TEXAS INSTRUMENTS

TI486DX2 Microprocessor SM Mode Programming

Reference Guide

1996

PC System Products

Texas Instruments

Printed in U.S.A.
0196

SRZU019A

TEXAS INSTRUMENTS

TI486DX2 Microprocessor SM Mode Programming

1996

PC System Products

Texas Instruments

Printed in U.S.A.
0196

SRZU019A

Reference Guide

[redacted]

[redacted]

TI486DX2 Microprocessor SM Code Programming

Reference Guide

SRZU019
February 1996

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Preface

Read This First

About This Manual

This reference guide provides information that is helpful when writing or converting software code to implement System Management (SM) mode control for the TI486DX2 microprocessor.

How to Use This Manual

This document contains the following chapters:

Chapter 1 Overview

Chapter 1 provides an overview of the special system management (SM) mode, which can be used to control power management and related functions. The chapter provides comparisons of different SM techniques used and illustrates a typical SM handler.

Chapter 2 SM Hardware Overview

Chapter 2 provides an overview of the hardware used to implement SM mode. In addition to discussing the SM-related inputs and outputs, cache coherency, and Configuration Control registers, descriptions of the SM instructions are included.

Chapter 3 Software Considerations

Chapter 3 contains information on several topics that are useful when developing SM software for the TI486DX2 microprocessor. Specific items are addressed that can simplify the implementation and debugging of system code. Example code, CPU save routines, instruction usage, and problem workarounds are included.

Chapter 4 Converting Intel SM Code to TI486DX2 SM Code

Chapter 4 contains information necessary to convert SM code originally written for Intel™ SL-enhanced processors to run on the TI486DX2 microprocessor. Major SM strategies are addressed, and specific actions are described that can ensure system integrity.

Appendix A SM Mode Macros

Appendix A contains macros that are used in the other code examples within this document.

Related Documentation From Texas Instruments

TI486DX2 Microprocessor Reference Guide (literature number SRZU018A) describes the TI486DX2 microprocessor, programmer interface, bus interface, and instruction set.

TI486DX2 Microprocessor data sheet (literature number SRZS006A) specifies terminal assignments, signal descriptions, electrical connections, electrical and ac characteristics, timing, and mechanical information for all versions of theTI486DX2 microprocessor.

TI486DX2 Microprocessor — Converting Cyrix™ and Intel™ 486 Microprocessor Designs Application Report (literature number SRZU007). This application report assists the system designer in converting existing 486DX2 and 486DX4 designs to support use of the TI486DX2 microprocessor.

TI486DX4 Microprocessor Reference Guide (literature number SRZU020A) describes the TI486DX4 microprocessor, programmer interface, bus interface, and instruction set.

TI486DX4 Microprocessors data sheet (literature number SRZS012A) is a companion document that specifies terminal assignments, signal descriptions, electrical connections, electrical and ac characteristics, timing, and mechanical information for all versions of theTI486DX4 microprocessor.

TI486DX4 Microprocessor SM Mode Programming Reference Guide (literature number SRZU021A) contains detailed information on programming the system management mode of the TI486DX2 and TI486DX4.

Trademarks

Intel is a trademark of Intel Corporation.

Microsoft is a trademark of Microsoft Corporation.

QEMM-386 is a trademark of Quarterdeck Office Systems.

Contents

1 Overview 1-1
1.1 Introduction 1-2
1.2 SM Mode Comparison Among Different Manufacturers 1-3
1.3 Overview of a Typical SM Handler 1-4

2 SM Hardware Overview 2-1
2.1 SM-Related I/O Pins 2-2
2.1.1 SMI# Pin 2-2
2.1.2 SMADS# Pin 2-2
2.1.3 RDY# Pin 2-3
2.1.4 A20M# Pin 2-4
2.2 Cache Coherency 2-5
2.3 Configuration Control Registers 2-6
2.3.1 Access 2-6
2.3.2 Descriptions 2-6
2.4 SM Instruction Summary 2-11
2.4.1 Data Format Used by Many SM Instructions 2-11
2.4.2 Macros for Implementing SM Instructions 2-11
2.4.3 Restore Segment Register and Descriptor (RSDC) 2-12
2.4.4 Restore LDTR and Descriptor (RSLDT) 2-12
2.4.5 Resume Normal Mode Operation (RSM) 2-12
2.4.6 Restore TR and Descriptor (RSTS) 2-12
2.4.7 Software-Generated SM Interrupt (SMINT) 2-13
2.4.8 Save Segment Register and Descriptor (SVDC) 2-13
2.4.9 Save LDTR and Descriptor (SVLDT) 2-13
2.4.10 Save TR and Descriptor (SVTS) 2-13

3 Software Considerations 3-1
3.1 Determining the Addresses Used for SM Memory 3-2
3.1.1 Determining SM Memory Size 3-2
3.1.2 Determining SM Base Address 3-2
3.2 Enabling SM Mode 3-4
3.3 SM Handler Entry Conditions 3-5
3.4 SM Save Space 3-6
3.4.1 Current IP and Next IP 3-7
3.4.2 I/O Trap Information 3-8
3.4.3 The S Bit 3-8
3.4.4 CS, CR0, EFLAGS, and DR7 Registers 3-8
3.5 Maintaining the CPU State 3-9

3.5.1 Preserving and Restoring Normal CPU Registers 3-10
3.5.2 Preserving and Restoring Segment Registers 3-11
3.5.3 Preserving Other Special Registers 3-12
3.5.4 Preserving the State of the Floating Point Unit 3-13
3.6 Initializing the SM Environment 3-14
3.7 Accessing Main Memory Coincident with SM Memory 3-15
3.8 I/O Restart 3-16
3.9 I/O Shadowing and Emulation 3-18
3.10 The HLT Instruction 3-19
3.10.1 HLT and Memory Managers 3-19
3.10.2 EMM386 Problem Overview 3-19
3.10.3 EMM386 Problem Discussion 3-20
3.10.4 EMM386 Problem Workaround: SM Handler Returns to HLT 3-21
3.10.5 EMM386 Workaround: SM Handler Modifies EMM386 Stack 3-24
3.10.6 EMM386 Workaround: Patching EMM386 3-26
3.11 Exiting the SM Handler 3-28
3.12 Debugging SM Code 3-29
3.13 Suspend Mode 3-30
3.14 SM Mode Select 3-31

4 Converting Intel SM Code to TI486DX2 SM Code 4-1
4.1 Differences in TI and Intel SM Implementations 4-2
4.2 SM Code Conversion 4-3
4.2.1 SM Memory Location and Size 4-3
4.2.2 SM Handler Entry Point 4-3
4.2.3 Access to SM Memory 4-3
4.2.4 Access to Main Memory at SM Memory Addresses 4-4
4.2.5 Register Save Area 4-4
4.2.6 Registers Saved 4-4
4.2.7 Processor Mode Within the SM Handler 4-5
4.2.8 Instruction Restart 4-6
4.2.9 NMI Servicing 4-7
4.2.10 Multiple SMIs 4-7
4.2.11 A20M# Input 4-7
4.2.12 SM Mode Select 4-8

A SM Mode Macros A-1
SM Mode Macro File A-2

B Glossary B-1

Figures

1–1 A Typical SM Handler for Power Management 1-4
2–1 Configuration Control Register 1 (CCR1) 2-7
2–2 Configuration Control Register 2 (CCR2) 2-8
2–3 Configuration Control Register 3 (CCR3) 2-9
2–4 SM Address Region (SMAR) Registers 2-10
2–5 DESCSEL Data Format 2-11
3–1 SMM Memory Space Header 3-6

Tables

2–1 Effects of SMAC and MMAC on ADS# and SMADS# 2-3
2–2 CCR1 SMM Bit Definitions 2-7
2–3 CCR2 SMM Bit Definitions 2-8
2–4 CCR3 Bit Definitions 2-9
2–5 SMAR Size Field 2-10
2–6 RSDC Instruction 2-12
2–7 RSLDT Instruction 2-12
2–8 RSM Instruction 2-12
2–9 RSTS Instruction 2-12
2–10 SMINT Instruction 2-13
2–11 SVDC Instruction 2-13
2–12 SVLDT Instruction 2-13
2–13 SVTS Instruction 2-13
3–1 SMM Memory Space Header 3-7
3–2 I/O Trap Information in the Save Space 3-8
3–3 Stack Contents When EMM386 GPF Handler Executes HLT 3-20
3–4 Registers Restored by the RSM Instruction 3-28
4–1 Differences in TI and Intel SM Mode Implementation 4-2

Chapter 1

Overview

This guide provides information to help you write software that takes advantage of the TI486DX2 System Management (SM) mode. The information here supplements that of the *TI486DX2 Microprocessor Reference Guide*, and is most useful when used in conjunction with that reference guide.

Topic **Page**

1.1 Introduction 1-2

1.2 SM Mode Comparison Among Different Manufacturers 1-3

1.3 Overview of a Typical SM Handler 1-4

1.1 Introduction

SM mode is a special operating mode whose main purpose is to provide a highly secure method for controlling power management within a system. Theoretically, SM mode can be transparent to any and all application and operating system software, including protected mode software. In practice, however, issues can arise with SM mode since it is not yet a standard part of the PC architecture.

In this document, *SM mode* means that the processor is currently servicing a system management interrupt (SMI), and *normal mode* means that the processor *is not* currently servicing an SMI. Regardless of whether the processor is in normal or SM mode, it executes code in either real, protected, or virtual-8086 mode.

When the processor is powered-up, its control registers are set so that SM mode cannot be entered. Typically, the basic input/output system (BIOS) reprograms these registers so that SM mode is accessible when one of two special events occur. If the BIOS (or some other software) does not enable SM mode, the processor operates in normal mode.

A transition from normal mode to SM mode occurs when SM mode is enabled, and the processor either detects an SM Interrupt or executes the *SMINT* instruction. A transition from SM mode back to normal mode occurs when the SM handler executes a special instruction specifically for that purpose, the *RSM* instruction.

1.2 SM Mode Comparison Among Different Manufacturers

Certain aspects of SM mode implementation are present in all CPUs that provide an SM operating mode. For example, all SM-capable CPUs define an area of memory that is to be used to save certain information about the state of the processor at the time SM mode was invoked. Other SM-related similarities are:

- A memory area is defined that contains the SM handler routine.
- The CPU begins executing the SM handler in real mode.
- A method of restarting I/O instructions is provided.

In contrast, some of the differences among the various SM-capable CPUs are:

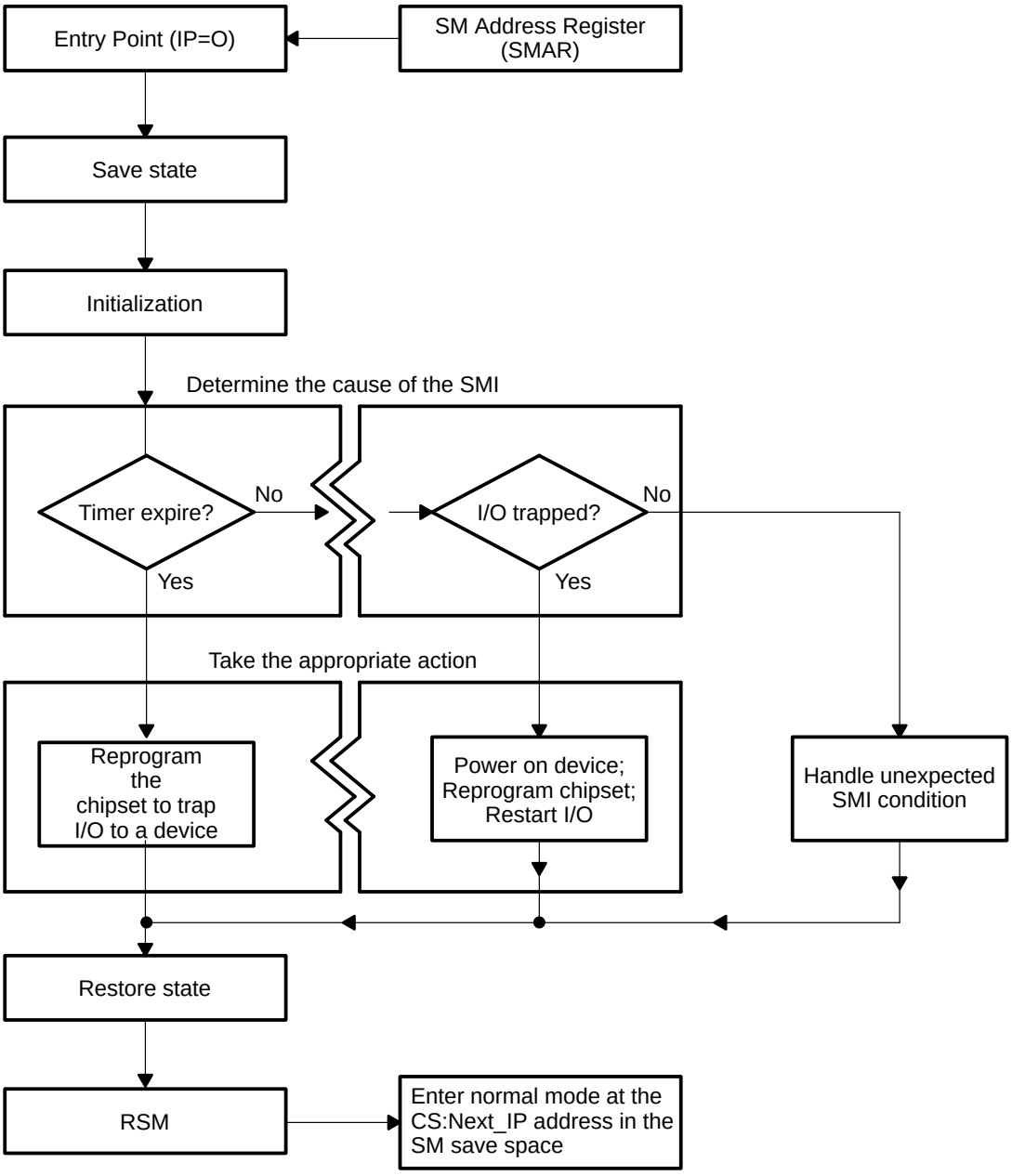
- The particular registers and other information saved upon entry to SM mode
- The location and size of the memory area where the data is saved
- The location and size of the memory where the SM handler resides
- The segment register contents (in particular, the hidden part of the segment register that holds the descriptor for the segment) at the start of the SM handler
- The instructions (opcodes) that are valid
- The time required to enter and exit SM mode

Section 4 of this document discusses many of these differences from the perspective of a programmer who must modify existing SM code written for an Intel™ SL processor to work properly on a TI486DX2.

1.3 Overview of a Typical SM Handler

SM mode is primarily associated with power management, but it can be used in some other capacity. This is because entry into SM mode is driven by hardware (a signal is asserted on the SMI# I/O pin of the TI486DX2) that is at the discretion of the system designer. Also, the functionality of the SM handler is unlimited, so the handler may be written to respond to the SM interrupt in any desired manner. Consequently, there is no *typical* SM handler. When discussing SM handlers designed for power management use, however, use Figure 1–1 as an example of a typical SM handler.

Figure 1–1. A Typical SM Handler for Power Management



When an SMI occurs (or the SMINT instruction is executed), the CPU enters SM mode. Some of the CPU state information is saved in a special area of memory called the SM save space. Then code segment (CS) and extended instruction pointer (EIP) are modified so that execution is passed to the SM handler. EIP is set to zero, while CS is set to the address programmed into the SM Address Register (SMAR).

One of the most important parts of the SM handler resides at its beginning — the save state. Here the handler must save all registers that can potentially be modified (with a few exceptions) before the handler terminates. Some special instructions available in SM mode can save the entire contents of a segment register, including both the programmer-visible 16 bits and the hidden 64 bits of descriptor information (Section 2.4, SM Instruction Summary on page 2-11, explains all of the special SM instructions in detail). These instructions, along with several MOV instructions, are typically used in the save state.

After the pertinent registers are saved, the handler can initialize the SM environment as desired. For example, it can create its own stack, set up an IDT so that interrupts can be serviced while in SM mode, or build the appropriate data structures and enter protected mode. Again, there are some special SM instructions that may be useful in initializing the environment for the SM handler.

In the typical routine, once the environment is set up, you can determine the cause (or source) of the SM Interrupt. The possible causes of the interrupt depend on your specific hardware setup and needs. One cause might be that a pre-programmed timer has expired, indicating that some device has not been used for a time and can be powered down. Another cause might be a trapped I/O cycle that is destined for a powered off device. Whatever the cause of the SMI, the next step in the handler is to respond appropriately.

As an example, suppose a timer expires that indicates the hard disk has been unused for some period of time. The SM handler can reprogram the chipset to disable the timer and to generate an SMI if any I/O activity is directed to the hard disk. Then the handler can power down the disk and resume normal mode. Later, if an SMI is generated because an I/O to the hard disk was trapped, the SM handler can:

1) Power up the drive
2) Reset and re-enable the timer
3) Reprogram the chipset so that hard disk I/O no longer causes an SMI
4) Resume normal mode, re-executing the I/O instruction that was originally trapped

After the SMI is properly serviced, the handler must restore the CPU to the same state that existed when the handler started. The code needed to do this is the inverse of the code in the save state. Once again, some special SM instructions can be used to restore both the visible and hidden parts of the segment registers.

Once the CPU is restored, the RSM instruction is used to return to normal mode and to the interrupted program. The CPU restores some of the informa-

tion that was saved when the SMI occurred and then transfers control to the instruction pointed to by the CS and Next IP fields in the SM save space.

Chapter 2

SM Hardware Overview

This chapter contains a basic overview of the TI486DX2 SM mode hardware implementation and a summary of the special software instructions recognized. The hardware overview is divided into three parts:

- The SM-related pins
- Cache coherency
- The internal control registers

Topic **Page**

2.1	**SM-Related I/O Pins**	**2-2**
2.2	**Cache Coherency**	**2-5**
2.3	**Configuration Control Registers**	**2-6**
2.4	**SM Instruction Summary**	**2-11**

2.1 SM-Related I/O Pins

The following four pins on the TI486DX2 are directly related to SM mode or exhibit behavior in SM mode that needs further explanation:

- SMI#
- SMADS#
- RDY#
- A20M#

2.1.1 SMI# Pin

External hardware generates an SMI by driving the SMI# pin low for at least one CLK period (two CLK periods if the pin is asserted asynchronously). When the CPU recognizes SMI# is low and begins to service the SM interrupt, it drives SMI# low, making it an output signal for the duration of the SM handler routine. When the handler routine terminates, the CPU drives SMI# high for one CLK period. After this CLK period, the CPU stops driving the SMI# pin and begins to monitor it again, waiting for external hardware to drive it low again (causing another SMI).

Because the CPU drives the SMI# pin low while servicing an SMI, it does not (and cannot) monitor the SMI# pin for another interrupt during the time the service routine executes. The external hardware that interfaces with the SMI# pin must wait until the CPU drives the SMI# pin high before attempting to generate a second SMI.

If a system is set up to trap I/O cycles for analysis/handling within an SM handler, the external hardware must drive the SMI# pin low at least two CLK cycles before the RDY# signal for the I/O cycle is asserted. This timing assures that the CPU can save information about the I/O cycle so that the SMI handler can restart the I/O instruction if necessary.

2.1.2 SMADS# Pin

The TI486DX2 has two separate address strobes, ADS# and SMADS#. These two strobes behave identically except that ADS# is the strobe used in normal mode, while SMADS# is the strobe used in SM mode. By having a separate address strobe for use in SM mode, the system designer can design and access a completely separate physical memory for SM mode use. If the designer wishes instead to use normal system random access memory (RAM) for the SM memory space, then the ADS# and SMADS# output pins can be logically combined (ORed).

Because of the flexibility provided by the SMADS# pin, you should know how SMADS# is used in the system. For example, if the two address strobes are simply ORed together, then the SM memory space is part of the main memory, and there is really no special access to or protection of that memory. At the other extreme, a separate physical RAM can be tied to the SMADS# pin such that it is only visible to the processor when SMADS# is in use. In this later case, the system has two separate memories that can share the same addresses, and there must be a way of accessing each. You should understand the conditions under which each of these memories is accessed.

Most of the time you do not need to be concerned about the distinction between ADS# and SMADS#. For example, the CPU automatically generates ADS# in normal mode and SMADS# in SM mode so that the correct memory is accessed. However, what if the SM handler needs to access variables in normal memory whose address is coincident with some part of the SM address space? Also, how can a normal mode program access SM memory (to load the SM handler in place, for example)? In these two special cases, the three bits in Configuration Control register 1 (CCR1) generate SMADS# while in normal mode and ADS# in SM mode. (Section 2.3, on page 2-6, discusses the configuration control registers and how they are accessed and modified.) The effect of setting these bits, SMI, SMAC and MMAC, is shown in Table 2–1.

Table 2–1. Effects of SMAC and MMAC on ADS# and SMADS#

Mode	SMI	Address in SM Region?	SMAC	MMAC	Data Strobe	Code Strobe
Normal	0	No active region	x	x	ADS	ADS
	1	No	x	x	ADS	ADS
	1	Yes	1	0	SMADS	SMADS
	1	Yes	0	0	ADS	ADS
SM	1	No	x	x	ADS	ADS
	1	Yes	0	1	ADS	SMADS
	1	Yes	0	0	SMADS	SMADS

The following points concerning Table 2–1 are important:

- SMADS can be generated only when SMI is enabled by setting the SMI bit in CCR1 to 1. If the SMI bit is not set, SMADS is never generated, and it is not possible to enter SM mode.
- In normal mode with the SMI and SMAC bits set to 1, a jump or call to an address that coincides with SM memory begins executing code from SM memory. Both of these bits must be set to load the SM handler into SM memory. Be careful you do not accidentally execute code in the SM memory space during the loading process.
- In SM mode, you cannot execute code from normal memory whose address coincides with any part of the SM memory space. Exercise care to place the SM address space where it does not shadow any normal memory from which you want to execute code during SM mode.

2.1.3 RDY# Pin

The TI486DX2 does not have separate RDY# pins for SM and normal modes. If the TI486DX2 is used with a chipset that has two RDY# outputs, those outputs can be logically ORed and connected to the single RDY# pin of the TI486DX2.

2.1.4 A20M# Pin

The A20M# input to the TI486DX2 is ignored whenever SM memory is accessed (whenever the SMADS# address strobe is used). During SM memory accesses, the processor behaves as if the A20M# input is not asserted (i.e., as if it is tied high). This behavior allows SM memory above 1M byte to be accessed correctly regardless of the state of A20M within the system.

2.2 Cache Coherency

Regardless of whether the TI486DX2 is in normal or SM mode, accesses to the SM memory space are never cached. This behavior eliminates any concern for internal cache coherency related to SM accesses. However, SM memory can be cached by an external cache controller, and the system designer must decide if memory accesses during SMADS# cycles are to be cached. If they are, the cache controller must maintain a distinction between normal memory and SM memory.

If the CPU is in write-back mode during SM mode, all write-back cycles are directed to normal memory using the ADS# address strobe. Since SM memory is never cached, an INVD or WBINVD instruction writes dirty cache data to normal memory (using ADS#) even if the cache addresses overlap SM memory space.

2.3 Configuration Control Registers

This section describes the TI486DX2 configuration control registers. These registers set and modify certain aspects of SM behavior. Additional information on the configuration control registers can be found in *TI486DX2 Microprocessor Reference Guide*.

All of the configuration control register bits are set to 0 when RESET is asserted, but assertion of WM_RST does not modify the settings of these registers.

2.3.1 Access

The configuration control registers are accessed by writing a register index to I/O port 22h and then writing or reading I/O port 23h. Every read or write of port 23h that is meant to read or write one of the configuration control registers must be preceded by writing a valid register index to port 22h. If port 23h is accessed without a proper register index written to port 22h, the port 23h I/O is directed off-chip.

For absolute safety in reading or modifying the configuration control registers, disable all interrupts (including SMI if the handler does any port 22h or 23h I/O accesses). If you do not disable interrupts, the following problem can occur. Suppose a program writes a valid register index to port 22h and is then interrupted. Suppose further that the interrupt service routine accesses port 22h or port 23h. When control returns to the original program, its access to port 23h may be to the wrong configuration register, or it might be directed externally. In that case, the program might read or write incorrect data, or it might not modify the desired configuration register at all.

The SM interrupt can be disabled in one of the configuration control registers, but this itself involves a configuration control register modification for which safety is sought. Accordingly, for safety in accessing configuration control registers the system designer should provide a means for software to disable the SM interrupt by forcing the SMI# pin high. The software can use this method to disable SMI during the short time that configuration registers are read or written, and then re-enable SMI when access to the configuration registers is complete.

2.3.2 Descriptions

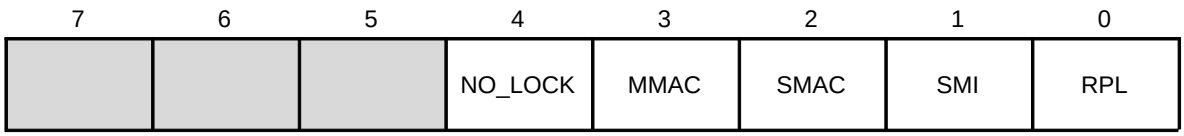
The portions of the configuration control registers that are applicable to SM mode and power management are described in the following subsections.

2.3.2.1 Configuration Control Register 1

The CCR1 register controls system management mode (SMM) features and enables SMM and cache-interface pins. CCR1 is illustrated in Figure 2–1, and the pin functions applicable to SMM are described in Table 2–2.

Figure 2–1. Configuration Control Register 1 (CCR1)

Register Index = C1h



= Reserved

Table 2–2. CCR1 SMM Bit Definitions

Bit Position	Name	Description
1	SMI	Enable SMM pins: If 1, enables SMI# I/O pin and SMADS# output pin. If 0, SMI# input pin ignored and SMADS# output pin floats.
2	SMAC	System management memory access: If 1, accesses to addresses within the SMM memory space cause external bus cycles to be issued when SMADS# output is active. SMI# input is ignored. If 0, access is unaffected.
3	MMAC	Main memory access: If 1, data accesses occurring within an SMI service route (or when SMAC is 1) cause accesses to main memory instead of SMM memory space. If 0, access is unaffected.

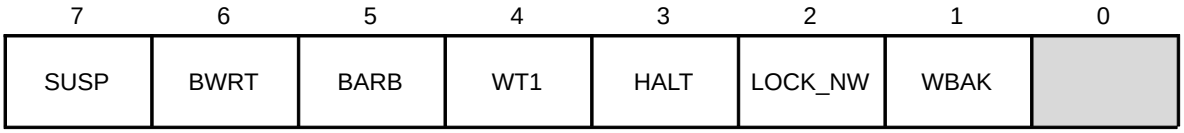
Note: Bits 4–0 are cleared to 0 at reset.

2.3.2.2 Configuration Control Register 2

The CCR2 register sets up internal cache operation and enables suspend control pins. CCR2 is illustrated in Figure 2–2, and the pin functions applicable to SMM are described in Table 2–3.

Figure 2–2. Configuration Control Register 2 (CCR2)

Register Index = C2h



= Reserved

Table 2–3. CCR2 SMM Bit Definitions

Bit Position	Name	Description
3	HALT	Suspend on halt: If 1, the CPU enters suspend mode following execution of a HLT instruction. If 0, suspend mode is not entered.
7	SUSP	Enable suspend pins: If 1, enables SUSP# input and SUSPA# output. If 0, SUSP# input pin is ignored and SUSPA# output pin floats.

Note: All bits are cleared to 0 at reset.

2.3.2.3 Configuration Control Register 3

The CCR3 register controls additional SMM features. CCR3 is illustrated in Figure 2–3, and the pin functions are described in Table 2–4.

Figure 2–3. Configuration Control Register 3 (CCR3)

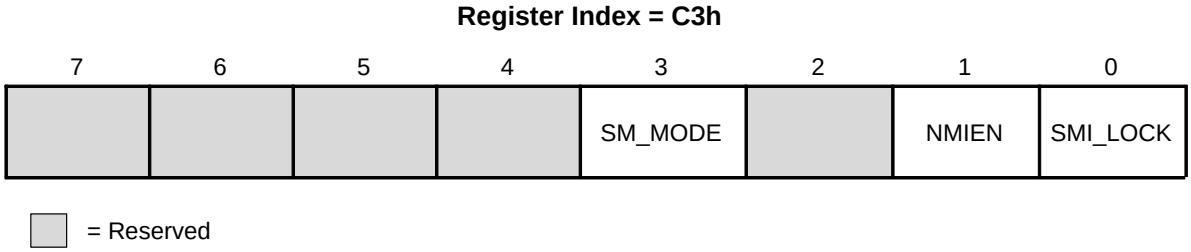


Table 2–4. CCR3 Bit Definitions

Bit Position	Name	Description
0	SMI_LOCK	SMM register lock: If 1, and not operating within a SMI handler, the following SMM control bits cannot be modified: CCR1 bits 1, 2, and 3 CCR3 bits 1 and 3 Any SMAR bit While operating within a SMI handler, these SMM control bits can be modified. Once set, the SMI_LOCK bit can be cleared only by asserting the RESET pin.
1	NMIEN	NMI enable: If 1, enables NMI during SMM. If 0, NMI is ignored during SMM.
3	SM_MODE	SM mode select: If 0, normal SM mode If 1, SL-compatible mode (SMI_LOCK must be 0)

Note: Bits 1, 0, and 3 are cleared to 0 at reset.

2.3.2.4 System Management Address Region Register

The System Management Address register (SMAR), shown in Figure 2–4, defines the location and size of the memory region associated with SMM memory space. The starting address of the SMM address region must be on a block size boundary. For example, a 128K-byte block is allowed to have a starting address of 0K bytes, 128K bytes, 256K bytes, etc. The SMM block size, shown in Table 2–5, must be defined for SMI# to be recognized.

Figure 2–4. SMAR (SMAR) Registers

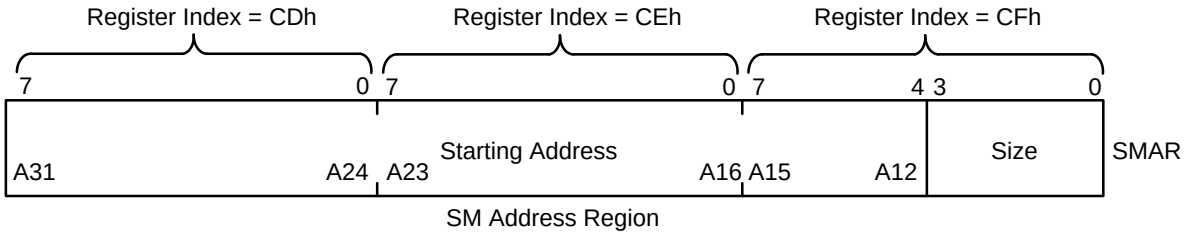


Table 2–5. SMAR Size Field

Bits 3–0	Block Size (bytes)	Bits 3–0	Block Size (bytes)
0h	Disabled	8h	512K
1h	4K	9h	1M
2h	8K	Ah	2M
3h	16K	Bh	4M
4h	32K	Ch	8M
5h	64K	Dh	16M
6h	128K	Eh	32M
7h	256K	Fh	4K (same as 1h)

2.4 SM Instruction Summary

The TI486DX2 recognizes eight instructions that are useful in SM programming situations. These instructions are valid when the following four conditions are true:

- The current privilege level is 0 (CPL = 0)
- SMI is enabled by having CCR1 bit 1 = 1
- The size field of the SMAR register is nonzero
- The processor is in SM mode *or* the SMAC bit = 1 (CCR1 bit 2)

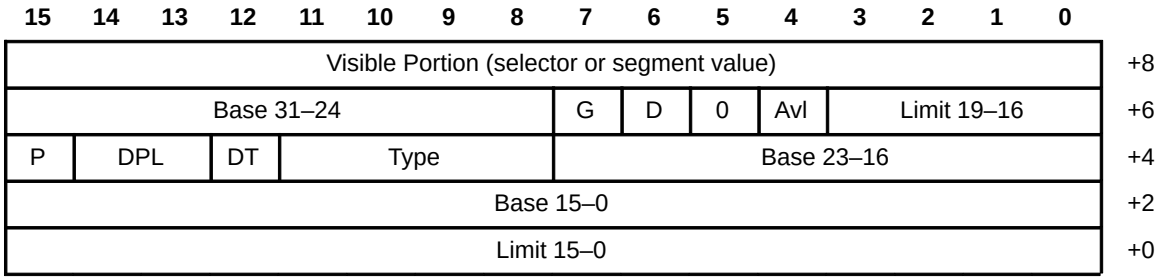
If any of the four conditions is not true and one of the special SM instructions is executed, an undefined opcode fault is generated.

2.4.1 Data Format Used by Many SM Instructions

Most of the SM instructions provide either read or write portions of CPU registers that programmers cannot view or modify. By knowing the data format that SM instructions read and write, you can create and use variables within an SM handler. For convenience, the data format used by SM instructions is referred to as *DESCSEL* (DESCriptor-SELector).

The DESCSEL data format, which is shown in Figure 2–5, is ten bytes in size. The first eight bytes are in the exact format of a descriptor that appears in a global descriptor table (GDT) or local descriptor table (LDT). These eight bytes describe the beginning, size, type, and other required attributes for a segment of memory. The last two bytes of the DESCSEL format are the contents of the visible and modifiable portion of a special register.

Figure 2–5. DESCSEL Data Format



2.4.2 Macros for Implementing SM Instructions

Since the SM instructions are an addition to the x86 instruction set, a set of macros is needed to make their use transparent to the SM programmer. Appendix A contains a set of macros that can be used to simplify SM programming of the TI486DX2. These macros are used in code examples throughout the remainder of this document.

2.4.3 Restore Segment Register and Descriptor (RSDC)

This instruction loads a variable of the DESCSEL format into one of the segment registers, modifying both the visible portion and the hidden descriptor portion. Any segment register except CS can be loaded using this instruction. An attempt to load CS using this instruction generates an invalid opcode fault. Table 2–6 provides the opcode, parameters, and core clock count.

Using this instruction, you can modify the limit field of any of the segment registers except CS without entering protected mode.

Table 2–6. RSDC Instruction

Instruction	Opcode	Parameters	Core Clocks
rsdc	0F 79 [mod sreg3 r/m] [16-bit descsel offset]	descsel, sreg3	10

2.4.4 Restore LDTR and Descriptor (RSLDT)

This instruction loads a variable of the DESCSEL format into the Local Descriptor Table Register (LDTR). Table 2–7 provides the opcode, parameters, and core clock count.

Table 2–7. RSLDT Instruction

Instruction	Opcode	Parameters	Core Clocks
rsldt	0F 7B [mod 000 r/m] [16-bit descsel offset]	descsel	10

2.4.5 Resume Normal Mode Operation (RSM)

This instruction causes the SM handler to terminate. The information in the SM save space (at the top of SM memory) is restored into the CPU. Any registers other than CS and EIP that the SM handler modified must be restored before this instruction executes. Table 2–8 provides the opcode, parameters, and core clock count.

Table 2–8. RSM Instruction

Instruction	Opcode	Parameters	Core Clocks
rsm	0F AA	None	76

2.4.6 Restore TR and Descriptor (RSTS)

This instruction loads a variable of the DESCSEL format into the Task register (TR). Table 2–9 provides the opcode, parameters, and core clock count.

Table 2–9. RSTS Instruction

Instruction	Opcode	Parameters	Core Clocks
rsts	0F 7D [mod 000 r/m] [16-bit descsel offset]	descsel	10

2.4.7 Software-Generated SM Interrupt (SMINT)

This instruction causes the SM handler to begin execution. The CPU operates as if the SMI# pin was sampled in the active (low) state, except that it does not drive the SMI# pin as it normally does for a hardware-generated SMI. Note that this is the only SM-related instruction that functions outside of SM mode, and only when the SMAC bit in CCR1 is set to 1. If the SMAC bit is not set, the SMINT instruction generates an invalid opcode fault. Table 2–10 provides the opcode, parameters, and core clock count.

Table 2–10. SMINT Instruction

Instruction	Opcode	Parameters	Core Clocks
smint	0F 7E	None	24

2.4.8 Save Segment Register and Descriptor (SVDC)

This instruction writes the contents of both the visible and hidden portions of a segment register to a variable of DESCSEL format. This instruction is typically used at the start of an SM handler to save the segment registers before any modification by the handler. The handler can execute the RSDC instruction before exiting to restore saved information. Table 2–11 provides the opcode, parameters, and core clock count.

Table 2–11. SVDC Instruction

Instruction	Opcode	Parameters	Core Clocks
svdc	0F 78 [mod sreg3 r/m] [16-bit descsel offset]	descsel, sreg3	18

2.4.9 Save LDTR and Descriptor (SVLDT)

This instruction writes the contents of both the visible and hidden portions of the Local Descriptor Table (LDT) register to a variable of DESCSEL format. Table 2–12 provides the opcode, parameters, and core clock count.

Table 2–12. SVLDT Instruction

Instruction	Opcode	Parameters	Core Clocks
svldt	0F 7A [mod 000 r/m] [16-bit descsel offset]	descsel	18

2.4.10 Save TR and Descriptor (SVTS)

This instruction writes the contents of both the visible and hidden portions of the Task register (TR) to a variable of DESCSEL format. Table 2–13 provides the opcode, parameters, and core clock count.

Table 2–13. SVTS Instruction

Instruction	Opcode	Parameters	Core Clocks
svts	0F 7C [mod 000 r/m] [16-bit descsel offset]	descsel	18

Chapter 3

Software Considerations

This chapter discusses SM software development for the TI486DX2 processor. The applicable topics are determined primarily by the complexity of your SM handler.

Topic **Page**

3.1 Determining the Addresses Used for SM Memory 3-2
3.2 Enabling SM Mode 3-4
3.3 SM Handler Entry Conditions 3-5
3.4 SM Save Space 3-6
3.5 Maintaining the CPU State 3-9
3.6 Initializing the SM Environment 3-14
3.7 Accessing Main Memory Coincident with SM Memory 3-15
3.8 I/O Restart 3-16
3.9 I/O Shadowing and Emulation 3-18
3.10 The HLT Instruction 3-19
3.11 Exiting the SM Handler 3-28
3.12 Debugging SM Code 3-29
3.13 Suspend Mode 3-30
3.14 SM Mode Select 3-31

3.1 Determining the Addresses Used for SM Memory

One of the first decisions you must make is what addresses to use for SM memory. These addresses are determined by the starting location (or base address) and the size of the SM memory.

The base address and size of SM memory are related by the restriction that the base address must be a multiple of the size. For example, if the SM memory size is 16K bytes, the SM base can be at address 0, 16K, 32K, 48K, 64K, and so forth.

3.1.1 Determining SM Memory Size

In general, you can locate SM memory at any address within the processor's 4G byte address space, and the SM size can be from 4K bytes to 32M bytes in various increments (see Table 2–5 on page 2-10). Practically, the size of your SM handler determines the minimum size of the SM memory space.

The TI486DX2 CPU sets the limit field of the CS descriptor to 64K bytes upon entry into SM mode. This new feature allows your SM handler to service interrupts, for example, without entering protected mode first and ignoring CS limit violations by the interrupt service routines that may get called. Unfortunately, this feature also introduces the possibility, albeit unlikely, of a CS limit violation within the SM handler itself. This error occurs only if you define your SM memory to be larger than 64K bytes, and you then use CS to execute code or reference data that is more than 64K bytes from the beginning of SM memory. There is currently no practical reason to have more than 64K bytes of SM code, so this potential problem should be of little consequence. If you need an SM memory space of more than 64K bytes, direct the SM handler to enter protected mode either to access the extra code/data directly or to reload the CS descriptor with a larger limit field.

3.1.2 Determining SM Base Address

If your SM handler is self-contained and does not call or jump to code in any other segment, you can place the SM memory at any base address. There are no issues to consider in this case. If your SM handler is not self-contained, other factors can affect your choice of base addresses.

Determining the base address is important when a separate memory is used for SM mode and is accessed via SMADS#. In that case, while in SM mode you can never execute code residing at normal memory addresses coincident with SM memory. For example, suppose your handler expected to execute some code located in the read-only memory (ROM) BIOS. You would not want any part of the SM memory space to overlap the memory space of the ROM BIOS because the handler would not be able to execute ROM BIOS code residing in the overlapping address space.

Determining the base address is also important because the processor executes the SM handler in real mode (unless the handler is written to enable protected mode). This can lead to trouble if the SM base address is at or above

1M byte and the handler makes a far call to another segment or is interrupted by an IRQ or NMI. In that case, the visible portion of the CS register that is pushed onto the stack does not have the correct value, and the eventual return from the call or interrupt is to the wrong address.

The reason the CS register can be incorrect is that only bits 19–12 of the SMAR register are placed into the high-order bits of CS upon entry into SM mode (the low-order 8 bits are set to zero). That is, the visible portion of CS is loaded with the linear segment address of the SM base, but when the base address is too large, the information in the high-order bits cannot fit into CS and is lost.

In summary, do not place the SM base address above the 1M byte boundary if the handler can make far calls or service interrupts unless the handler first sets up and enters protected mode.

3.2 Enabling SM Mode

Follow these steps to enable SM mode on the TI486DX2:

1) Set the SMAR register so that it defines the desired SM memory location and size.

2) Set the SMI and SMAC bits.

3) Point DS:SI to the beginning of the SM handler source (in ROM).

4) Point ES:DI to the beginning of SM memory.

5) Use REP MOVS to copy the SM handler into SM memory.

6) Clear the SMAC bit.

In step 2, the SMI bit is not set to enable the SMI# pin, although it does perform this function. Instead, setting the SMI bit allows SMADS# to be generated for the destination addresses of the REP MOVS instruction. SMAC must also be set to allow SMADS# to be generated and to disable the SMI# pin. Together, the SMI and SMAC bits allow SMADS# to be generated and to prohibit any SMI from being recognized during the time the handler is copied to SM memory.

After the handler is in place, clearing SMAC (in step 6) causes the SMI# pin to be enabled (since the SMI bit is still set from step 2). At this point, SM interrupts can occur and be serviced properly by the SM handler.

3.3 SM Handler Entry Conditions

When the SM handler begins execution, the following conditions are true:

- The CPU is in real mode, but code is fetched from the segment described by the base address in the CS descriptor, not from the segment in the visible portion of CS. (As soon as any value is placed into CS, the descriptor base field ceases to be used and the visible part of CS is used as it normally is in real mode. The CPU state at SM entry with respect to code fetches is exactly the same as the state when exiting protected mode—after the protected mode enable (PE) bit of CR0 has been cleared, but before CS has been reloaded by jumping to real mode code.)
- The most significant bit (MSB) of the visible part of the CS register contains bits A19–A12 of the SM starting address (as contained in the SMAR register), while the least significant bit (LSB) contains 0. If the SM starting address is below the 1M byte boundary, the value in the visible part of the CS register is equal to the paragraph address where SM memory begins. If the SM starting address is at or above the 1M byte boundary, the visible part of CS has no valid use.
- The limit field of the CS descriptor is set to 64K bytes.
- The EIP register is set to 0.
- The EFLAGS, CR0 and DR7 registers are set to their reset values (so interrupts are disabled).
- The segment registers (except CS) and the general-purpose registers hold whatever information they had during the interrupted program's execution.
- Certain information about the CPU's state at the time of the SM interrupt has been saved at the top of the SM memory space (in the SM save space).

3.4 SM Save Space

Some CPU state information is saved at the top of SM memory before execution flow passes to the SM handler routine. This portion of the SM memory is often referred to as the *SM save space*. The CPU saves the absolute minimum of information in the save space so that SM entry and exit speed is as fast as possible.

The information written to the save space is shown in Figure 3–1 and Table 3–1. As Figure 3–1 shows, of the typically used CPU registers, only CS, EIP, EFLAGS, and CR0 are saved automatically. Other registers that the SM handler can modify must be saved first at the start of the handler. This process can be done using SVDC, SVLDT, SVTS, and MOV instructions and is described in detail in Section 3.5.

Figure 3–1. SMM Memory Space Header

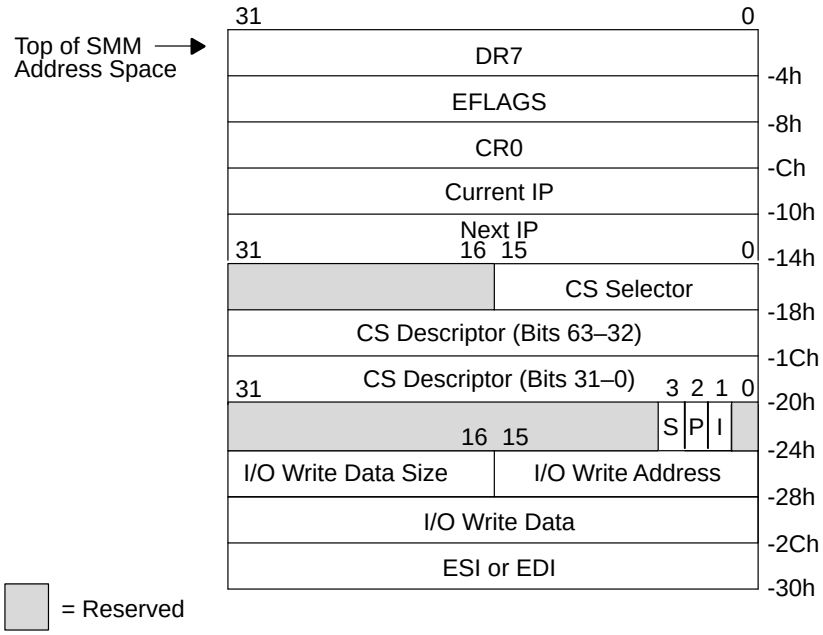


Table 3–1. SMM Memory Space Header

Name	Description	Size
DR7	Contents of Debug register 7	4 bytes
EFLAGS	Contents of the extended flag register	4 bytes
CR0	Contents of Control register 0	4 bytes
Current IP	Address of the instruction executed before servicing the SM interrupt	4 bytes
Next IP	Address of the next instruction that is executed after exiting the SM mode	4 bytes
N/A	Reserved	2 bytes
CS Selector	Code Segment register selector for the current code segment	2 bytes
CS Descriptor	Code register descriptor for the current code segment	8 bytes
N/A	Reserved	28 bits
S	Software SMM Entry Indicator: S is 1 if current SMM is the result of an SMINT instruction. S is 0 if current SMM is not the result of an SMINT instruction.	1 bit
P	REP INSx/OUTSx† Indicator: P is 1 if current instruction has a REP prefix. P is 0 if current instruction does not have REP prefix.	1 bit
I	IN, INSx, OUT, or OUTSx Indicator: I is 1 if current instruction performed is an I/O WRITE. I is 0 if current instruction performed is an I/O READ.	1 bit
N/A	Reserved	1 bit
I/O Write Data Size	Indicates size of data for the trapped I/O write: 01h = byte 03h = word 0Fh = DWord	2 bytes
I/O Write Address	Address of the trapped I/O write	2 bytes
I/O Write Data	Data associated with the trapped I/O write	4 bytes
ESI or EDI	Restored ESI or EDI value. Used when it is necessary to repeat a REP OUTSx or REP INSx instruction when one of the I/O cycles caused an SMI# trap.	4 bytes

† INSx = INS, INSB, INSW, or INSD instruction. OUTSx = OUTS, OUTSB, OUTSW, or OUTSD instruction.

3.4.1 Current IP and Next IP

The current IP and next IP fields in the save space can be quite useful within an SM handler. The current IP normally contains the offset of the instruction that executed before or caused the SM interrupt. The next IP normally holds the offset of the instruction that is to be executed when the SM handler terminates.

There are three cases when these two fields contain the same value:

- When the interrupted instruction has a REP prefix
- When the interrupted instruction is some form of the LOOP instruction
- When the last instruction executed before the SMI was a HLT

In the third case, both fields point to the instruction following the HLT.

A trapped I/O instruction that causes an SMI sometimes needs to be restarted (or re-executed) after the SM handler exits. To restart the instruction, have the handler use the MOV instruction to copy the value in the current IP field to the next IP field. When the handler terminates, EIP points to the I/O instruction that caused the SMI, not the instruction following it.

3.4.2 I/O Trap Information

In the case where a complex I/O operation, a REP OUTSD for example, is the source of an SMI, information in the low-order 4 dwords of the save space is needed to restart the instruction properly.

Table 3–2 describes this I/O trap information and how it can be used by the SM handler. Note that none of the data in Table 3–2 is valid unless the SMI was caused by a trapped I/O cycle. Further, the three I/O write fields are valid only when the I bit is also set.

Table 3–2. I/O Trap Information in the Save Space

Name	Bytes below top of SM memory (in hex)	Bit(s)	Description	Use
P	24	2	REP prefix indicator. If set to 1, the current instruction has a REP prefix; if set to 0, it does not.	When 1, increment ECX before restarting the I/O.
I	24	1	IN/OUT indicator. If set to 1, the current instruction is some form of OUT (an I/O write); if set to 0, it is some form of IN.	When 1, copy the ESI/EDI field to the ESI register before restarting the I/O; when 0, copy ESI/EDI to the EDI register.
I/O write data size	26	15:0	I/O write data size: 01h = byte; 03h = word; 0Fh = dword.	To determine which parts of the I/O write data field are valid.
I/O write address	28	15:0	The target address of a trapped OUT.	As applicable (for emulation or shadowing).
I/O write data	2C	31:0	The data that was written during a trapped OUT.	As applicable (for emulation or shadowing).
ESI/EDI	30	31:0	The value of ESI or EDI at the beginning of a trapped I/O instruction.	Reload ESI or EDI (depending on the I bit) with this value before restarting the I/O.

3.4.3 The S Bit

This bit of information resides in bit 3 of the byte located 24 bytes below the top of SM memory. When set to 1, this bit indicates that the SMINT instruction invoked the SM handler. If this bit is 0, the SM handler was invoked because external hardware drove the SMI# pin low.

3.4.4 CS, CR0, EFLAGS, and DR7 Registers

These registers are saved so that the interrupted program can be restored quickly to its pre-SMI state. However, the information contained in these saved registers can sometimes be useful within the handler. For example, the handler, which runs in real mode, can look at the saved CR0 information to see if the CPU executed in protected mode before it was interrupted. If so, the saved CR0 information also shows whether paging was enabled. The SM handler might also use the saved CS and IP information to examine the code stream that was executing when the interrupt occurred.

3.5 Maintaining the CPU State

As Figure 3–1 shows, relatively few of the CPU registers are saved in the SM save space. The SM handler must save any of the registers that are not automatically saved if it modifies them in any way. Also, it must restore the saved values before executing the RSM instruction.

If your design includes an SMI signaling that the system will be powered off and later powered on, continuing from its previous state, then the entire machine state must be saved to nonvolatile memory. This level of power management requires coupling between the BIOS power-on sequence and the SM handler. The BIOS could detect at power-on that the machine needed to be restored (perhaps by checking a bit in CMOS RAM), load the SM handler, and jump to the proper place in the handler to continue the restoration (or perhaps use the SMINT instruction).

3.5.1 Preserving and Restoring Normal CPU Registers

The CPU's general purpose, index, and pointer registers can be saved and restored using simple MOV instructions. The simplest way to perform these tasks is to reserve some locations within the SM memory space for saving registers, then use a CS: override when moving data to/from the locations. Their use is demonstrated in the following code example.

```
include SM.MAC                          ; See Appendix A

; SM handler begins here.

        ; Save the general purpose, index, and pointer registers.

        mov     cs:int_eax, eax
        mov     cs:int_ebx, ebx
        mov     cs:int_ecx, ecx
        mov     cs:int_edx, edx
        mov     cs:int_esi, esi
        mov     cs:int_edi, edi
        mov     cs:int_ebp, ebp
        mov     cs:int_esp, esp

        ; Other processing which can involve modifying the registers ...

        ; Restore saved register values before exiting.

        mov     eax, cs:int_eax
        mov     ebx, cs:int_ebx
        mov     ecx, cs:int_ecx
        mov     edx, cs:int_edx
        mov     esi, cs:int_esi
        mov     edi, cs:int_edi
        mov     ebp, cs:int_ebp
        mov     esp, cs:int_esp

        ; Exit the SM handler and return to interrupted program.

        rsm                             ; exit the SM handler

; Data area within the code segment for saving registers.

int_eax dd ?
int_ebx dd ?
int_ecx dd ?
int_edx dd ?
int_esi dd ?
int_edi dd ?
int_ebp dd ?
int_esp dd ?
```

3.5.2 Preserving and Restoring Segment Registers

The handler must be able to save and modify both the visible and the hidden parts of the segment registers (that is, the selector/segment and the descriptor) because the SM handler runs in real mode and can interrupt a protected mode program. This capability is not provided in the standard Intel architecture, so the TI486DX2 instruction set has been extended to provide it with two instructions: SVDC and RSDC (these instructions are described in subsections 2.4.8 and 2.4.3).

Here is an example showing how macros that use these two instructions can be used to save and restore segment registers in the SM handler.

```
include SM.MAC                          ; See Appendix A

; SM handler begins here.

        ; Save all segment registers (except CS which is already preserved
        ; in the SM save space) to Code Segment variable locations.

        svdc    int_ds, ds              ; saves <Reg> to cs:int_<Reg>
        svdc    int_es, es
        svdc    int_fs, fs
        svdc    int_gs, gs
        svdc    int_ss, ss

        ; Other processing which can involve modifying of segment registers ...

        ; Restore saved segment register values before exiting.

        rsdc    ds, int_ds              ; moves cs:int_<Reg> into <Reg>
        rsdc    es, int_es
        rsdc    fs, int_fs
        rsdc    gs, int_gs
        rsdc    ss, int_ss

        ; Exit the SM handler and return to interrupted program.

        rsm                             ; exit SM handler

; Data area within the code segment for saving registers.

int_ds  DESCSEL <>                      ; See section 2.4.1 and Appendix A
int_es  DESCSEL <>
int_fs  DESCSEL <>
int_gs  DESCSEL <>
int_ss  DESCSEL <>
```

3.5.3 Preserving Other Special Registers

The remainder of the CPU registers except for the LDTR and the TR can be saved and restored, if needed, using instructions provided in the normal instruction set. Some examples:

```
; Assure EAX has already been saved.
mov     eax, cr2        ; Save CR2
mov     cs:int_CR2, eax
mov     eax, dr6        ; Save DR6
mov     cs:int_DR6, eax
mov     eax, tr6        ; Save TR6
mov     cs:int_TR6, eax
db      66h                     ; So that SGDT saves all
sgdt    cs:int_GDTR             ; Save GDTR
```

The normal instruction set also contains instructions to save the LDTR (SLDT) and the TR (STR), but these instructions are valid only when the processor is operating in protected mode. Once again, the TI486DX2 instruction set has been enhanced to allow LDTR and TR to be saved and restored while in real mode within the SM handler. The four added instructions are SVLDT, SVTS, RSLDT, and RSTS. Their use is demonstrated in the following code example.

```
include SM.MAC                          ; See Appendix A

; SM handler begins here.

        ; Save LDTR and TR to Code Segment variable locations.

        svldt   int_ldt                 ; save LDTR to cs:int_ldt
        svts    int_tr                  ; save TR to cs:int_tr

        ; Other processing which can involve modifying LDTR and TR ...

        ; Restore saved LDTR and TR values before exiting.

        rsldt   int_ldt                 ; restore LDTR from cs:int_ldt
        rsts    int_tr                  ; restore TR from cs:int_tr

        ; Exit the SM handler and return to interrupted program.

        rsm                             ; exit SM handler

; Data area within the code segment for saving registers.

int_ldt DESCSEL <>                      ; See section 2.4.1 and Appendix A
int_tr  DESCSEL <>
```

3.5.4 Preserving the State of the Floating Point Unit

If the floating point unit (FPU) is to be powered off or if the SM handler executes FPU instructions, then the state of the FPU must be maintained for the benefit of the interrupted program. This is accomplished using the FNSAVE and FRSTOR instructions.

You must use the FNSAVE instruction instead of the FSAVE instruction to save the FPU state. This is because the FSAVE instruction causes the FPU to check for any existing error conditions before storing the state. If there is an unmasked exception pending, FSAVE waits for the exception to be serviced; however, this exception should be serviced by the interrupted program and not the SM handler. When you use FNSAVE, any pending exception is not serviced, and the interrupted program is able to do the servicing after the FPU is restored and the SM handler terminates.

Finally, you must perform the FPU save and restore in 32-bit protected mode. The following code example demonstrates the correct procedure.

```
        ; Save FPU state to Code Segment variable location.

        cli
        mov     eax, cr0                ; Enter protected mode.
        or      eax, 1
        mov     cr0, eax
        jmp     $+2                     ; Short jump to clear prefetch queue
        db      66h
        fnsave  cs:int_FPU
        and     eax, not 1              ; Leave protected mode.
        mov     cr0, eax
        sti

        ; Other processing which can use the FPU ...

        ; Restore FPU state before exiting.

        cli
        mov     eax, cr0                ; Enter protected mode.
        or      eax, 1
        mov     cr0, eax
        jmp     $+2                     ; Short jump to clear prefetch queue.
        db      66h
        frstor  cs:int_FPU              ; Restore previously saved state.
        and     eax, not 1              ; Leave protected mode.
        mov     cr0, eax
        sti

        ; Exit the SM handler and return to interrupted program.

        exit_sm                         ; exit_sm macro uses RSM instruction

; Data area within the code segment for saving registers.

int_FPU db 108 dup (?)
```

3.6 Initializing the SM Environment

Once all the registers the SM handler can modify have been saved, the environment for the handler's operation can be set up.

Typically, an SM handler needs to save and initialize the segment registers. Initialization of the segment registers is especially necessary when the interrupted program is operating in protected mode. In this case, the values in the limit fields of the segment register descriptors are unknown, and use of the segment registers could cause segment limit violations. Thus, placing known values into the segment register descriptors is necessary to create a robust SM handler.

Recall that the SM handler executes in real mode, where modifying the segment register descriptors is not normally possible. However, the RSDC instruction (see subsection 2.4.3) is available for use in SM mode, which lets you load any desired descriptors into the segment registers (except for CS). For example, you can load descriptors that have 4G byte limit fields into the segment registers so the handler never causes a segment limit violation. Of course, you may want to have segment limit violations reported because they can highlight errors in your software. In that case, you can load descriptors that exactly describe the segments you are using.

The following code example shows DS and ES being loaded with 4G byte limits and FS and GS being loaded with 64K byte limits.

```
; Assumes the current segment register values and descriptors have already
; been saved (using the svdc macro).

        rsdc    ds, D4G
        rsdc    es, D4G
        rsdc    fs, D64K
        rsdc    gs, D64K

        ; Other processing which eventually restores the CPU and RSMs ...

; Data area within the code segment for loading segment registers.

D4G     DESCSEL {0FFFFh,0,0,92h,8Fh,0,{0}}; See section 2.4.1 and Appendix A
D64K    DESCSEL {0FFFFh,0,0,92h,0,0,{0}}
```

3.7 Accessing Main Memory Coincident with SM Memory

If your SM handler resides in a separate memory using SMADS# for access, use the MMAC bit in CCR1 to access normal memory coincident with SM memory. Typically, you need to access this portion of main memory to save the entire machine state, but you may also have some variables residing there that the SM handler needs to access or modify.

Table 2–1, on page 2-3, shows how setting MMAC affects the data strobe used while the CPU is in SM mode. The following example demonstrates the use of MMAC.

```
; Need access to variables in main memory that overlap SM memory addresses.
; Set the MMAC bit to enable.

        mov     al, 0C1h                ; Read CCR1
        out     22h, al
        in      al, 23h
        or      al, 008h                ; Set MMAC bit
        mov     ah, al
        mov     al, 0C1h
        out     22h, al
        mov     al, ah
        out     23h, al                 ; Write CCR1 with MMAC set

        ; Any data accesses will now use ADS#

; Access no longer needed--disable it by clearing MMAC.

        mov     al, 0C1h                ; Read CCR1
        out     22h, al
        in      al, 23h
        and     al, not 008h            ; Clear MMAC bit
        mov     ah, al
        mov     al, 0C1h
        out     22h, al
        mov     al, ah
        out     23h, al                 ; Write CCR1 with MMAC cleared

        ; Any data accesses to SM addresses will now use SMADS#
```

3.8 I/O Restart

One of the primary power management functions that an SM handler can perform is to power off unused devices and to power on those peripherals when the system later tries to access them. Typically, the chipset is programmed to trap I/O accesses to these devices and to generate an SMI# when any I/O to them is detected. The SM handler then interrogates the chipset to determine the cause of the SMI. When the handler determines the SMI was generated due to I/O access to a powered-down peripheral, it powers the device on and returns control to the interrupted program. However, it must do so in a manner that allows the original I/O instruction to be re-executed. This type of action is know as an I/O restart. Typically the chipset needs to be reprogrammed before restarting so that the re-executed I/O is routed to the peripheral and does not cause another SMI.

The TI486DX2 places information in the SM save space that makes I/O restart a simple task. This information, whose use and location in the save space are shown in Table 3–2, is:

- The I bit, which is set when the trapped I/O is an OUT and cleared when it is an IN
- The P bit, which is set when the trapped I/O was prefixed with a REP and cleared otherwise
- A byte that indicates the size of the trapped I/O write (when the I bit is set)
- A word that indicates the target address of a trapped I/O write (when the I bit is set)
- A dword that contains the data that was to be written (when the I bit is set)
- A dword that contains ESI (when the I bit is set) or EDI (when the I bit is clear)

The following example shows how I/O instructions are restarted.

```
; Interrogation of the chipset has shown an I/O to a powered off device was
; trapped. The device has now been powered on, and the chipset reprogrammed
; so that the I/O will no longer be trapped. All that is left to do is to
; enable the I/O instruction to be re-executed by the interrupted program.

        mov     eax, cs:curr_ip         ; Copy current IP in header to next
        mov     cs:next_ip, eax         ;   IP in header.  Re-executes the I/O
                                        ;   after the RSM.
        mov     ecx, int_ecx            ; Restore ECX to its interrupted value
        bt      cs:smibits, 2           ; Move P bit (REP indicator) into carry
        adc     ecx, 0                  ; If P was set, then we add 1 to ECX

        bt      cs:smibits, 1           ; Copy I bit (IN/OUT indicator) to carry
        jc      out_trap                ; If set, then an OUT was trapped.

        mov     edi, cs:esi_edi         ; For IN, it holds EDI
        mov     esi, cs:int_esi
        jmp     trap_end
```

```
out_trap:
        mov     esi, cs:esi_edi         ; For OUT, it holds ESI
        mov     edi, cs:int_edi

trap_end:
        ; Restore EAX and any other necessary registers except for ECX, ESI,
        ; and EDI which were just restored above ...
        ...

        rsm                             ; Exit SM mode and re-execute the I/O

; Data area within the code segment for saving registers.

int_eax dd ?
int_ebx dd ?
...

ORG SMSIZE–30h                          ; SMSIZE is the size in bytes of SM mem.

; Variables in the SM save area––placed here by the CPU upon SM mode entry and
; used to restore the CPU state upon normal mode re-entry.

esi_edi dd ?                            ; ESI or EDI
iotrapd dd ?                            ; I/O write data
iotrapa dw ?                            ; I/O write address
iotraps dw ?                            ; I/O write size indicator
smibits dd ?                            ; S, P, and I bits
csdescl dd ?                            ; CS descriptor (low word)
csdesch dd ?                            ; CS descriptor (high word)
csselec dw ?                            ; CS selector/paragraph
smirsvd dw ?                            ; Reserved
next_ip dd ?                            ; Next user instruction to execute
curr_ip dd ?                            ; Interrupted user instruction
savecr0 dd ?                            ; Saved CR0 value
saveflg dd ?                            ; Saved EFLAGS
savedr7 dd ?                            ; Saved DR7

(end of example)
```

3.9 I/O Shadowing and Emulation

The I/O write data, address, and size information saved in the SM save area make the task of shadowing I/O writes simple. All that is required is to program the chipset so that the desired I/O ports are trapped and generate an SMI. The SM handler can then retrieve the port information and a copy of the data written to the port from the I/O write address and I/O write data fields in the SM save area. An SM handler can also emulate a device, if desired, since the I/O address and I/O write data is available in the SM save space.

3.10 The HLT Instruction

Use of the HLT instruction is sometimes desirable in power management code. Unfortunately, the instruction is privileged and cannot be executed by a virtual-8086 task. As the rest of this section describes, the ramifications of HLT's privileged status must be fully understood to design functional power management-based systems. If you are considering using the HLT instruction in your power management code, carefully read the remainder of this section; otherwise, you can skip over it.

3.10.1 HLT and Memory Managers

The privileged status of the HLT instruction is important any time the CPU runs in virtual-8086 mode, because execution of the HLT instruction in this mode causes a general protection fault (GPF). A computer generally runs in virtual-8086 mode when a memory manager (like QEMM-386™ or EMM386 from Microsoft™) has been loaded during the boot process.

Memory managers typically enter protected mode, taking total control of the CPU and running all other programs, including DOS, as virtual 8086 applications. If any one of these other programs, like your power management code, executes a privileged instruction such as HLT, the result is dependent upon the memory manager that was loaded. Some, like EMM386, execute the HLT instruction themselves (on behalf of the other program), while others, like QEMM-386, prevent execution of the HLT and pass control to the instruction following the HLT.

Obviously, your power management strategy does not work if the memory manager refuses to execute the HLT instruction on your behalf. For this reason, *avoid using the HLT instruction in your power management code if possible*. If you must use HLT, be aware that some other software running on the system could be incompatible with your power management software.

3.10.2 EMM386 Problem Overview

When EMM386 detects that you have tried to execute the HLT instruction, it executes the HLT for you. Unfortunately, a problem can occur that eventually causes a fatal GPF and system shutdown. This problem can occur only when the following conditions are met:

- ❑ EMM386 is active.
- ❑ Your power management code includes the HLT instruction.
- ❑ Interrupts are enabled when the power management code's HLT instruction is executed.
- ❑ An SM Interrupt occurs while the CPU is in the halt state.

Texas Instruments is currently working with Microsoft™ to modify EMM386 to avoid this fatal problem. Until the software is modified, avoid the problem as follows:

- ❑ Prohibit use of EMM386 on the system.
- ❑ Rewrite your power management code to eliminate use of the HLT instruction.
- ❑ Modify your SM handler to detect if the interrupted instruction was HLT and, if so, to return control to the HLT instruction and not the following one.
- ❑ Modify your SM handler to modify the EMM386 stack.
- ❑ Patch EMM386 to avoid the problem when interrupts are enabled.

Some of these alternatives may not be feasible for any given power management strategy. Of those that are, weigh the pros and cons carefully before making a decision. The first two options are self-explanatory. The last three, however, require more discussion to help you make an evaluation.

3.10.3 EMM386 Problem Discussion

This section describes the nature of the problem with EMM386, the HLT instruction, and SM mode.

When EMM386 is active and your power management code attempts to execute a HLT instruction, the CPU generates a general protection fault (GPF). This happens because your code is running as a virtual-8086 task, which cannot execute HLT.

EMM386 installs its own interrupt handlers when loaded into memory, including the one that deals with the GPF case. So, when your code executes HLT, the EMM386 GPF handler gets invoked. This handler eventually determines that the cause of the GPF was your attempt to execute HLT, and it executes the HLT on your behalf. However, before executing HLT, the handler checks to see if interrupts were enabled while your code was running. If they were, the handler pops three 32-bit registers off the stack; if they were not, the three registers are left on the stack. Thus, the stack contents at the time EMM386's GPF handler executes HLT is dependent on the interrupt status at the time your code attempted to execute HLT (see Table 3–2).

Table 3–3. Stack Contents When EMM386 GPF Handler Executes HLT

Interrupts Were Enabled	Interrupts Were Disabled
EBP	ESI
<GPF Stack Frame>	EBX
	EBP
	EBP
	<GPF Stack Frame>

The instructions in the EMM386 GPF handler that follow the HLT instruction (i.e. the ones that get executed if the halt state is exited due to an SM Interrupt) unconditionally pop the three 32-bit registers (ESI, EBX, and EBP) from the stack. If interrupts were enabled when your code executed the HLT instruction, then these three registers are no longer on the stack. They were popped before the handler executed the HLT instruction. This action corrupts the stack, and eventually another (this time fatal) GPF is generated and reported by EMM386.

If interrupts are disabled when your code attempts to execute HLT, the handler does not pop the three registers before it executes HLT on your behalf. When your SM handler exits, the GPF handler pops the three registers and continues properly.

The problem occurs only when an SMI causes the CPU to exit from the HLT state. That is, if interrupts are enabled and a timer tick interrupt, for example, wakens the CPU from the halt state, no fatal GPF is generated.

Without intimate knowledge of the EMM386 source code, no explanations of the actions described here can be provided. However, the following discussion explains three ways to avoid the EMM386 problem.

3.10.4 EMM386 Problem Workaround: SM Handler Returns to HLT

The EMM386 problem does not occur when an interrupt other than an SMI wakens the CPU from the halt state. You can modify your SM handler to avoid the problem as long as your power management strategy does not require that an SMI get the CPU out of the halt state.

This workaround functions by preventing the EMM386 GPF handler from executing the instructions following the HLT (the unconditional POP instructions) after the SM handler completes. The workaround requires the SM handler to determine if the instruction that executed before the SMI occurred was HLT. If it was, then the next IP field in the SM save space is decremented so that the EMM386 HLT instruction is re-executed when the SM handler terminates. The sequence of events is:

1) Your code attempts to execute HLT.

2) EMM386's GPF handler executes HLT for you (CPU enters halt state).

3) SMI occurs (CPU exits halt state and enters SM mode).

4) SM handler runs.

5) SM handler determines that the EMM GPF handler's HLT was the instruction prior to the SMI.

6) SM handler decrements the Next IP field so it points to the EMM386 GPF handler's HLT.

7) SM handler executes RSM (CPU exits SM mode).

8) EMM386's GPF handler re-executes HLT (CPU re-enters halt state)

9) Some interrupt other than an SMI occurs (CPU exits the halt state).

10) The interrupt in the last step is serviced.

11) Upon completion of the service routine, the instruction following your code's attempted HLT instruction executes.

Steps 3–8 above may never occur or may repeat many times before step 9 is reached.

The downside effects of this workaround are:

- ❏ You cannot use SMI# as a signal to exit the halt state.
- ❏ You must add code, and therefore execution time, to your SM handler.
- ❏ The code you must add is not fool proof†.

On the positive side, this workaround requires no changes to EMM386.

The following code attempts to detect whether the instruction before the SMI was a HLT that might have been generated by EMM386 and modifies the Next IP field if it was:

```
; ============================================================================
; Determine if the instruction prior to the SMI could have been an EMM386 HLT.
; ============================================================================

        rsdc    fs, D4G                 ; Load FS with a 4Gbyte limit descriptor
                                        ; and base of 0 (so it can access any
                                        ; linear address).

; Quick check: Current and Next IP fields are equal if prior instruction was
; HLT or LOOP or had a REP prefix. If they aren't equal, then wasn't HLT.

        mov     eax, cs:curr_ip
        cmp     eax, cs:next_ip
        jne     normal_smi_exit         ; Wasn't HLT.  Go to normal exit point.

; Quick check 2: If the program that was interrupted by the SMI wasn't running in
; protected mode, it could not be EMM386.

        mov     eax, cs:savcr0
        and     al, 1                   ; PE bit set?
        je      normal_smi_exit         ; No.  EMM386 was not interrupted.

; Place into EDX the linear address of the byte of code prior to the one that
; will be executed after the handler executes RSM.  See if that byte is F4h (the
; opcode for HLT).

        mov     eax, cs:csdesch         ; Get high 16 bits of CS base field
        rol     eax, 8
        exch    al,ah
        rol     eax, 16
        mov     ax, word ptr csdescl+2  ; Get low 16 bits of CS base field
        add     eax, cs:next_ip         ; EAX == Linear addr of next instruction
        dec     eax
        mov     edx, eax                ; Save linear addr of possible HLT instr.
```

† The code attempts to determine if the instruction before the SMI was a HLT. To do this, the code checks the byte before the one pointed to by the CS:EIP information that is saved in the SM save space. If that byte is F4h, the opcode for HLT, the handler assumes the prior instruction was in fact HLT. However, it is possible the F4h is actually the last byte of a multi-byte LOOP or REP-prefixed instruction and not a HLT instruction. The F4h could not, however, be part of any other instruction sequence, since the current IP and next IP fields are equal, and that guarantees the prior instruction was a HLT, a LOOP, or a REP-prefixed.

(continued from preceding page)

```
; Set MMAC in case the possible HLT instruction or the page tables are at
; addresses coincident with SM memory

        mov     al, 0C1h
        out     22h, al
        in      al, 23h
        mov     cl, al                  ; Save CCR1 value in CL
        or      al, 008h                ; Set MMAC
        mov     ah, al
        mov     al, 0C1h
        out     22h, al
        mov     al, ah
        out     23h, al

; See if paging was on in the interrupted program.  If so, convert the linear
; address into the proper physical address.

        bt      cs:savecr0, 31          ; Put PG bit into carry.
        jnc     paging_off              ; EDX == Physical address

        mov     eax, cr3                ; Compute pysical address from linear
        and     eax, 0FFFFF000h
        mov     ebx, edx                ; Linear address
        shr     ebx, 22                 ; Shift to directory offset field
        mov     eax, fs:[eax+ebx*4]

        and     eax, 0FFFFF000h
        mov     ebx, edx                ; Linear address
        shr     ebx, 12                 ; Shift to table offset field
        and     ebx, 03FFh              ; Zero out directory offset
        mov     eax, fs:[eax+ebx*4]

        and     eax, 0FFFFF000h
        and     edx, 0FFFh              ; Zero out directory and table offsets
        add     edx, eax                ; EDX == Physical address

paging_off:

        mov     bl, fs:[edx]            ; BL == possible HLT opcode

; Reset MMAC.

        mov     al, 0C1h
        out     22h, al
        mov     al, cl                  ; Saved CCR1 value
        out     23h, al

        cmp     bl, 0F4h                ; HLT opcode
        jne     normal_smi_exit         ; Wasn't HLT.  Go to normal exit point.
        dec     cs:next_ip              ; Re-execute HLT after RSM

normal_smi_exit:

; Restore all CPU registers the handler has used.
; <code to restore the registers is not shown>

        rsm                             ; Exit the SM handler
```

```
(continued from preceding page)

; Data area within the code segment for saving registers.

int_eax dd ?
int_ebx dd ?
int_ecx dd ?
...

; Descriptor and visible segment value for a 4G byte data segment.

D4G     DESCSEL {0FFFFh,0,0,92h,8Fh,{0,0}}; See section 2.4.1 and Appendix A

ORG SMSIZE–30h                          ; SMSIZE is the size in bytes of SM mem.

; Variables in the SM save area––placed here by the CPU upon SM mode entry and
; used to restore the CPU state upon normal mode re-entry.

esi_edi dd ?                            ; ESI or EDI
iotrapd dd ?                            ; I/O write data
iotrapa dw ?                            ; I/O write address
iotraps dw ?                            ; I/O write size indicator
smibits dd ?                            ; S, P, and I bits
csdescl dd ?                            ; CS descriptor (low word)
csdesch dd ?                            ; CS descriptor (high word)
csselec dw ?                            ; CS selector/paragraph
smirsvd dw ?                            ; Reserved
next_ip dd ?                            ; Next user instruction to execute
curr_ip dd ?                            ; Interrupted user instruction
savecr0 dd ?                            ; Saved CR0 value
saveflg dd ?                            ; Saved EFLAGS
savedr7 dd ?                            ; Saved DR7

(End of example)
```

3.10.5 EMM386 Workaround: SM Handler Modifies EMM386 Stack

This workaround functions by having the SM handler place the data onto the EMM386 stack that EMM386 incorrectly pops off after the SM handler completes. That is, when the handler completes, this workaround makes the stack look the same as if interrupts had been disabled at the time HLT was executed (see subsection 3.10.3 on page 3-20). The fatal GPF does not occur when interrupts are disabled, so they do not occur when this workaround is implemented.

The sequence of events for this workaround is:

1) Your code attempts to execute HLT.

2) EMM386's GPF handler executes HLT for you (CPU enters halt state).

3) SMI occurs (CPU exits halt state and enters SM mode).

4) SM handler runs.

5) SM handler determines that the EMM GPF handler's HLT was the instruction before the SMI.

6) SM handler places EBP, EBX, and ESI on the stack.

7) SM handler executes RSM (CPU exits SM mode).

8) EMM386's GPF handler pops the registers and returns from the GPF.

9) The instruction following your code's attempted HLT instruction executes.

The downside effects of this workaround are:

- ❏ You must add code, and therefore execution time, to your SM handler.
- ❏ The code you must add is not fool proof (same as before).
- ❏ A failure occurs if interrupts are ever disabled when your code executes HLT.
- ❏ If EMM386 is modified to correct the original problem, use of that version- causes this workaround to fail.

The advantages of this workaround are:

- ❏ Your power management strategy can use an SMI to exit the halt state.
- ❏ EMM386 needs no changes.

The following code contains the workaround.

```
; ============================================================================
; Determine if the instruction prior to the SMI could have been an EMM386 HLT.
; ============================================================================

; <Code here is exactly the same as the example in section 3.10.4 (approximately
; 90 lines worth) ...>

; Reset MMAC.

        mov     al, 0C1h
        out     22h, al
        mov     al, cl                  ; Saved CCR1 value
        out     23h, al

        mov     al, bl                  ; Move possible HLT opcode into AL

; Restore all CPU registers the handler has used EXCEPT EAX!
; <code to restore the registers is not shown>

        cmp     al, 0F4h                ; HLT opcode
        jne     normal_smi_exit         ; Wasn't HLT.  Go to normal exit point.

; It seems that the instruction prior to the SMI was a HLT executed by EMM386.
; Push 3 registers that EMM386 is going to (incorrectly) pop when it regains
; control after the RSM instruction.

        push    ebp
        push    ebx
        push    esi
        normal_smi_exit:
        mov     eax, cs:int_eax         ; Restore EAX
        rsm                             ; Exit the SM handler

; Data area within the code segment for saving registers.

; <The data section is identical to the one in the section 3.10.4 example>

(End of example)
```

3.10.6 EMM386 Workaround: Patching EMM386

This workaround functions by patching EMM386 so that it does not pop the three 32-bit registers from the stack after it regains control from the SM handler.

The downside effects of this workaround are:

- ❏ You must distribute a patched version of EMM386 (or the means by which your customers can patch it).
- ❏ The patch is not approved by Microsoft.
- ❏ The patch causes a failure if interrupts are ever disabled when your code executes HLT.

The advantages of this workaround are:

- ❏ Your power management strategy can use an SMI to exit the halt state.
- ❏ You do not have to add any code to your SM handler (so it runs faster).

Since the patch may depend on the specific version of EMM386, only a general description of it is included here. This patch simply modifies the target of a relative jump instruction that EMM386 executes after the SM handler returns control. The EMM386 code involved is:

```
SMI_return_target:

        pop     esi                     ; 2 bytes
        pop     ebx                     ; 2 bytes
        pop     ebp                     ; 2 bytes

Patch_return_target:

        add     sp,4
        iretd

; Lots of other code is in here ...

        pop     ds
        pop     esi
        pop     ebx
        pop     ebp
        add     sp,4
        push    ebp
        sti
        hlt                             ; The EMM386 HLT!
        cli
        jmp     SMI_return_target       ; 16-bit offset
```

The modification that needs to occur is the JMP destination: This needs to change from SMI_return_target to Patch_return_target. That way, the three registers are no longer incorrectly popped from the stack when the SM handler terminates.This patch appears to have no effect on interrupts other than an SMI, because the CLI and JMP instructions following the HLT are not executed after a normal interrupt takes the CPU out of the halt state.

To make the change, locate the JMP and add six to the current 16-bit relative offset. This causes the JMP target to move six bytes forward in memory from its previous location, so that the three 32-bit pops are no longer executed after the SM handler completes. For example, if the JMP opcodes are:

```
E9 03 FF
```

Then they would need to change to:

```
E9 09 FF
```

3.11 Exiting the SM Handler

In SM mode, execution of the RSM instruction causes the CPU to return to normal mode. Some of the information in the SM save space is restored into CPU registers as shown in Table 3–4.

Table 3–4. Registers Restored by the RSM Instruction

	Restored From	
Register Name	**Code Variable Name**	**At Offset SMISIZE**
CR0	savecr0	0Ch
DR7	savedr7	04h
EFLAGS	saveflg	08h
EIP	next_ip	14h
CS hidden (descriptor)	csdescl:csdesch	20h
CS visible (selector/paragraph)	csselec	18h

You can control which instruction of the interrupted program executes when the handler terminates by modifying the value in the next_ip field in the save space (as some of the examples have shown). This is the only field in the SM save space that you should ever attempt to modify.

RSM restores few of the CPU registers. Be careful to restore any of the registers the handler has modified before executing the RSM instruction. If you do not, the handler corrupts the interrupted program's registers and the CPU's resulting behavior is unpredictable.

3.12 Debugging SM Code

You can debug your SM handler using standard DOS debuggers as long as the debugger runs in real mode, the SM handler is loaded below the 1M byte boundary, and the SM handler remains in real mode.

You must place an INT 3 instruction in your SM handler to pass control to the debugger. Typically, you start the debugger, then trigger an SMI while the SM handler containing the INT 3 is loaded into the programmed SM memory. Be sure to have the handler save the complete CPU state before loading the INT 3, and then restore the complete CPU state before the RSM. Also, ensure that the segment registers contain 64K byte (or larger) limit fields before executing the INT 3 instruction.

If you use a debugger to set local or global breakpoints via the debug registers, the CPU cannot be interrupted once it enters SM mode. The reason is that DR7 is saved in the SM save space and then reloaded with its reset value. All of the local and global breakpoint enable bits are cleared. After an INT 3 is executed within the SM handler to pass control to the debugger, the debug registers perform as expected.

3.13 Suspend Mode

The TI486DX2 can enter a power-saving suspend mode using software and hardware. In the suspend mode with the clock still operating, the CPU current usage is reduced to approximately 40% of normal. If the input clock is stopped while the CPU is in suspend mode, current usage drops to about 1% of normal.

On power-up, the CPU is programmed so that suspend mode cannot be entered. The SUSP bit (bit 7) of CCR2 must be set to 1 to enable suspend mode. Once it has been set, the CPU enters suspend mode when the SUSP# input is asserted The CPU remains in suspend mode until SUSP# is deasserted. When the CPU enters suspend mode, it asserts a suspend acknowledge signal, SUSPA#. You can stop the input clock to save even more power at any time after the processor asserts SUSPA#. You can resume from the stopped clock state by restarting the clock and then negating SUSP#.

The HLT instruction can also place the TI486DX2 into suspend mode if the HALT bit (bit 3) of CCR2 is set (it is cleared at power-up). The CPU asserts the SUSPA# signal upon entry to suspend mode via HLT. As with the assertion of SUSP#, you can also stop the input clock.

Due to the virtual-8086 mode and software-related problems discussed throughout Section 3.10, *The HLT Instruction*, avoid using the HLT instruction if possible. Your system is more reliable if you use the hardware method, SUSP#, to enter suspend mode.

3.14 SM Mode Select

Bit 3 of CCR3, the SM_MODE bit, selects an SM-compatible hardware mode. When this bit is set, the SMI# and SMADS# pins behave like the SMI# and SMACT# pins of an SL-enhanced CPU (see subsection 2.8.7 of the *TI486DX2 Microprocessor Reference Guide*, on page 2–57). If you intend to set the SM_MODE bit, the chipset you use should support the SL hardware protocol for the SMI# and SMIACT# pins.

Additionally, you may have to reprogram the chipset when you set the SM_MODE bit. If the chipset contains a register that the BIOS programs with information about the type of CPU in use, this register may need to be modified to a value that the chipset interprets as an SL-protocol in the system. This reprogramming is necessary when the chipset uses the BIOS-programmed CPU information that determines only one hardware protocol per CPU type value for SMI (the SIS496/497 is an example of a chipset that operates in this manner). Failure to reprogram such a chipset can cause the system to hang according to the following steps:

- ❑ BIOS detects TI486DX2 and programs chipset accordingly.(See *Processor Initialization* in the *TI486DX2 Microprocessor Reference Guide*, on page 2–2.)
- ❑ SM handler is loaded.
- ❑ SM_MODE bit is set.
- ❑ SMI# is sampled active.
- ❑ SMADS# is asserted and remains so until RSM is executed.
- ❑ Chipset expects SMI protocol as if SM_MODE were not set, so the (continuously) asserted SMADS# signal is interpreted as an ever-starting memory cycle into the SM address space.
- ❑ System hangs.

Setting the SM_MODE bit affects SM software in two ways:

1) The SMINT instruction is not recognized and an undefined opcode fault is generated.
2) The SMAC and MMAC bits do not affect memory accesses. In systems where the SM handler needs to access normal memory that is shadowed by SM memory, the designer must provide some software method for the access.

Chapter 4

Converting Intel SM Code to TI486DX2 SM Code

This chapter contains information necessary to convert SM code originally written for the Intel™ SL-enhanced processors to run on the TI486DX2 microprocessor. The SM programming model for the TI486DX2 is different from the Intel SL and DX4 models. This chapter describes the differences between the TI486DX2 and Intel's SL and DX4 CPUs and the actions you must take to achieve the conversion.

Topic **Page**

4.1 Differences in TI and Intel SM Implementations 4-2

4.2 SM Code Conversion 4-3

4.1 Differences in TI and Intel SM Implementations

Table 4–1 compares SM mode on TI and Intel CPUs. Each area is discussed in more detail in the remainder of this section.

Table 4–1. Differences in TI and Intel SM Mode Implementation

Area of Difference	TI Feature/Method	Intel SL Feature/Method	Intel DX4 Feature/Method
SM Memory Location	Programmable	Fixed (30000h or 38000h)	Programmable
SM Memory Size	Programmable	Fixed (64K bytes or 32K bytes)	At the discretion of the system designer
Handler Entry Point	programmed_segment:0	3000:8000	SMBASE:3000
Access to SM Memory	Programmable via SMAC bit	Programmable via bit 3 of OMDCR; no access provided.	No access method provided. The system designer must provide a method to handle this task.
Access to main Memory at SM Memory Addresses	Programmable via MMAC bit	Access via EMS capability	No method provided. The system designer must provide a method to handle this task.
Register Save Area	Top of programmed location	3FFA:0008 – 3FFA:005F	Top of programmed location
Registers Saved	DR7, EFLAGS, CR0, current IP, next IP, and CS	CR0, CR3, EFLAGS, extended instruction pointer (EIP), DR6, DR7, TR, LDTR, and all general purpose and segment registers	CR0, CR3, EFLAGS, extended instruction pointer (EIP), DR6, DR7, TR, LDTR, and all general purpose and segment registers
Processor Mode Within the SM Handler	Real mode with CS limit of 64K and other segment limits of unspecified sizes	Real mode with 4G byte segment limits	Real mode with 4G byte segment limits
Instruction Restart	Copy current IP to next IP	Subtract system management last instruction length register contents from old EIP	Various methods
NMI Servicing	Disabled by default, but can be serviced	Cannot be serviced	Disabled by default, but can be serviced by executing an IRET within the SM handler.
Back-to-back Multiple SMIs	Not possible	Possible	Possible
A20M# Input	Ignored during SM mode	Not an issue due to fixed SM memory location below 1M byte.	Must be handled by the system desginer if SM memory is relocated above 1M byte.

4.2 SM Code Conversion

The SM code differences described in Table 4–1 require that you implement conversions to provide comparable or other desired functionality. The following subsections provide some specific actions you can take to achieve your goals.

4.2.1 SM Memory Location and Size

The first major difference in the SM implementations is the address space of the SM memory. In the Intel SL architecture, the SM address space is required to be a 32K-byte space at 38000h or a 64K-byte space at 30000h. The Intel DX4 CPU allows the SM memory space to be relocated by setting the SMBASE relocation slot, but this can be done only when servicing an SMI. This means that the SM memory location is also fixed on the Intel DX4 for at least the first occurrence of an SMI. You can program SMAR on the TI486DX2 to define an SM region that resides at any boundary that is a multiple of 4K bytes (4K, 8K, 16K, and so forth) in memory. This SM region can have various sizes ranging from 4K bytes to 32M bytes depending on the starting address (see Table 2–5 on page 2-10).

You must program the TI486DX2 to reflect your desired SM memory space because it has no default SM memory space defined. If you want to mimic the space used by the Intel SL and DX4 CPUs, you can program SMAR as follows:

32K at 38000: SMAR = 000384h

64K at 30000: SMAR = 000305h

You must add the code for programming the SM memory space to the BIOS.

4.2.2 SM Handler Entry Point

In the Intel SL architecture, the SM handler always starts at 3000:8000 (CS = 3000 and IP = 8000), while the Intel DX4 starts at offset 8000 (IP= 8000) from the contents of the SMBASE register. In the TI architecture, the SM handler starts at offset 0 of the SM address space (IP = 0). This difference is important if you have or write SM handler code that references locations that are relative to the current code segment.

Determine if the difference in IP addresses affects your SM code and make any changes that are necessary. This might be as simple as changing an *ORG 8000* to *ORG 0*, or it may be more complex.

4.2.3 Access to SM Memory

Before an SMI can be serviced, the SM handler must be copied from the ROM into the SM memory space. Access to the SM space (when not in SM mode) is controlled on both TI and Intel SL processors by modifying certain register settings, but the registers involved are different. On the TI486DX2, you must first define the SM memory space via SMAR and then modify certain bits of CCR1. The required steps are detailed in section 3.2, *Enabling SM Mode* on

page 3-4. On the Intel SL CPU, you must set bit 3 of the OMDCR register. On the Intel DX4 CPU, no method is provided to access SM memory in normal mode. The system designer must provide a method of access, and that method will vary depending on the design.

Modify the software that loads your SMI handler to use the method provided by the TI486DX2 (the SMAC bit). This software is typically located in the BIOS.

4.2.4 Access to Main Memory at SM Memory Addresses

TI and Intel provide different methods to access main memory that is coincidentally located with SM memory space. On the TI processor, set the MMAC bit of CCR1 to perform noncode-segment prefixed data accesses to main memory that resides at the same address as SM memory. If your SM code must access main memory that is coincidentally located with SM memory space, modify any references in your handler that access main memory residing with SM memory to use the MMAC bit (see section 3.7, *Accessing Main Memory Coincident with SM Memory* on page 3-15).

In the code written for the Intel SL processor, the EMS functions access noncode-segment prefixed data access The code written for an Intel DX4 design is unpredictable because in an Intel DX4 design this responsibility is also given to the system designer and can vary.

4.2.5 Register Save Area

Both the TI and Intel processors save processor state information at the top of the SM memory space. In the Intel processor, the location of the save space is fixed since the location of the SM memory is fixed. In the TI processor, the location of the save space is determined by the location and size of the SM memory.

The uppermost 12 double words (dwords) (48 bytes) of the SM memory are used for the register save area. So, for example, if you define SM memory to be an 8K byte area that begins at A0000, A1FD0 to A1FFF becomes the register save area.

4.2.6 Registers Saved

The TI and Intel processors differ greatly in the information that is placed into the register save area when an SMI begins. The Intel processor saves all CPU registers, so you need not be concerned with modifying registers in the SM handler.

To accelerate entry into the SM handler, the TI CPU saves a minimal amount of information in the register save area. In exchange for quicker access to the SM handler, the TI processor places the responsibility for saving register information upon you, the SM programmer. The TI CPU recognizes several opcodes (described in section 2.4, *SMM Instruction Summary* on page 2-11) you can use to save and restore register information. Use these opcodes to save and restore the segment registers your handler uses. In addition, if your

handler enters protected mode, opcodes are provided to save and restore the Task State Register and Local Descriptor Table Register. These opcodes save and restore both the visible and hidden portions of the segment, TS, and LDT registers.

Figure 3–1 shows the exact structure of the register save area. Note that none of the general purpose registers and none of the segment registers (except CS) are saved. You must modify the code at the beginning of your handler to save any of the segment and general purpose registers used by the handler. Section 3.5, *Maintaining the CPU State* on page 3-9, contains many code examples that demonstrate saving and restoring various CPU registers.

4.2.7 Processor Mode Within the SM Handler

Although both the TI and Intel CPUs are in real mode when the SM handler begins execution, they may be in different states. The Intel CPUs can place other, known values into the segment registers before passing control to the SM handler because the CPUs save all the segment registers in the save area. The TI CPUs do not save segment registers in the save area, and therefore do not modify the contents of any of the segment registers except CS.

This means the values in all of the segment registers except CS are generally useless to the handler until they are saved and modified. In addition, the CS register has a limit of 64K bytes.

If the TI486DX2 is operating in protected mode when an SMI occurs, the limits of the segment registers are those of the interrupted program, which may be larger or smaller than 64K bytes. An error occurs if the segment limits are smaller than 64K bytes and you try to access information above that limit. In that case, a segment limit violation occurs, even though the SM handler is executing in real mode. For this reason, you must load known values into any segment registers the handler uses before you attempt to use the registers.

One way you can place known values into the segment registers is to have the handler contain a GDT that it uses to enter protected mode. Once the CPU is in protected mode, the handler can load the appropriate selectors into the segment registers. Normally, this is your only choice, because the hidden portion of a segment register (its descriptor) is modified only when the segment register is loaded while the CPU is in protected mode.

On the TI486DX2, you can use one of the special opcodes, RSDC (Restore Segment register and Descriptor), to modify the hidden part of the descriptor, even though the SM routine is executing in real mode. Subsection NO TAG, on page NO TAG, describes the RSDC instruction.

Here is a short example of handler code that sets up DS and ES so they can be used to access any linear address.

```
include SM.MAC                          ; See Appendix A

; SM handler begins here.

smi_start:

        mov     cs:int_eax, eax         ; Save EAX .. EDX
        mov     cs:int_ebx, ebx
        mov     cs:int_ecx, ecx
        mov     cs:int_edx, edx
        svdc    int_ds, ds              ; Save DS and ES
        svdc    int_es, es

        rsdc    ds, D4G                 ; Make DS and ES able to access any
        rsdc    es, D4G                 ;  linear address.

        mov ebx,00400000h               ; Read dword at 4Mb boundary into
        mov eax,[ebx]                   ; EAX.

        ...

        rsdc    ds, int_ds              ; Restore DS and ES to entry values
        rsdc    es, int_es
        mov     eax, cs:int_eax         ; Restore EAX .. EDX
        mov     ebx, cs:int_ebx
        mov     ecx, cs:int_ecx
        mov     edx, cs:int_edx

        exit_sm                         ; Leave SM mode

; Data area within the code segment for saving registers.

int_eax dd      ?
int_ebx dd      ?
int_ecx dd      ?
int_edx dd      ?
int_ds  DESCSEL <>                      ; See Section 2.4.1 and Appendix A
int_es  DESCSEL <>

;       Lim = FFFFF, Base = 0, Gran = 4K, Type=R/W Data
D4G     DESCSEL {0FFFFh, 0, 0, 92h, 8Fh, 0, {0}}
```

If your handler uses any of the segment registers other than CS, if it makes any assumptions about any segment register contents, or if it enables interrupts, you must modify the code to:

1) Save the current segment register values (except for CS) using SVDC and RSDC

2) Load the segment registers with proper values, including the descriptor portion

3) Restore the old segment register values (except for CS) before exiting the handler

4.2.8 Instruction Restart

Two of the items the TI486DX2 saves upon entry to SM mode are the offset of the instruction that executed (or was executing) before the start of the SMI (the current IP) and the offset of the instruction to be executed when the SMI handler exits (the next IP).

If you need to restart the instruction that was executing before the SMI, you can copy the contents of the current IP field to the next IP field. This action causes the processor to resume execution at the prior instruction instead of the following one. This is most useful when an I/O trap is the source of an SMI (see section 3.8 on page 3-16).

Modify any parts of your handler that deal with instruction restart to use the current and next IP fields in the save area.

4.2.9 NMI Servicing

When the TI486DX2 enters the SM handler, NMIs are not enabled unless the NMIEN bit (bit 1) of CCR3 is set. The Intel CPU operates the same way. This bit should always be clear during normal mode.

If you want to service NMIs during SM mode, you can set up the proper environment in the handler, then set the NMIEN bit. You should clear it before exiting SM mode. Although the TI486DX2 and the Intel DX4 are similar in their handling of NMIs during SM mode (NMIs can be enabled if desired), they differ on how the NMIs are enabled. On the TI CPU you set NMIEN, but on the Intel DX4 CPU you must set up and execute an IRET.

4.2.10 Multiple SMIs

On Intel CPUs, multiple SMIs can be invoked without ever executing any operating system or application code. That is, while the Intel CPU is in SM mode it can store the fact that another SMI is being requested, and it can re-enter the SM handler immediately after exiting it from the first SMI.

The hardware SM implementation on the TI486DX2 is such that a single operating system or application code instruction can be executed between multiple SM interrupts. That is, the TI CPU does not acknowledge the second SMI until one clock cycle after the SM handler has terminated.

Evaluate your SM strategy to determine if this minor difference in TI SM implementation requires any changes to your SM handler.

4.2.11 A20M# Input

Because the Intel SL CPU requires SM memory to be fixed at a location below the 1M byte boundary, there is no conflict associated with the A20M# input and SM mode. However, both the TI486DX2 and the Intel DX4 CPUs allow the SM memory space to reside above the 1M byte boundary; thus, the A20M# input becomes a problem. The TI CPU handles the problem by ignoring the A20M# input while accessing SM memory. The Intel DX4 CPU makes no provision in this regard; the system designer must provide some method of handling this problem. The solution for any given Intel DX4–based system may require specific code in the SM handler. Such code is not required in a TI 486DX2 SM handler.

4.2.12 SM Mode Select

Bit 3 of CCR3 selects an Intel compatible hardware mode. When this bit is set, the SMI# and SMADS# pins behave like the SMI# and SMIACT# pins on the Intel CPU. (See subsection 2.8.7 of the *TI486DX2 Microprocessor Reference Guide*, on page 2–57). The setting of this bit affects SM software only, in that, the SMAC and MMAC bits do not function. The system designer will have to provide methods for the SM programmer to use for making these special types of memory accesses.

Appendix A

SM Mode Macros

This appendix shows the contents of a file named SM.MAC. It contains macros used in the other code examples within this document. You can use these macros or create others as you desire.

```
COMMENT ^
================================================================================

            Copyright (c) 1994, 1995 Texas Instruments, Incorporated

This file, SM.MAC, defines a set of macros for generating System Management (SM)
mode instruction opcodes, since no assembler directly supports these SM
instructions.

There are six SM instructions that are used to save and restore registers which
are not automatically saved when SM mode is entered, one instruction to enter
SM mode, and one instruction to exit SM mode.  These instructions support many
addressing modes, but the macros in this file only implement one mode--a 16-bit
memory reference (within the code segment as a CS: override is also used).
These macros could be made much more complex to allow other addressing modes,
but the additional complexity wouldn't provide much useful benefit.

Each of the macros that implements a register save or restore takes as a
parameter an offset in the code segment where the register should be saved to
or restored from.  The two macros that save and restore segment registers also
take the name of a segment register as a parameter.

NOTE: The variable "addr" must be type DESCSEL, and it must reside within
      the code segment.

================================================================================
^

svdc    MACRO   addr, reg               ; Save one of the segment registers
        SMMac   svdc, addr, reg, 78h
        ENDM

rsdc    MACRO   reg, addr               ; Restore one of the segment registers
        SMMac   rsdc, addr, reg, 79h
        ENDM

svldt   MACRO   addr                    ; Save the LDT register
        SMMac   svldt, addr, ldt, 7Ah
        ENDM

rsldt   MACRO   addr                    ; Restore the LDT register
        SMMac   rsldt, addr, ldt, 7Bh
        ENDM

svts    MACRO   addr                    ; Save the Task register
        SMMac   svts, addr, ts, 7Ch
        ENDM

rsts    MACRO   addr                    ; Restore the Task register
        SMMac   rsts, addr, ts, 7Dh
        ENDM

smint   MACRO                           ; Software SM Interrupt
        DB      00Fh, 07Eh
        ENDM

rsm     MACRO                           ; Exit from SM mode
        DB      00Fh, 0AAh
        ENDM
```

(continued on the following page)

(continued from the previous page)

```
SMMac   MACRO   mname, addr, reg, op

        ; CS: override and SM instruction opcode
        db      2Eh
        db      0Fh, op

        ; [mod sreg3 r/m] byte
        ifidni          <reg>, <cs>
                db      0Eh
        elseifidni      <reg>, <ds>
                db      1Eh
        elseifidni      <reg>, <fs>
                db      26h
        elseifidni      <reg>, <gs>
                db      2Eh
        elseifidni      <reg>, <ss>
                db      16h
        elseifidni      <reg>, <es>
                db      06h
        elseifidni      <reg>, <ts>
                db      06h
        elseifidni      <reg>, <ldt>
                db      06h
        else
                ECHO ERROR in macro <mname>:
                ECHO .     Register parameter unknown: <reg>
                ECHO .     Register parameter must be either CS, DS, ES, FS, GS,
                ECHO .     SS, TS, or LDT
                .ERR
        endif

        ; 16-bit displacement
        dw      offset addr

ENDM

; ---------------------------------------------------------------------------
; The following structure can be used to reserve space for use with the svdc
; macro, and it can be used to declare variables for use with the rsdc macro.
; The initialized values create a 64Kbyte limit data segment at paragraph address
; 0.  You may want to modify the initialized values.
; ---------------------------------------------------------------------------

DESCSEL STRUCT
        LimitLo         dw      0FFFFh  ; Limit = 64K
        BaseLo          dw      0       ; Linear base addr = 0
        BaseMid         db      0
        DType           db      92h     ; Application, present, data seg
        LimitHi         db      0       ; Limit = 64K
        BaseHi          db      0
        UNION
        Selector        dw      0       ; For use in protected mode
        ParaAddr        dw      0       ; For use in real mode
        ENDS
DESCSEL ENDS

(End of SM.MAC)
```

Appendix B

Glossary

A

assert: To apply a signal. An asserted signal is logically true.

B

BARB: A bit in CCR2 that enables or disables write-back of dirty cache data when a hold state is entered.

base: A field in a GDT or LDT entry that specifies the starting address of a segment

BIOS: *Basic Input Output System*. A set of routines that contain detailed instructions for activating computer and peripheral devices. The BIOS is normally implemented in nonvolatile memory.

bit: The fundamental unit of computer memory. A bit can be a 1 or a 0. A byte is made up of eight bits.

breakpoint: A point in a program at which to stop execution so that machine status may be determined.

byte: A sequence of eight bits. Represents one character of information.

C

cache: A small, high-speed memory that provides a temporary storage location for data most likely to be requested by the CPU. This allows for quick access of data and improved CPU performance (i.e., zero wait states).

cache (data) coherency: A consistent relationship between data in cache memory and data in other memories. Data coherence is necessary when a system has multiple memories. If several memories contain the same data word, modifying that data word in one memory causes the data to be inconsistent with the data stored in the other memories. Therefore, the other memories that have a copy of that same data word must either update or invalidate their copy.

cache flush: A memory operation that maintains cache consistency. In a cache flush, all locations with dirty bits are written to main memory. Then the cache contents are cleared (flushed).

CCR1, CCR2, CCR3: *Configuration Control registers 1, 2, and 3.* Configuration Control register 1 controls SMM features and enables SMM and cache interface pins. Configuration Control register 2 sets up internal cache operation and enables suspend control pins. Configuration Control register 3 controls the SMI lock, NMI enable, and SM mode features.

CPU: *Central Processing Unit.* The execution unit of the microprocessor. The CPU consists of control, shift, adder, multiplier, and limit units and a register file.

D

descriptor: A data structure that defines a segment's base, limit, and attributes.

E

EAX: *Extended* or 32-bit version of register AX. A register used by many mathematical and logical instructions.

EBP: *Extended* or 32-bit version of register BP. A register used as a pointer to the base of stack frames.

EBX: *Extended* or 32-bit version of register BX. A register used for indirect memory references.

ECX: *Extended* or 32-bit version of register CX. A register used as a counter by REP and LOOP instructions.

EDI: *Extended* or 32-bit version of register DI. A register used as a destination offset for string operations.

EDX: *Extended* or 32-bit version of register DX. A register used by many mathematical instructions.

EFLAG: *Extended* or 32-bit version of the *Flag* register. A register that contains status information and controls certain operations on the microprocessor.

EIP: *Extended* or 32-bit version of the *Instruction Pointer* register. A register that contains the offset into the current code segment of the next instruction to be executed.

ES: *Extra Segment* register. A register used as the destination segment of string instructions. Special segment override prefix ES allows the use of this additional Segment register.

ESI: *Extended* or 32-bit version of register SI. A register used as a source offset for string operations.

ESP: *Extended* or 32-bit version of register SP. A register used to locate the top of the stack.

F

far jump: A jump whose destination is in another code segment.

flush: Invalidate the entire contents of cache memory.

FPU: *Floating Point Unit*. A part of the microprocessor that accelerates the computation of floating-point arithmetic.

FS: Additional Data *Segment* register. This Segment register is used when the special segment override prefix FS is present.

G

GDT: *Global Descriptor Table*. Part of the selector mechanism that contains segment descriptors used when the TI bit in the Segment Selector register is set to zero.

GDTR: *Global Descriptor Table register*. A register that holds a 32-bit base address and 16-bit limit for the global-descriptor table.

GS: Additional Data *Segment* register. This Segment register is used when the special segment override prefix GS is present.

I

IDT: *Interrupt Descriptor Table*. An array of up to 256 8-byte interrupt descriptors, each of which points to an interrupt service routine.

IDTR: *Interrupt Descriptor Table register*. A register that holds a 32-bit base address and 16-bit limit for the interrupt-descriptor table.

IF: *Interrupt Flag*. A flag that, when set, enables the CPU to acknowledge and service maskable interrupts (INTR input pin).

INTR: *Interrupt.* A signal generated by external hardware that changes the normal sequential flow of a program by transferring program control to a selected service routine.

L

LDT: *Local Descriptor Table*. Part of the selector mechanism that contains segment descriptors used when the TI bit in the Segment Selector register is set to one.

LDTR: *Local Descriptor Table register*. A register that holds a 16-bit selector for the local-descriptor table.

limit: A field in a GDT or LDT entry that specifies the maximum allowable offset within a segment.

linear address: An address formed by combining the contents of a segment register and another register, the offset. The way these 2 registers are combined depends on the processor's operating mode. When the processor is in real or virtual-8086 mode, the segment register contents are multiplied by 16 and added to the offset. In protected mode, the segment register contents select an entry from the GDT or LDT. The base field of this entry is then added to the offset. See *GDT*, *LDT*.

M

MMAC: *Main Memory Access*. Storing data in or retrieving data from main memory, from within an SM handler.

N

NMI: *NonMaskable Interrupt*. A rising-edge-sensitive input that, when asserted, causes the processor to suspend execution of the current instruction stream and begin execution of an NMI interrupt service routine.

normal mode: The processing mode when the CPU is not handling an SMI.

P

paging: A memory management technique that allows logical addresses within a program (i.e., linear addresses) to access physical memory at (possibly) totally different physical addresses.

physical address: The address driven by the CPU onto the address pins. In real mode and in protected mode with paging disabled, the physical address is identical to the linear address. In protected mode with paging enabled, the physical address is formed by processing the linear address through the paging mechanism.

power management: Software designed to shut down unused parts of the computer to save power.

prefix: Bytes placed in front of an instruction to override segment defaults, change operand, address-size attributes, assert LOCK#, and repeat string instructions.

protected mode: The microprocessor's operating mode when the PE bit of Control register 0 is set. In protected mode, the enhanced memory management capabilities, which include segmentation and paging, are available. Code has one of four privilege levels, with some processor instructions restricted to the most-privileged code.

R

real mode: A processing mode designed to make the microprocessor function similar to an 8086 microprocessor. The TI486DX2 powers up or resets to real mode.

ROM: *Read Only Memory*. A permanent, unchangeable memory used in the PC to accomplish system startup. ROM stores the BIOS programs needed to perform diagnostics and instruct the computer in various operations. When using DOS, the contents of the ROM are placed in reserved memory.

S

SMAR: S*ystem Management Address Region*. Defines the location and size of the memory region associated with SMM memory space.

SMAC: *System Management Memory Access*. Storing data in and retrieving data from SMM memory space while in normal mode. See *SMAR*.

SMI: *System Management Interrupt*. An interrupt that causes the microprocessor to enter the system management mode. The system-management interrupt has a higher priority than any other interrupt, including NMI.

SMM: *System Management Mode*. A power management feature that allows various subsystems of the computer to be powered down when not in use to conserve power. The CPU is in SM mode when it is processing an SMI.

SS: *Stack Segment* register. A register containing segment selectors that index into tables located in memory. These tables hold the base address for each segment and other information related to memory addressing.

SUS: *SUSpend* bit. A bit in Configuration Control register 2 that enables or disables the SUSP# and SUSPA# pins, which control entry into the suspend mode.

T

TR: *Task register*. A register that holds a 16-bit selector for the current task-state segment (TSS) table. The TR is loaded and stored using the LTR and STR instructions, respectively.

TR3 through TR7: *Test registers 3* through *7*.

TSR: *Task State registers*. Registers that are saved and restored using the SVTS and RSTC instructions, respectively.

V

V86: *Virtual 8086*. See *virtual-8086 mode*.

virtual-8086 mode: An operating mode that allows multiple programs written for an 8086 CPU to execute concurrently.

visible: Contents (data, address components, and current states) of registers and stored data that the programmer can access, trap, or retrieve.

W

write back: An approach used to update the main memory. The CPU writes data into the cache and sets a dirty bit indicating that a word has been written into the cache but not into the main memory. The cache data is written back into the main memory later and the dirty bit is cleared. Write-back accesses memory less than a write-through cache, but its cache control logic is more complex.

write-through cache: A type of cache used in updating the main memory. Data is written to the main memory while it is written to cache, or immediately afterwards. The main memory always contains valid data, and blocks in cache can be overwritten without data loss. The hardware implementation remains relatively simple.

Index

A

A20M#
  input 4-7
  pin 2-4
access
  to main memory at SM memory addresses 4-4
  to SM memory 4-3
accessing main memory 3-15
address
  linear
    defined B-3
  physical
    defined B-4
  SM memory (determining) 3-2
ADS#
  effects of SMAC and MMAC on 2-3
assert
  defined B-1
asserted signal B-1

B

BARB
  defined B-1
basic input output system (BIOS) 1-2
BIOS 1-2, 3-10
  defined B-1
bit
  defined B-1
bit definitions
  CCR1 2-7
  CCR2 2-8
  CCR3 2-9
block size SMAR 2-10
block sizes
  address-region registers 2-10
breakpoint
  defined B-1
burst write cycles
  enabling 2-8
byte
  defined B-1

C

cache
  coherency 2-5
    defined B-1
    enabling 2-7
  defined B-1
  flush
    defined B-2
  write-through
    defined B-6
cache coherency
  defined B-1
  enabling 2-8
cache write mode locking 2-8
cache write-back mode enabling 2-8
cache write-through mode enabling 2-8
CCR1
  defined B-2
CCR2
  defined B-2
CCR3
  defined B-2
coherency
  cache 2-5, 2-7, 2-8
  defined B-1
comparison
  SM mode 1-3
configuration control registers 2-6
  access 2-6
  bit definitions 2-7 to 2-12
  defined B-2
  descriptions 2-6
conversion of SM code 4-3

CPU
defined B-2
maintaining state of 3-9
programming 3-30
restoring 1-5
current IP field 3-7

D

data coherency
defined B-1
debugging SM code 3-29
definitions
SM address region block size 2-10
SMM address region block size 2-10
descriptor
defined B-2
DESCSEL data format 2-11
disabling
NMI 2-9
RPL pins 2-7
SL-compatible mode 2-9
suspend pins 2-8
system management memory access 2-7

E

EAX
defined B-2
EBP
defined B-2
EBX
defined B-2
ECX
defined B-2
EDI
defined B-2
EDX
defined B-2
EFLAG
defined B-2
EIP
defined B-2
EMM386 problem
causes 3-19
details 3-20
workaround
detection code 3-22
EMM386 stack modification 3-24
patching EMM386 3-26
return to HLT 3-21
sequence 3-21
workaround sequence 3-21
EMM386 problem workaround 3-24
enabling
burst write cycles 2-8
cache coherency 2-7, 2-8
cache write-back mode 2-8
cache write-through mode 2-8
main memory access during SMM 2-7
NMI 2-9
RPL pins 2-7
SL-compatible mode 2-9
SM mode 3-4
steps required 3-4
SMI# pin 2-7
SMM pins 2-7
suspend mode on halt 2-8
suspend pins 2-8
system management memory access 2-7
entry into SM handler 4-3
ES
defined B-2
ESI
defined B-2
ESP
defined B-3
exiting the SM handler 3-28
extended flag
defined B-2
extended general purpose registers
defined B-3
extended instruction pointer (EIP)
defined B-2
extra segment (ES)
defined B-2

F

far jump
defined B-3
fields
current IP 3-7
next IP 3-7
floating point unit
preserving state of 3-14
floating point unit (FPU)
defined B-3
flush
defined B-3
FPU
defined B-3
preserving state of 3-13
save and restore procedure 3-13
FS
defined B-3

G

GDT
defined B-3
GDTR
defined B-3
global descriptor table (GDT)
defined B-3
global descriptor table register (GDTR)
defined B-3
GS
defined B-3

H

halt 2-8
HLT
and memory managers 3-19
instruction 3-19, 3-30
use of 3-19

I

I/O pins
SM-related 2-2
I/O restart 3-16
simplification of 3-16
I/O shadowing and emulation 3-18
I/O trap information 3-8
in save space 3-8
IDT
defined B-3
IDTR
defined B-3
IF
defined B-3
initializing SM environment 3-14
instructions
HLT 3-19, 3-30
MOV 1-5, 3-10
restart 4-6
RSDC 2-12
RSLDT 2-12
RSM 1-2, 1-5, 2-12
RSTS 2-12
SM 1-5
data format used by 2-11
macros for implementing 2-11
summary 2-11
validity 2-11
SMINT 1-2, 1-5, 2-13, 3-19
SVDC 2-13
SVLDT 2-13
SVTS 2-13
Intel and TI
differences in SM implementations 4-2
interrupt (INTR)
defined B-3
interrupt descriptor table (IDT)
defined B-3
interrupt descriptor table register (IDTR)
defined B-3
interrupt flag (IF)
defined B-3
interrupts
nonmaskable
defined B-4
software-generated SM 2-13
system management
causes 1-5
defined B-5
INTR
defined B-3

L

LDT 2-13
defined B-3
LDTR 2-12
defined B-3
restore 2-12
save 2-13
limit
defined B-3
linear address
defined B-4
local descriptor table (LDT) 2-13
defined B-3
local descriptor table register (LDTR)
defined B-3
LOCK# negate 2-7
locking
cache write mode 2-8
SMM 2-9

M

macros for implementing SM instructions 2-11
main memory
access to at SM memory addresses 4-4
main memory access (MMAC)
defined B-4
during SMM 2-7
maintaining CPU state 3-9
memory
read only
defined B-5
memory managers and HLT 3-19

memory space header
system-management mode 3-7
MMAC
defined B-4
during SMM 2-7
effects of on ADS# and SMADS# 2-3
use of 3-15
modes
normal 1-2
defined B-4
resume 2-12
processor within SM handler 4-5
protected
defined B-4
real
defined B-4
SM 1-2
comparison among manufacturers 1-3
SM comparison 1-3
SM macro file A-2
SM select 3-31
suspend 3-30
system management
defined B-5
virtual-8086 3-19, 3-30
defined B-5
write-back cache 2-8
write-through cache 2-8
MOV
instructions 1-5, 3-10
multiple SMIs 4-7

N

negate LOCK# 2-7
next IP field 3-7
NMI
defined B-4
disabling 2-9
enabling 2-9
servicing 4-7
nonmaskable interrupt (NMI)
defined B-4
servicing 4-7
normal mode 1-2
defined B-4
resume 2-12

O

OMDCR register 4-4
operating modes
normal 1-2
SM 1-2

P

paging
defined B-4
patching EMM386 3-26
physical address
defined B-4
pins
A20M# 2-4
ADS#
effects of SMAC and MMAC on 2-3
RDY# 2-3
SMADS# 2-2
effects of SMAC and MMAC on 2-3
SMI# 2-2
SM-related I/O 2-2
power management
and the SM mode 1-4
controlling 1-2
defined B-4
SM handler for 1-4
prefix
defined B-4
preserving and restoring registers 3-12
normal CPU 3-10
segment 3-11
preserving floating point unit (FPU) 3-13
processor mode within SM handler 4-5
programming CPU 3-30
protected mode
defined B-4

R

RDY# pin 2-3
read only memory (ROM)
defined B-5
real mode
defined B-4
registers
CCR1 bit definitions 2-7
configuration control 2-6, 2-7 to 2-9
access 2-6
descriptions 2-6
extended general purpose
defined B-3
OMDCR 4-4
preserving and restoring 3-12
normal CPU 3-10
segment 3-11
restored by RSM instruction 3-28
save area 4-4
saved 4-4
SM address-region register 2-10
SMAR address region 2-10
SMBASE 4-3

registers (continued)
stack segment
defined B-5
task
defined B-5
task state
defined B-5
restart instruction 4-6
restore
LDTR and descriptor (RSLDT) 2-12
normal mode 2-12
RSLDT 2-12
RSTS 2-12
segment register and descriptor (RSDC) 2-12
TR and descriptor 2-12
resume (RSM) 2-12
return to HLT
EMM386 problem workaround 3-21
ROM
defined B-5
RPL pins
disabling 2-7
enabling 2-7
RSDC 2-12
instruction 2-12
restore 2-12
RSLDT
instruction 2-12
restore 2-12
RSM
instruction 1-2, 1-5, 2-12
resume 2-12
RSM instruction
registers restored 3-28
RSTS
instruction 2-12
restore 2-12

S

save
LDTR and descriptor 2-13
local descriptor table register (LDTR) and descriptor (SVLDT) 2-13
registers 4-4
segment register and descriptor 2-13
SVTS 2-13
TR and descriptor (SVTS) 2-13
save and restore procedure
floating point unit (FPU) 3-13
save area
register 4-4
setting SMM address region block size 2-10
SL-compatible mode
disabling 2-9
enabling 2-9
SM
address register 1-5
address-region register 2-10
handler
overview 1-4
responses to SMI 1-5
instruction
summary 2-11
validity 2-11
instructions 1-5
data format used by 2-11
macros for implementing 2-11
interrupt
software-generated (SMINT) 2-13
interrupt (SMI) 1-5
causes 1-5
responses to 1-5
mode 1-3
comparison among manufacturers 1-3
introduction 1-2
related I/O pins 2-2
SM address register
block size 2-10
SM base address
determining 3-2
SM code
conversion 4-3
debugging 3-29
SM environment
initializing 3-14
SM handler
EMM386 problem workaround 3-24
entry conditions 3-5
entry point 4-3
exiting 3-28
processor mode within 4-5
returns to HLT 3-21
SM Interrupt (SMI) 1-2
SM memory
access to 4-3
addresses
access to main memory 4-4
addresses used for
determining 3-2
location 4-3
size 4-3
SM memory size
determining 3-2

SM mode
comparison
among different manufacturers 1-3
enabling 3-4
steps required 3-4
implementation
differences between TI and Intel SL 4-2
macro file A-2
purpose 1-2
select 3-31
SM save space 3-6
SMAC
defined B-5
effects of
on ADS# and SMADS# 2-3
SMADS# pin 2-2
disabling 2-7
effects of SMAC and MMAC on 2-3
SMAR
defined B-5
see SM address register 1-5
SMBASE register 4-3
SMI
defined B-5
effects on ADS# and SMADS# 2-3
multiple 4-7
SMI# pin 2-2
disabling 2-7
enabling 2-7
SMINT instruction 1-2, 1-5, 2-13, 3-9
SMM
clearing lock 2-9
defined B-5
locking 2-9
pins (enabling) 2-7
SS defined B-5
stack contents
when EMM386 GPF executes HLT 3-20
stack segment (SS)
defined B-5
SUS defined B-5
suspend
enter on halt 2-8, 3-30
suspend bit (SUS)
defined B-5
suspend mode 3-30
suspend pins
disabling 2-8
enabling 2-8
SVDC 2-13
instruction 2-13
SVLDT
instruction 2-13
save 2-13
SVTS
instruction 2-13
save 2-13
system management address region (SMAR)
defined B-5
system management interrupt (SMI)
defined B-5
system management memory access (SMAC)
defined B-5
disabling 2-7
enabling 2-7
system management mode (SMM)
defined B-5
see SM mode
system registers
address-region registers block sizes 2-10
configuration registers
CCR1 bit definitions 2-7
CCR2 bit definitions 2-8
CCR3 bit definitions 2-9
SM address-region register 2-10

T

task register
defined B-5
task state register (TSR)
defined B-5
TI and Intel
SM implementation differences 4-2
TI and Intel SM mode implementation
differences 4-2
TI486DX4 1-1, 1-3, 1-4, 2-1 to 2-6, 2-11
information in SM save to restart I/Os 3-16
TI486DX4 Microprocessor Reference Guide 1-1, 2-6
TR
defined B-5
restore 2-12
save 2-13
TR3
defined B-5
TR7
defined B-5
TSR
defined B-5

V

V86
defined B-5
virtual-8086 mode 3-19, 3-30
defined B-5
visible
defined B-5

W

write back
  cache 2-8
  defined B-6
write mode
  cache locking 2-8
write-through cache
  defined B-6
write-through mode
  cache 2-8

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.