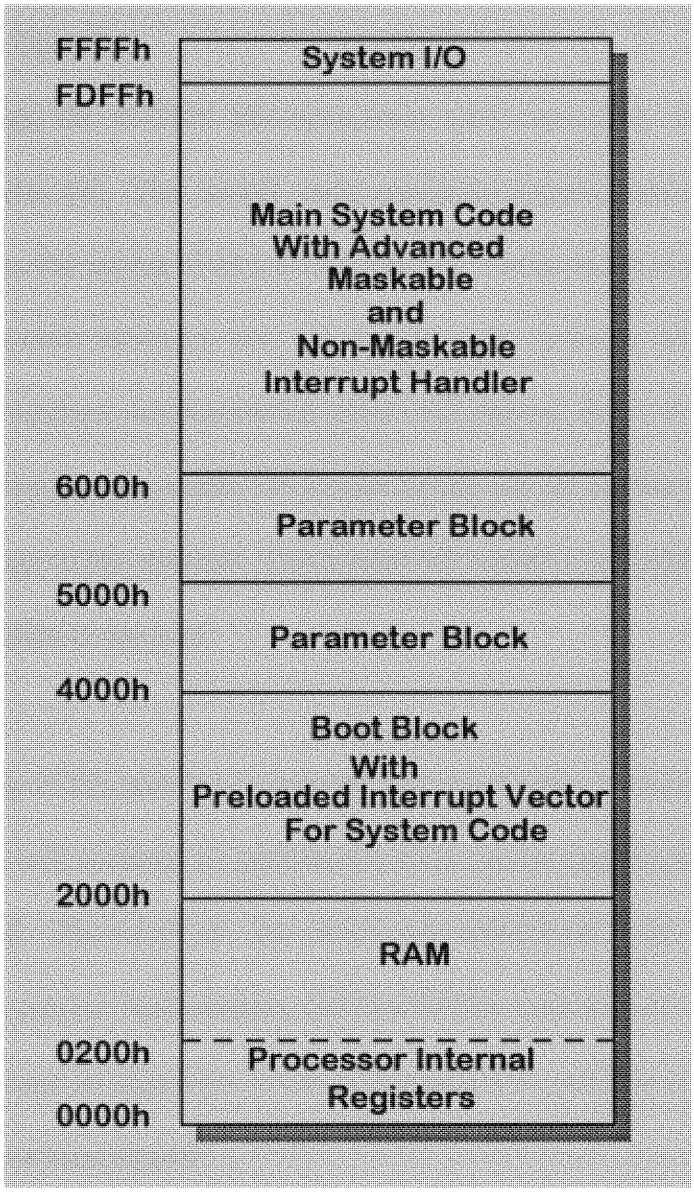

Figure 2.1 Data Block System Board Memory Map

2.3 *Data Block* Operation

On power-up, the 80C196 Microprocessor starts execution at location 2080h (system kernel code) [4]. The kernel code performs system hardware testing and initialization, such as the testing of all internal registers, communication ports, and system RAM. After the system hardware test, the kernel code performs initialization of the processor internal registers, communication ports, and internal flags. If any of the above testing or initialization processes fails, the system kernel will set an internal error flag and send an error message to the operator. The system will remain in a disabled state and wait until the error condition is resolved.

If the testing of the system is successful and initialization runs to completion, the system kernel code will continue to perform the main system (application) code check-sum calculation task. The result of this check-sum calculation is the only indicator of whether or not the current flash memory (6000h-FD00h) contents are valid. The check-sum calculation utilizes an algorithm which is guaranteed to produce a different result if one or more bits have been altered.

The stored flash memory check-sum test indicates whether or not the system software (application code) stored in the flash memory is valid. If the new check-sum result matches the stored value, the kernel code will set a flag and pass execution on to the main system block, and the application code will take over the task of system operation.

If the check-sum calculation does not produce a matching value, this means that either there has never been a download and the stored code is blank, or the application code has been corrupted. In either case the system will require a new download to re-program the flash memory. The system kernel will notify the operator of the state of system memory, at which point the operator should command a download.

2.4 Undesired Interrupt Occurring for Remote Updates

Modern flash memory products do not as yet provide the capability to read from one address location in the flash device while writing to another address in the same device. This means that any code that writes to flash must be downloaded to RAM. Therefore, to program or erase the flash memory, the boot code must be executed out of RAM (see Figure 2.2). The amount of RAM required depends on the complexity of the parameter storage data base. The code that must be downloaded from the system kernel (boot block) to RAM includes the flash memory read, write and erase routines.

So, in effect, during flash memory programming , the ***Data Block*** approach disables all Maskable interrupts. This yields a satisfactory result provided no interrupts are generated during the update, in which case the system software download process will be successful.

The kernel code includes preloaded interrupt vectors for the Maskable and non-Maskable interrupts. If an interrupt is generated, then the appropriate interrupt handler function will process the requested interrupt. The interrupt services are designed for

individual service needs. For example, if a power glitch occurs during normal system operation, an interrupt will be generated. The interrupt handler will perform a quick save of the last known operating state flag and set a delay to wait until the power returns to normal. Then it will restore the original system condition. Serial communication, keypad interfaces, and electrical sensors can be designed to operate under interrupt services.

All interrupt vectors are preloaded and fixed in the kernel code. In order to perform an update the contents of flash memory, including the interrupt handler routines addressed by these vectors, must be erased. Therefore the boot block (kernel code) can not be updated remotely without the risk of system failure due to an inopportune interrupt. In-house updates do not present as much of a problem with the Data Block architecture because the user can intervene in the event of a lockup.

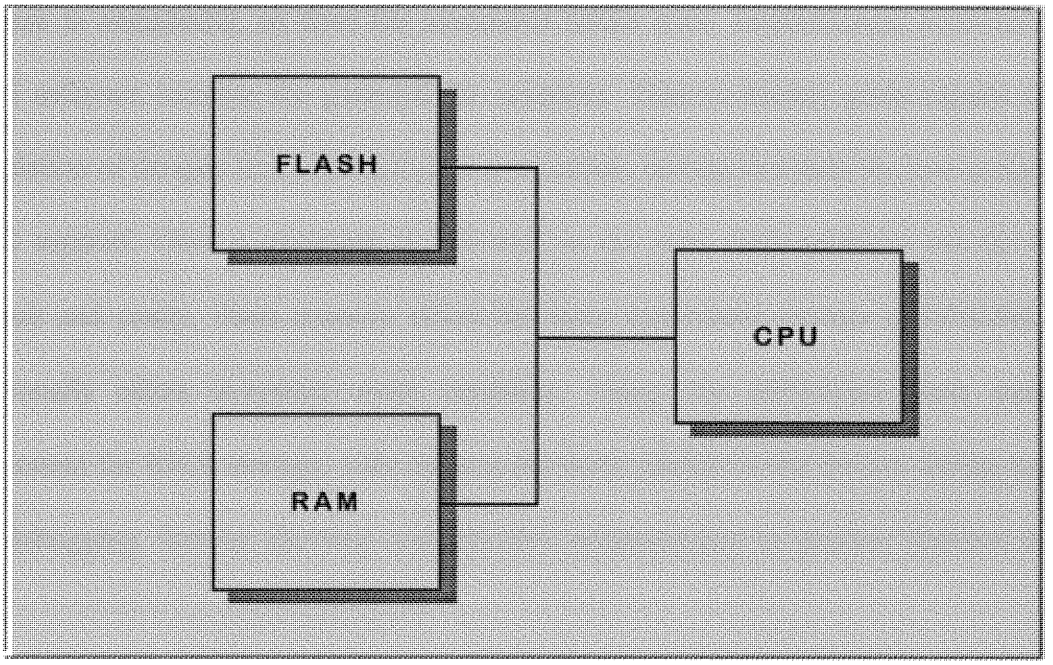

Figure 2.2 For In-System Write to Flash, Code is Executed out of RAM

If a power glitch occurs, a system designed with the ***Data Block*** approach will fail. The hardware and software have not been designed to handle non-Maskable interrupts during the system software update process. If during flash memory programming and/or erasing, a non-Maskable interrupt occurs, the interrupt vectors at location 2000h-3000h will not be available. This will cause the system to fail. This problem must be resolved in order to offer the facility to safely perform remote updates.

2.5 PC BIOS Update by Using of a Similar ***Data Block*** Design

In spite of its limitations, a similar ***Data Block*** design has been utilized in many modern systems. The PC BIOS code update is one example. As PC computing platforms increase in complexity, so does the associated BIOS code. Sophisticated hardware and BIOS software increase the potential for revisions. Time-to-market goals require faster completion of designs from conception to production, leaving less time for new-peripheral BIOS support. Once a computer is delivered to the user, code revisions are far more difficult and costly. Flash memory offers the same nonvolatile storage as EPROM, but additionally offers in-system write capability. Using the 28F001BX flash memory component for BIOS storage, code updates are done quickly in the factory during testing and debugging, thereby allowing cost-effective enhancements to product features in the field.

The previous design assumed that (DFFFFh-EFFFFh) RAM was available to the system. In Figure 2.3, we see that the BIOS is actually stored in the main block of the flash component from system address E0000h through FBFFFh. In this scenario, the processor jump vectors, BIOS checksum, and recovery code are stored in the 8 K byte boot block. This is the area the processor will jump to on power-up or after reset. The boot block code will execute the checksum routine in the main block to verify that the BIOS is valid. If successful, the processor will check the system RAM, copy the main block code to high memory DRAM, and jump to this area for the remainder of the Power On Self Test (POST), as well as any further BIOS calls. Optionally, the 28F001BX can then be disabled to save power.

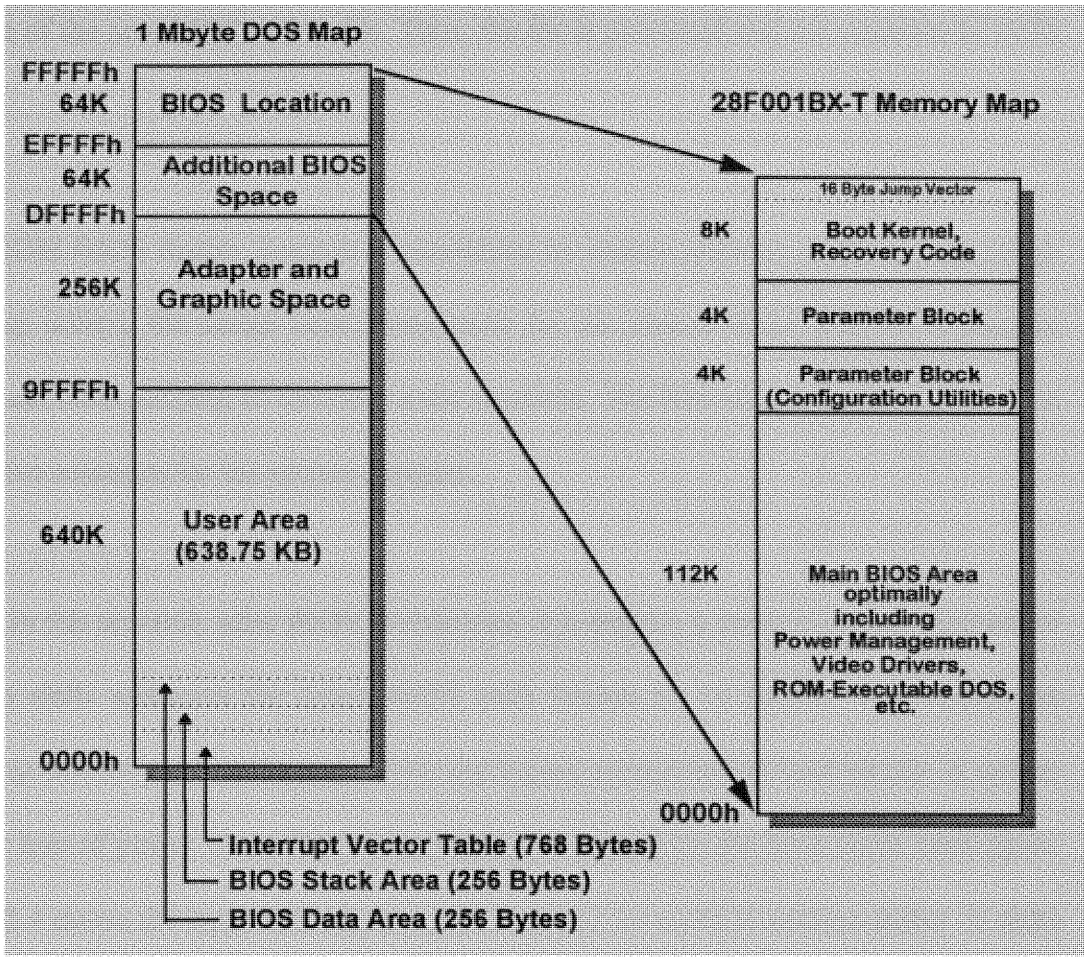

Figure 2.3 28F001BX-T in 1 Mbytes DOS Memory Map

If the BIOS checksum reveals an invalid or corrupt BIOS, the system RAM and floppy drive (or possibly modem) are initialized using the boot block recovery code. The system requests (through screen display or speaker "beeps") that the user install the BIOS update floppy disk. A search of the floppy disk is made the file name of the update code, and once found, the update code is used to re-initialize the main BIOS block. A system reboot restores normal operation. Alternatively, the BIOS recovery code could contain the

location of the new BIOS update file. Thus, the file is protected and not readable by DOS users.

If the ROM BIOS disable function is overridden by system software or the user (through the setup utility), the design must compensate for the altered BIOS location to prevent BIOS calls from jumping to incorrect code locations. The following two methods provide alternative solutions for the system designer.

In this scenario, after BIOS initialization is complete, a write to a latch, register or flip-flop shifts addresses for future BIOS code fetches by 16 Kbytes. This allows the system to correctly access the main block and bypass parameter and boot blocks. A system reset or loss of power clears this latch, allowing booting from the boot block once again. Figure 2.4 shows the input and output signals required for a μPLD address shifter. It shifts addresses in the range FFFFFh-E4000h (112 Kbytes) to FBFFFh-E0000h.

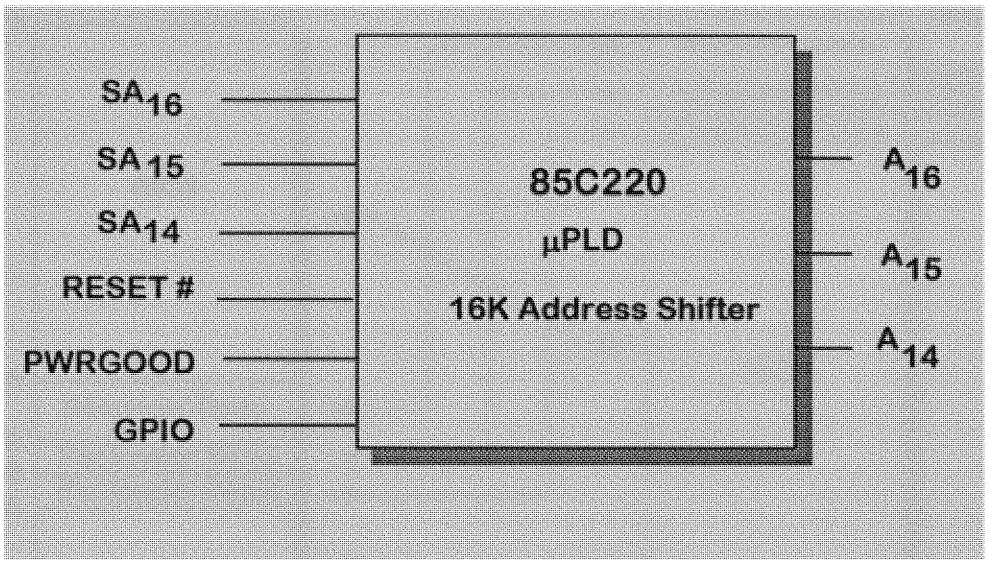

Figure 2.4 Address Shift Circuitry

Figure 2.5 presents an alternative approach to configuring the 28F001BX in the system memory map. This configuration follows the guidelines suggested by the flash manufacturer, but has the specific characteristics needed for the implementation of the Data Block architecture. Simple inversion of address line A_{16} to the 28F001BX moves the boot block to the lower half of flash memory as seen by the system. In normal operation, the processor boots and executes from the main array areas, which store the system BIOS, video BIOS and/or DOS in ROM.

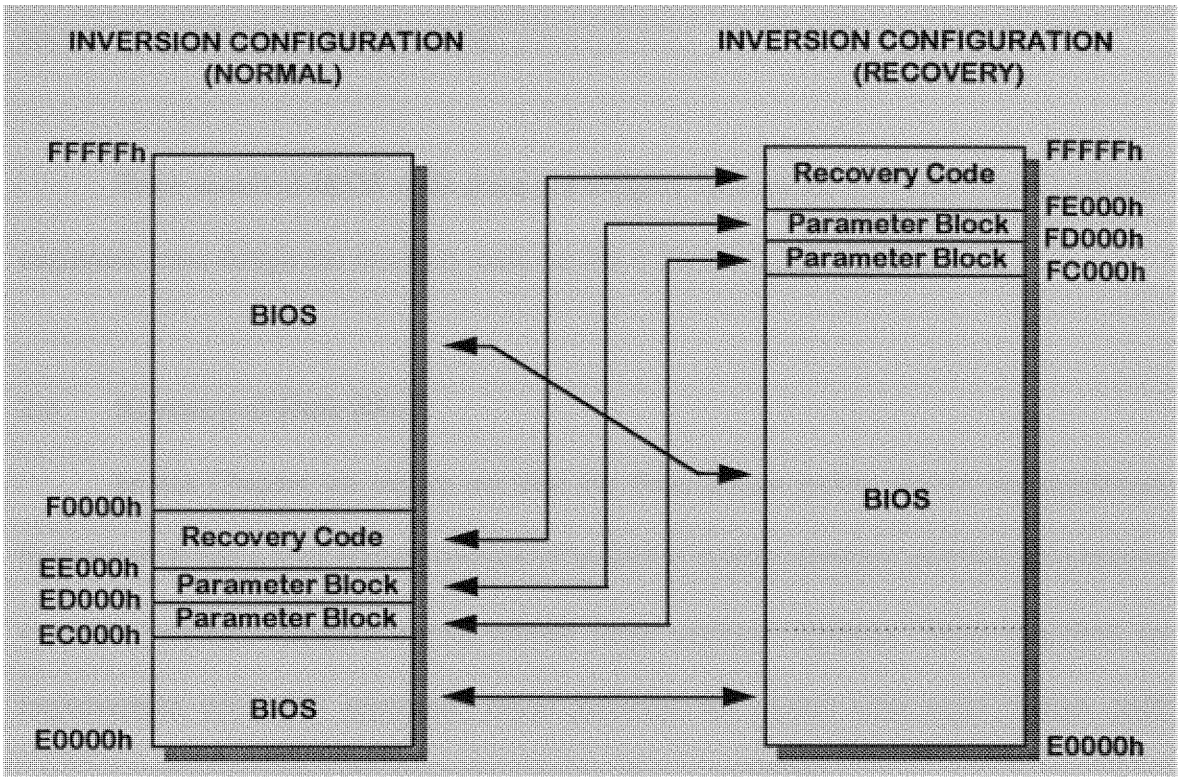

Figure 2.5 Memory Mapping Configuration

If power loss aborts a BIOS update, the main array block will be partially programmed/erased and the code in this block unusable. The system will "hang" or not boot at all. To boot from the boot recovery block, the polarity of address line A_{16} must be restored, producing the memory map shows in Figure 2.5. A keyboard sequence, switch on the back of the PC or jumper on the motherboard can toggle A_{16} restore logic and "un-invert" it, After reconfiguration, the processor boots from the boot block and executes its

recovery algorithm to restore main array block contents, Re-inverting A_{16} reinstates normal system boot-up and operation.

Since standard BIOS code does not support boot block recovery, BIOS software engineers must design the recovery code for the 8 Kbyte block.

2.6 Summary of *Data Block*

In this Chapter we clearly see that the ***Data Block*** system design has significant limitations. It has a very limited amount of available system memory, and fails to preserve essential interrupt handlers during system code updates. Chapter Three describes an improvement upon this approach, which provides the system with more system code memory and more comprehensive interrupt handling capabilities.

Chapter Three
Mode Page

3.1 *Mode Page* Design Layout

In the previous chapter we explored the first system (***Data Block***), which uses the boot block architecture for safe firmware updates, and gave the example of a PC BIOS update using a boot block flash. That system provides some desirable features/qualities, such as, simple hardware layout, and easy software modification. However, the system has certain limitations: there is limited system (application) code memory space, there is no guarantee of firmware update completion, and the system may fail at times, during the programming of the flash, when interrupt vectors and interrupt handlers are missing. It is still an acceptable design as long as the system is not updated remotely. If an on-site system firmware update fails, the system can always be reset by shutting off the power and then turning it back on again. In the case of a remote update, if the system firmware update fails, then there will be a problem. There is no way to reset the remote system. These concerns prompted me to improve the design of the system and robustness of the software. I had two goals for the next system design. The first one is to provide more system memory space and the second is to improve the stability of system firmware updates. The remainder of this chapter presents the second generation system which I designed with the above considerations in mind.

This subsystem is broken into three modes in which the system is operational, each representing a different system memory mapping. The three modes are Boot (Power-Up) Mode, Standard (Normal) Mode, and Extended Mode. The type of microprocessor and flash memory used is the same as in the first system (Intel 80C196KC and Intel 28F001BX). The processor's accessible memory locations are different in each memory mapping. (See Figure 3.1, Figure 3.2, and Figure 3.3).

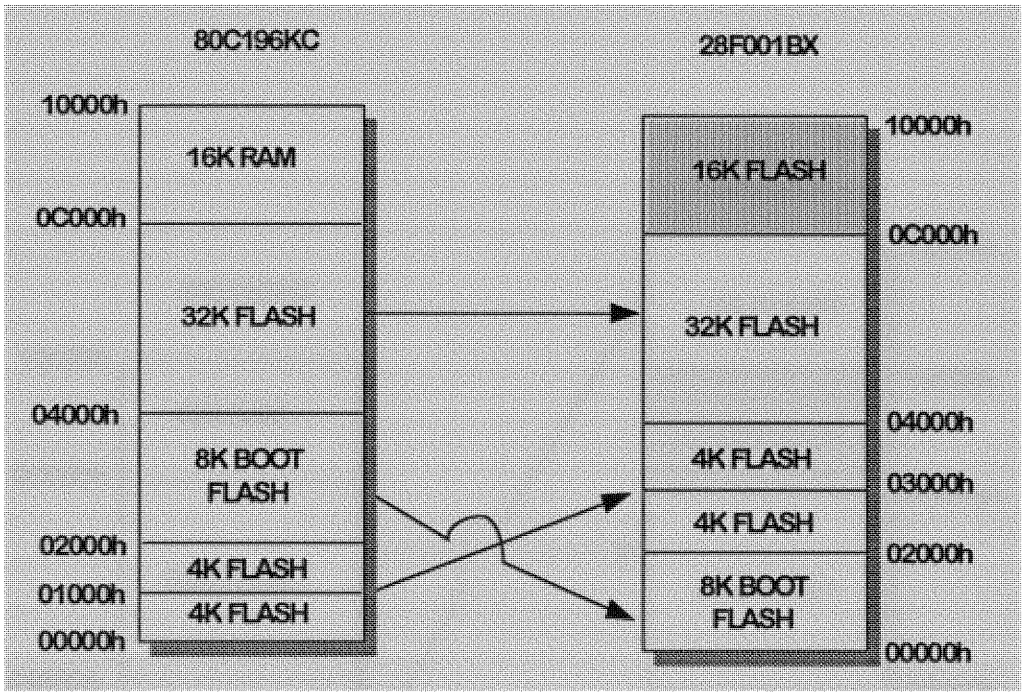

Figure 3.1 Boot Mode System Memory Map

The memory mapping depicted in Figure 3.1 is known as Boot Mode, or alternately as Power-Up Mode. In contrast with the mapping for ***Data Block*** , this mapping was specifically designed for the Boot Mode in the ***Mode Page*** approach. Every time the system is powered up, the system memory mapping will be in Boot Mode. The

boot block of the flash is accessible only in this mode. As we notice from the Figure 3.1, the rouse 0000h-01FFh is covered by the microprocessor internal registers, 0200h-0FFFh is flash parameter 1 block (physical flash location 02200h-02FFFh), 1000h-1FFFh is the flash parameter 2 block (physical flash location 03000h-03FFFh), 2000h-3FFFh is the flash boot block (physical flash location 00000h-01FFFh), 4000h-BFFFh is the lower part of flash main block(physical flash location 04000h-0BFFFh), C000h-FEBFh is the system RAM, FEC0h-FEFFh is the battery backup RAM (64 bytes), and FF00h-FFFFh is the I/O port.

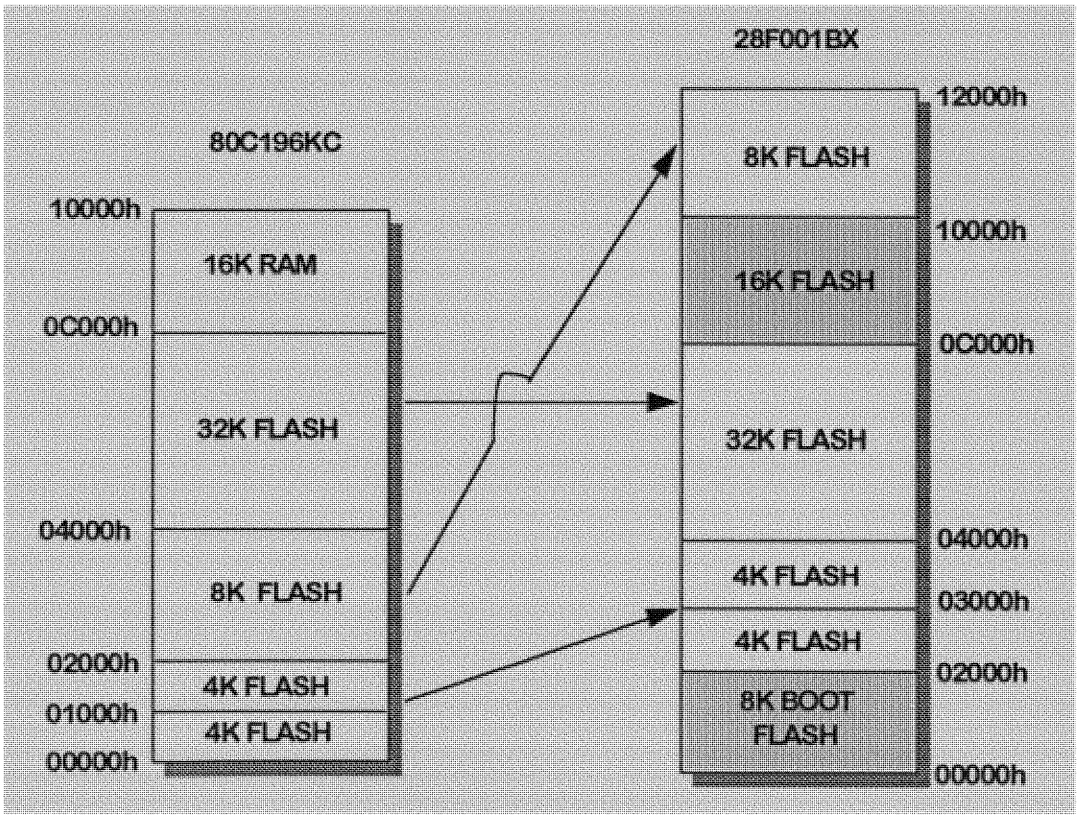

Figure 3.2 Normal Mode System Memory Map

Figure 3.2, represents the Standard Mode memory mapping. 0000h-01FFh is the microprocessor internal register, 0200h-0FFFh is the flash parameter 1 block (physical flash location 02200h-02FFFh), 1000h-1FFFh is the flash parameter 2 block (physical flash location 03000h-03FFFh), 2000h-3FFFh is the middle part of flash main block (physical flash location 10000h-11FFFh), and 4000h-BFFFh still is the lower part of flash (physical flash location 04000h-0BFFFh), C000h-FEBFh is the system RAM, FEC0h-FEFFh is the battery backup RAM (64 bytes), and FF00h-FFFFh is the I/O port.

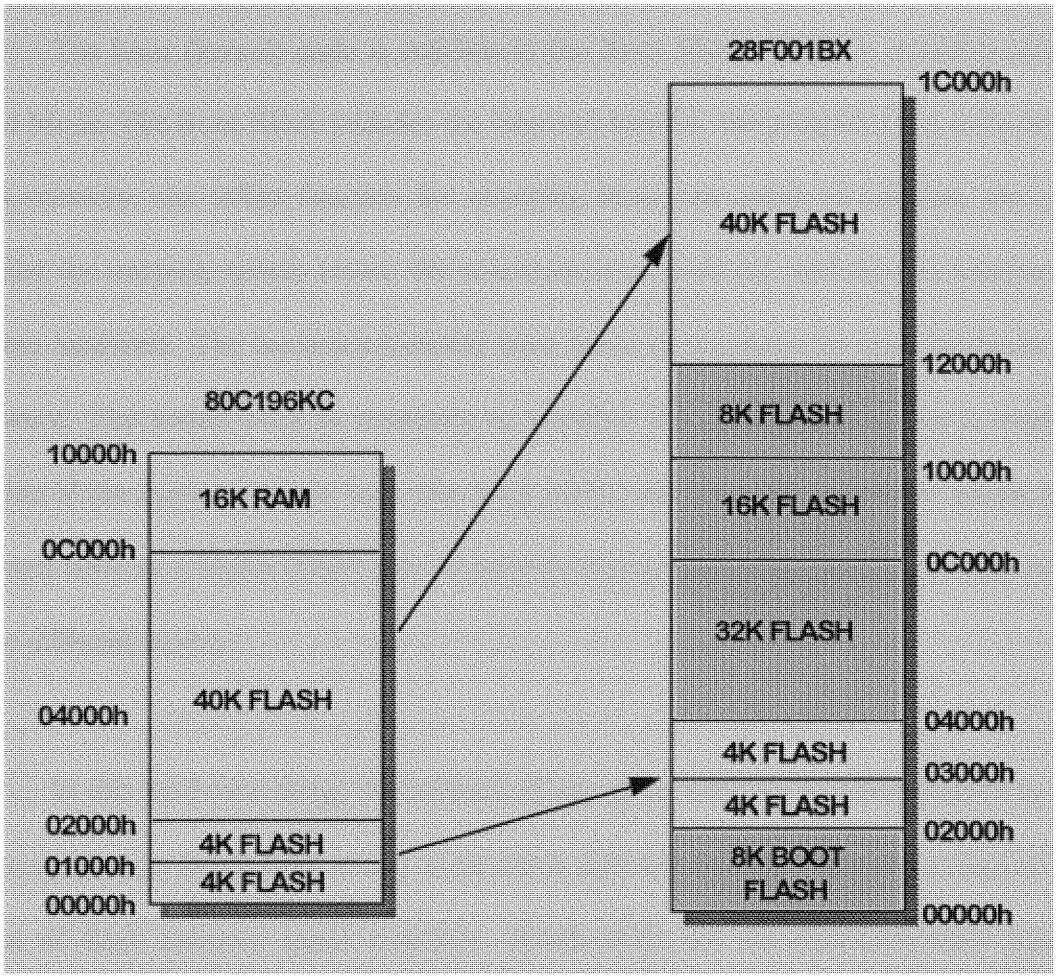

Figure 3.3 Extended Mode System Memory Map

The Extended Mode memory mapping is diagramed in Figure 3.3. 0000h-01FFh is covered by the microprocessor internal registers, 0200h-0FFFh is the flash parameter 1 block (physical flash location 02200h-02FFFh), 1000h-1FFFh is the flash parameter 2 block (physical flash location 03000h-03FFFh), 2000h-BFFFh is the upper part of flash main block (physical flash location 12000h-1BFFFh), C000h-FEBFh is the system RAM, FEC0h-FEFFh is the battery backup RAM (64 bytes), and FF00h-FFFFh is the I/O port.

The boot (kernel) code is resident in the Boot Block which is available only in Boot Mode. The Boot Block includes the interrupt handler, the programming software for flash memory, the utility function for changing from one mode to another, the error handler for trapping system errors, the communication protocol, and the test function for system hardware. The hardware-lockable boot block is protected by both hardware and software. The kernel code is necessary to initialize the system and invoke a recovery routine if the system code is lost. The boot block will store the necessary partial boot (kernel) code to a higher RAM area to operate during flash programming in each of the three different modes.

In this hardware design the address of any given memory location relative to the system processor is not fixed in any of the three different modes. There are a total of 88K bytes of flash memory space available. 0000h-1FFh are the microprocessor internal registers and C000h-FEBFh is the system RAM, which is equivalent to 12K bytes total RAM. This is an improvement over the previous (***Data Block***) design. In Boot Mode, the system boot (kernel) code and flash program code reside at 2000h-03FFFh. 0200h-0FFFh and 1000h-1FFFh are for storing the system data. 4000h-BFFFh holds the system application code. In Standard Mode, 2000h-3FFFh can be used for system code. In Extended Mode, 2000h-BFFFh has an additional 40K bytes of flash memory available to the processor, consequently, there are more than 64K bytes of available code space. By employing all three modes, this design provides more application code space and system RAM when compared with the ***Data Block*** system design. This architecture will be

referred to as ***Mode Page***. The ***Mode Page*** architecture gives software developers freedom to design more complex system code, which demands a larger memory space in order to complete for fill the necessary system requirements. The reason we call this design ***Mode Page*** is because the layout of the memory mapping is like a page in each mode.

The ***Mode Page*** design involves the integration of more complex system software. Hence the boot (kernel) code and system (application) software become more difficult to develop and debug. In particular, the boot code and system software always have to keep track of the mode in which the code is running. Before the real code is tested, we always have to know what type of interrupt could occur, and what interrupt handler routine should be placed in this memory mapping mode. Most of the program flow should be implemented as calls to functions. Also, these jumps must be directed to a location relative to the current memory address. The jump must be an offset rather than a fixed location since any section of the code, may correctly be mapped in a location different from the are originally assigned at compilation time.

How does this ***Mode Page*** system work? When the system is first powered-up, the hardware sets the memory up in Boot Mode (See Figure 3.1), The 80C196KC Microprocessor starts the first instruction at location 2080h (system kernel code). This kernel code immediately disables the maskable interrupts, because the system has not yet been tested and initialized. After disabling the interrupts, the kernel code performs a system hardware test and initialization, including the testing of all internal registers,

communication ports, and system RAM. If the system hardware test is passed, then the kernel code performs the initialization of the processor internal registers, communication ports, and system stack. If any part of the testing or initialization process fails, the system kernel will set an internal error flag in the system battery backup RAM and send error messages to the operator. The system will remain in a disabled state and will wait until the error condition has been removed.

Once the testing of the system hardware and initialization of the communication ports and system resources are completed, the system kernel code will reallocate (download) the flash program routines and the system (application) code check-sum routine to the system RAM area starting at C000h. This kernel operation represents the second stage of the process which begins with the system code check-sum calculation.

The check-sum is the net result of a word-wise logical operation performed over the entire contents of flash memory which are accessible to the processor (except the parameter blocks), and is used as a data fingerprint. The result of this check-sum calculation is the only indicator which shows whether the current flash memory is valid or not. This comprises all memory from the flash 28F001BX main block (Boot Mode 4000h-BFFFh, Standard Mode 2000h-3FFFh, and Extended Mode 2000h-BFFFh). Every time the system (application) code is updated, the boot (kernel) code performs this check-sum calculation, the result of which will be saved to a particular flash memory location which will be used for the (next) verification. If the check-sum calculation does not match the value of the check-sum which was preprogrammed from the previous system software

download, then this indicates that the content of the flash memory has been damaged. The check-sum calculation utilizes an algorithm which ensures that a modification to any bit in the flash memory code segment will cause a different result to be generated.

The kernel code does not include any preloaded interrupt vectors for the Maskable and non-Maskable interrupt service routines. One could design a system kernel code with interrupt service functions in this hardware layout, but any modification will be very difficult. In order to add additional interrupt service capability you would have to create a tremendous number of different interrupt routines for the different operational modes and different time intervals. The reason for not including the interrupt service in the boot code is because the execution of the boot code involves jumping from one mode to another. The interrupt handlers can only service interrupts in Boot Mode, and only at times during the hardware testing and initialization process. This is because the processor interrupt vectors are located between 2000h-3FFFh. Only in Boot Mode does the processor have direct access to the boot block (at location 2000h-3FFFh). From address 0200h to BFFFh, memory locations are not available for read during the system code update in any the three modes. Therefore, embedding the interrupt service in the boot code will not provide much of an advantage in terms of preventing system code update failure or system 'hang'. An additional, reason for not including the interrupt service in the boot code is to conserve boot code memory space. There is a total of only 8K bytes available for the boot code.

Assuming that no interrupts occur, the ***Mode Page*** design is a very reliable system for most instruments, such as printers, copy machines, fax machines, and biomedical

instruments. The design provides more memory than the processor can handle in the normal case. For example, a 16-Bit microprocessor can only operate 64K bytes.

For the system code, all interrupt services can be performed without any problems in the ***Mode Page*** design at mode operation. The system code resides in flash memory between addresses 2000h and BFFFh. All interrupt vectors for the process are included in this area. The ***Mode Page*** design allows the system to utilize processor resources in a very efficient way.

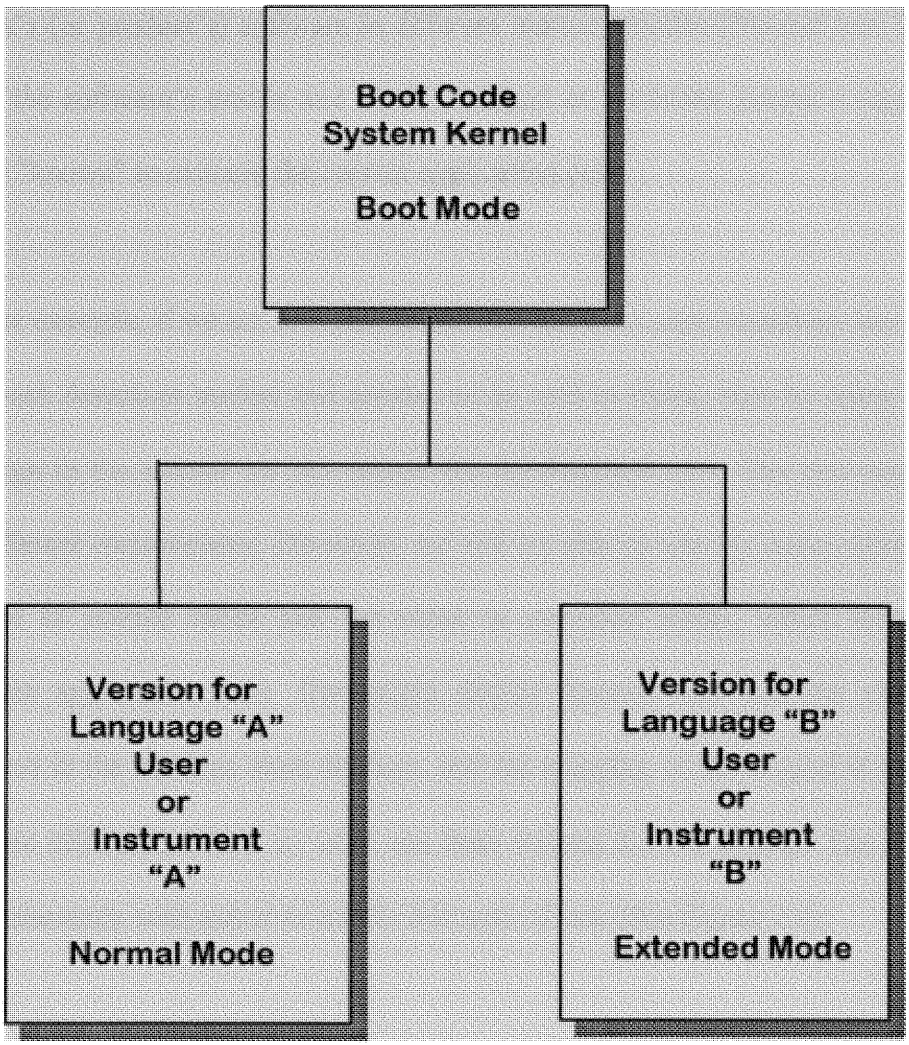

Figure 3.4 The Layout of Three Mode "Page"s in the System.

3.2 Dual System Application Reside in *Mode Page* Design

Also, because the ***Mode Page*** has the capability to swap from one memory mapping to another, this allows the system to have two totally different subsystem application software pages. See Figure 3.4. The beauty of having these two independent pages is that it makes the system flexible enough to fit into a variety of different products.

One single system board with the same boot kernel code can contain two different subsystem's (or product's) software in different memory modes (Normal and Extended). By using the software's enable and disable method or hardware jumper's signature, the processor "knows" to execute the correct application code.

For example, if we are to manufacture two different copy machines, one to be sold on the South American market and another to be sold in China, then by using this ***Mode Page*** design in the system we can reduce the development cost. The two markets have totally different requirements. However, we will be able to use the same system in both cases; one in normal mode and the other in extended mode, which harbor totally independent application code. One might display Spanish text and the other display Chinese text in the machine local displays, and the mapping of the keypad in the machine will operate differently based on what has been defined for the display. The Mode Page design is inherently well-suited to handle this type of task (See Figure 3.4).

Figure 3.5 shows the management of the boot code in the ***Mode Page*** system.

The Boot Code operations will proceed as follows (See Figure 3.5): The boot code will perform the processor internal register test, CPU I/O test, system hardware test, and system RAM test. After these tests, the processor is ready to perform the system flash memory test. This final task is the system application code verification, which we refer to as the 'check-sum calculation test'.

If the system application code check-sum testing produces an exact match, this means that the contents of flash memory remain identical to the original, which implies that the system application code is ready for execution. Now the boot code will examine the special jumper which provides necessary system information. The jumpers are designed to represent a combination of known facts which tells the boot code whether it should (or not) automatically execute the application code, and if so, in which mode. The information which is given by the jumpers can be in any combination; it is up to the boot code to make the appropriate decision. If the jumpers indicate that the kernel code is to perform the check-sum test only and always inquire from the operator, then the boot code will send an inquiry and wait until the operator gives the necessary information to start the system code execution.

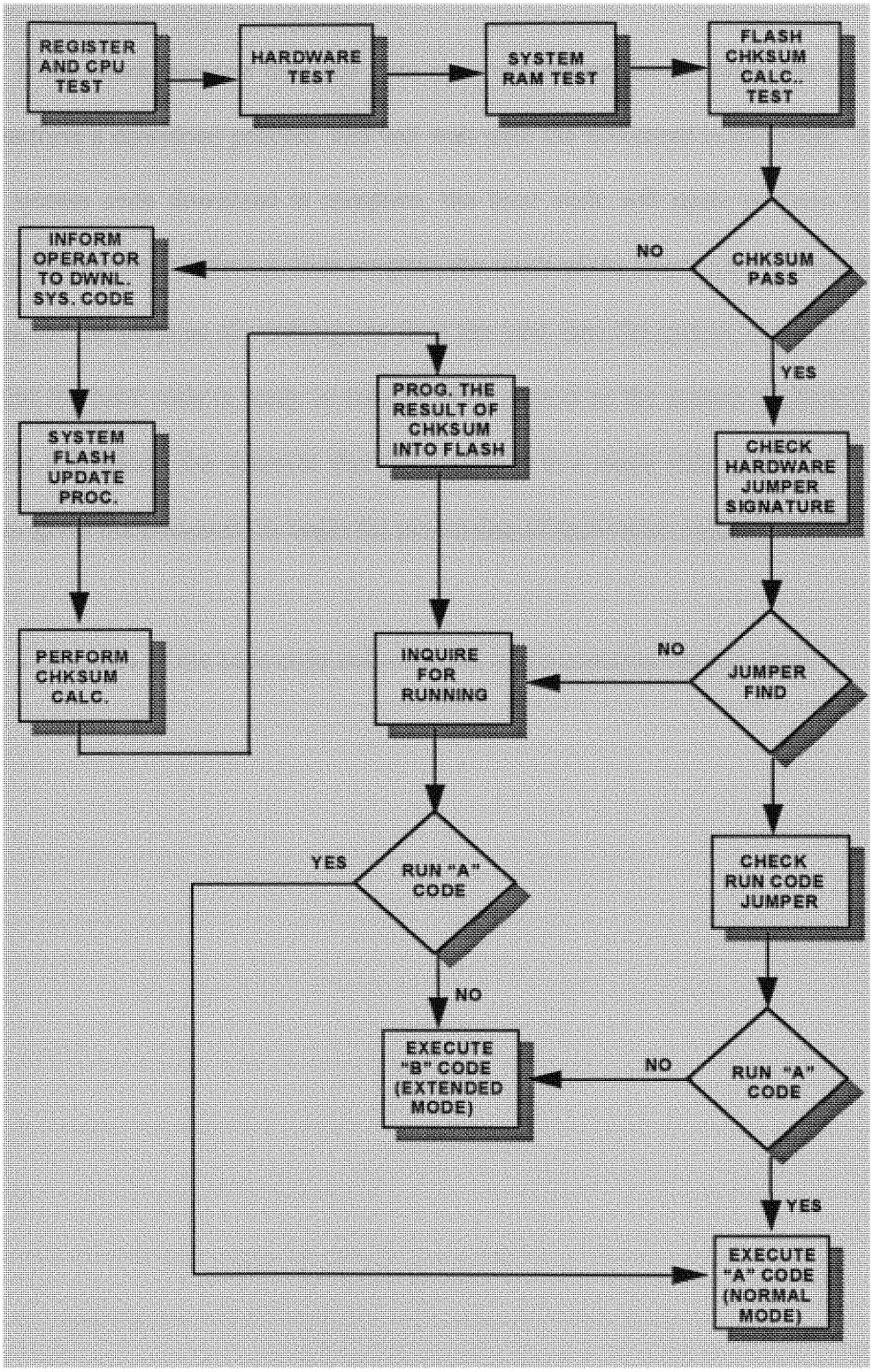

Figure 3.5 System Boot Code Operation for
Two Different System Application Programs.

If the check-sum test fails, the boot code will inform the operator that the system must perform a download process, and will wait until the download is initialized. As soon as the system code download is complete, the boot code will again perform the flash memory check-sum calculation, then program the result of the check-sum in a special flash location, for the purpose of verifying the flash the next time the system is powered on. At this point the boot code is ready to transfer flow control to the system application code. The boot will inform the operator that the system is in a ready state, then wait until the operator gives the command to begin the processing of application code.

The details of the boot code software will be discussed in Chapter Five.

Chapter Four
Flash Window

4.1 Introduction to *Flash Window* design

As mentioned in the introductory chapter the ***Flash Window*** approach and the Boot Software for this hardware design enable system recovery even if the system code firmware update is disrupted. This approach takes advantage of the incorporation of a hardware protected boot block. In general, the hardware-protected boot block guarantees that the code stored in the boot block cannot be erased. In this special application, the boot block is designed to store key kernel code that initiates the system and, if necessary, starts a recovery routine to restore the firmware.

Both the ***Data Block*** and ***Mode Page*** designs offered great value to us in the development of the ***Flash Window*** architecture, which has distinct advantages over the previous two designs. The ***Flash Window*** design provides an equivalent to 56K bytes of system RAM and 128K bytes of flash memory. This is almost three times as much as the 16-bit processor can normally handle. The memory map gives the software engineer an easy way to develop system application code. The ***Flash Window*** design is divided into two modes of memory mapping, which we called Boot Mode and Normal Mode (See Figure 4.1 and Figure 4.2). If only the 56K bytes of RAM memory are used for storing system code, the existence of these two mode will not be relevant to the designer. This is because the Boot Software initiates the system code every time, and then downloads the

system code from flash to RAM. The next chapter will discuss the software design in detail.

All the major hardware components remain the same as in the previous two system designs. In Boot Mode (Figure 4.1), the initial power-up condition initiates program execution (at CPU addresses 2000h-3FFFh) from the boot block of flash memory (physically located at address 00000h-01FFFh). In this mode, an 8K block of RAM (which is physically located at address 2000h-3FFFh, referred to as "RAM Page 1 Block") is mapped to CPU address A000-BFFFh. This allows the initial load of RAM-based interrupt service routines (called "Boot Handlers") and interrupt vectors for system application code. Additional RAM is available from address 4000h-9FFFh and C000h-FFFFh which may be used for mode switching and other purposes as required.

In the ***Flash Window*** architecture, the entire flash memory contents (128K) are readable via the window (from address 1000h-1FFFh) in both modes (Boot Mode and Normal Mode). This is an outstanding feature of this design as compared with the previous two designs. This distinguishing feature is the basis for the name ***Flash Window***. Boot Mode is always entered upon a hard reset or a software reset. In other words, any reset whether originating internally or externally will cause the system to enter Boot Mode. This guarantees that the system will always start the Boot Software and keep the system alive.

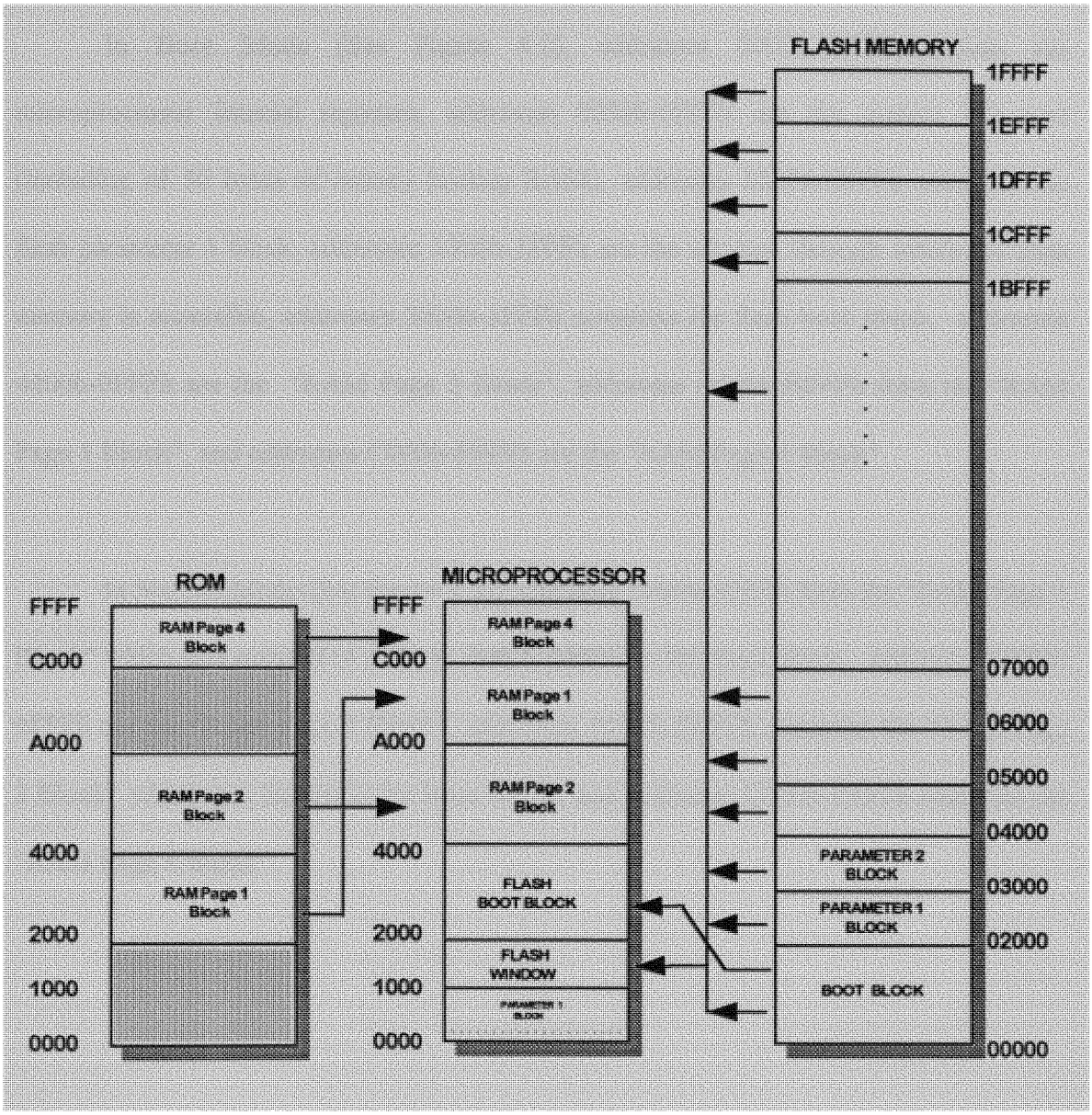

Figure 4.1 The Boot Mode of ***Flash Window***.

In Boot Mode (See Figure 4.1), addresses 0000h-01FFh contain the microprocessor internal registers, addresses 0200h-02FFh are the system I/O port (including of 50 bytes of battery backup RAM), addresses 0300h-0FFFh comprise the flash parameter 1 block, addresses 1000h-1FFFh are the window through which the flash memory is accessible, addresses 2000h-3FFFh comprise the flash boot block, addresses 4000h-9FFFh are the "RAM Page 2 Block", addresses A000h-BFFFh form the "RAM Page 1 Block", and addresses C000h-FFFFh are the "RAM Page 4 Block".

In Normal Mode (See Figure 4.2), addresses 2000h-3FFFh form the "RAM Page 1 Block" which was mapped to address A000h-BFFFh in the Boot Mode, addresses 4000h-9FFFh are the "RAM Page 2 Block", addresses A000h-BFFFh comprise the "RAM Page 3 Block" which was hidden in the Boot Mode, and addresses C000h-FFFFh comprise the "RAM Page 4 Block".

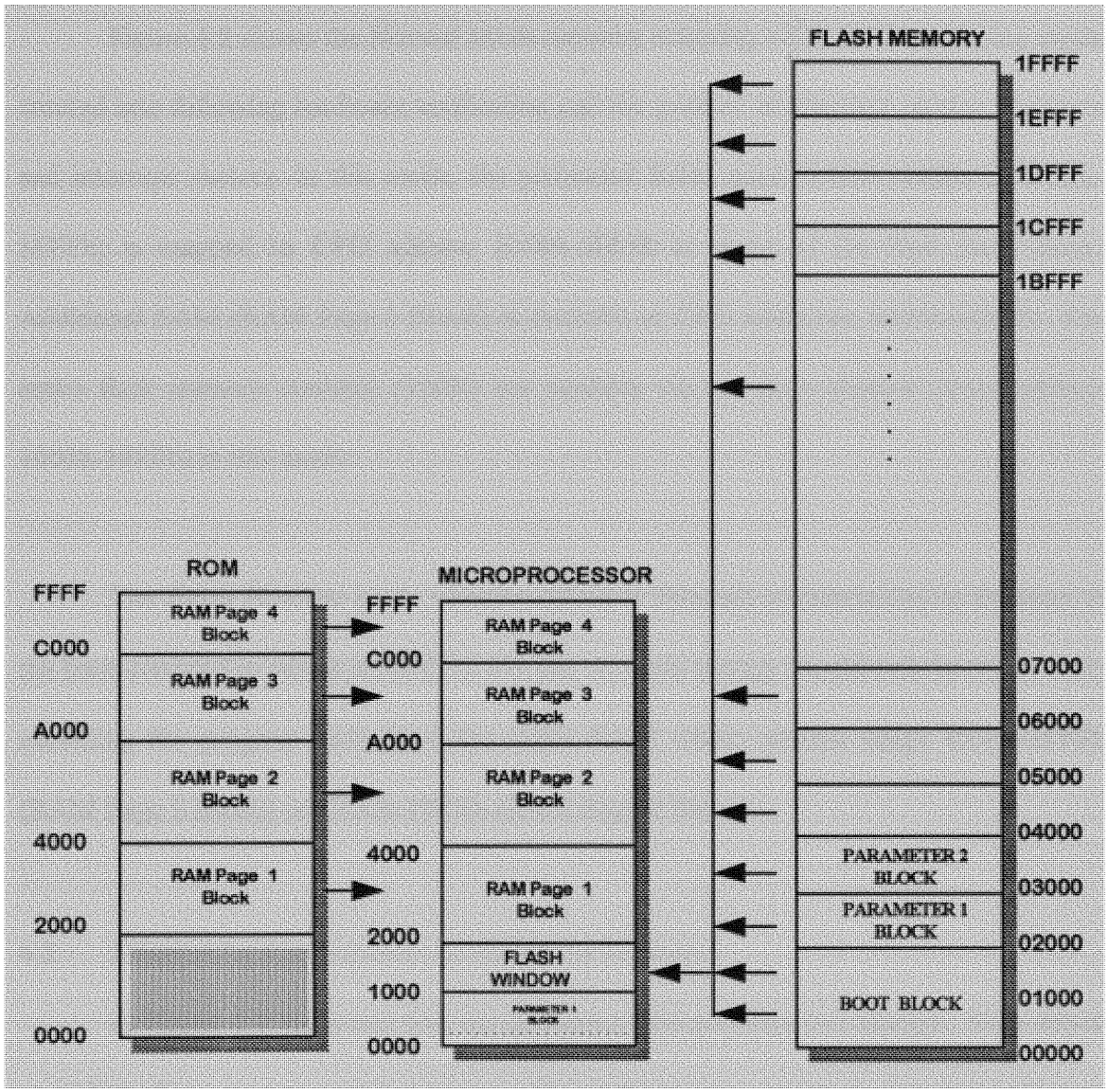

Figure 4.2 The Normal Mode of ***Flash Window***

Figure 4.2 shows the Normal Mode memory mapping which is the operating mode entered into following hardware verification and the preload of interrupt service functions and interrupt vectors. In this mode, the preloaded RAM page (which resides at 2000h-3FFFh) is mapped to CPU address 2000h-3FFFh in place of the flash boot block. Additional RAM (RAM Page 3 Block) is available at CPU addresses A000h-BFFFh. The entire flash memory contents (128K) are readable via the flash window (from address 1000h-1FFFh) in this mode.

The ***Flash Window*** mode selection (Boot or Normal) is determined by the state of the boot control line which is the CPU I/O pin. A mode change should only be made while executing the program code from the stable RAM within address range 4000h-9FFFh (RAM Page 2 Block) or C000h-FFFFh (RAM Page 4 Block).

4.2 Solution for Continuously Accepting the Interrupt

In Figure 4.2 we notice that the problem of the previous two designs, ***Data Block*** and ***Mode Page***, does not exist any more. The problem is encountered when responding to interrupts generated concurrently with flash memory programming or erase operations, when the interrupt vectors are located in flash memory. At these times flash memory and hence the interrupt vectors, are unavailable due to the operation characteristics of the flash components. Additionally, less than 88K bytes of physical 128K bytes of flash memory is accessible in the ***Mode Page*** configuration via the selection of three operating modes, and only 54K bytes of flash memory is available in the ***Data Block*** design.

The solution to the above problems from the ***Data Block*** and ***Mode Page*** configurations involves a fundamental change in operating architecture and requires that program execution is performed directly out of flash memory. The revised ***Flash Window*** architecture provides a RAM based program execution environment which can be loaded as desired from any location within the 128K bytes of flash memory. This is conceptually equivalent to the typical "PC" environment which provides for programs (and overlays) to be loaded from disk into RAM memory for execution. In our case, we solved the problem which we had previously simply by loading the interrupt vectors and interrupt service functions into the "RAM Page 1 Block" and staying in Normal Mode during flash memory programming and erase operations.

Following power-on or a system reset (internally or externally generated), the Boot Mode provides for initial program execution from the 8K bytes non-volatile boot block of flash memory. The primary purpose of the Boot Mode is to allow for the loading of up to 8K bytes of Boot Software (system kernel) code into RAM memory.

Before loading the Boot Software into the RAM area, the boot kernel code performs system hardware testing and initialization, such as the testing of all internal registers, communication ports, and system RAM. If it passes the system hardware test, then the kernel code performs the initialization of the processor internal registers, communication ports, and system stack. If any of the above testing or initialization processes fail, the system kernel will set an internal error flag in the system battery backup

RAM and send the error message to the operator. The system will remain in a disabled state and wait until the error condition has been removed. (See Figure 4.3)

4.3 Boot Code Operation

If the testing of the system hardware succeeds and initialization of the communication ports and system is completed, then the system kernel code will be loaded into the system RAM area which starts at address C000h or higher. Kernel operation enters the second stage in which it starts to perform the flash system (application) code check-sum calculation. The switching from Boot to Normal mode will be discussed in detail in chapter five.

The check-sum is calculated from the entire main block (112K bytes) of flash memory via the window at address 1000h-1FFFh. Every time an update of system (application) code in flash memory takes place, the boot (kernel) code will perform this check-sum calculation, then save the result at the end of the flash memory main block. The result will then be used for each successive follow-up verification will which occur every time the system is reset. The check-sum algorithm will be discussed in chapter five.

After the Boot Code finishes the check-sum calculation, it will inform the operator of the state of the system, and then perform the software block copy. This block-by-block copying operation serves to move the flash memory block into the system RAM. Once completed, execution will begin with the first instruction of the newly copied system code in RAM.

The way in which the CPU can access the system memory requires some explanation. In Boot Mode, the boot block resides at 2000h-3FFFh and the parameter 1 block at 0300h-0FFFh (physical flash memory location is at 02300h-02FFFh). The CPU accesses the flash main block (or entire flash memory 128K bytes 00000h-1FFFFh) via the flash window which is located at address 1000h-1FFFh. The entire flash is divided into thirty-two segments. Each of these segments is a 4K byte block which begins at addresses 00000h, 01000h, 02000h, ... , 1D000h, 1E000h, and 1F000h (these are all physical flash memory locations). Five bits of the data lines are used to control which block of the flash memory segment is showing through the window at the any given time. These data control lines can be assigned to a particular I/O location. The location that was selected for the prototype board is at I/O address 0250h.

For example, suppose we want to read 5 bytes from flash memory starting at address 14050h. Writing the value 14h (21st flash segment block starting from 0) into the control address 0250h, effectively selects the flash segment block 14000h-14FFFh to be accessible through the flash window. Then the desired 5 byte can be read by the CPU from locations 1050h-1055h.

The operation of the ***Flash window*** in Normal Mode is very similar. The difference is that in Normal Mode memory at addresses 2000h-3FFFh is used as a block of system RAM (RAM Page 1 Block), and A000h-BFFFh is used as an additional block of system RAM (RAM Page 3 Block). This makes the flash boot block not directly

accessible to the system in Normal Mode. In this mode, memory beginning at location 2000h to FFFFh is one contiguous RAM area and spanning a total of 56K bytes. This design is also ideal for some embedded systems which require a large amount of accessible RAM.

The ***Flash Window*** architecture provides 56K bytes of system RAM and access to the entire 128K bytes of flash memory. This offers a great capability for complex embedded applications, such as embedded video applications, embedded data acquisition, and embedded audio applications. The ***Flash Window*** is a very efficient solution for the above applications, because those complex applications normally require many memory intensive subsystems (video compression algorithms, video interface routines, real-time audio compression, image storage, and audio storage). The beauty of this design is that if 112 K bytes of flash memory are still not sufficient for a given application, more flash memory can be added as needed, and this modification will not cost much and will not affect system speed. The instrument which we are currently developing has an embedded video application which requires very large memory space for image data storage and extensive system RAM for the graphic data compression processing. The ***Flash Window*** architecture is the only design which can handle these tasks and it still use a 16-bit microprocessor. Additionally, as we mentioned in the beginning of this chapter, the ***Flash Window*** can continue to handle interrupts occurring at almost any time. To my knowledge, the is the first embedded system design that can be integrated into an instrument (product) and allows for the remote update of system code without the risk of failure.

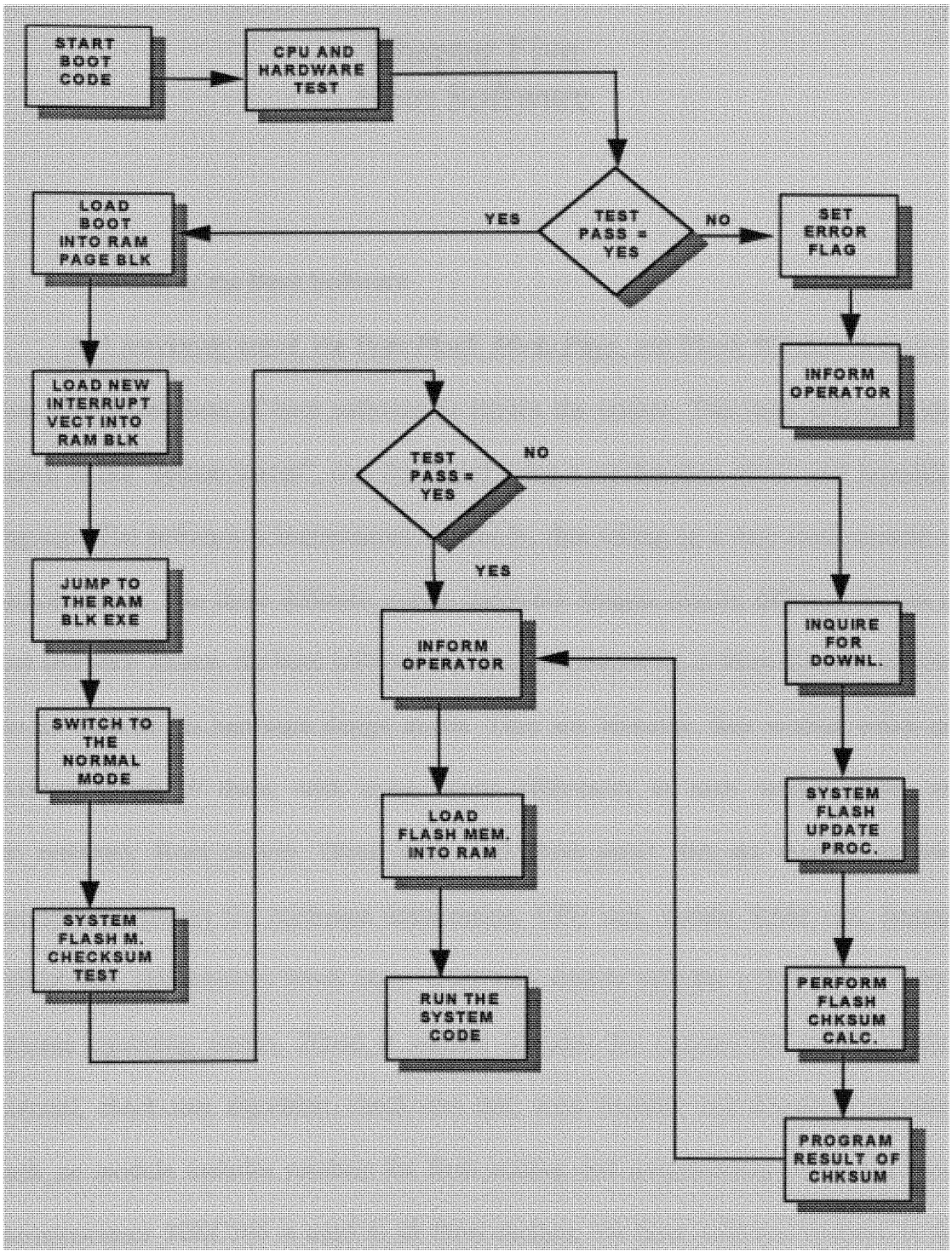

Figure 4.3 System Boot Code Operation
for the ***Flash Window*** design

Chapter Five
Boot Software

5.1 Overview of the Boot Software

As I previously noted, the ***Data Block***, ***Mode Page***, and ***Flash Window*** designs all must have some form of boot software to initiate the system. This boot software, also known as the "system kernel", was developed using a combination of 'C' and assembler language. The software is implemented compactly, thus taking up a very small amount of memory space. The three different system hardware designs require different flavors of boot software. Hence, the boot software that I developed comes in three distinct versions, one for each of the hardware design model. Thus are, however, four module which are common to all of them, with some minor differences in each version. They are Init_sys(asm), Boot_ram(c), Dload_st(c), and Flash_pe(c). Init_sys module includes assembler routine(s) for hardware initialization, CPU test, system RAM test, internal register test, and memory block copy. Boot_ram module includes a function to switch between operating modes. Dload_st module includes the routines for communication protocol, check-sum algorithm, check-sum test, error handler, and download function. Flash_pe module includes routines for flash erasing and programming. Since the same microprocessor 80KC196 is used in all three designs, the boot software routines in the boot software for the CPU test and hardware initialization are basically the same. The pseudo code for the Init_sys module will be briefly illustrated. The same Flash_pe module has been used in all three designs. However, a totally different memory address definition

listing for the flash program is required in each design, because the hardware memory mapping is dissimilar in each memory layout.

5.2 System Initialization

This section will examine how each boot software works. During system power-up, execution always starts in the Init_sys module. Init_sys begins by disabling the system interrupts. It then clears the CPU internal registers, sets the CPU I/O port with hardware design needs, and initializes all communication channels. Before trying to execute any code, it must verify that the microprocessor is functioning properly. To determine that the CPU is fully operational, the boot software must also perform an I/O test, a timer (clock) test, and a register test. If the CPU passes all of the above tests, then the processor is ready to perform the external hardware test.

The external hardware test is not a standard operation for the boot software. It depends on the design of the system. For example, the boot software for ***Mode Page*** and ***Flash Window*** systems will include two different system RAM tests. Changing of the mode is not required for testing the complete system RAM in the ***Mode Page*** approach. However, during the RAM test in the ***Flash Window*** approach, the memory mapping mode must be changed in order to access all RAM page blocks. The test is also dependent upon the layout of the external I/O map. If the system uses the serial port for communication, then the boot software has to test serial port(s) in order to determine their condition. The boot software routine for this serial port test must know the speed of the clock in the system. If any of the above tests or fail initializations, then the system is

rendered unoperational. The boot software sets an error flag and stays in an idling loop until the error condition has been removed. After completing the system hardware testing and initialization, the boot software sets up the stack pointer, and then moves on to verify the system code. (See Appendix A module name Init_sys for a complete code listing).

Because the system code resides in flash memory, When the system is to perform a download, it requires the flash erase and program operations. As part of the system code, the routines that handle these operations reside in flash memory. Because flash memory is not available during a program or erase operation, the boot software must reallocate these module to RAM (C000h or higher).

```
_INIT_A:
        DI                          ;GUARANTEE INTERRUPTS DISABLED
        CLRB    IOC2                ;Embedded Controller
CLEAR_HSI:                          ;INSURE HSI FIFO IS EMPTY
        LD      AX,HSI_TIME
;       INITIALIZE EXTERNAL PORTS
        STB     R0,PORT_A
;       INITIALIZE SERIAL PORT
SERIAL_INIT:
        LDB     IOC1,#_SEL_TXD  ;ENABLE SERIAL OUT PIN
;       INITIALIZE SERIAL DUARTS
DUART_INIT:
        LDB     AL,#0F0H            ;Guarantee not in
; ************** PERFORM HARD SELF TESTS **********************
CPU_TEST:
;       THE FOLLOWING ROUTINE TESTS TIMER1, TIMER2, AND THE SOFTWARE
;       TIMERS VIA THE HSO CAM.  ANY LOCKUP IS A RESULT OF EITHER A
;       TIMER OR CAM FAILURE.
        CLR     TIMER1              ;CONDITION TIMER1
HSO_TEST0:
        JBS     IOS0,_CAM_FULL_POS,HSO_TEST0 ;CHECK CAM EMPTY STATUS
                                             ;A LOCKUP HERE IS A BAD CAM
TIMER0_TEST:
        JBC     IOS1,_SWT0_EX_POS,TIMER0_TEST
;       CHECK THE INTERNAL REGISTERS
REG_TEST:
        CLR     BP                  ;CHECK THE LOWEST
;       CPU GOOD, SO TURN OFF CPU LED

CPU_OK:
        LDB     LEDS,R0
        ORB     LEDS,#NOT(CPU_LED) ;OK, TURN OFF CPU LED
        STB     LEDS,IO_PORT1

;       SET-UP AN INTERNAL STACK

        LD      SP,#INTERNAL_STACK;
;       NOW CHECK THE RAM

RAM_TEST:
        CALL    _RAM_TEST           ;& THE EXTERNAL RAM
RAM_OK:         ANDB    LEDS,#NOT(RAM_LED)  ;RAM OK, TURN OFF LED
        STB     LEDS,IO_PORT1
;       RAM IS GOOD SO, SET UP STACK
        LD      SP,#SYSTEM_STACK
;       TEST THE SERIAL DUARTS
DUART1A_TEST:
        CALL    DUART_TEST          ;MAKE CALL
;       NOW CHECK INTER-PROCESSOR COMMUNICATION
SERIAL_TEST:
        JBC     SP_STAT,_SP_TXE_POS,SERIAL_TEST
SERIAL_OK:
        ANDB    LEDS,#NOT(FLASH_LED)  ;FLASH OK, TURN OFF LED
        STB     LEDS,IO_PORT1
PASSED_TESTS:
;       MOVE DOWNLOADER CODE TO HIGH MEMORY RAM
MOVE_CODE:
        LD      CX,#DLOAD_START
        LD      DX,#RAM_BASE
        BMOV    CXL,BX
        LD      BP,#RAM_BASE
        BR      [BP]                ;JUMP TO DOWNLOADER
;       IF HARD ERROR LOOP HERE FOREVER
LOCKUP:         SJMP    LOCKUP      ;DROP DEAD!
```

Listing 1. Initialization functions of the Boot Code

In the above pseudo code, listing a branch instruction was used to transfer the execution from module Init_sys to the module boot (starting location is at RAM_BASE). From this point on, the system boot code execution begins to operate in the RAM page block. The first item sought by the boot software is the flash memory check-sum routine. If the result of the check-sum matches the previously stored value, this indicates that the contents of flash memory remain exactly the same as immediately following the previous download. The boot software will then load the system application code from flash memory into the system RAM, and send a message to the operator that the system is ready.

5.3 Flash Validation

If the result of the flash memory check-sum routine does not match, then the boot software will continue to execute until the flash memory has been updated. The following pseudo code for the check-sum test routine shows how the check-sum testing algorithm works. This example is from the ***Mode Page***'s boot software. The software performs the check-sum twice, once for each of the two different flash memory blocks. It reads the check-sum control value from the SYSTEM_ACRC_ADDRESS which is a special location set aside to store the result of the check-sum. It then applies the check-sum algorithm to determine whether or not the contents of the flash memory block are valid.

```
byte _Is_CRC_ok(void)
{
        byte result=0;          /* This is a flag for the test result, set = fail */
        word tempcrc=0;         /* A 'word' is a Check-Sum from last download */

        tempcrc = _Read_chksum_at_location(SYSTEM_ACRC_ADDRESS);

        if(_CalculateCRC(SYSTEM_MAIN_ADDRESS,
                SYSTEM_AEND_ADDRESS)==tempcrc)
                result=1;       /* It is passed in the block, set = pass */

        tempcrc = _Read_chksum_at_location(SYSTEM_TCRC_ADDRESS);

        if(_CalculateCRC(SYSTEM_0200_ADDRESS,
                        SYSTEM_MAIN_ADDRESS)!=tempcrc)
                result=0;       /* It is fail in the block, set =fail */
        /* else result remaining pass */

        return result;
}
```

Listing 2. The Function of Check-Sum Test

If the check-sum does not pass (result=0), then the boot software prepares for a system code update and notifies the operator that the application code must be download to the system.

5.4 Flash Memory Erasing and Programming

The system code download is divided into three steps: flash block erase, flash program, and flash block check-sum calculation. Before erasing the flash block, the Vpp needs to be set high (28F001BX need Vpp=+12V for erasing and programming). The details of the erase command sequence are shown in Figure 5.1. After the flash block is successfully erased, the flash memory is ready to be programmed, and the system application code downloads through the serial communication port. The details of this operation are shown in Figure 5.2. When the code download is complete, the boot

software performs the check-sum calculation and stores the result in a reserved location (SYSTEM_ACRC_ADDRESS) before it sets the Vpp to low [2].

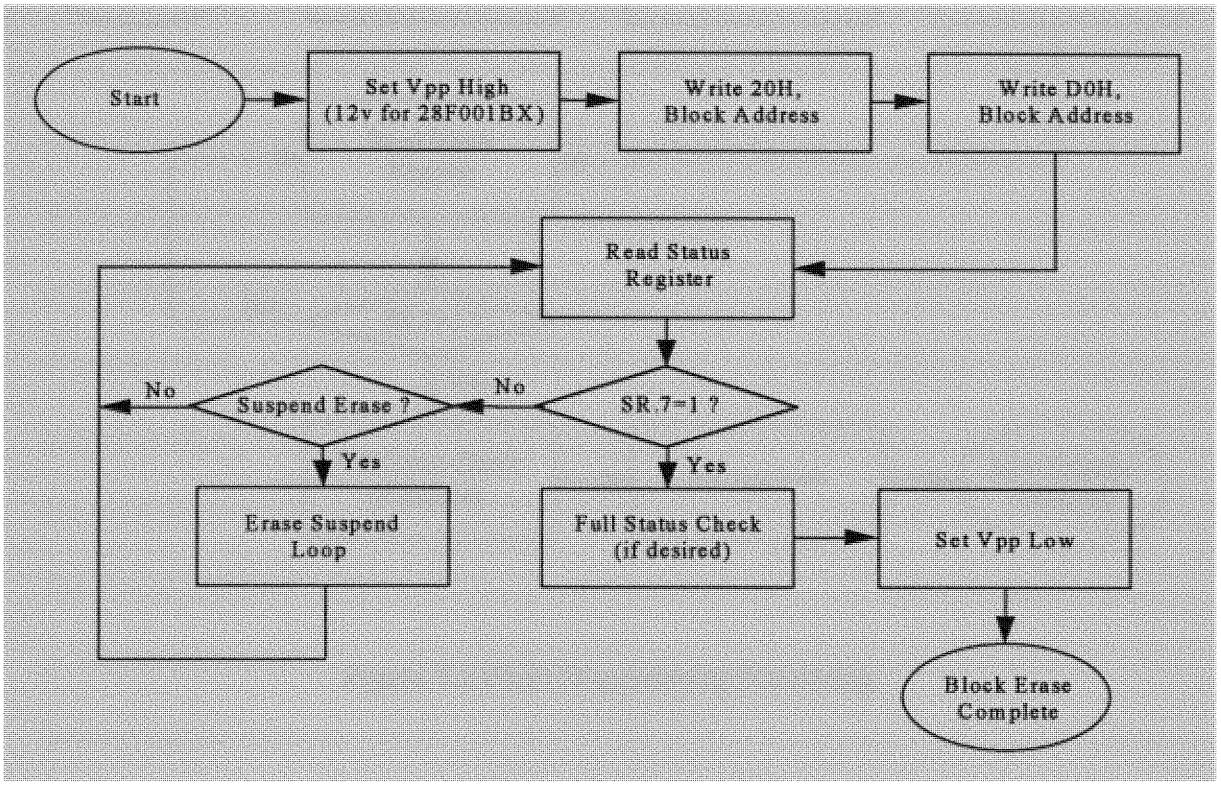

Figure 5.1 Flash Erasure Processing

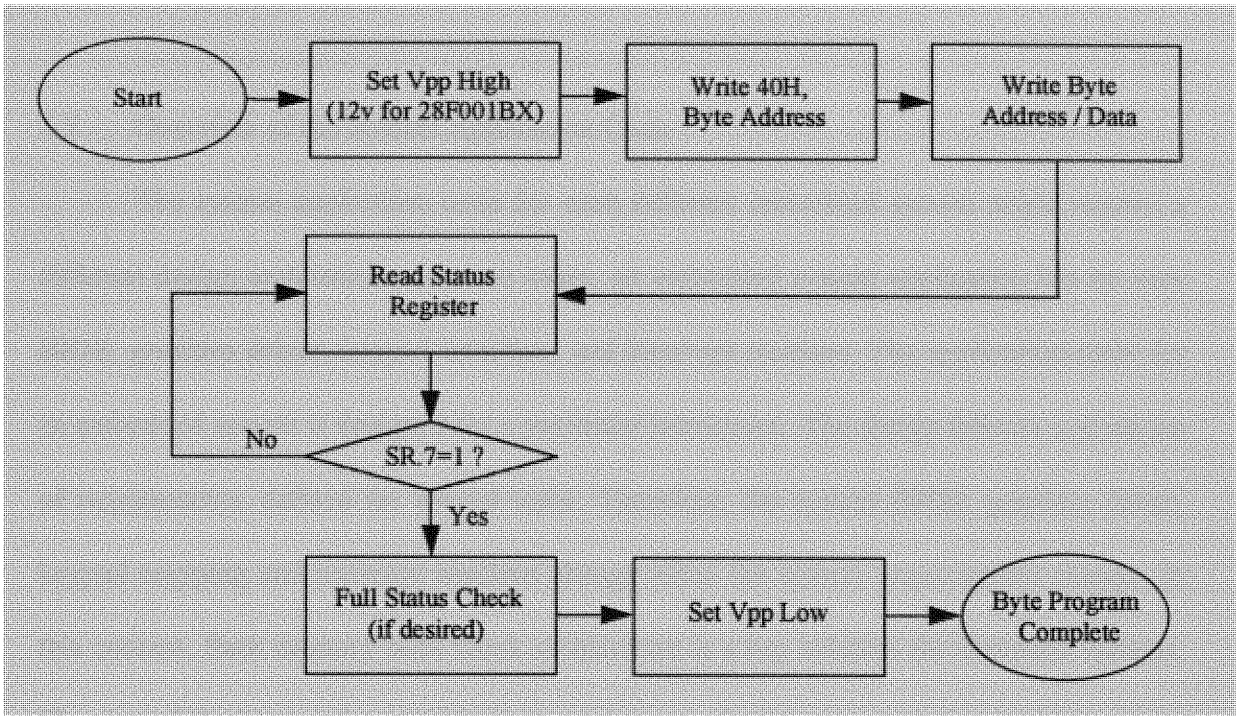

Figure 5.2 Flash Programming Processing

Figures 5.1 and 5.2 are flowcharts of programming and erasure processing for the 28F001BX boot block flash [1]. A different flash component might require a different command and processing sequence. After an initial device power-up, or after a return from deep power-down, the 28F001BX functions as read-only memory.

Writing (or programming) flash is the process of changing "1"s to "0"s. Erasing flash is the process of changing "0"s to "1"s. Flash memory is erased on a block-by-block basis. Block are defined by a fixed address range, and each different flash device has a

different define setting. When a block is erased, all address location within a block are erased in parallel, independent of other blocks in the flash memory device.[7]

5.5 Overview of the System Application Code

The system application code for each different embedded system might require a different design architecture. In most cases, an embedded system application has its own BIOS, graphics driver, keypad interface, and interrupt routines. The application code for an embedded system is designed for a particular instrument. The design follows a given set of requirements, defined for special tasks. Therefore, the system BIOS, graphics driver, keypad mapping and interrupt routines change when the system requirements change. In this context, flash memory boot block architecture is a key solution for firmware updates. The size of the boot software is kept to a minimum. It does not contain a graphics driver or keypad interface routine, only a very small portion of the communication routine, and a basic interrupt function which can start the system and prevent 'lockup'. This is the advantage of the flash memory boot block architecture with respect to firmware updates. It not only updates the basic system code, but also completely updates the system BIOS, graphics driver, keypad interface routine, and interrupt service functions.

In order to prevent system 'lockup' during the flash memory update, the RAM block must always be accessible for code execution. The system must be able to process interrupts at any time. A brief explanation of the interrupt structure of the 80C196KC will help to understand the above concerns. Twenty-eight sources of interrupts are available on this processor. These sources are gathered into 15 vectors, plus special vectors for the

user NMI (Non-Maskable Interrupt), the TRAP instruction, and Unimplemented Opcodes (Figure 5.3). These three special interrupts are available for the customer's use. The NMI, the external Non-Maskable Interrupt, is the highest priority interrupt. It accessed indirectly through a pointer in location 203Eh. The disabling of the interrupt instruction will not stop an NMI form occurring. The programmer must initialize the interrupt vector table with the starting addresses of the appropriate interrupt service routines. It is suggested that any unused interrupts be directed to an error handling routine. When some error condition, it is best to transfer flow control to routine lock into a continuous loop, write to an internal error log, and send the error message to the operator. More sophisticated routines might be appropriate for some production code recoveries.

The processing of all interrupts (except the NMI, TRAP and unimplemented opcode interrupts) can be disabled by clearing the I bit in the PSW (Program Status Word). Setting the I bit will enable interrupts that have mask register bits which are set. The I bit is controlled by the EI (Enable Interrupts) and DI (Disable Interrupts) instructions.

In the previous three models, we used the NMI for servicing the system's input AC power supply signal. When a power glitch occurs, it generates an interrupt to the NMI location, which was designed for safety purposes. The system's operation status will be saved when the supply power voltage drops, or the power is removed. Every time the system powers-up, the last saved operation status is checked. Then, the system may either resume or clean up the last operation processing as appropriate. This is just an example of

how NMI has been used. Of course, the NMI can be used to serve other applications. The nature of the NMI is such that, regardless of the operating condition (enable or disable), the NMI will generate an interrupt signal when it required. An extra precaution must be taken when the system is designed for remote firmware updates. If an interrupt vector and an interrupt service routine are not available while the system processes flash programming or erasure operations, then the occurrence of an NMI could lead to system 'lockup'. At this point, the only way the system can be restored is through direct human intervention (i.e. a physical system reset), which in the case of a remote update, is probably not possible. For sake of reliability, the system should have the capability to perform safe firmware updates. The remainder of this chapter will describe how the ***Flash Window*** design has achieved this goal.

As previously noted, upon power-on the system always will initially operate in Boot Mode. The boot software is fitted with a minimal set of essential interrupt service functions and their associated vectors. Some of these service functions may include a single return instruction. The boot software is designed to communicate with the download host processor only (which in our case can be a remote host). The download function of the boot software only accepts a few interrupts for servicing, and most other interrupts simply provide a return instruction. Thereby, the boot software does not necessarily need to have a BIOS or the communication interface functions for all channels. The downloaded system code always includes its own BIOS, graphics drivers, communication port drivers, and keypad (or keyboard) interface according to its needs.

Each time the processing system code performs an update, it updates not only the basic system application code, but also the system firmware.

5.6 System Code Download

The boot software copies the internal interface, programming, and erase modules (Dload_st, Flash_pe, and Boot_ram) into a higher area of RAM ('RAM Page 4 Block') before the system can perform update processing. The first instruction of the RAM code switches the memory mapping from Boot Mode to Normal Mode. The 'RAM Page 1 Block' is mapped starting at address 2000h. Then, the boot software (which resides in RAM) inserts new interrupt vectors into the 'RAM Page 1 Block'. These vectors have a common memory offset. Thus, all the interrupt vectors in the system memory now have a correct offset pointing to interrupt service functions in the 'RAM Page 4 Block' (in Normal Mode), and system execution can continue totally in the RAM base. After loading the modules into the 'RAM Page 4 Block', and loading the new vectors into the 'RAM Page 1 Block', the boot code performs the check-sum test. Regardless of the state of flash, the system is always ready to accept and respond to any interrupt. This guarantees that updates to the system firmware can be performed safely without the risk of system failure during the update process.

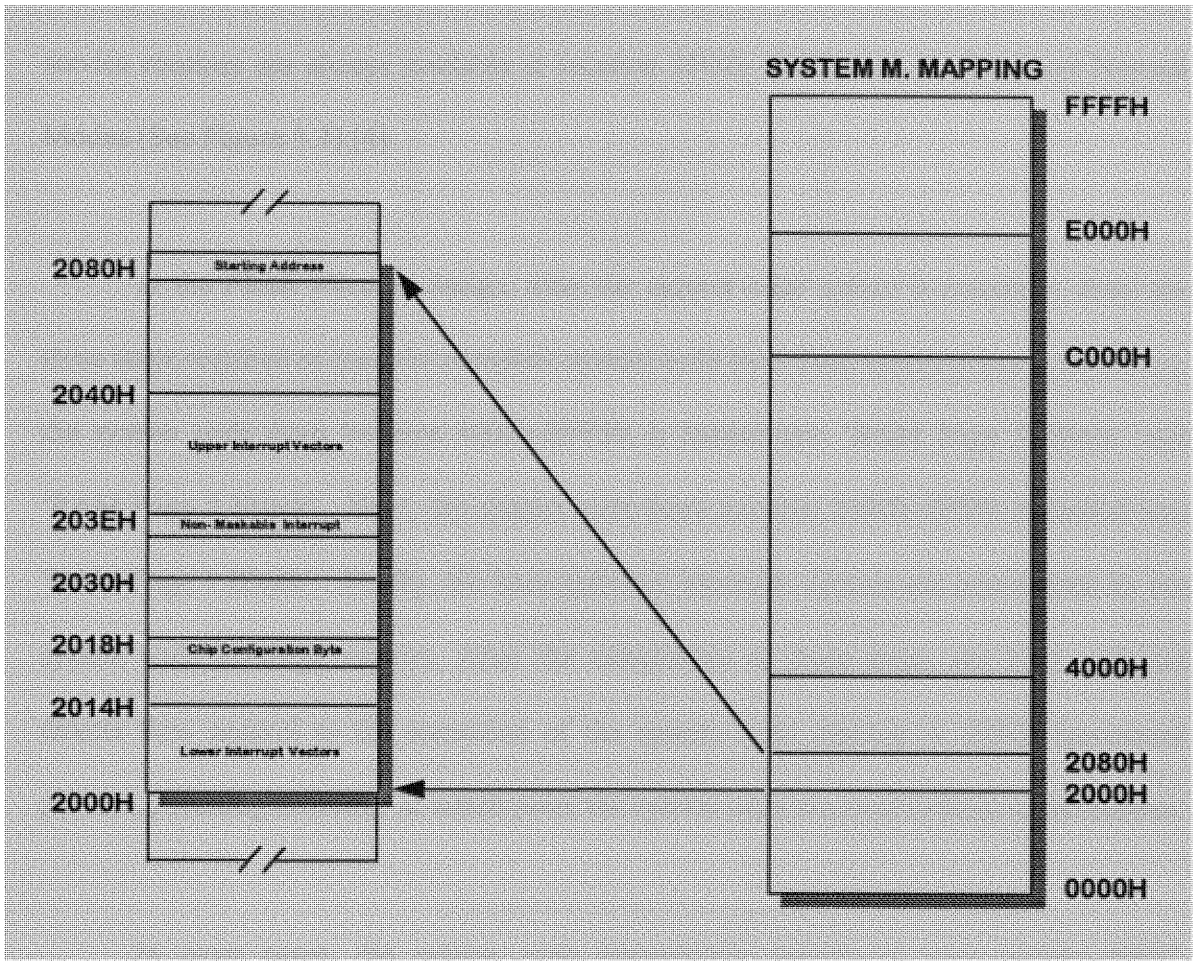

Figure 5.3 80C196 Processor Interrupt Vector Structure

There is a major difference between the initialization of the ***Flash Window*** and that of the other two designs. Before the mode is switched to Normal Mode in the ***Flash Window***, the boot software duplicates all interrupt vectors and service functions in the 'RAM Page 1 Block' and 'RAM Page 4 Block'. (Figures 3.2 and 3.3) In both Normal and Extended Modes of the ***Mode Page*** approach, the location 2000h-3FFFh is mapped to flash memory. Therefore, during the code update processing it is not accessible. So, incoming interrupts can not be served, causing the system to fail if they arise. This type of

failure, however, does not occur in the ***Flash Window*** architecture. Interrupt vectors and service routines are always available at location 2000h-2040h, in both Boot and Normal Modes (See Figure 5.3) [4].

In the ***Flash Window*** design, after the system code update is complete, there are three steps required to load the complete system code from flash memory into RAM. First, the system code from location 4000h-BFFFh is loaded into 'RAM Page 2 Block' and 'RAM Page 3 Block'. Next, the system code from location 2080h-3FFFh is loaded into 'RAM Page 1 Block'. Finally, the boot software interrupt vectors are overwritten by the system code interrupt vectors. After the completion of these steps, the boot software directs flow control to the execution of the system code starting at 2080h (processor power-up starting address), at which point the system code resumes control of the system.

In most cases, the system code immediately installs its own system BIOS, communication port driver, graphic driver (if any), keypad (or keyboard) interface configuration (if any), and begins execution. The boot software is invisible to the system until the next power-up (or system reset, or download request from the host operator) (Figure 5.4) [2].

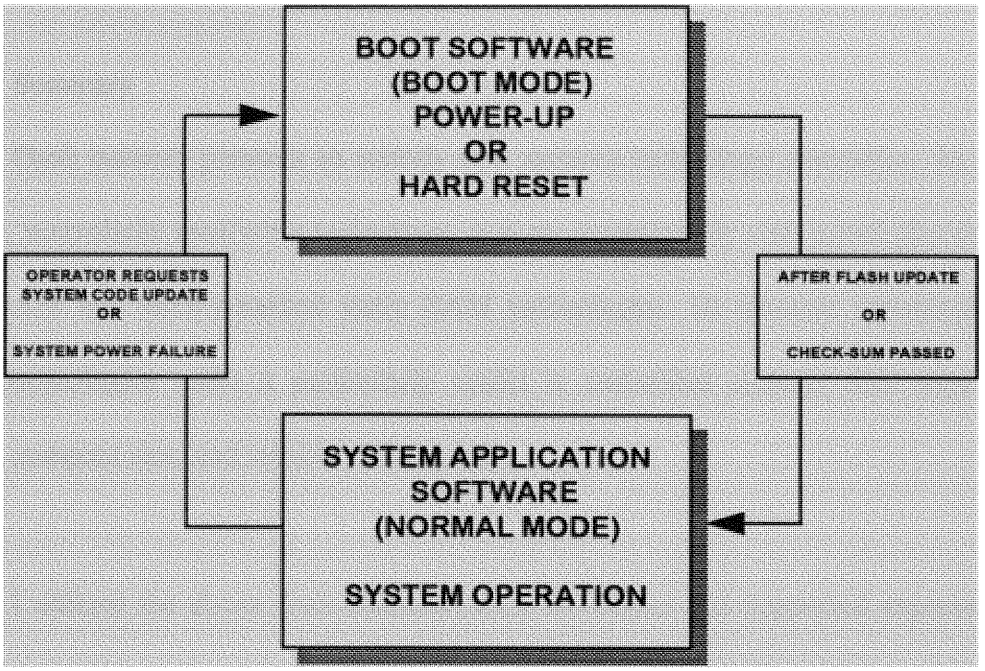

Figure 5.4 Code Execution

The above description of the ***Flash Window*** system code update processing clearly demonstrates the greater reliability of this approach over the ***Data Block*** and ***Mode Page*** designs. In addition, the ***Flash Window*** architecture provides more code memory space, and the system has access to all available flash memory (128K) in both modes. This design offers convenient system code development, because it runs in a RAM-based environment, and requires no mode change. It also provides the capability to debug the boot software with an In-Circuit-Emulator (ICE).

Chapter Six
Summary & Conclusions

6.1 Summary

Flash memory presents an entirely new memory technology alternative. As a high-density, nonvolatile read/write technology, it is an exceptionally well-suited alternative to the DRAM and battery-backed static RAM technologies. Its inherent advantages over these technologies make it particularly useful in embedded systems that require low power, compact size, and reliable code update.

I utilized the above flash characteristics, developed and supported experimentally three (physical model) implementations of embedded systems which had the capability of remote system firmware update. As previously mentioned, the three models used are ***Data Block***, ***Mode Page***, and ***Flash Window***. Each model has its own characteristics, and every design could be used in a different embedded system accordingly.

In Chapter Two, I presented the ***Data Block*** architecture. This design provides a simple memory mapping with limited system memory space. In this design, I used only one memory map. Therefore there was no need to involve mode switching in the system memory layout. The design simplified the development of system application code and boot kernel code. The design is convenient for debugging both the system and boot code with an (In-Circuit-Emulator) ICE. During the system code update processing, the design approach disables all Maskable interrupts and yields satisfactory results provided no non-

maskable interrupt are under this assumption the system software update process will be successful. Therefore, the design is good for simple systems not requiring remote update, such as, PC BIOS systems.

In Chapter Three, I described the ***Mode Page*** design. This system layout has more code memory storage space when compared with the ***Data Block***. It involved three mode pages which are Boot, Normal, and Extended. In this design, the CPU can access more flash memory storage space than in the previous design. Since, we deal with more than one mode in ***Mode Page*** integration of system software and boot code development is complicated. The Mode Page provides the capability of placing dual system application software residing in two independent memory 'pages'. This can reduce the development cost for some systems. However, the design has some shortcomings when handling interrupts in the system code update processing. Overall, the Mode Page design brought us one step closer to the final design.

In Chapter Four, I introduced the ***Flash Window*** architecture. This software/ hardware design enables the system to recover even if the system code update process has been interrupted. The CPU can access the entire flash memory (128K bytes) in both the Boot and Normal modes. There is a major difference between ***Flash window*** and the other pervious designs. A RAM block ('RAM Page 1 Block') is inserted at the starting address 2000h-3FFFh in the Normal Mode, this guarantees that all interrupts can be accepted regardless of the state of the system process. The system code always executes from a RAM based environment. This environment is convenient for our complex system

code which requires a large amount of RAM and code storage space. With this design, we achieved the proposed goal.

In Chapter Five, I discussed the Boot software which is divided into four modules. Init_sys is a module for system hardware initialization and testing. Boot_ram is the module which runs completely from the RAM base. Dload_st is a module for handling the communication, command processor, check-sum, mode control, and error handler. Flash_pe is a module for flash memory code update operation. The Init_sys is written in assembly language, and the other three modules were originally written in 'C' language.

6.2 Conclusion

The most critical timing operation is the copying of the system application code interrupt vector on top of the boot software interrupt vector in the ***Flash Window*** design. If an interrupt occurs at a time that the actual interrupt vector is in the process of being modified then an improper interrupt vector may point to an unexpected location. This may cause the system to lock-up or hang. Of course, the chances are very small for this situation to arise.

In the following table I illustrate the relationship between 'vulnerable time' and speed of the system clock. 'Vulnerable time' is defined as the time interval in which the system is updating interrupt vectors, and system clock is defined as the system processor input clock.

SPEED OF CLOCK	6 MHz	12 MHz	16 MHz
Write Cycle	2-3.4 μsec	1-1.7 μsec	.75-1.25 μsec
Modify a Vector	4-6.7 μsec	2-3.4 μsec	1.5-2.5 μsec
Vulnerable Time for Updating	6.7μsec	3.4 μsec	2.5μsec

Table 6.1 Relationship between Vulnerable Time and System Clock

After completing the implementation of the ***Flash Window*** design, the effective system reliability was tested. The reliability tests for overall system architecture and structural design were performed by a professional test engineering group. The testing procedure was performed during the system code updating process. The Test Engineering Group tried to apply non-maskable interrupts at pre-defined times, during the update process. Initially, the test engineers used a 16MHz (processor) clock in the system in a attempt to identify the effective 'vulnerable time'. At this clock rate the occurrence of the system lockup was virtually non-existent. Even at a lower clock rate of 6MHz the system could only be forced into lockup if a warning signal preceding the vulnerable period was provided to the testing system to start issuing non-maskable interrupts. In conclusion, although there is a small time interval in which the system can be made to lockup, the overall system update process, at reasonable microprocessor speed, has been proven to be reliable for real-life scenarios.

Today's system designer, software developer, and manufacturer are faced with a number of options when choosing a flash memory solution for updateable firmware. Design and implementation system flash memory architecture must meet the specific requirements of different applications. I hope that these three designs could serve as models or guidelines.

6.3 Future Work

A number of suggestions for further study are presented here. In the future, a complex system might require much more memory for code storage. The ***Flash Window*** design may be enhanced in the future to achieve a larger 'window' area or increase number of 'window' memory address control lines. In other words, this can be accomplished by using 1) an 8K size 'window' instead of 4K , 2) six or seven address control lines instead of five control lines to access the flash memory block in the 'window'. The result of these changes will double or quadruple the size of the CPU accessible flash memory. This is one of the efficient ways to increase the system memory space without upgrading the system processor.

APPENDIX A

GLOSSARY

BBS	- Bulletin Board System
DRAM	- Dynamic Random Access Memory
EEPROM	- Electrically Erasable Programmable Read-Only Memory
EMS	- Expanded Memory System
EPROM	- Electrically Programmable Read-Only Memory
ETOX	- EPROM Tunnel Oxide
FFS	- Flash File System
FITS	- Failures In Time(s)
GPIO	- General Purpose Input-Output
ICE	- In-Circuit-Emulator
MOS	- Metal Oxide Semiconductor
OEM	- Original Equipment Manufacturer
PSOP	- Plastic SOP
ROM	- Read Only Memory
RP#	- Reset/Powerdown (input)
SOP	- Small Outline Package
SRAM	- Static Random Access Memory
TSOP	- Thin Small Outline Package
XIP	- eXecute-In-Place
Vcc	- Power source for the flash device during all operational modes
Vpp	- The program/erase voltage
WE#	- Write Enable (input)
WSM	- Write Status Machine

APPENDIX B

THE RELIABILITY OF FLASH MEMORY

Data retention and lifetime reliability statistics for ETOX III flash are equivalent to those of EPROM. Representing the third generation of flash memory technology, the ETOX III 0.8μ process provides < 100 FITS (failures in time) @ 55°C in a specification that delivers 100,000 write cycles per block. This capability significantly exceeds the cycling requirements of even the most demanding applications. Code storage for embedded control programs used in standard computer applications is infrequently updated. Twenty-year system lifetimes may require fewer than 100 rewrites. Even routinely-changed data tables (used in navigational computers and black box controllers) only require about 1,000 write cycles over a 20-year period. The most demanding flash memory application of all, archival data storage in PC applications, typically requires about 5,000 write cycles in 20 years [7].

APPENDIX C

INITIALIZATION ROUTINE

```
;=============================================================================
;
;  FILE NAME: INIT_SYS.ASM
;
; DESCRIPTION: THIS FILE CONTAINS ROUTINES FOR INITIALIZING
;                    80c196 MICROCONTROLLER.  IT PERFORMS THE HARD &
;                    SOFT ERROR TESTS, SETS UP THE SERIAL PORT, &
;                    RECONFIGURES THE FLASH/RAM MEMORY MAP.  AFTER
;                    COMPLETING ITS TASKS, IT JUMPS INTO 'FRONTEND'.
;
;=============================================================================*/

INIT_A    MODULE MAIN
$NOLIST
$INCLUDE(HDR\SYSTEM.INC)
$INCLUDE(HDR\INIT96.CON)
$LIST

PUBLIC    _INIT_A
; EXTERNAL ASM96 FUNCTIONS
EXTRN VECTOR_JUMP, BRANCH
; EXTERNAL C96 FUNCTIONS
; THE FOLLOWING FLAG IS USED TO CREATE A VERSION
; FOR THE B PROCESSOR WHICH CAN BE USED WITH THE
; ICE196 WHILE DEBUGGING CODE. THIS FLAG MUST BE
; SET TO FALSE TO COMPILE THE ACTUAL PRODUCT CODE.

ICE196    EQU       TRUE
;         MEMORY VALUES
FLASH_BASE        EQU       2000H               ;START ADDRESS OF FLASH
FLASH_START       EQU       2100H:ENTRY         ;BASE ADDRESS OF FLASH
DLOAD_START       EQU       2500H               ;START OF DOWNLOAD BLOCK
; // on 23/6/94 DLOAD_START EQU       2800H
; // on 09/9/94 DLOAD_START EQU       2600H
;START OF DOWNLOAD BLOCK
DLOAD_TOP         EQU       03FFDH              ;TOP OF DOWNLOAD BLOCK
DLOAD_SIZE        EQU       DLOAD_TOP-DLOAD_START ;BLOCK SIZE
RT_CLOCK          EQU       0FEC0H              ;REAL TIME CLOCK ADDRESS
RAM_BASE          EQU       0C000H              ;BASE ADDRESS OF RAM
RAM_TOP                     EQU       RT_CLOCK-1          ;TOP OF RAM AREA
RAM_SIZE          EQU       RAM_TOP-RAM_BASE ;SIZE OF RAM AREA
RAM_WORDS         EQU       RAM_SIZE /2         ;RAM AREA IN WORDS
POWERUP                     EQU       2080H               ;POWER UP ENTRY POINT
INTERNAL_STACK EQU          100H-2              ;INITIAL INTERNAL STACK POINTER
SYSTEM_STACK      EQU       RT_CLOCK-2          ;NORMAL STACK POINTER
;         SERIAL I/O MODE:
;           MODE1, NO PARITY, 8 BITS, 1 STOP BIT
;           BAUDRATE = 38.4K
_SP_COM_MODE      EQU       _SP_MODE1+_SP_REN
BAUDRATE          EQU       BAUD96
;         CPU CONFIGURATION BYTE
CCB               EQU       NOPROTECT+WAIT1+ALE+WR ;CHIP CONFIG. BYTE
;         MACHINE INSTRUCTIONS
JUMP              EQU       0E7H      ;'LJMP' - LONG JUMP INSTRUCTION
A_NOP             EQU       0FDH      ;'NOP'  - NOP INSTRUCTION
A_NOP_JUMP        EQU       256*JUMP+A_NOP ;'NOP JUMP' AS A WORD
AD_OFFSET         EQU       32H                 ;5% A/D PRECISION OFFSET
; ******************** RESERVED REGISTERS **********************
RSEG      AT        40H
```

```
LEDS:   DSB     1           ;LED REGISTER
; ********** BEGIN GLOBAL DATA AREA **********
; SYSTEM GLOBAL ADDRESSES AFTER RE-MAPPING
; INTO RAM BETWEEN C000H - FEBFH

DSEG    AT      RAM_BASE

; WARNING!!! THE FOLLOWING 'EQUATE' MUST RESIDE AT THE TOP OF
;           DATA DEFINITION AREA SINCE IT IS USED TO EVALUTE
;           EXPRESSIONS DURING PROGRAM COMPILATION.

GLOBAL_BEGIN    EQU     $       ;BEGINNING OF GLOBAL AREA

; WARNING!!! THE FOLLOWING 'EQUATE' MUST RESIDE AT THE END OF
;           DATA DEFINITION AREA SINCE IT IS USED TO EVALUTE
;           EXPRESSIONS DURING PROGRAM COMPILATION.

GLOBAL_END      EQU     $       ;END OF GLOBAL AREA

; ********** BEGIN PROGRAM CODE AREA **********

CSEG    AT      _CCR
        ORG     _CCR    ;INITIALIZE CHIP CONFIG. REGISTER
        DCB     CCB
        ORG     201AH
JUMPSELF:
        SJMP    JUMPSELF ;RESERVED FOR INTEL TOOLS

;       **** HARDWARE RESET BEGINS HERE ****
;       ****  AND JUMPS TO FLASH_BASE  ****

        ORG     POWERUP

        DI                          ;DISABLE ALL INTERRUPTS
                                    ;UNTIL HARDWARE IS INITIALIZED
        LJMP    FLASH_START

CSEG    AT      FLASH_START
        ORG     FLASH_START

; HARDWARE INITIALIZATION BEGINS HERE
;*****************************************************************
; _INIT_A - INITIALIZE THE c196 A PROCESSOR HARDWARE
;
; C96 ENTRY:        via Hardware Power-up Reset or _init_a();
;
; RETURNS:          NOTHING
;
; ASM96 ENTRY:      NA
;
; ATTRIBUTE:        REENTRANT
;
; DESCRIPTION:      THIS ROUTINE INITIALIZES THE 87c196 MICROCONTROLLER
;                   FOR THE A PROCESSOR ON CPU CIRCUIT BOARD
;                   IT ALSO PERFORMS THE HARD/SOFT ERROR TESTS, SETS UP
;                   THE SERIAL PORTS, AND RECONFIGURES THE FLASH/RAM MAP.
;
;                   CAUTION!!!! THIS ROUTINE BEGINS WITHOUT A STACK
;                   POINTER.  UNTIL THE INTERNAL REGISTERS ARE TESTED
;                   NO CALLS, PUSHES, OR POPS MAY BE PERFORMED.  ONCE
;                   THE REGISTER TEST IS PASSED, THE STACK POINTER
;                   WILL BE TEMPORARILY SET UP IN THE REGISTERS SO
;                   THAT THE RAM_TEST ROUTINE MAY BE CALLED.  AFTER
;                   PASSING THE RAM TEST, THE STACK WILL BE SET UP
;                   AT THE TOP OF RAM.
;
;                   NOTE: THE FIRST INSTRUCTION 'DI' GUARANTEES THAT
;                   INTERRUPTS ARE DISABLED.  THE FIRST BYTE IN THE
;                   FLASH AT 2080 IS ALSO 'DI'.  THIS WORKS ONLY AT
```

```
;                   POWER UP OR FOLLOWING A 'RST' INSTRUCTION WHEN
;                   THE FLASH BLOCK AT 2000H IS ENABLED.  FOLLOWING
;                   INITIALIZATION THE BYTE AT 2080 IS LOADED IN RAM
;                   WITH A 'NOP' TO BE USED IN THE _CCP_ENTRY JUMP
;                   TABLE.  IF THE _INIT96 IS CALLED VIA SOFTWARE,
;                   THE 'DI' INSTRUCTION WILL DISABLE INTERRUPTS IN
;                   THE NORMAL HARDWARE MANNER JUST TO BE SAFE.
; ***************************************************************
_INIT_A:
        DI                          ;GUARANTEE INTERRUPTS DISABLED
        CLRB    WSR                 ;OPEN WINDOW 0
        CLR     INT_MASK            ;CLEAR INTERRUPT MASKS
        CLR     INT_PEND1           ;PENDING REGS.
        LDB     IOC2,#_CLEAR_CAM ;CLEAR THE CAM &
        LDB     PWM0_CONTROL,R0         ;INITIALIZE ALL OTHER REGS
        CLRB    IOC0                ;Ref: INTEL 16-Bit
        CLRB    IOC2                ;Embedded Controller
        LDB     IOC1,#20H
        LD      AX,#0FFH;Handbook p. 3-68
        LDB     HSI_MODE,AL         ;Every HSI transition
        LDB     IO_PORT1,AL         ;Port 1 LED's ON
        LDB     IO_PORT2,#0C1H
        LDB     WSR,#1              ;OPEN WINDOW 1
        LDB     IOC3,#0F2H          ;DISABLE PWM PORTS
        LDB     PWM1_CONTROL,R0
        LDB     PWM2_CONTROL,R0
        CLRB    WSR                 ;RETURN TO WINDOW 0
        LDB     CL,#8
CLEAR_HSI:                          ;INSURE HSI FIFO IS EMPTY
        LD      AX,HSI_TIME
        DJNZ    CL,CLEAR_HSI
        LDB     WSR,#15             ;OPEN WINDOW 15
        CLRB    IOS0                ;& CLEAR ALL STATUS REGS.
        CLRB    IOS1
        CLRB    IOS2
        CLRB    HSI_STATUS
        LD      AD_RESULT,#7FF0H
        CLR     TIMER1
        CLRB    WSR                 ;RETURN TO WINDOW 0
        CLR     TIMER2

;       INITIALIZE EXTERNAL PORTS
        STB     R0,PORT_A
        STB     R0,PORT_B
        STB     R0,PORT_C
        STB     R0,PORT_D
        STB     R0,PORT_E

;       INITIALIZE SERIAL PORT

SERIAL_INIT:
        LDB     IOC1,#_SEL_TXD  ;ENABLE SERIAL OUT PIN
        LD      AX,#BAUDRATE    ;SERIAL BAUD RATE
        LDB     IO_BAUD,AL      ;BASED ON
        LDB     IO_BAUD,AH      ;TIMER 1 CLK INPUT
        LDB     AL,SBUF         ;FLUSH THE INPUT BUFFER
        LDB     SP_CON,#_SP_COM_MODE ;SET COM MODE
        LDB     WSR,#15                 ;TURN ON WINDOW 15
                                        ;AND CLEAR ANY PENDING
        ANDB    INT_PEND1,#NOT(_MASK_TI) ;INTERRUPT
        LDB     SP_STAT,#(_SP_TI+_SP_TXE) ;INSURE SERIAL PORT STATUS
        CLRB    WSR

;       INITIALIZE SERIAL DUARTS

DUART_INIT:
        LDB     AL,#0F0H            ;Guarantee not in
        STB     AL,DUART1_COMMANDA  ;power down mode.
        STB     AL,DUART2_COMMANDA
```

```
        LDB     AL,#30H                 ;Guarantee Tx is
        STB     AL,DUART1_COMMANDA      ;reset
        STB     AL,DUART1_COMMANDB
        STB     AL,DUART2_COMMANDA
        STB     AL,DUART2_COMMANDB

        LDB     AL,#20H                 ;Guarantee Rx is
        STB     AL,DUART1_COMMANDA      ;reset
        STB     AL,DUART1_COMMANDB
        STB     AL,DUART2_COMMANDA
        STB     AL,DUART2_COMMANDB

        LDB     AL,#0B0H        ;Reset MR pointer
        STB     AL,DUART1_COMMANDA      ;MR --> MR0
        STB     AL,DUART1_COMMANDB
        STB     AL,DUART2_COMMANDA
        STB     AL,DUART2_COMMANDB

        CLRB    AL                      ;Disable watch dog,
        STB     AL,DUART1_MODEA                 ;RxINT, TxINT & set
        STB     AL,DUART1_MODEB                 ;Normal Baud rate
        STB     AL,DUART2_MODEA
        STB     AL,DUART2_MODEB

        LDB     AL,#93H                 ;8 bits, no parity, char
        STB     AL,DUART1_MODEA                 ;mode, RxRDY, & RTS
        STB     AL,DUART1_MODEB

        LDB     AL,#13H                 ;8 bits, no parity, &
        STB     AL,DUART2_MODEA                 ;char mode
        STB     AL,DUART2_MODEB

        LDB     AL,#17H                 ;1 stop bit, CTS,
        STB     AL,DUART1_MODEA                 ;normal mode
        STB     AL,DUART1_MODEB

        LDB     AL,#07H                 ;1 stop bit, no CTS,
        STB     AL,DUART2_MODEA                 ;normal mode
        STB     AL,DUART2_MODEB

        LDB     AL,#0FFH        ;Reset bits 0P0-OP9
        STB     AL,DUART1_CLR_BIT
        STB     AL,DUART2_CLR_BIT

        LDB     AL,#080H        ;BRG = set 2
        STB     AL,DUART1_AUX           ;EXTERNAL CLOCK
        STB     AL,DUART2_AUX           ;ACR register

        LDB     AL,#0BBH                ;RX & TX 9600 BAUD
        STB     AL,DUART1_CLOCKA
        STB     AL,DUART1_CLOCKB
        STB     AL,DUART2_CLOCKA
        STB     AL,DUART2_CLOCKB

        LDB     AL,#22H                 ;Set A & B interrupts
        STB     AL,DUART1_INT_MASK      ;on RxRDY
        STB     AL,DUART2_INT_MASK

        LDB     AL,DUART1_IN_CHG        ;Clear Input port &
        LDB     AL,DUART2_IN_CHG        ;ISR(7) bits

        LDB     AL,#45H                 ;Clear error bits
        STB     AL,DUART1_COMMANDA      ;Enable Rx & Tx
        STB     AL,DUART1_COMMANDB
        STB     AL,DUART2_COMMANDA
        STB     AL,DUART2_COMMANDB
; ********** PERFORM HARD SELF TESTS **********
CPU_TEST:
```

```
;       THE FOLLOWING ROUTINE TESTS TIMER1, TIMER2, AND THE SOFTWARE
;       TIMERS VIA THE HSO CAM.  ANY LOCKUP IS A RESULT OF EITHER A
;       TIMER OR CAM FAILURE.
        LDB     WSR,#15                 ;OPEN WINDOW 15
        CLR     TIMER1                  ;CONDITION TIMER1
        CLRB    WSR                     ;RETURN TO WINDOW 0
HSO_TEST0:
        JBS     IOS0,_CAM_FULL_POS,HSO_TEST0 ;CHECK CAM EMPTY STATUS
                                                ;A LOCKUP HERE IS A BAD CAM
        LDB     HSO_COMMAND,#_COM_SWT0
        ADD     HSO_TIME,TIMER1,#4000H      ;TIMES OUT IN 0.052293 SEC
HSO_TEST1:
        JBS     IOS0,_HSO_FULL_POS,HSO_TEST1 ;CHECK HSO BUFFER EMPTY
                                                ;A LOCKUP HERE IS A BAD CAM
        LDB     HSO_COMMAND,#_COM_SWT1
        ADD     HSO_TIME,TIMER1,#08000H ;TIMES OUT IN 0.104856 SEC
HSO_TEST2:
        JBS     IOS0,_HSO_FULL_POS,HSO_TEST2 ;CHECK HSO BUFFER EMPTY
                                                ;A LOCKUP HERE IS A BAD CAM
        CLR     TIMER2                  ;CONDITION TIMER2
        LDB     HSO_COMMAND,#_COM_SWT2
        ADD     HSO_TIME,TIMER1,#0C000H ;TIMES OUT IN 0.14222 SEC

HSO_TEST3:
        JBS     IOS0,_HSO_FULL_POS,HSO_TEST3 ;CHECK HSO BUFFER EMPTY
                                                ;A LOCKUP HERE IS A BAD CAM
        LDB     HSO_COMMAND,#_COM_SWT3
        ADD     HSO_TIME,TIMER2,#0FFFFH ;TIMES OUT IN 0.28444 SEC

;       NOW CHECK EACH SOFTWARE TIMER FOR TIME OUT IN ORDER
;       ANY LOCKUP HERE IS BAD TIMER OR CAM.
TIMER0_TEST:
        JBC     IOS1,_SWT0_EX_POS,TIMER0_TEST
TIMER1_TEST:
        JBC     IOS1,_SWT1_EX_POS,TIMER1_TEST
TIMER2_TEST:
        JBC     IOS1,_SWT2_EX_POS,TIMER2_TEST
TIMER3_TEST:
        JBC     IOS1,_SWT3_EX_POS,TIMER3_TEST

;       CHECK THE INTERNAL REGISTERS
REG_TEST:
        CLR     BP                      ;CHECK THE LOWEST
        CLR     AX                      ;PLM REGISTERS
        CMP     AX,BP
        BNE     LOCKUP                  ;EITHER ONE BAD?
        NOT     BP
        NOT     AX
        CMP     AX,BP
        BNE     LOCKUP                  ;TEST AGAIN
        LD      BP,#BX                  ;POINT TO BX REGISTER
REG_TEST_FF:
        ST      AX,[BP]                 ;WRITE FF INTO REGISTER
        CMP     AX,[BP]+
        BNE     LOCKUP                  ;BAD REGISTER?
        CMP     BP,#100H ;END OF REGISTERS?
        JNE     REG_TEST_FF             ;CHECK NEXT ONE
        LD      BP,#BX                  ;POINT TO BX REGISTER
REG_TEST_00:
        ST      R0,[BP]                 ;WRITE  INTO REGISTER
        CMP     R0,[BP]+
        BNE     LOCKUP                  ;BAD REGISTER?
        CMP     BP,#100H ;END OF REGISTERS?
        JNE     REG_TEST_00             ;CHECK NEXT ONE

;       CPU GOOD, SO TURN OFF CPU LED
CPU_OK:
        LDB     LEDS,R0
```

```
        ORB     LEDS,#NOT(CPU_LED) ;OK, TURN OFF CPU LED
        STB     LEDS,IO_PORT1

;       SET-UP AN INTERNAL STACK
        LD      SP,#INTERNAL_STACK;

;       NOW CHECK THE RAM
RAM_TEST:
        CALL    _RAM_TEST       ;& THE EXTERNAL RAM
        CMPB    AL,R0           ;IF BAD THEN LOCKUP
        BE      LOCKUP
RAM_OK: ANDB    LEDS,#NOT(RAM_LED) ;RAM OK, TURN OFF LED
        STB     LEDS,IO_PORT1

;       RAM IS GOOD SO, SET UP STACK
        LD      SP,#SYSTEM_STACK
        LDB     WSR,#15
        LDB     AH,IOC1         ;GET CURRENT PORT 2 CONFIG.
                                ;CLEAR TIMER1 &
        CLR     TIMER1          ;THE TIMER1 OVERFLOW BIT
        CLRB    WSR             ; (AMOUNG OTHERS)
        LDB     AL,IOS1         ;IN THE STATUS REGISTER

;       TEST THE SERIAL DUARTS

DUART1A_TEST:
        PUSH    #DUART1_A
        CALL    DUART_TEST      ;MAKE CALL
        ADD     SP,#2H          ;CLEAN UP STACK

DUART1B_TEST:
        PUSH    #DUART1_B
        CALL    DUART_TEST      ;MAKE CALL
        ADD     SP,#2H          ;CLEAN UP STACK

DUART2A_TEST:
        PUSH    #DUART2_A
        CALL    DUART_TEST      ;MAKE CALL
        ADD     SP,#2H          ;CLEAN UP STACK

DUART2B_TEST:
        PUSH    #DUART2_B
        CALL    DUART_TEST      ;MAKE CALL
        ADD     SP,#2H          ;CLEAN UP STACK

DUARTS_OK:
        ANDB    LEDS,#NOT(IO_LED) ;IO OK, TURN OFF LED
        STB     LEDS,IO_PORT1

IF(ICE196)

        LJMP    SERIAL_OK
ENDIF

;       NOW CHECK INTER-PROCESSOR COMMUNICATION
        LDB     CL,#5           ;5 LOOPS
        CLRB    BL              ;PREPARE CHARACTER
SERIAL_TEST:
        JBC     SP_STAT,_SP_TXE_POS,SERIAL_TEST
        LDB     AL,BL           ;GET CHAR
        STB     AL,SBUF         ;SEND IT

SERIAL_WAIT:
        JBC     SP_STAT,_SP_RI_POS,SERIAL_WAIT
        LDB     AL,SBUF         ;FETCH ECHO BACK
        CMPB    AL,BL
        JNE     LOCKUP
        CMPB    BL,#0FFH ;LAST ONE RECEIVED?
        JE      SERIAL_NEXT
```

```
        INC     BL                  ;NEXT CHAR
        SJMP    SERIAL_TEST
SERIAL_NEXT:
        DJNZ    CL,SERIAL_TEST      ;ANOTHER LOOP?

SERIAL_OK:
        ANDB    LEDS,#NOT(FLASH_LED) ;FLASH OK, TURN OFF LED
        STB     LEDS,IO_PORT1

PASSED_TESTS:

;       MOVE DOWNLOADER CODE TO HIGH MEMORY RAM

MOVE_CODE:
        LD      CX,#DLOAD_START
        LD      DX,#RAM_BASE
        LD      BX,#DLOAD_SIZE
        BMOV    CXL,BX
        LD      BP,#RAM_BASE
        BR      [BP]                ;JUMP TO DOWNLOADER

;*************************************************
;*      DROP DEAD ZONE                           *
;*************************************************
;       IF HARD ERROR LOOP HERE FOREVER
LOCKUP: SJMP    LOCKUP              ;DROP DEAD!
; ***************************************************************
; _RAM_TEST - PERFORM SEVERAL TESTS ON THE EXTERNAL RAM
;
; C96 ENTRY:        result= _ram_test();
; RETURNS:          result= TRUE if all tests pass, otherwise FALSE.
; REQUIRES:         extern boolean _ram_test(void);
;
; ASM96 ENTRY:      CALL    _RAM_TEST
;
; RETURNS:          AL= 1 (TRUE)  IF RAM TEST GOOD
;                   AL= 0 (FALSE ) IF RAM TESTS BAD
; REQUIRES:         $INCLUDE(INIT96.CON)
; GLOBALS:          AX
;
; DESCRIPTION:      THIS ROUTINE PERFORMS 6 TESTS ON THE SYSTEM MEMORY.
;                   THESE TESTS CHECK FOR ANY RAM BIT FAILURES AND
;                   CROSSTALK BETWEEN ADDRESS AND DATA BUS PINS.  UPON
;                   COMPLETION, ALL RAM IS FILLED WITH 0FFH OR AN 'RST'
;                   INSTRUCTION.  THE FINAL RESULT RETURNED TO THE
;                   CALLER IN THE AL REGISTER WILL BE 1 (TRUE) IF THE
;                   IS GOOD, OR 0 (FALSE) IF IT IS BAD.
; ***************************************************************

;       TEST RAM MEMORY & ADDRESS BUS
_RAM_TEST:
        PUSH    BP                  ;SAVE REGS.
        PUSH    CX
        PUSH    DX

;       THE FOLLOWING TESTS THE RAM 8 BIT ADDRESSING
        LD      CX,#RAM_SIZE        ;SET BYTE COUNT
        CLRB    AL
        LD      DX,#RAM_BASE
        LD      BP,DX               ;POINT TO TOP RAM BLOCK
RAM_8_BIT:
        STB     AL,[BP]             ;WRITE THE 8 BIT PATTERN
        CMPB    AL,[BP]+  ;& TEST IT
        BNE     BAD_RAM                     ;BAD RAM?
        INCB    AL                  ;CHANGE 8 BIT PATTERN
        DEC     CX
        JNE     RAM_8_BIT           ;OK, CHECK ANOTHER

;       THE FOLLOWING TESTS THE RAM 16 BIT ADDRESSING
```

```
        LD      CX,#RAM_WORDS  ;SET WORD COUNT
        CLR     AX
        LD      BP,DX          ;POINT TO TOP RAM BLOCK
RAM_16_BIT:
        ST      AX,[BP]        ;WRITE THE 16_BIT PATTERN
        CMP     AX,[BP]+ ;& TEST IT
        BNE     BAD_RAM                ;BAD RAM?
        INC     AX             ;CHANGE 16_BIT PATTERN
        DEC     CX
        JNE     RAM_16_BIT     ;OK, CHECK ANOTHER

;       THIS TESTS BIT CROSSTALK VIA TOGGLING
;       ADJACENT BITS
        LD      CX,#RAM_WORDS  ;SET WORD COUNT
        LD      AX,#5555H      ;PATTERN IS 55H
        LD      BP,DX          ;POINT TO TOP RAM BLOCK
RAM_55: ST      AX,[BP]        ;WRITE THE PATTERN
        CMP     AX,[BP]+ ;& TEST IT
        BNE     BAD_RAM                ;BAD RAM?
        DEC     CX
        JNE     RAM_55         ;OK, CHECK ANOTHER
        LD      CX,#RAM_WORDS  ;RE-SET WORD COUNT
        LD      AX,#0AAAAH     ;NEXT PATTERN IS AAAAH
        LD      BP,DX          ;RESET POINTER
RAM_AA: ST      AX,[BP]
        CMP     AX,[BP]+ ;TEST AAH PATTERN
        BNE     BAD_RAM                ;BAD RAM?
        DEC     CX
        JNE     RAM_AA

;       THIS TESTS CROSSTALK BETWEEN BYTES
;       IN ADJACENT WORDS
        LD      CX,#RAM_WORDS  ;SET WORD COUNT
        LD      AX,#00FFH      ;PATTERN IS 00FFH
        LD      BP,DX          ;POINT TO TOP RAM BLOCK
RAM_00FF:
        ST      AX,[BP]        ;WRITE THE PATTERN
        CMP     AX,[BP]+ ;& TEST IT
        BNE     BAD_RAM                ;BAD RAM?
        NOT     AX             ;REVERSE PATTERN
        DEC     CX
        JNE     RAM_00FF       ;OK, CHECK ANOTHER

;       THIS LAST TEST CHECKS ALL BITS SET
        LD      CX,#RAM_WORDS  ;RE-SET WORD COUNT
        LD      AX,#0FFFFH     ;PATTERN TO TEST IS FEFEH
        LD      BP,DX          ;RESET POINTER
RAM_FF: ST      AX,[BP]        ;WRITE THE PATTERN
        CMP     AX,[BP]+ ;& TEST IT
        BNE     BAD_RAM                ;BAD RAM?
        DEC     CX
        JNE     RAM_FF

; RAM HAS BEEN FILLED WITH FFH OR 'RST' INSTRUCTIONS
GOOD_RAM:
        LD      AX,#TRUE               ;RETURN TRUE
        SJMP    RAM_DONE

BAD_RAM:
        LD      AX,R0                  ;RETURN FALSE
RAM_DONE:
        POP     DX                     ;RESTORE REGS.
        POP     CX
        POP     BP
        RET                            ;EXIT TO CALLER

; ***************************************************************
; DUART_TEST - LOOP BACK TEST FOR DUART SERIAL PORT.
```

```
;
; ASM96 ENTRY:    CALL     DUART_TEST
; RETURNS:        NOTHING
; REQUIRES:       $INCLUDE(INIT96.CON)
;
; GLOBALS:        ALL PLM$REGS
;
; DESCRIPTION:    THIS ROUTINE PERFORMS A LOOP BACK TEST ON THE SERIAL
;                 DURART PORT.  THE TRANS-
;                 MITTERS SENDS 2 BLOCKS OF 256 SEQUENTIAL CHARACTERS
;                 WHICH ARE CAPTURED VIA THE RECEIVER IN LOOP BACK MODE.
;                 ANY ERRORS ARE FATAL AND FORCE THIS ROUTINE TO ENTER A
;                 TRANSMITTER LOCKUP AT 'TX_LOOP'.
; ***************************************************************

DUART_TEST:
;       NOW LOOPBACK TX-RX ON ALL 256 CHARS.
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    BP
        LD      BP,12[SP]       ;POINT TO DATA
        ADD     DX,BP,#2        ;CREATE COMMAND
                                        ;REG. ADDRESS
        LDB     AL,#15H                 ;Reset MR pointer &
                                        ;enable Tx & Rx
        STB     AL,[DX]                 ;MR --> MR1
        LD      DX,BP                   ;GET MODE REG. ADRS.
        LDB     AL,[DX]                 ;MR --> MR2

        LDB     AL,#(LOCAL_LOOP+ONE_STOP) ;SET LOOPBACK MODE
        STB     AL,[DX]
        LDB     BXH,#2          ;NO. OF TEST LOOPS

TX_TEST:
        LD      CX,#256         ;NO. CHARS TO XMIT
        CLRB    BL              ;BL= CHAR.
TX_WAIT:
        LD      DX,BP                   ;GET BASE ADRS.
        INC     DX                      ;PT. TO STATUS REG.
        LDB     AL,[DX]                 ;CHECK STATUS
        JBC     AL,TxRDY_POS,TX_WAIT
        INC     DX                      ;PT. TO Tx REG.
        INC     DX
        STB     BL,[DX]                 ;SEND IT

;       CHAR TRANSMITTED, NOW CHECK IF IT WAS
;       RECEIVED CORRECTLY.
        DEC     DX                      ;PT. TO STATUS REG.
        DEC     DX
RX_WAIT:
        LDB     AL,[DX]                 ;CHECK STATUS
        JBC     AL,RxRDY_POS,RX_WAIT
        INC     DX                      ;PT. TO Rx REG.
        INC     DX
        LDB     AL,[DX]
RX_TEST:
        CMPB    AL,BL                   ;CHAR SAME?
        JNE     TX_LOOP                 ;IF NOT BAD, LOOP BACK.
        INCB    BL                      ;SET UP NEXT CHAR
        DJNZW   CX,TX_WAIT              ;& CHECK ALL CHARS. FF-00
RX_LOOP:
        DJNZ    BXH,TX_TEST

;       SERIAL TEST PASSED, SO TURN OFF LOOP BACK
        LD      DX,BP                   ;CREATE COMMAND
        ADD     DX,#2                   ;REG. ADDRESS
        LDB     AL,#10H                 ;Reset MR pointer
```

```
        STB     AL,[DX]                 ;MR --> MR1
        LD      DX,BP                   ;GET MODE REG. ADRS.
        LDB     AL,[DX]                 ;MR --> MR2
        LDB     AL,#(NORMAL+ONE_STOP)   ;SET NORMAL MODE
        STB     AL,[DX]

;       CLEAN STACK AND RETURN TO CALLER
        POP     BP
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET                             ;AOK, RETURN TO CALLER

;       IF SERIAL IO ERROR, LOOP WITH TRANSMITTER ACTIVELY
;       SENDING A CONTINUOUS 'U' (ie. 055H) CHARACTER.
TX_LOOP:

;       SERIAL TEST PASSED, SO TURN OFF LOOP BACK
        LD      DX,BP                   ;CREATE COMMAND
        ADD     DX,#2                   ;REG. ADDRESS
        LDB     AL,#10H                 ;Reset MR pointer
        STB     AL,[DX]                 ;MR --> MR1
        LD      DX,BP                   ;GET MODE REG. ADRS.
        LDB     AL,[DX]                 ;MR --> MR2
        LDB     AL,#(NORMAL+ONE_STOP)   ;SET NORMAL MODE
        STB     AL,[DX]

TX_AGAIN:
        LDB     DX,BP                   ;CHECK STATUS
        INC     DX
        LDB     AL,[DX]
        JBC     AL,TxRDY_POS,TX_AGAIN
        LDB     AL,#055H        ;CHAR TO TX
        INC     DX
        INC     DX
        STB     AL,[DX]                 ;SEND IT
        SJMP    TX_AGAIN                ;CONTINUE TO LOOP

; ********** MAIN LINE CODE ENDS HERE **********
; PLACE DOWNLOADER ENTRY POINT HERE. THIS AND ALL
; FOLLOWING CODE IS MOVED INTO HIGH MEMORY RAM

        END
```

APPENDIX D

RAN BASE ROUTINE

```
/*=============================================================================

      FILE NAME: BOOT_RAM.C

  FUNCTION NAME(S): boot

=============================================================================*/
#pragma registers(220)
#pragma regconserve
#include <stddef.h>
#include <stdlib.h>
#include <ctype.h>
#include <setjmp.h>
#include <string.h>
#include <80c196.h>
#include "utility.h"
extern void dloada(void);

/*****************************************************************************
*
*   Function:  boot
*
* Description: Boots and executes the download code
*          in the A 196KC processor.  The boot block in
*                    the flash memory is swapped out with the 8K block
*                    in high memory above the 64K boundary (0x10000).
*
*           NOTE:     DO NOT PLACE ANY OTHER FUNCTIONS WITHIN THIS FILE.

*                     It is mandatory that this boot function's
*                     entry point be resident at address 0xc000.  The
*                     INIT_SYS.ASM program will move this boot function
*                     into address 0xc000 and jump to it.  Therefore, no
*                     other code must occupy that address.  If this is
*                     the only function within this file, then it will be
*                     properly loaded into memory at the correct entry point.
*
*                     All access to other functions (ie. download, etc.) must
*                     be via function calls to external functions within some
*                     other file.  Once all functions have been called, it is
*                     necessary for the last function to return to this boot
*                     function.  Upon returning to this function, boot will
*                     execute the program currently loaded into memory via a
*                     jump to address 0x2080.
*
*****************************************************************************/
void boot(void)
{
                                        /* Map flash into standard mode */
          ioport2= 0x01;                /* This line must be first */
          dloada();
                                        /* Run downloaded program */
           _execute(0x2080);            /* This line must be last */
}
```

APPENDIX E

CONTROLLER OF BOOT CODE

```
/*************************************************************
*            FILENAME: DLOAD_ST.C
*
*             PURPOSE:
*          The purpose of this module is to perform system software download.
*
*          DESCRIPTION:
*          This module is designed to resynch, establish communication with DMS and
*          processor to processor. Calculate CRC of the Processor "A" code flash
*          memory block and table "A" code flash memory block.
*
*          SPECIAL REQUIREMENTS:
*          None
*
*          FUNCTIONS:
*          dloada() - main function for this module
*          _PutAchar()          - lower level communication between DMS
*          _Putch_TB()          - lower level communication between two processors
*          _GetAchar()          - lower level communication between DMS
*          _Getch_FB()          - lower level communication between two processors
*          _HaveAchar() - character check
*          _what_to_do() - command execution
*          _flag_error() - error terminate
*          _send_to_MSG()       - communication protocol
*          _GetCmd() - communication protocol (between DMS)
*          _GetResp() - communication protocol (between two processors)
*          _CalculateCRC() - compares the System Software's CRC
*          _Write_buf_into_F_mem()        - programming Intel hex data
*          _Chk_dLoadCode_OK() - check Intel hex data
*          _Is_CRC_ok() - verification of System Software's CRC
*
*************************************************************
*/

#pragma registers(220)
#pragma regconserve

#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <setjmp.h>
#include <string.h>
#include <80c196.h>
#include "utility.h"
#include "pflash.h"
#include "flash.h"
#include "sm-dms.h"
#include "unpak.h"
#include "dloada.h"

/*
*********************************************************************
*          TITLE:
*          dloada()
*
*          PURPOSE:
*          The purpose of this function is to process and examine commands from
*          DMS.
```

```
*
*       ARGUMENTS:
*       void
*
*       DESCRIPTION:
*       This function controls the sequencing of SlideMaker software.
*       Dloada is main function for download processor "A" code that waits
*       the DMS command then call command process function _what_to_do.
*
*       GLOBAL VARIABLES:
*       byte AVOCS - comparision of CRC result
*
*       LOCAL VARIABLES:
*       byte CmdTy - temporary command buffer
*       byte MsgBuffer[130] - temporary data buffer
*       byte *MsgPtr -          data pointer
*
*       SUBORDINATE FUNCTIONS:
*       _HaveAchar();
*       _what_to_do();
*       _Is_CRC_ok();
*       _GetCmd();
*       _flag_error();
*
*************************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*************************************************************************
*/


void dloada(void)
{
        byte CmdTy;
        byte MsgBuffer[180];
        byte *MsgPtr=MsgBuffer;
        int  iy, iz;

        AVOCS=_Is_CRC_ok();
        if(AVOCS)
        {
                for(iy=0;iy<SYS_WAIT_TIME;iy++)
                        for(iz=0;iz<0x0100;iz++);
                _send_to_MSG(PRO_B, 1, STARTSYS_DMC, 0, MsgPtr);
                _GetResp(MsgPtr);
                TurnVpp(Off);
                for(iy=0;iy<SYS_WAIT_TIME;iy++)
                        for(iz=0;iz<0x0100;iz++);
                _execute(0x2080);
                return;
        }

        while(1)
        {
                if(_HaveAchar())
                {
                        CmdTy=_GetCmd(MsgPtr);
                        if((CmdTy>=FIRST_DMS_CMD)&&(CmdTy!=ERROR_SMR))
                                _what_to_do(&CmdTy, MsgPtr);
                        else
                                _flag_error(1, error);

                }
        }
```

```
        return;
}

/*
*************************************************************************
*       TITLE:
*       _PutAchar()
*
*       PURPOSE:
*       The purpose is for lower level communication between DMS
*
*       ARGUMENTS:
*       byte OutChar -          character to be transmitted
*
*       DESCRIPTION:
*       Function for serial communication.
*
*       GLOBAL VARIABLES:
*       Tx - Address for DUART transmitter
*
*       LOCAL VARIABLES:
*       None
*
*       SUBORDINATE FUNCTIONS:
*       None
*
*************************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*************************************************************************
*/


void _PutAchar(byte OutChar)
{
        while((Status&0x04)==0);
        Tx=OutChar;
        return;
}

/*
*************************************************************************
*       TITLE:
*       _Putch_TB()
*
*       PURPOSE:
*       The purpose is for lower level communication between two processors.
*
*       ARGUMENTS:
*       byte OutChar -          character to be transmitted
*
*       DESCRIPTION:
*       Function for serial communication.
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       None
*
*       SUBORDINATE FUNCTIONS:
*       None
*
```

```
*****************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*****************************************************************
*/


void _Putch_TB(byte OutChar)
{
        while((sp_stat&0x08)==0x00);
        sbuf=OutChar;
        return;
}

/*
*****************************************************************
*       TITLE:
*       _GetAchar()
*
*       PURPOSE:
*       The purpose if for lower level communication between DMS.
*
*       ARGUMENTS:
*       byte *ReceivedCh - character received
*
*       DESCRIPTION:
*       Function for serial communication.
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       unsigned int C_count - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       None
*
*****************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       TRUE - successfully received a character
*       FALSE - unsucessful in receiving a character
*
*****************************************************************
*/


byte _GetAchar(byte *ReceivedCh)
{
        unsigned int C_count;
        *ReceivedCh=0x00;
        for(C_count=0; C_count<(WAIT_CH_TIME); C_count++)
        {
                if((Status&0x01)==1)
                {
                        *ReceivedCh=Rx;
                        return TRUE;
                }
        }
        return FALSE;
}
```

```
/*
*************************************************************************
*       TITLE:
*       _Getch_FB()
*
*       PURPOSE:
*       The purpose is for lower level communication between two processors.
*
*       ARGUMENTS:
*       byte *ReceivedCh - received character
*
*       DESCRIPTION:
*       Function for serial communication.
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       unsigned int C_count - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       None
*
*************************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       TRUE - successfully received a character
*       FALSE - unsucessful in receiving a character
*
*************************************************************************
*/


byte _Getch_FB(byte *ReceivedCh)
{
        unsigned int C_count;
        *ReceivedCh=0x00;

        for(C_count=0; C_count<(WAIT_CH_TIME); C_count++)
        {
                if((sp_stat&0x40)==0x40)
                {
                        *ReceivedCh=sbuf;
                        return TRUE;
                }
        }
        return FALSE;
}

/*
*************************************************************************
*       TITLE:
*       _HaveAchar()
*
*       PURPOSE:
*       To flag that a character is arriving.
*
*       ARGUMENTS:
*       void
*
*       DESCRIPTION:
*       To verify that a new command is ready to be received.
*
*       GLOBAL VARIABLES:
*       None
```

```
*
*           LOCAL VARIABLES:
*           None
*
*           SUBORDINATE FUNCTIONS:
*           None
*
*****************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           TRUE - a command is arriving
*           FALSE - no command arriving
*
*****************************************************************
*/


byte _HaveAchar(void)
{
        return ((Status&0x01)==1);
}

/*
*****************************************************************
*           TITLE:
*           _what_to_do()
*
*           PURPOSE:
*           To control the sequencing of SlideMaker software downloading
*           according to the DMS instruction.
*
*           ARGUMENTS:
*           byte *CmdTy - command buffer
*           byte *MsgPtr -          data pointer
*
*           DESCRIPTION:
*           This function initiates the functions to process, and examines the
*           commands from the DMS, and returns the result of the command
*           execution status.
*
*           GLOBAL VARIABLES:
*           byte AVOCS - result of comparison System Software's CRC
*
*           LOCAL VARIABLES:
*           byte  ix - dummy variable
*           byte  cht -  dummy variable
*           char  dumyB - dummy variable
*           byte *BeginAddre - beginning address pointer
*           byte *EndinAddre - ending address pointer
*           byte *AcrcAddre -       System Software's CRC address pointer
*           byte *bloBy - block pointer
*           byte *cmdBy - command pointer
*           byte *lenBy - length pointer
*           byte *datBy - data pointer
*           byte *t_datBy - temporary data pointer
*           byte *programdata - program data pointer
*           byte buffer[180] - data buffer
*           byte *dataBuf - data buffer pointer
*           byte tempaddre - temporary address
*           word tempdata - temporary data
*           int  iy - dummy variable
*           int  iz - dummy variable
*
*           SUBORDINATE FUNCTIONS:
*           _send_to_MSG();
*           _flag_error();
```

```
*          TurnVpp();
*          erase_block();
*          _execute();
*          _Chk_dLoadCode_OK()
*          _Write_buf_into_F_mem();
*          _CalculateCRC();
*          _GetResp();
*          read_byte();
*          write_byte();
*
*************************************************************************
*
*          ERROR MESSAGE LIST:
*          None
*
*          RETURNS:
*          None
*
*************************************************************************
*/


int _what_to_do(byte *CmdTy, byte *MsgPtr)
{
          byte ix;
          byte      cht=0;
          char dumyB;
          byte *BeginAddre;
          byte *EndinAddre;
          byte *AcrcAddre ;
          byte *bloBy, *cmdBy, *lenBy, *datBy, *t_datBy;
          byte *programdata;
          byte buffer[180];
          byte *dataBuf=buffer;
          byte tempaddre;
          word tempdata;
          int iy, iz;

          bloBy=MsgPtr+1;
          cmdBy=MsgPtr+2;
          lenBy=MsgPtr+3;
          datBy=MsgPtr+4;

          if(*CmdTy == INQVER_DMC)
          {
                    AcrcAddre =(byte *)SYSTEM_ACRC_ADDRESS;
                    _send_to_MSG(PRO_B, *bloBy, *cmdBy, *lenBy, datBy);
                    cht=_GetResp(&buffer[0]);
                    if((cht>=FIRST_196_RES)&&(cht!=COMM_TIME_OUT))
                    {
                              bloBy=dataBuf+1;
                              cmdBy=dataBuf+2;
                              lenBy=dataBuf+3;
                              datBy=dataBuf+4;
                              t_datBy=datBy;
                              datBy+=(QurLen);

                              for(ix=0;ix<QurLen;ix++)
                                        read_byte (datBy++, AcrcAddre++);
                              *(datBy)=AVOCS;
                              datBy=t_datBy;
                              _send_to_MSG(_DMS, *bloBy, *cmdBy, (*lenBy)+QurLen, datBy);
                    }
                    else
                              _flag_error(*bloBy, cht);

                    TurnVpp(Off);
          }
```

```
if(*CmdTy == INQDOWNA_DMC)
{
        TurnVpp(On);
        BeginAddre=(byte *)SYSTEM_4000_ADDRESS;
        *(datBy)=12;
        *(datBy+1)=0;
        _send_to_MSG(_DMS, *bloBy, DOWNAWAIT_SMR, 2, datBy);
        erase_block(BeginAddre);
        BeginAddre=(byte *)0X0200;
        erase_block(BeginAddre);
        BeginAddre=(byte *)0X1000;
        erase_block(BeginAddre);
        AVOCS=0;
}

if(*CmdTy == STARTSYS_DMC)
{
        _send_to_MSG(PRO_B, *bloBy, *cmdBy, *lenBy, datBy);
        cht=_GetResp(&buffer[0]);
        if((cht>=FIRST_196_RES)&&(cht!=COMM_TIME_OUT))
        {
                bloBy=dataBuf+1;
                cmdBy=dataBuf+2;
                lenBy=dataBuf+3;
                datBy=dataBuf+4;

                if((AVOCS==1)&&(cht==COMPLETE_SMR))
                {
                        _send_to_MSG(_DMS, *bloBy, *cmdBy, *lenBy, datBy);
                }
                else
                {
                        *datBy=(byte)STARTING_SYS_ERR;
                        _send_to_MSG(_DMS, *bloBy, ERROR_SMR, 1, datBy);
                }
        }
        else
                _flag_error(*bloBy, cht);

        TurnVpp(Off);
        for(iy=0;iy<SYS_WAIT_TIME;iy++)
                for(iz=0;iz<0x10;iz++);
        _execute(0x2080);
}

if(*CmdTy == DOWNABLO_DMC)
{
        if(_Chk_dLoadCode_OK(*lenBy, datBy))
        {
                _Write_buf_into_F_mem(*lenBy, datBy);
                _send_to_MSG(_DMS, *bloBy, NEWABLO_SMR, 0, dataBuf);
                ioport1=(ioport1^0x04);
        }
        else
                _send_to_MSG(_DMS, *bloBy, OLDABLO_SMR, 0, dataBuf);
}


if(*CmdTy == BUILDACRC_DMC)
{
        dataBuf=&buffer;
        if((*(datBy+VerLen)=='C')||(*(datBy+VerLen)=='c'))
        {
                BeginAddre=(byte *)SYSTEM_MAIN_ADDRESS;
                EndinAddre=(byte *)SYSTEM_AEND_ADDRESS;
                AcrcAddre=(byte *)SYSTEM_ACRC_ADDRESS;
        }
        else
        {
```

```
                    BeginAddre=(byte *)SYSTEM_0200_ADDRESS;
                    EndinAddre=(byte *)SYSTEM_MAIN_ADDRESS;
                    AcrcAddre=(byte *)SYSTEM_TCRC_ADDRESS;
            }
            tempdata=_CalculateCRC(BeginAddre, EndinAddre);
            *dataBuf++ =        ((tempdata&0XFF00)>>8);
            *dataBuf++ =        (tempdata&0X00FF);
            dataBuf=&buffer;
            TurnVpp(On);
            write_byte (dataBuf++ ,AcrcAddre++);
            write_byte (dataBuf++ ,AcrcAddre++);
            for(ix=0;ix<VerLen;ix++)
                    write_byte (datBy++ ,AcrcAddre++);
            _send_to_MSG(_DMS, *bloBy, COMPLETE_SMR, 0, &buffer);
            AVOCS=1;
    }

    if(*CmdTy == BUILDBCRC_DMC)
    {
            _send_to_MSG(PRO_B, *bloBy, *cmdBy, *lenBy, datBy);
            cht=_GetResp(&buffer[0]);
            if((cht>=FIRST_196_RES)&&(cht!=COMM_TIME_OUT))
            {
                    bloBy=dataBuf+1;
                    cmdBy=dataBuf+2;
                    lenBy=dataBuf+3;
                    datBy=dataBuf+4;
                    _send_to_MSG(_DMS, *bloBy, *cmdBy, *lenBy, datBy);
            }
            else
                    _flag_error(*bloBy, cht);
    }

    if(*CmdTy == INQDOWNB_DMC)
    {
            _send_to_MSG(PRO_B, *bloBy, *cmdBy, *lenBy, datBy);
            cht=_GetResp(&buffer[0]);
            if((cht>=FIRST_196_RES)&&(cht!=COMM_TIME_OUT))
            {
                    bloBy=dataBuf+1;
                    cmdBy=dataBuf+2;
                    lenBy=dataBuf+3;
                    datBy=dataBuf+4;
                    _send_to_MSG(_DMS, *bloBy, *cmdBy, *lenBy, datBy);
            }
            else
                    _flag_error(*bloBy, cht);
    }

    if(*CmdTy == DOWNBBLO_DMC)
    {
            _send_to_MSG(PRO_B, *bloBy, *cmdBy, *lenBy, datBy);
            cht=_GetResp(&buffer[0]);
            if((cht>=FIRST_196_RES)&&(cht!=COMM_TIME_OUT))
            {
                    bloBy=dataBuf+1;
                    cmdBy=dataBuf+2;
                    lenBy=dataBuf+3;
                    datBy=dataBuf+4;
                    _send_to_MSG(_DMS, *bloBy, *cmdBy, *lenBy, datBy);
                    ioport1=(ioport1^0x08);
            }
            else
                    _flag_error(*bloBy, cht);

#if 0
            _send_to_MSG(PRO_B, *bloBy, *cmdBy, *lenBy, datBy);
            _send_to_MSG(_DMS, *bloBy, DOWNBWAIT_SMR, 0, &buffer);
            while(1)
```

```
                {
                        if(_Getch_FB(&cht))
                        {
                                if(cht==0x1B)
                                        break;
                                while((Status&0x04)==0);
                                Tx=cht;
                        }
                }
                _send_to_MSG(_DMS, *bloBy, NEWABLO_SMR, 0, &buffer);
#endif

        }

        return (1);
}

/*
*************************************************************************
*       TITLE:
*       _flag_error()
*
*       PURPOSE:
*       The purpose of this function is to determine the error type.
*
*       ARGUMENTS:
*       byte bloNo - temporary block number
*       byte result- error type
*
*       DESCRIPTION:
*       This function notifies the DMS that an error has occurred.
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte tempBuf[10] - temporary buffer
*       byte *dataBuf - data pointer
*
*       SUBORDINATE FUNCTIONS:
*       _send_to_MSG();
*
*************************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*************************************************************************
*/


void _flag_error(byte bloNo, byte result)
{

        byte tempBuf[10];                       /* tempBuf is the the buffer for */
        byte *dataBuf=tempBuf;                  /* error type and detail errors  */

#if 0
        if(result==0X00)
                *dataBuf=(byte)PROCE_B_COMM_ERR;
        if(result==0XFF)
                *dataBuf=(byte)EXEPARM_ERR;
        _send_to_MSG(_DMS, bloNo, ERROR_SMR, 1, dataBuf);
#endif

        *dataBuf=result;
```

```
        _send_to_MSG(_DMS, bloNo, ERROR_SMR, 1, dataBuf);
        error=(byte)NO_ERR;
        return;
}

/*
*********************************************************************
*       TITLE:
*       _send_to_MSG()
*
*       PURPOSE:
*       To process response and data to be sent to the DMS or processor "B".
*
*       ARGUMENTS:
*       byte whom -             _DMS or PRO_B
*       byte bloNo - block number
*       byte Respon - response type
*       byte DataLen -          data length
*       byte *DataBuf - data pointer
*
*       DESCRIPTION:
*       This function transmits the response type block number, data buffer, and data length,
*       formats and sends the message to the DMS or PRO_B.
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte ComBuf[180] - command buffer
*       byte *ptr -  pointer
*       byte *msgSta -          temporary pointer
*       byte msgLen - message length
*       byte its - dummy variable
*       byte lrc_lsb -          longitudinal redundency check
*       byte dummyByte - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       _PutAchar();
*       _Putch_TB();
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*********************************************************************
*/


int _send_to_MSG(byte whom, byte bloNo, byte Respon, byte DataLen, byte *DataBuf)
{
        byte ComBuf[180];
        byte *ptr=ComBuf;
        byte *msgSta=ptr;
        byte msgLen, its;
        byte lrc_lsb=STX, dummyByte;

        *ptr++=STX;
        dummyByte=*ptr++=bloNo;
        lrc_lsb=dummyByte^lrc_lsb;
        dummyByte=*ptr++=Respon;
        lrc_lsb=dummyByte^lrc_lsb;
        dummyByte=*ptr++=DataLen;
        lrc_lsb=(dummyByte^lrc_lsb);
        while(DataLen--)
```

```
        {
                dummyByte=*ptr++=*DataBuf++;
                lrc_lsb=(dummyByte^lrc_lsb);
        }
        *ptr++=lrc_lsb;
        *ptr++=ETX;
        msgLen=ptr-msgSta;

        if(whom==_DMS)
        {
                for(its=0; its<msgLen; its++)
                        _PutAchar(*msgSta++);
        }
        else  /* for processor 'B' */
        {
                for(its=0; its<msgLen; its++)
                        _Putch_TB(*msgSta++);
        }
        return;

}

/*
*********************************************************************
*       TITLE:
*       _GetCmd()
*
*       PURPOSE:
*       The purpose is to get command.
*
*       ARGUMENTS:
*       byte *MsgBuf -          message buffer pointer
*
*       DESCRIPTION:
*       Get a command from DMS.
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte lrc_lsb -          LRC
*       byte *ptr -  pointer
*       byte dummyCh -          dummy variable
*       byte d_Len - data length
*       byte timeout - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       None
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       result - successful command or incomplete message
*
*********************************************************************
*/

byte _GetCmd(byte *MsgBuf)
{
        byte lrc_lsb=0x00;
        byte *ptr=MsgBuf;
        byte dummyCh, d_Len;
        byte timeout=TRUE, State;
        State=StxState;
        error=(byte)DMS_TIME_ERR;
        while(1)
```

```
{
        if(_GetAchar(&dummyCh))
        {
                *ptr++=dummyCh;
                lrc_lsb = dummyCh^(lrc_lsb);
                if(State==StxState)
                {
                        if((dummyCh==NAK)||(dummyCh==ACK))
                        {
                                error=(byte)DMS_BADRSP_ERR;
                                break;
                        }
                        if(dummyCh!=STX)
                        {
                                error=(byte)DMS_BADRSP_ERR;
                                break;
                        }
                        State=BloState;
                }
                else
                {
                        if(State==BloState)
                                State=CmdState;
                        else
                        {
                                if(State==CmdState)
                                        State=LenState;
                                else
                                {
                                        if(State==LenState)
                                        {
                                                if(dummyCh>128)
                                                {
                                                        error=(byte)EXEPARM_ERR;
                                                        break;
                                                }
                                                if(dummyCh>0)
                                                {
                                                        State=DatState;
                                                        d_Len=dummyCh;
                                                }
                                                else
                                                        State=LrcState;
                                        }
                                        else
                                        {
                                                if(State==DatState)
                                                {
                                                        if(--d_Len)
                                                                State=DatState;
                                                        else
                                                                State=LrcState;
                                                }
                                                else
                                                {
                                                        if(State==LrcState)
                                                        {
                                                                lrc_lsb = STX^(lrc_lsb);
                                                                if(lrc_lsb)
                                                                {

error=(byte)BAD_LRC_ERR;
                                                                        break;
                                                                }
                                                                else
                                                                        State=EtxState;
                                                        }
                                                        else
                                                        {
```

```
                                                                    if(State==EtxState)
                                                                    {

        if((dummyCh!=EOT)&&(dummyCh!=ETX))
                                                                            {

        error=(byte)EXEPARM_ERR;
                                                                                    break;
                                                                            }
                                                                            timeout=FALSE;
                                                                            break;
                                                                    }
                                                            }
                                                    }
                                            }
                                    }
                            }
                    }
                    else
                            break;
        }

        if(timeout==TRUE)
                return (ERROR_SMR);
        else
        {       error=(byte)NO_ERR;
                return *(MsgBuf+2);
        }
}

/*
*********************************************************************
*       TITLE:
*       _GetResp()
*
*       PURPOSE:
*       Communication with two processors.
*
*       ARGUMENTS:
*       byte *MsgBuf -          message buffer pointer
*
*       DESCRIPTION:
*       Get a response from processor B.
*
*       GLOBAL VARIABLES:
*       None

*       LOCAL VARIABLES:
*       State - dummy variable
*       byte lrc_lsb -          LRC
*       byte *ptr -  pointer
*       byte dummyCh -          dummy variable
*       byte d_Len - data length
*       byte timeout - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       _Getch_FB();
*       Is_timeout();
*       start_timer();
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       result - successful command or incomplete message
*
```

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


byte _GetResp(byte *MsgBuf)
{
        byte lrc_lsb=STX;
        byte *ptr=MsgBuf;
        byte dummyCh, d_Len;
        byte timeout=TRUE, State;
        State=StxState;

        start_timer();
        while(1)
        {
                if((sp_stat&0x40)==0x40)
                {
                        dummyCh=sbuf;
                        timeout=FALSE;
                        break;
                }

                if(Is_timeout(_8_second))
                {
                        timeout=TRUE;
                }
        }

        if(timeout==TRUE)
                return (0x00);
        timeout=TRUE;

        *ptr++=dummyCh;
        if(State==StxState)
        {
                if(dummyCh==NAK)
                        return NAK;
                if(dummyCh==ACK)
                        return ACK;
                if(dummyCh!=STX)
                        return 0;
                lrc_lsb = dummyCh^(lrc_lsb);
                State=BloState;
        }

        while(1)
        {
                if(_Getch_FB(&dummyCh))
                {
                        *ptr++=dummyCh;
                        if(State==BloState)
                        {
                                lrc_lsb = dummyCh^(lrc_lsb);
                                State=CmdState;
                        }
                        else
                        {
                                if(State==CmdState)
                                {
                                        lrc_lsb = dummyCh^(lrc_lsb);
                                        State=LenState;
                                }
                                else
                                {
                                        if(State==LenState)
                                        {
                                                if(dummyCh>128)
                                                        return 0;
                                                if(dummyCh>0)
```

```
                                                {
                                                        State=DatState;
                                                        d_Len=dummyCh;
                                                        lrc_lsb = dummyCh^(lrc_lsb);
                                                }
                                                else
                                                        State=LrcState;
                                        }
                                        else
                                        {
                                                if(State==DatState)
                                                {
                                                        if(--d_Len)
                                                                State=DatState;
                                                        else
                                                                State=LrcState;
                                                        lrc_lsb = dummyCh^(lrc_lsb);
                                                }
                                                else
                                                {
                                                        if(State==LrcState)
                                                        {
                                                                if(lrc_lsb==dummyCh)
                                                                        State=EtxState;
                                                                else

        error=(byte)PROCE_B_COMM_ERR;
                                                        }
                                                        else
                                                        {
                                                                if(State==EtxState)
                                                                {
                                                                        timeout=FALSE;
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                }
                else
                        break;
        }
        if(timeout==TRUE)
                return (COMM_TIME_OUT);
        else
                return *(MsgBuf+2);
}

/*
 ********************************************************************
 *       TITLE:
 *       _CalculateCRC()
 *
 *       PURPOSE:
 *       To calculate System Software and Table CRC.
 *
 *       ARGUMENTS:
 *       byte *BloAddreBeg - first address of the block
 *       byte *BloAddreEnd - last address of the block
 *
 *       DESCRIPTION:
 *       This function calculates and returns CRC of System Software in code
 *       segment memory area.
 *
 *       GLOBAL VARIABLES:
 *       None
 *
 *       LOCAL VARIABLES:
```

```
*           byte dummyCrcL - dummy variable
*           byte dummyCrcH - dummy variable
*           byte *BegAddre - beginning address pointer
*           byte *EndAddre - ending address pointer
*           word SScrc - dummy variable
*           word TempCrc - dummy variable
*           unsigned long int SStotal - dummy variable
*           unsigned int HalfLen - dummy variable
*           unsigned int ByteLen - dummy variable
*
*           SUBORDINATE FUNCTIONS:
*           read_byte ();
*
*********************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           SScrc - result of CRC
*
*********************************************************************
*/


word _CalculateCRC(byte *BloAddreBeg, byte *BloAddreEnd)
{
            byte dummyCrcL;
            byte dummyCrcH;
            byte *BegAddre = (byte *)BloAddreBeg;
            byte *EndAddre = (byte *)BloAddreEnd;
            word SScrc=0, TempCrc;
            unsigned long int SStotal=0;
            unsigned int HalfLen, ByteLen;

            HalfLen=(unsigned int)(EndAddre-BegAddre)/2;
            for(ByteLen=0;ByteLen<HalfLen;ByteLen++)
            {
                        read_byte (&dummyCrcH, BegAddre++);
                        read_byte (&dummyCrcL, BegAddre++);
                        TempCrc=(0X00FF & dummyCrcH) << 8;
                        SStotal=( SStotal+(TempCrc | dummyCrcL) );
            }
            SScrc=(0X0000FFFF & SStotal);
            SScrc=(SScrc-((0XFFFF0000 & SStotal) >> 16));

            return SScrc;

}

/*
*********************************************************************
*           TITLE:
*           _Write_buf_into_F_mem()
*
*           PURPOSE:
*           To write download code into flash memory.
*
*           ARGUMENTS:
*           byte dataLen -          data length
*           byte *dLoadCode -       download data pointer
*
*           DESCRIPTION:
*           This function loads a buffer to flash memory at a given address.
*           The function pass parameters are starting memory address, length of the
*           buffer to be written, and message pointer.
*
*           GLOBAL VARIABLES:
*           ptw.pbyte - a byte of data
```

```
*           ptw.pbit -  4 bits of data
*
*           LOCAL VARIABLES:
*           byte  *StartAddre - starting address
*           byte  len -  length
*           byte  pos_count -       dummy variable
*           byte  CodeByte - dummy variable
*           int   lineNo -          dummy variable
*           word  TempAddre -       dummy variable
*
*           SUBORDINATE FUNCTIONS:
*           write_byte ();
*
*****************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           TRUE - successful program download data
*           FALSE - incomplete program
*
*****************************************************************
*/


byte _Write_buf_into_F_mem(byte dataLen, byte *dLoadCode)
{
            byte  *StartAddre;
            byte  len, pos_count;
            byte  CodeByte=0;
            int   lineNo;
            word  TempAddre;

            for(lineNo=0;lineNo<LTR;lineNo++)
            {
                        ptw.pbyte=*(dLoadCode+CodeByte++);
                        ptw.pbit.n2=QUCOR_X5(ptw.pbit.n2);
                        len=ptw.pbit.n2*(SI+1)+ptw.pbit.n1;

                        if(CodeByte>dataLen)
                                    return FALSE;

                        TempAddre=(*(dLoadCode+CodeByte++)<<8);
                        TempAddre=(TempAddre|*(dLoadCode+CodeByte++));
                        if(CodeByte>dataLen)
                                    return FALSE;
                        StartAddre=(byte *)(TempAddre);

                        for(pos_count=3;pos_count<len+3;pos_count++)
                        {
                                    write_byte (dLoadCode+CodeByte, StartAddre++);
                                    if(CodeByte>dataLen)
                                                return FALSE;
                                    CodeByte++;
                        }
                        CodeByte++;  /* Intel check 'SUM' byte */

                        if(CodeByte==dataLen)
                                    break;
            }
            return TRUE;

}

/*
 *****************************************************************
 *          TITLE:
 *          _Chk_dLoadCode_OK()
```

```
*
*       PURPOSE:
*       The purpose of this function is to check download code before program.
*
*       ARGUMENTS:
*       byte dataLen -          dummy variable
*       byte *dLoadCode - download data pointer
*
*       DESCRIPTION:
*       This function evaluates download code (INTEL hex format).
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte  *StartAddre - temporary starting address pointer
*       byte *STA_ADD - starting address pointer
*       byte *END_ADD - ending address pointer
*       byte  len - length
*       byte pos_count - dummy variable
*       byte  CodeByte -dummy variable
*       int   lineNo - dummy variable
*       byte  CkSum - dummy variable
*       word  TempAddre - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       None
*
*************************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       TRUE - result of Intel hex format is correct.
*       FALSE - incorrect Intel hex format
*
*************************************************************************
*/


byte _Chk_dLoadCode_OK(byte dataLen, byte *dLoadCode)
{
        byte  *StartAddre;
        byte *STA_ADD=(byte *)SYSTEM_0200_ADDRESS;
        byte *END_ADD=(byte *)SYSTEM_AEND_ADDRESS;
        byte  len, pos_count;
        byte  CodeByte=0;
        int   lineNo;
        byte  CkSum;
        word  TempAddre;

        for(lineNo=0;lineNo<LTR;lineNo++)
        {
                ptw.pbyte=*(dLoadCode+CodeByte++);
                ptw.pbit.n2=QUCOR_X5(ptw.pbit.n2);
                len=ptw.pbit.n2*(SI+1)+ptw.pbit.n1;

                if(CodeByte>dataLen)
                        return FALSE;
                CkSum=len;

                CkSum=CkSum+*(dLoadCode+CodeByte);
                TempAddre=(*(dLoadCode+CodeByte++)<<8); /* make sure 14/9/94 */
                CkSum=CkSum+*(dLoadCode+CodeByte);
                TempAddre=(TempAddre|*(dLoadCode+CodeByte++));
                if(CodeByte>(dataLen-2))
                        return FALSE;
                StartAddre=(byte *)(TempAddre);
```

```
                if((StartAddre<STA_ADD)||(StartAddre>END_ADD))
                        return FALSE;

                for(pos_count=3;pos_count<len+3;pos_count++)
                {
                        CkSum=CkSum+*(dLoadCode+CodeByte);
                        if(CodeByte>(dataLen-1))
                                return FALSE;
                        CodeByte++;
                }
                if((((CkSum^0xFF)+1)&0x00FF)!=(byte)*(dLoadCode+CodeByte))
                        /* Intel check 'SUM' byte */
                        return FALSE;
                CodeByte++;
                if(CodeByte>=(dataLen-2))
                        break;
        }
        return TRUE;

}

/*
*********************************************************************
*       TITLE:
*       _Is_CRC_ok()
*
*       PURPOSE:
*       The purpose of this function is to evaluate CRC.
*
*       ARGUMENTS:
*       void
*
*       DESCRIPTION:
*       This function compares System Software's CRC against
*       flash code block.
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte result - dummy variable
*       byte CrcBuffer[2] - dummy variable
*       byte *CrcPtr -          CRC pointer
*       byte *BeginAddre - beginning address pointer
*       byte *EndinAddre - ending address pointer
*       byte *AcrcAddre - CRC address pointer
*       word tempcrc -          dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       read_byte ();
*       _CalculateCRC();
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       result - compares CRC against flash code block
*
*********************************************************************
*/

byte _Is_CRC_ok(void)
{
        byte result=0;
        byte CrcBuffer[2];
        byte *CrcPtr=CrcBuffer;
```

```
    byte *BeginAddre=(byte *)SYSTEM_MAIN_ADDRESS;
    byte *EndinAddre=(byte *)SYSTEM_AEND_ADDRESS;
    byte *AcrcAddre =(byte *)SYSTEM_ACRC_ADDRESS;
    word tempcrc=0;

    read_byte (CrcPtr, AcrcAddre);
    tempcrc =((*CrcPtr&0X00FF)<<8);
    read_byte (CrcPtr, AcrcAddre+1);
    tempcrc =((*CrcPtr&0X00FF)|tempcrc);

    if(_CalculateCRC(BeginAddre, EndinAddre)==tempcrc)
            result=1;

    BeginAddre=(byte *)SYSTEM_0200_ADDRESS;
    EndinAddre=(byte *)SYSTEM_MAIN_ADDRESS;
    AcrcAddre =(byte *)SYSTEM_TCRC_ADDRESS;
    tempcrc=0;

    read_byte (CrcPtr, AcrcAddre);
    tempcrc =((*CrcPtr&0X00FF)<<8);
    read_byte (CrcPtr, AcrcAddre+1);
    tempcrc =((*CrcPtr&0X00FF)|tempcrc);

    if(_CalculateCRC(BeginAddre, EndinAddre)!=tempcrc)
            result=0;

    return result;
}
```

APPENDIX F

FLASH HANDLER ROUTINE

```
/*************************************************************
 *
 *          FILENAME:          FLASH_PE.C
 *
 *          PURPOSE:
 *          Controls block erase and byte program.
 *
 *          DESCRIPTION:
 *          This module will operates block erasing and byte programming.
 *          Initializes device powerup and powerdown mode. Handling all
 *          timeout cases which occur at processing flash memory and
 *          receiving message.
 *
 *          SPECIAL REQUIREMENTS:
 *          None
 *
 *          FUNCTIONS:
 *          TurnVpp() - control Vpp source for flash device
 *          TurnModeTo() - control operating memory mode
 *          erase_begin() - erasing memory
 *          erase_suspend() - erasing suspend
 *          erase_resume() - erasing resume
 *          finished() - check status cases
 *          erase_check() - erasing check
 *          program_begin() - programming memory
 *          program_check() - programming check
 *          status_reg_read() - setup read status reg. mode
 *          status_reg_clear() - reset status reg.
 *          read_array_mode() - setup read data mode
 *          read_byte() - read data from memory
 *          read_ID() - read device ID
 *          write_byte() - programming byte
 *          erase_block() - erasing memory block
 *          start_timer() - timer started
 *          Is_timeout() - check timeout
 *
 *************************************************************
 */

#pragma registers(220)
#pragma regconserve
#include <stddef.h>
#include <stdlib.h>
#include <ctype.h>
#include <setjmp.h>
#include <string.h>
#include <80c196.h>
#include "flash.h"
#include "f_def.h"

/*
 *********************************************************************
 *          TITLE:
 *          TurnVpp()
 *
 *          PURPOSE:
 *          Control Vpp source for flash device
 *
 *          ARGUMENTS:
```

```
*           byte On_Off - Vpp on /or off
*
*           DESCRIPTION: This function turns Vpp on /or off.
*
*           GLOBAL VARIABLES:
*           None
*
*           LOCAL VARIABLES:
*           int iyy - dummy variable
*
*           SUBORDINATE FUNCTIONS:
*           None
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           None
*
*************************************************************************
*/

void TurnVpp(byte On_Off)
{
            int iyy;
            if(On_Off)
            {
                        ioport2=(ioport2|0x20);
                        for(iyy=0;iyy<=10000;iyy++);
            }
            else
            {
                        ioport2=(ioport2&0xDF);
            }
}

/*
*************************************************************************
*           TITLE:
*           TurnModeTo()
*
*           PURPOSE:
*           Control operating memory mode
*
*           ARGUMENTS:
*           byte Which_Mode - boot mode, standard mode, /or extended mode
*
*           DESCRIPTION: This function turns memory mapping into following mode
*           POWER_UP_MODE, STANDARD_MODE, and EXTENDED_MODE.
*
*           GLOBAL VARIABLES:
*           None
*
*           LOCAL VARIABLES:
*           byte modedata - dummy variable
*
*           SUBORDINATE FUNCTIONS:
*           None
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           None
*
```

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


void TurnModeTo(byte Which_Mode)
{
        byte modedata=ioport2;
        ioport2=((modedata&0X3F)|Which_Mode);
}

/*
*************************************************************************
*       TITLE:
*       erase_begin()
*
*       PURPOSE:
*       erasing memory
*
*       ARGUMENTS:
*       byte *blckaddr - which block to be erased
*
*       DESCRIPTION: begins erase of the specified block
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       None
*
*       SUBORDINATE FUNCTIONS:
*       status_reg_clear();
*
*************************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*************************************************************************
*/


byte erase_begin (byte  *blckaddr)
{
        *blckaddr = (byte)READ_STATUS_REGISTER;
        if ( WSM_READY != *blckaddr )
                status_reg_clear(blckaddr);

        *blckaddr = (byte)ERASE_SETUP;
        *blckaddr = (byte)ERASE_CONFIRM;
        return (0);
}

/*
*************************************************************************
*       TITLE:
*       erase_suspend()
*
*       PURPOSE:
*       erasing suspend
*
*       ARGUMENTS:
*       byte *stataddr - block address
*
*       DESCRIPTION: suspends block erase to allow for reads in another block.
*
*       GLOBAL VARIABLES:
```

```
*           None
*
*           LOCAL VARIABLES:
*           None
*
*           SUBORDINATE FUNCTIONS:
*           start_timer();
*           Is_timeout();
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           1 - erase suspended
*           0 - it is not
*
*************************************************************************
*/


byte erase_suspend (byte *stataddr)
{

            *stataddr = (byte)SUSPEND_ERASE;
            *stataddr = (byte)READ_STATUS_REGISTER;
            start_timer();
            while ((*stataddr & READY_MASK) != WSM_READY)
                        if(Is_timeout(_6_second))
                        ;
            if ((*stataddr & SUSPEND_MASK) == ERASE_SUSPENDED)
                        return (0);
            return (1);
}

/*
*************************************************************************
*           TITLE:
*           erase_resume()
*
*           PURPOSE:
*           erase resume
*
*           ARGUMENTS:
*           byte *stataddr - erase block address
*
*           DESCRIPTION: resumes block erase
*
*           GLOBAL VARIABLES:
*           None
*
*           LOCAL VARIABLES:
*           None
*
*           SUBORDINATE FUNCTIONS:
*           start_timer();
*           Is_timeout();
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           1 - erase not resume
*           0 - erase resume
*
*************************************************************************
```

```
*/

byte erase_resume(byte *stataddr)
{
        *stataddr = (byte)READ_STATUS_REGISTER;
        if ((*stataddr & SUSPEND_MASK) != ERASE_SUSPENDED)
                return (1);
        *stataddr = (byte)RESUME_ERASE;
        start_timer();
        while ((*stataddr & SUSPEND_MASK) == ERASE_SUSPENDED)
                if(Is_timeout(_6_second))
                ;
        start_timer();
        while ((*stataddr & READY_MASK) == WSM_READY)
                if(Is_timeout(_6_second))
                ;
        return (0);
}

/*
*********************************************************************
*       TITLE:
*       finished()
*
*       PURPOSE:
*       check status case
*
*       ARGUMENTS:
*       byte *stataddr - block memory to be check
*
*       DESCRIPTION: checks to see if program/erase is completed
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       None
*
*       SUBORDINATE FUNCTIONS:
*       None
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       1 - incomplete
*       0 - complete
*
*********************************************************************
*/


byte finished (byte *stataddr)
{
        *stataddr = (byte)READ_STATUS_REGISTER;
        if (( *stataddr & READY_MASK) != WSM_READY)
                return (1);
        return (0);
}

/*
*********************************************************************
*       TITLE:
*       erase_check()
*
*       PURPOSE:
```

```
*           erase check
*
*           ARGUMENTS:
*           byte *stataddr - block to be erased
*
*           DESCRIPTION: this function checks status cases
*
*           GLOBAL VARIABLES:
*           None
*
*           LOCAL VARIABLES:
*           byte statdata - dummy variable
*
*           SUBORDINATE FUNCTIONS:
*           status_reg_read();
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           0 - no error
*           1 - Vpp low
*           2 - erase error
*           3 - seq. fail
*
*************************************************************************
*/


byte erase_check (byte *stataddr)
{
            byte statdata;
            statdata=status_reg_read(stataddr);
            if ((statdata & VPP_LOW_MASK) == VPP_LOW)
                        return (1);
            if ((statdata & ERASE_MASK) == ERASE_ERROR)
                        return (2);
            if ((statdata & ERASE_SEQ_MASK) == ERASE_SEQ_FAIL)
                        return (3);
            return (0);
}

/*
 *************************************************************************
 *          TITLE:
 *          program_begin()
 *
 *          PURPOSE:
 *          programming memory
 *
 *          ARGUMENTS:
 *          byte *paddr - program address
 *          byte pdata - program data
 *
 *          DESCRIPTION: begins byte programming sequence
 *
 *          GLOBAL VARIABLES:
 *          None
 *
 *          LOCAL VARIABLES:
 *          None
 *
 *          SUBORDINATE FUNCTIONS:
 *          None
 *
 *************************************************************************
 *
```

```
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           None
*
*************************************************************************
*/


byte program_begin(byte pdata, byte *paddr)
{
            *paddr = (byte)PROGRAM_SETUP_COMMAND;
            *paddr = pdata;
            return (0);
}

/*
*************************************************************************
*           TITLE:
*           program_check()
*
*           PURPOSE:
*           programming check
*
*           ARGUMENTS:
*           byte *stataddr - block to be check
*
*           DESCRIPTION: Full Status Register check for byte program
*
*           GLOBAL VARIABLES:
*           None
*
*           LOCAL VARIABLES:
*           byte statdata -
*
*           SUBORDINATE FUNCTIONS:
*           status_reg_read(stataddr);
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           0 - no error
*           1 - Vpp low
*           2 - error
*
*************************************************************************
*/


byte program_check(byte *stataddr)
{
            byte statdata;
            statdata=status_reg_read(stataddr);
            if ((statdata & VPP_LOW_MASK) == VPP_LOW)
                        return (1);
            if ((statdata & PROGRAM_ERROR_MASK) == PROGRAM_ERROR)
                        return (2);
            return (0);
}

/*
*************************************************************************
*           TITLE:
*           status_reg_read()
*
```

```
*           PURPOSE:
*           setup read status reg. mode
*
*           ARGUMENTS:
*           byte *stataddr - block to be operated
*
*           DESCRIPTION: reads the Status Register and returns its contents
*
*           GLOBAL VARIABLES:
*           None
*
*           LOCAL VARIABLES:
*           None
*
*           SUBORDINATE FUNCTIONS:
*           None
*
*****************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           *stataddr - status
*
*****************************************************************
*/


byte status_reg_read (byte *stataddr)
{
            *stataddr = (byte)READ_STATUS_REGISTER;
            return (*stataddr);
}

/*
*****************************************************************
*           TITLE:
*           status_reg_clear()
*
*           PURPOSE:
*           reset status reg.
*
*           ARGUMENTS:
*           byte *stataddr - block to be operated
*
*           DESCRIPTION: clears the status register
*
*           GLOBAL VARIABLES:
*           None
*
*           LOCAL VARIABLES:
*           None
*
*           SUBORDINATE FUNCTIONS:
*           None
*
*****************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           None
*
*****************************************************************
*/
```

```
byte status_reg_clear (byte *stataddr)
{
        *stataddr = (byte)CLEAR_STATUS_REGISTER;
        return;
}

/*
*********************************************************************
*       TITLE:
*       read_array_mode()
*
*       PURPOSE:
*       setup read data mode
*
*       ARGUMENTS:
*       byte *stataddr - block to be operated
*
*       DESCRIPTION: puts the device in Read Array Mode
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       None
*
*       SUBORDINATE FUNCTIONS:
*       None
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*********************************************************************
*/


byte read_array_mode (byte *stataddr)
{
        *stataddr = (byte)READ_ARRAY_MODE_COMMAND;
        return;
}

/*
*********************************************************************
*       TITLE:
*       read_byte()
*
*       PURPOSE:
*       read data from memory
*
*       ARGUMENTS:
*       byte *bytedata - data read from memory
*       byte *byteaddr - address to be read
*
*       DESCRIPTION: Reads a byte of data from the specified address
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       None
*
*       SUBORDINATE FUNCTIONS:
*       None
*
```

```
*****************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       None
*
*****************************************************************
*/


byte read_byte (byte *bytedata, byte *byteaddr)
{
        *byteaddr = (byte)READ_ARRAY_MODE_COMMAND;
        *bytedata = *byteaddr;
        return;
}

/*
*****************************************************************
*       TITLE:
*       write_byte()
*
*       PURPOSE:
*       programming
*
*       ARGUMENTS:
*       byte *bytedata - data to be program
*       byte *byteaddr - address to be program
*
*       DESCRIPTION: write a byte of data from the specified address
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte *statdata - dummy variable
*       byte pbegin - dummy variable
*       pcheck - dummy variable
*       pclear - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       program_begin ();
*       finished();
*       program_check();
*       status_reg_clear();
*
*****************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       Error - complete or/ error
*
*****************************************************************
*/


byte write_byte (byte *bytedata, byte *byteaddr)
{
        byte    Error=0;
        byte    *statdata;
        byte    pbegin, pcheck=1, pclear;

        start_timer();
        while (pcheck != 0)
        {
```

```
                pbegin = program_begin (*bytedata, byteaddr);
                while (finished(byteaddr) == 1)
                {
                        if(Is_timeout( _6_second ))
                        {
                                Error=10;
                                pcheck = 0;
                                break;
                        }
                }
                pcheck = program_check(byteaddr);
                Error+=pcheck;
                pclear = status_reg_clear(byteaddr);
                pcheck = 0;
        }

        return (Error);
}

/*
*********************************************************************
*       TITLE:
*       read_ID()
*
*       PURPOSE:
*       read device ID
*
*       ARGUMENTS:
*       byte *mfid - Intelligent Identifier (Mfr)
*       byte *dvid - Intelligent Identifier (Device)
*
*       DESCRIPTION: reads manufacturer and device ID
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte *stataddr - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       read_array_mode ();
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       0 - match ID
*       1 - not match ID
*
*********************************************************************
*/


byte read_ID (byte *mfid, byte *dvid)
{
        byte *stataddr;
        stataddr = (byte *)MANUFACTURER_ID_ADDR;
        *stataddr = (byte)ID_READ_COMMAND;
        *mfid = *stataddr++;
        *dvid = *stataddr;
        read_array_mode (stataddr);

        if ((*mfid != MANUFACTURER_ID) || (*dvid != DEVICE_ID))
                return (1);
        return (0);
}
```

```
/*
*********************************************************************
*       TITLE:
*       erase_block()
*
*       PURPOSE:
*       erasing memory block
*
*       ARGUMENTS:
*       byte *blckaddr - block to be erased
*
*       DESCRIPTION: erase flash memory block
*
*       GLOBAL VARIABLES:
*       None
*
*       LOCAL VARIABLES:
*       byte *statdata - dummy variable
*       byte *mmfid - dummy variable
*       byte *ddvid - dummy variable
*       byte echeck - dummy variable
*       byte Error - dummy variable
*
*       SUBORDINATE FUNCTIONS:
*       erase_begin();
*       finished();
*       read_array_mode();
*
*********************************************************************
*
*       ERROR MESSAGE LIST:
*       None
*
*       RETURNS:
*       Error - complete /or error
*
*********************************************************************
*/


byte erase_block (byte  *blckaddr)
{
        byte *statdata;
        byte *mmfid, *ddvid;
        byte  echeck, Error;

        Error=0;
        echeck = 4;
        start_timer();
        while (echeck != 0)
        {
                erase_begin(blckaddr);
                while (finished(blckaddr) ==1)
                        if(Is_timeout(_6_second))
                        {
                                Error=10;
                                echeck = 0;
                                break;
                        }
                echeck = erase_check(blckaddr);
                read_array_mode (blckaddr);

                Error+=echeck;
                echeck = 0;
        }
        return Error;
}

/*
```

```
*************************************************************************
*           TITLE:
* start_timer()
*
*           PURPOSE:
*           timer started
*
*           ARGUMENTS:
*           None
*
*           DESCRIPTION: Initial the Loop
*
*           GLOBAL VARIABLES:
*           Loop - total cycle of the timer loop
*
*           LOCAL VARIABLES:
*           None
*
*           SUBORDINATE FUNCTIONS:
*           None
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           None
*
*************************************************************************
*/

void start_timer()
{
          Loop=0X00;
          timer1=0X00;
          return;
}

/*
*************************************************************************
*           TITLE:
*           Is_timeout()
*
*           PURPOSE:
*           check timeout
*
*           ARGUMENTS:
*           byte No_Seconds - number of seconds for the timer
*
*           DESCRIPTION: Check time clock
*
*           GLOBAL VARIABLES:
*           Loop - total cycle of the timer loop
*
*           LOCAL VARIABLES:
*           None
*
*           SUBORDINATE FUNCTIONS:
*           None
*
*************************************************************************
*
*           ERROR MESSAGE LIST:
*           None
*
*           RETURNS:
*           TRUE - if time is up.
*
```

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

byte Is_timeout(byte No_Seconds)
{
                    if(timer1>=0XFFF0)
                    {
                              timer1=0x00;
                              Loop++;
                    }
                    if(Loop>=No_Seconds)
                              return On;
                    return Off;
}
```

Reference

[1] Intel "Flash Memory Volume I", 1995.

[2] H. Chi "Download/Memory Software Design Description", Coulter Corporation P/N #0301-0932, December 7, 1993.

[3] Intel "Vision 94", Intel's Embedded Solution Seminar Proceeding, 1994.

[4] Intel "16-Bit Embedded Controllers", 1990.

[5] AMD "Flash Memory Products", 1995.

[6] Dave Bursky "Flash-Memory Choices Boost Performance and Flexibility", Page 63, Electronic Design, May 30, 1995.

[7] Intel "Memory Products", 1993