ST486DX SMM

PROGRAMMING MANUAL

1st EDITION

NOVEMBER 1994

GENERAL INDEX

1. SMM OVERVIEW

1.1 Introduction

This Programmer's Guide has been written to aid programmers in the creation of software using the SGS-Thomson System Management Mode (SMM) for ST486DX CPUs. This guide should be used in conjunction with the *SGS-Thomson ST486DX and ST486DX2 Processors Data Book.* SMM programming related to the ST486SLC/e is covered in the ST486SLC/e *SMM Programmer's Guide.*

SMM provides the system designer with another processor operating mode. Within this document the standard x86 operating modes (real, v86 and protected) are referred to as normal mode. Normal mode operation can be interrupted by an SMI interrupt or special instruction that places the processor in System Management Mode (SMM). SMM can be used to enhance the functionality of the system by providing power management, register shadowing, peripheral emulation and other system level functions. SMM can be totally transparent to all application software, including protected mode operating systems.

1.2 SGS-Thomson SMM Features

SMM operation within one of the SGS-Thomson ST486DX microprocessors is similar to related operations performed by other x86 microprocessors. All processors with SMM capability, switch into real mode upon entry into the SMM interrupt handler. Each CPU has a unique SMM code locations. However, the SMM memory region for the SGS-Thomson CPU has a programmable location and size. All devices save some of the CPU registers upon entry to SMM. The SGS-Thomson CPU automatically saves minimal register information reducing the entry and exit clock count to as low as 100 clock cycles. This compares with Intel's clock overhead for a typical entry and exit of 633 clock cycles. The SGS-Thomson SMM implementation provides unique instructions that save additional segment registers as required by the programmer. The x86 MOV instruction can be used to save the general purpose registers.

Although all SMM capable CPUs provide I/O trapping, the SGS-Thomson CPUs simplify I/O type identification and instruction restarting. SGS-Thomson CPUs also make available to the SMM routine information which can simplify peripheral register shadowing.

SGS-Thomson provides a method to prevent SMM configuration registers from being accessed by applications. Access to the SMM configuration can be prevented by setting a bit in the CPU configuration space. Not allowing an application to disable or alter SMM operation is useful for anti-virus or security measures.

1.3 Typical SMM Routines

A typical SMM routine is illustrated in the flowchart shown in Figure 1-1. Upon entry to SMM, the CPU registers that will be used by the SMM routine, must be saved. The SMM environment is initialized by setting up an Interrupt Descriptor Table, initializing segment limits and setting up a stack. If the SMI was a result of an I/O bus cycle, the SMM routine can monitor peripheral activity, shadow read-only ports ,and/or emulate peripherals in software. If a peripheral was powered down, the SMM routine can power up a peripheral and reissue the I/O instruction. If the SMI was not caused by an I/O bus cycle, non-trap SMI functions can be serviced. If the instruction executing, when an SMI occurred, was a HLT instruction, the HLT instruction it should be restarted when the SMM routine is complete. Before normal operation is resumed, any CPU registers modified during the SMM routine must be restored to their previous state.

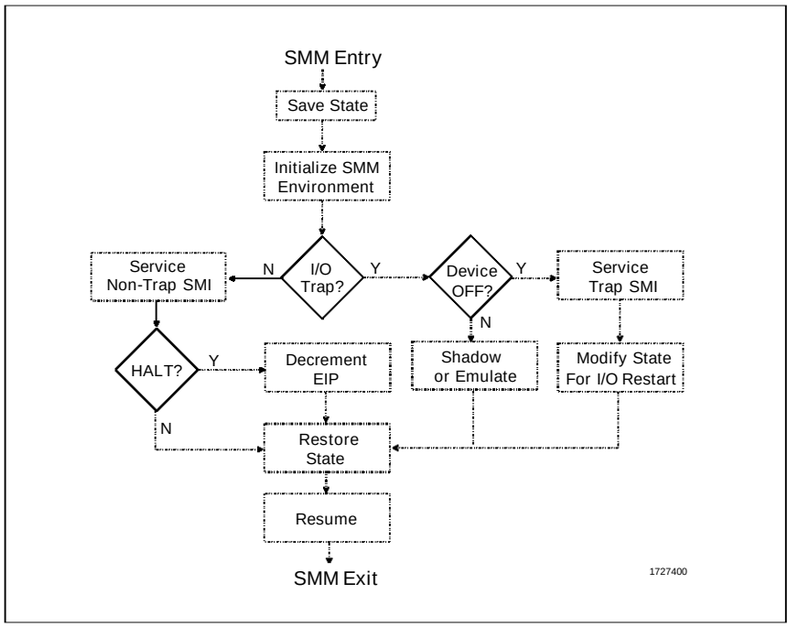

Figure 1 - 1. Typical SMM Routine

2. SGS-Thomson SMM IMPLEMENTATION

2.1 Hardware Background

2.1.1 SMM Pins

The signals at the SMI# and SMADS# pins are used to implement SMM. The SMI# pin is bi-directional. The SMI# pin is used by the chipset to signal the CPU that an SMI has occurred. While the CPU is in the process of servicing an SMI interrupt, the SMI# pin is an output used to signal the chipset that the SMM processing is occurring. The SMADS# address strobe signal is generated instead of an ADS# address strobe signal while executing or accessing data in SMM address space.

2.1.2 SMI# Pin Timing

To enter SMM mode, the SMI# signal must be asserted for at least one CLK period (Two clocks if SMI# is asserted asynchronously). To accomplish I/O trapping, the SMI# signal should be asserted two clocks before the RDY# for that I/O cycle. Once the CPU recognizes the active SMI# input, the CPU drives the SMI input low for the duration of the SMI routine. The SMI routine is terminated with an SMI specific resume instruction (RSM). When the RSM instruction is executed, the CPU drives the SMI pin high for one CLK period. The SMI# pin must be allowed to go high for one CLK at the end of the SMI routine in order for the next SMI to be recognized. Since the SMI# pin is bi-directional, not more than one SMI# interrupt can become active at one time.

2.1.3 Address Strobes

The CPU has two address strobes, ADS# and SMADS#. ADS# is the address strobe used during normal operations. The SMADS# address strobe replaces ADS# during SMM for memory accesses when data is written, read, or fetched in the SMM defined region. Using a separate address strobe increases chipset compatibility and control.

During an SMM interrupt routine, control can be transferred to main memory via a JMP, CALL, Jcc instruction, execution of a software interrupt (INT), or a hardware interrupt (INTR or NMI). Execution in main memory will cause ADS# to be generated for code and data outside of the defined SMM address region. (It is assumed, but not required, that the chipset ultimately translates SMADS# and a particular address to some other address.) To access data in main memory that overlaps the SMM address space, the MMAC bit (CCR1, bit 3) must be set. This allows ADS# strobes to be generated for data accesses in memory which overlap SMM memory while in SMM mode. It is not possible to execute code in main memory that overlaps SMM space while in SMM mode.

SMADS# can also be generated for memory reads/writes and code fetches within the defined SMM region when the SMAC bit (CCR1, bit 2) is set while in normal mode. The generation of SMADS# permits a program in normal mode to jump into SMM code space. The RSM instruction should not be executed after jumping into SMM space unless valid return information is first written into the SMM header.

2.1.4 Chipset RDY#

The SGS Thomson CPU has one RDY# input. Chipsets that implement the dual ready lines (one for SMM and one for normal memory) can logically OR the two ready lines together to produce a single RDY# line.

2.1.5 Cache Coherency

SMM memory is never cached in the CPU internal cache. This makes cache coherency completely transparent to the SMM programmer. If the CPU cache is in write-back mode, all write-back cycles will be directed to normal memory with the use of the ADS# signal. An INVD or WBINVD will write dirty data out to normal memory even if it overlaps with SMM space.

SMM memory can be cached by a external cache controller, but it is up to the cache designer to be sure to maintain a distinction between SMM memory space and normal memory space.

The A20M# input to the CPU is ignored for all SMM space accesses (any accesses which uses SMADS#).

2.2 Configuration Control Registers

This section describes how to use the Configuration Control Registers in SMM code. For a complete description of the Configuration Control Registers, refer to the *SGS-Thomson ST486DX and ST486DX2 Processors Data Book*.

All Configuration Control Register bits are set to 0 when RESET is asserted. Asserting WM_RST does not affect the configuration registers.

These registers are accessed by writing the register index to I/O port 22h. I/O port 23h is used for data transfer. Each data transfer to I/O port 23h must be preceded by an I/O port 22h register index selection, otherwise the port 23h access will be directed off chip. Before accessing these registers, all interrupts, including SMI, must be disabled. A problem could occur if an interrupt occurs after writing to port 22h but before accessing port 23h. The interrupt service routine might access port 22h or 23h. After returning from the interrupt, the access to port 23h would be redirected to another index or possibly off chip. Before accessing the Configuration Control Registers from outside of SMM mode, the chipset generation of SMI# interrupt must be disabled if the CPU SMI# input is enabled.

The portions of the Configuration Control Registers (CCR1, CCR2, and CCR3) which apply to SMM and power management are described in the following pages.

Table 2 - 1 CCR1 Register

Register INDEX = C1h

7	6	5	4	3	2	1	0
			NO-LOCK	**MMAC**	**SMAC**	**SMI**	**RPL**

Reserved

SMI Enable SMM Pins

SMI = 0: SMI# input pin is ignored and SMADS# output pin floats. Execution of SGS Thomson specific SMM instructions will generate an invalid opcode exception.

SMI = 1: SMI# input/output pin and SMADS# output pin are enabled. SMI must be set to 1 before any attempted access to SMM memory is made.

SMAC System Management Memory Access

SMAC = 0: All memory accesses in normal mode go to system memory with ADS# output active. In normal mode, execution of SGS Thomson specific SMM instructions generate an invalid opcode exception.

SMAC = 1: Memory accesses while in normal mode that fall within the specified SMM address region generate an SMADS# output and access SMM memory. SMI# input is ignored.

MMAC Main Memory Access

MMAC = 0: All Memory accesses while in SMM mode go to SMM memory with SMADS# output active.

MMAC = 1: Data accesses while in SMM mode that fall within the specified SMM address region will generate an ADS# output and access main memory. Code fetches are not effected by the MMAC bit. Code fetches from the SMM address region always generate an SMADS# output and access SMM memory. If both the SMAC and MMAC bits are set to 1, the MMAC bit has precedence.

Table 2 - 2 CCR2

Reg. INDEX = C2h

7	6	5	4	3	2	1	0
SUSP	**BWRT**	**BARB**	**WT1**	**HALT**	**LOCK-NW**	**WBAK**	**COP/Reserved**

HALT Suspend on HALT.

HALT = 0: CPU does not enter suspend mode following execution of a HLT instruction

HALT = 1: CPU enters suspend mode following execution of a HLT instruction.

SUSP Enable Suspend Pins.

SUSP = 0: SUSP# input is ignored and SUSPA# output floats.

SUSP = 1: SUSP# input and SUSPA# output are enabled.

Table 2 - 1 CCR3

Reg. INDEX = C3h

7	6	5	4	3	2	1	0
						NMIEN	**SMI-LOCK**

Reserved

SMI_LOCK SMM Register Lock.

SMI_LOCK = 0: Any program in normal mode, as well as SMM software, has access to all Configuration Control Registers.

SMI_LOCK = 1: The following Configuration Control Register bits can not be modified unless operating in SMM mode:
SMI, SMAC, MMAC, NMIEN, SMI_LOCK, and SMAR register size fields.

NMIEN NMI Enable.

NMIEN = 0: NMI (Non-Maskable Interrupt) is not recognized during SMM. One occurrence of NMI is latched and serviced after SMM mode is exited. The NMIEN bit should be cleared before executing a RSM instruction to exit SMM.

NMIEN = 1: NMI is enabled during SMM. This bit should only be set temporarily while in the SMM routine to allow NMI interrupts to be serviced. NMIEN should not be set

to 1 while in normal mode. If NMIEN = 1 when an SMI occurs, an NMI could occur before the SMM code has initialized the Interrupt Descriptor Table.

Table 2 - 3. SMAR SMM Address Region Registers

Reg. Index = CDh		Reg. Index = CEh		Reg. Index = CFh		
7	0	7	0	7	4	3 0
Base Address						Size
A31	A24	A23	A16	A15	A12	see table below

Table 2 - 4. SMAR SIZE FIELD

Bits 3-0	BLOCK SIZE		Bits 3-0	BLOCK SIZE
0h	Disable		8h	512 KBytes
1h	4 KBytes		9h	1 MBytes
2h	8 KBytes		Ah	2 MBytes
3h	16 KBytes		Bh	4 MBytes
4h	32 KBytes		Ch	8 MBytes
5h	64 KBytes		Dh	16 MBytes
6h	128 KBytes		Eh	32 MBytes
7h	256 KBytes		Fh	4 KBytes (same as 1h)

2.3 SMM Instruction Summary

SGS-Thomson has added seven new instructions to the X86 standard instruction set to aid in SMM programming. These instructions are only valid when:

CPL = 0 **and**

SMI is enabled (CCR1 bit 1 = 1) **and**

SMAR size > 0 **and**

either [in SMM mode **or** SMAC is on (CCR1 bit 2 =1)]

The CPU will generate an undefined opcode fault when the above conditions are not met and one of the SMM instructions are executed. The assembly language macro SMIMAC.INC listed in Appendix A will automatically generate the appropriate machine code when included in a source file containing SGS-Thomson SMM instructions.

Most of the SGS-Thomson SMM instructions are used to access the non-programmer visible internal descriptors. The standard x86 instructions can not access this information inside the CPU. This information is stored in memory in a 10 Byte area that is comprised of both the descriptor (8-Bytes) and the segment register/selector (2 Bytes). The 8 Byte descriptor is in the same format that is found in the GDT or LDT. If the data area is dword aligned, it will minimize the memory access time.

Table 2 - 5. Register and Descriptor Save Format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
SELECTOR or SEGMENT																+8
BASE 31 - 24								G	D	0	AVL	LIMIT 19 - 16				+6
P	DPL			DT	TYPE			BASE 23 - 16								+4
BASE 15 - 0																+2
LIMIT 15 - 0																+0

2.3.1 RSDC - Restore Register and Descriptor

Instruction	Opcode	Parameters	Core Clocks
RSDC	0F 79 [mod sreg3 r/m]	sreg3, mem80	10

RSDC loads the information at the mem80 into a segment register/selector and its associated descriptor. Attempting to use this instruction to load the Code Segment or Code Selector will generate an invalid opcode instruction. Code Segment or Code Selector is restored from the SMM header as part of the RSM instruction.

2.3.2 RSLDT - Restore LDT and Descriptor

Instruction	Opcode	Parameters	Core Clocks
RSLDT	0F 7B[mod 000 r/m]	mem80	10

RSLDT loads the information at the mem80 into Local Descriptor Table Register and its associated descriptor.

2.3.3 RSM - Resume Back to Normal Mode

Instruction	Opcode	Parameters	Core Clocks
RSM	0F AA	None	76

RSM will restore the state of the CPU from the SMM header at the top of SMM space and exit SMM. This is the last instruction executed in an SMI handler. After the CPU state is restored, the SMI# pin is driven inactive for one clock then floated so the pin can be driven by the system.

2.3.4 RSTS - Restore TSR and Descriptor

Instruction	Opcode	Parameters	Core Clocks
RSTS	0F 7D [mod 000 r/m]	mem80	10

RSTS loads the information at the mem80 address into the Task Register and its associated descriptor.

2.3.5 SMINT - Software SMM Interrupt

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

SMINT will cause the CPU to enter SMM as though the hardware SMI# pin was sampled low. The S bit in the SMM header is set. The SMI# signal is not driven by the CPU when SMM is entered with SMINT.

2.3.6 SVDC - Save Register and Descriptor

Instruction	Opcode	Parameters	Core Clocks
SVDC	0F 78 [mod sreg3 r/m]	sreg3, mem80	18

SVDC saves the contents of a segment register/selector and its associated descriptor to memory at mem80. This instruction can be used on any segment/selector including the Code Segment.

2.3.7 SVLDT - Save LDT and Descriptor

Instruction	Opcode	Parameters	Core Clocks
SVLDT	0F 7A [mod 000 r/m]	mem80	18

SVLDT saves the Local Descriptor Table Selector and non-programmer visible descriptor information at the address location mem80.

2.3.8 SVTS - Save TSR and Descriptor

Instruction	Opcode	Parameters	Core Clocks
SVTS	0F 7C	mem80	18

SVTS saves the Task Register and its associated descriptor to address location mem80.

3. SMM SOFTWARE CONSIDERATIONS

This section provides an overview of SGS-Thomson SMM coding and information helpful in developing SMM code.

3.1 Enabling SMM

Many systems have memory controllers that aid in the initialization of SMM memory. SGS-Thomson SMM features allow the initialization of SMM memory without external hardware memory remapping.

When loading SMM memory with an SMI interrupt handler it is important that the SMI# does not occur before the handler is loaded. This can be done by not setting SMAC=0 and SMI=1 before the SMI handler is installed. It is necessary to load SMAR with appropriate values before the SMM memory is accessible. To load SMM memory with a program it is first necessary to enable SMM memory without enabling the SMI pins by setting SMAC. Setting SMI=1 will then map the SMM memory region over main memory. The SMM region is physically mapped by the assertion of SMADS# to allow memory access within the SMM region. A REP MOV instruction can then be used to transfer the program to SMM memory. After initializing SMM memory, negate SMAC to activate potential SMI#s.

SMM space can be located anywhere in the 4-GByte address range. However, if the location of SMM space is beyond 1 Mbyte, the value in CS will truncate the segment above 16-bits when stored to the stack. This would prohibit doing calls or interrupts from real mode without restoring the 32-bit features of the 486 because of the incorrect return address on the stack.

```
; load SMM memory from system memory

include SMIMAC.INC
SMMBASE = 68000h
SMMSIZE = 4000h          ;SMM SIZE is 16K
SMI   = 1 shl 1
SMAC  = 1 shl 2
MMAC  = 1 shl 3

      mov  al, 0cdh      ;index SMAR, SMM baseA31-A24
      out  22h, al       ;select
      mov  al, 00h       ;set high SMM address to 00
      out  23h, al       ;write value
      mov  al, 0ceh      ;index SMAR,SMM baseA23-A16
      out  22h, al       ;select
      mov  al, 06h       ;set mid SMM address to 06h
      out  23h, al       ;write value
      mov  al, 0cfh      ;SMAR,SMM baseA15-A12 & SIZE
```

```
        out  22h, al       ;select
        mov  al, 083h      ;set SMM lower addr. 80h, 16K
        out  23h, al       ;write value
        mov  al, 0c1h      ;index to CCR1
        out  22h, al       ;select CCR1 register
        in   al, 23h       ;read current CCR1 value
        mov  ah,  al       ;save it
        mov  al, 0c1h      ;index to CCR1
        out  22h, al       ;select CCR1 register
        mov  al, ah
        or   al, SMI or SMAC; set SMI and SMAC
        out  23h, al       ;new value now in CCR1, SMM now
                           ;mapped in
        mov  ax, SMMBASE shr 4
        mov  es, ax
        mov  edi, 0        ;es:di = start of the SMM area
        mov  esi, offset SMI_ROUTINE  ;start of copy of SMM
        mov  ax, seg SMI_ROUTINE     ;routine in main memory
        mov  ds, ax
        mov  ecx, (SMI_ROUTINE_LENGTH+3)/4   ;calc. length


    ; this line copies the SMM routine from DS:ESI to ES:EDI
          rep
          movs dword ptr es:[edi],dword ptr ds:[esi]


    ; now disable SMI by clearing SMAC and SMI
          mov  al, 0c1h      ;index to CCR1
          out  22h, al       ;select CCR1 register
          mov  al, ah        ;AH is still old value
          and  al, NOT SMAC  ;disable SMAC, enable SMI#
          out  23h, al       ;write new value to CCR
```

3.2 SMM Handler Entry State

At the beginning of the SMM routine, before control is transferred to code executing at the SMM base, certain portions of the CPU state are saved at the top of SMM memory. To optimize the speed of SMM entry and exit, the CPU saves the minimum CPU state information necessary for an SMI interrupt handler to execute and return to the interrupted context. The information is saved to the SMM header at the top of the defined SMM region (starting at SMM base + size - 30h) as shown in Figure-3-1. Of the typically used program registers only the CS, IP, EFLAGS, CR0, and DR7 are saved upon entry to SMM. This requires that data accesses use a CS segment override to

save other registers and access data in SMM memory. To use any other register the SMM programmer must first save the contents using the SVDC instruction for segment registers or MOV operations for general purpose registers (See SGS Thomson SMM instruction description Section 2.3). It is possible to save all the CPU registers as needed. See Section 3.3 for an example saving and restoring the entire CPU state.

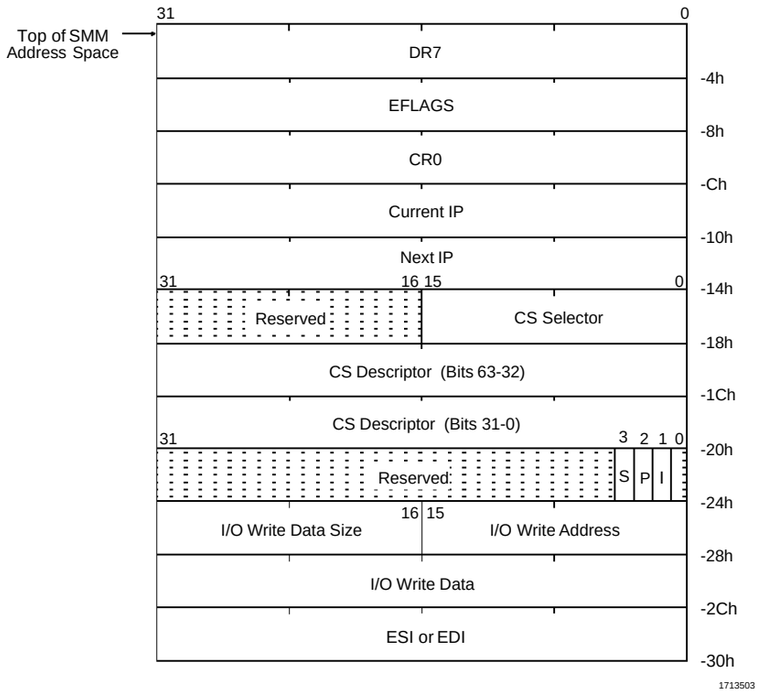

Figure 3 - 1. SMM Memory Space Header

Unique to the SGS-Thomson CPU is that the CPU saves the previous EIP (CURRENT_IP), before the SMI event, and the next EIP (NEXT_IP) that will be executed after exiting the SMI handler. Upon execution of an RSM instruction, control is returned to the NEXT_IP. The value of the NEXT_IP may need to be modified for restarting I/O instructions. This modification is a simple move (MOV) of the CURRENT_IP value to the NEXT_IP location. Execution is then returned to the I/O instruction, rather than to the instruction after the I/O instruction. Table 3-1 lists the SMM header information needed to restart an I/O instruction. The restarting of I/O instructions may also require modifications to the ESI, ECX and EDI depending on the instruction (see Section 3.6 for an example.)

The EFLAGS, CR0 and DR7 registers are set to their reset values upon entry to the SMI handler. Resetting these registers has implications for setting breakpoints using the debug registers. Breakpoints can not be set prior to the SMI interrupt using debug registers. A debugger will only be able to set a code breakpoint using INT 3 outside of the SMM handler. See Section 3.11 for restrictions on debugging SMM code. Once the SMI has occurred and the debugger has control in SMM space, the debug registers can be used for the remainder of the SMI handler execution.

If the S bit in the SMM header is set, the SMM entry was the result of an SMINT instruction.

Upon SMM entry, I/O trap information is stored in the SMM memory space header. This information allows restarting of I/O instructions, as well as the easy emulation of I/O functions by the SMM handler. This data is only valid if the instruction executing when the SMI occurred was an I/O instruction. The three I/O Write fields (I/O Write Data, I/O Write Address and I/O Write Data Size) are only valid when an I/O write was trapped.

Table 3 - 1 I/O Trap Information

Bit	Description	Size
P	REP INSx/OUTSx Indicator 0 = Current instruction has a REP prefix 1 = Current instruction does not have a REP prefix	1 bit
I	IN, INSx, OUT, or OUTSx Indicator 0 = Current instruction performed an I/O READ 1 = Current instruction performed an I/O WRITE	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 Date	Data written during I/O trapped write	4 Bytes
ESI or EDI	Value of appropriate index register before the trapped I/O instruction	4 Bytes

The values found in the I/O trap information fields are specified below for all cases.

Table 3 - 2 Valid I/O Trap Cases

Valid Cases	P	I	I/O Write Data Size	I/O Write Address	I/O Write Data	ESI or EDI
not an I/O ins.	x	x	x	x	x	x
IN	0	0	x	x	x	EDI
INS	0	0	x	x	x	EDI
REP INS	1	0	x	x	x	EDI
OUT al	0	1	01h	I/O Address	xxxxxxdd	ESI
OUT ax	0	1	03h	I/O Address	xxxxdddd	ESI
OUT eax	0	1	0Fh	I/O Address	dddddddd	ESI
OUTSB	0	1	01h	I/O Address	xxxxxxdd	ESI
OUTSW	0	1	03h	I/O Address	xxxxdddd	ESI
OUTSD	0	1	0Fh	I/O Address	dddddddd	ESI
REP OUTSB	1	1	01h	I/O Address	xxxxxxdd	ESI
REP OUTSW	1	1	03h	I/O Address	xxxxdddd	ESI
REP OUTSD	1	1	0Fh	I/O Address	dddddddd	ESI

x: invalid

Upon SMM entry, the CPU enters the following state:

Table 3 - 3 SMM Entry State

CS	SMM base specified by SMAR, CS limit is set to 4 GBytes
EIP	0000 0000h; Begins execution at the base of SMM memory
EFLAGS	0000 0002h; Reset State
CR0	6000 0010h; Real Mode, Cache in Write Through
DR7	0000 0400h; Traps disabled

3.3 Maintaining the CPU State

The following registers are not automatically saved/restored on SMM entry/exit.

General Purpose Registers:	EAX, EBX, ECX, EDX
Pointer and Index Registers:	EBP, ESI, EDI, ESP
Selector/Segment Registers:	DS, ES, SS, FS, GS
Descriptor Table Registers:	GDTR, IDTR, LDTR, TR
Control Registers:	CR2, CR3
Debug Registers:	DR0, DR1, DR2, DR3, DR6
Configuration Registers:	CCR1, CCR2, CCR3, SMAR
FPU Registers:	Entire FPU state.

If any of these registers need to be used by the SMM routine, the registers need to be saved after entry to the SMM routine and then restored prior to exit from SMM. Additionally, if power is to be removed from the CPU and the system is required to return to the same system state after power is reapplied, then the entire CPU state must be saved to a non-volatile memory subsystem such as a hard disk.

3.3.1 Maintaining Common CPU Registers

The following is an example of the instructions needed to save the entire CPU state and restore it. This code sequence will work from real mode if the conditions needed to execute SGS-Thomson SMM instructions are met (see Section 2.3).

```
; Save and Restore the common CPU registers.
; The information automatically saved in the
; header on entry to SMM is not saved again.
include SMIMAC.INC
      .386P      ;required for SMIMAC.INC macro
      mov   cs:save_eax,eax
      mov   cs:save_ebx,ebx
      mov   cs:save_ecx,ecx
      mov   cs:save_edx,edx
      mov   cs:save_esi,esi
      mov   cs:save_edi,edi
      mov   cs:save_ebp,ebp
      mov   cs:save_esp,esp
      svdc  cs:,save_ds,ds
      svdc  cs:,save_es,es
      svdc  cs:,save_fs,fs
      svdc  cs:,save_gs,gs
      svdc  cs:,save_ss,ss
      svldt cs:,save_ldt     ;sldt is not valid in real mode
```

```
        svts   cs:,save_tsr     ;str is not valid in real mode
        db     66h              ;32bit version saves everything
        sgdt   fword ptr cs:[save_gdt]
        db     66h              ;32bit version saves everything
        sidt   fword ptr cs:[save_idt]
    ; at the end of the SMM routine the following code
    ; sequence will reload the entire CPU state
        mov    eax,cs:save_eax
        mov    ebx,cs:save_ebx
        mov    ecx,cs:save_ecx
        mov    edx,cs:save_edx
        mov    esi,cs:save_esi
        mov    edi,cs:save_edi
        mov    ebp,cs:save_ebp
        mov    esp,cs:save_esp
        rsdc   ds,cs:,save_ds
        rsdc   es,cs:,save_es
        rsdc   fs,cs:,save_fs
        rsdc   gs,cs:,save_gs
        rsdc   ss,cs:,save_ss
        rsldt  cs:,save_ldt
        rsts   cs:,save_tsr
        db     66h
        sgdt   fword ptr cs:[save_gdt]
        db     66h
        sidt   fword ptr cs:[save_idt]
    ; the data space to save the CPU state is in
    ; the Code Segment for this example
    save_ds      dt   ?
    save_es      dt   ?
    save_fs      dt   ?
    save_gs      dt   ?
    save_ss      dt   ?
    save_ldt     dt   ?
    save_tsr     dt   ?
    save_eax     dd   ?
    save_ebx     dd   ?
    save_ecx     dd   ?
    save_edx     dd   ?
    save_esi     dd   ?
    save_edi     dd   ?
    save_ebp     dd   ?
    save_esp     dd   ?
    save_gdt     df   ?
    save_idt     df   ?
```

3.3.2 Maintaining Control Registers

CR0 is maintained in the SMM header. CR2 and CR3 need only be saved if the SMM routine will be entering protected mode and enabling paging. Most SMM routines will not need to enable paging. However, if the CPU is going to be powered off, these registers like all the others need to be saved.

3.3.3 Maintaining Debug Registers

DR7 is maintained in the SMM Header. Since DR7 is automatically initialized to the reset state on entry to SMM, the Global Disable bit (DR7 bit 13) will be cleared. This allows the SMM routine to access all of the Debug Registers. Returning from the SMM handler will reload DR7 with its previous value. In most cases, SMM routines will not make use of the Debug Registers and they will only need to be saved if the CPU needs to be powered down.

3.3.4 Maintaining Configuration Control Registers

The SMM routine should be written so that it maintains the Configuration Control Registers in the state that they were initialized to by the BIOS at power-up.

3.3.5 Maintaining FPU State

If power will be removed from the FPU, or if the SMM routine will execute FPU instructions, then the FPU state needs to be maintained for the application running before SMM was entered. If the FPU state is to be saved and restored from within SMM, there are certain guidelines that must be followed to make SMM completely transparent to the application program.

The complete state of the FPU can be saved and restored with the FNSAVE and FNRSTOR instructions. FNSAVE is used instead of the FSAVE because FSAVE will wait for the FPU to check for existing error conditions before storing the FPU state. If there is a unmasked FPU exception condition pending, the FSAVE instruction will wait until the exception condition is serviced. In order to be transparent to the application program, the SMM routine should not service the exception. If the FPU state is restored with the FNRSTOR instruction before returning to normal mode, the application program can correctly service the exception. Any FPU instructions can be executed within SMM once the FPU state has been saved.

The information saved with the FSAVE instruction varies depending on the operating mode of the CPU. To save and restore all FPU information, the 32-bit protected mode version of the FPU save

and restore instruction should be used. This can be accomplished by using the following code example:

```
; Save the FPU state
      mov    eax,CR0
      or     eax,00000001h
      mov    CR0,eax          ;set the PE bit in CR0
      jmp    $+2              ;clear the prefetch que
      db     66h              ;do 32bit version of fnsave
      fnsave [save_fpu]       ;saves fpu state to
                              ;the address DS:[save_fpu]
      mov    eax,CR0
      and    eax, 0FFFFFFFEh  ;clear PE bit in CR0
      mov    CR0,eax          ;return to real mode

;now the SMM routine can do any FPU instruction.
;Restore the FPU state before executing a RSM
      mov    eax,CR0
      or     eax,00000001h
      mov    CR0,eax          ;set the PE bit in CR0
      jmp    $+2              ;clear the prefetch que
      db     66h              ;do 32bit version of fnsave
      frstor [save_fpu]       ;restore the FPU state
                              ;Some assemblers may require
                              ;use of the fnrstor instruction
      mov    eax,CR0
      and    eax, 0FFFFFFFEh  ;clear PE bit in CR0
      mov    CR0,eax          ;return to real mode
```

Be sure that all interrupts are disabled before using this method for entering protected mode. Any attempt to load a selector register while in protected mode will shutdown the CPU since no GDT is set up. Setting up a GDT and doing a long jump to enter protected mode will also work correctly.

3.4 Initializing the SMM Environment

After entering SMM and saving the CPU registers that will be used by the SMM routine, a few registers need to be initialized.

Segment registers need to be initialized if the CPU was operating in protected mode when the SMI interrupt occurred. Segment registers that will be used by the SMM routine need to be loaded with known limits before they are used. The protected mode application could have set a segment limit to less than 64K. To avoid a protection error, all segment registers can be given limits of 4 GBytes. This can be done with the SGS Thomson RSDC instruction and will allow access to the

full 4-GBytes of possible system memory without entering protected mode. Once the limits of a segment register are set, the base can be changed by use of the MOV instruction.

An Interrupt Descriptor Table (IDT) needs to be set up in SMM memory before any interrupts or exceptions occur. Once initialized, the SMI handler can execute calls, jumps, and other changes of flow and will generate software interrupts and faults. The Interrupt Descriptor Table Register can be loaded with an LIDT instruction to point to a small IDT in SMM memory that can handle the possible interrupts and exceptions that might occur while in the SMM routine.

A stack should always be set up in SMM memory so that stack operations done within SMM do not affect the system memory.

```
; SMM environment initialization example
      rsdc   ds,cs:,seg4G     ;DS is a 4GByte segment, base=0
      rsdc   es,cs:,seg4G     ;ES is a 4GByte segment, base=0
      rsdc   fs,cs:,seg4G     ;FS is a 4GByte segment, base=0
      rsdc   gs,cs:,seg4G     ;GS is a 4GByte segment, base=0
      rsdc   ss,cs:,seg4G     ;SS is a 4GByte segment, base=0
      lidt   cs:smm_idt       ;load IDT base and limit for
                              ;SMM's IDT
      mov    esp, smm_stack
      jmp    continue_smm_code
;
;descriptor of 4GByte data segment for use by rsdc
seg4G       dw   0ffffh       ; limit 4G
            dw   0            ; base = 0
            db   0            ; base = 0
            db   10010011B    ; data segment, DPL=0,P=1
            db   8fh          ; limit = 4G,
            db   0h           ; base = 0
            dw   0            ; segment register = 0
smm_idt     dw   smm_idt_limit
            dd   smm_idt_base
```

3.5 Accessing Main Memory Overlapped by SMM Memory

When in SMM mode there are instances where the program needs access to the system memory that is overlapping with SMM memory. This need most commonly occurs when the SMM routine is trying to save the entire memory image to disk before powering down the system. To access main memory overlapping the SMM space (i.e., generate ADS# for memory MOV instructions rather than SMADS#) set the MMAC bit in CCR1. The following code will enable and then disable MMAC:

```
; Set MMAC to access main memory
MMAC = 1 shl 3
      mov    al, 0c1h        ;select CCR1
      out    22h, al
      in     al, 23h         ;get CCR1 current value
      mov    ah, al          ;save it
      mov    al, 0c1h        ;select CCR1 again
      out    22h, al
      mov    al, ah
      or     al, MMAC        ;set MMAC
      out    23h, al         ;write new value to CCR1
;Now all data memory access will use ADS#, Code fetches
;will continue to be done with SMADS# from SMM memory.
;
;Disable MMAC
      mov    al, 0c1h        ;select CCR1
      out    22h, al
      mov    al, ah          ;get old value of CCR1
      out    23h, al         ;and restore it
```

3.6 I/O Restart

When implementing power management into a system it is common to want to power down peripherals when they are not in use. When an I/O instruction is issued to a powered down device, the SMM routine is called to power up the peripheral and then reissue the I/O instruction. SGS-Thomson CPUs make it easy to restart the I/O instruction that has generated an SMI interrupt.

The system will generate an SMI interrupt when an I/O bus cycle to a powered-down peripheral is detected. The SMM routine should interrogate the system hardware to find out if the SMI was caused by an I/O trap. By checking the SMM header information, the SMM routine can determine the type of I/O instruction that was trapped. If the I/O instruction has a REP prefix, the ECX register needs to be incremented before restarting the instruction. If the I/O trap was on a string I/O instruction, the ESI or EDI registers must be restored to their previous value before restarting the instruction.

The following code example shows how easy I/O restart is with the SGS Thomson CPU.

```
;Restart the interrupted instruction
      mov    eax,dword ptr cs:[SMI_CURRENTIP]
      mov    dword ptr cs:[SMI_NEXTIP],eax
      mov    al,byte ptr cs:[SMI_BITS]

;test for REP instruction
```

```
        bt     ax,2             ;rep instruction?
                                ;(result to Carry)
        adc    ecx,0             ;if so, increment ecx
        test   al,1 shl 1       ;test bit 1 to see
                                ;if an OUTS or INS
        jnz    out_instr
    ; A port read (INS or IN) instruction caused the
    ; chipset to generate an SMI instruction.
    ; Restore EDI from SMM header.
        mov    edi, dword ptr cs:[SMI_ESIEDI]
        jmp    common1

    ; A port write (OUTS or OUT) instruction caused the
    ; chipset to generate an SMI instruction.
    ; Restore ESI from SMM header.
    out_instr:
        mov    esi, dword ptr cs:[SMI_ESIEDI]
    common1:
```

3.7 I/O Port Shadowing and Emulation

Some system peripherals contain write-only ports. In a system that does power management, these peripherals need to be powered off and then reinitialized when their functions are needed later. The SGS Thomson SMM implementation makes it very easy to monitor the last value written to specific I/O ports. This process is known as shadowing. If the system can generate an SMI whenever specific I/O addresses get accessed, the SMM routine can, transparently to the system, monitor the port activity. The SMM header contains the address of the I/O write as well as the data. In addition, information is saved which indicates whether it is a byte, word or dword write. With this information, shadowing system write-only ports becomes trivial.

Some peripheral components contain registers that must be programmed in a specific order. If an SMI interrupt occurs while an application is accessing this type of peripheral, the SMI routine must be sure to reload the peripheral registers to the same stage before returning to normal mode. If the SMM routine needs to access such a peripheral, the previous normal mode state must be restored. The previous accesses that were shadowed by previous SMM calls can be used to reload the peripheral registers back to the stage where the application was interrupted. The application can then continue where it left off accessing the peripheral.

In a similar way, the SGS-Thomson SMM implementation allows the SMM routine to emulate the function of peripheral components in software.

3.8 Return to HLT Instruction

To make an SMI interrupt truly transparent to the system, an SMI interrupt from a HLT instruction should return to the HLT instruction. There are known cases with DOS software where returning from an SMI handler to the instruction following the HLT will cause a system error. To determine if a HLT instruction was interrupted by the SMI, the opcode from memory needs to be interrogated. This code example describes how to determine if the current instruction is a HLT and how to restart it.

```
;This is the start of specific code to check if the SMI
;occurred while in a HLT instruction. If it did, then
;return back to the HLT instruction when SMI is finished.

     rsdc   fs,cs:,[seg4G]   ;FS is base=0 limit=4G data
                             ;segment to be used to check if
                             ;HLT instruction was executing

;on a SGS Thomson part, if the SMI occurred while in a HLT
;instruction, the CURRENT IP and the NEXT IP will both
;point to the instruction following the HLT.
     mov   eax,cs:dword ptr[SMI_CURRENTIP]
     cmp   eax,cs:dword ptr[SMI_NEXTIP]
     jne   not_hlt           ;can't be a HLT but could be
                             ;a LOOP or REP
;load EAX with CS base from the SMM header
     mov   ax,cs:word ptr [SMI_CSSELH+2]
     mov   al,cs:byte ptr [SMI_CSSELH]
     shl   eax,10h
     mov   ax,cs:word ptr[SMI_CSSELL+2]
;calculate linear address
     add   eax,cs:dword ptr [SMI_CURRENTIP]
     dec   eax               ;decrement to HLT instruction
     mov   edx,eax           ;save lin addr in edx
     mov   eax,cs:dword ptr [SMI_CR0] ;check if paging on
     test  eax,80000000h
     je    no_paging         ;if no paging then linear
                             ;address = physical address
;set MMAC to get access to Main memory
     mov   al,0c1h
     out   22h,al
     in    al,23h
     mov   cl,al             ;save old CCR1 value in cl
     mov   al,0c1h
     out   22h,al
```

```
      mov   al,cl
      or    al,08h            ;set MMAC bit in CCR1
      mov   al,0c1h
      out   23h,al
      mov   eax,CR3           ;get Page Directory Base Reg
      and   eax,0fffff000h
      mov   ebx,edx           ;linear address
      shr   ebx,22            ;get 10 byte Directory Entry
;read Directory Table
      mov   eax,dword ptr fs:[eax+ebx*4]
      and   eax,0fffff000h
      mov   ebx,edx           ;linear address
      shr   ebx,12
      and   ebx,03ffh         ;get 10 byte Page Table Entry
      mov   eax,dword ptr fs:[eax+ebx*4]
      and   eax,0fffff000h
      mov   ebx,edx           ;linear address
      and   ebx,0fffh         ;get 12 byte offset into page
;Get the physical address of the instruction before the
;Current IP.  Save in BL.
      mov   bl,byte ptr fs:[eax+ebx]
      mov   al,0c1h           ;set MMAC back to normal
      out   22h,al
      mov   al,cl
      out   23h,al            ;MMAC = 0
      jmp   got_inst

;If paging is not enabled then checking for the HLT
;instruction is easy since the linear address equals
;the physical address.

no_paging:
      mov   al,0c1h           ;set MMAC
      out   22h,al
      in    al,23h
      mov   ah,al
      mov   al,0c1h
      out   22h,al
      mov   al,ah
      or    al,08h
      out   23h,al
;get instruction interrupted by SMI
      mov   bl,byte ptr fs:[edx]
      mov   al,0c1h           ;store it in BL
      out   22h,al
```

```
      mov   al,ah
      out   23h,al           ;set MMAC back to normal

got_inst:
      cmp   bl,0f4h          ;was it a HLT instruction?
      jne   not_hlt          ;if not a F4 then not a HLT
                             ;set up SMM header to return
                             ;to the HLT instruction
      dec   cs:dword ptr [SMI_NEXTIP]

not_hlt:
      jmp    continue_SMI_routine

; data within the SMM Space Code Segment
seg4G dw    0ffffh           ;limit 15-0
      dw    0                ;base
      db    0                ;base
      db    10010011B        ;data segment, DPL=0, present
      db    8Fh              ;high limit =f, Gran =4K, 16 bit
      db    0                ;base
      dw    0
```

3.9 Exiting the SMI Handler

When the RSM instruction is executed at the end of the SMI handler, the EIP is loaded from the SMM header at the address (SMMbase + SMMsize - 14h) called NEXT_IP. This permits the instruction to be restarted if NEXT_IP was modified by the SMM program. The values of ECX, ESI, and EDI, prior to the execution of the instruction that was interrupted by SMI, can be restored from information in the header which pertains to the INx and OUTx instructions. See Section 3.6 for an example program to restart an I/O instruction. The only registers that are restored from the SMM header are CS, NEXT_IP, EFLAGS, CR0, and DR7. All other registers which were modified by the SMM program need to be restored before executing the RSM instruction.

3.10 Testing and Debugging SMM Code

An SMI routine can be debugged with standard debugging tools (such as DOS DEBUG) if the following requirements are met:

1. The debugger will only be able to set a code break point using INT 3 outside of the SMI handler. The debug control register DR7 is set to the reset value upon entry to the SMI handler. Therefore, any break conditions in DR0-3 will be disabled after entry to SMM. Debug registers can be used if set after entry to the SMI handler and DR0-3 are saved.
2. The debugger needs to be running in real mode and the SMM routine can not enter protected mode. This insures that normal system interrupts, BIOS calls and the debugger will work correctly from SMM mode.
3. Before an INT 3 break point is executed, all segment registers should have their limits modified to 64K, or larger, within the SMM routine.

4. Power Management Features

The SGS-Thomson CPU provides several methods and levels of power management. The fully static design allows clock stopping. Suspend Mode, SMM and 3.3 volt operation can be used to achieve optimum CPU and system power management. Table 4-1 summarizes the various power management options for the SGS-Thomson CPU.

Table 4 - 1 Power Management Options

Option	Typical Current Options
Reduced Clock Frequency	(13 x f_{CLK}(MHz)) + 150 mA @ 5.0 V
5.0V operation	610 mA @ 33 MHz, 765 mA @ 50 MHz
3.3V operation	360 mA @ 33 MHz
Remove Clock	150 mA
Suspend Mode clock operating	15 mA
Suspend Mode and Remove Clock	450 µA
Remove Power	0 mA

Note: Values listed are approximations. Refer to the appropriate SGS-Thomson data book for DC specifications.

4.1 Reducing the Clock Frequency

The SGS-Thomson CPU is a fully static design meaning the input clock frequency can be reduced or stopped without a loss of internal CPU data or state. The system designer can make decisions to reduce the clock frequency by using SGS-Thomson SMM capabilities, Advanced Power Management (APM) software API and/or chipset capabilities. When the clock is removed and then reasserted, execution will begin with the instruction where the clock was removed from the CPU.

4.2 Lowering the CPU Supply Voltage

SGS-Thomson CPUs are available that operate at either 3.3 or 5.0 volts. Parts rated at 3.3 volts have the letter 'V' included in the part number (Refer the appropriate SGS-Thomson data book for complete ordering information). The typical current (I_{cc}) drawn by the SGS-Thomson CPU is reduced by approximately 50% when operating at 3.3 instead of 5.0 volts. Operating the CPU at 3.3 volts can reduce CPU power consumption by over 70%, as the power consumption increases as the square of the power supply voltage ($P-=-V^2/R$ and $P-=-CV^2F$).

4.3 Suspend Mode

The SGS-Thomson CPU allows suspend mode to be entered either through software or hardware.

The software initiates suspend mode through execution of a HLT instruction if CCR2 bit 3 (HALT) is set. After the HLT instruction is executed, the CPU enters suspend mode and asserts the suspend acknowledge (SUSPA#) pin (if the SUSP bit in CCR2 was set to enable the SUSPA# pin).

Hardware initiates suspend mode by using two new pins on the CPU, SUSP# and SUSPA#. When SUSP# is asserted, the CPU first completes any pending instructions and bus cycles, and then enters suspend mode. Once in suspend mode, the SUSPA# pin is asserted by the CPU.

A. ASSEMBLER MACROS FOR SGS-THOMSON INSTRUCTIONS

The include file SMICAM.INC provides a complex set of macros which generate SMM opcodes along with the appropriate mod/rm bytes. In order to function, the macros require that the labels which are accessed correspond to the specified segment. Thus segment overrides must be passed to the macro as an argument.

Do not specify a segment override if the default segment for an address is being used. If an address size override is used, a final argument of '1' must be passed to the macro as well. Address size overrides must be presented explicitly to prevent the assembler from generating them automatically and breaking the macros.

```
;SMM Instruction Macros - SMIMAC.INC
;Macros which generate mod/rm automatically

svdc    MACRO   segover,addr,reg,adover
        domac   segover,addr,reg,adover,78h
        ENDM
rsdc    MACRO   reg,segover,addr,adover
        domac   segover,addr,reg,adover,79h
        ENDM
svldt   MACRO   segover,addr,adover
        domac   segover,addr,es,adover,7ah
        ENDM
rsldt   MACRO   segover,addr,adover
        domac   segover,addr,es,adover,7bh
        ENDM
svts    MACRO   segover,addr,adover
        domac   segover,addr,es,adover,7ch
        ENDM
rsts    MACRO   segover,addr,adover
        domac   segover,addr,es,adover,7dh
        ENDM
rsm     MACRO
        db      0fh,0aah
        ENDM
;Sub-Macro used by the above macro

domac   MACRO   segover,addr,reg,adover,op
        local   place1,place2,count
        count   = 0
        ifnb    < adover >
            count=count+1
        endif
        ifnb   < segover >
```

```
            count=count+1
        endif
        if      (count eq 0)
                nop          ;expanding the opcode one byte
        endif
        place1  = $
;pull off the proper prefix byte count
        mov     word ptr segover addr,reg
        org     place1+count
        mov     word ptr segover addr,reg
        place2  = $
;patch the opcode
        org     place1+(count*2)-1
        db      0Fh,op
        org     place2
ENDM

;Offset Definition for access into SMM space
SMI_SAVE STRUC
     $ESIEDI         DD      ?
     $IOWDATA        DD      ?
     $IOWADDR        DW      ?
     $IOWSIZE        DW      ?
     $BITS           DD      ?
     $CSSELL         DD      ?
     $CSSELH         DD      ?
     $CS             DW      ?
     $RES1           DW      ?
     $NEXTIP         DD      ?
     $CURRENTIP      DD      ?
     $CR0            DD      ?
     $EFLAGS         DD      ?
     $DR7            DD      ?
SMI_SAVE ENDS
SMI_ESIEDI      EQU  ($ESIEDI + SMMSIZE - SIZE SMI_SAVE)
SMI_IOWDATA     EQU  ($IOWDATA+ SMMSIZE - SIZE SMI_SAVE)
SMI_IOWADDR     EQU  ($IOWADDR+ SMMSIZE - SIZE SMI_SAVE)
SMI_IOWSIZE     EQU  ($IOWSIZE+ SMMSIZE - SIZE SMI_SAVE)
SMI_BITS        EQU  ($BITS   + SMMSIZE - SIZE SMI_SAVE)
SMI_CSSELL      EQU  ($CSSELL + SMMSIZE - SIZE SMI_SAVE)
SMI_CSSELH      EQU  ($CSSELH + SMMSIZE - SIZE SMI_SAVE)
SMI_CS          EQU  ($CS     + SMMSIZE - SIZE SMI_SAVE)
SMI_RES1        EQU  ($RES1   + SMMSIZE - SIZE SMI_SAVE)
SMI_NEXTIP      EQU  ($NEXTIP + SMMSIZE - SIZE SMI_SAVE)
SMI_CURRENTIP   EQU  ($CURRENTIP+ SMMSIZE -SIZE SMI_SAVE)
```

```
SMI_CR0         EQU  ($CR0    + SMMSIZE - SIZE SMI_SAVE)
SMI_EFLAGS      EQU  ($EFLAGS + SMMSIZE - SIZE SMI_SAVE)
SMI_DR7         EQU  ($DR7    + SMMSIZE - SIZE SMI_SAVE)
```

SMM Instruction macro example: TEST.ASM

```
                              .MODEL  SMALL
                              .386
                              ;SMM Macro Examples
                              ; by Dean C. Wills

                              include smimac.inc

0000                          .DATA
0000  0A*(??)                 there   db      10 dup (?)
000A                          .CODE
0000  2E 0F 78 1E 004E                svdc    cs:,hello,ds
0006  2E 0F 79 1E 004E                rsdc    ds,cs:,hello
000C  2E 0F 79 2E 004E                rsdc    gs,cs:,hello
0012 2E 67 2E 0F 78 9C 58 0000004E
                                svdc cs:,[eax+ebx*2+hello],1
001D  67| 0F 78 23                   svdc    ,[ebx],fs,1

0021  0F 78 2E 0000                  svdc    ,there,gs
0026  2E 0F 7A 06 004E               svldt   cs:,hello
002C  2E 0F 7B 06 004E               rsldt   cs:,hello

0032  2E 0F 7D 06 004E               rsts    cs:,hello
0038 2E 67 2E 0F 7C 84 58 0000004E
                                svts cs:,[eax+ebx*2+hello],1
0043  67| 0F 7A 03                   svldt   ,[ebx],1
0047  0F 7C 06 0000                  svts    ,there
004C  0F AA                           rsm

004E  0A*(??)                         hello   db   10 dup (?)
end
```

Information furnished is believed to be accurate and reliable. However, SGS-THOMSON Microelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of SGS-THOMSON Microelectronics. Specification mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. SGS-THOMSON Microelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of SGS-THOMSON Microelectronics.

© 1995 SGS-THOMSON Microelectronics – Printed in Italy – All Rights Reserved

SGS-THOMSON Microelectronics GROUP OF COMPANIES
Australia - Brazil - China - France - Germany - Hong Kong - Italy - Japan - Korea - Malaysia - Malta - Morocco - The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - Thailand - United Kingdom - U.S.A.