Revision 0; 8/11

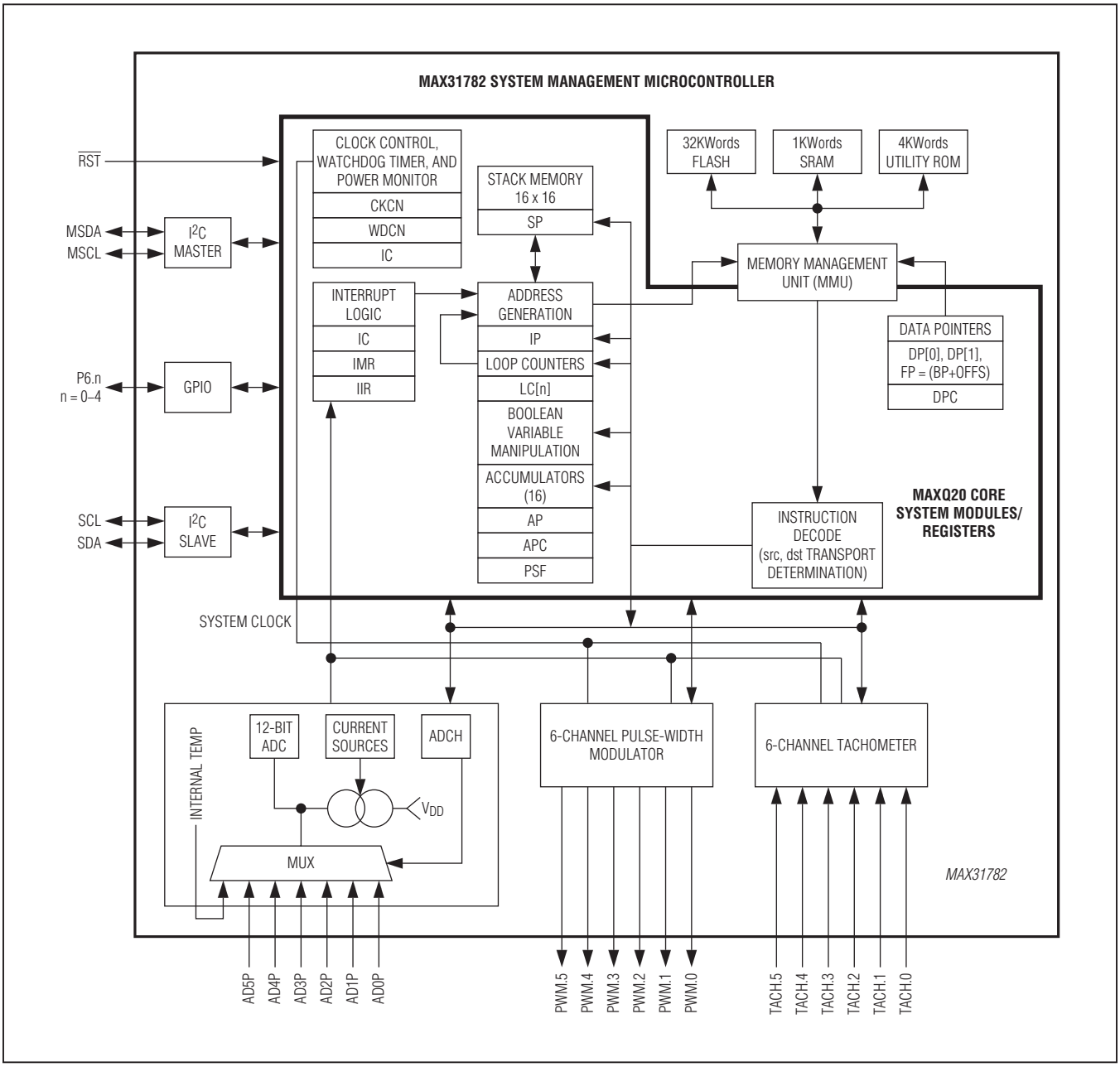
MAX31782 User's Guide

maxim integrated™

TABLE OF CONTENTS

SECTION 1: OVERVIEW

The MAX31782 system management microcontroller provides a complete solution for the monitoring and controlling of complex system physical health characteristics. The MAX31782 is based on the high-performance 16-bit family of MAXQ® reduced instruction set computing (RISC) microcontrollers. The MAX31782 provides generous amounts of flash program memory and SRAM data memory.



MAXQ is a registered trademark of Maxim Integrated Products, Inc.

Some of the resources and features that the MAX31782 provides for monitoring and controlling a complex system include the following:

- Remote temperature measurement of diode connected transistors on up to 6 channels
- Accurate voltage measurement using the 12-bit analog to digital converter (ADC) on up to 6 channels
- Internal temperature sensor
- Independent slave and master I^2C-compatable interfaces
- Six independent PWM outputs and tachometer Inputs
- Hardware multiplier unit
- 32KWords of flash and 1KWords of SRAM memory
- Included ROM routines that allow bootloading and in-application programming flash memory
- In-system debugging

This document is provided as a supplement to the MAX31782 IC data sheet. This user's guide provides the information necessary to develop applications using the MAX31782. All electrical and timing specifications, pin descriptions, package information, and ordering information can be found in the MAX31782 IC data sheet.

SECTION 2: ARCHITECTURE

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 2: ARCHITECTURE

The MAX31782 contains a MAXQ20 low-cost, high-performance, CMOS, fully static microcontroller with flash memory. It is structured on a highly advanced, 16-accumulator-based, 16-bit RISC architecture. Fetch and execution operations are completed in one cycle without pipelining, since the instruction contains both the op code and data. The highly efficient core is supported by 16 accumulators and a 16-level hardware stack, enabling fast subroutine calling and task switching.

Data can be quickly and efficiently manipulated with three internal data pointers. Two of these data pointers, DP0 and DP1, are stand-alone 16-bit pointers. The third data pointer, frame pointer, is composed of a 16-bit base pointer (BP) and an 8-bit offset register (OFFS). All three pointers support postincrement/decrement functionality for read operations and preincrement/decrement for write operations. For the frame pointer (FP = BP[OFFS]), the increment/decrement operation is executed on the OFFS register and does not affect the base pointer. Multiple data pointers allow more than one function to access data memory without having to save and restore data pointers each time.

Stack functionality is provided by dedicated memory with a 16-bit width and a depth of 16. An on-chip memory management unit (MMU) allows logical remapping of the program and data spaces, and thus facilitates in-system programming and fast access to data tables, arrays, and constants located in flash memory.

This section provides details on the following topics.

1) Instruction decoding
2) Register space
3) Memory types
4) Program and data memory mapping and access
5) Data alignment
6) Reset conditions
7) Clock generation
8) Power modes

2.1 Instruction Decoding

The MAX31782 uses the standard 16-bit MAXQ20 core instruction set, which is described in *SECTION 20: Instruction Set Summary*. Every instruction is encoded as a single 16-bit word. The instruction word format is shown in Figure 2-1.

FORMAT	DESTINATION							SOURCE								
f	d	d	d	d	d	d	d	s	s	s	s	s	s	s	s

Figure 2-1. Instruction Word Format

Bit 15 (f) indicates the format for the source field of the instruction as follows:

If f equals 0, the instruction is an immediate source instruction. The source field represents an immediate 8-bit value.

If f equals 1, the instruction is a register source instruction. The source field represents the register from which the source value is read.

Bits 14 to 8 (ddddddd) represent the destination for the transfer. This value always represents a destination register. The lower four bits contain the module specifier and the upper three bits contain the register index in that module.

Bits 7 to 0 (ssssssss) represent the source for the transfer. Depending on the value of the format field, this can either be an immediate value or a source register. If this field represents a register, the lower four bits contain the module specifier and the upper four bits contain the register index in that module.

This instruction word format presents the following limitations.

1) There are 32 registers per register module, but only 4 bits are allocated to designate the source register and only 3 bits are allocated to designate the destination register.
2) The source field only provides 8 bits of data for an immediate value; however, a 16-bit immediate value can be required.

The MAX31782 uses a prefix register (PFX) to address these limitations. The PFX register provides the additional bits required to access all 32 registers within a module. The PFX register also provides the additional 8 bits of data required to make a 16-bit immediate data source. The data that is written to the PFX register survives for only one clock cycle. This means the write to the PFX register must occur immediately prior to the instruction requiring the PFX register. The PFX register is cleared to zero after one cycle so it does not affect any other instructions. The write to the PFX register is done automatically by the assembler and requires one additional execution cycle. So, while most instructions execute in a single cycle, two cycles are needed for instructions that require the PFX register.

The architecture of the MAX31782 is transport-triggered. This means that writing to or reading from certain register locations also causes side effects to occur. These side effects form the basis of the MAX31782's higher level op codes, such as ADDC, OR, and JUMP. While these op codes are actually implemented as MOVE instructions between certain register locations, the encoding is handled by the assembler and need not be a concern to the programmer. The unused "empty" locations in the system register modules are used for these higher level op codes.

The instruction set is designed to be highly orthogonal. All arithmetic and logical operations that use two registers can use any register along with the accumulator. Data can be transferred between any two registers in a single instruction.

2.2 Register Space

The MAX31782 provides a total of 13 register modules broken up into two different groups. These groupings are descriptive only, as there is no difference between accessing the two register groups from a programming perspective. The two groups are:

1) Peripheral Registers: These are the lower six modules (Modules 0h through 5h). The peripheral registers in the MAX31782 are used for functionalities such as ADC, PWM outputs, tachometer inputs, GPIO, etc. The peripheral registers are not used to implement op codes.
2) System Registers: These are modules 8h, 9h, and Bh through Fh. The system registers in the MAX31782 are used to implement higher level op codes as well as the following common system features.
   - 16-bit ALU and associated status flags (zero, equals, carry, sign, overflow)
   - 16 working accumulator registers, each 16-bit, along with associated control registers
   - Instruction pointer
   - Registers for interrupt control, handling, and identification
   - Auto-decrementing loop counters for fast, compact looping
   - Two data pointer registers and a frame pointer for data memory access

Each system register module has 16 registers, while each peripheral register module has 32 registers. The number of cycles required to access a particular register depends upon the register's index within the module. The access times based upon the register index are grouped as follows:

- The first eight registers (index 0h to 7h) in each module can be read from or written to in a single cycle.
- The second eight registers (index 8h to 0Fh) can be read from in a single cycle and written to in two cycles (by using the PFX register).
- The last 16 registers (10h to 1Fh) in peripheral register modules can be read or written in two cycles (always requiring use of the PFX register).

Registers can be 8 or 16 bits in length. Some registers can contain reserved bits. The user should not write to any reserved bits. Data transfers between registers of different sizes are handled as shown in Table 2-1.

- If the source and destination registers are both 8 bits wide, data is copied bit to bit.
- If the source register is 8 bits wide and the destination register is 16 bits wide, the data from the source register is transferred into the lower 8 bits of the destination register. The upper 8 bits of the destination register are set to the current value of the PFX register; this value is normally zero, but it can be set to a different value by the previous instruction if needed. The PFX register reverts back to zero after one cycle, so this must be done by the instruction immediately before the one that is using the value.
- If the source register is 16 bits wide and the destination register is 8 bits wide, the lower 8 bits of the source are transferred to the destination register.
- If both registers are 16 bits wide, data is copied bit to bit.

The above rules apply to all data movements between defined registers. Data transfer to/from undefined register locations has the following behavior:

- If the destination is an undefined register, the MOVE is a dummy operation but can trigger an underlying operation according to the source register (e.g., @DPn--).
- If the destination is a defined register and the source is undefined, the source data for the transfer depends upon the source module width. If the source is from a module containing 8-bit or 8-bit and 16-bit source registers, the source data is equal to the prefix data as the upper 8 bits and 00h as the lower 8 bits. If the source is from a module containing only 16-bit source registers, 0000h source data is used for the transfer.

Table 2-1. Register-to-Register Transfer Operations

SOURCE REGISTER SIZE (BITS)	DESTINATION REGISTER SIZE (BITS)	PREFIX SET?	DESTINATION SET TO VALUE	
			HIGH 8 BITS	LOW 8 BITS
8	8	X	—	Source [7:0]
8	16	No	00h	Source [7:0]
8	16	Yes	PFX [7:0]	Source [7:0]
16	8	X	—	Source [7:0]
16	16	X	Source [15:8]	Source [7:0]

2.3 Memory Types

In addition to the internal register space, the MAX31782 incorporates the following memory types:

- 32KWords of flash memory
- 1KWords of SRAM
- 4KWords of utility ROM contain a debugger and program loader
- 16-level stack memory for storage of program return addresses and general-purpose use

The memory on the MAX31782 is organized according to a Harvard architecture. This means that there are separate busses for both program and data memory. Stack memory is also separate and is accessed through a dedicated register set.

2.3.1 Flash Memory

The MAX31782 contains 32KWords (32K x 16) of flash memory. The flash memory begins at address 0000h and is contiguous through word address 7FFFh. The flash memory can also be used for storing lookup tables and other nonvolatile data.

The incorporation of flash memory allows the contents of the flash memory to be upgraded in the field, either by the application or by one of the bootloaders (JTAG or I^2C). Writing to flash memory must be done indirectly by using routines that are provided by the utility ROM. See *SECTION 21: Utility ROM* and *SECTION 18: In-System Programming* for more details.

2.3.2 SRAM Memory

The MAX31782 contains 1KWords (1K x 16) of SRAM memory. The SRAM memory address begins at address 0000h and is contiguous through word address 03FFh. The contents of the SRAM are indeterminate after power-on reset, but are maintained during stop mode and non-POR resets.

When using the in-circuit debugging features, the highest 19 bytes of the SRAM must be reserved for saved state storage and working space for the debugging routines. If in-circuit debug is not used, the entire 1KWords of SRAM is available for application use.

2.3.3 Utility ROM

The utility ROM is a 4KWord segment of memory. The utility ROM memory address begins at word address 8000h and is contiguous through word address 8FFFh. The utility ROM is programmed at the factory and cannot be modified. The utility ROM provides the following system utility functions:

- Reset vector (not user code reset vector)
- In-system programming (bootstrap loader) over JTAG or I^2C-compatible interfaces
- In-circuit debug routines
- Routines for in-application flash programming

Following any reset, the MAX31782 automatically starts execution at the reset vector, which is address 8000h in the utility ROM. The ROM code determines whether the program execution should immediately jump to the start of application code (flash address 0000h), or to one of the special routines mentioned. Routines within the utility ROM are firmware-accessible and can be called as subroutines by the application software. See *SECTION 21: Utility ROM*, *SECTION 18: In-System Programming*, and *SECTION 17: In-Circuit Debug Mode* for more information on the routines provided by the utility ROM.

2.3.4 Stack Memory

A 16-bit, 16-level on-chip stack provides storage for program return addresses and general-purpose use. The stack is used automatically by the processor when the CALL, RET, and RETI instructions are executed, and when an interrupt is serviced. The stack can also be used explicitly to store and retrieve data by using the @SP- - source, @++SP destination, or the PUSH, POP, and POPI instructions. The POPI instruction acts identically to the POP instruction except that it additionally clears the INS bit.

The width of the stack is 16 bits to accommodate the instruction pointer size. On reset, the stack pointer SP initializes to the top of the stack (0Fh). The CALL, PUSH, and interrupt vectoring operations first increment SP and then store a value at @SP. The RET, RETI, POP, and POPI operations first retrieve the value at @SP and then decrement SP.

The stack memory is initialized to indeterminate values upon reset or power-up. Stack memory is dedicated for stack operations only and cannot be accessed by the MAX31782 program or data busses.

When using the in-circuit debugging features, one word of the stack must be reserved for the debugging routines. If in-circuit debug is not used, the entire stack is available for application use.

2.4 Program and Data Memory Mapping and Access

The memory on the MAX31782 is implemented using a Harvard architecture, with separate buses for program and data memory. The memory management unit (MMU) allows the MAX31782 to also support a pseudo-Von Neumann memory map. The pseudo-Von Neumann memory map allows each of the memory segments (flash, SRAM, and utility ROM) to be logically mapped into a single contiguous memory map. This allows all the memory segments to be accessed as both program and memory data. The advantages the pseudo-Von Neumann memory map provides are:

- Program execution can occur from the flash, SRAM, or utility ROM memory segments.
- The SRAM and flash memory segments can both be used for data memory.

Using the pseudo-Von Neumann memory map does have one restriction. This restriction is that a particular memory segment cannot be simultaneously accessed as both program and data memory.

2.4.1 Program Memory Access

The instructions that the MAX31782 is executing reside in what is defined as the program memory. The MMU fetches the instructions using the program bus. The instruction pointer (IP) register designates the program memory address of the next instruction to fetch. The IP register is read/write accessible by the user software. A write to the IP register forces program flow to the new address on the next cycle following the write. The content of the IP register is incremented by 1 automatically after each fetch operation. From an implementation perspective, system interrupts and branching instructions simply change the contents of the IP register and force the op code to fetch from a new program location.

2.4.2 Program Memory Mapping

The MAX31782's mapping of the three memory segments (flash, SRAM, and utility ROM) as program memory is shown in Figure 2-2. The mapping of memory segments into program space is always the same. When referring to memory as program memory, all addresses are given as word addresses. The 32KWord flash memory segment is located at memory location 0000h through 7FFFh and is logically divided into two pages, each containing 16KWords. The utility ROM is located from location 8000h through 8FFFh, followed by the SRAM memory segment at location A000h through A3FFh. The user code reset vector, which is the first instruction of user program code that is executed, is located at flash memory address 0000h. User program code should always begin at this address.

2.4.3 Data Memory Access

Data memory mapping and access control are handled by the memory management unit (MMU). Read/write access to data memory can be in word or in byte mode. The MAX31782 provides three pointers that can be used for indirect accessing of data memory. The MAX31782 has two data pointers (@DPn) and one frame pointer (@BP[OFFS]). These pointers are implemented as registers that can be directly accessed by user software. A data memory access requires only one system clock period.

2.4.3.1 Data Pointers

To access data memory, the data pointers are used as one of the operands in a MOVE instruction. If the data pointer is used as a source, the core performs a load operation that reads data from the memory location addressed by the data pointer. If the data pointer is used as destination, the core performs a store operation that writes data to the memory location addressed by the data pointer. Following are some examples of setting and using a data pointer.

```
move DP[0], #0100h       ; set pointer DP[0] to address 100h
move Acc, @DP[0]         ; read data from location 100h
move @DP[0], Acc         ; write to location 100h
```

The address pointed to by the data pointers can be automatically incremented or decremented. If the data pointer is used as a source, the pointer can be incremented or decremented after the data access. If the data pointer is used as a destination, the increment or decrement can occur prior to the data access. Following are examples of using the data pointers increment/decrement features.

```
move Acc, @DP[0]++       ; increment DP[0] after read
move Acc, @DP[1]--       ; decrement DP[1] after read
move @++DP[0], Acc       ; increment DP[0] before write
move @--DP[1], Acc       ; decrement DP[0] before write
```

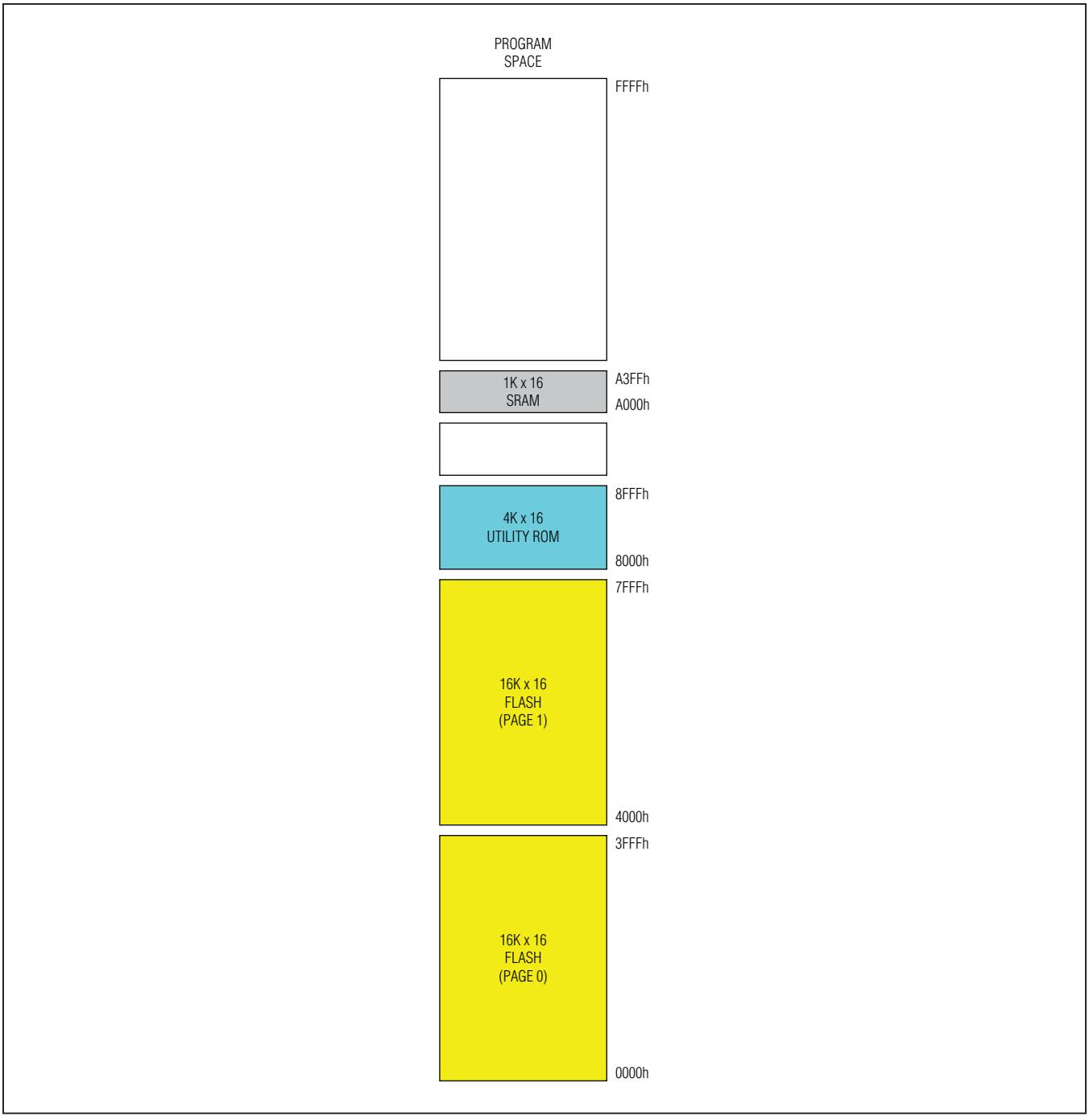


Figure 2-2. Program Memory Mapping

2.4.3.2 Frame Pointer

The frame pointer (BP[OFFS]) is formed by the 16-bit unsigned addition of the 16-bit frame pointer base register (BP) and the 8-bit frame pointer offset register (OFFS). The method the MAX31782 uses to access data using the frame pointer is similar to the data pointers. When increments or decrements are used, only the value of OFFS is incremented or decremented. The base pointer (BP) remains unaffected by increments or decrements of the OFFS register, including when the OFFS register rolls over from FFh to 00h or from 00h to FFh. Following are examples of how to use the frame pointer.

```
move BP, #0100h          ; set base pointer to address 100h
move OFFS, #10h          ; set the offset to 10h,
move Acc, @BP[OFFS]      ; read data from location 0110h
move @BP[OFFS], Acc      ; write data to location 0110h
move Acc, @BP[OFFS++]    ; increment OFFS after read
move Acc, @BP[OFFS++]    ; decrement OFFS after read
move @BP[++OFFS], Acc    ; increment OFFS before write
move @BP[--OFFS], Acc    ; decrement OFFS before write
```

2.4.4 Data Memory Mapping

The MAX31782's pseudo-Von Neumann memory map allows the MMU to read data from each of the three memory segments (flash, SRAM, utility ROM). The MMU can also write data directly to the SRAM memory segment. Data memory can be written to the flash memory segment, but because writing to flash requires the use of the utility ROM routines, this is not a direct access. The logical mapping of the three memory segments as data memory varies depending upon:

- from which memory segment instructions are currently being executed
- if data memory is being accessed in word or byte mode

In all cases, whichever memory segment is currently being used as program memory cannot be accessed as data memory.

When the program is currently executing instructions from either the SRAM or utility ROM memory segments, the flash memory is mapped to half the data memory space. If word access mode is selected, both pages (32KWords) can be logically mapped to data memory space. If byte access mode is selected, only one page (32KB) can be logically mapped to half of the data memory space. When operating in byte access mode, the selection of which flash page is mapped into data memory space is determined by the code data access bit (CDA0):

CDA0	SELECTED PAGE IN BYTE MODE	SELECTED PAGE IN WORD MODE
0	P0	P0 and P1
1	P1	P0 and P1

The next three sections detail the mapping of the different memory segments as data memory depending upon from which memory segment instructions are currently being executed.

2.4.4.1 Memory Map When Executing from Flash Memory

When executing from the flash memory:

- Read and write operations of SRAM memory are executed normally.
- The utility ROM can be read as data, starting at 8000h of the data space. The utility ROM cannot be written.

Figure 2-3 illustrates the mapping of the SRAM and utility ROM memory segments into data memory space when code is executing from the flash memory segment.

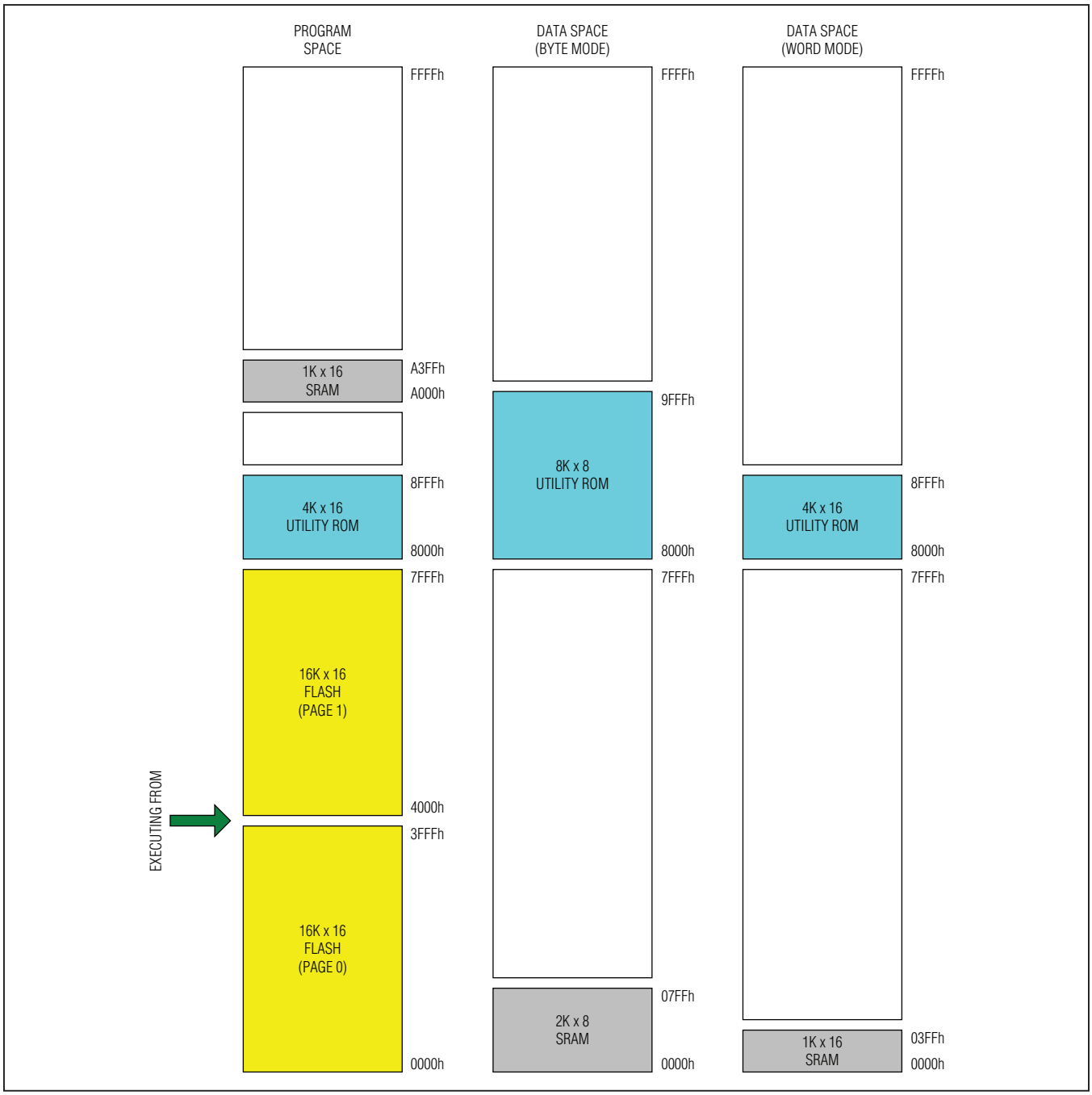


Figure 2-3. Memory Map When Executing from Flash Memory

2.4.4.2 Memory Map When Executing from Utility ROM

When executing from the utility ROM:

- Read and write operations of SRAM memory are executed normally.
- Reading of flash memory is executed normally. Writing to flash memory requires the use of the utility ROM routines.
- One page (byte access mode) or both pages (word access mode) of the flash memory can be accessed as data with an offset of 8000h as determined by the CDA0 bit.

Figure 2-4 illustrates the mapping of the SRAM and flash memory segments into data memory space when code is executing from the utility ROM memory segment.

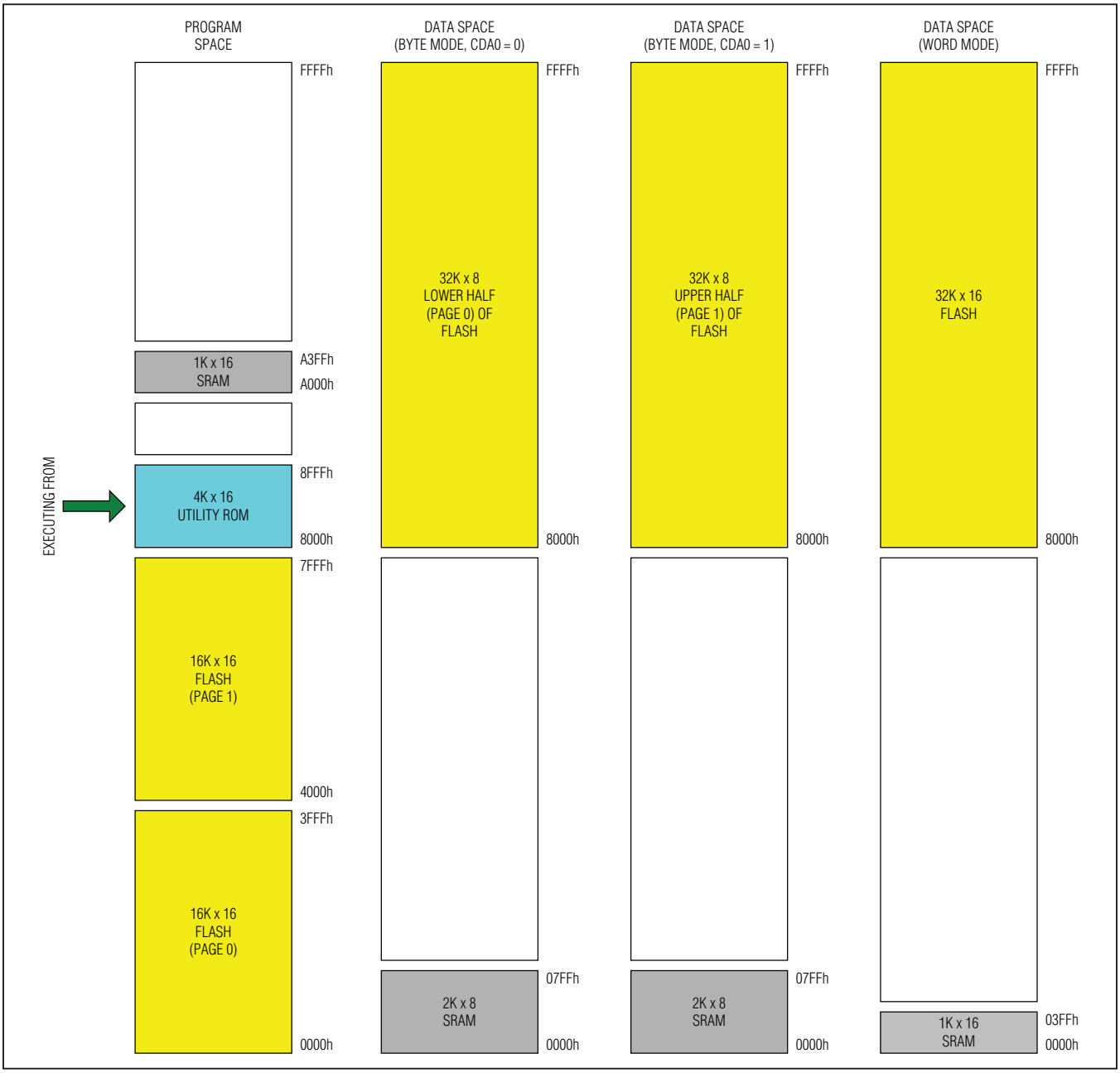


Figure 2-4. Memory Map When Executing from Utility ROM

2.4.4.3 Memory Map When Executing from SRAM

When executing from the SRAM:

The utility ROM can be read as data, starting at 8000h of the data space. The utility ROM cannot be written.

Reading of flash memory is executed normally. Writing to flash memory requires the use of the utility ROM routines.

One page (byte access mode) or both pages (word access mode) of the flash memory can be accessed as data with an offset of 0000h. For byte access mode, the page of flash accessed is determined by the CDA0 bit.

Figure 2-5 illustrates the mapping of the flash and utility ROM memory segments into data memory space when code is executing from the SRAM memory segment.

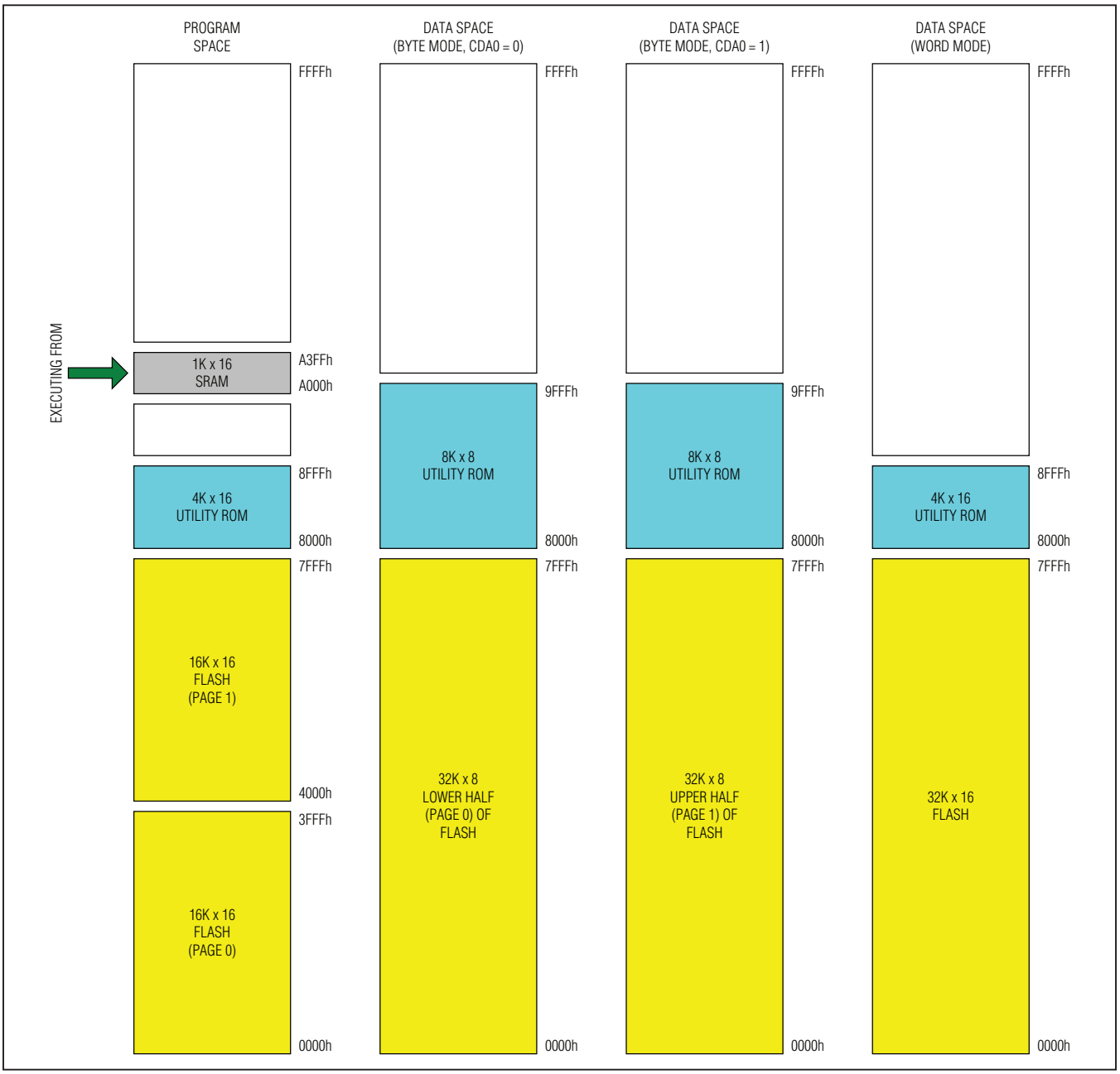


Figure 2-5. Memory Map When Executing from SRAM

2.5 Data Alignment

To support merged program and data memory operation while maintaining efficient memory space usage, the data memory must be able to support both byte and word mode accessing. Data is aligned in data memory as words, but the effective data address is resolved to bytes. This data alignment allows program instruction fetching in words while maintaining data accessibility at the byte level. It is important to realize that this accessibility requires strict word alignment. All executable or data words must align to an even address in byte mode. Care must be taken when updating a code segment as misalignment of words likely results in loss of program execution control.

Memory is always read as a complete word, whether for program fetch or data access. The program decoder always uses a full 16-bit word. The data access can utilize a word or an individual byte. Data memory is organized as two byte-wide memory banks with common word address decode but two 8-bit data buses. In byte mode, data pointer hardware reads out the full word containing the selected byte using the effective data word address pointer (the least significant bit of the byte data pointer is not initially used). Then, the least significant data pointer bit functions as the byte select that is used to place the correct byte on the data bus. For write access, data pointer hardware addresses a particular word using the effective data word address while the least significant bit selects the corresponding data bank for write. The contents of the other byte are left unaffected.

2.6 Reset Conditions

The MAX31782 has several possible sources of reset:

- Power-On/Brownout Reset
- Watchdog Timer Reset
- External Reset
- Internal System Reset

Once a reset condition has completed or been removed, code execution begins at the beginning of utility ROM, which is address 8000h. The utility ROM code interrogates the I2C_SPE, JTAG_SPE, and PWL bits to determine if bootloading is necessary. If bootloading is not required, execution jumps to the user code reset vector, which is at flash memory address 0000h.

The $\overline{RST}$ pin is an output as well as an input. If a reset condition is generated by one of the MAX31782's internal reset sources (brownout, watchdog timer, or internal reset), an output reset pulse is generated on the $\overline{RST}$ pin while the MAX31782 remains in reset.

2.6.1 Power-On/Brownout Reset

The MAX31782 provides a power-on reset (POR) circuit to ensure proper initialization of internal device states and analog circuits. The POR voltage threshold range is between approximately 1.1V and 1.7V. When V_{DD} is below the POR level, the state of all the MAX31782 pins, including $\overline{RST}$, is indeterminate.

The MAX31782 also includes brownout detection capability. This is an on-chip precision reference and comparator that monitors the supply voltage, V_{DD}, to ensure that it is within acceptable limits. If V_{DD} is below the brownout level (V_{BO}), the power monitor generates a reset. This can occur when:

- The MAX31782 is being powered up and V_{DD} is above the POR level but still less than V_{BO}.
- V_{DD} drops from an acceptable level to less than V_{BO}.

Once V_{DD} exceeds V_{BO}, the MAX31782 exits the reset condition and the internal oscillator starts up. After approximately 1000 clock cycles ($t_{SU:MOSC}$) the MAX31782 performs the following tasks.

- All registers and circuits enter their reset state.
- The POR flag in the watchdog control register (WDCN) is set to indicate the source of the reset.
- The MAX31782 begins normal operation (CPU state).
- Code execution begins at utility ROM location 8000h.

The transition between POR, brownout, and normal operation is detailed in Figure 2-6. **Note:** If V_{DD} is below V_{BO}, there is a chance that the SRAM was corrupted. If the POR flag in WDCN is set, all data in SRAM should be reinitialized.

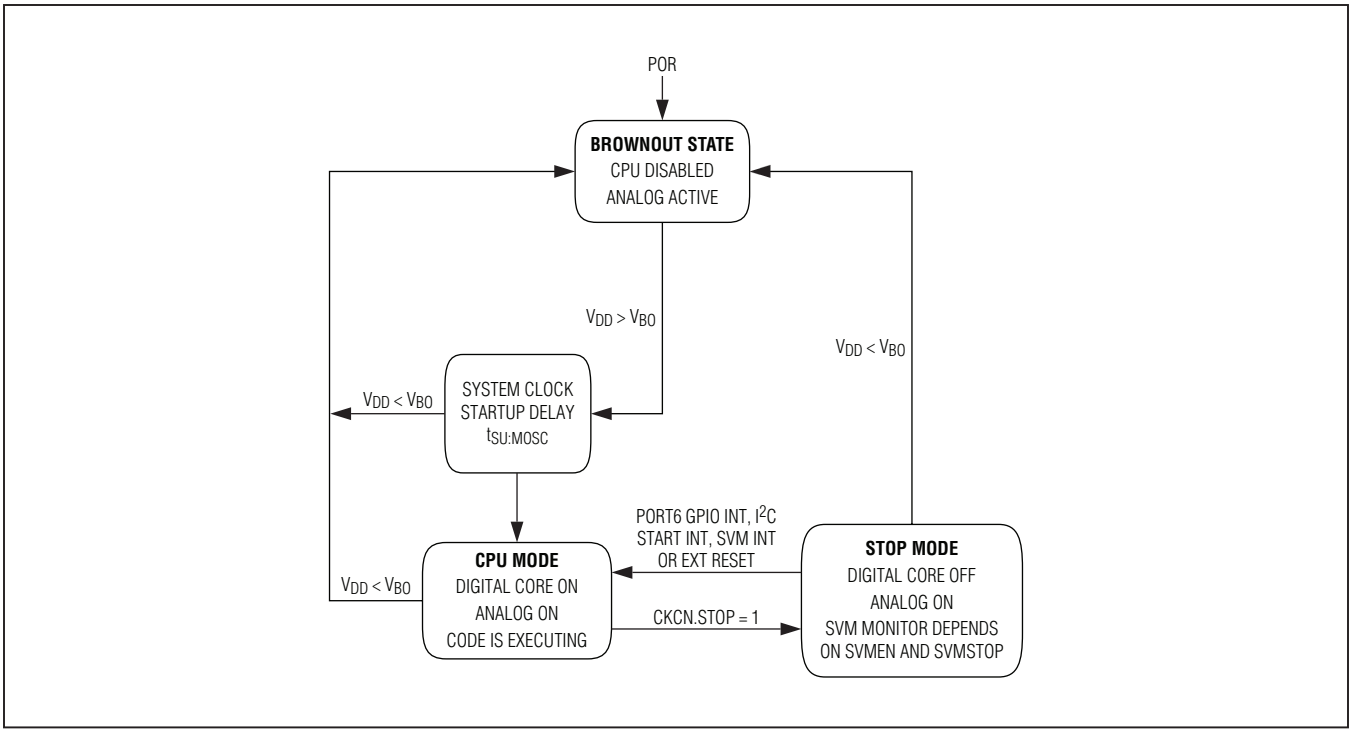


Figure 2-6. MAX31782 State Diagram

2.6.2 Watchdog Timer Reset

The watchdog timer is a programmable hardware timer that can be used to reset the processor in case a software lockup or other unrecoverable error occurs. Once the watchdog is enabled, software must reset the watchdog timer periodically. If the processor does not reset the watchdog timer before it elapses, the watchdog can initiate a reset.

If the watchdog resets the processor, the MAX31782 remains in reset and holds the $\overline{RST}$ pin low for 12 clock cycles. When a reset occurs due to a watchdog timeout, the watchdog timer reset flag (WTRF) in the WDCN register is set to indicate the source of the reset.

2.6.3 External Reset

During normal operation, the MAX31782 is placed into external reset when the $\overline{RST}$ pin is held at logic 0 for at least four clock cycles. Once the MAX31782 enters reset mode, it remains in reset as long as the $\overline{RST}$ pin is held at logic 0. After the $\overline{RST}$ pin returns to logic 1, the processor exits reset within 12 clock cycles.

An external reset pulse on the $\overline{RST}$ pin can also bring the MAX31782 out of its low-power stop mode. When this occurs, the MAX31782 resets and returns to normal CPU mode operation within 10 clock cycles.

2.6.4 Internal System Resets

There are two possible sources of internal system resets. An internal reset holds the MAX31782 in reset mode for 12 clock cycles.

1) When data BBh is written to the special I^2C slave address 34h.
2) When in-system programming is complete and the ROD bit is set to 1.

2.7 Clock Generation

The MAX31782 generates its 4MHz instruction clock using an internal oscillator. This oscillator starts up when V_{DD} exceeds the brownout voltage level, V_{BO}. There is a delay of approximately 1000 clock cycles ($t_{SU:MOSC}$) between when the oscillator starts and when clocking of the MAX31782 begins. This delay ensures that the clock is stable prior to beginning normal operation.

2.8 Power Modes

The MAX31782 has two modes of operation. These two modes of operation are detailed in the state diagram as shown in Figure 2-6.

1) Normal CPU mode
2) Stop mode

The MAX31782 enters stop mode when the STOP bit in the system clock control register (CKCN) is set. Upon entering stop mode, the digital core is no longer clocked, thus making the core inactive. In stop mode, the ADC is also disabled and the Supply Voltage Monitor (SVM) can be disabled. The internal oscillator, brownout detection, and regulators (REG18 and REG25 pins) remain active during stop mode. Table 2-2 details the state of the MAX31782's analog and digital blocks during the different modes of operation.

Table 2-2. State of Circuits During Different Modes

CKCN.STOP	SVM.SVMEN	SVM.SVMSTOP	POWER MODE	CPU	REGULATORS		INTERNAL OSCILLATOR	BROWNOUT DETECTION	SVM MONITOR	ADC
					1.8V	2.5V				
0	0	X	CPU Mode	On	On	On	On	On	Off	On/Off
0	1	X	CPU Mode	On	On	On	On	On	On	On/Off
1	0	X	Stop Mode	Off	On	On	On	On	Off	Off
1	1	0	Stop Mode	Off	On	On	On	On	Off	Off
1	1	1	Stop Mode	Off	On	On	On	On	On	Off

The MAX31782 exits stop mode when any of the following interrupt conditions occurs:

- GPIO interrupt from Port 6
- I^2C START interrupt
- SVM interrupt
- External reset

The interrupt sources listed must be enabled prior to entering stop mode if they are going to be used to bring the MAX31782 out of stop mode. After receiving one of these interrupts, the MAX31782 exits stop mode and returns to CPU mode within 10 system clock cycles. If an interrupt causes the system to come out of stop mode, the program execution starts from the point where stop mode was asserted. However, if an external reset is used to come out of stop mode, the program execution begins from utility ROM location 8000h.

SECTION 3: SYSTEM REGISTER DESCRIPTIONS

This section contains the following information:

LIST OF TABLES

SECTION 3: SYSTEM REGISTER DESCRIPTIONS

Most MAX31782 functions are controlled by sets of registers. These registers provide a working space for memory operations as well as configuring and addressing peripheral registers on the device. Registers are divided into two major types: system registers and peripheral registers. The common register set, also known as the system registers, includes ALU access and control registers, accumulator registers, data pointers, interrupt vectors and control, and stack pointer. The peripheral registers define additional functionality and the functionality is broken up into discrete modules.

This section describes the MAX31782's system registers. Table 3-1 shows the MAX31782 system register map. Table 3-2 explains system register bit functions. This is followed by a detailed bit description.

Table 3-1. System Register Map

REGISTER INDEX	REGISTER MODULE						
	AP (8h)	A (9h)	PFX (Bh)	IP (Ch)	SP (Dh)	DPC (Eh)	DP (Fh)
00h	AP	A[0]	PFX[0]	IP	—	—	—
01h	APC	A[1]	PFX[1]	—	SP	—	—
02h	—	A[2]	PFX[2]	—	IV	—	—
03h	—	A[3]	PFX[3]	—	—	OFFS	DP[0]
04h	PSF	A[4]	PFX[4]	—	—	DPC	—
05h	IC	A[5]	PFX[5]	—	—	GR	—
06h	IMR	A[6]	PFX[6]	—	LC[0]	GRL	—
07h	—	A[7]	PFX[7]	—	LC[1]	BP	DP[1]
08h	SC	A[8]	—	—	—	GRS	—
09h	—	A[9]	—	—	—	GRH	—
0Ah	—	A[10]	—	—	—	GRXL	—
0Bh	IIR	A[11]	—	—	—	FP	—
0Ch	—	A[12]	—	—	—	—	—
0Dh	—	A[13]	—	—	—	—	—
0Eh	CKCN	A[14]	—	—	—	—	—
0Fh	WDCN	A[15]	—	—	—	—	—

Table 3-2. System Register Bit Functions

REGISTER	BIT															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AP									—	—	—	—	AP (4 bits)			
APC									CLR	IDS	—	—	—	MOD2	MOD1	MOD0
PSF									Z	S	—	GPF1	GPF0	OV	C	E
IC									—	—	—	—	—	—	INS	IGE
IMR									IMS	—	IM5	IM4	IM3	IM2	IM1	IM0
SC									TAP	—	—	CDA0	—	ROD	PWL	—
IIR									IIS	—	II5	II4	II3	II2	II1	II0
CKCN									—	—	—	STOP	—	—	—	—
WDCN									POR	EWDI	WD1	WD0	WDIF	WTRF	EWT	RWT
A[n] (n = 15:0)	A[n] (16 bits)															
PFX[n] (n = 7:0)	PFX[n] (16 bits)															
IP	IP (16 bits)															
SP	—	—	—	—	—	—	—	—	—	—	—	—	SP (4 bits)			
IV	IV (16 bits)															
LC[0]	LC[0] (16 bits)															
LC[1]	LC[1] (16 bits)															
OFFS									OFFS (8 bits)							
DPC	—	—	—	—	—	—	—	—	—	—	—	WBS2	WBS1	WBS0	SDPS1	SDPS0
GR	GR (16 bits)															
GRL									GRL (8 bits)							
BP	BP (16 bits)															
GRS	GRS (16 bits) = (GRL:GRH)															
GRH									GRH (8 bits)							
GRXL	GRXL (16 bits) = (GRL.7, 8 bits): (GRL, 8 bits)															
FP	FP = BP[OFFS] (16 bits)															
DP[0]	DP[0] (16 bits)															
DP[1]	DP[1] (16 bits)															

3.1 System Register Bit Descriptions

3.1.1 Accumulator Pointer Register (AP, 8h[0h])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
AP.[3:0]	Active Accumulator Select. These bits select which of the 16 accumulator registers are used for arithmetic and logical operations. If the APC register has been set to perform automatic increment/decrement of the active accumulator, this setting is automatically changed after each arithmetic or logical operation. If a 'MOVE AP, Acc' instruction is executed, any enabled AP inc/dec/modulo control takes precedence over the transfer of Acc data into AP.
AP.[7:4]	Reserved. All reads return 0.

3.1.2 Accumulator Pointer Control Register (APC, 8h[1h])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Unrestricted direct read/write access.

<table>
<tr><th>BIT</th><th colspan="2">FUNCTION</th></tr>
<tr><td rowspan="8">APC.[2:0]
(MOD[2:0])</td><td colspan="2">Accumulator Pointer Auto Increment/Decrement Modulus. If these bits are set to a nonzero value, the accumulator pointer (AP[3:0]) is automatically incremented or decremented following each arithmetic or logical operation. The mode for the auto increment/decrement is determined as follows:</td></tr>
<tr><td>MOD[2:0]</td><td>AUTO INCREMENT/DECREMENT MODE</td></tr>
<tr><td>000</td><td>No auto increment/decrement (default)</td></tr>
<tr><td>001</td><td>Increment/decrement AP[0] modulo 2</td></tr>
<tr><td>010</td><td>Increment/decrement AP[1:0] modulo 4</td></tr>
<tr><td>011</td><td>Increment/decrement AP[2:0] modulo 8</td></tr>
<tr><td>100</td><td>Increment/decrement AP modulo 16</td></tr>
<tr><td>101 to 111</td><td>Reserved (modulo 16 when set)</td></tr>
<tr><td>APC.[5:3]</td><td colspan="2">Reserved. All reads return 0.</td></tr>
<tr><td>APC.6 (IDS)</td><td colspan="2">Increment/Decrement Select. If this bit is set to 0, the accumulator pointer AP is incremented following each arithmetic or logical operation according to MOD[2:0]. If this bit is set to 1, the accumulator pointer AP is decremented following each arithmetic or logical operation according to MOD[2:0]. If MOD[2:0] is set to 000, the setting of this bit is ignored.</td></tr>
<tr><td>APC.7 (CLR)</td><td colspan="2">AP Clear. Writing this bit to 1 clears the accumulator pointer AP to 0. Once set, this bit is automatically reset to 0 by hardware. If a 'MOVE APC, Acc' instruction is executed requesting that AP be set to 0 (i.e., CLR = 1), the AP clear function overrides any enabled inc/dec/modulo control. All reads from this bit return 0.</td></tr>
</table>

3.1.3 Processor Status Flags Register (PSF, 8h[4h])

Initialization: This register is cleared to 80h on all forms of reset.

Access: Bit 7 (Z), bit 6 (S), and bit 2 (OV) are read-only. Bits 4 and 3 (GPF1, GPF0), bit 1 (C), and bit 0 (E) are unrestricted read/write.

BIT	FUNCTION
PSF.0 (E)	Equals Flag. This bit flag is set to 1 whenever a compare operation (CMP) returns an equal result. If a CMP operation returns not equal, this bit is cleared.
PSF.1 (C)	Carry Flag. This bit flag is set to 1 whenever an add or subtract operation (ADD, ADDC, SUB, SUBB) returns a carry or borrow. This bit flag is cleared to 0 whenever an add or subtract operation does not return a carry or borrow. Many other instructions potentially affect the carry bit.
PSF.2 (OV)	Overflow Flag. This flag is set to 1 if there is a carry out of bit 14 but not out of bit 15, or a carry out of bit 15 but not out of bit 14 from the last arithmetic operation, otherwise, the OV flag remains as 0. OV indicates a negative number resulted as the sum of two positive operands, or a positive sum resulted from two negative operands.
PSF.3 (GPF0)	General-Purpose Flag 0
PSF.4 (GPF1)	General-Purpose Flag 1. These general-purpose flag bits are provided for user software control.
PSF.5	Reserved. All reads return 0.
PSF.6 (S)	Sign Flag. This bit flag mirrors the current value of the high bit of the active accumulator (Acc.15).
PSF.7 (Z)	Zero Flag. The value of this bit flag equals 1 whenever the active accumulator is equal to zero, and it equals 0 otherwise.

3.1.4 Interrupt and Control Register (IC, 8h[5h])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
IC.0 (IGE)	Interrupt Global Enable. This bit enables the interrupt handler if set to 1. No interrupt to the CPU is allowed if this bit is cleared to 0.
IC.1 (INS)	Interrupt In Service. The INS is set by the interrupt handler automatically when an interrupt is acknowledged. No further interrupts occur as long as the INS bit remains set. The interrupt service routine can clear the INS bit to allow interrupt nesting. Otherwise, the INS bit is cleared automatically by the interrupt handler upon execution of an RETI/POPI instruction.
IC.[7:2]	Reserved. All reads return 0.

3.1.5 Interrupt Mask Register (IMR, 8h[6h])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Unrestricted read/write access.

The first six bits in this register are interrupt mask bits for modules 0 to 5, one bit per module. The eighth bit, IMS, serves as a mask for any system module interrupt sources. Setting a mask bit allows the enabled interrupt sources for the associated module or system (for the case of IMS) to generate interrupt requests. Clearing the mask bit effectively disables all interrupt sources associated with that specific module or all system interrupt sources (for the case of IMS). The interrupt mask register is intended to facilitate user-definable interrupt prioritization.

BIT	FUNCTION
IMR.0 (IM0)	Interrupt Mask for Register Module 0
IMR.1 (IM1)	Interrupt Mask for Register Module 1
IMR.2 (IM2)	Interrupt Mask for Register Module 2
IMR.3 (IM3)	Interrupt Mask for Register Module 3
IMR.4 (IM4)	Interrupt Mask for Register Module 4
IMR.5 (IM5)	Interrupt Mask for Register Module 5
IMR.6	Reserved. Reads return 0.
IMR.7 (IMS)	Interrupt Mask for System Modules

3.1.6 System Control Register (SC, 8h[8h])

Initialization: This register is reset to 100000s0b on all reset. Bit 1 (PWL) is set to 1 on a power-on reset only.

Access: Unrestricted read/write access.

<table>
<tr><th>BIT</th><th colspan="3">FUNCTION</th></tr>
<tr><td>SC.0</td><td colspan="3">Reserved. All reads return 0.</td></tr>
<tr><td>SC.1 (PWL)</td><td colspan="3">Password Lock. This bit defaults to 1 on a power-on reset. When this bit is 1, it requires a 32-byte password to be matched with the password in the program space before allowing access to the ROM loader's utilities for read/write of program memory and debug functions. Clearing this bit to 0 disables the password protection to the ROM loader.</td></tr>
<tr><td>SC.2 (ROD)</td><td colspan="3">ROM Operation Done. This bit is used to signify completion of a ROM operation sequence to the control units. This allows the debug engine to determine the status of a ROM sequence. Setting this bit to logic 1 causes an internal system reset if the JTAG_SPE bit is also set. Setting the ROD bit clears the JTAG_SPE bit if it is set and the ROD bit is automatically cleared by hardware once the control unit acknowledges the done indication.</td></tr>
<tr><td>SC.3</td><td colspan="3">Reserved. All reads return 0.</td></tr>
<tr><td rowspan="4">SC.4 (CDA0)</td><td colspan="3">Code Data Access Bit 0. The CDA bit is used to logically map physical program memory page to the data space for read/write access:</td></tr>
<tr><td>CDA0</td><td>BYTE MODE ACTIVE PAGE</td><td>WORD MODE ACTIVE PAGE</td></tr>
<tr><td>0</td><td>P0</td><td>P0 and P1</td></tr>
<tr><td>1</td><td>P1</td><td>P0 and P1</td></tr>
<tr><td colspan="3">The logical addresses depend on which memory segment is executing. Note that CDA1 (normally at bit position SC.5) is not implemented since the maximum flash memory size is 64KB or 32KWords.</td></tr>
<tr><td>SC.[6:5]</td><td colspan="3">Reserved. All reads return 0.</td></tr>
<tr><td>SC.7 (TAP)</td><td colspan="3">Test Access (JTAG) Port Enable. This bit controls whether the test access port special-function pins are enabled. The TAP defaults to being enabled. Clearing this bit to 0 disables the TAP special-function pins on the JTAG pins.</td></tr>
</table>

3.1.7 Interrupt Identification Register (IIR, 8h[Bh])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Read-only.

The first six bits in this register indicate interrupts pending in modules 0 to 5, one bit per module. The eighth bit, IIS, indicates a pending system interrupt, such as from the watchdog timer. The interrupt pending flags are set only for enabled interrupt sources waiting for service. The interrupt pending flag is cleared when the pending interrupt sources within that module are disabled or when the interrupt flags are cleared by software.

BIT	FUNCTION
IIR.0 (II0)	Interrupt Identifier Flag for Register Module 0
IIR.1 (II1)	Interrupt Identifier Flag for Register Module 1
IIR.2 (II2)	Interrupt Identifier Flag for Register Module 2
IIR.3 (II3)	Interrupt Identifier Flag for Register Module 3
IIR.4 (II4)	Interrupt Identifier Flag for Register Module 4
IIR.5 (II5)	Interrupt Identifier Flag for Register Module 5
IIR.6	Reserved. Reads return 0.
IIR.7 (IIS)	Interrupt Identifier Flag for System Modules

3.1.8 System Clock Control Register (CKCN, 8h[Eh])

Initialization: This register is cleared to 10h on all forms of reset.

Access: Unrestricted read/write access.

BIT	FUNCTION
CKCN.[3:0]	Reserved. All reads return 0.
CKCN.4 (STOP)	Stop Mode Select. Setting this bit to 1 stops program execution and commences low-power operation. This bit is cleared by a reset or any of the enabled external interrupts. Setting and resetting the STOP bit does not change the system clock source and its divide ratio.
CKCN.[6:5]	Reserved. All reads return 0.
CKCN.7	Reserved. All reads return 1.

3.1.9 Watchdog Control Register (WDCN, 8h[Fh])

Initialization: Bits 5, 4, 3, and 0 are cleared to 0 on all forms of reset; for others, see individual bit descriptions.

Access: Unrestricted direct read/write access.

<table>
<tr><th>BIT</th><th colspan="4">FUNCTION</th></tr>
<tr><td>WDCN.0
(RWT)</td><td colspan="4">Reset Watchdog Timer. Setting this bit to 1 resets the watchdog timer count. If watchdog interrupt and/or reset modes are enabled, the software must set this bit to 1 before the watchdog timer elapses to prevent an interrupt or reset from occurring. This bit always returns 0 when read.</td></tr>
<tr><td>WDCN.1
(EWT)</td><td colspan="4">Enable Watchdog Timer Reset. If this bit is set to 1 when the watchdog timer elapses, the watchdog resets the processor 512 system clock cycles later unless action is taken to disable the reset event. Clearing this bit to 0 prevents a watchdog reset from occurring but does not stop the watchdog timer or prevent watchdog interrupts from occurring if EWDI = 1. If EWT = 0 and EWDI = 0, the watchdog timer is stopped. If the watchdog timer is stopped (EWT = 0 and EWDI = 0), setting the EWT bit resets the watchdog interval and reset counter, and enables the watchdog timer. This bit is cleared on power-on reset and is unaffected by other forms of reset.</td></tr>
<tr><td>WDCN.2
(WTRF)</td><td colspan="4">Watchdog Timer Reset Flag. This bit is set to 1 when the watchdog resets the processor. Software can check this bit following a reset to determine if the watchdog was the source of the reset. Setting this bit to 1 in software does not cause a watchdog reset. This bit is cleared by power-on reset only and is unaffected by other forms of reset. It should also be cleared by software following any reset so that the source of the next reset can be correctly determined by software. This bit is only set to 1 when a watchdog reset actually occurs, so if EWT is cleared to 0 when the watchdog timer elapses, this bit is not set.</td></tr>
<tr><td>WDCN.3
(WDIF)</td><td colspan="4">Watchdog Interrupt Flag. This bit is set to 1 when the watchdog timer interval has elapsed or can be set to 1 by user software. When WDIF = 1, an interrupt request occurs if the watchdog interrupt has been enabled (EWDI = 1) and not otherwise masked, or prevented by an interrupt already in service (i.e., IGE = 1, IMS = 1, and INS = 0 must be true for the interrupt to occur). This bit should be cleared by software before exiting the interrupt service routine to avoid repeated interrupts. Furthermore, if the watchdog reset has been enabled (EWT = 1), a reset is scheduled to occur 512 system clock cycles following setting of the WDIF bit.</td></tr>
<tr><td>WDCN.4
(WD0);
WDCN.5
(WD1)</td><td colspan="4">Watchdog Timer Mode Select Bit 0; Watchdog Timer Mode Select Bit 1. These bits determine the watchdog interval or the length of time between resetting of watchdog timer and the watchdog generated interrupt in terms of system clocks. Modifying the watchdog interval through the WD[1:0] bits automatically resets the watchdog timer unless the 512 system clock reset counter is already in progress, in which case, changing the WD[1:0] bits does not affect the watchdog timer or reset counter.
<table>
<tr><th>WD1</th><th>WD0</th><th>CLOCKS UNTIL INTERRUPT</th><th>CLOCKS UNTIL RESET</th></tr>
<tr><td>0</td><td>0</td><td>2^{12}</td><td>2^{12} + 512</td></tr>
<tr><td>0</td><td>1</td><td>2^{15}</td><td>2^{15} + 512</td></tr>
<tr><td>1</td><td>0</td><td>2^{18}</td><td>2^{18} + 512</td></tr>
<tr><td>1</td><td>1</td><td>2^{21}</td><td>2^{21} + 512</td></tr>
</table></td></tr>
<tr><td>WDCN.6
(EWDI)</td><td colspan="4">Watchdog Interrupt Enable. If this bit is set to 1, an interrupt request can be generated when the WDIF bit is set to 1 by any means. If this bit is cleared to 0, no interrupt occurs when WDIF is set to 1; however, it does not stop the watchdog timer or prevent watchdog resets from occurring if EWT = 1. If EWT = 0 and EWDI = 0, the watchdog timer is stopped. If the watchdog timer is stopped (EWT = 0 and EWDI = 0), setting the EWDI bit resets the watchdog interval and reset counter, and enables the watchdog timer. This bit is cleared to 0 by power-on reset and is unaffected by other forms of reset.</td></tr>
<tr><td>WDCN.7
(POR)</td><td colspan="4">Power-On-Reset Flag. This bit is set to 1 whenever a power-on/brownout reset occurs. It is unaffected by other forms of reset. This bit can be checked by software following a reset to determine if a power-on/brownout reset occurred. It should always be cleared by software following a reset to ensure that the sources of following resets can be determined correctly.</td></tr>
</table>

3.1.10 Accumulator n Register (A[n], 9h[nh])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
A[n].[15:0]	This register acts as the accumulator for all ALU arithmetic and logical operations when selected by the accumulator pointer (AP). It can also be used as a general-purpose working register.

3.1.11 Prefix Register (PFX[n], Bh[n])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

<table>
<tr><th>BIT</th><th colspan="3">FUNCTION</th></tr>
<tr><td rowspan="12">PFX[n].[15:0]</td><td colspan="3">The Prefix register provides a means of supplying an additional 8 bits of high-order data for use by the succeeding instruction as well as providing additional indexing capabilities. This register only holds any data written to it for one execution cycle, after which it reverts to 0000h. Although this is a 16-bit register, only the lower 8 bits are actually used for prefixing purposes by the next instruction. Writing to or reading from any index in the prefix module selects the same 16-bit register. However, when the PFX register is written, the index n used for the PFX[n] write also determines the high-order bits for the register source and destination specified in the following instruction.</td></tr>
<tr><td rowspan="2">WRITE TO</td><td colspan="2">SOURCE, DESTINATION INDEX SELECTION</td></tr>
<tr><td>SOURCE REGISTER RANGE</td><td>DESTINATION REGISTER RANGE</td></tr>
<tr><td>PFX[0]</td><td>0h to Fh</td><td>0h to 7h</td></tr>
<tr><td>PFX[1]</td><td>10h to 1Fh</td><td>0h to 7h</td></tr>
<tr><td>PFX[2]</td><td>0h to Fh</td><td>8h to Fh</td></tr>
<tr><td>PFX[3]</td><td>10h to 1Fh</td><td>8h to Fh</td></tr>
<tr><td>PFX[4]</td><td>0h to Fh</td><td>10h to 17h</td></tr>
<tr><td>PFX[5]</td><td>10h to 1Fh</td><td>10h to 17h</td></tr>
<tr><td>PFX[6]</td><td>0h to Fh</td><td>18h to 1Fh</td></tr>
<tr><td>PFX[7]</td><td>10h to 1Fh</td><td>18h to 1Fh</td></tr>
<tr><td colspan="3">The index selection reverts to 0 (default mode allowing selection of registers 0h to 7h for destinations) after one cycle in the same manner as the contents of the PFX register.</td></tr>
</table>

3.1.12 Instruction Pointer Register (IP, Ch[0h])

Initialization: This register is cleared to 8000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
IP.[15:0]	This register contains the address of the next instruction to be executed and is automatically incremented by 1 after each program fetch. Writing an address value to this register caused program flow to jump to that address. Reading from this register does not affect program flow.

3.1.13 Stack Pointer Register (SP, Dh[1h])

Initialization: This register is cleared to 000Fh on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
SP.[3:0]	These four bits indicate the current top of the hardware stack, from 0h to Fh. This pointer is incremented after a value is pushed on the stack and decremented before a value is popped from the stack.
SP.[15:4]	Reserved. All reads return 0.

3.1.14 Interrupt Vector Register (IV, Dh[2h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
IV.[15:0]	This register contains the address of the interrupt service routine. The interrupt handler generates a CALL to this address whenever an interrupt is acknowledged.

3.1.15 Loop Counter 0 Register (LC[0], Dh[6h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
LC[0].[15:0]	This register is used as the loop counter for the DJNZ LC[0], src operation. This operation decrements LC[0] by one and then jumps to the address specified in the instruction by src.

3.1.16 Loop Counter 1 Register (LC[1], Dh[7h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
LC[1].[15:0]	This register is used as the loop counter for the DJNZ LC[1], src operation. This operation decrements LC[1] by one and then jumps to the address specified in the instruction by src.

3.1.17 Frame Pointer Offset Register (OFFS, Eh[3h])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
OFFS.[7:0]	This 8-bit register provides the Frame Pointer (FP) offset from the base pointer (BP). The Frame Pointer is formed by unsigned addition of Frame Pointer Base register (BP) and Frame Pointer Offset register (OFFS). The contents of this register can be postincremented or postdecremented when using the Frame Pointer for read operations and can be preincremented or predecremented when using the Frame Pointer for write operations. A carry out or borrow resulting from an increment/decrement operation has no affect on the Frame Pointer Base register (BP).

3.1.18 Data Pointer Control Register (DPC, Eh[4h])

Initialization: This register is cleared to 001Ch on all forms of reset.

Access: Unrestricted direct read/write access.

<table>
<tr><th>BIT</th><th colspan="3">FUNCTION</th></tr>
<tr><td rowspan="7">DPC.[1:0]
(SDPS[1:0])</td><td colspan="3">Source Data Pointer Select Bits 1:0. These bits select one of the three data pointers as the active source pointer for the load operation. A new data pointer must be selected before being used to read data memory:</td></tr>
<tr><td>SDPS1</td><td>SDPS0</td><td>SOURCE POINTER SELECTION</td></tr>
<tr><td>0</td><td>0</td><td>DP[0]</td></tr>
<tr><td>0</td><td>1</td><td>DP[1]</td></tr>
<tr><td>1</td><td>0</td><td>FP (BP[OFFS])</td></tr>
<tr><td>1</td><td>1</td><td>Reserved (select FP if set)</td></tr>
<tr><td colspan="3">These bits default to 00b but do not activate DP[0] as an active source pointer until the SDPS bits are explicitly cleared to 00b or the DP[0] register is written by an instruction. Also, modifying the register contents of a data/frame pointer register (DP[0], DP[1], BP, or OFFS) changes the setting of the SDPS bits to reflect the active source pointer selection.</td></tr>
<tr><td>DPC.2 (WBS0)</td><td colspan="3">Word/Byte Select 0. This bit selects access mode for DP[0]. When WBS0 is set to logic 1, the DP[0] is operated in word mode for data memory access; when WBS0 is cleared to logic 0, DP[0] is operated in byte mode for data memory access.</td></tr>
<tr><td>DPC.3 (WBS1)</td><td colspan="3">Word/Byte Select 1. This bit selects access mode for DP[1]. When WBS1 is set to logic 1, the DP[1] is operated in word mode for data memory access; when WBS1 is cleared to logic 0, DP[1] is operated in byte mode for data memory access.</td></tr>
<tr><td>DPC.4 (WBS2)</td><td colspan="3">Word/Byte Select 2. This bit selects access mode for BP[OFFS]. When WBS2 is set to logic 1, the BP[OFFS] is operated in word mode for data memory access; when WBS2 is cleared to logic 0, BP[OFFS] is operated in byte mode for data memory access.</td></tr>
<tr><td>DPC.[15:5]</td><td colspan="3">Reserved. Read returns 0.</td></tr>
</table>

3.1.19 General Register (GR, Eh[5h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
GR.[15:0]	This register is intended primarily for supporting byte operations on 16-bit data. The 16-bit register is byte-readable, byte-writable through the corresponding GRL and GRH 8-bit registers and byte-swappable through the GRS 16-bit register.

3.1.20 General Register Low Byte (GRL, Eh[6h])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
GRL.[7:0]	This register reflects the low byte of the GR register and is intended primarily for supporting byte operations on 16-bit data. Any data written to the GRL register is also stored in the low byte of the GR register.

3.1.21 Frame Pointer Base Register (BP, Eh[7h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
BP.[15:0]	This register serves as the base pointer for the frame pointer (FP). The frame pointer is formed by unsigned addition of frame pointer base register (BP) and frame pointer offset register (OFFS). The content of this base pointer register is not affected by increment/decrement operations performed on the offset (OFFS) register.

3.1.22 General Register Byte-Swapped (GRS, Eh[8h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted read-only access.

BIT	FUNCTION
GRS.[15:0]	This register is intended primarily for supporting byte operations on 16-bit data. This 16-bit read-only register returns the byte-swapped value for the data contained in the GR register.

3.1.23 General Register High Byte (GRH, Eh[9h])

Initialization: This register is cleared to 00h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
GRH.[7:0]	This register reflects the high byte of the GR register and is intended primarily for supporting byte operations on 16-bit data. Any data written to the GRH register is also stored in the high byte of the GR register.

3.1.24 General Register Sign Extended Low Byte (GRXL, Eh[Ah])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read-only access.

BIT	FUNCTION
GRXL.[15:0]	This register provides the sign extended low byte of GR as a 16-bit source.

3.1.25 Frame Pointer Register (FP, Eh[Bh])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read-only access.

BIT	FUNCTION
FP.[15:0]	This register provides the current value of the frame pointer (BP[OFFS]).

3.1.26 Data Pointer 0 Register (DP[0], Fh[3h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
DP[0].[15:0]	This register is used as a pointer to access data memory. DP[0] can be automatically incremented or decremented following each read operation, or can be automatically incremented or decremented before each write operation.

3.1.27 Data Pointer 1 Register (DP[1], Fh[7h])

Initialization: This register is cleared to 0000h on all forms of reset.

Access: Unrestricted direct read/write access.

BIT	FUNCTION
DP[1].[15:0]	This register is used as a pointer to access data memory. DP[1] can be automatically incremented or decremented following each read operation, or can be automatically incremented or decremented before each write operation.

SECTION 4: PERIPHERAL REGISTER MODULES

The MAX31782 has six peripheral register modules, Modules 0 through 5. This section describes the MAX31782's peripheral registers. Table 4-1 shows the MAX31782 peripheral register map. Table 4-2 explains peripheral register bit functions. Detailed peripheral register bit descriptions and default values appear in the corresponding function block description section.

Table 4-1. Peripheral Register Map

INDEX	M0	M1	M2	M3	M4	M5
00h	PO2	I2CBUF_M	I2CBUF_S	PWMC0	PWMC2	MCNT
01h	PO1	I2CST_M	I2CST_S	PWMR0	PWMR2	MA
02h		I2CIE_M	I2CIE_S	PWMC1	PWMC3	MB
03h	MIIR0	PO6	MIIR2	PWMR1	PWMR3	MC2
04h		MIIR1		SMBUS		MC1
05h				TACHR0	TACHR2	MC0
06h	TB0C	EIF6	ADST			MC1R
07h	TB0R	EIE6	ADADDR	TACHR1	TACHR3	MC0R
08h	PI2	PI6	ADCN	PWMV0	PWMV2	PWMV4
09h	PI1	SVM	ADDATA	PWMCN0	PWMCN2	PWMCN4
0Ah				PWMV1	PWMV3	PWMC4
0Bh	TB0V			PWMCN1	PWMCN3	PWMR4
0Ch		I2CCN_M	I2CCN_S	TACHV0	TACHV2	TACHV4
0Dh	TB0CN	I2CCK_M	I2CCK_S	TACHCN0	TACHCN2	TACHCN4
0Eh		I2CTO_M	I2CTO_S	TACHV1	TACHV3	
0Fh		I2CSLA_M	I2CSLA_S	TACHCN1	TACHCN3	TACHR4
10h	PD2	EIES6		MIIR3	MIIR4	
11h	PD1					TACHR5
12h		PD6				TACHV5
13h						TACHCN5
14h						PWMC5
15h			I2C_SPB			PWMR5
16h		ETS	DEV_NUM			PWMV5
17h		ADCG1				PWMCN5
18h		ADCG5	ICDT0			MIIR5
19h		ADVOFF	ICDT1			
1Ah		TOEX	ICDC			
1Bh			ICDF			
1Ch			ICDB			
1Dh			ICDA			
1Eh			ICDD			
1Fh						

Table 4-2. Peripheral Register Bit Functions

MODULE 0																	
REGISTER	**INDEX**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PO2	00h									PO2[7:0]							
PO1	01h									—	—	PO1[5:0]					
MIIR0	03h	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TB0
TB0C	06h	TB0C[15:0]															
TB0R	07h	TB0R[15:0]															
PI2	08h									PI2[7:0]							
PI1	09h									—	—	PI1[5:0]					
TB0V	0Bh	TB0V[15:0]															
TB0CN	0Dh	C/$\overline{TB}$	—	—	TBCS	TBCR	TBPS[2:0]			TFB	EXFB	TBOE	DCEN	EXENB	TRB	ETB	CP/$\overline{RLB}$
PD2	10h									PD2[7:0]							
PD1	11h									—	—	PD1[5:0]					

MODULE 1																	
REGISTER	**INDEX**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
I2CBUF_M	00h	—	—	—	—	—	—	—	—	D[7:0]							
I2CST_M	01h	I2CBUS	I2CBUSY	—	—	I2CSPI	I2CSCL	I2CROI	I2CGCI	I2CNACKI	—	I2CAMI	I2CTOI	I2CSTRI	I2CRXI	I2CTXI	I2CSRI
I2CIE_M	02h	—	—	—	—	I2CSPIE	—	I2CROIE	I2CGCIE	I2CNACKIE	—	I2CAMIE	I2CTOIE	I2CSTRIE	I2CRXIE	I2CTXIE	I2CSRIE
PO6	03h									PO6_7	PO6_6	—	PO6_4	PO6_3	PO6_2	PO6_1	PO6_0
MIIR1	04h	—	—	—	—	—	—	I2CM_WU	I2CM	P6_7	P6_6	SVM	P6_4	P6_3	P6_2	P6_1	P6_0
EIF6	06h									IFP6_7	IFP6_6	—	IFP6_4	IFP6_3	IFP6_2	IFP6_1	IFP6_0
EIE6	07h									IEP6_7	IEP6_7	—	IEP6_4	IEP6_3	IEP6_2	IEP6_1	IEP6_0
PI6	08h									PI6_7	PI6_6	—	PI6_4	PI6_3	PI6_2	PI6_1	PI6_0
SVM	09h	—	—	—	—	SVMTH[3:0]				—	—	—	SVMSTOP	SVMI	SVMIE	SVMRDY	SVMEN
I2CCN_M	0Ch	—	—	—	—	—	—	I2CSTREN	I2CGCEN	I2CSTOP	I2CSTART	I2CACK	I2CSTRS	—	I2CMODE	I2CMST	I2CEN
I2CCK_M	0Dh	I2CCKH[7:0]								I2CCKL[7:0]							
I2CTO_M	0Eh									I2CTO[7:0]							
I2CSLA_M	0Fh	—	—	—	—	—	—	—	—	—	A[6:0]						
EIES6	10h									IESP6_7	IESP6_7	—	IESP6_4	IESP6_3	IESP6_2	IESP6_1	IESP6_0
PD6	12h									PD6[7:0]							
ETS	16h	—	—	—	—	—	—	—	—	EXTERNAL TEMP SLOPE [7:0]							
ADCG1	17h	—	ADC VOLTAGE SCALE TRIM FOR GAIN 1 [14:0]														
ADCG5	18h	—	ADC VOLTAGE SCALE TRIM FOR GAIN 5 [14:0]														
ADVOFF	19h	ADC VOLTAGE OFFSET [15:0]															
TOEX	1Ah	EXTERNAL TEMP OFFSET [15:0]															

Table 4-2. Peripheral Register Bit Functions (continued)

MODULE 2																	
REGISTER	**INDEX**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
I2CBUF_S	00h	—	—	—	—	—	—	—	—	D[7:0]							
I2CST_S	01h	I2CBUS	I2CBUSY	—	—	I2CSPI	I2CSCL	I2CROI	I2CGCI	I2CNACKI	—	I2CAMI	I2CTOI	I2CSTRI	I2CRXI	I2CTXI	I2CSRI
I2CIE_S	02h	—	—	—	—	I2CSPIE	—	I2CROIE	I2CGCIE	I2CNACKIE	—	I2CAMIE	I2CTOIE	I2CSTRIE	I2CRXIE	I2CTXIE	I2CSRIE
MIIR2	03h	—	—	—	—	—	—	—	—	—	—	—	—	—	I2CS_WU	I2CS	ADC
ADST	06h	—	—	—	—	ADDAT[3:0]				—	ADCONV	ADDAI	ADCFG	ADIDX[3:0]			
ADADDR	07h	—	—	—	—	ADBADD[3:0]				—	ADSTART[2:0]			—	ADEND[2:0]		
ADCN	08h	—	ADCCLK[2:0]			ADDAINV[1:0]		—	—	IREFEN	ADCONT	ADDAIE	—	ADACQ[3:0]			
ADDATA	09h	ADDATA[15:0], SEE *SECTION 6: Analog-to-Digital Converter (ADC)* FOR DETAILS															
I2CCN_S	0Ch	—	—	—	—	—	—	I2CSTREN	I2CGCEN	I2CSTOP	I2CSTART	I2CACK	I2CSTRS	—	I2CMODE	I2CMST	I2CEN
I2CCK_S	0Dh	I2CCKH[7:0]								I2CCKL[7:0]							
I2CTO_S	0Eh									I2CTO[7:0]							
I2CSLA_S	0Fh	—	—	—	—	—	—	—	—	—	A[6:0]						
I2C_SPB	15h	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	I2C_SPE
DEV_NUM	16h									DEVICE NUMBER[7:0]							
ICDT0	18h	ICDT0[15:0]															
ICDT1	19h	ICDT1[15:0]															
ICDC	1Ah									DME	—	REGE	—	CMD[3:0]			
ICDF	1Bh									—	—	—	—	PSS1	PSS0	JTAG_SPE	TXC
ICDB	1Ch									ICDB[7:0]							
ICDA	1Dh	ICDA[15:0]															
ICDD	1Eh	ICDD[15:0]															

MODULE 3																	
REGISTER	**INDEX**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PWMC0	00h	PWMC0[15:0]															
PWMR0	01h	PWMR0[15:0]															
PWMC1	02h	PWMC1[15:0]															
PWMR1	03h	PWMR1[15:0]															
SMBUS	04h	—	—	—	—	—	—	—	—	—	—	—	—	RESET_S	RESET_M	SMB_MOD_S	SMB_MOD_M
TACHR0	05h	TACHR0[15:0]															
TACHR1	07h	TACHR1[15:0]															
PWMV0	08h	PWMV0[15:0]															
PWMCN0	09h	—	—	—	PWMCS	PWMCR	PWMPS[2:0]			TFB	—	—	DCEN	—	PWMEN	ETB	—
PWMV1	0Ah	PWMV1[15:0]															
PWMCN1	0Bh	—	—	—	PWMCS	PWMCR	PWMPS[2:0]			TFB	—	—	DCEN	—	PWMEN	ETB	—
TACHV0	0Ch	TACHV0[15:0]															
TACHCN0	0Dh	—	TRPS[1:0]		—	—	TPS[2:0]			TF	TEXF	—	—	TEXEN	TACHE	TACHIE	—
TACHV1	0Eh	TACHV1[15:0]															
TACHCN1	0Fh	—	TRPS[1:0]		—	—	TPS[2:0]			TF	TEXF	—	—	TEXEN	TACHE	TACHIE	—
MIIR3	10h	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TACH1	TACH0

Table 4-2. Peripheral Register Bit Functions (continued)

MODULE 4																	
REGISTER	**INDEX**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PWMC2	00h	PWMC2[15:0]															
PWMR2	01h	PWMR2[15:0]															
PWMC3	02h	PWMC3[15:0]															
PWMR3	03h	PWMR3[15:0]															
TACHR2	05h	TACHR2[15:0]															
TACHR3	07h	TACHR3[15:0]															
PWMV2	08h	PWMV2[15:0]															
PWMCN2	09h	—	—	—	PWMCS	PWMCR	PWMPS[2:0]			TFB	—	—	DCEN	—	PWMEN	ETB	—
PWMV3	0Ah	PWMV3[15:0]															
PWMCN3	0Bh	—	—	—	PWMCS	PWMCR	PWMPS[2:0]			TFB	—	—	DCEN	—	PWMEN	ETB	—
TACHV2	0Ch	TACHV2[15:0]															
TACHCN2	0Dh	—	TRPS[1:0]		—	—	TPS[2:0]			TF	TEXF	—	—	TEXEN	TACHE	TACHIE	—
TACHV3	0Eh	TACHV3[15:0]															
TACHCN3	0Fh	—	TRPS[1:0]		—	—	TPS[2:0]			TF	TEXF	—	—	TEXEN	TACHE	TACHIE	—
MIIR4	10h	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TACH3	TACH2

MODULE 5																	
REGISTER	**INDEX**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
MCNT	00h									OF	MCW	CLD	SQU	OPCS	MSUB	MMAC	SUS
MA	01h	MA[15:0]															
MB	02h	MB[15:0]															
MC2	03h	MC2[15:0]															
MC1	04h	MC1[15:0]															
MC0	05h	MC0[15:0]															
MC1R	06h	MC1R[15:0]															
MC0R	07h	MC0R[15:0]															
PWMV4	08h	PWMV4[15:0]															
PWMCN4	09h	—	—	—	PWMCS	PWMCR	PWMPS[2:0]			TFB	—	—	DCEN	—	PWMEN	ETB	—
PWMC4	0Ah	PWMC4[15:0]															
PWMR4	0Bh	PWMR4[15:0]															
TACHV4	0Ch	TACHV4[15:0]															
TACHCN4	0Dh	—	TRPS[1:0]		—	—	TPS[2:0]			TF	TEXF	—	—	TEXEN	TACHE	TACHIE	—
TACHR4	0Fh	TACHR4[15:0]															
TACHR5	11h	TACHR5[15:0]															
TACHV5	12h	TACHV5[15:0]															
TACHCN5	13h	—	TRPS[1:0]		—	—	TPS[2:0]			TF	TEXF	—	—	TEXEN	TACHE	TACHIE	—
PWMC5	14h	PWMC5[15:0]															
PWMR5	15h	PWMR5[15:0]															
PWMV5	16h	PWMV5[15:0]															
PWMCN5	17h	—	—	—	PWMCS	PWMCR	PWMPS[2:0]			TFB	—	—	DCEN	—	PWMEN	ETB	—
MIIR5	18h	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TACH5	TACH4

SECTION 5: INTERRUPTS

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 5: INTERRUPTS

The MAX31782 provides a single, programmable interrupt vector (IV) that can be used to handle internal and external interrupts. Interrupts can be generated from system level sources (e.g., watchdog timer) or by sources associated with the peripheral modules. Only one interrupt can be handled at a time, and all interrupts naturally have the same priority. A programmable interrupt mask register (IMR) allows software-controlled prioritization and nesting of high-priority interrupts. Figure 5-1 shows a diagram of the interrupt hierarchy.

5.1 Servicing Interrupts

For the MAX31782 to service an interrupt, interrupts must be enabled globally, modularly, and locally. The interrupt global enable (IGE) bit located in the interrupt and control (IC) register acts as a global interrupt mask. This bit defaults to 0, and it must be set to 1 before any interrupt takes place.

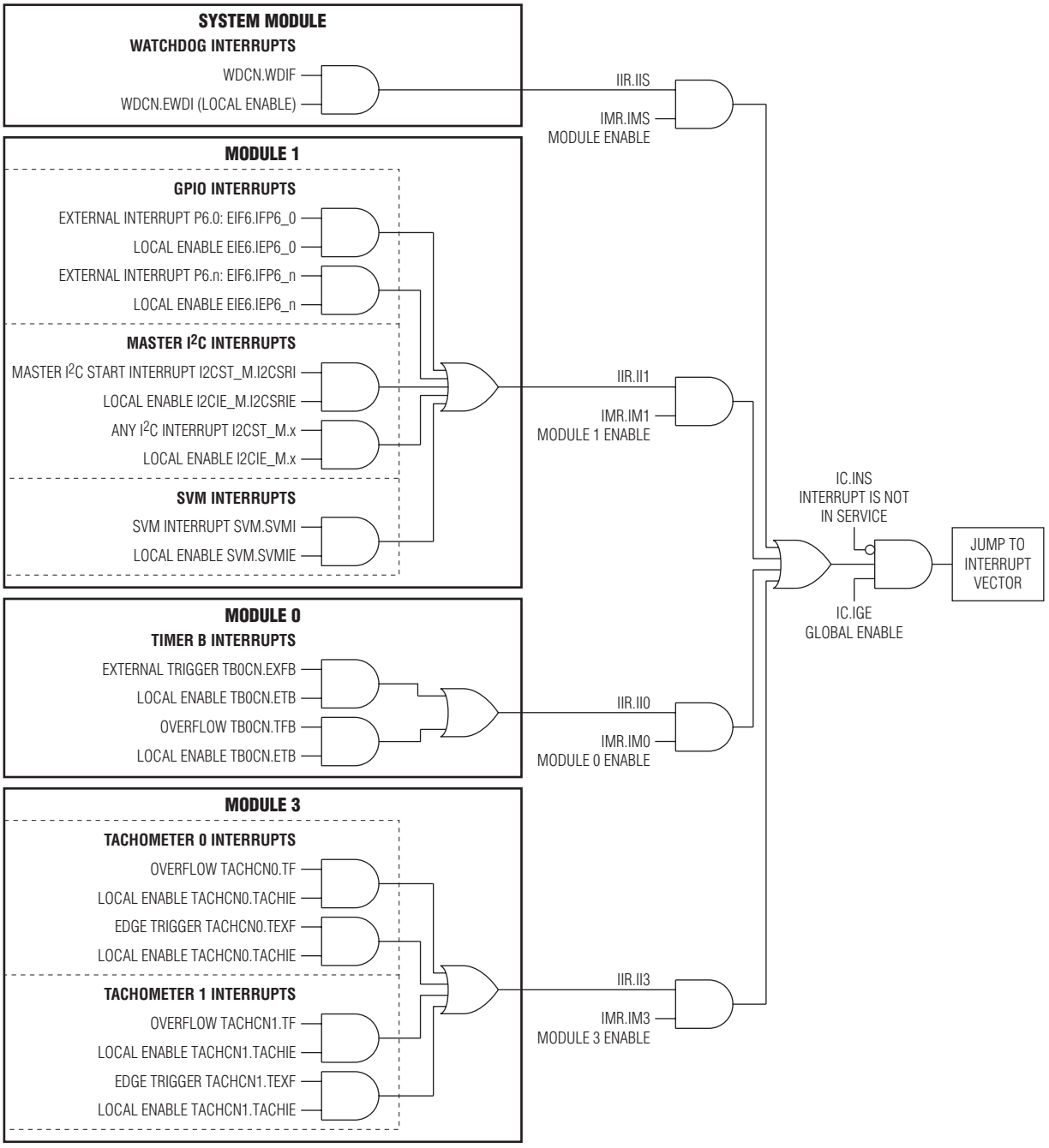
The local interrupt-enable bit for a particular source is in one of the peripheral registers associated with that peripheral module, or in a system register for any system interrupt source. Between the global and local enables are intermediate per-module and system interrupt mask bits. These mask bits reside in the interrupt mask system register (IMR). By implementing intermediate per-module masking capability in a single register, interrupt sources spanning multiple modules can be selectively enabled/disabled in a single instruction. This promotes a simple, fast, and user-definable interrupt prioritization scheme. The interrupt source-enable hierarchy is illustrated in Figure 5-1 as well as Table 5-1.

When an interrupt condition occurs, its individual flag is set, even if the interrupt source is disabled at the local, module, or global level. Interrupt flags must be cleared within the user interrupt routine to avoid repeated interrupts from the same source.

Since all interrupts vector to the address contained in the interrupt vector register (IV), the interrupt identification register (IIR) can be used by the interrupt service routine to determine the module source of an interrupt. The IIR register contains a bit flag for each peripheral module and one flag associated with all system interrupts; if the bit for a module is set, then an interrupt is pending that was initiated by that module.

The MAX31782 provides two ways to determine which block inside a module caused an interrupt to occur. Each module has a module interrupt identification register (MIIR) that indicates which of the module's interrupt sources has a pending interrupt. The peripheral register bits inside the module also provide a way to differentiate among interrupt sources. Section *5.2 Module Interrupt Identification Registers* has more detail on the MIIR registers.

The IV register provides the location of the interrupt service routine. It can be set to any location within program memory. The IV register defaults to 0000h on reset or power-up, so if it is not changed to a different address, the user program must determine whether a jump to 0000h came from a reset or interrupt source.



NOTE: ONLY A FEW OF THE MAX31782 MODULES AND INTERRUPT SOURCES ARE SHOWN IN THIS INTERRUPT HIERARCHY FIGURE. REFER TO THE CORRESPONDING SECTIONS OF THIS USER'S GUIDE FOR MORE DETAILED INFORMATION ABOUT ALL THE POSSIBLE INTERRUPTS.

Figure 5-1. Interrupt Hierarchy

Table 5-1. Interrupt Sources and Control Bits

INTERRUPT	INTERRUPT FLAG	LOCAL ENABLE BIT	MODULE INTERRUPT IDENTIFICATION BIT	INTERRUPT IDENTIFICATION BIT	MODULE ENABLE BIT
Timer B: External Trigger	TB0CN.EXFB	TB0CN.ETB	MIIR0.TB0	IIR.II0	IMR.IM0
Timer B: Overflow	TB0CN.TFB				
I²C Master START Interrupt	I2CST_M.I2CSRI	I2CIE_M.I2CSRIE	MIIR1.I2CM	IIR.II1	IMR.IM1
I²C Master Transmit Complete Interrupt	I2CST_M.I2CTXI	I2CIE_M.I2CTXIE			
I²C Master Receive Ready Interrupt	I2CST_M. I2CRXI	I2CIE_M.I2CRXIE			
I²C Master Clock Stretch Interrupt	I2CST_M.I2CSTRI	I2CIE_M.I2CSTRIE			
I²C Master Timeout Interrupt	I2CST_M.I2CTOI	I2CIE_M.I2CTOIE			
I²C Master NACK Interrupt	I2CST_M.I2CNACKI	I2CIE_M.I2CNACKIE			
I²C Master Receiver Overrun Interrupt	I2CST_M.I2CROI	I2CIE_M.I2CROIE			
I²C Master STOP Interrupt	I2CST_M.I2CSPI	I2CIE_M.I2CSPIE			
I²C Master Wake-Up Interrupt	I2CST_M.I2CSRI	I2CIE_M.I2CSRIE	MIIR1.I2CM_WU		
External Interrupt P6.0	EIF6.IFP6_0	EIE6.IEP6_0	MIIR1.P6_0		
External Interrupt P6.1	EIF6.IFP6_1	EIE6.IEP6_1	MIIR1.P6_1		
External Interrupt P6.2	EIF6.IFP6_2	EIE6.IEP6_2	MIIR1. P6_2		
External Interrupt P6.3	EIF6.IFP6_3	EIE6.IEP6_3	MIIR1. P6_3		
External Interrupt P6.4	EIF6.IFP6_4	EIE6.IEP6_4	MIIR1. P6_4		
External Interrupt P6.6	EIF6.IFP6_6	EIE6.IEP6_6	MIIR1. P6_6		
External Interrupt P6.7	EIF6.IFP6_7	EIE6.IEP6_7	MIIR1. P6_7		
Supply Voltage Monitor Interrupt	SVM.SVMI	SVM.SVMIE	MIIR1.SVM		
I²C Slave START Interrupt	I2CST_S.I2CSRI	I2CIE_S.I2CSRIE	MIIR2.I2CS	IIR.II2	IMR.IM2
I²C Slave Transmit Complete Interrupt	I2CST_S.I2CTXI	I2CIE_S.I2CTXIE			
I²C Slave Receive Ready Interrupt	I2CST_S. I2CRXI	I2CIE_S.I2CRXIE			
I²C Slave Clock Stretch Interrupt	I2CST_S.I2CSTRI	I2CIE_S.I2CSTRIE			
I²C Slave Timeout Interrupt	I2CST_S.I2CTOI	I2CIE_S.I2CTOIE			
I²C Slave Address Match Interrupt	I2CST_S.I2CAMI	I2CIE_S.I2CAMIE			
I²C Slave NACK Interrupt	I2CST_S.I2CNACKI	I2CIE_S.I2CNACKIE			
I²C Slave General Call Interrupt	I2ST_S.I2CGCI	I2CIE_S.I2CGCIE			
I²C Slave Receiver Overrun Interrupt	I2CST_S.I2CROI	I2CIE_S.I2CROIE			
I²C Slave STOP Interrupt	I2CST_S.I2CSPI	I2CIE_S.I2CSPIE			
I²C Slave Wake-Up Interrupt	I2CST_S.I2CSRI	I2CIE_S.I2CSRIE	MIIR2.I2CS_WU		
ADC Data Available Interrupt	ADST.ADDAI	ADCN.ADDAIE	MIIR2.ADC		
TACH.0 Overflow	TACHCN0.TF	TACHCN0.TACHIE	MIIR3.TACH0	IIR.II3	IMR.IM3
External TACH.0 Trigger	TACHCN0.TEXF				
TACH.1 Overflow	TACHCN1.TF	TACHCN1.TACHIE	MIIR3.TACH1		
External TACH.1 Trigger	TACHCN1.TEXF				
TACH.2 Overflow	TACHCN2.TF	TACHCN2.TACHIE	MIIR4.TACH2	IIR.II4	IMR.IM4
External TACH.2 Trigger	TACHCN2.TEXF				
TACH.3 Overflow	TACHCN3.TF	TACHCN3.TACHIE	MIIR4.TACH3		
External TACH.3 Trigger	TACHCN3.TEXF				
TACH.4 Overflow	TACHCN4.TF	TACHCN4.TACHIE	MIIR5.TACH4	IIR.II5	IMR.IM5
External TACH.4 Trigger	TACHCN4.TEXF				
TACH.5 Overflow	TACHCN5.TF	TACHCN5.TACHIE	MIIR5.TACH5		
External TACH.5 Trigger	TACHCN5.TEXF				
Watchdog Interrupt	WDCN.WDIF	WDCN.EWDI	N/A	IIR.IIS	IMR.IMS

5.2 Module Interrupt Identification Registers

The MIIR registers are implemented to indicate which particular function within a peripheral module has caused the interrupt. The MAX31782 has six peripheral modules, M0 to M5. An MIIR register is implemented in each peripheral module. The MIIR registers are 16-bit read-only registers and all of them default to 0000h on system reset.

Each defined bit in an MIIR register is the final interrupt from a specific function, i.e., the interrupt enable bit(s) ANDed with the interrupt flag(s). A function can have multiple flags, but they all are ANDed with corresponding enable bits and combined to create a single interrupt identification bit for that specific function. For example, the I2C master has several interrupt sources; however, they all are combined to form a single identification bit, MIIR1.I2CM. The individual register bit functions are defined as follows.

5.2.1 Peripheral Module 0 Interrupt Identification Register (MIIR0, M0[03h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TB0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

BIT	NAME	DESCRIPTION
15:1	—	Reserved. A read returns 0.
0	TB0	This bit is set when an interrupt is generated by the Timer/Counter B module.

5.2.2 Peripheral Module 1 Interrupt Identification Register (MIIR1, M1[04h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	I2CM_WU	I2CM	P6_7	P6_6	SVM	P6_4	P6_3	P6_2	P6_1	P6_0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

BIT	NAME	DESCRIPTION
15:10	—	Reserved. A read returns 0.
9	I2CM_WU	This bit is set when there is a wake-up interrupt from the I2C master block. For this to occur, the I2C master block must be operating as a slave. See *SECTION 8: I2C-Compatible Master Interface* for more details on this operation. The wake-up interrupt function is identical to the function described for the I2CS_WU bit in MIIR2.
8	I2CM	This bit is set when there is an interrupt from the I2C master block. The I2C interrupt is a combination of all interrupts defined in the I2CST_M register for the I2C master block. See *SECTION 8: I2C-Compatible Master Interface* for more details on the individual interrupts.
7	P6_7	This bit is set when there is an external GPIO Interrupt at P6.7 (slave I2C SDA).
6	P6_6	This bit is set when there is an external GPIO Interrupt at P6.6 (slave I2C SCL).
5	SVM	This bit is set when there is an interrupt from supply voltage monitor (SVM).
4	P6_4	This bit is set when there is an external interrupt at P6.4.
3	P6_3	This bit is set when there is an external interrupt at P6.3.
2	P6_2	This bit is set when there is an external interrupt at P6.2.
1	P6_1	This bit is set when there is an external interrupt at P6.1.
0	P6_0	This bit is set when there is an external interrupt at P6.0.

5.2.3 Peripheral Module 2 Interrupt Identification Register (MIIR2, M2[03h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	—	I2CS_WU	I2CS	ADC
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

BIT	NAME	DESCRIPTION
15:3	—	Reserved. A read returns 0.
2	I2CS_WU	This bit is set when there is a wake-up interrupt from the I²C slave block. A wake-up interrupt is defined as an I²C START signal only when the CPU is operating in stop mode. As the CPU clock is not running in stop mode, this is an asynchronous interrupt. This interrupt causes the MAX31782 to automatically transition from stop mode to CPU mode. The wake-up interrupt shares the same enable bits as the slave I²C START interrupt I2CSRI. Once set, this bit is cleared by clearing the I2CST_S.I2CSRI bit.
1	I2CS	This bit is set when there is an interrupt from the I²C slave block. The I²C interrupt is a combination of all interrupts defined in the I2CST_S register for the I²C slave block. See *SECTION 7: I²C-Compatible Slave Interface* for more details on the individual interrupts.
0	ADC	This bit is set when there is an interrupt from the ADC.

5.2.4 Peripheral Module 3 Interrupt Identification Register (MIIR3, M3[10h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TACH1	TACH0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

BIT	NAME	DESCRIPTION
15:2	—	Reserved. A read returns 0.
1	TACH1	This bit is set when there is an interrupt from tachometer 1 (TACH.1).
0	TACH0	This bit is set when there is an interrupt from tachometer 0 (TACH.0).

5.2.5 Peripheral Module 4 Interrupt Identification Register (MIIR4, M4[10h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TACH3	TACH2
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

BIT	NAME	DESCRIPTION
15:2	—	Reserved. A read returns 0.
1	TACH3	This bit is set when there is an interrupt from tachometer 3 (TACH.3).
0	TACH2	This bit is set when there is an interrupt from tachometer 2 (TACH.2).

5.2.6 Peripheral Module 5 Interrupt Identification Register (MIIR5, M5[18h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	—	—	TACH5	TACH4
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	R	r	r	r	r	r

BIT	NAME	DESCRIPTION
15:2	—	Reserved. A read returns 0.
1	TACH5	This bit is set when there is an interrupt from tachometer 5 (TACH.5).
0	TACH4	This bit is set when there is an interrupt from tachometer 4 (TACH.4).

5.3 Interrupt System Operation

The interrupt handler hardware responds to any interrupt event when it is enabled. An interrupt event occurs when an interrupt flag is set. All interrupt requests are sampled at the rising edge of the clock and can be serviced by the processor one clock cycle later, assuming the request does not hit the interrupt exception window. The one-cycle stall between detection and acknowledgement/servicing is due to the fact that the current instruction may also be accessing the stack. For this reason, the CPU must allow the current instruction to complete before pushing the stack and vectoring to IV. If an interrupt exception window is generated by the currently executing instruction, the following instruction must be executed, so the interrupt service routine is delayed an additional cycle.

Interrupt operation in the MAX31782 CPU is essentially a state machine generated long CALL instruction. When the interrupt handler services an interrupt, it temporarily takes control of the CPU to perform the following sequence of actions:

1) The next instruction fetch from program memory is cancelled.
2) The return address is pushed on to the stack.
3) The INS bit is set to 1 to prevent recursive interrupt calls.
4) The instruction pointer is set to the location of the interrupt service routine (contained in the IV register).
5) The CPU begins executing the interrupt service routine.

Once the interrupt service routine completes, it should use the RETI instruction to return to the main program. Execution of RETI involves the following sequence of actions:

1) The return address is popped off the stack.
2) The INS bit is cleared to 0 to re-enable interrupt handling.
3) The instruction pointer is set to the return address that was popped off the stack.
4) The CPU continues execution of the main program.

Pending interrupt requests do not interrupt an RETI instruction; a new interrupt is serviced after first being acknowledged in the execution cycle that follows the RETI instruction and then after the standard one stall cycle of interrupt latency. This means there are at least two cycles between back-to-back interrupts.

5.3.1 Synchronous vs. Asynchronous Interrupt Sources

Interrupt sources can be classified as either asynchronous or synchronous. All internal interrupts are synchronous interrupts. An internal interrupt is directly routed to the interrupt handler that can be recognized in one cycle. All external interrupts are asynchronous interrupts by nature. When the device is not in stop mode, asynchronous interrupt sources are passed through a 3-clock sampling/glitch filter circuit before being routed to the interrupt handler. The sampling/glitch filter circuit is running on the system clock. An interrupt request with a pulse width less than three system clock cycles is not recognized. Note that the granularity of interrupt source is at module level. Synchronous interrupts and sampled asynchronous interrupts assigned to the same module produce a single interrupt to the interrupt handler.

5.3.2 Interrupt Prioritization by Software

All interrupt sources of the MAX31782 naturally have the same priority. However, when CPU operation vectors to the programmed interrupt vector address, the order in which potential interrupt sources are interrogated is left entirely up to the user, as this often depends upon the system design and application requirements. The IMR system register provides the ability to knowingly block interrupts from modules considered to be of lesser priority and manually re-enable the interrupt servicing by the CPU (by setting INS = 0). Using this procedure, a given interrupt service routine can continue executing, only to be interrupted by higher priority interrupts. An example demonstrating this software prioritization is provided in the *19.8 Handling Interrupts* section.

5.3.3 Interrupt Exception Window

An interrupt exception window is a noninterruptable execution cycle. During this cycle, the interrupt handler does not respond to any interrupt requests. All interrupts that would normally be serviced during an interrupt exception window are delayed until the next execution cycle.

Interrupt exception windows are used when two or more instructions must be executed consecutively without any delays in between. Currently, there is a single condition in the MAX31782 that causes an interrupt exception window: activation of the PFX register.

When the PFX register is activated by writing a value to it, it retains that value only for the next clock cycle. For the prefix value to be used properly by the next instruction, the instruction that sets the prefix value and the instruction that uses it must always be executed back to back. Therefore, writing to the PFX register causes an interrupt exception window on the next cycle. If an interrupt occurs during an interrupt exception window, an additional latency of one cycle in the interrupt handling is caused since the interrupt is not serviced until the next cycle.

SECTION 6: ANALOG-TO-DIGITAL CONVERTER (ADC)

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 6: ANALOG-TO-DIGITAL CONVERTER (ADC)

The MAX31782 contains a 12-bit analog-to-digital converter (ADC) with a 7-input mux (Figure 6-1). The mux selects the ADC input from six external channels and one internal channel. The six external channels can operate in fully differential voltage mode or in single-ended voltage mode. In addition, any of the six external channels can be configured to measure the temperature of an external diode. The internal channel is used exclusively to measure the die temperature.

6.1 Detailed Description

6.1.1 Conversion Modes

The ADC in the MAX31782 can operate in three modes, which are selected using the EXTEMP and ADCH bits in the configuration register:

1) Voltage Conversion Mode
2) External Temperature Sensing Mode
3) Internal Temperature Sensing Mode

In voltage conversion mode (EXTEMP = 0) and ADCH ≠ 6 or 7, the ADC reference can be either internal (IREFEN = 1) or external (IREFEN = 0). If the internal reference is used, the ADC full scale can be set to either 1.225V (ADGAIN = 0) or 5.5V (ADGAIN = 1). When an external reference is desired, the reference supply needs to be connected to pin AD3N. See *6.1.6 Using an External Reference* for more information about using an external reference.

When external temperature sensing mode is selected, current is forced into an external diode that is connected between the user specified channel pins (set by ADCH[2:0]). The diode voltage is converted into a digital value that gives the temperature value. The ADC automatically uses the internal reference when performing a temperature conversion. Whenever ADCH[2:0] = 6 or 7, internal temperature sensing mode is enabled and EXTEMP has no effect.

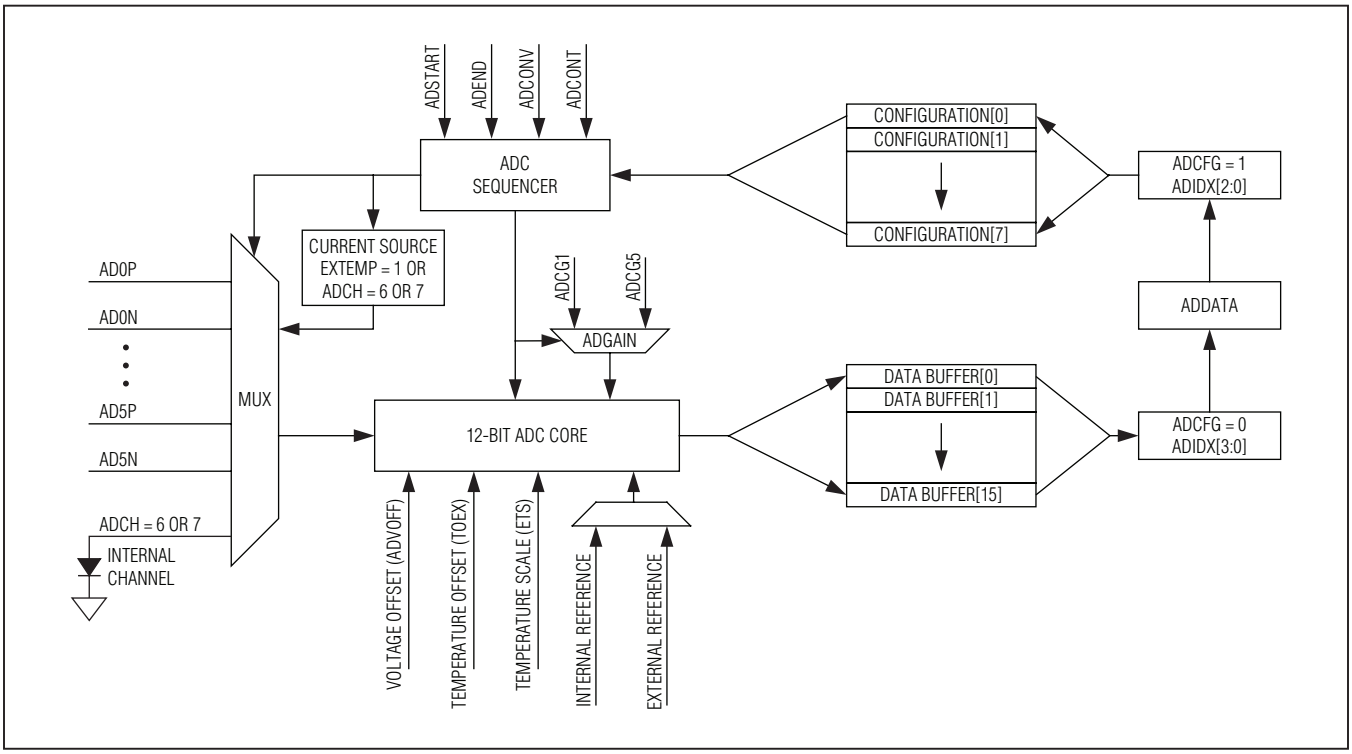


Figure 6-1. ADC Functional Block Diagram

6.1.2 Conversion Sequencing

The MAX31782 ADC performs a user-defined sequence of up to eight conversions. Each conversion in a sequence is set up using one of the eight ADC configuration registers. The configuration registers are accessed by writing to the ADDATA register when ADST.ADCFG = 1. The configuration register pointed to by ADDATA is selected using the ADIDX bits in the ADST register. The individual configuration registers allows each of the conversions in a sequence to select from the following options. For more information, see the *6.2.4 ADC Data and Configuration Register (ADDATA)* description.

- External temperature or voltage conversion
- Full-scale range
- Extended acquisition enable
- ADC conversion data alignment (left or right)
- Differential or single-ended conversion
- ADC channel selection

A sequence is set up in the ADADDR register by defining the starting conversion configuration address (ADSTART) and an ending conversion configuration address (ADEND). The configuration start address designates the configuration register to be used for the first conversion in a sequence. The configuration end address designates the configuration register used for the last conversion in a sequence. A single channel conversion can be viewed as a special case for sequence conversion, where the starting and ending configuration address is the same. The configuration registers can be viewed as a circular register array where ADSTART does not have to be less than ADEND. For example, if ADSTART = 1 and ADEND = 5, the sequence of conversions would be configurations 1, 2, 3, 4, 5. If ADSTART = 5 and ADEND = 1, the sequence of conversions would be configurations 5, 6, 7, 0, 1.

The ADC has two conversion sequence modes, single and continuous, which is set by the ADCONT bit. The start conversion bit (ADCONV) is used to start all conversions. In single sequence mode (ADCONT = 0), ADCONV remains set until the ADC has finished conversion on the last channel in the sequence. In continuous mode (ADCONT = 1), the ADCONV bit remains set until the continuous mode is stopped. Writing a 0 to the ADCONV bit stops the ADC operation at the completion of the current ADC conversion. Writing a 1 to the ADCONV bit when ADCONV bit is already set to 1 is ignored by the ADC controller.

6.1.3 ADC Conversion Time

The ADC clock is derived from the system clock with divide ratio defined by ADC clock divider bits (ADCN.ADCCLK[2:0]). Each sample takes 17 ADC clock cycles to complete. Three of the 17 ADC clock cycles are used for sample acquisition, and the remaining 14 clocks are used for data conversion. The ADC automatically reads each measurement twice and outputs the average of the two readings. This makes the resulting time for one complete conversion 34 ADC clock cycles.

Knowing this, it is possible to calculate the fastest ADC sample rate. The fastest ADC clock is:

ADC Clock = Sysclk/16 = 4MHz/16 = 250kHz

One conversion requires 34 ADC clocks:

Sample Rate = ADC Clock/34 Clocks = 250kHz/34 = 7.353ksps, or 136µs per sample

The ADC has an internal power management system that automatically shuts down the ADC when conversions are complete by clearing ADCONV to 0. After being shut down, the ADC begins conversions again when the ADCONV bit is set to 1 again. After ADCONV is set to 1, the ADC requires 20 ADCCLK cycles to set up and power-up prior to beginning the first conversion of the sequence.

In applications where extending the acquisition time is desired, the user can make use of the ADC acquisition extension bits (ADCN.ADACQ[3:0]). When the ADC acquisition extension is enabled (ADACQEN = 1), the sample is acquired over a prolonged period. The extended acquisition time is determined by ADACQ[3:0] and the ADC clock divider used. Table 6-1 shows the extended acquisition time in terms of ADC clocks at different ADACQ[3:0] and clock divider settings. The total acquisition time, ACQ, is the extended acquisition time (ADACQ, as listed in Table 6-1) plus three ADC clock cycles. Figure 6-2 shows the clocking required for one conversion.

Table 6-1. Extended Acquisition Time in Terms of ADC Clocks

ADACQ[3:0]	ADC CLOCK DIVIDER = 16	ADC CLOCK DIVIDER = 32	ADC CLOCK DIVIDER = 64	ADC CLOCK DIVIDER = 128
0	16	16	16	16
1	32	32	32	32
2	48	48	48	16
3	64	64	64	32
4	80	80	16	16
5	96	96	32	32
6	112	112	48	16
7	128	128	64	32
8	144	16	16	16
9	160	32	32	32
10	176	48	48	16
11	192	64	64	32
12	208	80	16	16
13	224	96	32	32
14	240	112	48	16
15	256	128	64	32

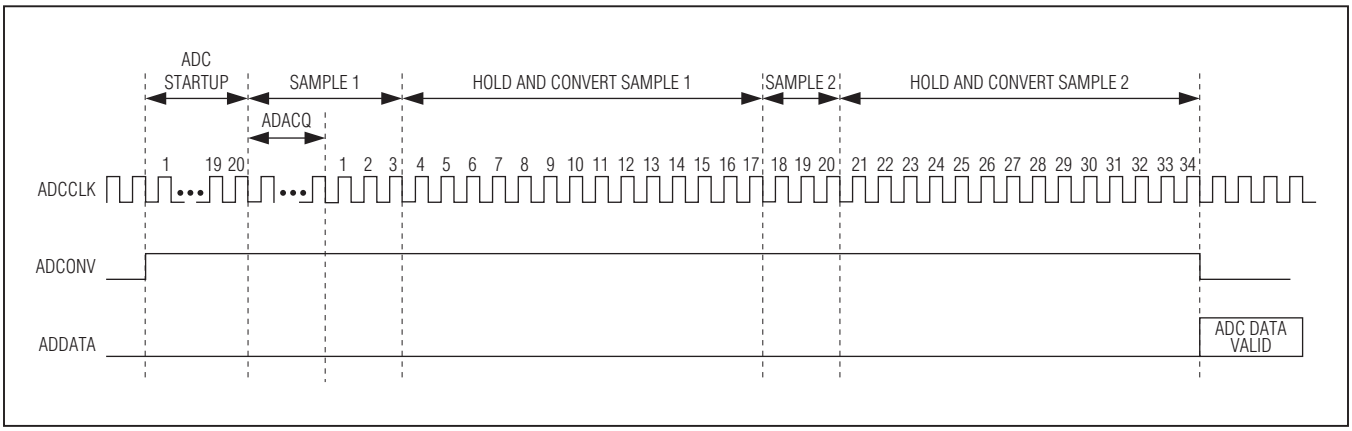


Figure 6-2. Extended Acquisition Time

The time required for the ADC to make a temperature measurement is greater than the time required for a voltage measurement. When a temperature conversion is performed, the ADC's internal current source forces current into the diode connected to the channel. As the current is being sourced, the ADC integrates the voltage across the diode. This is known as the integration time (t_{INT}). The integration time lasts approximately 2.91ms. This integration time is a constant and does not scale when the ADC clock speed is changed. When the integration time is complete, the ADC performs a voltage conversion of the integration. The voltage conversion is a normal voltage conversion and takes 34 ADC clock cycles. A temperature conversion requires that this integration and conversion process be performed twice. The extended acquisition time function does not apply when in temperature sensing mode. The time required for a complete temperature conversion can be calculated to be:

$$2 \times (t_{INT} + 34 \text{ ADC clocks})$$

6.1.4 ADC Data Reading

The ADC has a circular data buffer that holds the results from 16 conversions. This buffer is accessed by reading the ADDATA register when ADCFG is set to 0. The data buffer pointed to by ADST.ADIDX[3:0] is the buffer returned when ADDATA is read. ADIDX is automatically incremented following a read of ADDATA. This allows repeated reads of ADDATA to return the results from multiple conversions.

When ADCONV is set to 1, the conversion always starts writing to the buffer location indexed by the ADADDR. ADBADD[3:0] bits. As each result is written to the data buffer, the ADDAT[3:0] bits in ADST update to indicate which data buffer location was written to last. The ADC continues writing to the data buffer until the end of the buffer. Once the end of the data buffer is reached, the ADC index rolls over and writes data buffer 0.

When the ADC is operated in continuous sequence mode (ADCONT = 1), the data buffer is continuously written. For example, with a sequence of seven conversions with ADBADD[3:0] equal to 0, the first sequence writes to data buffer location 0 to 6, the second sequence writes to location 7 to 13 and the third sequence writes to 14, 15, 0 to 4. If the ADC is operating in single sequence mode, each time a new sequence is initiated by writing ADCCONV to 1, the ADC begins writing to the location specified by ADBADD[3:0].

6.1.5 ADC Interrupts

The MAX31782 provides an interrupt flag (ADST.ADDAI) that is set when conversions are complete. This flag generates an interrupt if enabled by setting the ADCN.ADDAIE interrupt enable bit. The condition that causes the ADDAI flag to be set can be selected using the ADCN.ADDAINV[1:0] bits.

Table 6-2. ADC Interrupt Intervals

ADDAINV[1:0]	SET ADDAI AFTER
00	Every ADC sample
01	End of every sequence(ADSTART to ADEND)
10	Every 12 ADC samples
11	Every 16 ADC samples

For a sequence that uses only one configuration register (ADSTART = ADEND), setting ADDAINV = 00 generates an interrupt with the same interval as ADDAINV = 01. In both cases, the ADDAI flag is set after every sample. The ADDAI flag can be cleared by software writing a 0, or it is automatically cleared when a new conversion sequence is started by setting ADCONV to a 1.

6.1.6 Using an External Reference

The ADC converter can use an external reference instead of the internal reference. When IREFEN = 0, the external reference option is enabled. The external reference needs to be applied to pin AD3N. When the external reference is used, voltage conversions can still be performed on the AD3P pin if they are done in single-ended mode (ADDIFF = 0). The voltage applied as an external reference must be between 1.1 V and 1.3 V.

The ADC converter automatically uses the gain setting in ADCG1 when an external reference is being used. Changing the ADCG setting has no effect on the conversion. The gain that is applied by ADCG1 probably needs to be adjusted to meet the needs of the application. See *6.2.8 ADC Voltage Scale Trim Registers (ADCG1 and ADCG5)* for more details on changing the gain.

6.1.7 Stop Mode Operation

The ADC converter supports stop mode operation. On entry into stop mode, the ADC is completely shut down to conserve power. On exiting stop mode, the ADC waits until ADCONV = 1 before starting up. When ADCONV is set to 1, the ADC waits 20 ADC clock cycles for setup and power-up before acquisition commences.

To prevent erroneous behavior, any ADC conversions in progress should be completed or aborted prior to entry into stop mode. If conversions are still ongoing on entry to stop mode, any in progress conversion are aborted and the ADCONV bit is reset to 0.

6.2 ADC Register Descriptions

The ADC is controlled by ADC SFR registers. Four of the registers, ADST, ADADDR, ADCN, and ADDATA, are used for setup, control, and reading from the ADC. There are five other registers, ETS, ADCG1, ADCG5, ADVOFF, and TOEX, which are used to adjust the gains and offsets applied to ADC results. To avoid undesired operations, the user should not write to bits labeled as reserved.

6.2.1 ADC Control Register (ADCN)

Register Address: M2[08h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	ADCCLK2	ADCCLK1	ADCCLK0	ADDAINV1	ADDAINV0	—	—	IREFEN	ADCONT	ADDAIE	—	ADACQ3	ADACQ2	ADACQ1	ADACQ0
Reset	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	rw*	rw*	rw*	rw*	rw*	r	r	rw*	rw*	rw*	r	rw*	rw*	rw*	rw*

**Unrestricted read, but can only be written to when ADCONV = 0.*

BIT	NAME	DESCRIPTION
15	—	Reserved. The user should not write to this bit.
14:12	ADCCLK[2:0]	ADC Clock Divider. These bits select the ADC conversion clock in relationship to the system clock.

ADCCLK[2:0]	READ AS	ADC CLOCK
000	011	Sysclk/16
001	011	Sysclk/16
010	011	Sysclk/16
011	011	Sysclk/16
100	100	Sysclk/16
101	101	Sysclk/32
110	110	Sysclk/64
111	111	Sysclk/128

BIT	NAME	DESCRIPTION
11:10	ADDAINV[1:0]	ADC Data Available Interrupt Interval. These bits select the condition for setting data available interrupt flag (ADDAI).

ADDAINV[1:0]	SET ADDAI AT
00	Every ADC sample
01	End of every sequence (ADSTART to ADEND)
10	Every 12 ADC samples
11	Every 16 ADC samples

BIT	NAME	DESCRIPTION
9:8	—	Reserved. The user should not write to these bits.
7	IREFEN	Internal Reference Enable. For voltage mode inputs, setting this bit to 1 enables the internal reference and clearing this bit to 0 chooses external reference sourced from pin AD3N. If the channel select bits equal 6 or 7 or if the external temperature mode is chosen, then the internal reference is chosen regardless of IREFEN setting. If an external reference is desired, see *6.1.6 Using an External Reference* for more information. When the internal reference is used, the FS can be set to factory programmed settings, 1.225V or 5.5V. The appropriate FS is chosen by the ADGAIN bit described in the configuration section.
6	ADCONT	ADC Continuous Sequence Mode. Setting this bit to 1 enables the continuous sequence mode. Clearing this bit to 0 disables the continuous sequence mode. In single sequence mode, the ADC conversion stops after the end of the sequence.
5	ADDAIE	ADC Data Available Interrupt Enable. Setting the ADDAIE bit to 1 enable an interrupt to be generated to the CPU when the ADDAI = 1. Clearing this bit to 0 disables an interrupt from generating when ADDAI = 1.
4	—	Reserved. The user should not write to this bit.
3:0	ADACQ[3:0]	ADC Acquisition Extension Bits [3:0]. These bits are used to extend sample acquisition time if the corresponding ADC acquisition extension is enabled (ADACQEN = 1). See *6.1.3 ADC Conversion Time* for details.

6.2.2 ADC Status Register (ADST)

Register Address: M2[06h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	ADDAT3	ADDAT2	ADDAT1	ADDAT0	—	ADCONV	ADDAI	ADCFG	ADIDX3	ADIDX2	ADIDX1	ADIDX0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw	rw	rw

BIT	NAME	DESCRIPTION
15:12	—	Reserved. The user should not write to these bits.
11:8	ADDAT[3:0]	ADC Data Available Address Bits [3:0]. These bits indicate the memory location last written to by the ADC. These bits are read-only.
7	—	Reserved. The user should not write to this bit.
6	ADCONV	ADC Start Conversion. Set this bit to 1 start the conversion process. This bit remains set until the conversion process is finished. In single sequence mode, this bit is cleared to 0 when the conversion sequence is finished. In continuous sequence mode, this bit remains set until the ADC conversion is stopped. To stop ADC conversion at any time, write 0 to this bit. The ADC stops acquiring data after the current conversion is finished, or, if the ADC is waiting during extended acquisition time, the ADC stops immediately. This bit is cleared to 0 on entry of stop mode.
5	ADDAI	ADC Data Available Interrupt Flag. This bit is set to 1 when the condition matching ADDAINV bits are met. The ADC memory location last written by the ADC is available at ADDAT. This flag causes an interrupt if the ADDAIE is enabled. This bit is cleared by software writing a 0 or when software changes ADCONV bit from 0 to 1.
4	ADCFG	ADC Conversion Configuration Register Select. This bit selects the target register pointed to by ADIDX. When ADCFG is set to 1, the ADIDX[2:0] configuration register is selected for read/write access. When ADCFG is cleared to 0, the ADIDX[3:0] data buffer location is selected for reading only.
3:0	ADIDX[3:0]	ADC Register Index Bits [3:0]. These bits together with ADCFG select the source/destination for ADDATA access. When ADCFG = 0, ADIDX[3:0] are used to address one of the 16 data buffers. When ADCFG = 1, only ADIDX[2:0] are used to address one of the eight configuration registers. This register value is auto-incremented on successive access (read/write) of ADDATA register.

6.2.3 ADC Address Register (ADADDR)

Register Address: M2[07h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	ADBADD3	ADBADD2	ADBADD1	ADBADD0	—	ADSTART2	ADSTART1	ADSTART0	—	ADEND2	ADEND1	ADEND0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	rw*	rw*	rw*	rw*	r	rw*	rw*	rw*	r	rw*	rw*	rw*

**Unrestricted read, but can only be written to when ADCONV = 0.*

BIT	NAME	DESCRIPTION
15:12	—	Reserved. The user should not write to these bits.
11:8	ADBADD[3:0]	ADC Data Buffer Address Bits [3:0]. These bits indicate the first ADC acquisition data memory location. These bits can be written to only when ADCONV = 0.
7	—	Reserved. The user should not write to this bit.
6:4	ADSTART[2:0]	ADC Conversion Configuration Start Address Bits [2:0]. These bits select the first conversion configuration register.
3	—	Reserved. The user should not write to this bit.
2:0	ADEND[2:0]	ADC Conversion Configuration Ending Address Bits [2:0]. These bits select the last conversion configuration register. This register is inclusive when defining the sequence.

6.2.4 ADC Data and Configuration Register (ADDATA)

Register Address: M2[09h]

The ADDATA register is used to set up the ADC sequence configurations and also to read the results of the ADC conversions. If the ADST.ADCFG bit is set to 1, writing to ADDATA writes to one of the configuration registers. If ADST.ADCFG is set to 0, reading from ADDATA reads one of the conversion results.

6.2.4.1 ADC Configuration Register (ADDATA when ADCFG = 1)

When ADCFG = 1, writing to the ADDATA register writes to one of the configuration registers. The configuration register written to is selected by the ADIDX[2:0] bits. The ADIDX[2:0] bits automatically increment after a write to ADDATA. This allows consecutive writes of ADDATA to set up consecutive configuration registers. The configuration registers are reset to 0 on all forms of reset and are not writable by the user.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	EXTEMP	ADGAIN	ADACQEN	ADALIGN	ADDIFF	ADCH2	ADCH1	ADCH0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	rw*	rw*	rw*	rw*	rw*	rw*	rw*	rw*

**When ADCFG = 1, unrestricted read, but can only be written to when ADCONV = 0.*

BIT	NAME	DESCRIPTION
15:8	—	Reserved. The user should not write to these bits.
7	EXTEMP	External Temperature Mode: Setting this bit to one chooses external temperature sensing operation. If this bit is set to zero, the ADC operates in normal voltage conversion mode for ADCH = 0–5. For ADCH = 6 or 7, the internal temperature is measured regardless of the setting of this bit. For external temperature measurement, the ADC does the following: ● A current source generated on the chip is directed to one of six positive ADC channel pins (AD0P–AD5P) based on channel select bits described in the configuration section. ● The current passing through the external diode is expected to return on the negative input pins (AD0N–AD5N) of the corresponding channel. ● The voltage across the positive and negative inputs of the channel is then scaled appropriately to produce a temperature sample with a slope of 2.4mV/°C. ● The internal reference is chosen during both temperature measurement modes, external and internal, regardless of IREFEN bit value. ● The slope of the temperature can additionally be controlled by up to +2% in increments of ~0.25°C to accommodate the variance in ideality factor of the diode being used. The slope comes preprogrammed with a 2N3904 used as a reference. The control register is at ETS (M1[16]) SFR. ● The slope of the internal temperature sensor is not user adjustable and set at the factory. ● The measured temperature is reported in the ADDATA register.
6	ADGAIN	ADC Reference Select. This bit selects the ADC scale factor.

IREFEN	ADGAIN	ADCSCALE
1	0	ADCG1
1	1	ADCG5
0	X	ADCG1

5	ADACQEN	ADC Acquisition Extension Enable. Setting this bit to 1 enables additional acquisition time to be inserted prior to this conversion. Clearing this bit to 0 disables the extended acquisition time.
4	ADALIGN	ADC Data Alignment Select. This bit selects the ADC data alignment mode. Setting this bit to 1 returns ADC data left-aligned in ADDATA [15:3] with ADDATA[2:0] zero padded. Clearing this bit to 0 returns ADC data in right-aligned format in ADDATA[12:0] with ADDATA[15:13] sign-extended by ADDATA[12].

<table>
<tr><td>3</td><td>ADDIFF</td><td colspan="3">ADC Differential Mode Select. In voltage mode, this bit selects the ADC conversion mode. When this bit is set to 1, the ADC conversion is in differential mode. When this bit is cleared to 0, the ADC conversion is performed in single-ended mode. During single-ended mode, the sample is measured between AD0P–AD5P and ground. If AD0P–AD5P transitions below 0, negative numbers are reported. No clamping of data is performed for negative inputs. The firmware needs to clamp the negative reading. In temperature mode, ADDIFF is a "don't care." The part automatically selects differential mode for temperature measurement.</td></tr>
<tr><td rowspan="9">2:0</td><td rowspan="9">ADCH[2:0]</td><td colspan="3">ADC Channel Select. These bits select the input channel source for the current ADC conversion.</td></tr>
<tr><td>ADCH[2:0]</td><td>ADDIFF = 0</td><td>ADDIFF = 1</td></tr>
<tr><td>000</td><td>AD0P</td><td>AD0P–AD0N</td></tr>
<tr><td>001</td><td>AD1P</td><td>AD1P–AD1N</td></tr>
<tr><td>010</td><td>AD2P</td><td>AD2P–AD2N</td></tr>
<tr><td>011</td><td>AD3P</td><td>AD3P–AD3N</td></tr>
<tr><td>100</td><td>AD4P</td><td>AD4P–AD4N</td></tr>
<tr><td>101</td><td>AD5P</td><td>AD5P–AD5N</td></tr>
<tr><td>11X</td><td colspan="2">Internal Temperature Mode</td></tr>
</table>

6.2.4.2 ADC Data Buffer (ADDATA when ADCFG = 0)

When ADCFG = 0, reading from the ADDATA register reads the ADC results stored in one of the 16 data buffers. The data buffer to read from is selected with the ADIDX[3:0] bits. Reading this register returns the 13-bit (12-bits plus a sign bit) ADC conversion data plus selected data buffer memory. The ADIDX[3:0] bits automatically increment after a read of ADDATA. This allows multiple reads of ADDATA to access consecutive data buffer locations without needing to change ADIDX. The data buffers are reset to 0 on all forms of reset and are not writable by the user.

The data that is read from the ADC buffer can be from either a temperature or voltage conversion. Also, the data can be right-aligned or left-aligned. Table 6-3 shows the returned bit weighting for each type of conversion.

Table 6-3. ADC Data Bit Weighting

Bit	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Temperature Right-Aligned	S	S	S	S	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}
Temperature Left-Aligned	S	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}	0	0	0
Voltage Right-Aligned	S	S	S	S	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
Voltage Left-Aligned	S	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	0	0	0

6.2.5 External Temperature Slope Control Register (ETS)

Register Address: M1[16h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	ETS7	ETS6	ETS5	ETS4	ETS3	ETS2	ETS1	ETS0
Reset	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0
Access	r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw	rw	rw	rw

The ETS register changes the slope of external temperature measurements to compensate for changes in diode ideality factor. The MAX31782 is factory calibrated to work with a diode-connected 2N3904 NPN transistor with an ideality factor of 1.004. Table 6-4 shows the possible settings for the ETS register and the corresponding ideality factor. Table 6-4 also shows the change in the reported temperature for each ETS register setting when the external diodes are at room temperature. The ETS register should only be set to the values shown in Table 6-4.

Table 6-4. ETS Register Settings

ETS	IDEALITY FACTOR	TEMPERATURE DELTA FROM 0x7C (°C)
0x00	1.0244	-7.37
0x04	1.0232	-7.01
0x0C	1.0220	-6.63
0x10	1.0208	-6.36
0x14	1.0196	-5.96
0x1C	1.0184	-5.62
0x20	1.0172	-5.43
0x24	1.0160	-5.08
0x2C	1.0148	-4.71
0x30	1.0136	-4.51
0x34	1.0124	-3.97
0x3C	1.0112	-3.7
0x40	1.0100	-3.65
0x44	1.0088	-3.23
0x4C	1.0076	-2.86
0x50	1.0064	-2.65
0x54	1.0052	-2.23
0x5C	1.0040	-1.96
0x60	1.0028	-1.75
0x64	1.0016	-1.3
0x6C	1.0004	-0.97
0x70	0.9992	-0.67
0x74	0.9980	-0.33
0x7C	0.9968	0

6.2.6 ADC External Temperature Offset Register (TOEX)

Register Address: M1[1Ah]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	S	S	S	S	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
Reset	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

s = special, initial value is dependent on trim settings

The TOEX register contains the temperature offset for the external temperature measurements. The default value of this register is -273 (Kelvin to Celsius) plus any offset that was calibrated out at the factory. This offset is applied to the raw data from the ADC prior to the value being stored into the data buffer. The final result stored in the data buffer is raw_adc + TOEX, where raw_adc is the converted temperature in Kelvin.

6.2.7 ADC Voltage Offset Register (ADVOFF)

Register Address: M1[19h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	S	S	S	S	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Reset	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

s = special, initial value is dependent on trim settings

The ADVOFF register contains the ADC voltage offset for the voltage mode. This is calibrated at the factory to cancel out any offset that can be present in the ADC. The user can add or subtract any offset that they desire by altering this register. This offset is applied to the raw data from the ADC prior to the value being stored into the data buffer. The value stored in the data buffer is raw_adc + ADVOFF, where raw_adc is the converted voltage without any offset compensation.

6.2.8 ADC Voltage Scale Trim Registers (ADCG1 and ADCG5)

ADCG1 Register Address: M1[17h]

ADCG5 Register Address: M1[18h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	Don't Care	ADCG14	ADCG13	ADCG12	ADCG11	ADCG10	ADCG9	ADCG8	ADCG7	ADCG6	ADCG5	ADCG4	ADCG3	ADCG2	ADCG1	ADCG0
Reset	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

s = special, initial value is dependent on trim settings

The ADCG1 and ADCG5 registers are used to adjust the ADC full scale by changing the gain applied to the ADC reference (internal or external). These registers are set at the factory to work with the internal reference. The internal reference voltage is set to 1.215V and cannot be changed by the user. When using the internal reference, ADCG1 and ADCG5 are factory calibrated to produce ADC full scale levels of 1.225V and 5.5V respectably.

The ADCG1 and ADCG5 registers are provided so the ADC full scale can be adjusted to meet the needs of the targeted application. Only bits ADCG[14:0] are used to adjust the full scale level. Some basic settings are the following:

- ADCG = 2000h: The full scale is 1x the reference level.
- ADCG = 1000h: The full scale is 2x the reference level.
- ADCG = 0800h: The full scale is 4x the reference level.

It is not recommended that a gain other than 1x, 2x, or 4x be used. This is because the weightings of the ADCG[10:0] bits are nonlinear. An application specific program needs to be developed that tests the ADC full scale for each possible code setting until the proper full scale is achieved.

6.3 ADC Code Examples

6.3.1 One Sequence of Four Temperature and Voltage Conversions

```
ADCN_bit.IREFEN = 1;      //enable the internal reference
ADCN_bit.ADCONT = 0;      //run a single conversion sequence

ADST_bit.ADCFG = 1;       //set ADDATA as ADCFG
ADST_bit.ADIDX = 0;       //ADIDX = 0, set to ADCFG[0]

ADDATA = 0x08;            //ADCFG[0]: Differential voltage CH0, 1.225V FS, Right Aligned
ADDATA = 0x41;            //ADCFG[1]: Single ended voltage CH1, 5.5V FS, Right Aligned
ADDATA = 0x85;            //ADCFG[2]: External temperature CH5, right aligned
ADDATA = 0x86;            //ADCFG[3]: Internal temperature, CH6, right aligned

ADADDR_bit.ADSTART = 0;   //start sequence with ADCFG[0]
ADADDR_bit.ADEND = 3;     //end sequence with ADCFG[3]

ADST_bit.ADCONV = 1;      //start the conversions
while(ADST_bit.ADCONV)    //wait for conversions to complete
   ;

ADST_bit.ADCFG = 0;       //set ADDATA to data buffer
ADST_bit.ADIDX = 0;       //set ADDATA to data buffer[0]

ch0_volt = ADDATA;        //read and store ch0 voltage to variable
ch1_volt = ADDATA;        //read and store ch1 voltage to variable
ch5_temp = ADDATA;        //read and store ch5 temperature to variable
int_temp = ADDATA;        //read and store internal temperature to variable
```

6.3.2 Continuous Conversion of 16 Samples

```
ADCN_bit.IREFEN = 1;      //enable the internal reference
ADCN_bit.ADDAINV = 3;     //set the interrupt flag after 16 conversions
ADCN_bit.ADCONT = 1;      //run continuous conversions

ADST_bit.ADCFG = 1;       //set ADDATA as ADCFG
ADST_bit.ADIDX = 0;       //ADIDX = 0, set to ADCFG[0]

ADDATA = 0x08;            //ADCFG[0]: Differential voltage, ch0, 1.225V FS, Right Aligned

ADADDR = 0x0000;          //ADSTART=0, ADEND=0, sequence is only ADCFG[0]

ADST_bit.ADDAI = 0;       //clear the interrupt flag
ADST_bit.ADCONV = 1;      //start the conversion

while(!ADST_bit.ADDAI)    //wait for 16 conversions to complete
   ;

ADST_bit.ADCONV = 0;      //stop the converter
ADST_bit.ADDAI = 0;       //clear the interrupt flag

ADST_bit.ADCFG = 0;       //set ADDATA to data buffer
ADST_bit.ADIDX = 0;       //set ADDATA to data buffer[0]

for(i=0; i<16; i++)
   ADC[i] = ADDATA;       //read all 16 conversions
```

SECTION 7: I²C-COMPATIBLE SLAVE INTERFACE

This section contains the following information:

LIST OF FIGURES

SECTION 7: I²C-COMPATIBLE SLAVE INTERFACE

The MAX31782 provides an I²C-compatible slave controller that allows the MAX31782 to communicate with a host device. This controller can also operate as an SMBus slave. Also designed into the I²C slave controller is the ability to bootload the MAX31782 with new user flash memory. The I²C slave interface can be set up to provide system interrupts after each I²C event. Figure 7-1 shows the basic operation flow of the I²C slave controller. The blocks in Figure 7-1 that are shaded are shown in more detail in Figure 7-2.

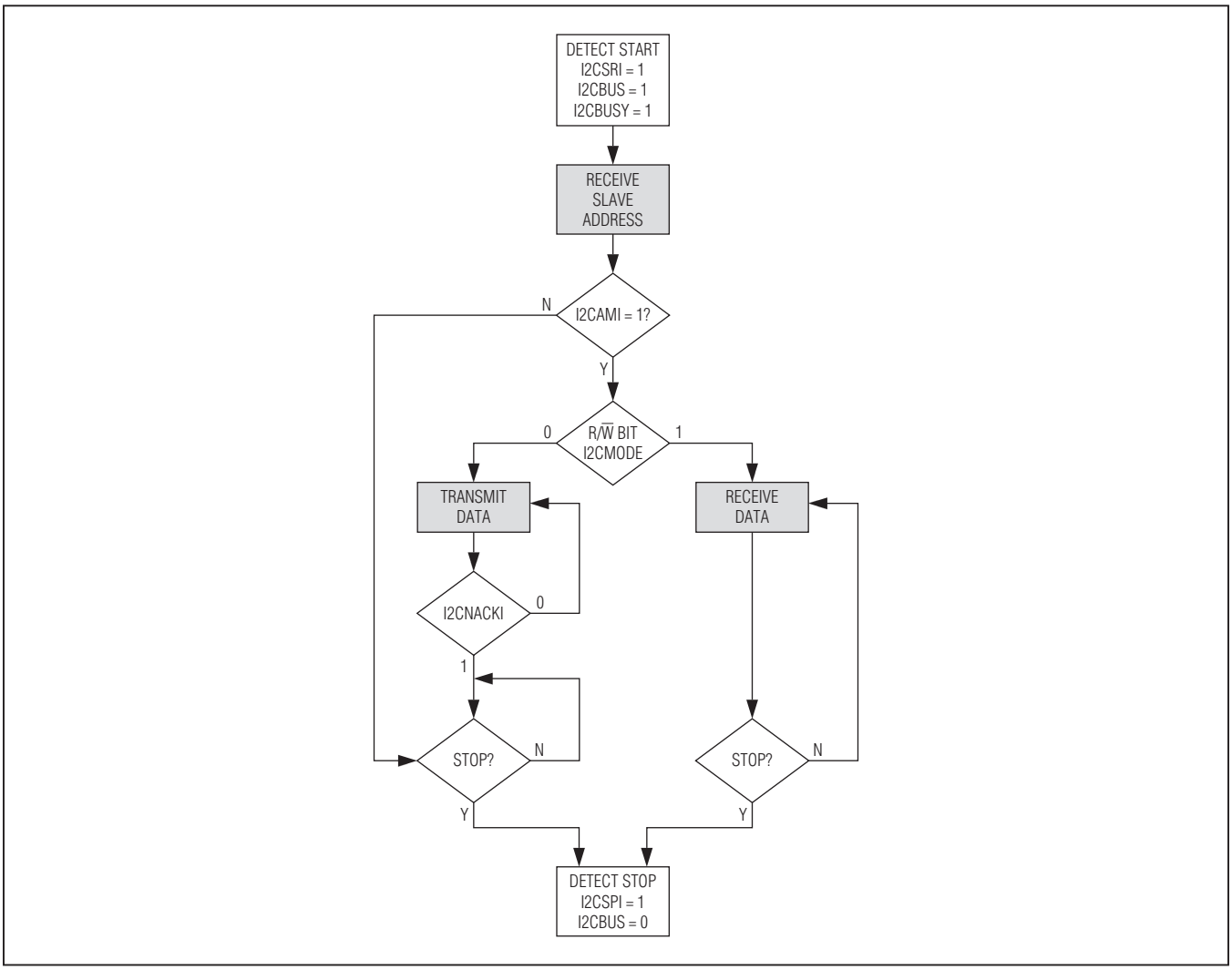


Figure 7-1. Slave I²C Flow

7.1 Detailed Description

7.1.1 Default Operation

The I²C slave controller is enabled (I2CCN_S.I2CEN=1) by default. As long as the I²C slave controller is enabled, the MAX31782 I²C bootloader can operate. This allows bootloading of blank devices without any setup of the I²C slave controller. Prior to the I²C slave controller being used for normal data communication, some software setup is required. This setup includes setting an I²C slave address and telling the slave controller which I²C events should generate interrupts.

7.1.2 Slave Address

Prior to communication, an I2C slave address may need to be selected. The I2C slave controller normally responds to two slave addresses. The I2C bootloader uses address 34h. This bootloader address cannot be changed and should not be used as the device slave address for normal communication. The second slave address is the address used for communication with the host. This slave address is set using the I2CSLA_S register. The address contained in the I2CSLA_S register is the address without the R/$\overline{W}$ bit. For example, the default I2C slave address is 36h, meaning the I2CSLA_S register contains 1Bh. If an address other than 36h is desired, the I2CSLA_S register can be programmed with this new address.

The I2C slave controller can also be programmed to respond to a third address, the general call address, which is 00h. This feature can be enabled by setting the I2CCN_S.I2CGCEN bit to 1.

7.1.3 I2C START Detection

The I2C slave controller always monitors the I2C bus for an I2C START, which is a high-to-low transition on SDA while SCL is held high. If an I2C START (or restart) condition is detected, the I2C slave sets the I2CSRI bit in the I2CST_S register, which can cause an interrupt if enabled. The detection of a START brings the I2C controller out of its idle state. Following a START, the I2C controller begins to monitor data on the I2C bus and the I2CBUSY bit is set to 1. The I2CBUS bit is also set to 1 indicate that the I2C bus is currently busy.

7.1.4 I2C STOP Detection

The I2C slave controller also always montors the I2C bus for an I2C STOP, which is a low-to-high transition on SDA while SCL is held high. If an I2C STOP condition is detected, the I2C slave controller sets the I2CSPI bit in the I2CST_S register, which can cause an interrupt if enabled. The I2CBUS bit is cleared to 0 following a STOP to indicate that the I2C bus is no longer busy.

7.1.5 Slave Address Matching

Following an I2C START or restart, the I2C slave controller knows that the next byte of data transmit by the host should be the slave address. The I2C slave automatically monitors for the slave address without any software interaction required. The I2C slave controller compares the first 7 bits received to the slave address programmed into I2CSLA_S.

After receiving the first 8 bits of data following a START, the I2C controller compares the first 7 bits to the value programmed into the I2CSLA_S register. If the received slave address matches I2CSLA_S, the I2C slave controller does the following steps, as illustrated in Figure 7-2.

- Transmits an ACK or NACK on the 9th clock based upon the setting of the I2CCN_S.I2CACK bit.
- Sets the I2CCN_S.I2CMODE bit with the value of the received R/$\overline{W}$ bit. This bit can be used by software to determine if the I2C slave controller would be asked to receive or transmit data.
- Sets the I2CST_S.I2CAMI bit to indicate that a slave address match was made. The setting of this bit can generate an interrupt if enabled.
- Clears the I2CBUSY flag.

Upon completion of the slave data byte (7 bits of slave address + R/$\overline{W}$ bit + ACK/NACK), the I2C slave controller enters one of three states:

- Data Transmit: The slave address matched and the R/$\overline{W}$ bit was a 1. The host is now expecting to clock data from the MAX31782. The MAX31782 retains control of the SDA line so data can be transmit to the host.
- Data Receive: The slave address matched and the R/$\overline{W}$ bit was a 0. The host wants to write data to the MAX31782. After the ACK/NACK bit, the MAX31782 releases SDA and prepares to receive a byte of data.
- Wait for START/STOP: The received slave address did not match I2CSLA_S. The controller enters idle state and waits for the next START condition or STOP condition.

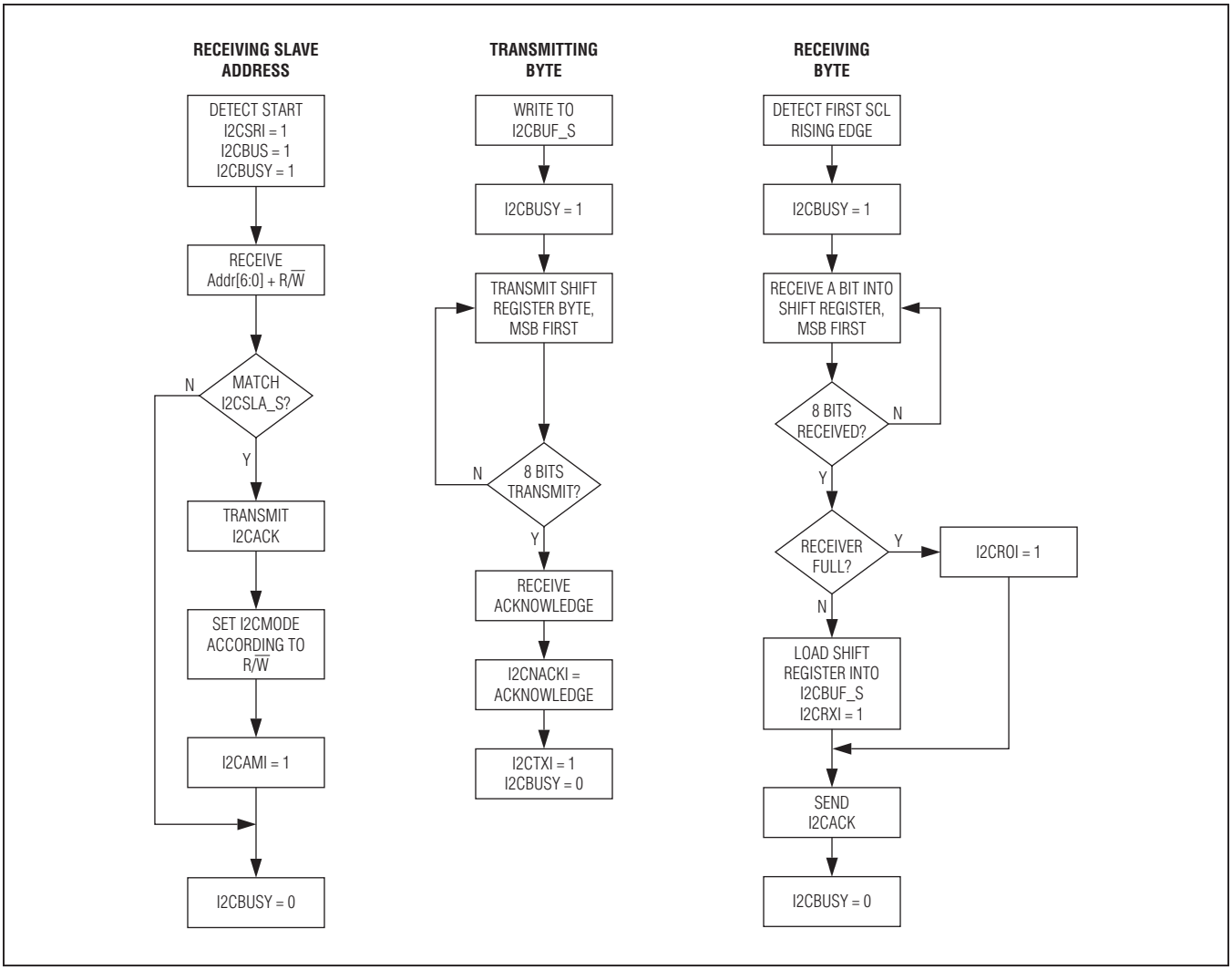


Figure 7-2. Slave I²C Data Flow

7.1.6 Transmitting Data

The MAX31782 I²C slave controller enters into data transmission mode after receiving a matching slave address with the R/$\overline{W}$ bit set to a 1. The steps of data transmission are shown in Figure 7-2. Data transmission is started by software loading a byte of data into the I2CBUF_S register. Loading I2CBUF_S causes the I2CBUSY bit in I2CST_S to be set. Once set, a write to I2CBUF_S is ignored. The first bit of data (most significant bit) is shifted to SDA when SCL is low. Each of the next 7 bits is then shifted following high-to-low transitions of SCL.

Following the 8th data (least significant bit) being shifted to SDA, the SDA line is released by the MAX31782 slave controller. This allows the host to signal an ACK or NACK during the 9th clock cycle. The MAX31782 I²C slave controller samples the acknowledge bit following the rising 9th SCL rising edge. After the acknowledge bit is sampled, the MAX31782 I²C slave controller performs the following tasks:

- Sets the I2CST_S.I2CTXI flag to indicate that the I²C slave controller transmit a complete byte. This can generate an interrupt if enabled.
- Sets or clears the I2CST_S.I2CNACKI flag to reflect the received acknowledge bit. The setting of I2CNACKI can generate and interrupt if enabled.

- Clears the I2CST_S.I2CBUSY flag to indicate that the I²C slave controller is not actively participating in the transfer of data.

The detection of an ACK by the MAX31782 I²C slave controller indicates that the host wants to receive another byte of data. The I²C slave controller maintains control of SDA following the ACK. The next byte to transmit needs to be loaded into I2CBUF_S prior to the host starting to clock this next byte. However, data cannot be loaded into I2CBUF_S prior to I2CBUSY being cleared, which indicates that all the bits in I2CBUF_S have been shifted onto SDA.

The detection of a NACK indicates that the host does not want to receive any additional data. The MAX31782 I²C slave controller releases control of SDA following the reception of the NACK bit. After the NACK, the slave controller enters idle state and monitors the I²C bus for a START or STOP condition.

7.1.7 Receiving Data

The MAX31782 I²C slave controller enters data reception mode after receiving a matching slave address with the R/$\overline{W}$ bit set to a 0. The steps of data reception are shown in Figure 7-2. The reception process begins when the I²C slave controller detects the first rising edge of SCL. This first rising edge sets I2CBUSY and also clock the first bit (MSB) of data from SDA into the data shift register.

When receiving data, the MAX31782 I²C slave controller uses a double buffer consisting of the I2CBUF_S register and the shift register. This allows the I²C module to continue receiving data while the previous data byte is being processed. After a complete byte (8 bits) of data are received, the I²C slave controller attempts to copy the received data from the shift register to I2CBUF_S. There are two possible results from the I²C slave controllers attempt to copy the shift register to I2CBUF_S.

1) If I2CBUF_S is empty, the I²C slave controller copies the data from the shift register into I2CBUF_S. The I2CRXI flag is set to indicate a received byte is ready to be read. The setting of I2CRXI can generate an interrupt if enabled.
2) If I2CBUF_S is full, the data in the shift register cannot be copied into I2CBUF_S. This causes a receive overrun condition. The receive overrun flag, I2CROI, is set, which can generate an interrupt if enabled. I2CBUF_S is full if it was not read by software following the reception of a previous byte.

After receiving a byte of data and the I2CRXI flag being set, it is up to software to read I2CBUF_S prior to a second byte being received. Reading the I2CBUF_S register returns the received data and also clears I2CBUF_S. As long as the previous byte of data is read from I2CBUF_S before the next byte has completed, receive overrun does not occur.

When in receive overrun and the I2CROI bit is set, any new incoming data is not shifted into the I2C slave controller. The controller responds to any bytes received with a NACK regardless of the setting of the I2CACK bit. The receive overrun condition and the I2CROI flag can only be cleared by software reading the first byte received from I2CBUF_S. When the receive overrun condition is cleared, the I²C slave controller copies the second byte that was received into I2CBUF, and again set I2CRXI to indicate a byte of data was received. The I²C slave controller resumes its normal operation in the next SCL clock cycle after I2CROI is cleared. To avoid losing any data, I2CROI must to be cleared prior to the first SCL clock rising edge of the next byte.

After the 9th bit of any byte has been received, the I2CBUSY bit is cleared to indicate that the controller is no longer participating in a data transaction. The value in I2CACK is transmitted to the host on the 9th SCL clock cycle, assuming the I²C slave controller is not operating in receive overrun.

7.1.8 Clock Stretching

If a slave device cannot receive or transmit another complete byte of data, it can hold SCL low, forcing the master to wait. Data transfer continues when the slave is ready for another byte of data and releases SCL.

The I2C slave controller can hold SCL low at the completion of each byte being transferred. If the I2C clock stretch enable bit (I2CSTREN) is set to a 1, the I2C controller holds SCL low after the clock pulse defined by the I2C clock stretch select bit (I2CSTRS). If I2CSTRS = 0, the I2C controller holds SCL low after the falling edge of the 9th clock pulse. Otherwise, if I2CSTRS = 1, the I2C controller holds SCL low after the falling edge of the 8th clock pulse. When the I2C controller is holding SCL low, the I2C clock stretch interrupt bit (I2CSTRI) is set. The I2C slave controller holds SCL low until I2CSTRI is cleared to 0 by software. Figure 7-3 shows the I2C slave controller clock stretching after receiving the 9th clock of a byte.

Normally when the I2C slave controller is receiving data, the value of I2CACK is output after the 8th clock falling edge. However, if clock stretching is enabled after the 8th clock, the I2C slave controller continually outputs the I2CACK bit until clock stretching is released by software. This allows software time to inspect data that was received before responding with an appropriate acknowledge bit.

Most applications that use the MAX31782's I2C slave controller need to use clock stretching. Generally the application is set to only respond to interrupts from the I2C slave controller, therefore it does not have to continuously poll the slave I2C controller. After each byte transfer is complete, the I2C slave controller needs to either read the received byte from I2CBUF_S or write the next byte to transmit to I2CBUF_S. Without using clock stretching, the host can begin clocking the next byte before the I2C slave controller is prepared. A few conditions that can require clock stretching to be enabled are listed below.

- When a slave address match is made and the R/$\overline{W}$ bit is set, the I2C slave controller is expected to transmit a byte of data to the host. This byte of data needs to be written to I2CBUF_S after the 8th clock of the slave address (when I2CBUSY is cleared) and prior to the first clock of the data byte. If clock stretching is not used, software may not be able to write the correct data into I2CBUF_S prior to the first clock of the data byte.
- Following the transmission of one byte of data to the host, another byte may be requested by the host sending an ACK bit. The I2C slave controller has between the 9th clock of the first data byte (when I2CBUSY is cleared) and the first clock of the second byte to write to I2CBUF_S. If clock stretching is not used, software may not be able to write the next byte to I2CBUF_S prior to the first clock of the second byte.
- After a byte is received by the I2C slave controller it may be necessary to stretch the clock. This allows software time to read the byte from I2CBUF_S and do any data processing. Without using clock stretching, there is a chance that a second byte could be sent prior to the software reading the first byte, creating a receive overrun condition. Any additional data that is sent after this time is lost.

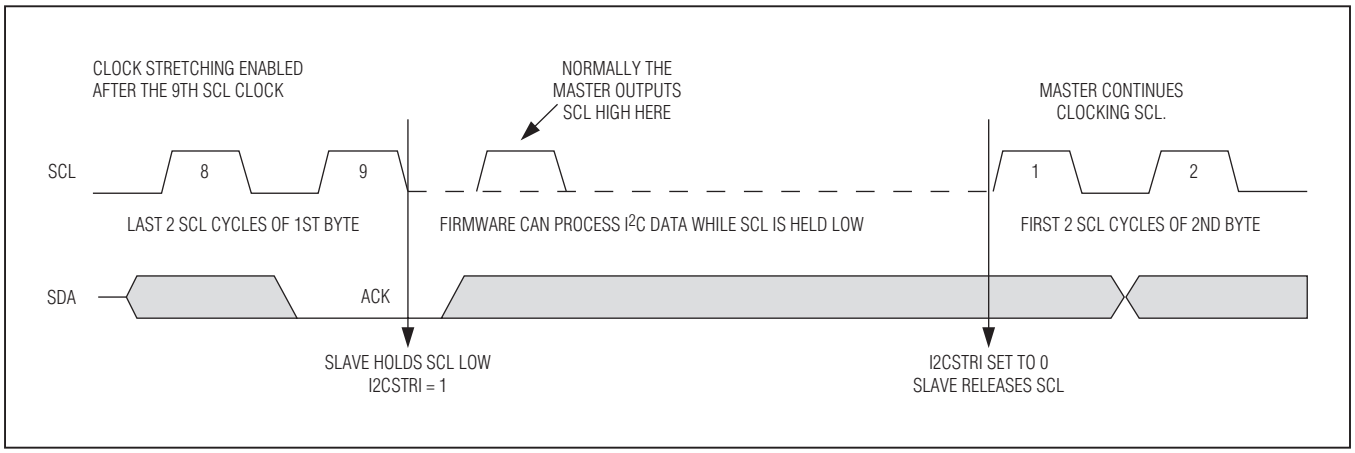


Figure 7-3. Slave I2C Clock Stretching

7.1.9 SMBus Timeout

The I2C slave controller can also be used for SMBus or PMBus™ communication. To maintain SMBus compatibility, a 30ms timer is implemented by the I2C slave controller. The purpose of this timer is to issue a timeout interrupt when SCL is low for greater than 30ms. The timer only starts when **none** of the following conditions are true:

1) The I2C slave controller is in the idle state and there is no communications on the I2C bus. The timer should not generate interrupts if the I2C slave controller is in the idle state regardless of how long SCL is low.
2) The SMBus mode bit is not set. This ensures the SMBus timeout functionality does not interfere with normal I2C functionality.
3) SCL is high. The timer is inactive whenever SCL is high. The timer resets when it is inactive.
4) The I2C slave controller is disabled or used as a master I2C controller. The timer is not needed in this case.

The following description explains when the SMBus timer starts, assuming that all other START conditions are met. When the MAX31782's I2C slave controller is idle and it receives a START, it exits the idle state and the timer becomes active (starts counting) any time SCL goes low. If following the START the master addresses a different slave on the bus, the I2C slave controller returns to the idle state and the timer is reset and becomes inactive. In short, as soon as SCL goes low following a START, the SMBus timer becomes active until the I2C slave controller re-enters idle state.

When a timeout occurs, the timeout bit (I2CTOI) is set, which can generate an interrupt if enabled. If a timeout occurs, it may be necessary to reset the I2C slave controller. See *7.1.10 Resetting the I2C Slave Controller* for more details.

SMBus mode selection is controlled by the SMBUS register. When the slave SMBus mode operation bit (SMB_MOD_S) is set to 1, the SMBus timeout functionality is enabled.

7.1.10 Resetting the I2C Slave Controller

The I2C slave controller can be reset by setting the RESET_S bit in the SMBUS register. After a delay of at least one system clock, this bit needs to be cleared to 0 by software and the reset is complete. A reset forces the I2C slave controller to release both SDA and SCL if they are being held low by the I2C slave controller. The reset also turns off the I2C slave controller (I2CEN = 0), resets all the I2C registers, and resets the internal state machine of the I2C slave controller. Following a reset, the I2C slave controller must be reinitialized, including enabled (I2CEN = 1) before it can be used again.

7.1.11 Operation as a Master

The MAX31782 contains two I2C interfaces, the slave (SDA and SCL) and master (MSDA and MSCL). These are two totally separate blocks within the MAX31782. However, both of the blocks are identical. Because of this, it is possible to operate the slave as a master and also operate the master as a slave.

To operate the slave (SDA and SCL) as a master I2C interface, the I2CMST bit in I2CCN_S needs to be set to a 1. When the slave is operating as a master, it uses the same registers (I2CCN_S, I2CST_S, etc) that it uses for slave operation. However, the bits in these registers have different functionality, as described in *SECTION 8: I2C-Compatible Master Interface*. The SMBUS.RESET_S bit can still be used to reset this interface (SDA and SCL) when operating as a master. The SMBUS.SMB_MOD_S bit has no effect when the interface is operating in master mode. See *SECTION 8: I2C-Compatible Master Interface* for details on initializing and using a master I2C interface.

Note: When the I2C slave interface is changed to operate in master mode, the I2C bootloader is not available.

7.1.12 GPIO

When the I2C slave controller is disabled (I2CCN_S.I2CEN = 0), the SDA and SCL pins can be used as GPIO pins. The SDA pin is mapped to GPIO port P6.7 and SCL is mapped to GPIO port P6.6. When used as GPIO outputs, the SDA and SCL pins can only be open-drain outputs. See *SECTION 11: General-Purpose Input/Output (GPIO) Pins* for more information on using SDA and SCL as GPIO pins.

Note: When the I2C slave interface is disabled, the I2C bootloader is not available.

PMBus is a trademark of SMIF, Inc.

7.2 I²C Slave Controller Register Descriptions

The following registers are used to control the I²C slave interface, which uses the SDA and SCL pins. These registers control the I²C slave interface if it is operating as either a slave or master. The bit descriptions detail how to use these registers when operating in slave mode. When operating in master mode, some of the bits and registers have different functionality. See *SECTION 8: I²C-Compatible Master Interface* section for more information on how to control the I²C slave interface when it is operating as a master.

7.2.1 I²C Slave Control Register (I2CCN_S)

Address: M2[0Ch]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	I2CSTREN	I2CGCEN	I2CSTOP	I2CSTART	I2CACK	I2CSTRS	—	I2CMODE	I2CMST	I2CEN
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Access	r	r	r	r	r	r	rw*	rw*	r	r	rw*	rw*	r	r	r	rw*

**Unrestricted read. Unrestricted write access when I2CBUSY = 0. Writes to I2CEN are disabled when I2CBUSY = 1.*

BIT	NAME	DESCRIPTION
15:10	—	Reserved. The user should not write to these bits.
9	I2CSTREN	I²C Slave Clock Stretch Enable. Setting this bit to 1 stretches the clock (holds SCL low) at the end of the clock cycle specified in I2CSTRS. Clearing this bit disables clock stretching.
8	I2CGCEN	I²C Slave General Call Enable. Setting this bit to 1 enables the I²C to respond to a general call address (address = 0000 0000). Clearing this bit to 0 disables the response to general call address.
7	I2CSTOP	This bit has no function when operating in slave mode.
6	I2CSTART	This bit has no function when operating in slave mode.
5	I2CACK	I²C Slave Data Acknowledge Bit. This bit selects the acknowledge bit returned by the I²C controller while acting as a receiver. Setting this bit to 1 generates a NACK (leaving SDA high). Clearing the I2CACK bit to 0 generates an ACK (pulling SDA low) during the acknowledgement cycle. This bit retains its value unless changed by software or hardware.
4	I2CSTRS	I²C Slave Clock Stretch Select. Setting this bit to 1 enables clock stretching after the falling edge of the 8th clock cycle. Clearing this bit to 0 enables clock stretching after the falling edge of the 9th clock cycle. This bit has no effect when clock stretching is disabled (I2CSTREN = 0).
3	—	Reserved. The user should not write to this bit.
2	I2CMODE	I²C Slave Transfer Mode Select. This bit reflects the actual R/$\overline{W}$ bit value in the current I²C transfer and is set by hardware. Software writing to this bit is ignored.
1	I2CMST	I²C Master Mode Enable. Setting this bit to 1 enables I²C master functionality on the SDA and SCL pins. See *SECTION 8: I²C-Compatible Master Interface* for more details. Setting this bit to 0 enables I²C slave functionality.
0	I2CEN	I²C Slave Enable. This bit enables the I²C slave function. When set to 1, I²C slave communication is enabled. When cleared to 0, the I²C function is disabled.

7.2.2 I²C Slave Status Register (I2CST_S)

Address: M2[01h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	I2CBUS	I2CBUSY	—	—	I2CSPI	I2CSCL	I2CROI	I2CGCI	I2CNACKI	—	I2CAMI	I2CTOI	I2CSTRI	I2CRXI	I2CTXI	I2CSRI
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r*	r*	r	r	rw	r*	rw	rw	rw*	r	rw	rw	rw*	rw*	rw	rw

**Set by hardware only.*

BIT	NAME	DESCRIPTION
15	I2CBUS	I²C Slave Bus Busy. This bit is set to 1 when a START/repeated START condition is detected and cleared to 0 when the STOP condition is detected. This bit is reset to 0 on all forms of reset or when I2CEN = 0. This bit is controlled by hardware and is read only.
14	I2CBUSY	I²C Slave Busy. This bit is used to indicate the current status of the I²C module. The I2CBUSY is set to 1 when the I²C controller is actively participating in a transaction. This bit is controlled by hardware and is read only.
13:12	—	Reserved. The user should not write to these bits.
11	I2CSPI	I²C Slave STOP Interrupt Flag. This bit is set to 1 when a STOP condition is detected. This bit must be cleared to 0 by software once set. Setting this bit to 1 by software causes an interrupt if enabled.
10	I2CSCL	I²C Slave SCL Status. This bit reflects the logic state of SCL signal. This bit is set to 1 when SCL is at a logic-high (1), and cleared to 0 when SCL is at a logic-low (0). This bit is controlled by hardware and is read-only.
9	I2CROI	I²C Slave Receiver Overrun Flag. This bit indicates a receive overrun when set to 1. This bit is set to 1 if the receiver has received 2 bytes since the last CPU read of I2CBUF_S. This bit can only be cleared to 0 by software reading the I2CBUF_S. Setting this bit to 1 by software causes an interrupt if enabled.
8	I2CGCI	I²C Slave General Call Interrupt Flag. This bit is set to 1 when the general call is enabled (I2CGCEN = 1) and the general call address (00h) is received. This bit must be cleared to 0 by software once set. Setting this bit to 1 by software causes an interrupt if enabled.
7	I2CNACKI	I²C Slave NACK Interrupt Flag. This bit is set by hardware to either a 1 if a NACK was received from the host or a 0 if an ACK was received from the host. The setting of this bit to a 1 causes an interrupt if enabled. This bit can be cleared to 0 by software once set.
6	—	Reserved. The user should not write to these bits.
5	I2CAMI	I²C Slave Address Match Interrupt Flag. This bit is set to 1 when the I²C controller receives an address that matches the contents of the slave address register (I2CSLA_S). This bit must be cleared to 0 by software once set. Setting this bit to 1 by software causes an interrupt if enabled.
4	I2CTOI	I²C Slave Timeout Interrupt Flag. This bit is set to 1 if SMBus timeout is enabled and SCL is low longer than 30ms. This bit must be cleared to 0 by software once set. Setting this to 1 causes an interrupt if enabled.
3	I2CSTRI	I²C Slave Clock Stretch Interrupt Flag. This bit indicates that the I²C slave controller is operating with clock stretching enabled and is currently holding the SCL clock signal low. The I²C controller releases SCL after this bit has been cleared to 0. This bit must be cleared to 0 by software once set. This bit is set by hardware only.
2	I2CRXI	I²C Slave Receive Ready Interrupt Flag. This bit indicates that a data byte has been received in the I²C buffer. This bit must be cleared by software once set. This bit is set by hardware only.
1	I2CTXI	I²C Slave Transmit Complete Interrupt Flag. This bit indicates that an address or a data byte has been successfully shifted out and the I²C controller has received an acknowledgment from the receiver (NACK or ACK). This bit must be cleared by software once set. Setting this bit to 1 by software causes an interrupt if enabled.
0	I2CSRI	I²C Slave START Interrupt Flag. This bit is set to 1 when a START condition (or restart) is detected. This bit must be cleared to 0 by software once set. Setting this bit to 1 by software causes an interrupt if enabled.

7.2.3 I²C Slave Interrupt Enable Register (I2CIE_S)

Address: M2[02h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	I2CSPIE	—	I2CROIE	I2CGCIE	I2CNACKIE	—	I2CAMIE	I2CTOIE	I2CSTRIE	I2CRXIE	I2CTXIE	I2CSRIE
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	rw	r	rw	rw	rw	r	rw	rw	rw	rw	rw	rw

BIT	NAME	DESCRIPTION
15:12	—	Reserved. The user should not write to these bits.
11	I2CSPIE	I²C Slave STOP Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when a STOP condition is detected (I2CSPI = 1). Clearing this bit to 0 disables the STOP detection interrupt.
10	—	Reserved. The user should not write to this bit.
9	I2CROIE	I²C Slave Receiver Overrun Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when a receiver overrun condition is detected (I2CROI = 1). Clearing this bit to 0 disables the receiver overrun detection interrupt.
8	I2CGCIE	I²C Slave General Call Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when a general call is detected (I2CGCI = 1). Clearing this bit to 0 disables the general call interrupt.
7	I2CNACKIE	I²C Slave NACK Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when a NACK is detected (I2CNACKI = 1). Clearing this bit to 0 disables the NACK detection interrupt.
6	—	Reserved. The user should not write to this bit.
5	I2CAMIE	I²C Slave Address Match Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when the I²C controller detects an address that matches the I2CSLA_S value (I2CAMI = 1). Clearing this bit to 0 disables the address match interrupt.
4	I2CTOIE	I²C Slave Timeout Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when an SMBus timeout condition is detected (I2CTOI = 1). Clearing this bit to 0 disables the timeout interrupt.
3	I2CSTRIE	I²C Slave Clock Stretch Interrupt Enable. Setting this bit to 1 generates an interrupt to the CPU when the clock stretch interrupt flag is set (I2CSTRI = 1). Clearing this bit disables the clock stretch interrupt.
2	I2CRXIE	I²C Slave Receive Ready Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when the receive ready interrupt flag is set (I2CRXI = 1). Clearing this bit to 0 disables the receive ready interrupt.
1	I2CTXIE	I²C Slave Transmit Complete Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when the transmit complete interrupt flag is set (I2CTXI = 1). Clearing this bit to 0 disables the transmit complete interrupt.
0	I2CSRIE	I²C Slave START Interrupt Enable. Setting this bit to 1 causes an interrupt to the CPU when a START condition is detected (I2CSRI = 1). Clearing this bit to 0 disables the START detection interrupt.

7.2.4 I²C Slave Address Register (I2CSLA_S)

Address: M2[0Fh]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	A6	A5	A4	A3	A2	A1	A0
Reset	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1
Access	r	r	r	R	r	r	r	r	r	rw	rw	rw	rw	rw	rw	rw

BIT	NAME	DESCRIPTION
15:7	—	Reserved. The user should not write to these bits.
6:0	A[6:0]	These address bits contain the address of the I²C slave interface. When a match to this address is detected, the I²C controller automatically acknowledges the host with the I2CACK bit value and the I2CAMI flag is set to 1. An interrupt is generated if enabled. The address in I2CSLA is the device slave address without the R/$\overline{W}$ bit. For example, the default value of I2CSLA_S is 1Bh, which produces a device slave address of 36h.

7.2.5 I²C Slave Data Buffer Register (I2CBUF_S)

Address: M2[00h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	D7	D6	D5	D4	D3	D2	D1	D0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	R	r	r	r	r	rw*	rw*	rw*	rw*	rw*	rw*	rw*	rw*

**Unrestricted read access. This register can be written to only when I2CBUSY = 0.*

BIT	NAME	DESCRIPTION
15:8	—	Reserved. The user should not write to these bits.
7:0	D[7:0]	Data for I²C transfer is read from or written to this location. The I²C transmit and receive buffers are separate, but both are addressed at this location.

7.2.6 SMBus Mode Selection Register (SMBUS)

Address: M3[04h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	RESET_S	RESET_M	SMB_MOD_S	SMB_MOD_M
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	rw	rw	rw	rw

This register contains bits that are used for both the I²C slave interface (SDA and SCL) and the I²C master interface (MSDA and MSCL). For operation of the slave interface, only the slave bits should be used.

BIT	NAME	DESCRIPTION
15:4	—	Reserved. The user should not write to these bits.
3	RESET_S	I²C Slave Reset Bit. This bit can be used by the software to unconditionally reset and disable the I²C slave interface. After at least one system clock cycle, this bit must be cleared by software. After this bit is toggled, all the relevant I²C slave registers need to be reinitialized.
2	RESET_M	This bit does not affect the slave I²C interface (SDA and SCL).
1	SMB_MOD_S	Slave SMBus Mode Operation. When this bit is set to a 1, SMBus timeout functionality is enabled for the I²C slave interface. When this bit is cleared to 0, the SMBus timeout functionality is disabled. See *7.1.9 SMBus Timeout* for more details.
0	SMB_MOD_M	This bit does not affect the slave I²C interface (SDA and SCL).

7.2.7 I²C Slave Clock Control Register (I2CCK_S)

Address: M2[0Dh]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	I2CCKH7	I2CCKH6	I2CCKH5	I2CCKH4	I2CCKH3	I2CCKH2	I2CCKH1	I2CCKH0	I2CCKL7	I2CCKL6	I2CCKL5	I2CCKL4	I2CCKL3	I2CCKL2	I2CCKL1	I2CCKL0
Reset	0	0	0	1	0	0	1	1	0	0	0	1	0	0	1	1
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

This register has no function when operating in slave mode.

7.2.8 I²C Slave Timeout Register (I2CTO_S)

Address: M2[0Eh]

Bit	7	6	5	4	3	2	1	0
Name	I2CTO7	I2CTO6	I2CTO5	I2CTO4	I2CTO3	I2CTO2	I2CTO1	I2CTO0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw

This register has no function when operating in slave mode.

SECTION 8: I²C-COMPATIBLE MASTER INTERFACE

This section contains the following information:

LIST OF FIGURES

SECTION 8: I²C-COMPATIBLE MASTER INTERFACE

The MAX31782 provides an I²C-compatible master controller that allows the MAX31782 to communicate with a slave device. The I²C master interface can be setup to provide system interrupts after each I²C event.

8.1 Detailed Description

8.1.1 Description of Master I²C Interface

The master I²C interface uses the MSDA and MSCL pins. These pins are the master I²C controller's connection to the SDA and SCL pins of an I²C bus. In addition to driving these pins, the I²C master port also senses the state of both MSDA and MSCL. This allows the I²C master port to offer bus error detection and allows a slave device to clock stretch. The MSDA and MSCL pins are open-drain output pins and require external pullup resistors to achieve a high logic level.

Unless explicitly stated, all references to SDA and SCL in this section refer to the SDA and SCL lines of the I²C bus, not the MAX31782's I²C slave interface SDA and SCL pins.

8.1.2 Default Operation

The I²C master controller is disabled by default. The I²C master controller is enabled by setting the I2CEN and I2CMST bits in the I2CCN_M register to a 1. Prior to the I²C master controller being used for communication, some software setup is required. This setup includes setting the clock rate, timeout period, and which I²C events should generate interrupts. The MAX31782 master I²C controller is not intended to be used on an I²C bus that has multiple masters connected to the bus.

8.1.3 I²C Clock Generation

In an I²C system, the master is responsible for generating the SCL signal. The MAX31782 I²C master controller provides complete control over the clock rate and duty cycle. The I²C master controller generates SCL from the system clock. The bit rate is controlled by the I²C clock control register (I2CCK_M).

The high period of SCL clock is defined by the high byte of the I²C clock control register (I2CCKH), whereas the low period of SCL is defined by the low byte (I2CCKL). The minimum clock high period is three system clocks while the minimum low period has to be at least five system clock periods. The I²C clock characteristics can be defined by the following equations:

- SCL Low Time = System Clock Period x (I2CCKL[7:0] + 1)
- SCL High Time = System Clock Period x (I2CCKH[7:0] + 1)
- I²C Clock Rate = System Clock Frequency/(I2CCLK[7:0] + I2CCKH[7:0] + 2)

One feature of the master I²C controller is that it also monitors SCL while the clock is being output. This allows the controller to ensure that the SCL level is at the desired level prior to beginning the count for SCL Low or High Time. Figure 8-1 illustrates the SCL sampling performed by the master I²C controller. When SCL is released by the master I²C controller,

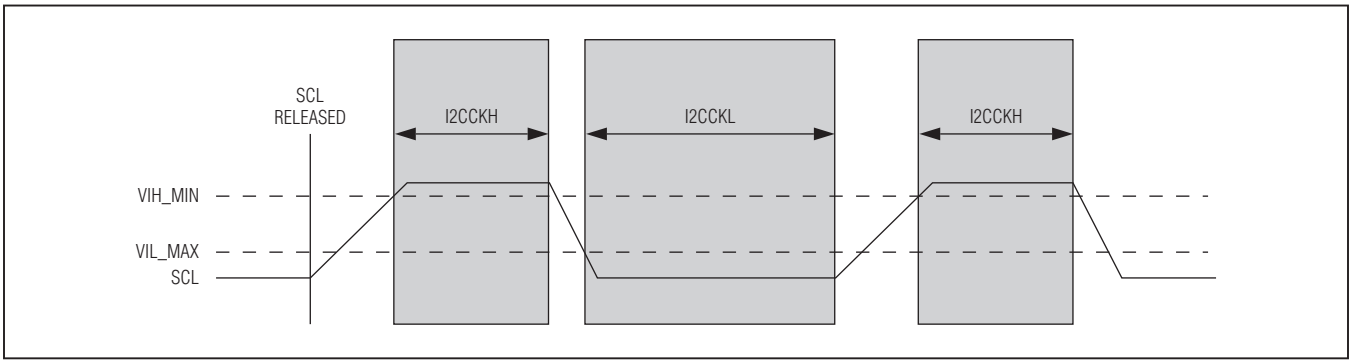


Figure 8-1. I²C Clock Generation

there is a rise time that is determined by the capacitive loading and pullup resistance on the SCL line. When the controller senses the SCL line has reached a high logic level, the count for SCL High Time begins. The same is true for a falling edge. The SCL Low Time only begins after the controller senses the SCL line at a low logic level.

Figure 8-1 also illustrates that the calculated I2C clock period will not be exactly accurate because the rise and fall time of SCL is not taken into consideration. The actual clock period will be the period set by the I2CCK_M register plus any rise and fall time.

The master I2C controller's ability to monitor the state of SCL allows the master to operate with slave devices that clock stretch. A slave device may clock stretch, or hold SCL low, while it is busy or processing data. The master I2C controller will always release SCL after holding it low for the SCL Low Time duration. By monitoring the state of SCL, the master I2C controller realizes that SCL has not been released and does not begin the SCL High Time count. Only after the master controller detects a high state on SCL will it begin the I2CCKH count. This is illustrated in Figure 8-2.

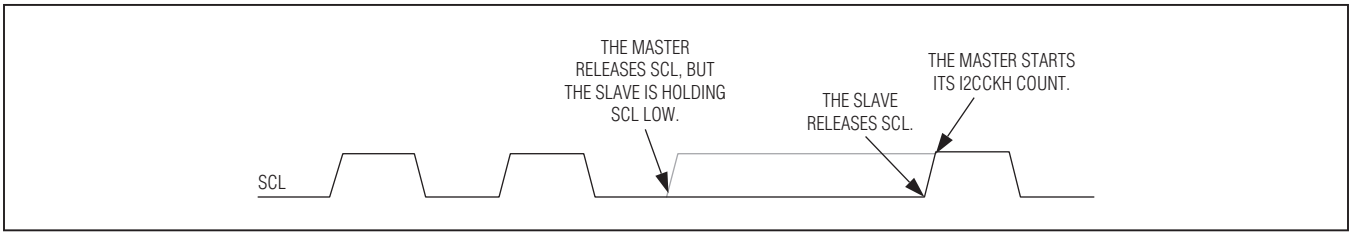


Figure 8-2. Master I2C Clock Generation During Slave Clock Stretching

8.1.4 Timeout

The master I2C controller has a programmable timeout function that allows the controller to recover from a bus error. The timeout period is determined by the setting of the I2C master timeout register (I2CTO_M) using the following equation:

$$\text{Timeout Period} = \text{I2C Bit Rate} \times (\text{I2CTO}[7:0]+1)$$

where I2C Bit Rate is determined by the setting of the I2CCK_M register. The timeout can be disabled by clearing the I2CTO_M register to 0. The I2C timeout timer starts counting:

- When the I2CSTART bit is set to 1. The I2C controller monitors the status of SDA and SCL until it can generate a START condition. If the controller has to wait longer than the period specified in the timeout register, the I2C controller concludes that there is a bus error and sets the I2CTOI flag.

  If the I2C controller has started a transfer (after the first bit rising edge), it waits for the current byte transfer to finish (after the 9th bit (acknowledge) has been transmit) before generating the START condition. In this case, the timeout timer starts counting after the end of the 9th bit low time.
- After the master I2C controller attempts to generate a STOP condition. If a STOP is not detected (I2CSPI = 1) during the timeout period, the I2CTOI flag is set.

  If the I2C controller has started a transfer (after the first bit rising edge), it waits for the current byte transfer to finish (after the 9th bit (acknowledge) has been transmit) before generating the STOP condition. In this case, the timeout timer starts counting after the end of the 9th bit low time.
- Whenever SCL goes low. If the SCL line is low for a period longer than specified in the timeout register, the I2C controller concludes that there is a bus error and sets the I2CTOI flag.

For all these cases, when the I2C timeout period is reached, the I2CTOI flag is set. The setting of I2CTOI can generate an interrupt if enabled. If the master I2C controller is in the process of transferring data when the timeout occurs, the controller aborts the current transfer and clears the I2CBUSY flag. The I2CBUS flag continues to be set until a STOP condition is detected or I2CEN is set to 0.

8.1.5 Generating a START

To initiate a data transfer, the I2C master controller must first issue a START command. The master I2C controller's flow when attempting to issue a START command is shown in Figure 8-3. A START command is generated by setting the I2CSTART bit to 1. The I2C controller monitors the status of SDA and SCL until it can generate a START condition. If the controller has to wait longer than the period specified in the timeout register, the I2C controller concludes that there is a bus error and sets the I2CTOI flag.

If the bus is not busy, the I2C master controller attempts to generate a START. Because the SDA line is feedback into the device, when the master generates a START, it can also detect the START condition. When a start condition is detected, the I2C START interrupt flag (I2CSRI) will be set and an interrupt will be generated if enabled. The I2CBUS bit will be set to indicate that the I2C bus is now in use and the I2CSTART bit will be cleared.

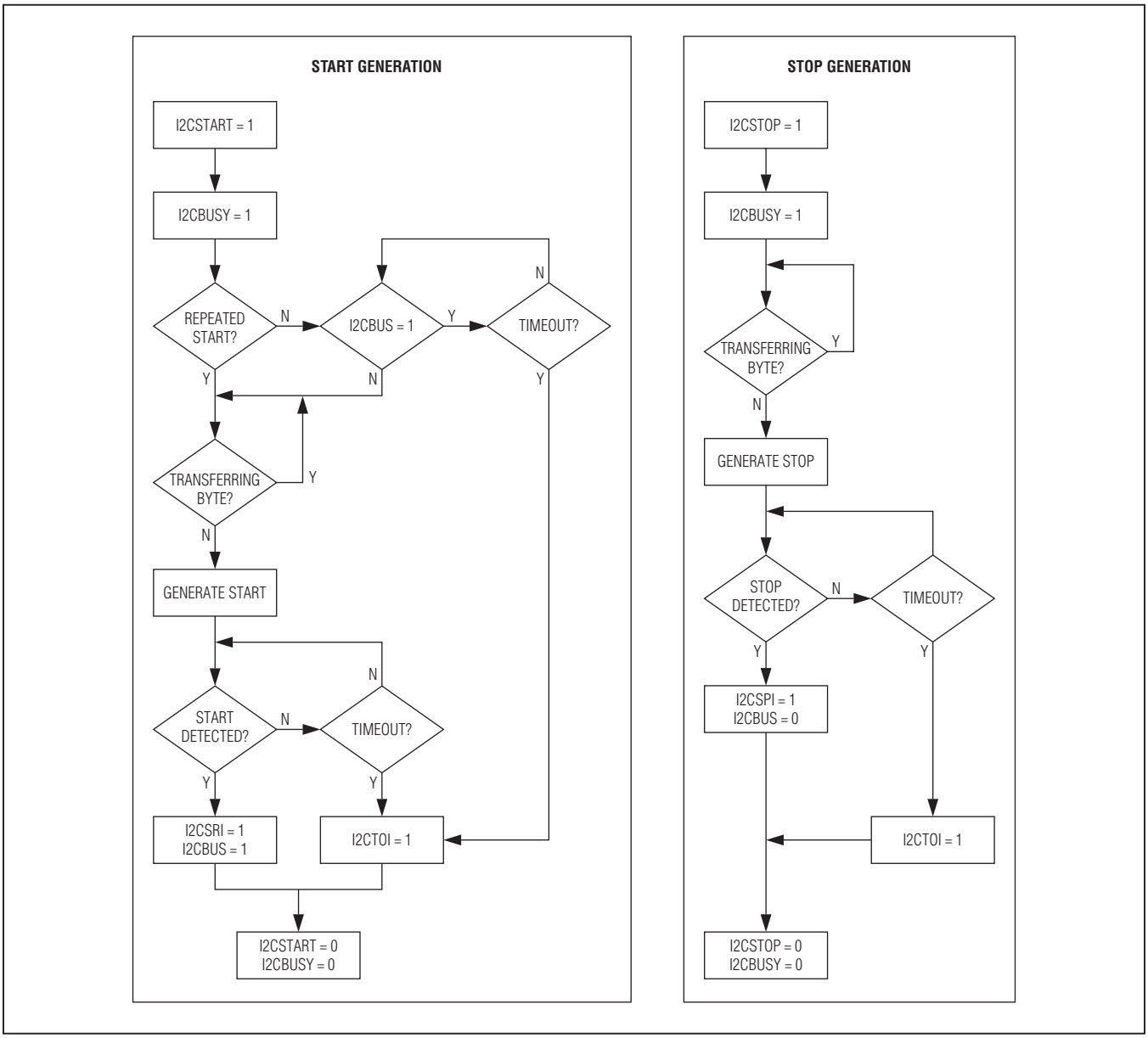


Figure 8-3. Master I2C Generated START and STOP

When the I2CSTART bit is set to a 1, the I²C controller starts its timeout timer if enabled (I2CTO_M ≠ 0). If the timer expires before the START can be generated, the I²C timeout interrupt flag (I2CTOI) will be set and an interrupt generated if enabled. If a timeout occurs, the I²C master controller will reset to an idle state and the I2CSTART bit will be cleared.

If the I2CSTART bit is set when the I²C controller is in the middle of a byte transfer (after the first bit rising edge), the controller will wait for the current byte transfer to finish (after the 9th bit) before generating the START condition. In this case, the timeout timer will not start counting until after the end of the 9th bit low time.

8.1.6 Generating a STOP

To end an I²C transfer, a STOP must be transmit. A STOP is generated by setting the I2CSTOP bit. The master I²C controller's flow when attempting to issue a STOP command is shown in Figure 8-3.

If the I2CSTOP bit is set when the I²C controller is in the middle of a byte transfer (after the first bit rising edge), it will wait for the current byte transfer to finish (after the 9th bit) before generating the STOP condition.

Because the SDA line is feedback into the device, when the master generates a STOP, it will also detect the STOP condition. When a STOP condition is detected, the I²C STOP interrupt flag (I2CSPI) will be set and an interrupt will be generated enabled. The I2CBUS bit will be cleared to indicate that the I²C bus is now idle and the I2CSTOP bit will be cleared.

When the master I²C controller attempts to generate the STOP condition, it will also start the timeout timer if this feature is enabled. If a timeout is generated before the STOP condition is detected, a timeout will occur. When a timeout occurs, the I2CTOI bit will be set, which can generate an interrupt if enabled, and the I2CSTOP bit will also be cleared to 0.

8.1.7 Transmitting a Slave Address

The first byte after an I²C start or restart condition is the slave address byte. This byte, which is transmit by the master, contains seven bits of slave address followed by the R/$\overline{W}$ bit. The transmission of the slave address begins with writing the address to I2CBUF_M.

The slave address written to I2CBUF_M is a seven-bit address that does not contain the R/$\overline{W}$ bit. Figure 8-4 shows the format for slave address 36h. The address bits A[6:0], which is the slave address excluding the R/$\overline{W}$ bit is written to I2CBUF_M[6:0]. For example, to transmit slave address 36h, I2CBUF_M must be set to 1Bh. The I2CMODE bit will be insert into the R/$\overline{W}$ bit when the slave address is transmit.

A6	A5	A4	A3	A2	A1	A0	R/$\overline{W}$
0	0	1	1	0	1	1	0

SLAVE ADDRESS 36h SHOWN.

Figure 8-4. Slave Address Format

After the slave address has been written to I2CBUF_M, the I²C master controller will set the I2CBUSY bit to indicate the controller is actively participating in a transaction. The seven bits in I2CBUF_M[6:0] will be transmit on SDA. The data for the 8th bit transmit, which is the R/$\overline{W}$ bit, is the value of the I2CMODE bit. The I²C master then issues the 9th clock, which is for the acknowledge bit, and reads SDA for an acknowledgment from a slave device. The I²C master controller then performs the following steps. This is illustrated in Figure 8-5.

- Set the I2CNACKI bit with the value of the received acknowledgement.
- The I2CTXI bit will then be set to indicate a byte was transmit.
- Clear the I2CBUSY flag.

Upon transmitting the slave data byte (7 bits of slave address + R/$\overline{W}$ bit + acknowledge), the I2C master controller will enter one of the three states.

- Data Transmit: The I2CMODE (R/$\overline{W}$) bit was set to a 0, indicating that the master will be writing data to a slave device. The MAX31782 will retain control of the SDA line.
- Data Receive: The I2CMODE (R/$\overline{W}$) bit was set to a 1, indicating that the master will be receiving data from a slave. The MAX31782 releases control of SDA to allow a slave device to output data. The MAX31782 master I2C controller automatically begins clocking bytes of data from the slave.
- The slave address was NACKed. The master I2C controller will retain control of SDA and is able to transmit data.

8.1.8 Transmitting Data

The MAX31782 I2C master controller enters into data transmission mode after transmitting a slave address with the R/$\overline{W}$ bit (I2CMODE) set to a 0. The steps of data transmission are shown in Figure 8-5. Data transmission is started by software loading a byte of data into the I2CBUF_M register. Loading I2CBUF_M causes the I2CBUSY bit to be set. Once set, writes to I2CBUF_M will be ignored. The first bit of data (most significant bit) will be shifted to SDA when SCL is low. Each of the next seven bits will then be shifted following high to low transitions of SCL.

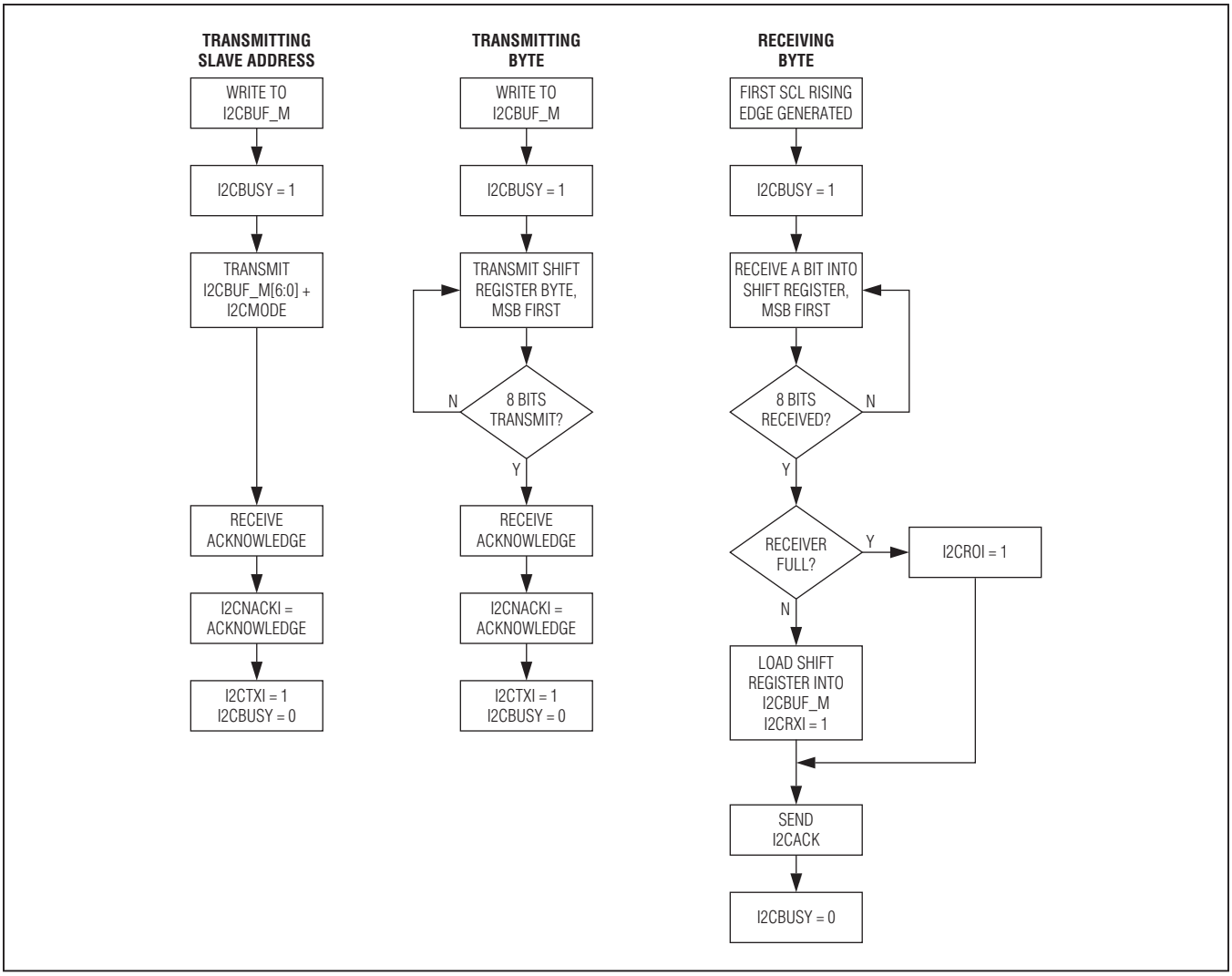


Figure 8-5. Master I2C Data Flowchart

Following the 8th bit of data (least significant bit) being shifted to SDA, the SDA line will be released by the MAX31782 master controller. This allows the slave to signal an ACK or NACK during the 9th clock cycle. The MAX31782 I2C master controller samples the acknowledge bit following the 9th SCL rising edge. After the acknowledge bit is sampled, the MAX31782 I2C master controller will perform the following tasks:

- Set or clear the I2CNACKI flag to reflect the received acknowledge bit. The setting of I2CNACKI can generate an interrupt if enabled.
- Set the I2CTXI flag to indicate that the I2C master controller transmit a complete byte. This can generate an interrupt if enabled.
- Clear the I2CBUSY flag to indicate that the I2C master controller is not actively participating in the transfer of data.

8.1.9 Receiving Data

The MAX31782 I2C master controller enters data reception mode after transmitting a slave address with the R/$\overline{W}$ bit (I2CMODE) set to a 1. The steps of data reception are shown in Figure 8-5. After transmitting the slave address, the master controller will switch to receiver mode and automatically begin outputting SCL clock pulses and shifting in data from SDA.

When receiving data, the MAX31782 I2C master controller uses a double buffer consisting of the I2CBUF_M register and the shift register. This allows the I2C module to continue receiving data while the previous data byte is being processed. When a full byte of data (8 bits) has been received by the I2C master controller, the master must send an acknowledgement to the slave. This occurs during the 9th clock cycle when the value in I2CACK is transmit to the slave.

After a complete byte (8 bits) of data are received, the I2C master controller will attempt to copy the received data from the shift register to I2CBUF_M. There are two possible results from the I2C master controller's attempt to copy the shift register to I2CBUF_M.

1) If I2CBUF_M is empty, the I2C master controller will copy the data from the shift register into I2CBUF_M. The I2CRXI flag will be set to indicate a received byte is ready to be read. The setting of I2CRXI can generate an interrupt if enabled.
2) If I2CBUF_M is full, the data in the shift register cannot be copied into I2CBUF_M. This causes a receive overrun condition. The receive overrun flag, I2CROI, will be set which can generate an interrupt if enabled. I2CBUF_M will be full if it was not read by software following the reception of a previous byte.

After receiving a byte of data and the I2CRXI flag being set, it is up to software to read I2CBUF_M prior to a second byte being received. Reading the I2CBUF_M register returns the received data and also clears I2CBUF_M. As long as the previous byte of data is read from I2CBUF_M before the next byte has completed, receive overrun will not occur.

When receive overrun is detected and I2CROI bit is set, the MAX31782 master I2C controller will stop outputting SCL clocks and not clock the acknowledge bit until the receive overrun condition is cleared. The receive overrun condition and the I2CROI flag can only be cleared by software reading the first byte received from I2CBUF_M. When the receive overrun condition is cleared, the I2C master controller will copy the second byte that was received into I2CBUF_M, and again set I2CRXI to indicate a byte of data was received. The I2C master controller will resume clocking SCL after satisfying SCL low time requirements.

The master I2C controller will continue to automatically clock bytes of data until any of the following conditions occur.

1) A receive overrun condition occurs.
2) A STOP command is issued (I2CSTOP = 1) prior to the master I2C controller beginning to clock a new byte.
3) The master I2C controller has clock stretching enabled and the clock is currently being held low by the master.

8.1.10 I2C Master Clock Stretching

The master I2C controller is capable of clock stretching at the end of each transfer cycle. Clock stretching is when SCL is held low. If the I2C clock stretch enable bit (I2CSTREN) is set to a 1, the I2C controller holds SCL low after the clock pulse defined by the I2C clock stretch select bit (I2CSTRS). If I2CSTRS = 0, the I2C controller holds SCL low after the falling edge of the 9th clock pulse. If I2CSTRS = 1, the I2C controller holds SCL low after the falling edge of the 8th clock pulse. When the I2C controller is holding SCL low, the I2C clock stretch interrupt flag (I2CSTRI) is set, which can generate an interrupt if enabled. The I2C slave controller holds SCL low until I2CSTRI is cleared to 0 by software.

If clock stretching is enabled after the 8th clock pulse, the master I2C controller will continue outputting the value of the I2CACK bit until clock stretching is released by clearing I2CSTRI. This allows software time to examine the data that was received prior to sending an ACK or NACK to the slave. The continuous output of I2CACK will occur even if the master I2C controller is transmitting data. In this mode, the slave should be sending the acknowledgement. To allow the slave to send the proper acknowledgement, the I2CACK bit should be set to a 1, which prompts the master I2C controller to release SDA.

The master I2C controller may need to use clock stretching when receiving data from a slave. When receiving data, the master I2C controller automatically generates clock pulses. Without using clock stretching, this automatic clock generation is only halted when a STOP command is issued or a receive overrun occurs. If clock stretching is enabled, software can control when each byte of data is clocked from the slave device.

8.1.11 Resetting the I2C Master Controller

The I2C master controller can be reset by setting the RESET_M bit in the SMBUS register. After a delay of at least one system clock, this bit needs to be cleared by software to complete the reset. A reset will force the master I2C controller to release both MSDA and MSCL if they are being held low by the I2C master controller. A reset will also turn off the I2C master controller (I2CEN = 0), reset all of the master I2C registers, and reset the I2C master controller's internal state machine. Following a reset, the I2C master controller must be reinitialized before it can be used again.

After a reset, the master I2C controller will be in a known state but the slave devices may be in an unknown state. It is recommended that the master I2C controller attempts to reset the slave devices prior to beginning communication. A reset of slave devices can be performed by outputting at least nine clock pulses on the MSCL line while MSDA is high. This easiest way to achieve this is to use MSDA and MSCL as GPIO pins (see *SECTION 11: General-Purpose Input/Output (GPIO) Pins*) while the master I2C controller is disabled (I2CEN = 0). After the nine clock pulses, a STOP command should be generated. This can be done either using GPIO, or by enabling the master I2C controller and generating a STOP.

8.1.12 Operation as a Slave

The MAX31782 contains two I2C interfaces, the master (MSDA and MSCL) and slave (MAX31782 SDA and SCL pins). These are two totally separate blocks within the MAX31782. However, both of the blocks are identical. Because of this, it is possible to operate the master as a slave and also operate the slave as a master.

To operate the master (MSDA and MSCL) as a slave I2C interface, the I2CMST bit in I2CCN_M needs to be set to a 0. When the master is operating as a slave, it will use the same registers (I2CCN_M, I2CST_M, etc) that it uses for master operation. However, the bits in these registers will have different functionality, as described in *SECTION 7: I2C-Compatible Slave Interface*. The SMBUS.RESET_M bit can still be used to reset this interface (MSDA and MSCL) when operating as a slave. The SMBUS.SMB_MOD_M bit only affects the interface when it is operating as a slave. See *SECTION 7: I2C-Compatible Slave Interface* for details on initializing and using a slave I2C interface.

8.1.13 GPIO

When the I2C master controller is disabled (I2CEN = 0), the MSDA and MSCL pins can be used as GPIO pins. The MSDA pin is mapped to GPIO port P2.7 and MSCL is mapped to GPIO port P2.6. When used as GPIO outputs, the MSDA and MSCL pins are only capable of being open-drain outputs. See *SECTION 11: General-Purpose Input/Output (GPIO) Pins* for more information on using MSDA and MSCL as GPIO pins.

8.2 I²C Master Controller Register Descriptions

Following are the registers that are used to control the I²C master interface, which is the MSDA and MSCL pins. These registers are used to control the I²C master interface if it is operating as either a master or slave. The bit descriptions below detail how to use these registers when operating in master mode. When operating in slave mode, some of the bits and registers have different functionality. See *SECTION 7: I²C-Compatible Slave Interface* for more information on how to control the I²C master interface when it is operating as a slave.

8.2.1 I²C Master Control Register (I2CCN_M)

Address: M1[0Ch]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	I2CSTREN	I2CGCEN	I2CSTOP	I2CSTART	I2CACK	I2CSTRS	—	I2CMODE	I2CMST	I2CEN
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Access	r	r	r	r	r	r	rw	rw	rw	rw	rw	rw	r	rw*	rw*	rw*

**Unrestricted read. Unrestricted write access when I2CBUSY = 0. Writes to I2CEN are disabled when I2CBUSY = 1.*

BIT	NAME	DESCRIPTION
15:10	—	Reserved. The user should not write to these bits.
9	I2CSTREN	I²C Master Clock Stretch Enable. Setting this bit to 1 stretches the clock (hold SCL low) at the end of the clock cycle specified by I2CSTRS. Clearing this bit disables clock stretching.
8	I2CGCEN	This bit has no function when operating in master mode.
7	I2CSTOP	I²C STOP Enable. Setting this bit to 1 generates a STOP condition. This bit is automatically cleared to 0 after the STOP condition has been generated. The setting of I2CSTOP starts the timeout timer if enabled. If the timeout timer expires before the STOP condition is generated, the I2CTOI flag is set, which can generate an interrupt if enabled. A timeout also clears the I2CSTOP bit.
6	I2CSTART	I²C START Enable. Setting this bit to 1 generates a START or repeated START condition. This bit is automatically cleared to 0 after the START condition has been generated. The setting of I2CSTART starts the timeout timer if enabled. If the timeout timer expires before the START condition is generated, the I2CTOI flag is set, which can generate an interrupt if enabled. A timeout also clears the I2CSTART bit.
5	I2CACK	I²C Master Data Acknowledge Bit. This bit selects the acknowledge bit returned by the master I²C controller while acting as a receiver. Setting this bit to 1 generates a NACK (leaving SDA high). Clearing the I2CACK bit to 0 generates an ACK (pulling SDA low) during the acknowledgement cycle. This bit retains its value unless changed by software or hardware.
4	I2CSTRS	I²C Master Clock Stretch Select. Setting this bit to 1 enables clock stretching after the falling edge of the 8th clock cycle. Clearing this bit to 0 enables clock stretching after the falling edge of the 9th clock cycle. This bit has no effect when clock stretching is disabled (I2CSTREN = 0).
3	—	Reserved. The user should not write to this bit.
2	I2CMODE	I²C Master Transfer Mode Select. When the I2CMODE bit is set to 1, the master is operating in receiver mode (reading from slave). When the I2CMODE bit is cleared to 0, the master is operating in transmitter mode (writing to slave).
1	I2CMST	I²C Master Mode Enable. Setting this bit to 1 enables I²C master functionality on the MSDA and MSCL pins. Setting this bit to 0 enables I²C slave functionality. See *SECTION 7: I²C-Compatible Slave Interface* section for more details.
0	I2CEN	I²C Enable. This bit enables the I²C master interface. When set to 1, the I²C master interface is enabled. When cleared to 0, the I²C function is disabled.

***Note:** The I2CSTART and I2CSTOP bits are mutually exclusive. If both bits are set at the same time, it is considered an invalid operation and the I²C controller ignores the request and resets both bits to 0. Setting the I2CSTART bit to 1 while I2CSTOP = 1 is an invalid operation and is ignored, leaving the I2CSTART bit cleared to 0.*

8.2.2 I²C Master Status Register (I2CST_M)

Address: M1[01h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	I2CBUS	I2CBUSY	—	—	I2CSPI	I2CSCL	I2CROI	I2CGCI	I2CNACKI	—	I2CAMI	I2CTOI	I2CSTRI	I2CRXI	I2CTXI	I2CSRI
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r*	r*	r	r	rw	r*	rw	rw	rw*	r	rw	rw	rw*	rw*	rw	rw

**Set by hardware only.*

BIT	NAME	DESCRIPTION
15	I2CBUS	I²C Master Bus Busy. This bit is set to 1 when a START/repeated START condition is detected and cleared to 0 when the STOP condition is detected. This bit is reset to 0 when I2CEN = 0. This bit is controlled by hardware and is read only.
14	I2CBUSY	I²C Master Busy. This bit is used to indicate the current status of the I²C controller. The I2CBUSY is set to 1 when the I²C controller is actively participating in a transaction. This bit is controlled by hardware and is read only.
13:12	—	Reserved. The user should not write to these bits.
11	I2CSPI	I²C Master STOP Interrupt Flag. This bit is set to 1 when a STOP condition is detected. This bit must be cleared to 0 by software once set. Setting this bit to 1 by software causes an interrupt if enabled.
10	I2CSCL	I²C Master SCL Status. This bit reflects the logic state of the SCL signal. This bit is set to 1 when SCL is at a high logic level and cleared to 0 when SCL is at a low logic level. This bit is controlled by hardware and is read only.
9	I2CROI	I²C Master Receiver Overrun Flag. This bit indicates a receive overrun when set to 1. This bit is set to 1 if the receiver has received 2 bytes since the last software reading of I2CBUF_M. This bit can only be cleared to 0 by software reading I2CBUF_M. Setting this bit to 1 by software causes an interrupt if enabled.
8	I2CGCI	This bit has no function when operating in master mode.
7	I2CNACKI	I²C Master NACK Interrupt Flag. This bit is set by hardware to a 1 if a NACK was received from a slave or a 0 if an ACK was received from a slave. The setting of this bit to a 1 by hardware causes an interrupt if enabled. This bit can be cleared to 0 by software once set. This bit is set by hardware only.
6	—	Reserved. The user should not write to this bit.
5	I2CAMI	This bit has no function when operating in master mode.
4	I2CTOI	I²C Master Timeout Interrupt Flag. This bit is set to a 1 if the I²C controller cannot generate a START or STOP condition or the SCL low time is greater than the timeout value specified in the I2CTO_M register. This bit must be cleared to 0 by software once set. Setting this bit to 1 by software causes an interrupt if enabled.
3	I2CSTRI	I²C Master Clock Stretch Interrupt Flag. This bit indicates that the I²C master controller is operating with clock stretching enabled and is currently holding the SCL clock signal low. The I²C controller releases SCL after this bit has been cleared to 0. This bit must be cleared to 0 by software once set. This bit is set by hardware only.
2	I2CRXI	I²C Master Receive Ready Interrupt Flag. This bit indicates that a data byte has been received in I2CBUF_M. This bit must be cleared by software once set. This bit is set by hardware only.
1	I2CTXI	I²C Master Transmit Complete Interrupt Flag. This bit indicates that an address or a data byte has been successfully shifted out and the I²C controller has received an acknowledgment from the receiver (ACK or NACK). This bit must be cleared by software once set. Setting this bit to 1 by software causes an interrupt if enabled.
0	I2CSRI	I²C Master START Interrupt Flag. This bit is set to 1 when a START condition (or restart) is detected. This bit must be cleared to 0 by software once set. Setting this bit to 1 by software causes an interrupt if enabled.

8.2.3 I²C Master Interrupt Enable Register (I2CIE_M)

Address: M1[02h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	I2CSPIE	—	I2CROIE	I2CGCIE	I2CNACKIE	—	I2CAMIE	I2CTOIE	I2CSTRIE	I2CRXIE	I2CTXIE	I2CSRIE
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	rw	r	rw	rw	rw	r	rw	rw	rw	rw	rw	rw

BIT	NAME	DESCRIPTION
15:12	—	Reserved. The user should not write to these bits.
11	I2CSPIE	I²C Master STOP Interrupt Enable. Setting this bit to 1 enables an interrupt when a STOP condition is detected (I2CSPI = 1). Clearing this bit to 0 disables the STOP detection interrupt.
10	—	Reserved. The user should not write to this bit.
9	I2CROIE	I²C Master Receiver Overrun Interrupt Enable. Setting this bit to 1 enables an interrupt when a receiver overrun condition is detected (I2ROI = 1). Clearing this bit to 0 disables the receiver overrun detection interrupt.
8	I2CGCIE	This bit has no function when operating in master mode.
7	I2CNACKIE	I²C Master NACK Interrupt Enable. Setting this bit to 1 enables an interrupt when a NACK is detected (I2CNACKI = 1). Clearing this bit to 0 disables the NACK detection interrupt.
6	—	Reserved. The user should not write to this bit.
5	I2CAMIE	This bit has no function when operating in master mode.
4	I2CTOIE	I²C Master Timeout Interrupt Enable. Setting this bit to 1 enables an interrupt when a timeout condition is detected (I2CTOI = 1). Clearing this bit to 0 disables the timeout interrupt.
3	I2CSTRIE	I²C Master Clock Stretch Interrupt Enable. Setting this bit to 1 enables an interrupt when the clock stretch interrupt flag is set (I2CSTRI = 1). Clearing this bit disables the clock stretch interrupt.
2	I2CRXIE	I²C Master Receive Ready Interrupt Enable. Setting this bit to 1 enables an interrupt when the receive ready interrupt flag is set (I2CRXI = 1). Clearing this bit to 0 disables the receive ready interrupt.
1	I2CTXIE	I²C Master Transmit Complete Interrupt Enable. Setting this bit to 1 enables an interrupt when the transmit complete interrupt flag is set (I2CTXI = 1). Clearing this bit to 0 disables the transmit complete interrupt.
0	I2CSRIE	I²C Master START Interrupt Enable. Setting this bit to 1 enables an interrupt when a START condition is detected (I2CSRI = 1). Clearing this bit to 0 disables the START detection interrupt.

8.2.4 I²C Master Data Buffer Register (I2CBUF_M)

Address: M1[00h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	D7	D6	D5	D4	D3	D2	D1	D0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	rw*	rw*	rw*	rw*	rw*	rw*	rw*	rw*

**Unrestricted read access. This register can be written to only when I2CBUSY = 0.*

BIT	NAME	DESCRIPTION
15:8	—	Reserved. The user should not write to these bits.
7:0	D[7:0]	Data for I²C transfer is read from or written to this location. The I²C transmit and receive buffers are separate but both are addressed at this location.

8.2.5 I2C Master Clock Control Register (I2CCK_M)

Address: M1[0Dh]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	I2CCKH7	I2CCKH6	I2CCKH5	I2CCKH4	I2CCKH3	I2CCKH2	I2CCKH1	I2CCKH0	I2CCKL7	I2CCKL6	I2CCKL5	I2CCKL4	I2CCKL3	I2CCKL2	I2CCKL1	I2CCKL0
Reset	0	0	0	1	0	0	1	1	0	0	0	1	0	0	1	1
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

BIT	NAME	DESCRIPTION
15:8	I2CCKH[7:0]	These bits define the high period of the I2C clock. This period is defined by the number of system clocks. The high time duration is calculated using the following equation: I2C High Time Period = System Clock Period x (I2CCKH[7:0] + 1) I2CCKH[7:0] must be set to a minimum value of 2 to ensure proper operation. Any value less than 2 is set to 2.
7:0	I2CCKL[7:0]	These bits define the low period of the I2C clock. This period is defined by the number of system clocks. The low time duration is calculated using the following equation: I2C Low Time Period = System Clock Period x (I2CCKL[7:0] + 1) I2CCKL[7:0] must be set to a minimum value of 4 to ensure proper operation. Any value less than 4 is set to 4.

8.2.6 I2C Master Timeout Register (I2CTO_M)

Address: M1[0Eh]

Bit	7	6	5	4	3	2	1	0
Name	I2CTO7	I2CTO6	I2CTO5	I2CTO4	I2CTO3	I2CTO2	I2CTO1	I2CTO0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw

The I2CTO_M register determines the length of the timeout interval. The timeout interval is defined by the number of I2C bit periods (SCL high + SCL low). When cleared to 00h, the timeout function is disabled. When set to any other value, the I2C controller waits until the timeout expires and sets the I2CTOI flag. The timeout period is:

$$\text{I2C Timeout} = \text{I2C Bit Rate} \times (\text{I2CTO}[7:0] + 1)$$

The timeout timer resets to 0 and starts to count after each of the following events.

- The I2CSTART bit is set.
- The I2CSTOP bit is set.
- Any time SCL goes low.

8.2.7 I2C Master Address Register (I2CSLA_M)

Address: M1[0Fh]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	A6	A5	A4	A3	A2	A1	A0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw	rw	rw	rw

This register has no function when operating in master mode.

8.2.8 SMBus Mode Selection Register (SMBUS)

Address: M3[04h]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	RESET_S	RESET_M	SMB_MOD_S	SMB_MOD_M
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	rw	rw	rw	rw

This register contains bits that are used for both the I2C slave interface (SDA and SCL) and the I2C master interface (MSDA and MSCL). For operation of the master interface, only the master bits should be used.

BIT	NAME	DESCRIPTION
15:4	—	Reserved. The user should not write to these bits.
3	RESET_S	This bit does not affect the master I2C interface (MSDA and MSCL).
2	RESET_M	I2C Master Reset Bit. This bit can be used by the software to unconditionally reset and disable the I2C master interface. After at least one system clock cycle, this bit must be cleared by software. After this bit is toggled, all the relevant I2C master registers need to be reinitialized.
1	SMB_MOD_S	This bit does not affect the master I2C interface (MSDA and MSCL).
0	SMB_MOD_M	This bit enables the SMBUS timeout feature only when the master I2C interface (MSDA and MSCL) is enabled to be a slave interface. See the *8.1.12 Operation as a Slave* section for more details.

SECTION 9: PWM OUTPUTS

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 9: PWM OUTPUTS

The MAX31782 provides six independent PWM output pins that can be used for power-supply margining or fan speed control. When the PWM output functionality of a pin is disabled, that pin can be used as a general-purpose input/output (GPIO). A diagram for one individual PWM output block is shown in Figure 9-1.

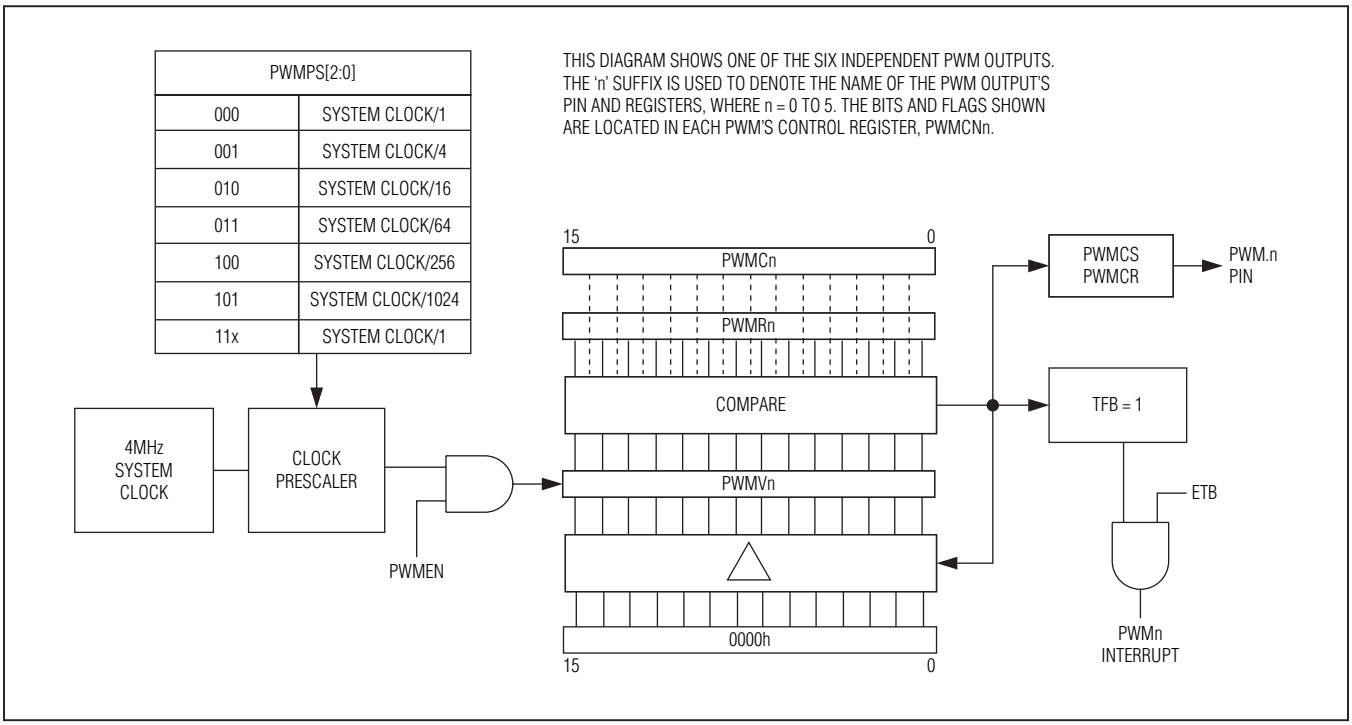


Figure 9-1. PWM Output Block Diagram

9.1 Detailed Description

9.1.1 PWM Pin Mapping and GPIO Muliplexing

Table 9-1 shows the mapping of each PWM Output. This table also shows that the PWM pins are mapped to GPIO port P1[5:0]. When a PWM output pin's functionality is disabled (PWMCS = 0 or PWMCR = 0), the pin can be used as a GPIO. See *SECTION 11: General-Purpose Input/Output (GPIO) Pins* for information on using the PWM pins as GPIO.

Table 9-1. PWM/GPIO Pin Multiplexing

PWM OUTPUT PIN	MAX31782 PIN NUMBER	GPIO PIN
PWM.0	28	P1.0
PWM.1	26	P1.1
PWM.2	24	P1.2
PWM.3	20	P1.3
PWM.4	18	P1.4
PWM.5	16	P1.5

9.1.2 PWM Operation

A PWM output pin is enabled when either the PWMCS or PWMCR bit is set to 1. Table 9-2 describes how these bits determine the specific PWM operation. The PWM counter does not begin operating until the PWMEN bit is set to 1.

Table 9-2. PWM Output Modes

PWMCS:PWMCR	PWM MODE	TBB PIN FUNCTION	INITIAL STATE WHEN PWMEN = 0	NOTES
00	None	None (Disabled)	No change	
01	Reset	Reset on PWMCn Match Set on 0000h	Low	Will not output a 0% duty cycle.
10	Set	Set on PWMCn Match Reset on PWMRn Match	High	Will not output a 100% duty cycle
11	Toggle	Toggle on PWMCn Match	No change	

The PWM can provide up to 16-bit resolution of the frequency or duty cycle. A timed setting or clearing of the PWM.n pin can also be generated without the need for the MAX31782 to time the event or use GPIO. This is accomplished by setting the compare register (PWMCn) to a value greater than the reload register (PWMRn). This functionality is illustrated in Figure 9-2 and Figure 9-3. The PWM can operate in a normal up-count-only configuration (DCEN = 0), or in a count up/down configuration (DCEN = 1).

9.1.3 Normal PWM Output Operation

When operating in PWM output mode and configured for up count (DCEN = 0), the value in PWMVn is incremented until it reaches the reload value, PWMRn. At this point, PWMVn reloads with 0000h, the TFB flag is set (which can generate an interrupt if enabled), and counting continues. Figure 9-2 illustrates the PWM waveforms when the PWM is operating with DCEN = 0. The period of the PWM waveform is set by the value in the PWMRn register. The set and reset modes provide similar functionality. The formulas for period and duty cycle are:

$$\text{PWM PERIOD} = (\text{PWMRn} + 1) \times \text{PWM.n CLOCK PERIOD}$$

$$\text{Duty Cycle in Set Mode} = \frac{\text{PWMRn} - \text{PWMCn}}{\text{PWMRn} + 1}$$

$$\text{Duty Cycle in Reset Mode} = \frac{\text{PWMCn}}{\text{PWMRn} + 1}$$

The toggle mode generates a 50% duty-cycle waveform if the PWMCn register remains fixed. The period of the waveform is:

$$\text{PERIOD} = 2 \times (\text{PWMRn} + 1) \times \text{PWM.n CLOCK PERIOD}$$

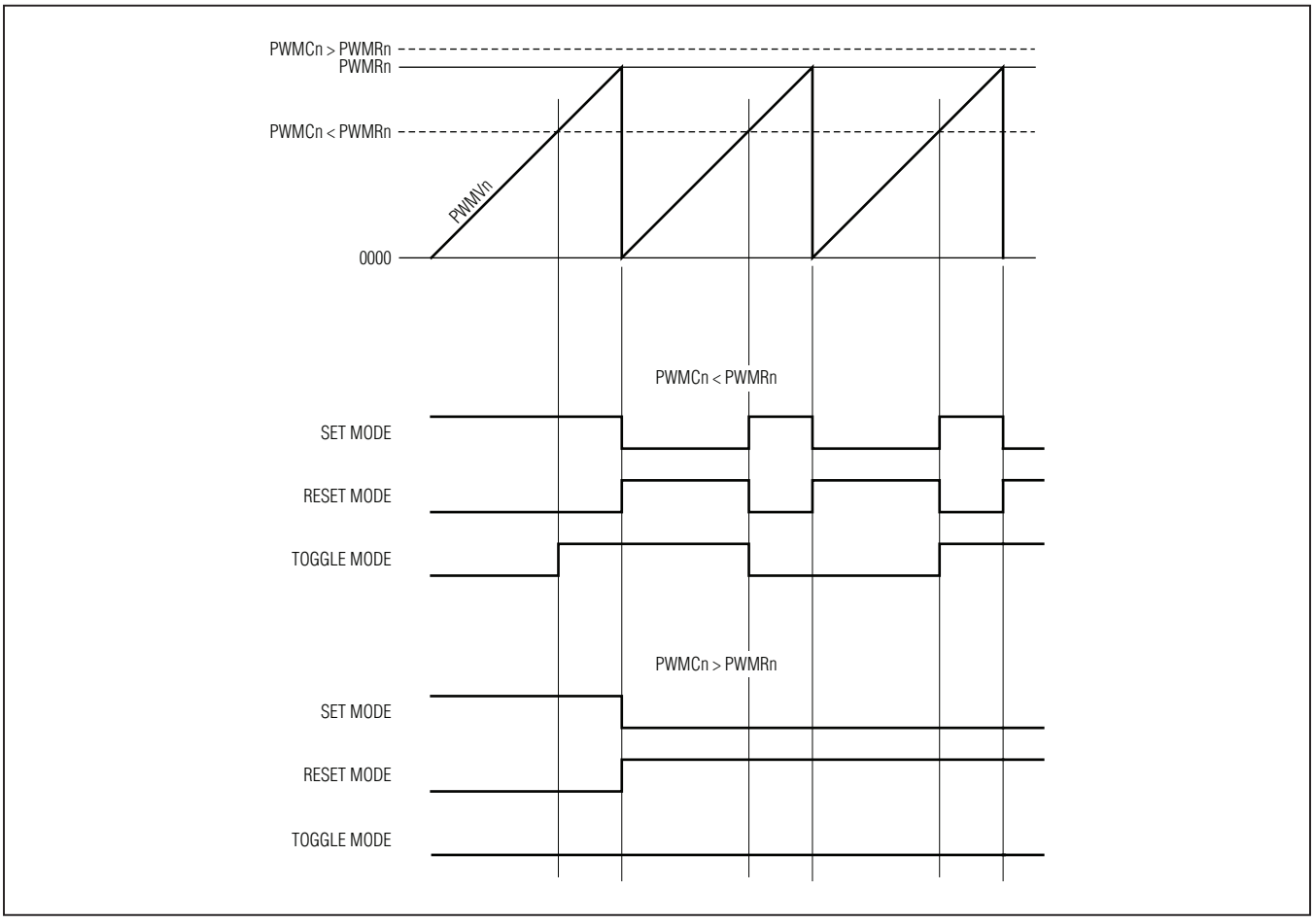


Figure 9-2. PWM Output Waveform in Normal PWM Output Mode

9.1.4 Up/Down Count PWM Output Operation

The PWM can also operate in an up/down count configuration by setting DCEN = 1. The value in PWMVn counts upward until it reaches the value in the reload register (PWMRn). On the next cycle the count reverses direction and starts counting down. When PWMVn reaches 0000h, the count again reverses direction and begins counting up.

When operating in an up/down count configuration and either set or reset mode, the PWM effectively allows 17-bit resolution. In set mode the duty cycle is always less than 50%, and in reset mode the duty cycle is always greater than 50%. The toggle mode provides a center-aligned 16-bit PWM with twice the period of the normal PWM output mode. Figure 9-3 illustrates the PWM waveforms when operating in up/down count PWM output mode. The up/down count PWM output period and duty cycle are calculated as follows:

$$\text{Period} = 2 \times \text{PWMRn} \times \text{PWM.n CLOCK PERIOD}$$

$$\text{Duty Cycle in Set Mode} = \frac{\text{PWMRn} + \text{PWMCn}}{2 \times \text{PWMRn}}$$

$$\text{Duty Cycle in Reset Mode} = \frac{\text{PWMCn}}{2 \times \text{PWMRn}}$$

$$\text{Duty Cycle in Toggle Mode} = \frac{\text{PWMRn} - \text{PWMCn}}{\text{PWMRn}}$$

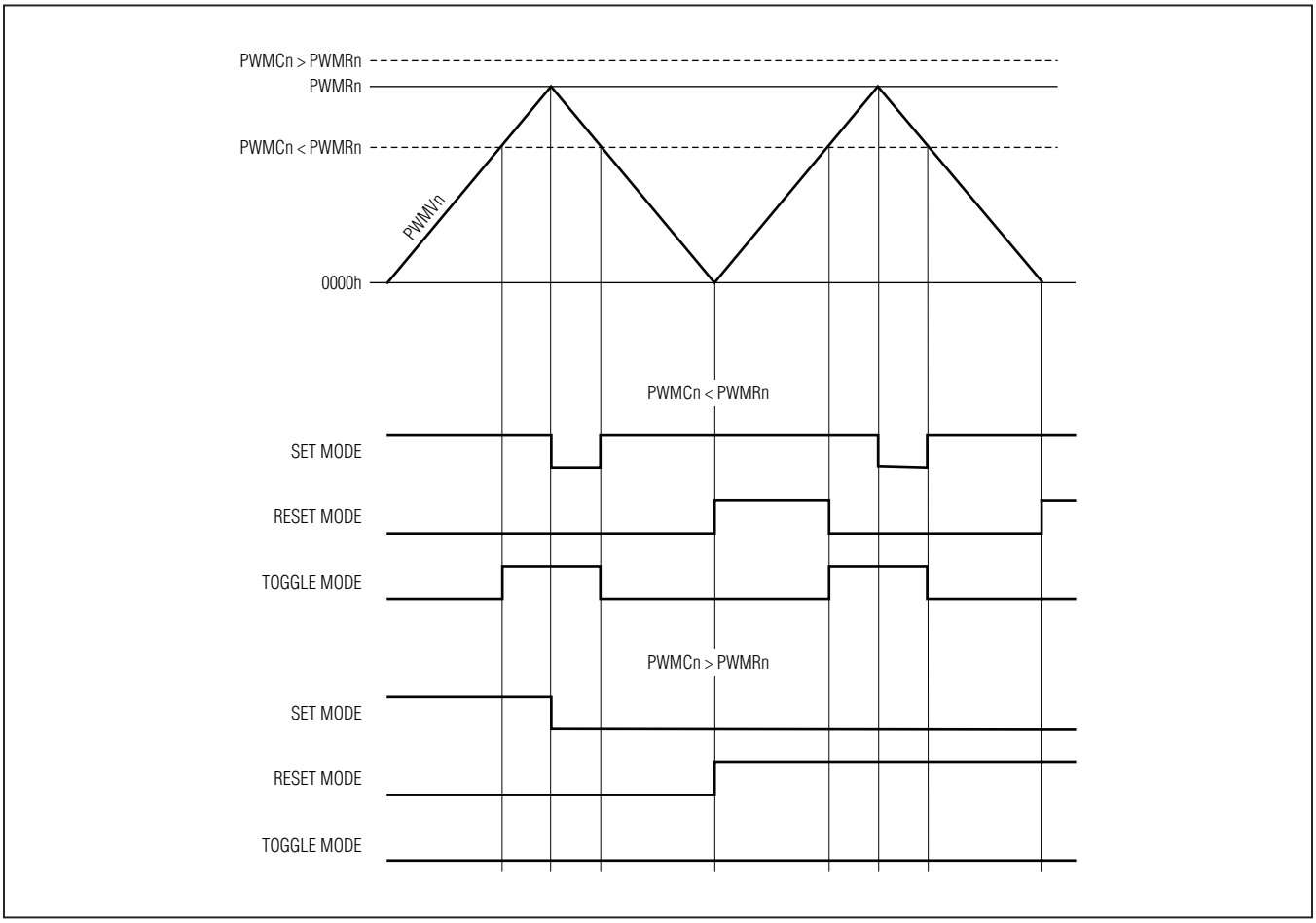


Figure 9-3. PWM Waveform in Up/Down Count PWM Output Mode

9.2 PWM Output Register Descriptions

The following peripheral registers are used to control the PWM outputs of the MAX31782. Each of the six independent PWM outputs has four associated registers. Since there are six independent PWM outputs, the registers are described in a batch manner. For example, the control register is denoted as PWMCNn, where n = 0 to 5. Each PWM register is independent, meaning each PWM can be configured and operated differently.

9.2.1 PWM Control Register (PWMCNn)

The PWM control register, PWMCNn, is used to set up and start the PWM output. To avoid undesired operation, the user should **not** modify the reserved bits in the PWMCNn registers.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	PWMCS	PWMCR	PWMPS2	PWMPS1	PWMPS0	TFB	—	—	DCEN	—	PWMEN	ETB	—
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	rw	rw	rw	rw	rw	rw	r	r	rw	r	rw	rw	r

BIT	NAME	DESCRIPTION	
15:13	—	Reserved. The user should not write to these bits.	
12:11	PWMCS, PWMCR	PWM Pin Output Set/Reset Mode Bits. These mode bits define if the PWM output function is enabled on the PWM.n pin, the initial output starting state when the PWM is disabled (PWMEN = 0), and what compare mode output function is used.	
10:8	PWMPS[2:0]	PWM Clock Prescaler Bits. These bits select the clock prescaler applied to the system clock, which is then used as the PWM clock. The PWMPS[2:0] bits should be configured by the user when the timer is stopped (PWMEN = 0). While hardware does not prevent changing the PWMPS[2:0] bits when the PWM is running, the resulting behavior is nondeterministic.	
		PWMPS[2:0]	**PWM INPUT CLOCK**
		000	Sysclk
		001	Sysclk/4
		010	Sysclk/16
		011	Sysclk/64
		100	Sysclk/256
		101	Sysclk/1024
		11x	Sysclk
7	TFB	PWM Overflow Flag. This bit is set when the PWM overflows or reaches PWMRn and is reloaded to 0000h. The TFB flag is also set when PWMVn is equal to 0000h in when counting down. The setting of this flag causes an interrupt if enabled. This flag must be cleared by software.	
6:5	—	Reserved. The user should not write to these bits.	
4	DCEN	Down-Count Enable. The DCEN bit controls if the PWM operates in normal PWM mode and counts up only (DCEN = 0), or operates in up/down count mode and counts up and down (DCEN = 1).	
3	—	Reserved. The user should not write to this bit.	
2	PWMEN	PWM Run Control. This bit enables PWM operation when set to 1. Clearing this bit to 0 halts the PWM operation and preserves the current count in PWMVn.	
1	ETB	PWM Interrupt. Setting this bit to 1 enables interrupts from the TFB flag.	
0	—	Reserved. The user should not write to this bit.	

9.2.2 PWM Value Register (PWMVn)

The PWM value register, PWMVn, holds the 16-bit value of the PWM's counter. Enabling or disabling the PWM with the PWMEN bit does not reset the PWMVn register. The PWMVn register must be cleared by software. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

9.2.3 PWM Reload Register (PWMRn)

The PWM reload register, PWMRn, is a 16-bit register that is used as a comparison to the PWMVn register. A reload of the PWMVn register occurs when PWMVn matches PWMRn. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

9.2.4 PWM Compare Register (PWMCn)

The PWM compare register, PWMCn, is a 16-bit register that is used as a comparison to the PWMVn register. Depending upon the mode of PWM operation, the PWM.n pin is driven high or low when a match between PWMVn and PWMCn occurs. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

9.2.5 PWM Register Locations

The addresses for the PWM output registers are given as "Mx[yy]," where x is the module number (from 0 to 5 decimal) and yy is the register index (from 00h to 1Fh hexadecimal). Table 9-3 shows the addresses of these registers for each of the six PWM outputs (PWM.n).

Table 9-3. PWM Register Addresses

REGISTER NAME	INDIVIDUAL PWM OUTPUT NUMBER					
	n = 0	n = 1	n = 2	n = 3	n = 4	n = 5
PWMCNn	M3[09h]	M3[0Bh]	M4[09h]	M4[0Bh]	M5[09h]	M5[17h]
PWMVn	M3[08h]	M3[0Ah]	M4[08h]	M4[0Ah]	M5[08h]	M5[16h]
PWMRn	M3[01h]	M3[03h]	M4[01h]	M4[03h]	M5[0Bh]	M5[15h]
PWMCn	M3[00h]	M3[02h]	M4[00h]	M4[02h]	M5[0Ah]	M5[14h]

9.3 PWM Output Code Example

Creating a 40% duty cycle 25kHz signal:

```
PWMCN0_bit.PWMPS = 0;        //PWM.0 input clk = sysclk
PWMR0 = 159;                 //PWM period = 160 sysclks
PWMC0 = 64;                  //duty cycle = 64/160
PWMCN0_bit.PWMCR = 1;        //set to reset mode
PWMCN0_bit.PWMCS = 0;        //set to reset mode
PWMCN0_bit.PWMEN = 1;        //enable PWM.0
```

SECTION 10: FAN TACHOMETER

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 10: FAN TACHOMETER

The MAX31782 provides six independent fan tachometers that can be used to monitor the speed of six fans independently. When the fan tachometer functionality of a pin is disabled, that pin can be used as a general-purpose input/output (GPIO). Figure 10-1 shows a diagram for one individual fan tachometer block.

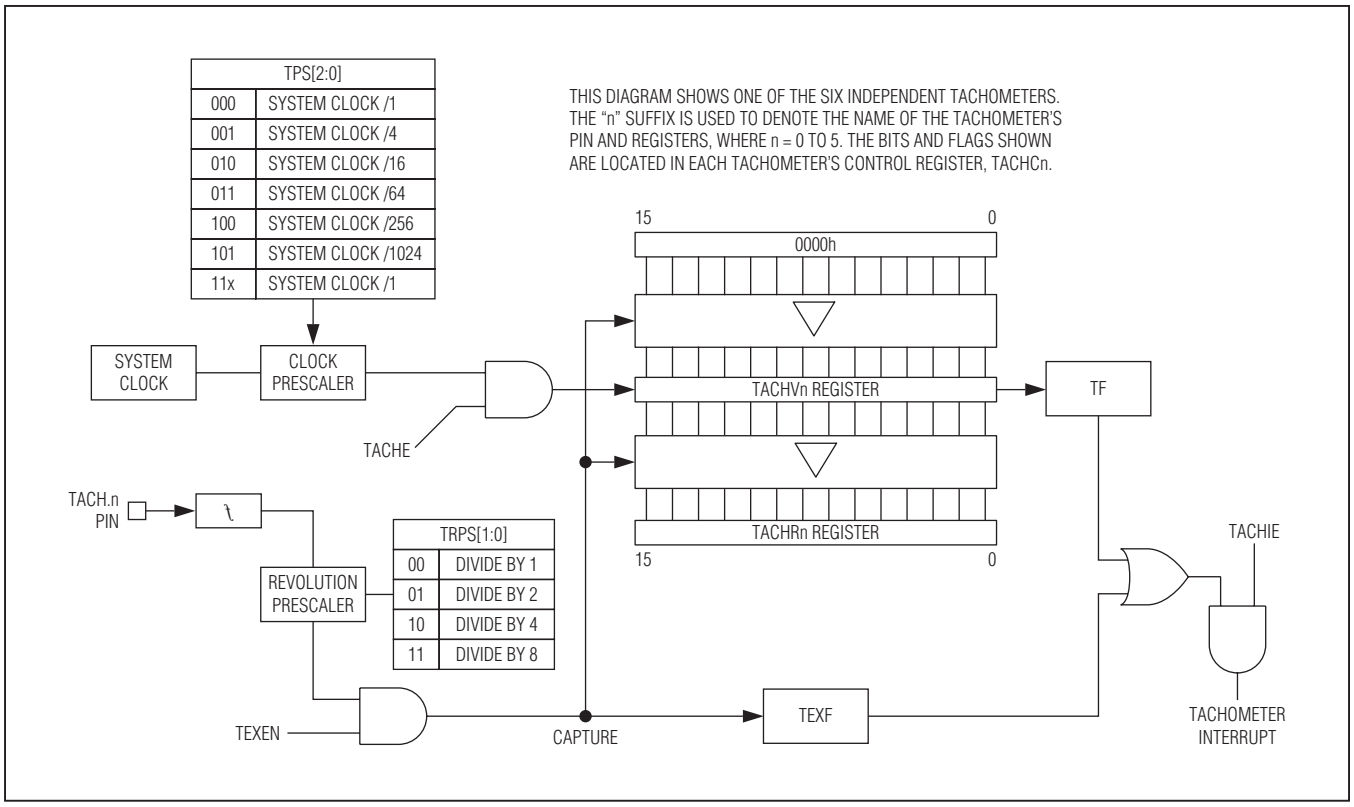


Figure 10-1. Tachometer Input Block Diagram

10.1 Fan Tachometer Detailed Description

When a tachometer is initially enabled (TACHE = 1), it begins counting up from the TACHV value. The frequency of the counter is derived from the MAX31782's 4MHz system clock (f_{MOSC}). The tachometer can use a divided version of the system clock by using the timer prescaler (TPS[2:0] bits). When the TACHV count value reaches FFFFh, the counter rolls over to 0000h and continues counting. When an overflow occurs, the TF flag is set, which can generate an interrupt if enabled.

The tachometer feature works by capturing the number of system clocks, or divided system clocks, that occur during one revolution of a fan. The tachometer block is triggered on the falling edge of the tachometer pin. Many fans output multiple pulses per revolution. The tachometer block contains a revolution prescaler to compensate for fans that do output multiple pulses per revolution. The revolution prescaler (TRPS[1:0] bits) can be programmed to work with fans that output 1, 2, 4, or 8 pulses per revolution. When the number of falling edges received at the tachometer pin matches the pulses per revolution defined by the revolution prescaler, tachometer block captures the number of system clock counts for the fan revolution. When a capture is triggered the following occurs:

1) The value in the TACHV counter register is copied to the capture register (TACHR).
2) The TACHV register is reset to 0000h and continues counting.
3) The TEXF flag is set. This causes an interrupt if enabled.

Note that the TEXF flag can be set (and causes an interrupt if enabled) even if the tachometer is not enabled (TACHE = 0). If the TEXEN bit is set to logic 0, falling edges on the tachometer pin do not trigger a capture event.

Following is an example of how to calculate the fan speed after the tachometer has captured one revolution. This example assumes that the clock prescaler is set to be divide by 16 (010h) and the value read from the TACHR register is 1000h. The tachometer clock is calculated to be:

Tachometer Clock = Sysclk/16 = 4MHz/16 = 250kHz

The frequency of the fan revolution can then be calculated as:

Fan Frequency = Tachometer Clock/TACHR = 250kHz/1000h = 61Hz, which equals 3660 RPM

10.2 Timer/Fan Tachometer Register Descriptions

The following peripheral registers are used to control the fan tachometer of the MAX31782. Each of the six independent tachometers has three associated registers. Because there are six independent tachometers, the registers are described in a batch manner. For example, the control register is denoted as TACHCNn, where n = 0 to 5. Each tachometer's registers are independent, meaning each tachometer can be configured and operated differently.

10.2.1 Tachometer Control Register (TACHCNn)

The tachometer control register, TACHCNn, is used to set up and start the tachometer, and is also where tachometer interrupt flags are located. It should be noted that the user should **not** modify the reserved bits in the TACHCNn registers. Otherwise, undesired operation can occur.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	TRPS.1	TRPS.0	—	—	TPS.2	TPS.1	TPS.0	TF	TEXF	—	—	TEXEN	TACHE	TACHIE	—
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Access	r	rw	rw	r	r	rw	rw	rw	rw	rw	r	r	rw	rw	rw	r

BIT	NAME	DESCRIPTION	
15	—	Reserved. The user should not write to this bit.	
14:13	TRPS[1:0]	Revolution Prescaler. These bits are used to set the number of tachometer pin falling edges are required to trigger a capture. This allows the tachometer to easily work with fans that produce multiple pulses per revolution.	
		TRPS[1:0]	**PRESCALER**
		00	1 pulse per revolution
		01	2 pulses per revolution
		10	4 pulses per revolution
		11	8 pulses per revolution
12:11	—	Reserved. The user should not write to these bits.	
10:9	TPS[2:0]	Clock Prescale. These bits select the frequency of the clock input to the tachometer. The tachometer clock is a divided version of the system clock. The TPS[2:0] bits should be configured by the user when the timer is stopped (TACHE = 0). While hardware does not prevent changing the TPS[2:0] bits when the timer is running, the resultant behavior is nondeterministic.	
		TPS[2:0]	**TACHOMETER INPUT CLOCK**
		000	Sysclk/1
		001	Sysclk/4
		010	Sysclk/16
		011	Sysclk/64
		100	Sysclk/256
		101	Sysclk/1024
		11x	Sysclk/1
7	TF	Overflow Flag. This bit is set when the tachometer's TACHV register overflows from FFFFh to 0000h. An interrupt will be generated if TACHIE=1. This flag must be cleared by software.	
6	TEXF	External Tachometer Trigger Flag. A falling edge on the tachometer's pin (TACH.n) causes this flag to be set if enabled (TEXEN = 1). The TEXF flag is only set once the tachometer revolution prescaler condition is met. This flag must be cleared by software. Setting this bit to 1 forces a tachometer interrupt if enabled. **Note 1:** The revolution prescaler always triggers on the first tachometer pulse received, then, depending on division factor, it triggers again after 1, 2, 4, or 8 tachometer pulses. **Note 2:** This flag is set on a falling edge of the tachometer pin even if the tachometer is disabled (TACHE = 0).	
5:4	—	Reserved. The user should not write to these bits.	
3	TEXEN	External Enable. Setting this bit to 1 enables the capture function on a falling edge of the tachometer pin (TACH.n).	
2	TACHE	Run Control. This bit enables the tachometer operation when set to 1. Clearing this bit to 0 halts the tachometer operation and preserves the current count in TACHV.	
1	TACHIE	Enable Tachometer Interrupt. Setting this bit to 1 enables the interrupt from the TF and TEXF flags.	
0	—	Reserved. The user should not write to this bit.	

10.2.2 Tachometer Value Register (TACHVn)

The tachometer value register, TACHVn, holds the 16-bit value of the tachometer's up-counting timer. Enabling/disabling the tachometer with the TACHE bit does not reset this count value; it must be cleared explicitly by software. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

10.2.3 Tachometer Capture Register (TACHRn)

The tachometer capture register, TACHRn, stores the 16-bit value in TACHVn when a tachometer capture occurs. The TACHRn value indicates how many prescaled system clock pulses occurred during one revolution of the fan, assuming the revolution prescaler is set correctly. The value in TACHRn is typically used to calculate fan speed in fan control applications. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

10.2.3 Tachometer Register Locations

The addresses for the tachometer registers are given as "Mx[yy]," where x is the module number (from 0 to 5 decimal) and yy is the register index (from 00h to 1Fh hexadecimal). Table 10-1 shows the address for these registers for each of the six tachometer blocks (TACH.n).

Table 10-1. Tachometer Register Addresses

REGISTER NAME	INDIVIDUAL TACHOMETER NUMBER					
	n = 0	n = 1	n = 2	n = 3	n = 4	n = 5
TACHCNn	M3[0Dh]	M3[0Fh]	M4[0Dh]	M4[0Fh]	M5[0Dh]	M5[13h]
TACHRn	M3[05h]	M3[07h]	M4[05h]	M4[07h]	M5[0Fh]	M5[11h]
TACHVn	M3[0Ch]	M3[0Eh]	M4[0Ch]	M4[0Eh]	M5[0Ch]	M5[12h]

10.3 Tachometer Pin and GPIO Multiplexing

When the tachometer's pin functionality is disabled (TEXEN = 0), that pin can be used as a GPIO. The tachometer pins are mapped to GPIO port P2[5:0]. Table 10-2 shows the mapping of the MAX31782 tachometer pins. Refer to *SECTION 11: General-Purpose Input/Output (GPIO) Pins* for information on using the tachometer pins as GPIO.

Table 10-2. Tachometer/GPIO Pin Multiplexing Input Pins

TACHOMETER INPUT PIN	MAX31782 PIN	GPIO PIN
TACH.0	30	P2.0
TACH.1	27	P2.1
TACH.2	25	P2.2
TACH.3	23	P2.3
TACH.4	19	P2.4
TACH.5	17	P2.5

10.4 Tachometer Code Example

The following pseudocode shows how to set up tachometer 0. This example does not generate any interrupts, but instead the captured tachometer value can be periodically polled by software.

Tachometer setup:

```
TACHCN0_bit.TPS = 3;        //tachometer clock is sysclk / 64 or 62.5kHz
TACHCN0_bit.TRPS = 1;       //set for 2 pulses per revolution
TACHCN0_bit.TEXEN = 1;      //enable edge capture of TACH.0 pin
TACHCN0_bit.TACHE = 1;      //start the tachometer count
```

Reading the tachometer:

```
tach_counts = TACHR0;       //store the captured tachometer counts for the last
                            revolution in a variable
```

SECTION 11: GENERAL-PURPOSE INPUT/OUTPUT (GPIO) PINS

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 11: GENERAL-PURPOSE INPUT/OUTPUT (GPIO) PINS

The MAX31782 provides general-purpose input/output (GPIO) functionality on 21 pins. In addition to the GPIO functionality, each of these pins is multiplexed with at least one other function, which is classified as either a special function or alternate function.

Special functions override the GPIO register settings of the port pin when they are enabled. Once the special function takes control, normal control of the port pin is lost until the special function is disabled.

Alternate functions operate in parallel with the GPIO register settings for the port pin, and generally consist of input-only functions. When an alternate function is enabled for a port pin, the port pin's output state can still be controlled by the GPIO register settings, or driven by external hardware.

Table 11-1 details all the GPIO pins as well as what other functions are multiplexed with each pin. With the exception of a few pins, which are described in further detail later, the GPIO pins operate as shown in the GPIO block diagram (Figure 11-1). Some of the features of these GPIO pins include the following:

- CMOS output drivers
- Schmitt trigger inputs
- Optional weak pullup to V_{DD} when operating in input mode

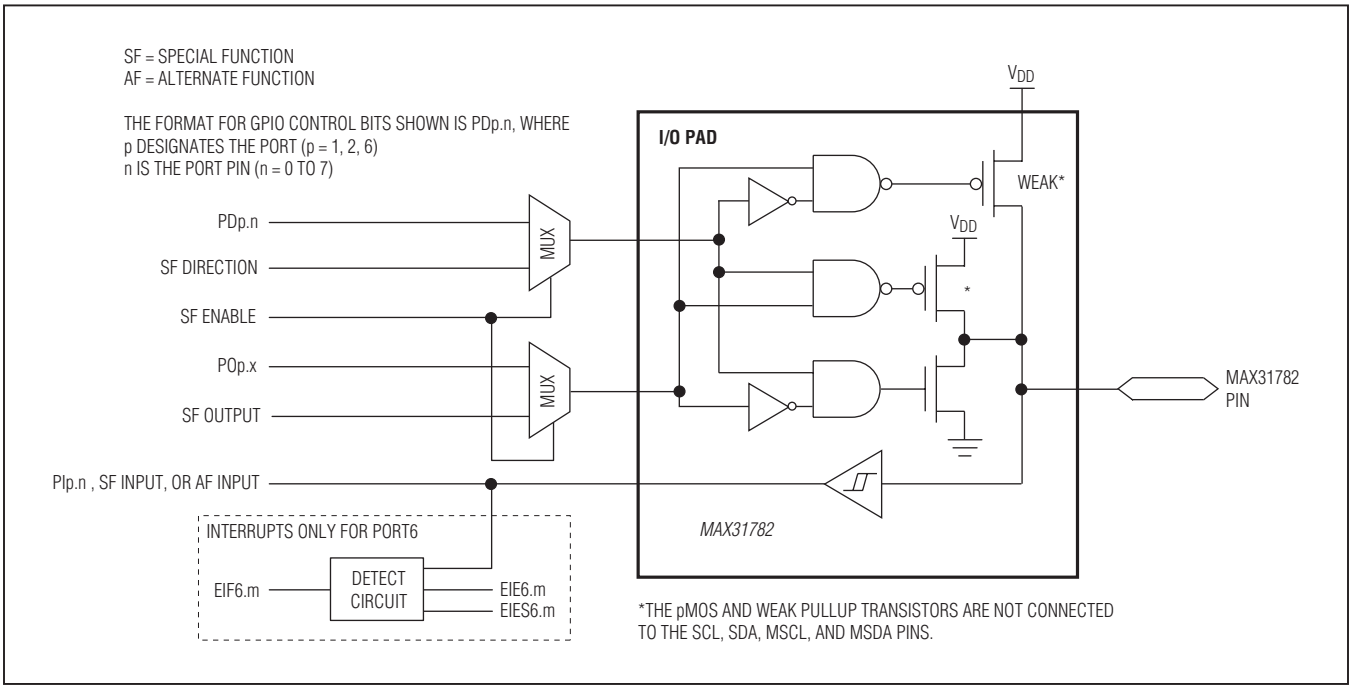


Figure 11-1. GPIO Pin Block Diagram

Table 11-1. GPIO Pins and Multiplexed Functions

PIN	NAME	PORT INDEX	ALTERNATE FUNCTION(S)	ALTERNATE FUNCTION ENABLE	SPECIAL FUNCTION	SPECIAL FUNCTION ENABLE	RESET STATE
28	PWM.0	P1.0	—	—	PWM.0	PWMCN0.PWMCR or PWMCS = 1	GPIO
26	PWM.1	P1.1	—	—	PWM.1	PWMCN1.PWMCR or PWMCS = 1	GPIO
24	PWM.2	P1.2	—	—	PWM.2	PWMCN2.PWMCR or PWMCS = 1	GPIO
20	PWM.3	P1.3	—	—	PWM.3	PWMCN3.PWMCR or PWMCS = 1	GPIO
18	PWM.4	P1.4	—	—	PWM.4	PWMCN4.PWMCR or PWMCS = 1	GPIO
16	PWM.5	P1.5	—	—	PWM.5	PWMCN5.PWMCR or PWMCS = 1	GPIO
30	TACH.0	P2.0	TACH.0	TACHCN0.TEXEN = 1	—	—	GPIO
27	TACH.1	P2.1	TACH.1	TACHCN1.TEXEN = 1	—	—	GPIO
25	TACH.2	P2.2	TACH.2	TACHCN2.TEXEN = 1	—	—	GPIO
23	TACH.3	P2.3	TACH.3	TACHCN3.TEXEN = 1	—	—	GPIO
19	TACH.4	P2.4	TACH.4	TACHCN4.TEXEN = 1	—	—	GPIO
17	TACH.5	P2.5	TACH.5	TACHCN5.TEXEN = 1	—	—	GPIO
15	MSCL	P2.6	—	—	MSCL	I2CCN_M.I2CEN = 1	GPIO
14	MSDA	P2.7	—	—	MSDA	I2CCN_M.I2CEN = 1	GPIO
38	P6.0/TCK	P6.0	TCK	SC.TAP = 1	—	—	TCK
37	P6.1/TDI	P6.1	TDI	SC.TAP = 1	—	—	TDI
35	P6.2/TMS/ TBB	P6.2	TMS, TBB Input	SC.TAP = 1, TB0CN.EXENB = 1	TBB Output	TB0CN.TBCR or TBCS = 1	TMS
34	P6.3/TDO	P6.3	—	—	TDO	SC.TAP = 1	TDO
33	P6.4/TBA	P6.4	TBA Input	TB0CN.CnTB = 1	TBA Output	TB0CN.CnTB = 0 and TBCN.TBOE = 1	GPIO
32	SCL	P6.6	—	—	SCL	I2CCN_S.I2CEN = 1	SCL
31	SDA	P6.7	—	—	SDA	I2CCN_S.I2CEN = 1	SDA

TCK: Test Access Port (TAP) Clock
TDI: Test Access Port (TAP) Data Input
TMS: Test Access Port (TAP) Mode Select
TDO: Test Access Port (TAP) Data Output
TBB: Timer/Counter B Input/Output B
TBA: Timer/Counter B Input/Output A

From a software perspective, each of the GPIO ports (port 1, port 2, and port 6) has three special-function registers (POp, PIp, and PDp, where p = 1, 2, or 6). Port 6 has three additional registers that allow for GPIO interrupts from the port. Each GPIO port is designed to provide programming flexibility for any application. Table 11-2 lists the associated registers and their module addresses. The user should not write to any reserved bits as this can cause undesired behavior.

Table 11-2. GPIO Registers

REGISTER	FUNCTION	PORT 1	PORT 2	PORT 6
POp	Port Output Register	M0[1h]	M0[0h]	M1[03h]
PIp	Port Input Register	M0[9h]	M0[8h]	M1[08h]
PDp	Port Direction Register	M0[11h]	M0[10h]	M1[12h]
EIF6	Port 6 External Interrupt Flag Register	—	—	M1[06h]
EIE6	Port 6 External Interrupt Enable Register	—	—	M1[07h]
EIES6	Port 6 External Interrupt Edge Select Register	—	—	M1[10h]

11.1 GPIO Port 1 Register Descriptions

Port 1 provides six GPIO pins that are multiplexed with PWM functionality. The PWM function is enabled when either the PWMCNn.PWMCR or PWMCS bits are a 1, where n = 0 to 5. If both of these bits are a 0, the pin operates as a GPIO. The port 1 pins provide all the functionality shown in the GPIO block diagram (Figure 11-1). This port does not provide GPIO interrupts.

11.1.1 GPIO Direction Register Port 1 (PD1)

Bit	7	6	5	4	3	2	1	0
Name	—	—	PD1_5	PD1_4	PD1_3	PD1_2	PD1_1	PD1_0
Reset	0	0	0	0	0	0	0	0
Access	r	r	rw	rw	rw	rw	rw	rw

PD1 is an 8-bit register used to determine the direction of the pins when they are used as GPIO pins. Each pin is independently controlled by its direction bit. When PD1.n (n = 0 to 5) is set to 1, the pin is an output; data in the PO1.n bit is driven on the pin. When PD1.n is cleared to 0, the pin is an input, and allows an external signal to drive the pin. Note that each port pin has a weak pullup circuit when functioning as an input. The p-channel pullup transistor is controlled by the PO1.n bit. If PO1.n is set to 1, the corresponding weak pullup is turned on; if the PO1.n bit is cleared to 0, the weak pullup is turned off and the pin's input is high impedance. When the port 1 pins are operating as PWM pins, the data in PD1 does not affect PWM operation.

11.1.2 GPIO Output Register Port 1 (PO1)

Bit	7	6	5	4	3	2	1	0
Name	—	—	PO1_5	PO1_4	PO1_3	PO1_2	PO1_1	PO1_0
Reset	1	1	1	1	1	1	1	1
Access	r	r	rw	rw	rw	rw	rw	rw

PO1 is an 8-bit register that controls the output data of a GPIO pin. If the pin is setup to be an output (PD1.n = 1), the data in PO1.n is output on the pin. If the pin is set as an input (PD1.n = 0), setting PO1.n to a 1 enables a p-channel weak pullup, otherwise the pin's input is high impedance. When the port 1 pins are operating as PWM pins, the data in PO1 does not affect PWM operation. Changing the direction of the pin does not change the data content of PO1.n.

11.1.3 GPIO Input Register for Port 1 (PI1)

Bit	7	6	5	4	3	2	1	0
Name	—	—	PI1_5	PI1_4	PI1_3	PI1_2	PI1_1	PI1_0
Reset	1	1	s	s	s	s	s	s
Access	r	r	r	r	r	r	r	r

PI1 is an 8-bit register that contains the data that is applied to the GPIO pins. The PI1 input register contains valid input data even when the pin is not operating as a GPIO. The reset value for this register is dependent on the logical states applied to the pins. Note that each pin has a weak pullup circuit when functioning as an input and the p-channel pullup transistor is controlled by the PO1.n bit.

11.2 GPIO Port 2 Register Descriptions

Port 2 provides eight GPIO pins that are multiplexed with the tachometers and master I2C port. This port does not provide GPIO interrupts.

The tachometer function is an alternate function. This means that the GPIO functions are fully supported, even when the pin is operating as a tachometer. If the tachometer is enabled while the pin is being operated as an output GPIO pin, a high-to-low output transition is monitored by the tachometer and can cause a tachometer interrupt. The tachometer functionality is disabled by setting the TACHCNn.TEXEN bit to a 0, where n = 0 to 5.

GPIO pins P2.6 and P2.7 are multiplexed with the master I2C port. The master I2C port is a special function and disables GPIO output when enabled (I2CCN_M.I2CEN = 1). These two pins are open-drain output pins and do not have the p-channel drive transistor or weak internal pullup. An external pullup resistor is required to achieve a high-logic level.

11.2.1 GPIO Direction Register Port 2 (PD2)

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Name	PD2_7	PD2_6	PD2_5	PD2_4	PD2_3	PD2_2	PD2_1	PD2_0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw

PD2 is an 8-bit register used to determine the direction of the pins when they are used as GPIO pins. Each pin is independently controlled by its direction bit. When PD2.n (n = 0 to 7) is set to 1, the pin is an output; data in the PO2.n bit is driven on the pin. When PD2.n is cleared to 0, the pin is an input, and allows an external signal to drive the pin. Note that each port pin has a weak pullup circuit when functioning as an input. The p-channel pullup transistor is controlled by the PO2.n bit. If PO2.n is set to 1, the corresponding weak pullup is turned on; if the PO2.n bit is cleared to 0, the weak pullup is turned off and the pin's input is high impedance. The weak pullup transistor is not available on pins P2.6 and P2.7.

11.2.2 GPIO Output Register Port 2 (PO2)

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Name	PO2_7	PO2_6	PO2_5	PO2_4	PO2_3	PO2_2	PO2_1	PO2_0
Reset	1	1	1	1	1	1	1	1
Access	rw	rw	rw	rw	rw	rw	rw	rw

PO2 is an 8-bit register that controls the output data of a GPIO pin. If the pin is setup to be an output (PD2.n = 1), the data in PO2.n is output on the pin. If the pin is set as an input (PD2.n = 0), setting PO2.n to a 1 enables a p-channel weak pullup, otherwise the pin's input is high impedance. If the P2.6 and P2.7 pins (master I2C port) are driven as an output, they operate as open-drain outputs. An external pullup resistor is required to achieve a high-logic level.

11.2.3 GPIO Input Register for Port 2 (PI2)

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Name	PI2_7	PI2_6	PI2_5	PI2_4	PI2_3	PI2_2	PI2_1	PI2_0
Reset	s	s	s	s	s	s	s	s
Access	r	r	r	r	r	r	r	r

PI2 is an 8-bit register that contains the data that is applied to the GPIO pins. The PI2 input register contains valid input data even when the pin is not operating as a GPIO. The reset value for this register is dependent on the logical states applied to the pins. Note that each pin, except P2.6 and P2.7, has a weak pullup circuit when functioning as an input, and the p-channel pullup transistor is controlled by the PO2.n bit.

11.3 GPIO Port 6 Register Descriptions

Port 6 provides seven GPIO pins that are multiplexed with the test access port (TAP), Timer B, and slave I2C port. See Table 11-1 for more details about the multiplexed functions and how to enable or disable these functions.

Note that SCL and SDA pins can be configured as GPIOs (P6.6 and P6.7, respectively) with open drain if needed, although this is not the typical application. In this case, bits 6 and 7 in the port 6 SFRs control the GPIO functions of the SCL and SDA pins, respectively. SCL and SDA are open-drain outputs and do not have the p-channel drive transistor or weak internal pullup. External pullups are required to realize a logic-high. The user should also be aware that once SCL and SDA are converted to GPIO, they can no longer perform I2C communications. The host cannot talk to the device through the I2C-compatible slave interface or use the I2C bootloader. See the *SECTION 7: I2C-Compatible Slave Interface* for more information.

On device reset, the TAP port is active, allowing for in-circuit debugging and programming. The TAP TDO pin (P6.3) is a logic-high output following a device reset. Extra precautions must be taken to ensure that this pin does not cause any undesirable operations following a reset.

Port 6 also provides GPIO interrupts on all the pins. A GPIO interrupt can be generated when the pin is being operated as a GPIO, or a special or alternate function. Three additional registers—EIF6, EIE6, and EIES6—are used to control the GPIO interrupts.

11.3.1 GPIO Direction Register Port 6 (PD6)

Bit	7	6	5	4	3	2	1	0
Name	PD6_7	PD6_6	—	PD6_4	PD6_3	PD6_2	PD6_1	PD6_0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	r	rw	rw	rw	rw	rw

PD6 is an 8-bit register used to determine the direction of the pins when they are used as GPIO pins. Each pin is independently controlled by its direction bit. When PD6.n (n = 0 to 7 excluding 5) is set to 1, the pin is an output; data in the PO6.n bit is driven on the pin. When PD6.n is cleared to 0, the pin is an input, and allows an external signal to drive the pin. Note that each port pin except P6.6 and P6.7 has a weak pullup circuit when functioning as an input. The p-channel pullup transistor is controlled by the PO6.n bit. If PO6.n is set to 1, the corresponding weak pullup is turned on; if the PO6.n bit is cleared to 0, the weak pullup is turned off and the pin's input is high impedance. The weak pullup transistor is not available on pins P6.6 and P6.7.

11.3.2 GPIO Output Register Port 6 (PO6)

Bit	7	6	5	4	3	2	1	0
Name	PO6_7	PO6_6	—	PO6_4	PO6_3	PO6_2	PO6_1	PO6_0
Reset	1	1	1	1	1	1	1	1
Access	rw	rw	r	rw	rw	rw	rw	rw

PO6 is an 8-bit register that controls the output data of a GPIO pin. If the pin is set up to be an output (PD6.n = 1), the data in PO6.n is output on the pin. If the pin is set as an input (PD6.n = 0), setting PO6.n to a 1 enables a p-channel weak pullup; otherwise, the pin's input is high impedance. If the P6.6 and P6.7 pins (slave I2C port) are driven as an output, they operate as open-drain outputs. An external pullup resistor is required to achieve a high-logic level.

11.3.3 GPIO Input Register for Port 6 (PI6)

Bit	7	6	5	4	3	2	1	0
Name	PI6_7	PI6_6	—	PI6_4	PI6_3	PI6_2	PI6_1	PI6_0
Reset	s	s	1	s	s	s	s	s
Access	r	r	r	r	r	r	r	r

PI6 is an 8-bit register that contains the data that is applied to the GPIO pins. The PI6 input register contains valid input data even when the pin is not operating as a GPIO. The reset value for this register is dependent on the logical states applied to the pins. Note that each pin has a weak pullup circuit when functioning as an input and the p-channel pullup transistor is controlled by the PO6.n bit.

11.3.4 GPIO Port 6 External Interrupt Edge Select Register (EIES6)

Bit	7	6	5	4	3	2	1	0
Name	IESP6_7	IESP6_6	—	IESP6_4	IESP6_3	IESP6_2	IESP6_1	IESP6_0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	r	rw	rw	rw	rw	rw

The EIES6 register sets the interrupt edge select to trigger an interrupt on either a rising or falling edge. Setting the IESP6_n bits to 0 triggers the corresponding interrupt on a positive edge. When these bits are set to 1, the interrupt is on a negative edge.

11.3.5 GPIO Port 6 External Interrupt Flag Register (EIF6)

Bit	7	6	5	4	3	2	1	0
Name	IFP6_7	IFP6_6	—	IFP6_4	IFP6_3	IFP6_2	IFP6_1	IFP6_0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	r	rw	rw	rw	rw	rw

These bits are set when a negative edge (IESP6.n = 1) or a positive edge (IESP6.n = 0) is detected on the P6.n pin. Setting any of the bits to 1 generates an interrupt to the CPU if the corresponding interrupt is enabled. These bits remain set until cleared by software or a reset. These bits must be cleared by software before exiting the interrupt service routine or another interrupt is generated as long as the bit remains set.

11.3.6 GPIO Port 6 External Interrupt Enable Register (EIE6)

Bit	7	6	5	4	3	2	1	0
Name	IEP6_7	IEP6_6	—	IEP6_4	IEP6_3	IEP6_2	IEP6_1	IEP6_0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	r	rw	rw	rw	rw	rw

Setting any of these bits to 1 enables the corresponding external interrupt. Clearing any of the bits to 0 disables the corresponding interrupt function.

11.4 GPIO Code Example

```
//set pin 6.4 as a high output
PD6 |= 0x10;       //set direction PD6.4 to 1 for an output
PO6 |= 0x10;       //set the output PO6.4 high

//set pin 6.4 as a high-impedance input
PD6 &= ~0x10;      //set direction PD6.4 to 0 for input
PO6 &= ~0x10;      //set PO6.4 low to disable weak pullup

//enable the pin 6.4 weak pullup
PD6 &= ~0x10;      //set direction PD6.4 to 0 for input
PO6 |= 0x10;       //set PO6.4 high to enable weak pullup
```

SECTION 12: TIMER B MODULE

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 12: TIMER B MODULE

The MAX31782 provides one Timer B module that can be configured to provide different timer, counter, clock, or PWM functions. The Timer B uses the TBB and TBA pins, which are also used for JTAG and GPIO operation. Table 12-1 details these pins.

Table 12-1. Timer B Pins

TIMER B PIN	MAX31782 PIN NUMBER	GPIO PIN	JTAG PIN
TBA	33	P6.4	—
TBB	35	P6.2	TMS

12.1 Detailed Description

The Timer B is a 16-bit programmable module that supports input clock prescaling and set/reset/toggle PWM output control functionality. Another distinguishing characteristic of Timer B is that its count ranges from 0000h to the value stored in the 16-bit capture/reload register (TBR) instead of FFFFh as in some timers.

The possible Timer B operating modes and related control bits are shown in Table 12-2. A complete description of each mode is contained in the subsequent sections.

All timer operation and functionality is set using the Timer B control register, TB0CN. Three other registers are used to hold the current timer/counter value (TB0V), the capture/reload value (TB0R), and a compare value (TB0C).

In all modes of operation, the timer is enabled by setting the Timer B run control bit (TRB) in the Timer B control register to 1. If this bit is cleared to 0 (reset default condition), no timer activity is possible.

When the Timer B is operated as a timer (i.e., it counts scaled system clocks), the TBPS[2:0] bits in the timer control register determine the factor by which the active system clock is divided (prescaled) before being counted by the timer. Other relevant control bits are described in the following mode descriptions. A complete listing of the Timer B registers and bits with their effects on timer operation are given in *12.2 Timer B Register Descriptions*.

The Timer B pins, TBA and TBB, are used for GPIO and JTAG, respectively, by default. The Timer B functionality of these pins is enabled through the TB0CN register. The following sections detail the TB0CN configurations required depending on the desired Timer B function. To use the TBB pin, the JTAG port must also be disabled by setting the TAP bit in the SC register to 0.

Table 12-2. Timer B Mode Summary

TIMER B OPERATIONAL MODE	TB0CN REGISTER BIT SETTINGS						
	TBCS:TBCR	TBOE	DCEN	EXENB	C/$\overline{TB}$	CP/$\overline{RLB}$	OPTIONAL CONTROL
Auto-Reload	00	0	0	0	X	0	
Auto-Reload Using TBB Pin	00	0	0	1	X	0	
Capture Using TBB Pin	00	0	0	1	X	1	
Up/Down Count Using TBB Pin	00	0	1	0	X	0	
Up-Count PWM/Output Control	≠00	X	0	X	X	0	
Up/Down PWM/Output Control	≠00	X	1	X	X	0	
—	—	0	X	X	1	X	Input Clock = TBA Pin
Clock Output on TBA Pin	—	1	X	X	0	0	

12.1.1 Auto-Reload Mode

The 16-bit auto-reload mode of Timer B is established by clearing the CP/$\overline{RLB}$ bit to 0. In this mode, the timer performs a simple 16-bit timer or counter function that is reset to 0000h when a match between the Timer B count value register (TB0V) and the Timer B capture/reload register (TB0R) occurs. A block diagram of auto-reload mode is illustrated in Figure 12-1. If the C/$\overline{TB}$ bit is a logic 0, the timer's input clock is a prescaled system clock. When C/$\overline{TB}$ is a logic 1, pulses on the TBA pin are counted. As in all modes, counting or timing is enabled or disabled with the TRB bit.

When enabled in auto-reload mode, the Timer B begins counting up from the current value contained in the TB0V register. When the value in the TB0V register reaches the value in the capture/reload register TB0R, the TFB flag is set to 1, which can generate an interrupt if enabled. Also when this match is made, the timer reloads the TB0V register with 0000h and continues timing or counting from 0000h. The reload value contained in the TB0R register is preloaded by software. The TB0R register cannot be used for the capture function while also performing auto-reload.

While in auto-reload mode, the Timer B can also be forced to reload the TB0V register with 0000h using the TBB pin. If the EXENB bit is set to 1, a 1 to 0 transition (falling edge) on the TBB pin causes a reload. If the EXENB bit is cleared to 0, the TBB pin is ignored.

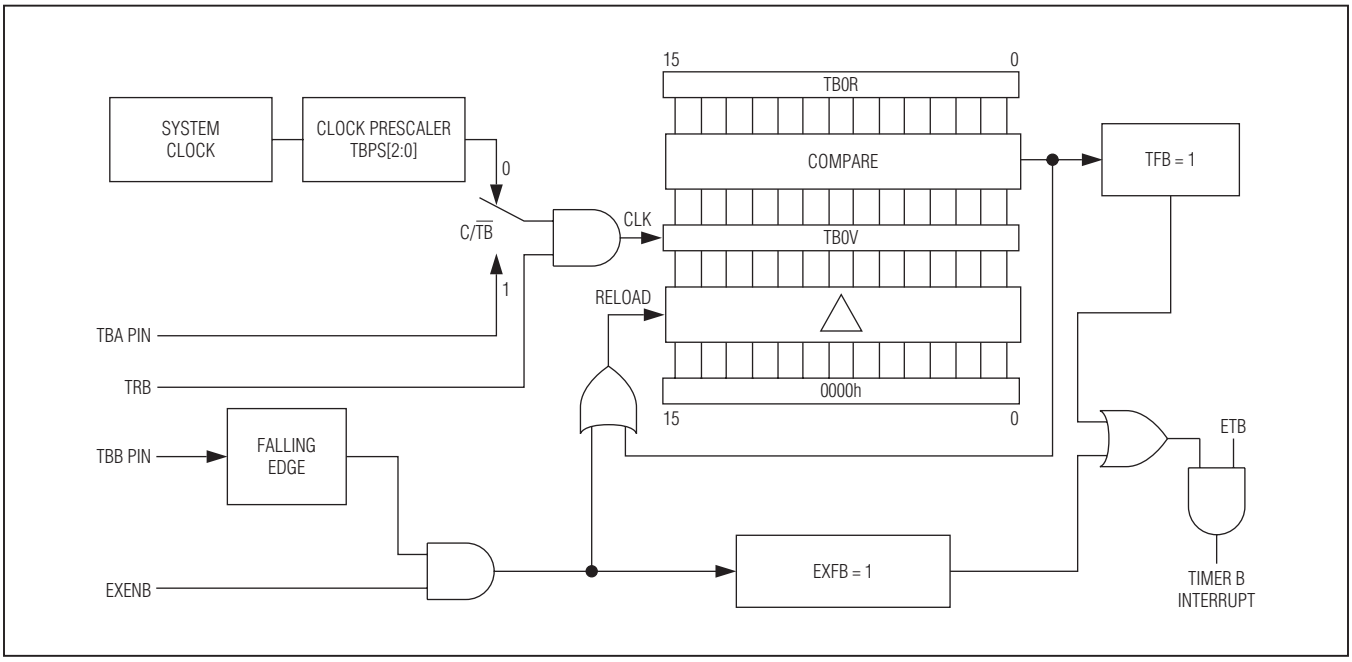


Figure 12-1. Auto-Reload Mode Block Diagram

12.1.2 Up/Down Count with Auto-Reload

The 16-Bit up/down count auto-reload mode is enabled by clearing the capture/reload bit (CP/$\overline{RLB}$) to 0 and setting the down count enable bit (DCEN) to 1. This mode is illustrated in Figure 12-2. When DCEN is set to 1 the Timer B either counts up or down, depending upon the state of the TBB pin. If the TBB pin is high, the Timer B counts up and, if the TBB pin is low, the Timer B counts down. When DCEN = 0, the Timer B only counts up.

When counting up and an overflow occurs (a match between the value in the TB0V and TB0R register), the TB0V register reloads with a value of 0000h and continues counting. When the timer is counting down and an underflow occurs (the TB0V register reaches 0000h), the TB0V register is reloaded with the value in the TB0R register and downward counting continues.

Note that in this mode of operation an overflow or underflow of the timer is provided to an edge-detection circuit as well as to the TFB bit. This edge-detection circuit toggles the EXFB bit on every overflow or underflow. Therefore, the EXFB bit behaves as a 17th bit of the counter, and can be used as such.

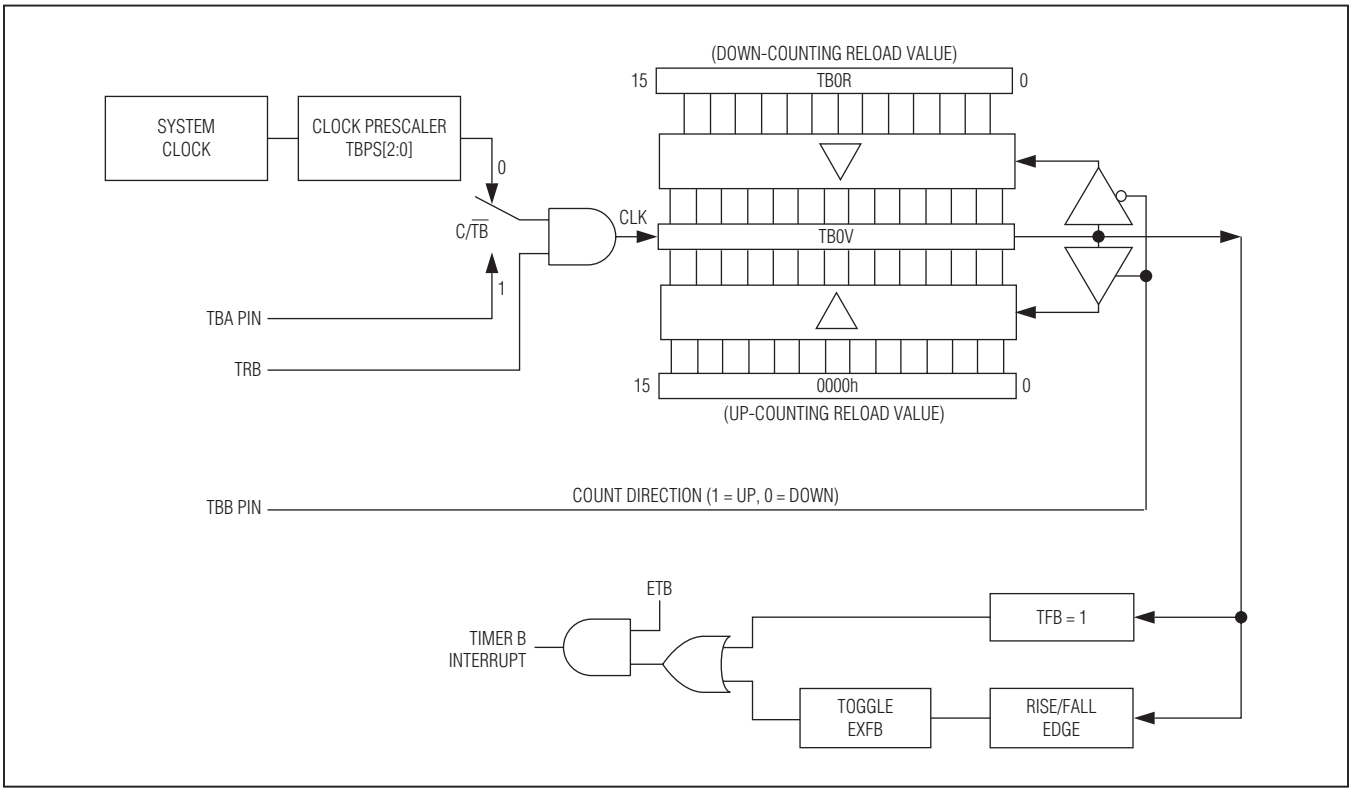


Figure 12-2. Up/Down Count with Auto-Reload Mode Block Diagram

12.1.3 Capture Mode

The Timer B 16-bit capture mode is configured by setting the CP/$\overline{RLB}$ bit to 1. A block diagram of this mode is shown in Figure 12-3. In capture mode, the Timer B can be clocked either by a prescaled version of the system clock or falling edges of the TBA pin. When the timer is enabled in capture mode, it begins counting up from the value contained in the TB0V register until reaching an overflow state. An overflow state is when the TB0V register changes from FFFFh to 0000h. When this happens, the timer overflow flag (TFB), is set, which can generate an interrupt if enabled. After an overflow the timer continues counting upward. This counting is repeated without processor intervention until the timer is disabled (TRB = 0).

The current value in TB0V is captured and copied into the capture/reload register (TB0R) when a falling edge occurs on the TBB pin and the external enable bit (EXENB) of the control register is set to 1. The EXFB flag is set when a capture occurs, which can generate an interrupt if enabled. If the EXENB bit is cleared to 0, transitions on the TBB pin do not cause a capture event.

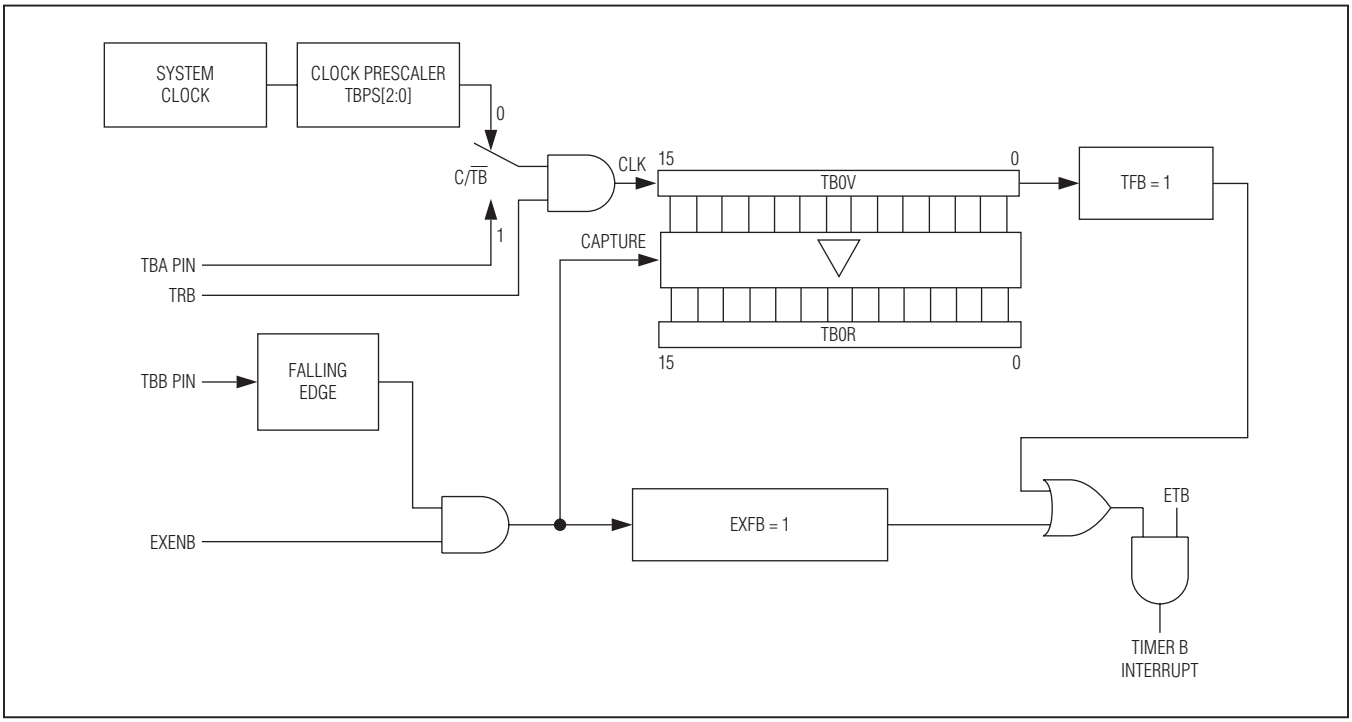


Figure 12-3. Capture Mode Block Diagram

12.1.4 Clock Output Mode

The Timer B can be configured to drive a clock output on the TBA pin as shown in Figure 12-4. For the timer to operate in this mode, the capture/reload select bit (CP/$\overline{RLB}$) and the counter/timer select bit (C/$\overline{TB}$) must be cleared to 0 and the Timer B output enable bit (TBOE) must be set to 1. In this mode, the DCEN bit has no effect. The clock signal output is a 50% duty cycle square wave with a frequency given by the equation:

$$\text{TBA OUTPUT FREQUENCY} = \frac{\text{TIMER B Clock}}{2 \times (\text{TB0R} + 1)}$$

where Timer B clock is the prescaled version of the system clock. The prescaler is set using the TBPS[2:0] bits. In clock output mode, the setting of the TFB overflow flag does not cause an interrupt.

For example, if the Timer B clock is 1MHz and the TB0R register is set to 0004h, the TBA output frequency is 100kHz.

When the Timer B is operating in clock output mode, it can also monitor and generate an interrupt when falling edges occur on the TBB pin. This is enabled by setting the EXENB and ETB bits.

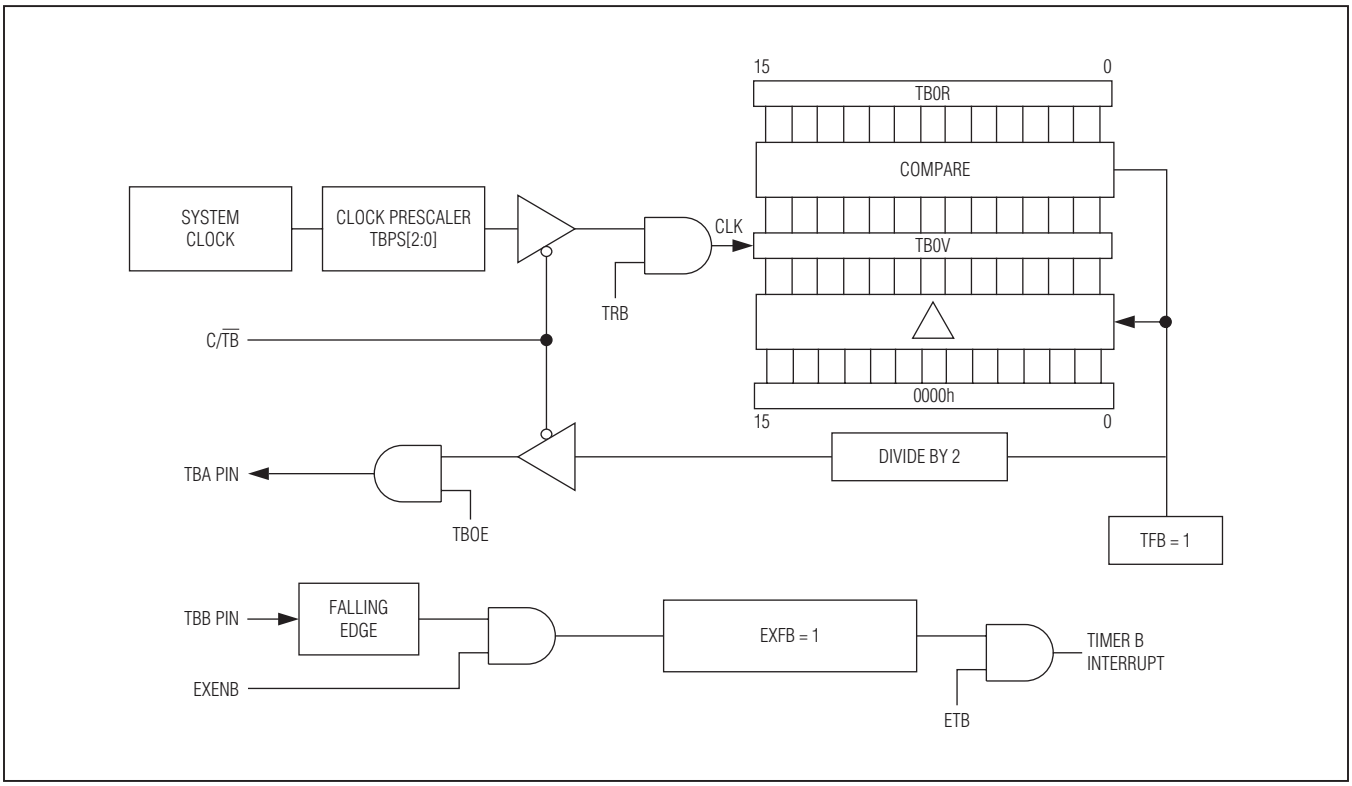


Figure 12-4. Clock Output Mode Block Diagram

12.1.5 PWM Output Mode

The PWM output mode is enabled when the Timer B is enabled (TRB = 1) and either the TBCS or TBCR bit is set to 1. Table 12-3 describes how these bits determine the specific PWM operation. When operating as a PWM output, the Timer B can provide up to 16-bit resolution of the PWM frequency or duty cycle. The counter in the Timer B can operate as count up only, or count up/down.

Table 12-3. PWM Output Modes

TBCS:TBCR	PWM MODE	TBB PIN FUNCTION	INITIAL STATE	NOTES
00	None	None (Disabled)	No change	
01	Reset	Reset on TB0C Match, Set on 0000h	Low	Will not output a 0% duty cycle.
10	Set	Set on TB0C Match, Reset on TB0R Match	High	Will not output a 100% duty cycle.
11	Toggle	Toggle on TB0C Match	No change	

Figure 12-5 shows a block diagram of the Timer B module when it is operating in PWM output mode. The TBB input function (EXENB = 1) and the PWM/output control function (TBCS or TBCR ≠ 0) can be enabled at the same time. In this configuration, the detection of a falling edge on the TBB pin results in the setting of the EXFB interrupt flag, but does not force an auto-reload.

A timed setting or clearing of the TBB pin can also be generated without the need for the CPU to time the event or use GPIO. This is accomplished by setting the compare register (TB0C) to a value greater than the reload register (TB0R). This functionality is illustrated in Figure 12-6 and Figure 12-7.

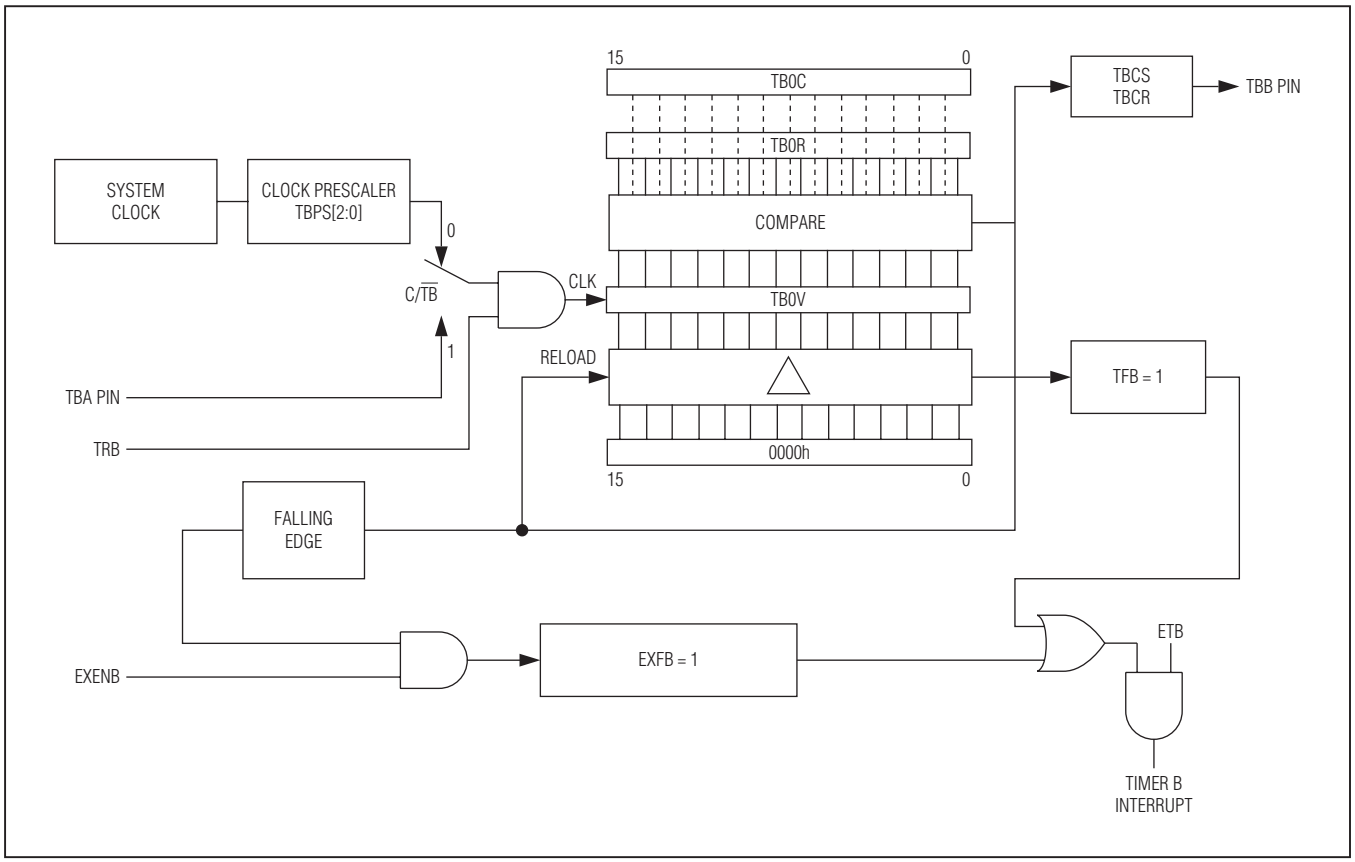


Figure 12-5. PWM Output Mode Block Diagram

12.1.5.1 Up Count PWM Output Mode

When operating in PWM output mode and configured for up count (DCEN = 0), the value in TB0V is incremented until it reaches the reload value, TB0R. At this point, TB0V is reloaded with 0000h, the TFB flag is set (which can generate an interrupt if enabled), and counting continues. Figure 12-6 illustrates the PWM waveforms when the Timer B is operating in up count PWM output mode. The period of the PWM waveform is set by the value in the TB0R register. The set and reset modes provide similar functionality. The formulas for period and duty cycle are:

$$\text{PWM PERIOD} = (\text{TB0R} + 1) \times \text{TIMER B CLOCK PERIOD}$$

$$\text{Duty Cycle in Set Mode} = \frac{\text{TB0R} - \text{TB0C}}{\text{TB0R} + 1}$$

$$\text{Duty Cycle in Reset Mode} = \frac{\text{TB0C}}{\text{TB0R} + 1}$$

The toggle mode generates a 50% duty-cycle waveform if the TB0C register remains fixed with the Timer B running. The period of the waveform is:

$$\text{PERIOD} = 2 \times (\text{TB0R} + 1) \times \text{TIMER B CLOCK PERIOD}$$

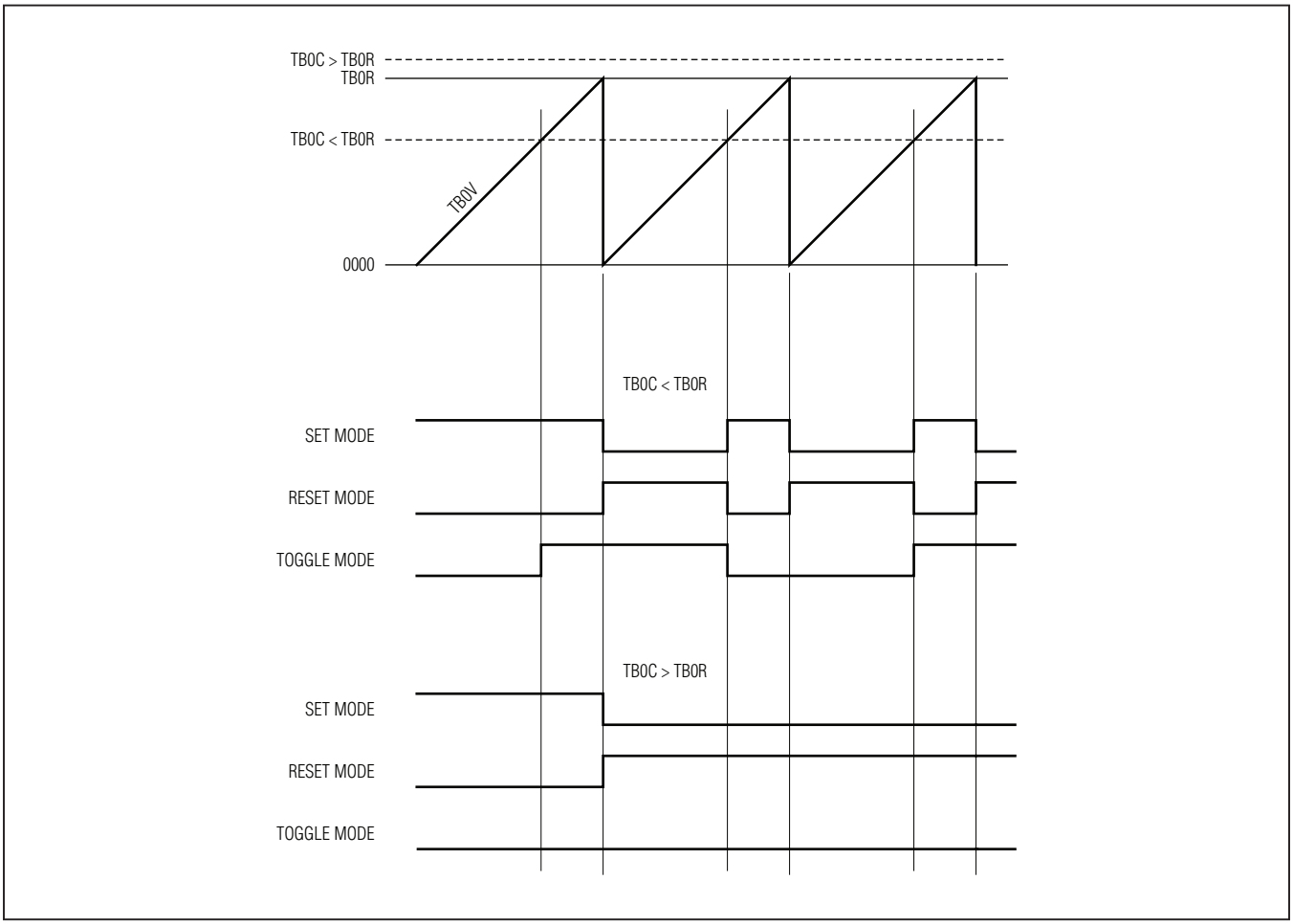


Figure 12-6. TBB Pin Waveform in Up Count PWM Output Mode

12.1.5.2 Up/Down Count PWM Output Mode

The Timer B can also operate in an up/down count configuration when in PWM output mode by setting DCEN = 1. The timer counts upward until it reaches the value in the reload register (TB0R). On the next cycle, it reverses the count direction and starts counting down. When the TB0V counter reaches 0000h, it again reverses direction and begins counting up.

When operating in an up/down count configuration and either set or reset mode, the PWM effectively allows 17-bit resolution. In set mode the duty cycle is always less than 50%, and in reset mode the duty cycle is always greater than 50%. The toggle mode provides a center-aligned 16-bit PWM with twice the period of the up counting PWM output mode. Figure 12-7 illustrates the PWM waveforms when the Timer B is operating in up/down count PWM output mode. The up/down count PWM output period and duty cycle are calculated as follows:

$$PERIOD = 2 \times TB0R \times TIMER\ B\ CLOCK\ PERIOD$$

$$\text{Duty Cycle in Set Mode} = \frac{TB0R + TB0C}{2 \times TB0R}$$

$$\text{Duty Cycle in Reset Mode} = \frac{TB0C}{2 \times TB0R}$$

$$\text{Duty Cycle in Toggle Mode} = \frac{TB0R - TB0C}{TB0R}$$

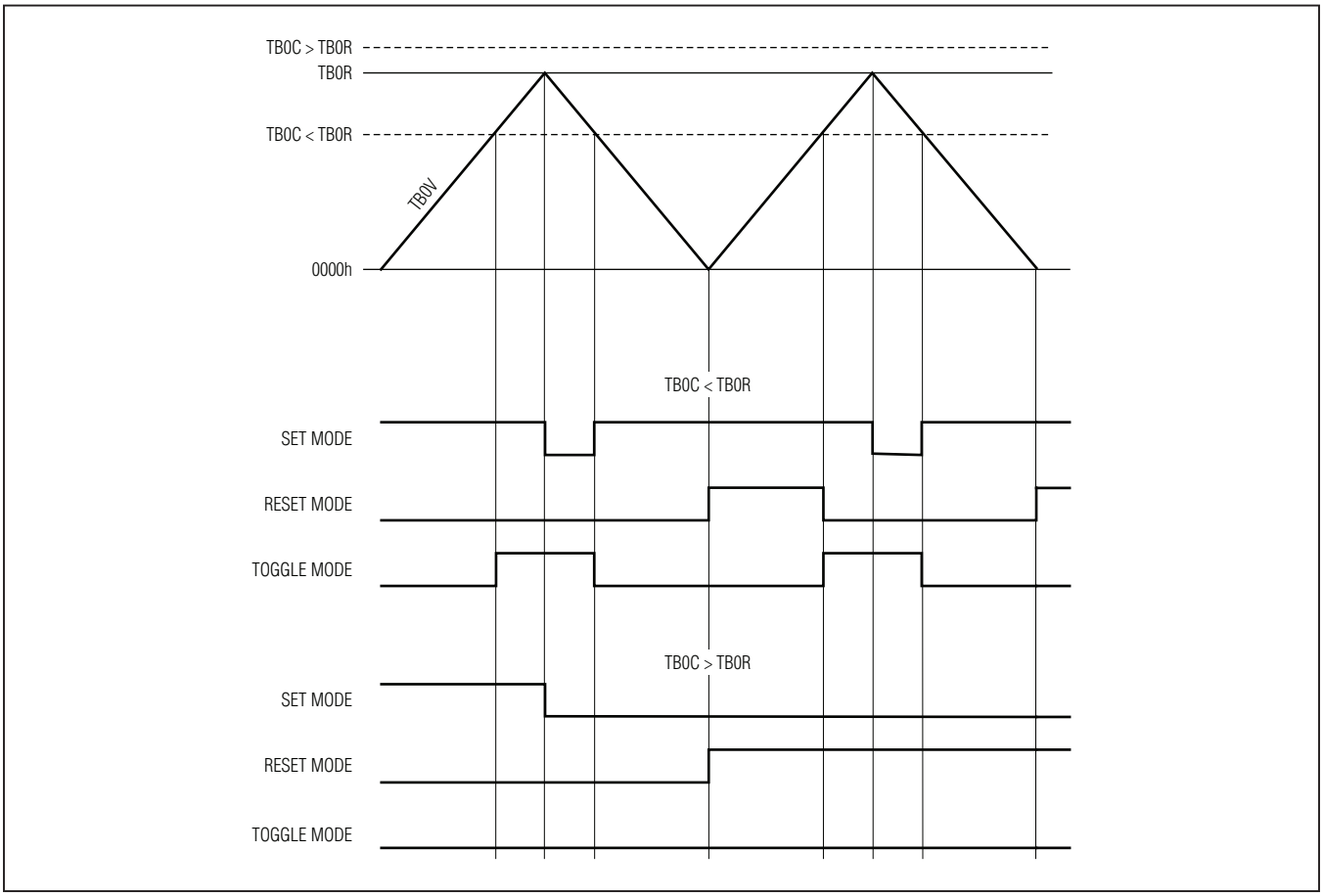


Figure 12-7. TBB Pin Waveform in Up/Down Count PWM Output Mode

12.2 Timer B Register Descriptions

The following peripheral registers are used to control the Timer B timer and counter functions. Addresses of registers are given as "Mx[yy]," where x is the module number (from 0 to 5 decimal) and yy is the register index (from 00h to 1Fh hexadecimal).

12.2.1 Timer B Control Register (TB0CN)

Register Address: M0[0Dh]

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	C/$\overline{TB}$	—	—	TBCS	TBCR	TBPS2	TBPS1	TBPS0	TFB	EXFB	TBOE	DCEN	EXENB	TRB	ETB	CP/$\overline{RLB}$
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

BIT	NAME	DESCRIPTION
15	C/$\overline{TB}$	Counter/Timer Select. This bit determines whether Timer B functions as a timer or counter. Setting this bit to 1 configures the Timer B to count negative transitions on the TBA pin. Clearing this bit to 0 configures the Timer B to function as a timer. The speed of the Timer B when operating as a timer is determined by the TBPS[2:0] bits.
14:13	—	Reserved. The user should not write to these bits.
12:11	TBCS, TBCR	TBB Pin Output Set/Reset Mode Bits. These mode bits define whether the PWM mode output function is enabled on the TBB pin and what compare mode output function is in effect. Note that the TBB pin still has certain input functionality when the PWM output function is enabled.
10:8	TBPS[2:0]	Timer B Clock Prescaler Bits. These bits select the clock prescaler applied to the system clock, which is then used as the Timer B clock. The TBPS[2:0] bits should be configured by the user when the timer is stopped (TRB = 0). While hardware does not prevent changing the TBPS[2:0] bits when the timer is running, the resultant behavior is nondeterministic.

TBPS[2:0]	TIMER INPUT CLOCK
000	Sysclk
001	Sysclk/4
010	Sysclk/16
011	Sysclk/64
100	Sysclk/256
101	Sysclk/1024
11x	Sysclk

7	TFB	Timer B Overflow Flag. This bit is set when Timer B overflows or reaches TB0R and is reloaded to 0000h. The TFB flag is also set when TB0V is equal to 0000h in down-count mode. The setting of this flag will cause an interrupt if enabled. This flag must be cleared by software. In clock output mode (TBOE = 1), the TFB flag is set on an overflow; however, the TBOE = 1 condition prevents this flag from causing an interrupt when ETB = 1.
6	EXFB	External Timer B Trigger Flag. When the Timer B is configured as a Timer (C/$\overline{TB}$ = 0) and operating in the following modes: • Capture Mode (CP/$\overline{RLB}$ = 1) • Auto-Reload Mode (CP/$\overline{RLB}$ = DCEN = 0) • PWM Mode (CP/$\overline{RLB}$ = 1 and TBCS:TBCR ≠ 00) A negative transition on the TBB pin causes the EXFB flag to be set if EXENB = 1. This flag is set when a negative edge is detected, even if the Timer B is disabled (TRB = 0). The setting of this flag causes an interrupt if enabled. If set by a negative transition, this flag must be cleared by software. When operating in up/down count with auto-reload (CP/$\overline{RLB}$ = 0, DCEN = 1, and TBCS:TBCR = 00), the EXFB flag toggles whenever the Timer B overflows or underflows. Overflow/underflow condition is described in TFB bit description. In this mode, EXFB can be used as the 17th timer bit and does not cause an interrupt.

BIT	NAME	DESCRIPTION
5	TBOE	Timer B Output Enable. Setting this bit to 1 enables the clock output function on the TBA pin if C/$\overline{TB}$ = 0. Clearing this bit to 0 allows the TBA pin to function as either a standard GPIO pin or a counter input for the Timer B.
4	DCEN	Down-Count Enable. In the compare modes, the DCEN bit controls whether the timer counts up and resets (DCEN = 0), or counts up and down (DCEN = 1). The DCEN bit only affect these two modes: • Up/down count with auto-reload: When DCEN = 1, the TBB pin controls the direction that the Timer B counts. The Timer B counts up if the TBB pin is 1 and counts down if the TBB pin is 0. Clearing this bit to 0 causes Timer B to count up only. • Up/down count PWM output mode: When DCEN = 1, the up/down count control of Timer B is controlled internally based upon the count in relation to the register settings.
3	EXENB	Timer B External Enable. Setting this bit to 1 enables the capture/reload function on the TBB pin for a negative transition. A reload results in TB0V being reset to 0000h. Clearing this bit to 0 causes the Timer B to ignore all external events on TBB pin. When operating in PWM output mode, enabling the TBB input function (EXENB = 1) allows PWM output negative transitions to set the EXFB flag; however, no reload occurs as a result of the external negative-edge detection.
2	TRB	Timer B Run Control. This bit enables Timer B operation when set to 1. Clearing this bit to 0 halts the Timer B operation and preserves the current count in TB0V.
1	ETB	Enable Timer B Interrupt. Setting this bit to 1 enables interrupts from the TFB or EXFB flags.
0	CP/$\overline{RLB}$	Capture/Reload Select. Setting this bit to 1 enables capture mode. Clearing this bit to 0 causes an auto-reload to occur when a Timer B overflow or a falling edge on TBB (EXENB = 1) is detected. It is not intended that the Timer B compare functionality should be used when operating in capture mode.

12.2.2 Timer B Value Register (TB0V)

Register Address: M0[0Bh]

The Timer B value register, TB0V, holds the 16-bit value of the Timer B counter or timer. Enabling or disabling the Timer B with the TRB bit does not reset the TB0V register. The TB0V register must be cleared by software. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

12.2.3 Timer B Capture/Reload Register (TB0R)

Register Address: M0[07h]

The Timer B capture/reload register, TB0R, is a 16-bit register that has two different functions depending on the Timer B mode of operation. When operating in capture mode, the current value in TB0V is copied to TB0R when a capture event occurs. When operating as a timer or counter, a reload of the TB0V register occurs when TB0V matches TB0R. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

12.2.4 Timer B Compare Register (TB0C)

Register Address: M0[06h]

The Timer B compare register, TB0C, is a 16-bit register that is used as a comparison to the TB0V register. Depending upon the mode of operation, the Timer B takes different actions when a match between TB0V and TB0C occurs. This register is cleared to 0000h on all forms of reset and has unrestricted read/write access.

12.3 Timer B Code Examples

12.3.1 Auto-Reload Mode

Creating a 10ms interrupt (10ms at 4MHz = 40,000 clock cycles):

```
TB0R = 40000;               //set the Reload Register
TB0V = 0x0000;              //clear the Value Register
TB0CN_bit.CPnRLB = 0;       //clear for auto reload
TB0CN_bit.ETB = 1;          //enable the interrupt
TB0CN_bit.TRB = 1;          //enable the Timer B operation
```

12.3.2 Clock Output Mode

Creating a 100kHz clock on the TBA pin:

```
TB0CN_bit.CPnRLB = 0;       //clear for reload
TB0CN_bit.TBPS = 1;         //prescaler: divide sysclk/4 for 1MHz Timer B Clock
TB0R = 4;                   //set for 100 kHz output frequency
TB0CN_bit.TBOE = 1;         //enable output on TBA pin
TB0CN_bit.TRB = 1;          //enable timer operation
```

12.3.3 PWM Output Mode

Creating a 40% duty cycle 100kHz signal:

```
TB0CN_bit.TBPS = 0;         //Timer B input clk = sysclk
TB0R = 39;                  //PWM period = 40 sysclks
TB0C = 16;                  //duty cycle = 16/40
TB0CN_bit.TBCR = 1;         //set to reset mode
TB0CN_bit.TBCS = 0;         //set to reset mode
TB0CN_bit.TRB = 1;          //enable Timer B
```

SECTION 13: SUPPLY VOLTAGE MONITOR

The MAX31782 provides features to allow monitoring of its power supply. The supply voltage monitor (SVM) monitors the V_{DD} power supply and can alert the processor through an interrupt if V_{DD} falls below a programmable threshold.

The MAX31782 provides the following power-monitoring features:

- SVM compares V_{DD} against a programmable threshold from approximately 2.7V to 5.3V.
- Optional SVM interrupt can be triggered when V_{DD} drops below the programmed threshold.
- SVM interrupt can be used to trigger exit from stop mode.

13.1 Supply Voltage Monitor Register (SVM) Descriptions

The peripheral register SVM, located in Module 1, Index 9, controls the supply voltage monitor.

13.1.1 Supply Voltage Monitor Register (SVM)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	SVTH3	SVTH2	SVTH1	SVTH0	—	—	—	SVMSTOP	SVMI	SVMIE	SVMRDY	SVMEN
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	rw*	rw*	rw*	rw*	r	r	r	rw	rw	rw	r	rw

**SVTH[3:0] bits can only be written when the SVM is not running (SVMEN = 0).*

BIT	NAME	DESCRIPTION
15:12	—	Reserved. The user should not write to these bits.
11:8	SVTH[3:0]	Supply Voltage Threshold Bits [3:0]. These bits are used to select a user-defined supply voltage threshold. If VDD is below this threshold, the SVMI bit is set and an interrupt can be generated if enabled. The threshold level can be adjusted from 2.3V to 5.3V in 0.2V increments. The default value is 00h (2.3V). Supply Voltage Monitor Threshold = 2.3V + SVTH[3:0] x 0.2V Note that the SVTH[3:0] bits can only be modified when SVMEN = 0. Writing to these bits is ignored if SVMEN = 1. SVM thresholds of 2.3V and 2.5V have no actual use because the MAX31782 enters brownout at 2.5V.
7:5	—	Reserved. The user should not write to these bits.
4	SVMSTOP	Stop Mode Supply Voltage Monitor Enable. This bit controls the operation of the SVM when the CPU is in stop mode. 0 = The SVM is disabled during stop mode. 1 = The SVM is enabled during stop mode (if SVMEN = 1).
3	SVMI	Supply Voltage Monitor Interrupt. This bit is set to 1 when the VDD supply voltage falls below the threshold defined by SVTH[3:0]. If SVMIE = 1, setting this bit to 1 by either hardware or software triggers an interrupt. This bit must be cleared by software, but if VDD is still below the threshold, the bit is immediately set again by hardware.
2	SVMIE	Supply Voltage Monitor Interrupt Enable. Setting this bit to 1 allows an interrupt to be generated (if not otherwise masked) when SVMI is set to 1. Clearing this bit to 0 disables the SVM interrupt.
1	SVMRDY	Supply Voltage Monitor Ready. This read-only status bit indicates whether the SVM is ready for use. 0 = The SVM is disabled (SVMEN = 0), stop mode was entered with SVMSTOP = 0, or the SVM is in the process of powering up. 1 = The SVM is enabled and ready for use.
0	SVMEN	Supply Voltage Monitor Enable. Setting this bit to 1 enables the SVM and begins monitoring VDD against the programmed (SVTH[3:0]) threshold. After SVMEN is set, SVMRDY is set in approximately 20µs. Clearing this bit to 0 disables the SVM.

SECTION 14: HARDWARE MULTIPLIER

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 14: Hardware Multiplier

The hardware multiplier module can be used by the MAX31782 to support high-speed multiplications. The hardware multiplier module is equipped with two 16-bit operand registers, a 32-bit read-only result register, and an accumulator of 48-bit width. The multiplier can complete a 16-bit x 16-bit multiply-and-accumulate/subtract operation in a single cycle. The hardware multiplier module supports the following operations without interfering with the normal core functions:

- Signed or Unsigned Multiply (16-bit x 16-bit)
- Signed or Unsigned Multiply-Accumulate (16-bit x 16-bit)
- Signed or Unsigned Multiply-Subtract (16-bit x 16-bit)
- Signed Multiply and Negate (16-bit x 16-bit)

14.1 Hardware Multiplier Organization

The hardware multiplier consists of two 16-bit, parallel-load operand registers (MA, MB), a read-only result register formed by two parallel 16-bit registers (MC1R and MC0R), an accumulator, which is formed by three 16-bit parallel registers (MC2, MC1, and MC0), and a status/control register (MCNT). Figure 14-1 shows a block diagram of the hardware multiplier.

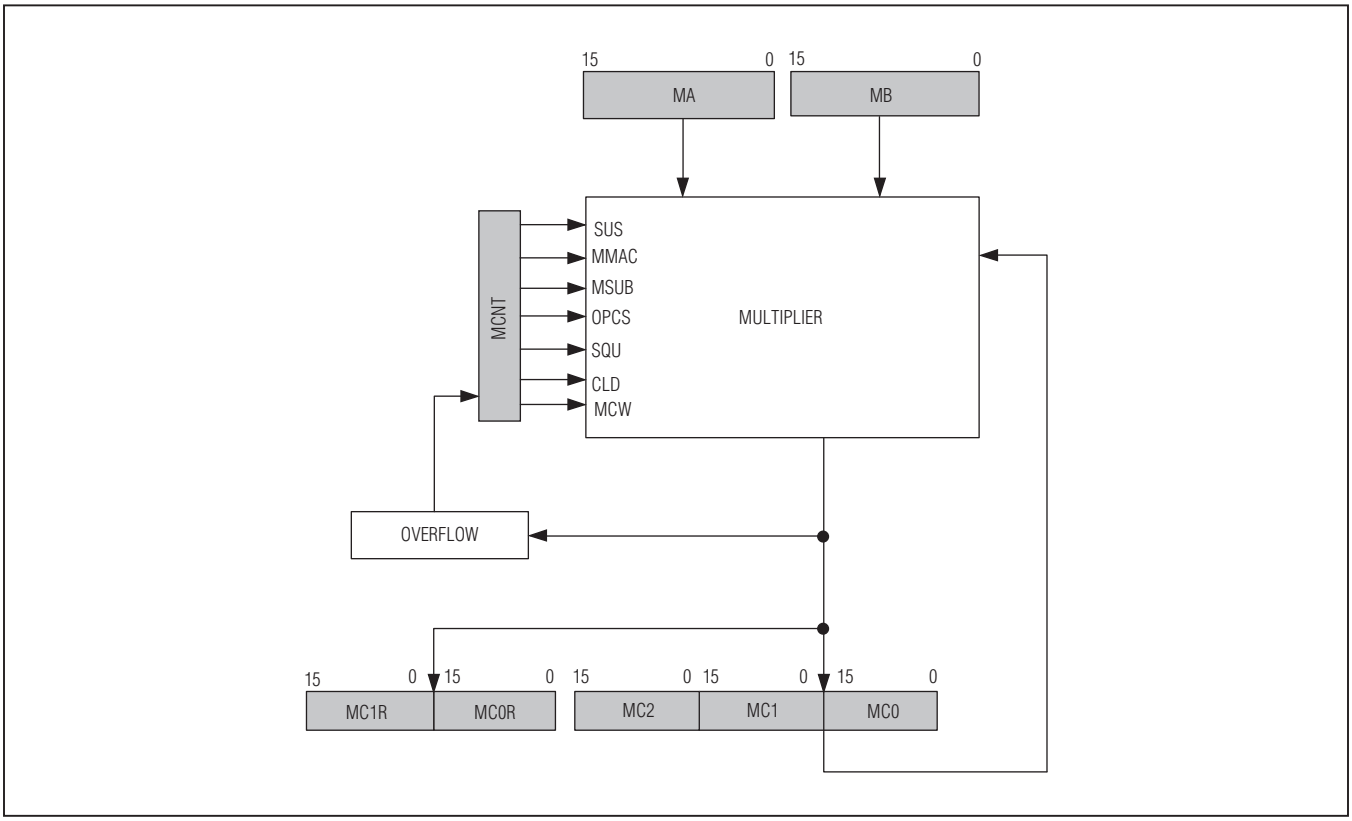


Figure 14-1. Multiplier Organization

14.2 Hardware Multiplier Controls

The selection of operation to be performed by the multiplier is determined by four control bits in the MCNT register: SUS, MSUB, MMAC, and SQU. The number of operands that must be loaded to trigger the specified operation is dictated by the OPCS bit setting, except when the square function is enabled (SQU = 1). Enabling the square function implicitly defines that only a single operand (either MA or MB) needs to be loaded to trigger the square operation, independent of the OPCS bit setting. The MCNT register bits must be configured to select the desired operation and operand count prior to loading the operand(s) to trigger the multiplier operation. Any write to MCNT automatically resets the operand load counter of the multiplier, but does not affect the operand registers, unless such action is requested using the Clear Data Registers (CLD) control bit. Once the desired operation has been specified via the MCNT register bits, loading the prescribed number of operands triggers the respective multiply, multiply-accumulate/subtract or multiply-negate operation.

14.3 Register Output Selection

The Hardware Multiplier implements the MC Register Write Select (MCW) control bit so that writing of the result to the MC2:MC0 registers can be blocked to preserve the MC registers (accumulator). When the MCW bit is configured to logic 1, the result for the given operation is not written to the MC registers. When the MCW bit is configured to logic 0, the MC registers are updated with the result of the operation. The MC1R, MC0R read-only register pair are updated independent of the MCW bit setting. This register pair always reflects the output that would normally be placed in MC1:MC0, given that MCW = 1 or MMAC = 0. When MCW = 0 and MMAC = 1, the MC1R:MC0R content may not match the MC1:MC0 register content, but it will be predictable and may be useful in certain situations. See Table 14-1 for details.

14.3.1 Signed-Unsigned Operand Selection

The operands can be either signed or unsigned numbers, but the data type must be defined by the user software via the Signed-Unsigned (SUS) bit prior to triggering the operation. For an unsigned operation, the Signed-Unsigned bit (SUS) in the MCNT register must be set to 1; for a signed operation, the SUS bit must be cleared to 0. The multiplier treats unsigned numbers as absolute magnitude. For a 16-bit positional binary number, this represents a value in the range 0 to 2^{16} - 1 (FFFFh). The signed number representation is a two's-complement value, where the most significant bit is defined as a sign bit. The range of a 16-bit two's-complement number is $-2^{(16-1)}$ (8000h) to $+2^{(16-1)}$ - 1 (7FFFh). The product of any signed operation will be sign extended before being stored or accumulated/subtracted into the MC registers. The SUS bit should always be configured to logic 0 (i.e., signed operands) for the multiply-negate operation. Attempting an unsigned multiply-negate operation results in incorrect results and setting of the OF bit. Modifying the operand data type selection via the SUS bit does not alter the contents of the MC registers. The MC registers are read/write accessible and can be modified by user code when necessary.

14.3.2 Operand Count Selection

The OPCS bit allows selection of single operand or two operands operation for the multiply and multiply-accumulate/subtract operations. When the OPCS bit is cleared to 0, the multiply or multiply-accumulate/subtract operation established by the SUS, MSUB, and MMAC bits is triggered once two operands are loaded, one to each of the MA and MB registers. When OPCS is set to 1, the operation commences once data is loaded to either MA or MB. The OPCS bit is ignored when the square operation is enabled (SQU), since loading of data to the MA or MB register actually writes to both registers.

14.4 Hardware Multiplier Operations

The control bits, which specify data type (SUS), operand count (OPCS or SQU), and destination control (MCW), have already been described. However, there are two additional MCNT register bits that serve to define the Hardware Multiplier operation. The multiply-accumulate/subtract and multiply-negate operations are enabled by the Multiply-Accumulate Enable (MMAC) and Multiply Negate (MSUB) bits in the MCNT register. When the MMAC bit is set to 1, the multiplier performs a multiply-accumulate (if MSUB = 0) or a multiply-subtract (if MSUB = 1). If MMAC is configured to 0, the multiplier result is not accumulated or subtracted, but can be stored directly (if MSUB = 0) or negated (if MSUB = 1) before storage. The multiply-negate operation (MMAC = 0, MSUB = 1) is only allowable for signed data operands (SUS = 0). For unsigned multiply-accumulate/subtract operations, the OF bit is set when a carry-out/borrow-in from the

most significant bit of the MC register occurs. For a signed two's-complement multiply-accumulate/subtract operations, the OF bit is set when the carry-out/borrow-in from the most significant magnitude position of the MC register is different from the carryout/

borrow-in of the sign position of the MC register. Since there is no overflow condition for multiply and multiply-negate operations, the OF bit is always cleared for these operations with one exception. The OF bit will be set to logic 1 if an unsigned multiply-negate (invalid operation) is requested. Table 14-1 shows the operations supported by the multiplier and associated MCNT control bit settings.

14.4.1 Accessing the Multiplier

There are no restrictions on how quickly data is entered into the operand registers or the order of data entry. The only requirement to do a calculation is to perform the loading of MA and/or MB registers having specified data type and operation in the MCNT register. The multiplier keeps track of the writes to the MA and MB registers, and starts calculation immediately after the prescribed number of operands is loaded. If two operands are specified for the operation, the multiplier waits for the second operand to be loaded into the other operand register before starting the actual calculation. If for any reason software needs to reload the first operand, it should either reload that same operand register or use the CLD bit in the MCNT register to reinitialize the multiplier; otherwise, loading data to another operand register triggers the calculation. The CLD bit is a self-clearing bit that can be used for multiplier initialization. When it is set, it clears all data registers and the OF bit to zero and resets the multiplier operand write counter.

The specified hardware multiplier operation begins when the final operand(s) is loaded and will complete in a single cycle. The read-only MC1R, MC0R result registers can be accessed in the very next cycle unless accumulation/subtraction with MC2:0 is requested (MCW = 0 and MMAC = 1), in which case, one cycle is required so that stable data can be read. When MCW = 0, the MC2:0 registers always require one wait cycle before the operation result is accessible. The single wait cycle needed for updating the MC2:0 registers with a calculated result does not prevent initiating another calculation. Back-to-back operations can be triggered (independent of data type and operand count) without the need of wait state between the loadings of operands.

Table 14-1. Hardware Multiplier Operations

MCW:MSUB:MMAC	OPERATION	MC2	MC1	MC0	MC1R:MC0R	OF STATUS
000	Multiply	MA*MB			MA*MB	No
001	Multiply-Accumulate	MC+(MA*MB)			32lsbits of (MC+2*(MA*MB))	Yes
010	Multiply-Negate (SUS = 0 only)	-(MA*MB)			-(MA*MB)	No
011	Multiply-Subtract	MC-(MA*MB)			32lsbits of (MC-2*(MA*MB))	Yes
100	Multiply	MC2	MC1	MC0	MA*MB	No
101	Multiply-Accumulate	MC2	MC1	MC0	32lsbits of (MC+(MA*MB))	No
110	Multiply-Negate (SUS = 0 only)	MC2	MC1	MC0	-(MA*MB)	No
111	Multiply-Subtract	MC2	MC1	MC0	32lsbits of (MC-(MA*MB))	No

14.5 Hardware Multiplier Peripheral Registers

The hardware multiplier registers are detailed in the following sections. Addresses of registers are given as "Mx[yy]," where x is the module number (from 0 to 5 decimal) and yy is the register index (from 00h to 1Fh hexadecimal).

Table 14-2. Hardware Multiplier Registers

REGISTER	ADDRESS	FUNCTION
MCNT	M5[00h]	Multiplier Control Register. Selects operation, data type, operand count, hardware square function, and write option on the MC register. Also contains the overflow flag and the clear control for operand registers and accumulator.
MA	M5[01h]	Multiplier Operand A Register. Used by the user software to load one of the 16-bit values for a hardware multiplier operation.
MB	M5[02h]	Multiplier Operand B Register. Used by the user software to load one of the 16-bit values for a hardware multiplier operation.
MC2	M5[03h]	Multiplier Accumulate Register 2. Contains the two most significant bytes of the accumulator register. The 48-bit accumulator is formed by MC2, MC1, and MC0. The most significant bit of this register is the signed bit for signed operations.
MC1	M5[04h]	Multiplier Accumulate Register 1. Contains bytes 3 and 2 of the accumulator register. The 48-bit accumulator is formed by MC2, MC1, and MC0.
MC0	M5[05h]	Multiplier Accumulate Register 0. Contains the two least significant bytes of the accumulator register. The 48-bit accumulator is formed by MC2, MC1, and MC0.
MC1R	M5[06h]	Multiplier Read Register 1. Contains bytes 1 and 0 result from the last operation when MCW bit is 1 or the last operation is either multiply-only or multiply-negate. The contents of this register remain until an SFR related to the multiplier has been changed.
MC0R	M5[07h]	Multiplier Read Register 0. Contains bytes 3 and 2 result from the last operation when MCW bit is 1 or the last operation is either multiply-only or multiply-negate. The contents of this register remain unchanged until an SFR related to the multiplier has been changed.

14.5.1 Multiplier Control Register (MCNT)

Bit	7	6	5	4	3	2	1	0
Name	OF	MCW	CLD	SQU	OPCS	MSUB	MMAC	SUS
Reset	0	0	0	0	0	0	0	0
Access	r	rw	rw	rw	rw	rw	rw	rw

BIT	NAME	DESCRIPTION
7	OF	**Overflow Flag**. This bit is set to logic 1 when an overflow occurred for the last operation. This bit can be set for accumulation/subtraction operations or unsigned multiply-negate attempts. This bit is automatically cleared to 0 following a reset, starting a multiplier operation, or setting of the CLD bit to 0.
6	MCW	**MC Register Write Select**. The state of the MCW bit determines if an operation result will be placed into the accumulator registers (MC). 0 = The result will be written to the MC registers. 1 = The result is not written to the MC registers (MC register content is unchanged).
5	CLD	**Clear Data register**. This bit initializes the operand registers and the accumulator of the multiplier. When it is set to 1, the contents of all data registers and the OF bit are cleared to 0 and the operand load counter is reset immediately. This bit is cleared by hardware automatically. Writing a 0 to this bit has no effect.
4	SQU	**Square Function Enable**. This bit supports the hardware square function. When this bit is set to logic 1, a square operation is initiated after an operand is written to either the MA or the MB register. Writing data to either of the operand registers writes to both registers and triggers the specified square or square-accumulate/subtract operation. Setting this bit to 1 also overrides the OPCS bit setting. When SQU is cleared to logic 0, the hardware square function is disabled. 0 = Square function disabled 1 = Square function enabled
3	OPCS	**Operand Count Select**. This bit defines how many operands must be loaded to trigger a multiply or multiply-accumulate/subtract operation (except when SQU = 1, since this implicitly specifies a single operand). When this bit is cleared to logic 0, both operands (MA and MB) must be written to trigger the operation. When this bit is set to 1, the specified operation is triggered once either operand is written. 0 = Both operands (MA and MB) must be written to trigger the multiplier operation. 1 = Loading one operand (MA or MB) triggers the multiplier operation.
2	MSUB	**Multiply-Accumulate Negate**. Configuring this bit to logic 1 enables negation of the product for signed multiply operations and subtraction of the product from the accumulator (MC[2:0]) when MMAC = 1. When MSUB is configured to logic 0, the product of multiply operations will not be negated and accumulation is selected when MMAC = 1.
1	MMAC	**Multiply-Accumulate Enable**. This bit enables the accumulate or subtract operation (as per MSUB) for the hardware multiplier. When this bit is cleared to logic 0, the multiplier performs only multiply operations. When this bit is set to logic 1, the multiplier performs a multiply-accumulate or multiply-subtract operation based upon the MSUB bit. 0 = Accumulate/subtract operation disabled 1 = Accumulate/subtract operation enabled
0	SUS	**Signed-Unsigned**. This bit determines the data type of the operands. When this bit is cleared to logic 0, the operands are treated as two's complement values and the multiplier performs a signed operation. When this bit is set to logic 1, the operands are treated as absolute magnitudes and the multiplier performs an unsigned operation. 0 = Signed operands 1 = Unsigned operands

14.5.2 Multiplier Operand A Register (MA)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	MA.15	MA.14	MA.13	MA.12	MA.11	MA.10	MA.9	MA.8	MA.7	MA.6	MA.5	MA.4	MA.3	MA.2	MA.1	MA.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Multiplier Operand A Register. This operand A register is used by the application code to load 16-bit values for multiplier operations.

14.5.3 Multiplier Operand B Register (MB)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	MB.15	MB.14	MB.13	MB.12	MB.11	MB.10	MB.9	MB.8	MB.7	MB.6	MB.5	MB.4	MB.3	MB.2	MB.1	MB.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Multiplier Operand B Register. This operand B register is used by the application code to load 16-bit values for multiplier operations.

14.5.4 Multiplier Accumulator 2 Register (MC2)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	MC2.15	MC2.14	MC2.13	MC2.12	MC2.11	MC2.10	MC2.9	MC2.8	MC2.7	MC2.6	MC2.5	MC2.4	MC2.3	MC2.2	MC2.1	MC2.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Multiplier Accumulator 2 Register. The MC2 register represents the two most significant bytes of the accumulator register. The 48-bit accumulator is formed by MC2, MC1, and MC0. For a signed operation, the most significant bit of this register is the sign bit.

14.5.5 Multiplier Accumulator 1 Register (MC1)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	MC1.15	MC1.14	MC1.13	MC1.12	MC1.11	MC1.10	MC1.9	MC1.8	MC1.7	MC1.6	MC1.5	MC1.4	MC1.3	MC1.2	MC1.1	MC1.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Multiplier Accumulator 1 Register: The MC1 register represents bytes 3 and 2 of the accumulator register. The 48-bit accumulator is formed by MC2, MC1, and MC0.

14.5.6 Multiplier Accumulator 0 Register (MC0)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	MC0.15	MC0.14	MC0.13	MC0.12	MC0.11	MC0.10	MC0.9	MC0.8	MC0.7	MC0.6	MC0.5	MC0.4	MC0.3	MC0.2	MC0.1	MC0.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Multiplier Accumulator 0 Register: The MC0 register represents the two least significant bytes of the accumulator register. The 48-bit accumulator is formed by MC2, MC1, and MC0.

14.5.7 Multiplier Read Register 1 (MC1R)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	MC1R.15	MC1R.14	MC1R.13	MC1R.12	MC1R.11	MC1R.10	MC1R.9	MC1R.8	MC1R.7	MC1R.6	MC1R.5	MC1R.4	MC1R.3	MC1R.2	MC1R.1	MC1R.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Multiplier Read Register 1: The MC1R register represents bytes 3 and 2 from the result of the last operation when MCW = 1 or the last operation was a multiply or multiply-negate. When MCW = 0 and the last operation was a multiply-accumulate/subtract, the contents of this register may or may not agree with the contents of MC1 due to the combinatorial nature of the adder. The contents of this register can change if MCNT, MA, MB, or MC[2:0] is changed.

14.5.8 Multiplier Read Register 0 (MC0R)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	MC0R.15	MC0R.14	MC0R.13	MC0R.12	MC0R.11	MC0R.10	MC0R.9	MC0R.8	MC0R.7	MC0R.6	MC0R.5	MC0R.4	MC0R.3	MC0R.2	MC0R.1	MC0R.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Multiplier Read Register 0: The MC1R register represents bytes 1 and 0 from the result of the last operation when MCW = 1 or the last operation was a multiply or multiply-negate. When MCW = 0 and the last operation was a multiply-accumulate/subtract, the contents of this register may or may not agree with the contents of MC0 due to the combinatorial nature of the adder. The contents of this register can change if MCNT, MA, MB, or MC[2:0] is changed.

14.6 Hardware Multiplier Examples

The following are code examples of multiplier operations.

```
;Unsigned Multiply 16-bit x 16-bit
move    MCNT, #21h          ; CLD=1, SUS=1 (unsigned)
move    MA, #0FFFh          ; MC2:0=0000_0000_0000h
move    MB, #1001h          ; MC1R:MC0R= 00FF_FFFFh
                            ; MC2:0=0000_00FF_FFFFh

;Signed Multiply 16-bit x 16-bit
move    MCNT, #20h          ; CLD=1, SUS=0 (signed)
move    MA, #F001h          ; MC2:0=0000_0000_0000h
move    MB, #1001h          ; MC1R:MC0R= FF00_0001h
                            ; MC2:0=FFFF_FF00_0001h

;Unsigned Multiply-Accumulate 16-bit x 16-bit
                            ; MC2:0=0000_0100_0001h
move    MCNT, #03h          ; MMAC=1, SUS=1 (unsigned)
move    MA, #0FFFh          ;
move    MB, #1001h          ;
                            ; MC1R:MC0R=02FF_FFFFh
                            ; MC2:0=0000_0200_0000h
```

```
;Signed Multiply-Accumulate 16-bit x 16-bit
                          ; MC2:0=0000_0100_0001h
move    MCNT, #02h        ; SUS=0 (signed)
move    MA, #F001h        ;
move    MB, #1001h        ;
                          ; MC1R:MC0R= FF00_0003h
                          ; MC2:0=0000_0000_0002h

;Unsigned Multiply-Subtract 16-bit x 16-bit
                          ; MC2:0=0000_0100_0001h
move    MCNT, #07h        ; MMAC=1, MSUB=1, SUS=1 (unsigned)
move    MA, #0FFFh        ;
move    MB, #1001h        ;
                          ; MC1R:MC0R=FF00_0003h
                          ; MC2:0=0000_0000_0002h

;Signed Multiply-Subtract 16-bit x 16-bit
                          ; MC2:0=0000_0100_0001h
move    MCNT, #06h        ; MMAC=1, MSUB=1, SUS=0 (signed)
move    MA, #F001h        ;
move    MB, #1001h        ;
                          ; MC1R:MC0R= 02FF_FFFFh
                          ; MC2:0=0000_0200_0000h

;Signed Multiply Negate 16-bit x 16-bit
move    MCNT, #24h        ; CLD=1, MSUB=1, SUS=0 (signed)
move    MA, #F001h        ; MC2:0=0000_0000_0000h
move    MB, #1001h        ; MC1R:MC0R =00FF_FFFFh
                          ; MC2:0=0000_00FF_FFFFh
```

SECTION 15: WATCHDOG TIMER

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 15: WATCHDOG TIMER

The watchdog timer is a user-programmable clock counter that can serve as a time-base generator, an event timer, or a system supervisor. As shown in Figure 15-1, the timer is driven by the main system clock and is supplied to a series of dividers. If the watchdog interrupt and the watchdog reset are disabled (WDCN.EWDI = 0 and WDCN.EWT = 0), the watchdog timer and its input clock are disabled. Whenever the watchdog timer is disabled, the watchdog interval timer (per WDCN.WD[1:0] bits) and 512 clock reset counter are reset if either the interrupt or reset function is enabled. When the watchdog timer is initially enabled, there is a one-clock to three-clock-cycle delay before it starts. The divider output is selectable, and determines the interval between timeouts. When the timeout is reached, an interrupt flag is set, and if enabled, an interrupt occurs. A watchdog-reset function is also provided in addition to the watchdog interrupt. The reset and interrupt are completely discrete functions that can be acknowledged or ignored, together or separately for various applications.

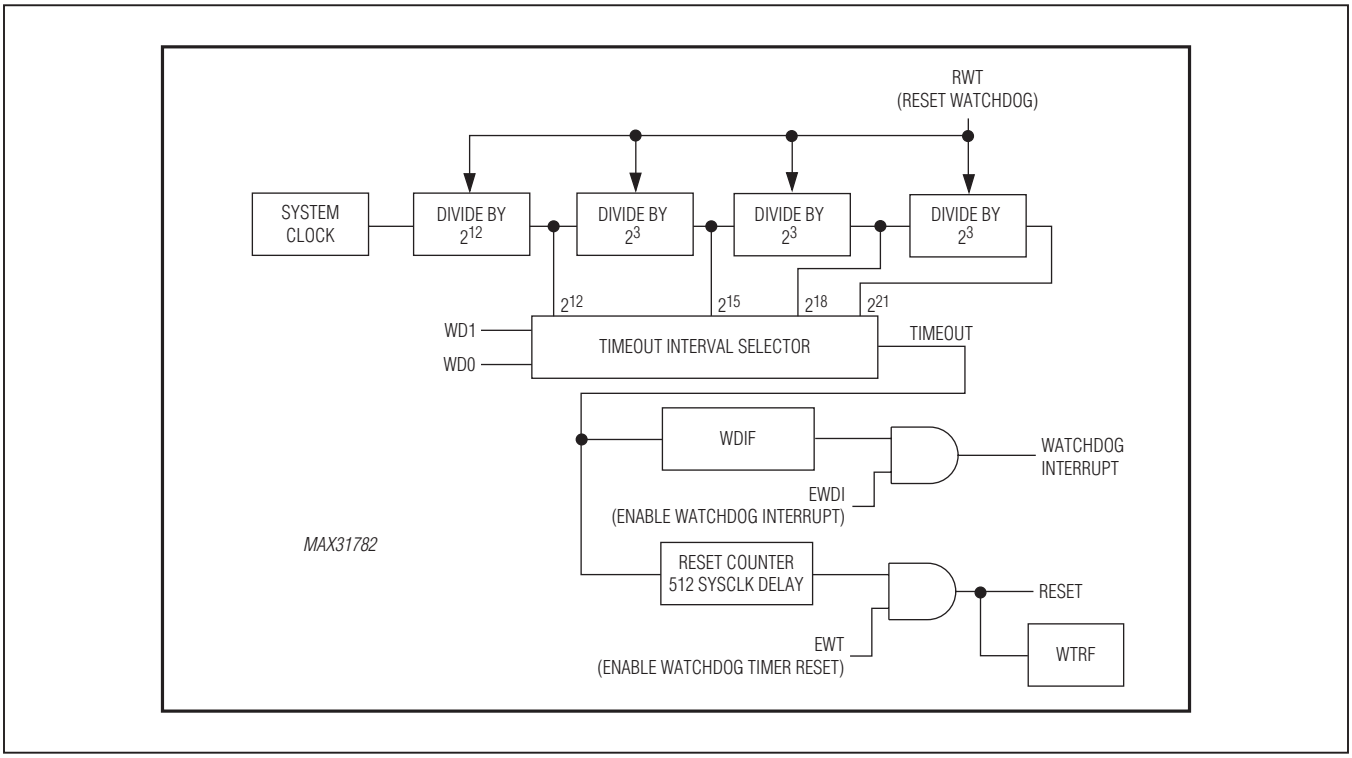


Figure 15-1. Watchdog Timer Block Diagram

15.1 Watchdog Timer Description

When the watchdog timer is enabled, it begins counting system clock cycles. The watchdog count is reset any time RWT is set to 1. If the watchdog count reaches the time interval set by WD[1:0], a watchdog timeout occurs, setting the watchdog interrupt flag (WDCN.WDIF). A watchdog timeout also generates an interrupt and/or reset to the MAX31782. Table 15-1 describes the possible states of the watchdog timer.

Table 15-1. Watchdog Operating States

EWT	EWDI	WDIF	ACTIONS
x	x	0	No interrupt has occurred.
0	0	x	Watchdog disable, clock is gated off.
0	1	1	Watchdog interrupt has occurred.
1	0	1	No interrupt has been generated. Watchdog reset occurs in 512 system clock cycles if RWT is not set or WDIF is not cleared.
1	1	1	Watchdog interrupt has occurred. Watchdog reset occurs in 512 system clock cycles if RWT is not set or WDIF is not cleared.

15.1.2 Watchdog Timer Interrupt Operation

The watchdog interrupt is enabled using the enable watchdog timer interrupt (WDCN.EWDI) bit. When the timeout occurs, the watchdog timer sets the watchdog interrupt flag bit (WDCN.WDIF), and an interrupt occurs if the interrupt global enable (IC.IGE) and system interrupt mask (IMR.IMS) are set and an interrupt is not currently being serviced (IC.INS = 0). The watchdog interrupt flag must be cleared by software.

15.1.2 Watchdog Timer Reset Operation

To reset the MAX31782, the watchdog timer reset function must be enabled by setting the enable watchdog timer reset (WDCN.EWT) bit. When a watchdog timeout occurs, the WDIF flag is set and an interrupt is generated if enabled. Following the timeout, the watchdog counts an additional 512 system clock cycles. To avoid a reset, software must either set the RWT bit or clear the EWT bit. This can occur at any time during the watchdog timer interval or the additional 512 system clock cycles after WDIF is set. At the end of the 512 system clock cycles the MAX31782 is reset. When the reset occurs, the watchdog timer reset flag (WDCN.WTRF) automatically is set to indicate the cause of the reset. Software must clear this bit manually.

15.1.3 Watchdog Timer Applications

Using the watchdog interrupt during software development can allow the user to select ideal watchdog reset locations. Code is first developed without enabling the watchdog interrupt or reset functions. Once the program is complete, the watchdog interrupt function is enabled to identify the required locations in code to set the RWT bit. Incrementally adding instructions to reset the watchdog timer prior to each address location (identified by the watchdog interrupt) allows the code to eventually run without receiving a watchdog interrupt. At this point the watchdog timer reset can be enabled without the potential of generating unwanted resets. At the same time the watchdog interrupt can also be disabled. Proper use of the watchdog interrupt with the watchdog reset allows interrupt software to survey the system for errant conditions.

When using the watchdog timer as a system monitor, the watchdog reset function should be used. If the interrupt function were used, the purpose of the watchdog would be defeated. For example, assume the system is executing errant code prior to the watchdog interrupt. The interrupt would temporarily force the system back into control by vectoring the CPU to the interrupt service routine. Restarting the watchdog and exiting by an RETI or RET would return the processor to the errant code. By using the watchdog reset function, the processor is restarted from the beginning of the program and therefore placed into a known state.

The watchdog timer is controlled by the Watchdog Timer Control register, WDCN. The WDCN register is one of the system registers and is located in Module 8, Register 15.

15.2.4 Watchdog Timer Control Register (WDCN)

Bit	7	6	5	4	3	2	1	0
Name	POR	EWDI	WD1	WD0	WDIF	WTRF	EWT	RWT
Reset	s*	s*	0	0	0	s*	s*	0
Access	rw	rw	rw	rw	rw	rw	rw	rw

**Bits 5, 4, 3 and 0 are cleared to 0 on all forms of reset; for others, see individual bit descriptions.*

<table>
<tr><th>BIT</th><th>NAME</th><th>DESCRIPTION</th></tr>
<tr><td>7</td><td>POR</td><td>Power-On-Reset Flag. This bit is set to 1 whenever a power-on/brownout reset occurs. It is unaffected by other forms of reset. This bit can be checked by software following a reset to determine if a power-on/brownout reset occurred. It should always be cleared by software following a reset to ensure that the sources of following resets can be determined correctly.</td></tr>
<tr><td>6</td><td>EWDI</td><td>Enable Watchdog Timer Interrupt. If this bit is set to 1, an interrupt request can be generated when the WDIF bit is set to 1 by any means. If this bit is cleared to 0, no interrupt occurs when WDIF is set to 1; however, it does not stop the watchdog timer or prevent watchdog resets from occurring if EWT = 1. If EWT = 0 and EWDI = 0, the watchdog timer is stopped. If the watchdog timer is stopped (EWT = 0 and EWDI = 0), setting the EWDI bit resets the watchdog interval and reset counter and enables the watchdog timer. This bit is cleared to 0 by power-on reset and is unaffected by other forms of reset.</td></tr>
<tr><td>5:4</td><td>WD[1:0]</td><td>Watchdog Timer Interval Control Bits. These bits determine the watchdog timeout interval. The timeout interval is set in terms of system clocks. Modifying the watchdog interval automatically resets the watchdog timer unless the 512 system clock reset counter is already in progress, in which case, changing the WD[1:0] bits does not affect the watchdog timer or reset counter.
<table>
<tr><th>WD1</th><th>WD0</th><th>CLOCKS UNTIL INTERRUPT</th><th>CLOCKS UNTIL RESET</th></tr>
<tr><td>0</td><td>0</td><td>2^{12}</td><td>2^{12} + 512</td></tr>
<tr><td>0</td><td>1</td><td>2^{15}</td><td>2^{15} + 512</td></tr>
<tr><td>1</td><td>0</td><td>2^{18}</td><td>2^{18} + 512</td></tr>
<tr><td>1</td><td>1</td><td>2^{21}</td><td>2^{21} + 512</td></tr>
</table></td></tr>
<tr><td>3</td><td>WDIF</td><td>Watchdog Interrupt Flag. This bit is set to 1 when the watchdog timer interval has elapsed or can be set to 1 by user software. When WDIF = 1, an interrupt request occurs if the watchdog interrupt has been enabled (EWDI = 1) and not otherwise masked or prevented by an interrupt already in service (i.e., IGE = 1, IMS = 1, and INS = 0 must be true for the interrupt to occur). This bit should be cleared by software before exiting the interrupt service routine to avoid repeated interrupts. Furthermore, if the watchdog reset has been enabled (EWT = 1), a reset is scheduled to occur 512 system clock cycles following setting of the WDIF bit.</td></tr>
<tr><td>2</td><td>WTRF</td><td>Watchdog Timer Reset Flag. This bit is set to 1 when the watchdog resets the processor. Software can check this bit following a reset to determine if the watchdog was the source of the reset. Setting this bit to 1 in software does not cause a watchdog reset. This bit is cleared by power-on reset only and is unaffected by other forms of reset. It should also be cleared by software following any reset so that the source of the next reset can be correctly determined by software. This bit is only set to 1 when a watchdog reset actually occurs. If EWT is cleared to 0 when the watchdog timer elapses, this bit is not set.</td></tr>
<tr><td>1</td><td>EWT</td><td>Enable Watchdog Timer Reset. If this bit is set to 1 when the watchdog timer elapses, the watchdog resets the MAX31782 512 system clock cycles later unless action is taken to disable the reset event. Clearing this bit to 0 prevents a watchdog reset from occurring but does not stop the watchdog timer or prevent watchdog interrupts from occurring if EWDI = 1. If EWT = 0 and EWDI = 0, the watchdog timer is stopped. If the watchdog timer is stopped (EWT = 0 and EWDI = 0), setting the EWT bit resets the watchdog interval and reset counter and enables the watchdog timer. This bit is cleared on power-on reset and is unaffected by other forms of reset.</td></tr>
<tr><td>0</td><td>RWT</td><td>Reset Watchdog Timer. Setting this bit to 1 resets the watchdog timer count. If watchdog interrupt and/or reset modes are enabled, the software must set this bit to 1 before the watchdog timer elapses to prevent an interrupt or reset from occurring. This bit always returns 0 when read.</td></tr>
</table>

SECTION 16: TEST ACCESS PORT (TAP)

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 16: TEST ACCESS PORT (TAP)

The MAX31782 incorporates a test access port (TAP) and TAP controller for communication with a host device across a 4-wire synchronous serial interface. The TAP can be used by the MAX31782 to support in-system programming and/or in-circuit debug. The TAP is compatible with the JTAG IEEE standard 1149 and is formed by four interface signals as described in Table 16-1. For detailed information on the TAP and TAP controller, refer to IEEE STD 1149.1 "IEEE Standard Test Access Port and Boundary-Scan Architecture."

Table 16-1. Test Access Port Pins

EXTERNAL PIN SIGNAL	FUNCTION
TDO (Test Data Output)	Serial-Data Output. This signal is used to serially transfer internal data to the external host. Data is transferred least significant bit first. Data is driven out only on the falling edge of TCK, only during TAP Shift-IR or Shift-DR states and is otherwise inactive.
TDI (Test Data Input)	Serial-Data Input. This signal is used to receive data serially transferred by the host. Data is received least significant bit first and is sampled on the rising edge of TCK. TDI is weakly pulled high internally when TAP = 1.
TCK (Test Clock Input)	Serial Shift Clock Provided by Host. When this signal is stopped at 0, storage elements in the TAP logic must retain their data indefinitely. TCK is weakly pulled high internally when TAP = 1.
TMS (Test Mode Select Input)	Mode Select Input. This signal is sampled at the rising edge of TCK and controls movement between TAP states. TMS is weakly pulled high internally when TAP = 1.

These pins default to the TAP/JTAG function on reset, which means that the part is always ready for in-circuit debugging or in-circuit programming operations following any reset. Once an application has been loaded and starts running, the TAP/JTAG port can still be used for in-circuit debugging operations. If in-circuit debugging functionality is not needed, the associated port pins can be reclaimed for application use by setting the TAP bit (SC.7) bit to 0. This disables the TAP/JTAG interface and allows the four pins to operate as normal port pins. See Figure 16-1.

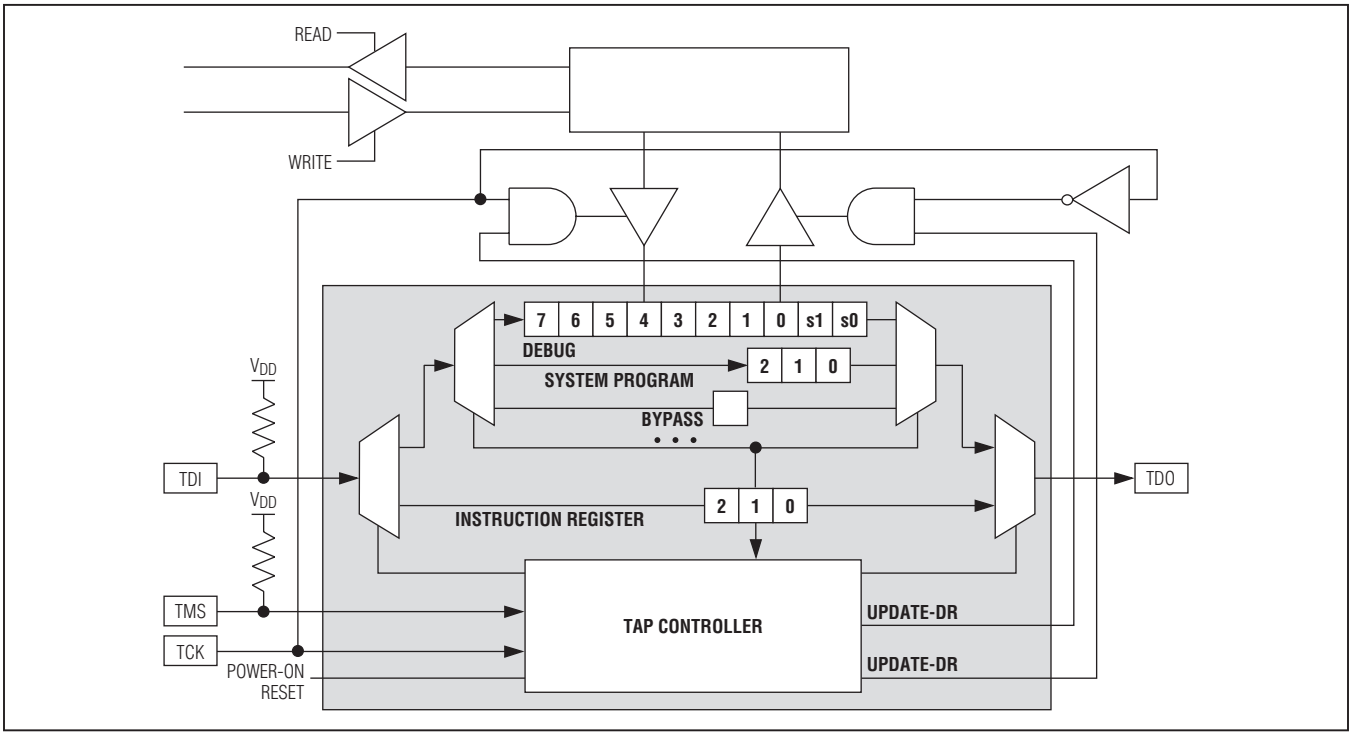


Figure 16-1. TAP and TAP Controller

16.1 TAP Controller

The TAP controller is a synchronous state machine that responds to changes at the TMS and TCK signals. Based on its state transition, the controller provides the clock and control sequence for TAP operation. The performance of the TAP is dependent on the TCK clock frequency. The maximum TCK clock frequency should be limited to 1/8 the system clock frequency. This section provides a brief description of the state machine and its state transitions. The state diagram in Figure 16-2 summarizes the transitions caused by the TMS signal sampling on the rising edge at TCK. The TMS signal value is presented adjacent to each state transition in the figure.

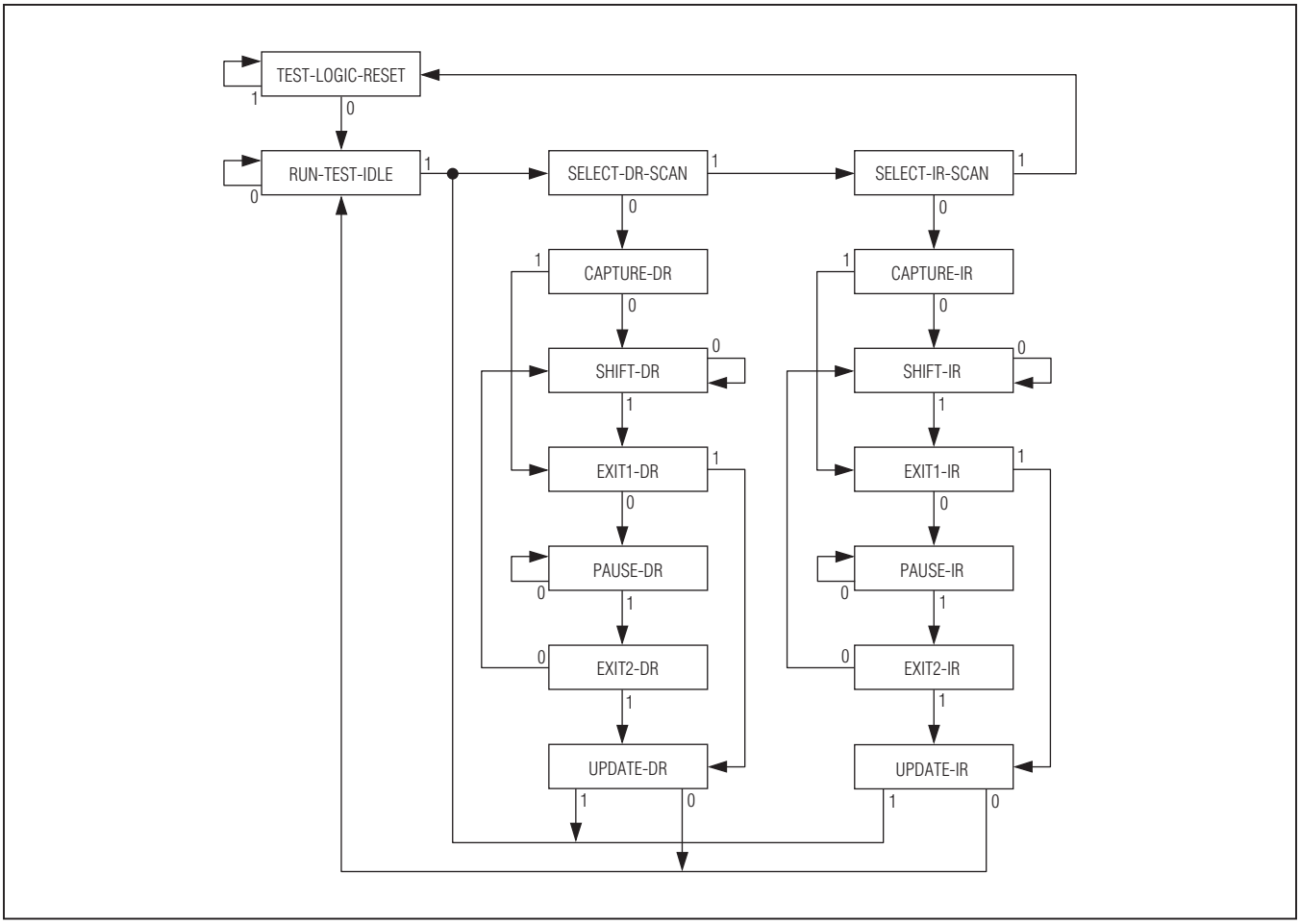


Figure 16-2. TAP Controller State Diagram

16.2 TAP State Control

The TAP provides an independent serial channel to communicate synchronously with the host system. The TAP state control is achieved through host manipulation of the test mode select (TMS) and test clock (TCK) signals. The TMS signal is sampled at the rising edge of TCK and decoded by the TAP controller to control movement between the TAP states. The TDI input and TDO output are meaningful once the TAP is in a serial shift state (i.e., Shift-IR or Shift-DR).

16.2.1 Test-Logic-Reset

On a power-on reset, the TAP controller is initialized to the Test-Logic-Reset state and the instruction register (IR[2:0]) is initialized to the Bypass instruction so that it does not affect normal system operation. No matter what the state of the controller, it enters Test-Logic-Reset when TMS is held high for at least five rising edges of TCK. The controller remains in the Test-Logic-Reset state if TMS remains high. An erroneous low signal on the TMS can cause the controller to move into the Run-Test-Idle state, but no disturbance is caused to system operation if the TMS signal is returned and kept at the intended logic-high for three rising edges of TCK, since this returns the controller to the Test-Logic-Reset state.

16.2.2 Run-Test-Idle

As illustrated in Figure 16-2, the Run-Test-Idle state is simply an intermediate state for getting to one of the two state sequences in which the controller performs meaningful operations:

- Controller state sequence (IR-Scan) or
- Data register state sequence (DR-Scan)

16.2.3 IR-Scan Sequence

The controller state sequence allows instructions (e.g., Debug and System Programming) to be shifted into the instruction register starting from the Select-IR-Scan state. In the TAP, the instruction register is connected between the TDI input and the TDO output. Inside the IR-Scan Sequence, the Capture-IR state loads a fixed binary pattern (001b) into the 3-bit shift register and the Shift-IR state causes shifting of TDI data into the shift register and serial output to TDO, least significant bit first. Once the desired instruction is in the shift register, the instruction can be latched into the parallel instruction register (IR[2:0]) on the falling edge of TCK in the Update-IR state. The contents of the 3-bit instruction shift register and parallel instruction register (IR[2:0]) are summarized with respect to the TAP controller states in Table 16-2.

Table 16-2. Instruction Register Content vs. TAP Controller State

TAP CONTROLLER STATE	INSTRUCTION SHIFT REGISTER	PARALLEL (3-BIT) INSTRUCTION REGISTER (IR[2:0])
Test-Logic-Reset	Undefined	Set to Bypass (011b) Instruction
Capture-IR	Load 001b at the rising edge of TCK	Retain last state
Shift-IR	Input data via TDI and Shift towards TDO at the rising edge of TCK	Retain last state
Exit1-IR Exit2-IR Pause-IR	Retain last state	Retain last state
Update-IR	Retain last state	Load from shift register at the falling edge of TCK
All other states	Undefined	Retain last state

When the parallel instruction register (IR2:0) is updated, the TAP controller decodes the instruction and performs any necessary operations, including activation of the data shift register to be used for the particular instruction during data register shift sequences (DR-Scan). The length of the activated shift register depends upon the value loaded to the instruction register (IR2:0). The supported instruction register encodings and associated data register selections are shown in Table 16-3.

Table 16-3. Instruction Register (IR[2:0]) Encodings

IR[2:0]	INSTRUCTION	FUNCTION	SERIAL DATA SHIFT REGISTER SELECTION
000	Extest	No operation	Unchanged, retain previous selection
001	Sample/Preload	No operation	Unchanged, retain previous selection
010	Debug	In-circuit debug mode	10-bit shift register
011	Bypass	No operation (default)	1-bit shift register
100	System Programming	Bootstrap function	3-bit shift register
101	Bypass	No operation (default)	1-bit shift register
110	Reserved	Reserved	Reserved
111	Bypass	No operation (default)	1-bit shift register

The Extest (IR[2:0] = 000b) and Sample/Preload (IR[2:0] = 001b) instructions are mandated by the JTAG standard; however, the MAX31782 does not intend to make practical use of these instructions. Hence, these instructions are treated as no-operations but can be entered into the instruction register without affecting the on-chip system logic or pins and do not change the existing serial data register selection between TDI and TDO.

The Bypass (IR[2:0] = 011b, 101b, or 111b) instruction is also mandated by the JTAG standard. The Bypass instruction is fully implemented by the MAX31782 to provide a minimum length serial data path between the TDI and the TDO pins. This is accomplished by providing a single-cell bypass shift register. When the instruction register is updated with the Bypass instruction, a single bypass register bit is connected serially between TDI and TDO in the Shift-DR state. The instruction register automatically defaults to the Bypass instruction when the TAP is in the Test-Logic-Reset state. The Bypass instruction has no affect on the operation of the on-chip system logic.

The Debug (IR[2:0] = 010b) and System Programming (IR[2:0] = 100b) instructions are private instructions that are intended solely for in-circuit debug and in-system programming operations, respectively. If the instruction register is updated with the Debug instruction, a 10-bit serial shift register is formed between the TDI and TDO pins in the Shift-DR state. If the System Programming instruction is entered into the instruction register (IR[2:0]), a 3-bit serial data shift register is formed between the TDI and TDO pins in the Shift-DR state.

Instruction register (IR[2:0]) settings other than those listed and described are reserved for internal use. As shown in Figure 16-2, the instruction register serves to select the length of the serial data register between TDI and TDO during the Shift-DR state.

16.2.4 DR-Scan Sequence

Once the instruction register has been configured to a desired state (mode), transactions are performed through a data buffer register associated with that mode. These data transactions are executed serially in a manner analogous to the process used to load the instruction register and are grouped in the TAP controller state sequence starting from the Select-DR-Scan state. In the TAP controller state sequence, the Shift-DR state allows internal data to be shifted out through the TDO pin while the external data is shifted in simultaneously through the TDI pin. Once a complete data pattern is shifted in, input data can be latched into the parallel buffer of the selected register on the falling edge of TCK in the Update-DR state. On the same TCK falling edge, in the Update-DR state, the internal parallel buffer is loaded to the data shift register for output. This Shift-DR/Update-DR process serves as the basis for passing information between the external host and the MAX31782. These data register transactions occur in the data register portion of the TAP controller state sequence diagram and have no affect on the instruction register.

16.3 Communication via TAP

The TAP controller is in Test-Logic-Reset state after a power-on reset. During this initial state, the instruction register contains the Bypass instruction and the serial path defined between the TDI and TDO pins for the Shift-DR state is the 1-bit bypass register. All TAP signals (TCK, TMS, TDI, and TDO) default to being weakly pulled high internally on any reset. The TAP controller remains in the Test-Logic-Reset state as long as TMS is held high. The TCK and TMS signals can be manipulated by the host to transition to other TAP states. The TAP controller remains in a given state whenever TCK is held low.

For the host to establish a specific data communication link, a private instruction must be loaded into the IR[2:0] register. Once the instruction is latched in the instruction parallel buffer at the Update-IR state, it is recognized by the TAP controller and the communication channel is established. In-circuit debug or in-system programming commands and data can be exchanged between the host and the MAX31782 by operating in the data register portion of the state sequence (i.e., DR-Scan). The TAP retains the private instruction that was loaded into IR[2:0] until a new instruction is shifted in, or until the TAP controller returns to the Test-Logic-Reset state.

16.3.1 TAP Communication Examples—IR-Scan and DR-Scan

Figure 16-3 and Figure 16-4 illustrate examples of communication between the host JTAG controller and the TAP of the MAX31782. The host controls the TCK and TMS signals to move through the desired TAP states while accessing the selected shift register through the TDI input and TDO output pair.

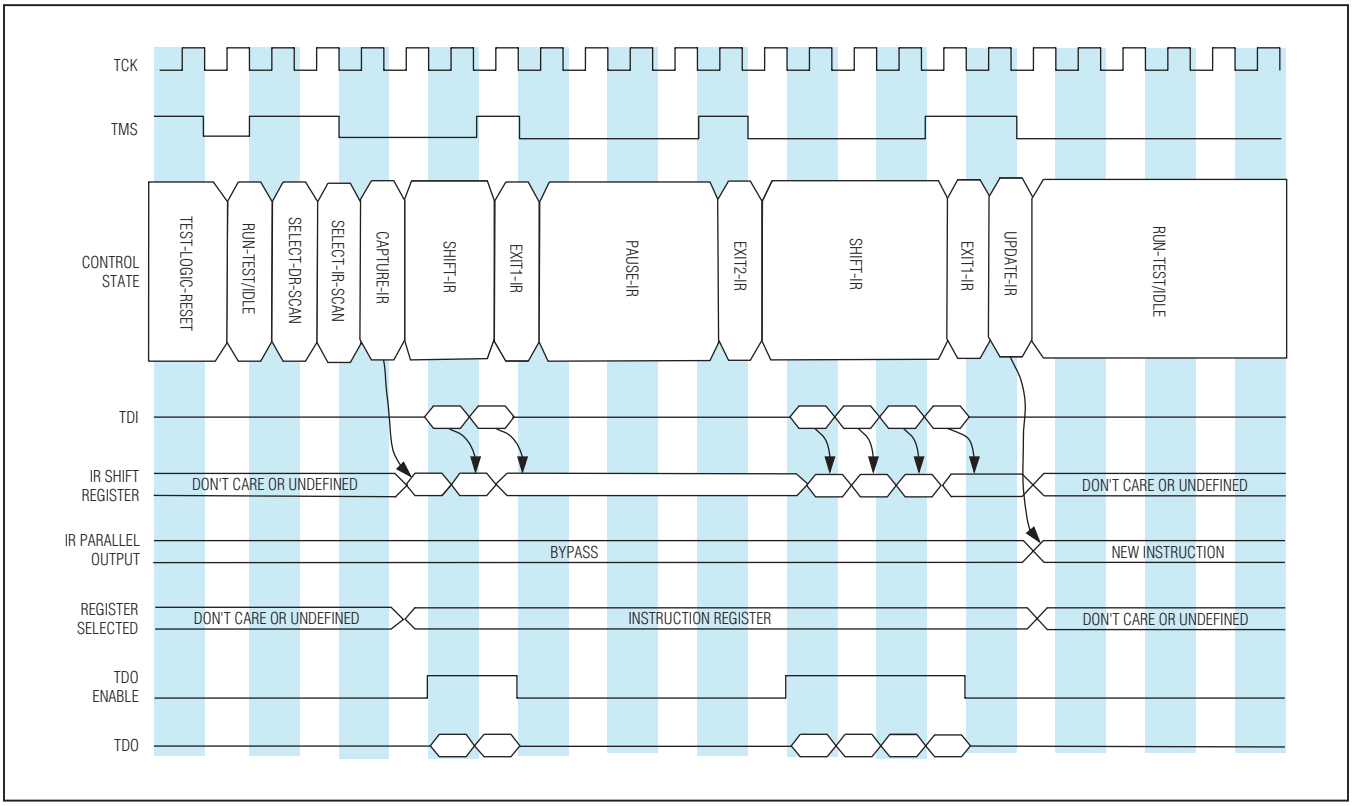


Figure 16-3. TAP Controller Debug Mode IR-Scan Example

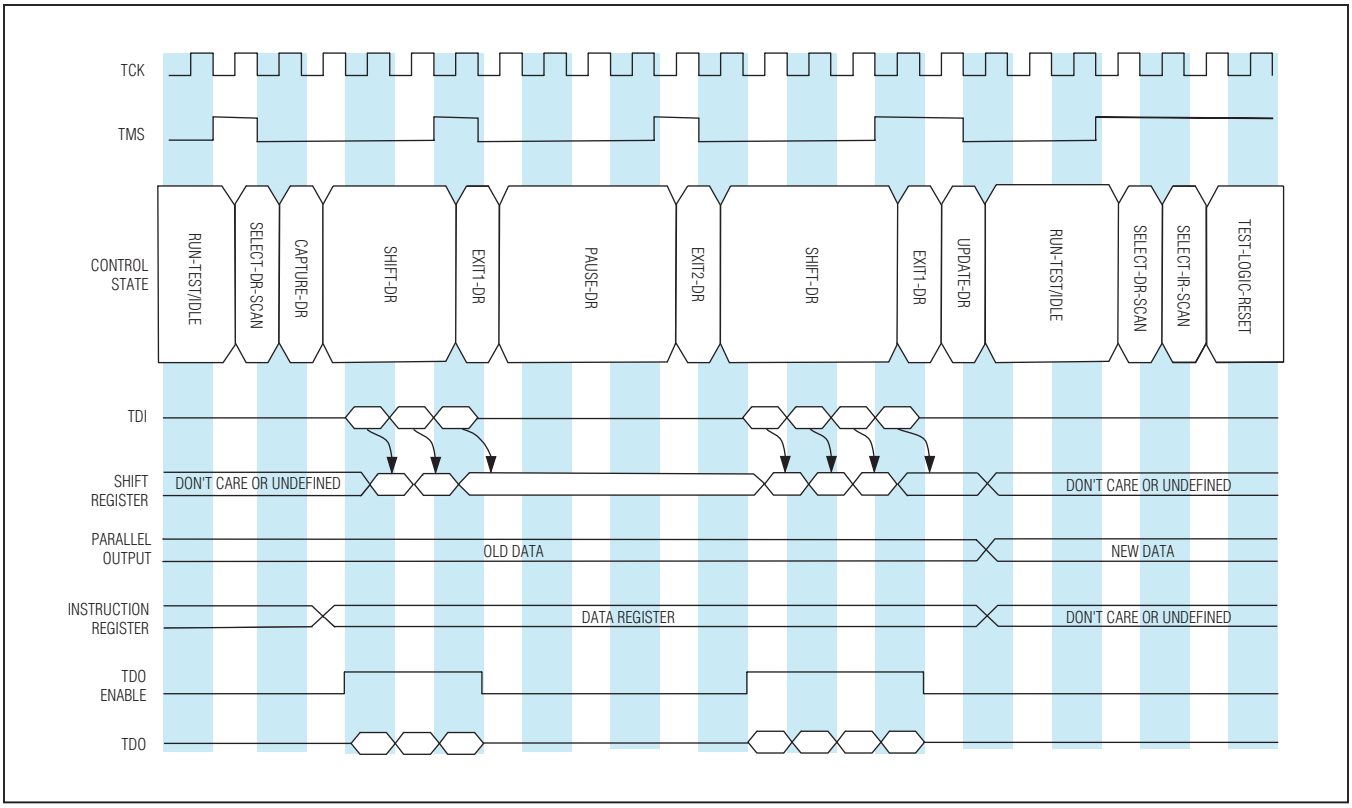


Figure 16-4. TAP Controller Debug Mode DR-Scan Example

SECTION 17: IN-CIRCUIT DEBUG MODE

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 17: IN-CIRCUIT DEBUG MODE

The MAX31782 is equipped with embedded debug hardware and embedded ROM firmware developed for the purpose of providing in-circuit debugging capability to the user application. The in-circuit debug mode uses the JTAG-compatible Test Access Port (TAP) as its means of communication between the host and the MAX31782. Figure 17-1 shows a block diagram of the in-circuit debugger. The in-circuit debug hardware and software features include:

- A debug engine
- A set of registers providing the ability to set breakpoints on register, code, or data
- A set of debug service routines stored in a ROM.

Collectively, these hardware and software features allow two basic modes of in-circuit debugging:

- Background mode allows the host to configure and set up the in-circuit debugger while the CPU continues to execute the normal program. Debug mode can be invoked from background mode.
- Debug mode allows the debug engine to take control of the CPU, providing read write access to internal registers and memory, and single-step trace operation.

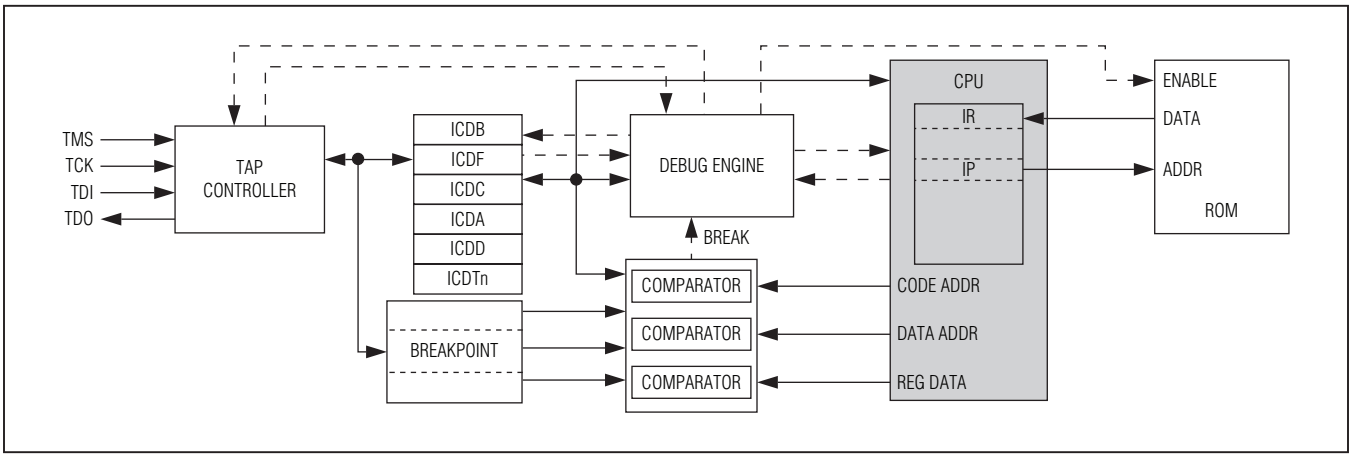


Figure 17-1. In-Circuit Debugger

The embedded hardware debug engine is implemented as a stand-alone hardware block in the MAX31782. The debug engine can be enabled for monitoring internal activities and interacting with selected internal registers while the CPU is executing user code. This capability allows the user to employ the embedded debug engine to debug the actual system, in place of the in-circuit emulator that uses external hardware to duplicate operation of the microcontroller outside of the real application environment.

To enable a communication link between the host and the microcontroller debug engine, the Debug instruction (010b) must be loaded into the TAP instruction register using the IR-Scan sequence. Once the instruction is latched in the instruction parallel buffer (IR[2:0]) and is recognized by the TAP controller in the Update-IR state, the 10-bit data shift register is activated as the communication channel for DR-Scan sequences. The TAP instruction register retains the Debug instruction until a new instruction is shifted via an IR-Scan or the TAP controller returns to the Test-Logic-Reset state.

The host now can transmit and receive serial data through the 10-bit data shift register that exists between the TDI input and TDO output during DR-Scan sequences. All background and debug mode communication (commands, data input/output, and status) occurs via this serial channel. Each 10-bit exchange of data between the host and the MAX31782 internal hardware is composed of two status bits and a single byte of command or data. The 10-bit word is always transmitted least significant bit first with the format shown in Figure 17-2. The details of the two status bits are shown in Table 17-1.

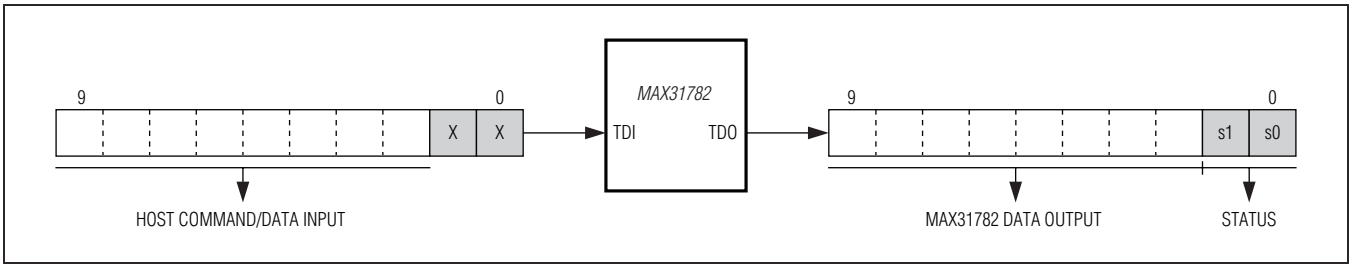


Figure 17-2. 10-Bit Word Format

Table 17-1. Status Bits

s[1:0]	STATUS/CONDITION
00	**Non-Debug.** Default condition, background mode, or debug engine inactive.
01	**Debug Idle.** Debug engine is ready to receive data from the host (command, data).
10	**Debug Busy.** Debug engine is busy without valid data (i.e., ROM code execution, trace operations).
11	**Debug Valid.** Debug engine is busy with valid data.

The data byte portion of the 10-bit shift register is interfaced directly to the ICDB parallel register. The ICDB register functions as the holding data register for both transmit and receive operations. On the falling edge of TCK in the Update-DR state, the outgoing data is loaded from the ICDB parallel register to the debug shift register and the incoming shift register data is latched in the ICDB parallel register.

17.1 Background Mode Operation

When the instruction register is loaded with the Debug instruction (IR[2:0] = 010b), the host can communicate with the MAX31782 in a background mode using TAP DR-Scan sequences without disturbing CPU operation. Note, however, that JTAG in-system programming also requires use of the 10-bit debug shift register and, if enabled (JTAG_SPE = 1, PSS[1:0] = 0), takes precedence over background mode communication. When operating in background mode, the status bits are always cleared to 00b (non-debug), which indicates that the MAX31782 is ready to receive background mode commands.

The host can perform the following operations from background mode:

- read/write internal breakpoint registers (BP0–BP5)
- read/write internal in-circuit debug registers (ICDC, ICDF, ICDA, ICDD)
- monitor to determine when a breakpoint match has occurred
- directly invoke debug mode

Table 17-2 shows the background mode commands supported by the MAX31782. Encodings not listed in this table are not supported in background mode and are treated as no operations.

Table 17-2. Background Mode Commands

OP CODE	COMMAND	OPERATION
0000–0000	No Operation	**No operation.** Default state for debug shift register.
0000–0001	Read ICDC	**Read control data from the ICDC.** The contents of the ICDC register is loaded into the debug shift register through the ICDB register for host read. This command requires one follow-on transfer cycle.
0000–0010	Read ICDF	**Read flags from the ICDF.** The contents of the ICDF register (1 byte) are loaded into the debug shift register through the ICDB register for host read. This command requires one follow-on transfer cycle.

Table 17-2. Background Mode Commands (continued)

OP CODE	COMMAND	OPERATION
0000–0011	Read ICDA	**Read data from the ICDA.** The contents of the ICDA register are loaded into the debug shift register through the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0000–0100	Read ICDD	**Read data from the ICDD.** The contents of the ICDD register are loaded into the debug shift register through the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0000–0101	Read BP0	**Read data from the BP0**.The contents of the BP0 register are loaded into the debug shift register through the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0000–0110	Read BP1	**Read data from the BP1.** The contents of the BP1 register are loaded into the debug shift register through the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0000–0111	Read BP2	**Read data from the BP2.** The contents of the BP2 register are loaded into the debug shift register through the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0000–1000	Read BP3	**Read data from the BP3.** The contents of the BP3 register are loaded into the debug shift register through the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0000–1001	Read BP4	**Read data from the BP4.** The contents of the BP4 register are loaded into the debug shift register via the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0000–1010	Read BP5	**Read data from the BP5.** The contents of the BP5 register are loaded into the debug shift register via the ICDB register for host read. This command requires two follow-on transfer cycles with the least significant byte first.
0001–0001	Write ICDC	**Write control data to the ICDC.** The contents of ICDB are loaded into the ICDC register by the debug engine at the end of the data transfer cycle.
0001–0011	Write ICDA	**Write data to the ICDA.** The contents of ICDB are loaded into the ICDA register by the debug engine at the end of the data transfer cycles. Data is transferred with the least significant byte first.
0001–0100	Write ICDD	**Write data to the ICDD.** The contents of ICDB are loaded into the ICDD register by the debug engine at the end of data transfer cycles. Data is transferred with the least significant byte first.
0001–0101	Write BP0	**Write data to the BP0.** The contents of ICDB are loaded into the BP0 register by the debug engine at the end of data transfer cycles. Data is transferred with the least significant byte first.
0001–0110	Write BP1	**Write data to the BP1.** The contents of ICDB are loaded into the BP1 register by the debug engine at the end of data transfer cycles. Data is transferred with the least significant byte first.
0001–0111	Write BP2	**Write data to the BP2.** The contents of ICDB are loaded into the BP2 register by the debug engine at the end of data transfer cycles. Data is transferred with the least significant byte first.
0001–1000	Write BP3	**Write data to the BP3.** The contents of ICDB are loaded into the BP3 register by the debug engine at the end of data transfer cycles. Data is transferred with the least significant byte first.
0001–1001	Write BP4	**Write data to the BP4.** The contents of ICDB are loaded into the BP4 register by the debug engine at the end of data transfer cycles. Data is transferred with the least significant byte first.
0001–1010	Write BP5	**Write data to the BP5.** The contents of ICDB are loaded into the BP5 register by the debug engine at the end of data transfer cycles. Data is transferred with the least significant byte first.
0001–1111	Debug	**Debug command.** This command forces the debug engine into debug mode and halts the CPU operation at the completion of the current instruction after the debug command is recognized by the debug engine.

17.1.1 Breakpoint Registers

The MAX31782 incorporates six breakpoint registers (BP0–BP5) that are configurable by the host for establishing different types of breakpoint mechanisms. The first four breakpoint registers (BP0–BP3) are 16-bit registers that are configurable as program memory address breakpoints. When enabled, the debug engine forces a break when a match between the breakpoint register and the program memory execution address occurs. The final two 16-bit breakpoint registers (BP4, BP5) are configurable in one of two possible capacities. They may be configured as data memory address breakpoints or may be configured to support register access breakpoints. In either case, if breakpoints are enabled and the defined breakpoint match occurs, the debug engine generates a break condition. The six breakpoint registers are documented below.

17.1.1.1 Breakpoint 0 Register (BP0)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	BP0.15	BP0.14	BP0.13	BP0.12	BP0.11	BP0.10	BP0.9	BP0.8	BP0.7	BP0.6	BP0.5	BP0.4	BP0.3	BP0.2	BP0.1	BP0.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

The breakpoint 0 register is accessible only through background mode read/write commands. Breakpoint registers BP0, BP1, BP2, and BP3 serve as program memory address breakpoints. When DME bit is set in background mode, the debug engine monitors the program-address bus activity while the CPU is executing the user program. If an address match is detected, a break occurs, allowing the debug engine to take control of the CPU and enter debug mode.

17.1.1.2 Breakpoint 1 Register (BP1)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	BP1.15	BP1.14	BP1.13	BP1.12	BP1.11	BP1.10	BP1.9	BP1.8	BP1.7	BP1.6	BP1.5	BP1.4	BP1.3	BP1.2	BP1.1	BP1.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

The breakpoint 1 register is accessible only via background mode read/write commands. Breakpoint registers BP0, BP1, BP2, and BP3 serve as program memory address breakpoints. When DME bit is set in background mode, the debug engine monitors the program-address bus activity while the CPU is executing the user program. If an address match is detected, a break occurs, allowing the debug engine to take control of the CPU and enter debug mode.

17.1.1.3 Breakpoint 2 Register (BP2)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	BP2.15	BP2.14	BP2.13	BP2.12	BP2.11	BP2.10	BP2.9	BP2.8	BP2.7	BP2.6	BP2.5	BP2.4	BP2.3	BP2.2	BP2.1	BP2.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

The breakpoint 2 register is accessible only via background mode read/write commands. Breakpoint registers BP0, BP1, BP2, and BP3 serve as program memory address breakpoints. When DME bit is set in background mode, the debug engine monitors the program-address bus activity while the CPU is executing the user program. If an address match is detected, a break occurs, allowing the debug engine to take control of the CPU and enter debug mode.

17.1.1.4 Breakpoint 3 Register (BP3)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	BP3.15	BP3.14	BP3.13	BP3.12	BP3.11	BP3.10	BP3.9	BP3.8	BP3.7	BP3.6	BP3.5	BP3.4	BP3.3	BP3.2	BP3.1	BP3.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

The breakpoint 3 register is accessible only via background mode read/write commands. Breakpoint registers BP0, BP1, BP2, and BP3 serve as program memory address breakpoints. When DME bit is set in background mode, the debug engine monitors the program-address bus activity while the CPU is executing the user program. If an address match is detected, a break occurs, allowing the debug engine to take control of the CPU and enter debug mode.

17.1.1.5 Breakpoint 4 Register (BP4)

The breakpoint 4 register is accessible only via background mode read/write commands.

When REGE = 0: This register serves as one of the two data memory address breakpoints. When DME is set in background mode, the debug engine monitors the data memory address bus activity while the CPU is executing the user program. If an address match is detected, a break occurs, allowing the debug engine to take over control of the CPU and enter debug mode.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	BP4.15	BP4.14	BP4.13	BP4.12	BP4.11	BP4.10	BP4.9	BP4.8	BP4.7	BP4.6	BP4.5	BP4.4	BP4.3	BP4.2	BP4.1	BP4.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

When REGE = 1: This register serves as one of the two register breakpoints. A break occurs when the destination register address for the executed instruction matches with the specified module and index. The destination module is indicated by the M[3:0] bits and the register within that module is defined by the R[4:0] bits.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	-	-	R.4	R.3	R.2	R.1	R.0	M.3	M.2	M.1	M.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

17.1.1.6 Breakpoint 5 Register (BP5)

The breakpoint 5 register is accessible only via background mode read/write commands.

When REGE = 0: This register serves as one of the two data memory address breakpoints. When DME is set in background mode, the debug engine monitors the data memory address bus activity while the CPU is executing the user program. If an address match is detected, a break occurs, allowing the debug engine to take over control of the CPU and enter debug mode.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	BP5.15	BP5.14	BP5.13	BP5.12	BP5.11	BP5.10	BP5.9	BP5.8	BP5.7	BP5.6	BP5.5	BP5.4	BP5.3	BP5.2	BP5.1	BP5.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s*	s*	s*	s*	s*	s**	s**	s**	s**

s = special

When REGE = 1: This register serves as one of the two register breakpoints. The destination module is indicated by the M[3:0] bits and the register within that module is defined by the R[4:0] bits. A break occurs when the following two conditions are met:

1) The destination register address for the executed instruction matches with the specified module and index.
2) The bit pattern written to the destination register matches those bits specified for comparison by the ICDD data register and ICDA mask register. Only those ICDD data bits with their corresponding ICDA mask bits are compared. When all bits in the ICDA register are cleared, Condition 2 becomes a don't care.

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	R.4	R.3	R.2	R.1	R.0	M.3	M.2	M.1	M.0
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

17.1.2 Using Breakpoints

All breakpoint registers (BP0–BP5) default to the FFFFh state on power-on reset or when the Test-Logic-Reset TAP state is entered. The breakpoint registers are accessible only with background mode read/write commands issued over the TAP communication link. The breakpoint registers are not read/write accessible to the CPU.

Setting the debug mode enable (DME) bit in the ICDC register to logic 1 enables all six breakpoint registers for breakpoint match comparison. The state of the break-on register enable (REGE) bit in the ICDC register determines whether the BP4 and BP5 breakpoints should be used as data memory address breakpoints (REGE = 0) or as register breakpoints (REGE = 1).

When using the register matching breakpoints, it is important to realize that Debug mode operations (e.g., read data memory, write data memory, etc.) require use of ICDA and ICDD for passing of information between the host and MAX31782 ROM routines. It is advised that these registers be saved and restored or be reconfigured before returning to the background mode if register breakpoints are to remain enabled.

When a breakpoint match occurs, the debug engine forces a break and the MAX31782 enters Debug Mode. If a breakpoint match occurs on an instruction that activates the PFX register, the break is held off until the prefixed operation completes. The host can assess whether Debug mode has been entered by monitoring the status bits of the 10-bit word shifted out of the TDO pin. The status bits change from the Non-debug (00b) state associated with background mode to the Debug-Idle (01b) state when Debug Mode is entered. Debug mode can also be manually invoked by host issuance of the 'Debug' background command.

17.2 Debug Mode

There are two ways to enter the Debug Mode from Background Mode:

1) Issuance of the Debug command directly by the host via the TAP communication port, or
2) Breakpoint matching mechanism.

The host can issue the Debug background command to the debug engine. This direct Debug Mode entry is non-deterministic. The response time varies dependent on system conditions when the command is issued. The breakpoint mechanism provides a more controllable response, but requires that the breakpoints be initially configured in Background mode. No matter the method of entry, the debug engine takes control of the CPU in the same manner. Debug mode entry is similar to the state machine flow of an interrupt except that the target execution address is x8010h which resides in the utility ROM instead of the address specified by the IV register that is used for interrupts. On debug mode entry, the following actions occur:

1) Blocks the next instruction fetch from program memory
2) Pushes the return address onto the stack
3) Sets the contents of IP to x8010h
4) Clears the IGE bit to 0 to disable interrupt handler if it is not already clear.
5) Halts CPU operation

Once in Debug mode, further breakpoint matches or host issuance of the Debug command are treated as no operations and do not disturb debug engine operation. Entering debug mode also stops the clocks to all timers, including the watchdog timer. Temporarily disabling these functions allows debug mode operations without disrupting the relationship between the original user program code and hardware timed functions. No interrupt request can be granted since the interrupt handler is also halted as a result of IGE = 0.

17.2.1 Debug Mode Commands

The debug engine sets the data shift register status bits to 01b (debug-idle) to indicate that it is ready to accept debug commands from the host.

The host can perform the following operations from debug mode:

- read register map
- read program stack
- read/write register
- read/write data memory
- single step of CPU (trace)
- return to background mode
- unlock password

The only operations directly controlled by the debug engine are single step and return. All other operations are assisted by debug service routines contained in the utility ROM. These operations require that multiple bytes be transmitted and/or received by the host, however each operation always begins with host transmission of a command byte. This command byte is decoded by the debug engine in order to determine the quantity, sequence, and destination for follow-on bytes received from the host. Even though there is no timing window specified for receiving the complete command and follow-on data, the debug engine must receive the correct number of bytes for a particular command before executing that command. If command and follow-on data are transmitted out of byte order or proper sequence, the only way to resolve this situation is to disable the debug engine by changing the instruction register (IR2:0) and reloading the Debug instruction. Once the debug engine has received the proper number of command and follow-on bytes for a given ROM assisted operation, it responds with the following actions:

- Updates the command bits (CMD[3:0]) in the ICDC register to reflect the host request
- Enables the ROM if it is not been enabled
- Forces a jump to ROM address x8010h
- Sets the data shift register status bits to 10b (debug-busy)

The ROM code performs a read to the ICDC register CMD[3:0] bits to determine its course of action. Some commands can be processed by the ROM without receiving data from the host beyond the initially supplied follow-on bytes, while others (e.g., Unlock Password) require additional data from the host. Some commands need only to provide an indication of completion to the host, while others (e.g., Read Register Map) need to supply multiple bytes of output data. To accomplish data flow control between the host and ROM, the status bits should be used by the host to assess when the ROM is ready for additional data and/or when the ROM is providing valid data output. Internally, the ROM can ascertain when new data is available or when it may output the next data byte via the TXC flag. The TXC flag is an important indicator between the debug engine and the utility ROM debug routines. The utility ROM firmware sets the TXC flag to 1 to indicate that valid data has been loaded to the ICDB register. The debug engine clears the TXC flag to 0 to indicate completion of a data shift cycle, thus allowing the ROM to continue execution of a requested task that is still in progress. The utility ROM signals that it has completed a requested task by setting the ROM Operation Done (ROD) bit of the SC register to logic 1. The ROD bit is reset by the debug engine when it recognizes the done condition.

Table 17-3 shows the debug mode commands supported by the MAX31782. Note that background mode commands are supported inside debug mode, however, the documentation of these commands can be found in the *17.1 Background Mode Operation* section of the document. Encodings not listed in this table are not supported in debug mode and are treated as no operations.

Table 17-3. Debug Mode Commands

OP CODE	COMMAND	OPERATION
0010–0000	No Operation	**No operation.**
0010–0001	Read register Map	**Read data from internal registers.** This command forces the debug engine to update the CMD[3:0] bits in the ICDC to 0001b and perform a jump to ROM code at x8010h. The ROM debug service routine loads register data to ICDB for host capture/read, starting at the lowest register location in module 0, one byte at a time in a successive order until all internal registers are read and output to the host.
0010–0010	Read data memory	**Read data from data memory.** This command requires four follow-on transfer cycles, two for the starting address and two for the word read count, starting with the LSB address and ending with the MSB read count. The input address must be based memory map when executing from utility ROM, as shown in Figure 2-4. The address is moved to the ICDA register and the word read count is moved to the ICDD register by the debug engine. This information is directly accessible by the ROM code. At the completion of this command period, the debug engine updates the CMD[3:0] bits to 0010b and performs a jump to ROM code at x8010h. The ROM debug service routine loads ICDB from data memory according to address and count information provided by the host.
0010–0011	Read program stack	**Read data from program stack.** This command requires four follow-on transfer cycles, two for the starting address and two for the read count, starting with the LSB address and ending with the MSB read count. The address is moved to the ICDA register and the read count is moved to the ICDD register by the debug engine. This information is directly accessible by the ROM code. At the completion of this command period, the debug engine updates the CMD[3:0] bits to 0011b and performs a jump to ROM code at x8010h. The ROM debug service routine pops data out from the stack according to the information received in the ICDA and ICDD register. The address input is the highest value that is used, as words are popped off the stack and returned in descending order.
0010–0100	Write register	**Write data to a selected register.** This command requires four follow-on transfer cycles, two for the register address and two for the data, starting with the LSB address and ending with the MSB data. The address is moved to the ICDA register and the data is moved to the ICDD register by the debug engine. This information is directly accessible by the ROM code. At the completion of this command period, the debug engine updates the CMD[3:0] bits to 0100b and performs a jump to ROM code at x8010h. The ROM debug service routine updates the select register according to the information received in the ICDA and ICDD registers. Any register location can be written using this command, including reserved locations and those used for op code support. No protection is provided by the debugging interface, and avoiding side effects is the responsibility of the host system communicating with the MAX31782. Writing to the IP register alters the address that execution resumes from when the debugging engine exits.
0010–0101	Write data memory	**Write data to a selected data memory location.** This command requires four follow-on transfer cycles, two for the memory address and two for the data, starting with the LSB address and ending with the MSB data. The input address must be based memory map when executing from utility ROM, as shown in Figure 2-4. The address is moved to the ICDA register and the data is moved to the ICDD register by the debug engine. This information is directly accessible by the ROM code. At the completion of this command period, the debug engine updates the CMD[3:0] bits to 0101b and performs a jump to ROM code at x8010h. The ROM debug service routine updates the selected data memory location according to the information received in the ICDA and ICDD registers.
0010–0110	Trace	**Trace command.** This command allows single stepping the CPU and requires no follow-on transfer cycle. The trace operation is a 'debug mode exit, one cycle CPU execution, debug mode entry' sequence.
0010–0111	Return	**Return command.** This command terminates the debug mode and returns the debug engine to background mode. This allows the CPU to resume its normal operation at the point where it has been last interrupted.

Table 17-3. Debug Mode Commands (continued)

OP CODE	COMMAND	OPERATION
0010–1000	Unlock password	**Unlock the password lock.** This command requires 32 follow-on transfer cycles each containing a byte value to be compared with the program memory password for the purpose of clearing the PWL bit and granting access to protected debug and loader functions. When this command is received, the debug engine updates the CMD[3:0] bit to 1000b and performs a jump to ROM code at x8010h. Data is loaded to the ICDB register when each byte of data is received, beginning with the LSB of the least significant word first and end with the MSB of the most significant word.
0010–1001	Read register	**Read from a selected internal register.** This command requires two follow-on transfer cycles, starting with the LSB address and ending with the MSB address. The address is moved to ICDA register by the debug engine. This information is directly accessible by the ROM code. At the completion of this command period, the debug engine updates the CMD[3:0] bits to 1001b and performs a jump to ROM code at x8010h. The ROM debug service routine always assumes a 16-bit register length and returns the requested data LSB first. Reading a register through the debug interface returns the value that was in that register before the debugging engine was invoked. An exception to this rule is the SP register; reading the SP register through the debug interface actually returns the value (SP+1).

17.2.2 Read Register Map Command Host-ROM Interaction

A read register map command reads out data contents for all implemented system and peripheral registers. The host does not specify a target register but instead should expect register data output in successive order, starting with the lowest order register in register module 0. Data is loaded by the ROM to the 8-bit ICDB register and is output one byte per transfer cycle. Thus, for a 16-bit register, two transfer cycles are necessary. The host initiates each transfer cycle to shift out the data bytes and finds valid data output tagged with a debug-valid (status = 11b). At the end of each transfer cycle, the debug engine clears the TXC flag to signal the ROM service routine that another byte may be loaded to ICDB. The ROM service routine sets the TXC flag each time after loading data to the ICDB register. This process is repeated until all registers have been read and output to the host. The host system recognizes the completion of the register read when the status debug-idle is presented. This indicates that the debug engine is ready for another operation.

This command outputs all peripheral registers in the range M0[00h] to M5[17h], along with a fixed set of system registers. The following formatting rules apply to the returned data:

- All peripheral registers are output as 16 bits, least significant byte first. If the register is an 8-bit register, the top is returned as 00h.
- System registers are output as 8 bits or 16 bits, least significant byte first.
- Registers I2CBUF_S, I2CBUF_M, and ADDATA are not read. Their values are returned as 0000h.
- Nonimplemented and reserved peripheral registers in the range M0[00h] to M5[17h] are represented as empty word values in Table 17-4. These values should be ignored.

The first byte output by this command is the value 184 (B8h), which represents the number of words output for peripheral register. There are a total of 216 words that are output by this command. Table 17-4 lists all of the registers output and the order in which they are output .

Table 17-4. Output from Read Register Map Command

WORD	REGISTER	WORD	REGISTER	WORD	REGISTER	WORD	REGISTER	WORD	REGISTER	WORD	REGISTER	WORD	REGISTER
0	PO2	32	----	64	----	96	PWMC0	128	PWMC2	160	MCNT	192	A[3]
1	PO1	33	I2CST_M	65	I2CST_S	97	PWMR0	129	PWMR2	161	MA	193	A[4]
2	----	34	I2CIE_M	66	I2CIE_S	98	PWMC1	130	PWMC3	162	MB	194	A[5]
3	MIIR0	35	PO6	67	MIIR2	99	PWMR1	131	PWMR3	163	MC2	195	A6[]
4	----	36	MIIR1	68	----	100	SMBUS	132	----	164	MC1	196	A[7]
5	----	37	----	69	----	101	TACHR0	133	TACHR2	165	MC0	197	A[8]
6	TB0C	38	EIF6	70	ADST	102	----	134	----	166	MC1R	198	A[9]
7	TB0R	39	EIE6	71	ADADDR	103	TACHR1	135	TACHR3	167	MC0R	199	A[10]
8	PI2	40	PI6	72	ADCN	104	PWMV0	136	PWMV2	168	PWMV4	200	A[11]
9	PI1	41	SVM	73	----	105	PWMCN0	137	PWMCN2	169	PWMCN4	201	A[12]
10	----	42	----	74	----	106	PWMV1	138	PWMV3	170	PWMC4	202	A[13]
11	TB0V	43	----	75	----	107	PWMCN1	139	PWMCN3	171	PWMR4	203	A[14]
12	----	44	I2CCN_M	76	I2CCN_S	108	TACHV0	140	TACHV2	172	TACHV4	204	A[15]
13	TB0CN	45	I2CCK_M	77	I2CCK_S	109	TACHCN0	141	TACHCN2	173	TACHCN4	205	IP
14	----	46	I2CTO_M	78	I2CTO_S	110	TACHV1	142	TACHV3	174	----	206	SP
15	----	47	I2CSLA_M	79	I2CSLA_S	111	TACHCN1	143	TACHCN3	175	TACHR4	207	IV
16	PD2	48	EIES6	80	----	112	MIIR3	144	MIIR4	176	----	208	LC[0]
17	PD1	49	----	81	----	113	----	145	----	177	TACHR5	209	LC[1]
18	----	50	PD6	82	----	114	----	146	----	178	TACHV5	210	OFFS
19	----	51	----	83	----	115	----	147	----	179	TACHCN5	211	DPC
20	----	52	----	84	----	116	----	148	----	180	PWMC5	212	GR
21	----	53	----	85	I2C_SPB	117	----	149	----	181	PWMR5	213	BP
22	----	54	ETS	86	DEV_NUM	118	----	150	----	182	PWMV5	214	DP[0]
23	----	55	ADCG1	87	----	119	----	151	----	183	PWMCN5	215	DP[1]
24	----	56	ADCG5	88	ICDT0	120	----	152	----	184	AP APC		
25	----	57	ADVOFF	89	ICDT1	121	----	153	----	185	PSF IC		
26	----	58	TOEX	90	ICDC	122	----	154	----	186	IMR SC		
27	----	59	----	91	ICDF	123	----	155	----	187	IIR CKCN		
28	----	60	----	92	ICDB	124	----	156	----	188	WDCN 00h		
29	----	61	----	93	ICDA	125	----	157	----	189	A[0]		
30	----	62	----	94	ICDD	126	----	158	----	190	A[1]		
31	----	63	----	95	----	127	----	159	----	191	A[2]		

17.2.3 Single Step Operation (Trace)

The debug engine supports single step operation in debug mode by executing a Trace command from the host. The debug engine allows the CPU to return to its normal program execution for one cycle and then forces a debug mode re-entry. The steps for the Trace command are:

1) Set status to 10b (debug-busy)
2) Pop the return address from the stack
3) Set the IGE bit to logic 1 if debug mode was activated when IGE = 1.
4) Supply the CPU with an instruction addressed by the return address
5) Stall the CPU at the end of the instruction execution
6) Block the next instruction fetch from program memory
7) Push the return address onto the stack
8) Set the contents of IP to x8010h
9) Clear the IGE bit to 0 to disable the interrupt handler
10) Halt CPU operation
11) Set the status to debug-idle

Note that the trace operation uses a return address from the stack as a legitimate address for program fetching. The host must maintain consistency of program flow during the debug process. The Instruction Pointer is automatically incremented after each trace operation, thus a new return address is pushed onto the stack before returning the control to the debug engine. Also, note that the interrupt handler is an essential part of the CPU and a pending interrupt could be granted during single-step operation since the IGE bit state present on debug mode entry is restored for the single step.

17.2.4 Return

To terminate the debug mode and return the debug engine to background mode, the host must issue a Return command to the debug engine. This command causes the following actions:

1) Pop the return address from the stack.
2) Set the IGE bit to logic 1 if debug mode was activated when IGE = 1.
3) Supply the CPU with an instruction addressed by the return address.
4) Allow the CPU to execute the normal user program.
5) Set the status to 00b (non-debug).

To prevent a possible endless breakpoint matching loop, no break occurs for a breakpoint match on the first instruction after returning from debug mode to background mode. Returning to background mode also enables all internal timer functions.

17.2.5 Debug Mode Special Considerations

The following are special considerations when using debug mode.

- Special caution should be exercised when using the Write Register command on register bits that globally affect system operation (e.g., IGE, STOP). If the write register command is used to invoke stop mode (setting STOP = 1), the $\overline{RST}$ pin may be asserted to reset the debug engine and return to the background mode of operation.
- Single stepping ('Trace') through any IGE bit change operation results in the debug engine overriding the bit change since it retains the IGE bit setting captured when active debug mode was entered.
- Single stepping ('Trace') into an operation that sets STOP = 1 when IGE = 1 effectively allows enabled interrupts normally capable of causing exit from stop mode to do so.
- Single stepping ('Trace') through any memory read instruction that reads from the utility ROM (such as 'move Acc,' @DP[0] with DP[0] set to 8000h) causes the memory read to return an incorrect value.
- Single stepping ('Trace') cannot be used when executing code from the utility ROM.
- Data memory allocation is important during system development if in-circuit debug is planned. The top 32-byte memory location may be used by the debug service routine during debug mode. The data contents in these locations may be altered and cannot be recovered.
- One available stack location is needed for debug mode. If the stack is full when entering debug mode, the oldest data in the stack is overwritten.
- Any signal sampling that relies upon the internal system clock (e.g., counter inputs) can be unreliable since the system clock is turned off inside active debug mode between debug mode commands.

17.3 In-Circuit Debug Peripheral Registers

The following peripheral registers are used to control the in-circuit debug mode of the MAX31782. Addresses of registers are given as "Mx[yy]," where x is the module number (from 0 to 5 decimal) and yy is the register index (from 00h to 1Fh hexadecimal). Fields in the bit definition tables are defined as follows:

- Name: Symbolic names of bits or bit fields in this register.
- Reset: The value of each bit in this register following a standard reset. If this field reads "unchanged," the given bit is unaffected by standard reset. If this field reads "s," the given bit does not have a fixed 0 or 1 reset value because its value is determined by another internal state or external condition.
- POR: If present this field defines the value of each bit in this register following a power-on reset (as opposed to a standard reset). Some bits are unaffected by standard resets and are set/cleared by POR only.
- Access: Bits can be read-only (r) or read/write (rw). Any special restrictions or conditions that could apply when reading or writing this bit are detailed in the bit description.

17.3.1 In-Circuit Debug Temp 0 Register (ICDT0, M2[18h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	ICDT0.15	ICDT0.14	ICDT0.13	ICDT0.12	ICDT0.11	ICDT0.10	ICDT0.9	ICDT0.8	ICDT0.7	ICDT0.6	ICDT0.5	ICDT0.4	ICDT0.3	ICDT0.2	ICDT0.1	ICDT0.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

This register is read/write accessible by the CPU only in background mode or debug mode. This register is intended for use by the utility ROM routines as temporary storage to save registers that might otherwise have to be placed in the stack.

17.3.2 In-Circuit Debug Temp 1 Register (ICDT1, M2[19h])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	ICDT1.15	ICDT1.14	ICDT1.13	ICDT1.12	ICDT1.11	ICDT1.10	ICDT1.9	ICDT1.8	ICDT1.7	ICDT1.6	ICDT1.5	ICDT1.4	ICDT1.3	ICDT1.2	ICDT1.1	ICDT1.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

s = special

This register is read/write accessible by the CPU only in background mode or debug mode. This register is intended for use by the utility ROM routines as temporary storage to save registers that might otherwise have to be placed in the stack.

17.3.3 In-Circuit Debug Control Register (ICDC, M2[1Ah])

Bit	7	6	5	4	3	2	1	0
Name	DME	-	REGE	-	CMD3	CMD2	CMD1	CMD0
Reset	0	0	0	0	0	0	0	0
Access	rs	r	rs	r	rs	rs	rs	rs

r = read, s = special

<table>
<tr><th>BIT</th><th>NAME</th><th colspan="2">DESCRIPTION</th></tr>
<tr><td>7</td><td>DME</td><td colspan="2">Debug Mode Enable (DME). When this bit is cleared to 0, background mode commands may be executed, but breakpoints are disabled. When this bit is set to 1, breakpoints are enabled while background mode commands still may be entered. This bit may only be set or cleared from background debug mode. This bit has no meaning for the ROM code.</td></tr>
<tr><td>6</td><td>Reserved</td><td colspan="2">Reserved. Do not write to this bit.</td></tr>
<tr><td>5</td><td>REGE</td><td colspan="2">Break-On Register Enable. The REGE bit is used to enable the break-on register function. When REGE bit is set to 1, BP4 and BP5 are used as register breakpoints. A break occurs when the content of BP4 is matched with the destination address of the current instruction. For BP5, a break occurs only on a selected data pattern for a selected destination register addressed by BP5. The data pattern is determined by the contents in the ICDA and ICDD register. The REGE bit alone does not enable register breakpoints, but simply changes the manner in which BP4, BP5 are used. The DME bit still must be set to a logic 1 for any breakpoint to occur. This bit has no meaning for the ROM code.</td></tr>
<tr><td>4</td><td>Reserved</td><td colspan="2">Reserved. Do not write to this bit.</td></tr>
<tr><td rowspan="11">3:0</td><td rowspan="11">CMD3:0</td><td colspan="2">These bits reflect the current host command in debug mode. These bits are set by the debug engine and allow the ROM code to determine the course of action</td></tr>
<tr><td>CMD3:0</td><td>Action</td></tr>
<tr><td>0000</td><td>No operation</td></tr>
<tr><td>0001</td><td>Read register</td></tr>
<tr><td>0010</td><td>Read data memory</td></tr>
<tr><td>0011</td><td>Read stack memory</td></tr>
<tr><td>0100</td><td>Write register</td></tr>
<tr><td>0101</td><td>Write data memory</td></tr>
<tr><td>1000</td><td>Unlock password</td></tr>
<tr><td>1001</td><td>Read selected register</td></tr>
<tr><td>Other</td><td>Reserved</td></tr>
</table>

17.3.4 In-Circuit Debug Flag Register (ICDF, M2[1Bh])

Bit	7	6	5	4	3	2	1	0
Name	-	-	-	-	PSS1	PSS0	JTAG_SPE	TXC
Reset	0	0	0	0	0	0	0	0
Access	r	r	r	r	rw	rw	rw	rw

r = read, s = special

BIT	NAME	DESCRIPTION		
7:4	Reserved	Reserved. Do not write to these bits.		
3:2	PSS[1:0]	Programming Source Select Bits [1:0]. These bits are used to select a programming interface during In-System programming when JTAG_SPE is set to 1, otherwise, the logic values of these bits have no meaning:		
		PSS1	**PSS0**	**Interface/Action**
		0	0	JTAG
		0	1	I2C
		1	x	Exit Loader
1	JTAG_SPE	System Program Enable. The JTAG_SPE bit is used for In-System programming support and its logical state, when read by the CPU, always reflects the logical-OR of the JTAG_SPE bit that is write accessible by the CPU and the SPE bit of the System Programming Buffer (SPB) Register in the TAP Module (which is accessible via JTAG). The logical state of this bit determines the program flow after a reset. When it is set to logic 1, In-System programming will be executed by the Utility ROM. When it is cleared to 0, execution will be transferred to user code. This bit allows read/write access by the CPU and is cleared to 0 only on a power-on reset or Test-Logic-Reset. The JTAG SPE bit will be cleared by hardware when the ROD bit is set. CPU writes to the JTAG_SPE bit (0 or 1) will result in clearing of the PSS[1:0] bits.		
0	TXC	Serial Transfer Complete. This bit is set by hardware at the end of a transfer cycle at the TAP communication link. The TXC bit helps the debug engine to recognize host requests, either command or data. This bit is normally set by ROM code to signify or request the sending or receiving of data. The TXC bit is cleared by the debug engine once set. CPU writes to the TXC bit results in clearing of the PSS[1:0] bits.		

17.3.5 In-Circuit Debug Buffer Register (ICDB, M2[1Ch])

Bit	7	6	5	4	3	2	1	0
Name	ICDB.7	ICDB.6	ICDB.5	ICDB.4	ICDB.3	ICDB.2	ICDB.1	ICDB.0
Reset	0	0	0	0	0	0	0	0
Access	rw	rw	rw	rw	rw	rw	rw	rw

This register serves as the parallel holding buffer for the debug shift register of the TAP. Data is read from or written to ICDB for serial communication between the debug routines and the external host.

17.3.6 In-Circuit Debug Address Register (ICDA, M2[1Dh])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	ICDA.15	ICDA.14	ICDA.13	ICDA.12	ICDA.11	ICDA.10	ICDA.9	ICDA.8	ICDA.7	ICDA.6	ICDA.5	ICDA.4	ICDA.3	ICDA.2	ICDA.1	ICDA.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

This register is used by the debug engine to store addresses so that ROM code can view that information. This register is also used by the debug engine as a mask register to mask out don't care bits in the ICDD register when BP5 is used as a register breakpoint. When a bit in this register is set to 1, the corresponding bit location in the ICDD register is compared to the data being written to the destination register to determine if a break should be generated. When a bit in this register is cleared, the corresponding bit in the ICDD register becomes a don't care and is not compared against the data being written. When all bits in this register are cleared, any updated data pattern causes a break when the BP5 register matches the destination register address of the current instruction.

17.3.7 In-Circuit Debug Data Register (ICDD, M2[1Eh])

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	ICDD.15	ICDD.14	ICDD.13	ICDD.12	ICDD.11	ICDD.10	ICDD.9	ICDD.8	ICDD.7	ICDD.6	ICDD.5	ICDD.4	ICDD.3	ICDD.2	ICDD.1	ICDD.0
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

This register is used by the debug engine to store data or read count so that ROM code can view that information. This register is also used by the debug engine as a data register for content matching when BP5 is used as a register breakpoint. In this case, only data bits in this register with their corresponding mask bits in the ICDA register set are compared with the updated destination data to determine if a break should be generated.

SECTION 18: IN-SYSTEM PROGRAMMING

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 18: IN-SYSTEM PROGRAMMING

The MAX31782 contains an internal bootstrap loader utilizing the JTAG or I2C interfaces. As a result, system software can be upgraded in-system, eliminating the need for a costly hardware retrofit when software updates are required. After each device reset, MAX31782 ROM code is executed which determines if bootloader operation is desired. Figure 18-1 provides information on how the MAX31782 enters into bootloader operation.

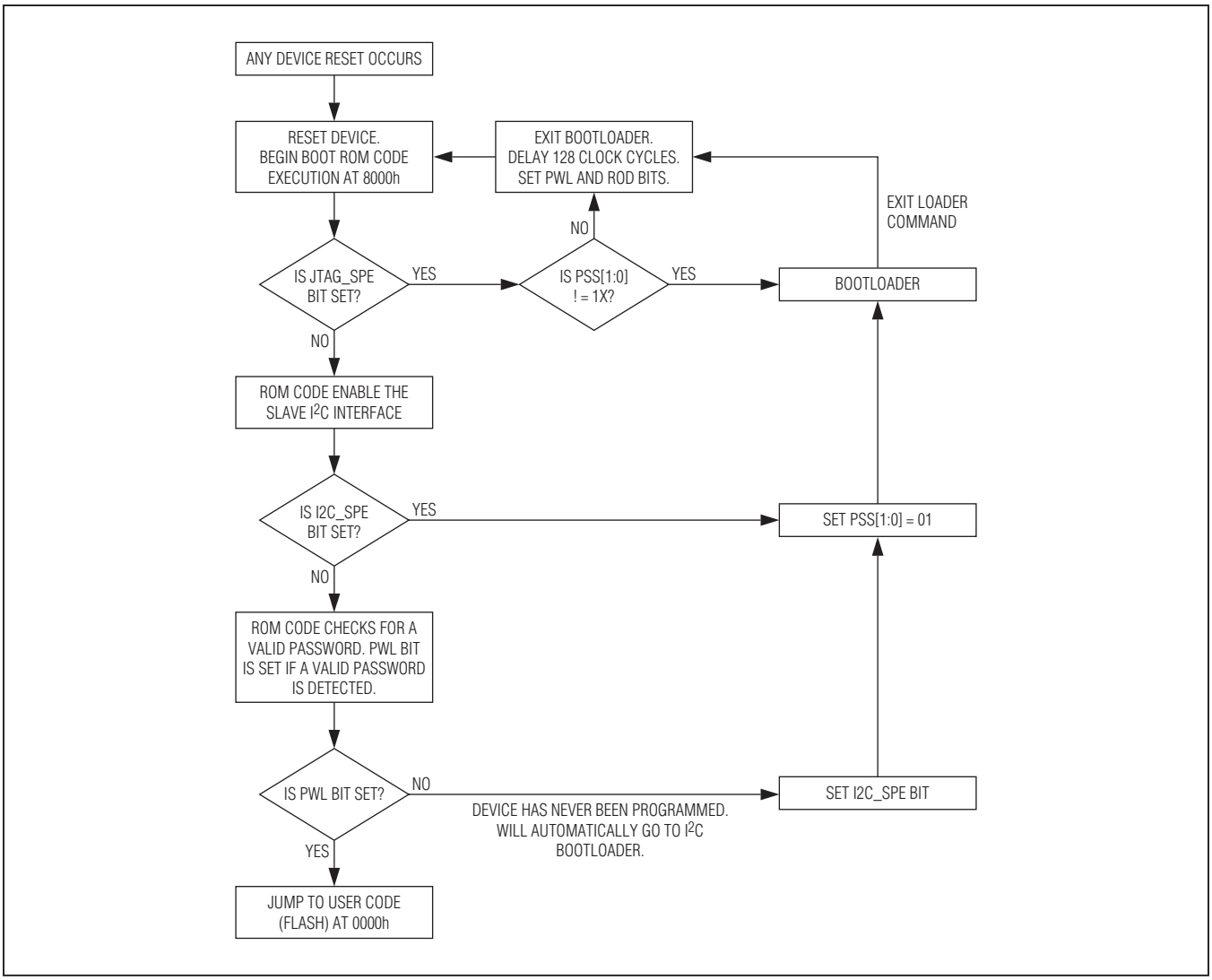


Figure 18-1. Entering Bootloader Operation

18.1 Detailed Description

Following every reset, device ROM code is executed that determines if the MAX31782 should enter into a bootloader mode. First, the ICDF register, which is not cleared by a reset, is read to see if the system programming enable (SPE) bit is set. See *18.1.2 Entering JTAG Bootloader* for more details on setting the SPE bit. If SPE is set, the MAX31782 enters into bootloader operation.

If SPE is not set, the MAX31782 then enables the slave I2C interface. The I2C_SPE bit in the I2C_SPB register is read to determine if I2C bootloader operation is desired. The I2C_SPB register is not cleared by a reset. See *18.1.3 Entering I2C Bootloader* for more details on setting the I2C_SPE bit. If I2C_SPE is set, the MAX31782 sets the PSS[1:0] bits to 01, which designates I2C bootloader, and enters bootloader operation.

If the I2C_SPE bit is not set, the ROM code then checks for a valid password in flash. See *18.1.1 Password Protection* for more details about the password. If there is not a valid password, the MAX31782 ROM code assumes that the program memory is blank and the device has never been programmed. The ROM code sets the I2C_SPE bit and the PSS[1:0] bits to 01 and then enters I2C bootloader operation. Because of this operation, it is required that all programs contain a valid password in order for the MAX31782 to enter normal operation following a reset.

If none of the preceding conditions have been met, the MAX31782 ROM code will be complete. The MAX31782 then jumps to program memory location 0000h and begin normal program execution.

18.1.1 Password Protection

The MAX31782 uses a password to protect the contents of the program memory from simple access and viewing. The password resides in the 32 bytes of program memory at byte address 0020h to 003Fh. A valid password is defined as any value that does not contain all 0000h or FFFFh. Following a reset, the password lock bit (PWL) in the SC register is set if the MAX31782 contains a valid password.

To protect the program memory, MAX31782 grants full access to in-system programming, in-application programming, or in-circuit debugging only after a password match has occurred. When a password match occurs, the PWL bit is cleared to 0. When bootloading the device, the password can be matched using the Password Match command, through either the JTAG or I2C interface.

18.1.2 Entering JTAG Bootloader

To enable the bootstrap loader and establish a desired communication channel through JTAG, the system programming instruction (100b) must be loaded into the TAP instruction register using the IR-Scan sequence. The TAP retains the System Programming instruction until a new instruction is shifted in or the TAP controller returns to the Test-Logic-Reset state. See *SECTION 16: Test Access Port (TAP)* for more information regarding the JTAG port.

Once the instruction is latched in the instruction parallel buffer (IR[2:0]) and is recognized by the TAP controller in the Update-IR state, a 3-bit data shift register is activated as the communication channel for DR-Scan sequences. This 3-bit shift register formed between the TDI and TDO pins is directly interfaced to the 3-bit serial programming buffer (SPB). Table 18-1 provides a detailed description of the system programming buffer (SPB). The data content of the SPB is reflected in the ICDF register, which allows read and write access by the CPU. These bits are cleared by power-on reset or Test-Logic-Reset of the TAP controller.

Table 18-1. System Programming Buffer (SPB)

<table>
<tr><th>BIT</th><th>NAME</th><th colspan="3">DESCRIPTION</th></tr>
<tr><td rowspan="5">2:1</td><td rowspan="5">PSS[1:0]</td><td colspan="3">Programming Source Select. These bits select the programming interface source.</td></tr>
<tr><td>PSS1</td><td>PSS0</td><td>PROGRAMMING SOURCE</td></tr>
<tr><td>0</td><td>0</td><td>JTAG</td></tr>
<tr><td>0</td><td>1</td><td>I2C</td></tr>
<tr><td>1</td><td>x</td><td>Exit loader</td></tr>
<tr><td>0</td><td>SPE</td><td colspan="3">System Programming Enable (SPE). Setting this bit to a 1 denotes that JTAG bootloading is desired upon exiting reset. The logic state of SPE is examined by the utility ROM following a reset to determine the program flow. When SPE = 1, the bootstrap loader selected by the PSS[1:0] bits is activated to perform a bootstrap loader function. If SPE = 0, the utility ROM determines if I2C bootloading is required before transferring execution control to the normal user program.</td></tr>
</table>

Following a reset, if the system programming buffer is set for JTAG bootloading, the bootload routine is entered. The host must now load the Debug instruction (010b) into the TAP instruction register (IR[2:0]), which enables the 10-bit Debug shift register between TDI and TDO. When operating in JTAG bootloader mode, the debug state machines are disabled and the sole purpose of the debug hardware is to simultaneously transfer the data byte shifted in from the host to the in-circuit debug buffer egister (ICDB) and transfer the contents of an internal holding register (loaded by ROM code writes of ICDB) into the shift register for output to the host. The 8 most significant bits of the 10-bit shift register interface directly to the ICDB. The transfer between the shift register and the ICDB register occurs on the falling edge of TCK at the Update-DR state. The debug hardware additionally clears the TXC bit in the ICDF register at this point. The ROM loader code controls the status bit output to the host by asserting TXC = 1 when it has valid data to be shifted out.

The two least significant bits of the 10-bit shift register are status bits. The JTAG bootloader has the benefit of using the same status bit handshaking hardware that is used for in-circuit debugging. The description of the status bits is described in Table 18-2.

Note: When using the JTAG port, the clock rate (TCK) must be kept below 1/8 of the system clock rate.

Table 18-2. JTAG Bootloader Status Bits

BITS 1:0	STATUS	CONDITION
00	Reserved	Invalid condition.
01	Reserved	Invalid condition.
10	Loader-Busy	ROM loader is busy executing code or processing the current command.
11	Loader-Valid	ROM loader is supplying valid output data to the host in current shift operation.

18.1.3 Entering I²C Bootloader

The MAX31782 also has built-in functionality that allows bootloading over I²C. Bootloading through I²C allows the system to update the firmware using only the I²C bus without JTAG or firmware intervention. To access the bootloading function, slave address 34h is used. This slave address is setup by hardware and cannot be changed through firmware. As long as the slave I²C port is enabled, which is the default, the MAX31782 always responds to this slave address without any firmware interaction required. This address should not be used for any purpose other than the special bootloading features. Table 18-3 details the special functions that can be performed using slave address 34h.

Table 18-3. Special Functions of Address 34h

COMMAND BYTE	ACTION
F0h	Sets the I2C_SPE bit in the I2C_SPB register to enable bootloading through I²C. This bit is not cleared on device reset.
BBh	Executes a reset of the MAX31782 when an I²C STOP is received.
All other bytes	The I2C_SPE bit in I2C_SPB is cleared. The MAX31782 NACKs this byte.

To enter the I²C bootloader, the host must first write slave address 34h with data F0h and then issue a STOP command. When the STOP command is received, the I²C _SPE bit is set. The MAX31782 must then be reset. This can be done using either the $\overline{RST}$ pin or by using the I²C self-reset. To do an I²C self-reset the host needs to write slave address 34h with data BBh. Upon receiving an I²C STOP, a reset is performed.

The I²C bootloader can also be entered if a part has never been programmed and does not contain a valid password. See *18.1.1 Password Protection* for more details about the password. Any device that does not have a password set has the I2C_SPE bit set by ROM code and enters I²C bootloader operation. The ROM code also clears the PWL bit, which allows full access to all of the bootloader commands.

18.1.4 I²C System Programming Buffer Register (I2C_SPB)

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	I2C_SPE
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Access	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	rw

BIT	NAME	DESCRIPTION
15:1	—	Reserved. The user should not write to these bits.
0	I2C_SPE	Setting this bit to a 1, by writing to slave address 34h, denotes that I²C bootloading is desired upon exiting reset. The logic state of I2C_SPE is examined by the utility ROM following a reset to determine the program flow. When I2C_SPE = 1, the I²C bootstrap loader is activated to perform a bootstrap loader function. Also, if the MAX31782 does not contain a valid password, this bit is set during reset, followed by entry into the I²C bootstrap loader.

18.2 Bootloader Operation

Once in bootloader mode, the JTAG and I²C interfaces both use the same commands. How these commands are implemented will be different between the two interfaces. Table 18-4 shows an example command and parameters. The next two sections detail how to implement these commands using either the JTAG or I²C interface.

Table 18-4. Example Bootload Command

Byte(s)	Command	Data In	NOP	Data Out	Return	Dummy RX
Input	Command	Data In	00h	00h	00h	00h
Output	X	X	X	Data Out	3Eh	X

BYTE NAME	DESCRIPTION
Command	All bootloader commands begin with a single command byte. The upper four bits of this command byte define the command family (from 0 to 15) and the lower four bits define the specific command within that family.
Data In	Data bytes that are input to the bootloader that are required for the command. The number of Data In bytes varies for each command. Some commands do not require any Data In bytes.
NOP	The NOP byte is only used for JTAG mode. This is a byte of 00h that is clocked into TDI, while TDO is ignored.
Data Out	Data Out is any data that is returned by the bootloader. The number of Data Out bytes varies for each command. Some commands do not output any Data Out bytes.
Return	A return value of 3Eh is output by the bootloader following the successful completion of a command. If the Return byte is read prior to 3Eh being loaded by the bootloader, the read will return the data that is currently in the shift register. The value 3Eh is only loaded into the shift register once. Any subsequent reads will return invalid data. In JTAG bootload mode, status bits will tell when ROM loader is sending valid 3Eh.
Dummy RX	The Dummy RX byte is only required for I²C mode. This is a dummy read of one byte, followed by a NACK.

18.2.1 JTAG Bootloader Protocol

The JTAG port consists of a shift register. As data is clocked into TDI, data is clocked out of TDO. Each "byte" on the JTAG port is actually 10 bits. The two least significant bits are the status bits described in Table 18-2. The data that is input to the device on the TDI pin should have the two status bits set to 0. The following steps are required for each command.

1) Transmit the Command byte on TDI. Ignore the returned data on TDO.
2) Transmit any Data In bytes on TDI. Ignore the returned data on TDO.
3) Transmit the NOP byte of 00h, on TDI. Ignore the returned data on TDO.

4) Possibly poll returned data until command execution completes.
5) Transmit 00h on TDI for each Data Out byte. Read the Data Out byte on TDO.
6) Transmit 00h on TDI and verify that the Return byte output on TDO is 3Eh.
7) The Dummy RX byte is not required for the JTAG bootloader to operate.

Some of the bootloader commands, such as the erase and CRC commands require extra time to execute. For these commands, the two status bits can be used to verify the state of the bootloader. After issuing any of these commands, the NOP command can continuously be sent to the bootloader. If the returned status bits are 10, the bootloader is still busy processing the command. If the status bits are 11, the bootloader has completed execution of the command. The first byte that was returned with status bits 11 is the first byte of valid returned data from the bootloader.

18.2.2 I²C Bootloader Protocol

After entering the I²C bootloader, all I²C communication takes place on the default I²C slave address 36h. When writing data to the MAX31782, slave address 36h (R/$\overline{W}$ bit = 0) is used. To read data from the MAX31782 I²C bootloader, slave address 37h (R/$\overline{W}$ bit = 1) is used. The I²C bootloader does not return the status bits that are available from the JTAG bootloader. The following I²C steps are required to send each command

1) Send an I²C START, followed by writing slave address 36h (R/$\overline{W}$ bit set to write).
2) Write command byte.
3) Write any Data In bytes.
4) The NOP byte is not required for the I²C interface. Sending a NOP byte when using the I²C bootloader places the bootloader into an unknown state. Instead, an I²C restart needs to be issued, followed by writing slave address 37h (R/$\overline{W}$ bit set to read).
5) Possibly poll returned data until command execution completes.
6) Read and ACK all Data Out bytes.
7) Read and ACK the Return byte, verify that 3Eh was returned.
8) Read and NACK the Dummy RX byte. Ignore the returned data.
9) Send an I²C STOP.

Some of the bootloader commands such as the erase and CRC commands require extra time to execute. For these commands, the I²C port can be continuously polled to determine when the command completes. This polling is done by reading the returned data bytes after sending slave address 37h. The I²C bootloader returns data B7h while it is currently busy. When data other than B7h is returned, the bootloader is returning valid data. An example of polling for the CRC Code command is shown in Figure 18-2. After sending slave address 37h, the I²C bootloader outputs B7h until the command has finished execution. The I²C master needs to continue reading and returning ACK's until a string of four bytes with values B7h, YYh, ZZh, 3Eh is returned. The master then reads the Dummy RX byte and NACKs this byte.

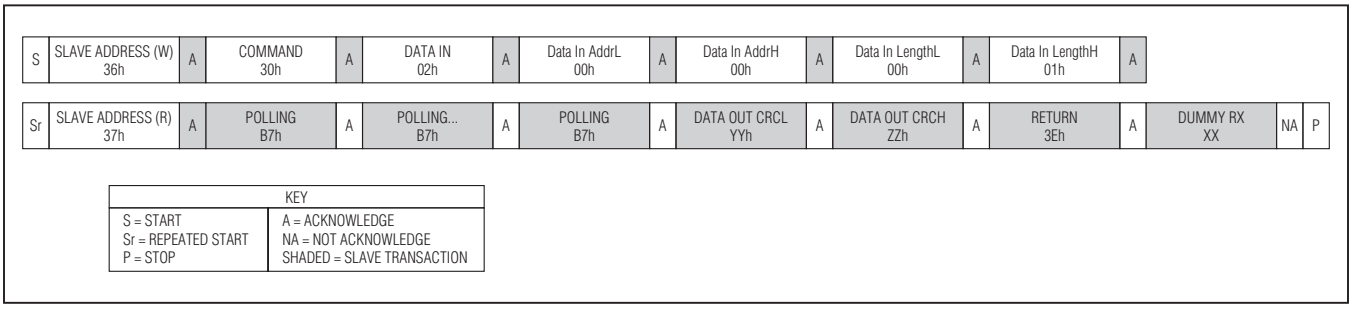


Figure 18-2. I²C Bootloader Polling

18.3 Bootloader Commands

Commands for the MAX31782 loader are grouped into families. All bootloader commands begin with a single command byte. The upper 4 bits of this command byte define the command family (from 0 to 15), while the lower 4 bits define the specific command within that family. The loader command families are shown in Table 18-5.

Table 18-5. Command Families

COMMAND FAMILY	FAMILY DESCRIPTION
0	Required
1	Load
2	Dump
3	CRC
4	Verify
5	Load and Verify
E	Fixed Length Erase

All commands, except those in Family 0, are password protected. The password must first be matched before these commands can be executed. This is done using the Password Match command, which clears the PWL bit if a match is made.

Bootloader commands that fail for any reason set the bootloader status byte to an error code value describing the reason for the failure. This status byte can be read by means of the Get Status command.

For proper bootloader operation, all bytes of data listed for the command must be written or read from the bootloader. This includes the Return byte, and for the I^2C bootloader, the Dummy RX byte. If all bytes are not read, the bootloader remains in an unknown state even after a new command is sent to the bootloader.

Following are descriptions of the bootloader commands that are available for use by the MAX31782 bootloader.

18.3.1 Command 00h—No Operation

	Byte 1
	Command
Input	00h
Output	X

This is a No Operation command. This command can be sent at any time without the bootloader taking action. This command is not password protected.

18.3.2 Command 01h—Exit Loader

	Byte 1
	Command
Input	01h
Output	X

This command causes the bootloader to exit. When exiting, the bootloader clears the JTAG_SPE and I2C_SPE bits and then performs an internal reset of the device. Following the reset, code execution jumps to the beginning of application code at address 0000h. This command is not password protected.

18.3.3 Command 02h—Master Erase

	Byte 1	Byte 2	Byte 3	Byte 4
	Command	**NOP**	**Return**	**Dummy RX**
Input	02h	00h	00h	00h
Output	X	X	3Eh	X

This command erases (sets to FFFFh) all words in the program flash memory and writes all words in the data SRAM to zero. This command is not password protected. After this command completes, the password lock bit is automatically cleared, allowing access to all bootloader commands. This command requires approximately 40 ms to complete. Polling for a return value of 3Eh can be performed during this execution time to determine when the master erase has completed.

18.3.4 Command 03h—Password Match

	Byte 1	Bytes 2 to 33	Byte 34	Byte 35	Byte 36
	Command	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	03h	32-Byte Password	00h	00h	00h
Output	X	X	X	3Eh	X

This command accepts a 32-byte password value, which is matched against the password in program memory from byte address 0020h through 003Fh. If the entered value matches the password in program memory, the password lock bit is cleared. This command is not password protected.

18.3.5 Command 04h—Get Status

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
	Command	**NOP**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	04h	00h	00h	00h	00h	00h
Output	X	X	Flags	Status Code	3Eh	X

The Status Flags and Status Code returned by the Get Status command are defined in Table 18-6 and Table 18-7. This command is not password protected. The Status Codes are set whenever an error condition occurs and only reflect the last error. The Status Codes are cleared:

- When the bootloader is initially entered.
- At the start of execution of all commands except Familiy 0 commands.
- At the start of execution of the Family 0 Master Erase and Set Access Mode commands.

Table 18-6. Bootloader Status Flags

FLAG BIT	MEANING
8:3	**Reserved.**
2	**Word/Byte Mode Supported.** 0 – The bootloader supports byte mode only. 1 – The bootloader supports word mode as well as byte mode. **(Note: the MAX31782 supports byte mode only)**
1	**Word/Byte Mode.** 0 – The bootloader is currently in byte mode for memory reads/writes. 1 – The bootloader is currently in word mode for memory reads/writes. **(Note: the MAX31782 supports byte mode only)**
0	**Password Lock.** This bit will match the SC.PWL bit. 0 – The password is unlocked or had a default value; password-protected commands may be used. 1 – The password is locked. Password-protected commands may not be used.

Table 18-7. Bootloader Status Codes

STATUS VALUE	MEANING
00	**No Error.** The last command completed successfully.
01	**Family Not Supported.** An attempt was made to use a command from a family which the bootloader does not support.
02	**Invalid Command.** An attempt was made to use a nonexistent command within a supported command family.
03	**No Password Match.** An attempt was made to use a password-protected command without first matching a valid password. Or, the Password Match command was called with an incorrect password value.
04	**Bad Parameter.** An input parameter passed to the command was out of range or otherwise invalid.
05	**Verify Failed.** The verification step failed on a Load/Verify or Verify command.
06	**Unknown Register.** An attempt was made to read from or write to a nonexistent register.
07	**Word Mode Not Supported.** An attempt was made to set word mode access, but the bootloader supports byte mode access only.
08	**Master Erase Failed.** The bootloader was unable to perform master erase.

18.3.6 Command 05h—Get Supported Commands

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
	Command	**NOP**	**Data Out**	**Data Out**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	05h	00h	00h	00h	00h	00h	00h	00h
Output	X	X	SupportL	SupportH	00h	00h	3Eh	X

The SupportL (LSB) and SupportH (MSB) bytes form a 16-bit value that indicates which command families the bootloader supports. If bit 0 is set to 1, it indicates that Family 0 is supported. If bit 1 is set to 1, it indicates that Family 1 is supported. The value returned by the MAX31782 is 403Fh, indicating that command families 0, 1, 2, 3, 4, 5 and E are supported. This command is not password protected.

18.3.7 Command 06h—Get Code Size

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
	Command	**NOP**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	06h	00h	00h	00h	00h	00h
Output	X	X	SizeL	SizeH	3Eh	X

This command returns SizeH:SizeL, which represents the size of available code memory in words minus 1. The MAX31782 returns a value of 7FFFh, which indicates 32kWords of program memory are available. This command is not password protected.

18.3.8 Command 07h—Get Data Size

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
	Command	**NOP**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	07h	00h	00h	00h	00h	00h
Output	X	X	SizeL	SizeH	3Eh	X

This command returns SizeH:SizeL, which represents the size of available data memory in words minus 1. The MAX31782 returns a value of 03FFh, which indicates 1kWords of data memory are available. This command is not password protected.

18.3.9 Command 08h—Get Loader Version

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
	Command	**NOP**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	08h	00h	00h	00h	00h	00h
Output	X	X	VersionL	VersionH	3Eh	X

This command returns the device's bootloader version. The format of the version is VersionH.VersionL. For example, if VersionL returns 00h and VersionH returns 01h, this corresponds to bootloader version 1.0. This command is not password protected.

18.3.10 Command 09h—Get Utility ROM Version

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
	Command	**NOP**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	09h	00h	00h	00h	00h	00h
Output	X	X	VersionL	VersionH	3Eh	X

This command returns the device's ROM code version. The format of the ROM version is VersionH.VersionL. For example, if VersionL returns 00h and VersionH returns 01h, this corresponds to ROM version 1.0. This command is not password protected.

18.3.11 Command 0Eh—Get Device Number

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
	Command	**NOP**	**Data Out**	**Return**	**Dummy RX**
Input	0Eh	00h	00h	00h	00h
Output	X	X	DEV_NUM	3Eh	X

This command returns the value that is stored in the DEV_NUM register. This command is not password protected.

18.3.12 Command 10h—Load Code

	Byte 1	Byte 2	Byte 3	Byte 4	(Length) Bytes	Byte Length+5	Byte Length+6	Byte Length+7
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	10h	Length	AddressL	AddressH	Data to load	00h	00h	00h
Output	X	X	X	X	X	X	3Eh	X

This command programs (Length) bytes of data into the program flash starting at byte address (AddressH:AddressL). The bootloader writes one 16-bit word to flash at a time. The low bit of the address is always forced to zero because instructions in program flash are word aligned. If an odd number of bytes are input, the final word written to the program flash has its most significant byte set to 00h. Memory locations in flash that have been previously loaded must be erased (Master Erase or Page Erase command) before they can be loaded with a new value. The MAX31782 uses a little-endian memory architecture where the least significant byte of each word is loaded first. For example, if you load bytes (11h, 22h, 33h, 44h) starting at address 0000h, the first two words of program space are written to 2211h, 4433h. This command is password protected.

The time required to write one word of data to flash is approximately 80µs. To guarantee correct programming, a bootloading program needs to ensure that there is at least 100µs of time between when the bootloader receives two words of data. The easiest way to do this is to limit the clock rate to 100kHz. The time to transmit one word of data with a 100kHz clock exceeds 100µs, thus giving the previously transmit word time to be programmed into flash prior to processing the next word. If a faster clock rate is used, delays need to be added to ensure that words are not transmit at rates faster than 100µs.

The JTAG bootloader also supports polling using the status bits as a method to determine when a word has successfully been written into flash. When sending the first two bytes of program data to load, the status bits should return as 11 to signify that the bootloader is valid. After sending the second byte, the bootloader begins writing this first word to flash and is busy. If a third byte of data is written while the bootloader is busy programming the first word, the status bits return as 10, which is loader busy. Upon receiving a status of 10, the third byte needs to be sent again until the status bits return as 11, or loader valid. When this code is returned the third byte has been received and the fourth byte can now be sent. If using the JTAG bootloader with a clock faster than 100kHz, this polling method should be used for every byte that is transmit to the bootloader.

18.3.13 Command 11h—Load Data

	Byte 1	Byte 2	Byte 3	Byte 4	(Length) Bytes	Byte Length+5	Byte Length+6	Byte Length+7
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	11h	Length	AddressL	AddressH	Data to load	00h	00h	00h
Output	X	X	X	X	X	X	3Eh	X

This command writes (Length) bytes of data into the data SRAM starting at byte address (AddressH:AddressL). The MAX31782 uses a little-endian memory architecture where the least significant byte of each word is loaded first. For example, if you load bytes (11h, 22h, 33h, 44h) starting at address 0000h, the first two words of memory space are written to 2211h, 4433h. This command is password protected.

18.3.14 Command 20h—Dump Code

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 5	Byte 6	Length Bytes	Byte Length+7	Byte Length+8
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Data Out**	**Return**	**Dummy RX**
Input	20h	2	AddrL	AddrH	LengthL	LengthH	00h	00h	00h	00h
Output	X	X	X	X	X	X	X	Memory	3Eh	X

This command returns the contents of the program flash memory. The memory dump begins at byte address AddrH:AddrL and contains LengthH:LengthL bytes. This command is password protected.

18.3.15 Command 21h—Dump Data

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 5	Byte 6	Length Bytes	Byte Length+7	Byte Length+8
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Data Out**	**Return**	**Dummy RX**
Input	21h	2	AddrL	AddrH	LengthL	LengthH	00h	00h	00h	00h
Output	X	X	X	X	X	X	X	Memory	3Eh	X

This command returns the contents of the SRAM memory. The memory dump begins at byte address AddrH:AddrL and contains LengthH:LengthL bytes. This command is password protected.

18.3.16 Command 30h—CRC Code

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	30h	2	AddrL	AddrH	LengthL	LengthH	00h	00h	00h	00h	00h
Output	X	X	X	X	X	X	X	CRCL	CRCH	3Eh	X

This command returns the CRC-16 value (CRCH:CRCL) of the (LengthH:LengthL) bytes of program flash starting at (AddrH:AddrL). The formula for the CRC calculation is $X^{16} + X^{15} + X^2 + 1$. This command is password protected.

The CRC calculation takes approximately 45 system clock cycles per byte. During this time polling should be performed to determine when the loader has finished executing the CRC calculation. If using the I2C loader, use the polling method shown in Figure 18-2. When a data string is read that has B7, CRCL, CRCH, 3Eh, the host knows that the calculation completed successfully. If using the JTAG loader, the JTAG status bits can be used to determine when the CRC calculation is complete.

18.3.17 Command 31h—CRC Data

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Data Out**	**Data Out**	**Return**	**Dummy RX**
Input	31h	2	AddrL	AddrH	LengthL	LengthH	00h	00h	00h	00h	00h
Output	X	X	X	X	X	X	X	CRCL	CRCH	3Eh	X

This command returns the CRC-16 value (CRCH:CRCL) of the (LengthH:LengthL) bytes of data memory starting at (AddrH:AddrL). The formula for the CRC calculation is $X^{16} + X^{15} + X^2 + 1$. This command is password protected.

The CRC calculation takes approximately 45 system clock cycles per byte. During this time polling should be performed to determine when the loader has finished executing the CRC calculation. If using the I2C loader, use the polling method shown in Figure 18-2. When a data string is read that has B7, CRCL, CRCH, 3Eh, the host knows that the calculation completed successfully. If using the JTAG loader, the JTAG status bits can be used to determine when the CRC calculation is complete.

18.3.18 Command 40h—Verify Code

	Byte 1	Byte 2	Byte 3	Byte 4	(Length) Bytes	Byte Length+5	Byte Length+6	Byte Length+7
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	40h	Length	AddrL	AddrH	Data to Verify	00h	00h	00h
Output	X	X	X	X	X	X	3Eh	X

This command operates in the same manner as the Load Code command, except that instead of programming the input data into flash memory, it verifies that the input data matches the data already in code space. If the data does not match, the status code is set to reflect this failure. This command is password protected.

18.3.19 Command 41h—Verify Data

	Byte 1	Byte 2	Byte 3	Byte 4	(Length) Bytes	Byte Length+5	Byte Length+6	Byte Length+7
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	41h	Length	AddrL	AddrH	Data to Verify	00h	00h	00h
Output	X	X	X	X	X	X	3Eh	X

This command operates in the same manner as the Load Data command, except that instead of writing the input data into SRAM, it verifies that the input data matches the data already in data space. If the data does not match, the status code is set to reflect this failure. This command is password protected.

18.3.20 Command 50h—Load and Verify Code

	Byte 1	Byte 2	Byte 3	Byte 4	(Length) Bytes	Byte Length+5	Byte Length+6	Byte Length+7
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	50h	Length	AddrL	AddrH	Data to load and verify	00h	00h	00h
Output	X	X	X	X	X	X	3Eh	X

This command provides the combined functionality of the Load Code and Verify Code commands. After each word of data is written to flash memory, the loader reads this memory location and verifies that the data matches the input data. If the verification fails, the status code is set to reflect this failure. All the guidelines that are listed for the Load Code command must be followed for the Load and Verify Code command. This command is password protected.

18.3.21 Command 51h—Load and Verify Data

	Byte 1	Byte 2	Byte 3	Byte 4	(Length) Bytes	Byte Length+5	Byte Length+6	Byte Length+7
	Command	**Data In**	**Data In**	**Data In**	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	51h	Length	AddrL	AddrH	Data to load and verify	00h	00h	00h
Output	X	X	X	X	X	X	3Eh	X

This command provides the combined functionality of the Load Data and Verify Data commands. After each word of data is written to SRAM memory, the loader reads this memory location and verifies that the data matches the input data. If the verification fails, the status code is set to reflect this failure. The guidelines that are listed for the Load Data command must be followed for the Load and Verify Data command. This command is password protected.

18.3.22 Command E0h—Code Page Erase

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
	Command	**Data In**	**Data In**	**Data In**	**NOP**	**Return**	**Dummy RX**
Input	E0h	0	PageNum	0	00h	00h	00h
Output	X	X	X	X	X	3Eh	X

This command erases (programs to FFFFh) all words in a 256 word (512 byte) page of the program flash memory. The MAX31782 has 128 pages of flash. The input PageNum indicates which page to erase. For example, PageNum = 1 would erase byte addresses 000h through 1FFh and PageNum = 2 would erase byte addresses 200h through 3FFh. This command requires approximately 40ms to complete. Polling can be performed during this execution time to determine when the page erase has completed. This command is password protected.

SECTION 19: PROGRAMMING

This section contains the following information:

LIST OF TABLES

SECTION 19: PROGRAMMING

The following section provides a programming overview of the MAX31782. For full details on the instruction set, as well as System Register and Peripheral Register detailed bit descriptions, see the appropriate sections in this user's guide.

19.1 Addressing Modes

The instruction set for the MAX31782 provides three different addressing modes: direct, indirect, and immediate. The direct addressing mode can be used to specify either source or destination registers, such as:

```
move      A[0], A[1]   ; copy accumulator 1 to accumulator 0
push      A[0]         ; push accumulator 0 on the stack
add       A[1]         ; add accumulator 1 to the active accumulator
```

Direct addressing is also used to specify addressable bits within registers.

```
move      C, Acc.0     ; copy bit zero of the active accumulator
                       ; to the carry flag
move      PO0.3, #1    ; set bit three of port 0 Output register
```

Indirect addressing, in which a register contains a source or destination address, is used only in a few cases.

```
move      @DP[0], A[0] ; copy accumulator 0 to the data memory
                       ; location pointed to by data pointer 0
move      A[0], @SP--  ; where @SP-- is used to pop the data pointed to
                       ; by the stack pointer register
```

Immediate addressing is used to provide values to be directly loaded into registers or used as operands.

```
move      A[0], #10h   ; set accumulator 1 to 10h/16d
```

19.2 Prefixing Operations

All instructions on the MAX31782 are 16 bits long and execute in a single cycle. However, some operations require more data than can be specified in a single cycle or require that high-order register-index bits be set to achieve the desired transfer. In these cases, the pre- fix register module PFX is loaded with temporary data and/or required register index bits to be used by the following instruction. The PFX module only holds loaded data for a single cycle before it clears to zero.

Instruction prefixing is required for the following operations, which effectively makes them two-cycle operations:

- When providing a 16-bit immediate value for an operation (e.g., loading a 16-bit register, ALU operation, supplying an absolute pro- gram branch destination), the PFX module must be loaded in the previous cycle with the high byte of the 16-bit immediate value unless that high byte is zero. One exception to this rule is when supplying an absolute branch destination to 00xxh. In this case, PFX still must be written with 00h. Otherwise, the branch instruction would be considered a relative one instead of the desired absolute branch.
- When selecting registers with indexes greater than 07h within a module as destinations for a transfer or registers with indexes greater than 0Fh within a module as sources, the PFX[n] register must be loaded in the previous cycle. This can be combined with the previous item.

Generally, prefixing operations can be inserted automatically by the assembler as needed, so that (for example)

```
move      DP[0], #1234h
```

actually assembles as

```
move      PFX[0], #12h
move      DP[0], #34h
```

However, the operation

```
move      DP[0], #0055h
```

does not require a prefixing operation even though the register DP[0] is 16-bit. This is because the prefix value defaults to zero, so the line

```
move      PFX[0], #00h
```

is not required.

19.3 Reading and Writing Registers

All functions in the MAX31782 are accessed through registers, either directly or indirectly. This section discusses loading registers with immediate values and transferring values between registers of the same size and different sizes.

19.3.1 Loading an 8-Bit Register with an Immediate Value

Any writable 8-bit register with a sub-index from 0h to 7h within its module can be loaded with an immediate value in a single cycle using the MOVE instruction.

```
move      AP, #05h     ; load accumulator pointer register with 5 hex
```

Writable 8-bit registers with sub-indexes 8h and higher can be loaded with an immediate value using MOVE as well, but an additional cycle is required to set the prefix value for the destination.

```
move      WDCN, #33h   ; assembles to:    move PFX[2], #00h
                       ;                  move (WDCN-80h), #33h
```

19.3.2 Loading a 16-Bit Register with a 16-Bit Immediate Value

Any writable 16-bit register with a sub-index from 0h to 07h can be loaded with an immediate value in a single cycle if the high byte of that immediate value is zero.

```
move      LC[0], #0010h; prefix defaults to zero for high byte
```

If the high byte of that immediate value is not zero or if the 16-bit destination sub-index is greater than 7h, an extra cycle is required to load the prefix value for the high byte and/or the high-order register index bits.

```
                       ; high byte <> #00h
move      LC[0], #0110h; assembles to:    move PFX[0], #01h
                       ;                  move LC[0], #10h
                       ; destination sub-index > 7h
move      A[8], #0034h ; assembles to:    move PFX[2], #00h
                       ;                  move (A[8]-80h), #34h
```

19.3.3 Moving Values Between Registers of the Same Size

Moving data between same-size registers can be done in a single-cycle MOVE if the destination register's index is from 0h to 7h and the source register index is between 0h and Fh.

```
move      A[0], A[8]   ; copy accumulator 8 to accumulator 0
move      LC[0], LC[1] ; copy loop counter 1 to loop counter 0
```

If the destination register's index is greater than 7h or if the source register index is greater than Fh, prefixing is required.

```
move      A[15], A[0]  ; assembles to:    move PFX[2], #00h
                       ;                  move (A[15]-80h), A[0]
```

19.3.4 Moving Values Between Registers of Different Sizes

Before covering some transfer scenarios that might arise, a special register must be introduced that will be used in many of these cases. The 16-bit General Register (GR) is expressly provided for performing byte singulation of 16-bit words. The high and low bytes of GR are individually accessible in the GRH and GRL registers respectively. A read-only GRS register makes a byte-swapped version of GR accessible and the GRXL register provides a sign-extended version of GRL.

8-bit destination ← low byte (16-bit source)

The simplest transfer possibility would be loading an 8-bit register with the low byte of a 16-bit register. This transfer does not require use of GR and requires a prefix only if the destination or source register are outside of the single cycle write or read regions, 0–7h and 0–Fh, respectively.

```
move      OFFS, LC[0]  ; copy the low byte of LC[0] to the OFFS register
move      IMR, @DP[1]  ; copy the low byte @DP[1] to the IMR register
move      WDCN, LC[0]  ; assembles to:    move PFX[2], #00h
                       ;                  move (WDCON-80h), LC[0]
```

8-bit destination ← high byte (16-bit source)

If, however, we needed to load an 8-bit register with the high byte of a 16-bit source, it would be best to use the GR register. Transferring the 16-bit source to the GR register adds a single cycle.

```
move      GR, LC[0]    ; move LC[0] to the GR register
move      IC, GRH      ; copy the high byte into the IC register
```

16-bit destination ← concatenation (8-bit source, 8-bit source)

Two 8-bit source registers can be concatenated and stored into a 16-bit destination by using the prefix register to hold the high-order byte for the concatenated transfer. An additional cycle may be required if either source byte register index is greater than 0Fh or the 16-bit destination is greater than 07h.

```
move      PFX[0], IC   ; load high order source byte IC into PFX
move      @++SP, AP    ; store @DP[0] the concatenation of IC:AP
                       ; 16-bit destination sub-index: dst=08h
                       ; 8-bit source sub-indexes:
                       ; high=10h, low=11h
move      PFX[1], #00h ;
move      PFX[3], high ; PFX=00:high
move      dst, low     ; dst=high:low
```

Low (16-bit destination) ← 8-bit source

To modify only the low byte of a given 16-bit destination, the 16-bit register should be moved into the GR register such that the high byte can be singulated and the low byte written exclusively. An additional cycle is required if the destination index is greater than 0Fh.

```
move      GR, DP[0]    ; move DP[0] to the GR register
move      PFX[0], GRH  ; get the high byte of DP[0] via GRH
move      DP[0], #20h  ; store the new DP[0] value
                       ; 16-bit destination sub-index: dst=10h
                       ; 8-bit source sub-index: src=11h
move      PFX[1], #00h ;
move      GR, dst      ; read dst word to the GR register
move      PFX[5], GRH  ; get the high byte of dst via GRH
move      dst, src     ; store the new dst value
```

High (16-bit destination) ← 8-bit source

To modify only the high byte of a given 16-bit destination, the 16-bit register should be moved into the GR register such that the low byte can be singulated and the high byte can be written exclusively. Additional cycles are required if the destination index is greater than 0Fh or if the source index is greater than 0Fh.

```
move      GR, DP[0]    ; move DP[0] to the GR register
move      PFX[0], #20h ; get the high byte of DP[0] via GRH
move      DP[0], GRL   ; store the new DP[0] value
                       ; 16-bit destination sub-index: dst=10h
                       ; 8-bit source sub-index: src=11h
move      PFX[1], #00h ;
move      GR, dst      ; read dst word to the GR register
move      PFX[1], #00h
move      PFX[4], src  ; get the new src byte
move      dst, GRL     ; store the new dst value
```

If the high byte needs to be cleared to 00h, the operation can be shortened by transferring only the GRL byte to the 16-bit destination (example follows):

```
move      GR, DP[0]    ; move DP[0] to the GR register
move      DP[0], GRL   ; store the new DP[0] value, 00h used for high byte
```

19.4 Reading and Writing Register Bits

The MOVE instruction can also be used to directly set or clear any one of the lowest 8 bits of a peripheral register in module 0h-5h or a system register in module 8h. The set or clear operation will not affect the upper byte of a 16-bit register that is the target of the set or clear operation. If a set or clear instruction is used on a destination register that does not support this type of operation, the register high byte will be written with the prefix data and the low byte will be written with the bit mask (i.e., all 0s with a single 1 for the set bit operation or all ones with a single 0 for the clear bit operation).

Register bits can be set or cleared individually using the MOVE instruction as follows.

```
move      IGE, #1      ; set IGE (Interrupt Global Enable) bit
move      APC.6, #0    ; clear IDS bit (APC.6)
```

As with other instructions, prefixing is required to select destination registers beyond index 07h.

The MOVE instruction may also be used to transfer any one of the lowest 8 bits from a register source or any bit of the active accumulator (Acc) to the Carry flag. There is no restriction on the source register module for the 'MOVE C, src. bit' instruction.

```
move      C, IIR.3     ; copy IIR.3 to Carry
move      C, Acc.7     ; copy Acc.7 to Carry
```

Prefixing is required to select source registers beyond index 15h.

19.5 Using the Arithmetic and Logic Unit

The MAX31782 provides a 16-bit ALU, which allows operations to be performed between the active accumulator and any other register. The MAX31782 provides 16 accumulator registers, of which any one may be selected as the active accumulator.

19.5.1 Selecting the Active Accumulator

Any of the 16 accumulator registers A[0] through A[15] may be selected as the active accumulator by setting the low four bits of the

Accumulator Pointer Register (AP) to the index of the accumulator register you want to select.

```
move      AP, #01h     ; select A[1] as the active accumulator
move      AP, #0Fh     ; select A[15] as the active accumulator
```

The current active accumulator can be accessed as the Acc register, which is also the register used as the implicit destination for all arithmetic and logical operations.

```
move      A[0], #55h   ; set A[0]
                       ;      = 0055 hex
move      AP, #00h     ; select A[0] as active accumulator
move      Acc, #55h    ; set A[0]
                       ;      = 0055 hex
```

19.5.2 Enabling Auto-Increment and Auto-Decrement

The accumulator pointer AP can be set to automatically increment or decrement after each arithmetic or logical operation. This is useful for operations involving a number of accumulator registers, such as adding or subtracting two multibyte integers.

If auto-increment/decrement is enabled, the AP register increments or decrements after any of the following operations:

- ADD src (Add source to active accumulator)
- ADDC src (Add source to active accumulator with carry)
- SUB src (Subtract source from active accumulator)
- SUBB src (Subtract source from active accumulator with borrow)
- AND src (Logical AND active accumulator with source)
- OR src (Logical OR active accumulator with source)
- XOR src (Logical XOR active accumulator with source)
- CPL (Bit-wise complement active accumulator)
- NEG (Negate active accumulator)
- SLA (Arithmetic shift left on active accumulator)
- SLA2 (Arithmetic shift left active accumulator two bit positions)
- SLA4 (Arithmetic shift left active accumulator four bit positions)
- SRA (Arithmetic shift right on active accumulator)
- SRA2 (Arithmetic shift right active accumulator two bit positions)
- SRA4 (Arithmetic shift right active accumulator four bit positions)
- RL (Rotate active accumulator left)
- RLC (Rotate active accumulator left through Carry flag)
- RR (Rotate active accumulator right)
- RRC (Rotate active accumulator right through Carry flag)
- SR (Logical shift active accumulator right)

- MOVE Acc, src (Copy data from source to active accumulator)
- MOVE dst, Acc (Copy data from active accumulator to destination)
- MOVE Acc, Acc (Recirculation of active accumulator contents)
- XCHN (Exchange nibbles within each byte of active accumulator)
- XCH (Exchange active accumulator bytes)

The active accumulator may not be the source in any instruction where it is also the implicit destination.

There is an additional notation that can be used to refer to the active accumulator for the instruction "MOVE dst, Acc." If the instruction is instead written as "MOVE dst, A[AP]," the source value is still the active accumulator, but no AP auto-increment or auto-decrement function will take place, even if this function is enabled. Note that the active accumulator may not be the destination for the MOVE dst, A[AP] instruction (i.e. MOVE Acc, A[AP] is prohibited).

So, the two instructions

```
move    A[7], Acc
move    A[7], A[AP]
```

are equivalent, except that the first instruction triggers auto-inc/dec (if it is enabled), while the second one will never do so.

The Accumulator Pointer Control Register (APC) controls the auto-inc/dec mode as well as selects the range of bits (modulo) in the AP register that will be incremented or decremented. There are nine different unique settings for the APC register, as listed in Table 19-1.

Table 19-1. Accumulator Pointer Control Register Settings

APC.2 (MOD2)	APC.1 (MOD1)	APC.0 (MOD0)	APC.6 (IDS)	APC	AUTO INCREMENT/DECREMENT SETTING
0	0	0	X	00h	No auto-increment/decrement (default mode)
0	0	1	0	01h	Increment bit 0 of AP (modulo 2)
0	0	1	1	41h	Decrement bit 0 of AP (modulo 2)
0	1	0	0	02h	Increment bits [1:0] of AP (modulo 4)
0	1	0	1	42h	Decrement bits [1:0] of AP (modulo 4)
0	1	1	0	03h	Increment bits [2:0] of AP (modulo 8)
0	1	1	1	43h	Decrement bits [2:0] of AP (modulo 8)
1	0	0	0	04h	Increment all 4 bits of AP (modulo 16)
1	0	0	1	44h	Decrement all 4 bits of AP (modulo 16)

For the modulo increment or decrement operation, the selected range of bits in AP are incremented or decremented. However, if these bits roll over or under, they simply wrap around without affecting the remaining bits in the accumulator pointer. So, the operations can be defined as follows:

- Increment modulo 2: AP = AP[3:1] + ((AP[0] + 1) mod 2)
- Decrement modulo 2: AP = AP[3:1] + ((AP[0] - 1) mod 2)
- Increment modulo 4: AP = AP[3:2] + ((AP[1:0] + 1) mod 4)
- Decrement modulo 4: AP = AP[3:2] + ((AP[1:0] - 1) mod 4)
- Increment modulo 8: AP = AP[3] + ((AP[2:0] + 1) mod 8)
- Decrement modulo 8: AP = AP[3] + ((AP[2:0] - 1) mod 8)
- Increment modulo 16: AP = (AP + 1) mod 16
- Decrement modulo 16: AP = (AP - 1) mod 16

For this example, assume that all 16 accumulator registers are initially set to zero.

```
move      AP, #02h     ; select A[2] as active accumulator
move      APC, #02h    ; auto-increment AP[1:0] modulo 4
                       ;      AP    A[0]  A[1]  A[2]  A[3]
                       ;      02    0000  0000  0000  0000
add       #01h         ;      03    0000  0000  0001  0000
add       #02h         ;      00    0000  0000  0001  0002
add       #03h         ;      01    0003  0000  0001  0002
add       #04h         ;      02    0003  0004  0001  0002
add       #05h         ;      03    0003  0004  0006  0002
```

19.5.3 ALU Operations Using the Active Accumulator and a Source

The following arithmetic and logical operations can use any register or immediate value as a source. The active accumulator Acc is always used as the second operand and the implicit destination. Also, Acc may not be used as the source for any of these operations.

```
add       A[4]         ; Acc = Acc + A[4]
addc      #32h
                       ; Acc = Acc + 0032h + Carry
sub       A[15]        ; Acc = Acc - A[15]
subb      A[1]         ; Acc = Acc - A[1] - Carry
cmp       #00h
                       ; If (Acc == 0000h), set Equals flag
and       A[0]         ; Acc = Acc AND A[0]
or        #55h
                       ;      Acc = Acc OR #0055h
xor       A[1]         ;      Acc = Acc XOR A[1]
```

19.5.4 ALU Operations Using Only the Active Accumulator

The following arithmetic and logical operations operate only on the active accumulator.

```
cpl                    ; Acc = NOT Acc
neg                    ; Acc = (NOT Acc) + 1
rl                     ; Rotate accumulator left (not using Carry)
rlc                    ; Rotate accumulator left through Carry
rr                     ; Rotate accumulator right (not using Carry)
rrc                    ; Rotate accumulator right through Carry
sla                    ; Shift accumulator left arithmetically once
sla2                   ; Shift accumulator left arithmetically twice
sla4                   ; Shift accumulator left arithmetically four times
sr                     ; Shift accumulator right, set Carry to Acc.0,
                       ; set Acc.15 to zero
sra                    ; Shift accumulator right arithmetically once
sra2                   ; Shift accumulator right arithmetically twice
sra4                   ; Shift accumulator right arithmetically four times
xchn                   ; Swap low and high nibbles of each Acc byte
xch                    ; Swap low byte and high byte of Acc
```

19.5.5 ALU Bit Operations Using Only the Active Accumulator

The following operations operate on single bits of the current active accumulator in conjunction with the Carry flag. Any of these operations may use an Acc bit from 0 to 15.

```
move    C, Acc.0    ; copy bit 0 of accumulator to Carry
move    Acc.5, C    ; copy Carry to bit 5 of accumulator
and     Acc.3       ; Acc.3 = Acc.3 AND Carry
or      Acc.0       ; Acc.0 = Acc.0 OR Carry
xor     Acc.1       ; Acc.1 = Acc.1 OR Carry
```

None of the above bit operations cause the auto-increment, auto-decrement, or modulo operations defined by the accumulator pointer control (APC) register.

19.5.6 Example: Adding Two 4-Byte Numbers Using Auto-Increment

```
move    A[0], #5678h ; First number – 12345678h
move    A[1], #1234h
move    A[2], #0AAAAh; Second number – 0AAAAAAAh
move    A[3], #0AAAh
move    APC, #81h    ; Active Acc = A[0], increment low bit = mod 2
add     A[2]         ; A[0] = 5678h + AAAAh = 0122h + Carry
addc    A[3]         ; A[1] = 1234h + AAAh + 1 = 1CDFh
                     ; 12345678h + 0AAAAAAAh = 1CDF0122h
```

9.6 Processor Status Flag Operations

The Processor Status Flag (PSF) register contains five flags that are used to indicate and store the results of arithmetic and logical operations, four of which can also be used for conditional program branching.

19.6.1 Sign Flag

The Sign flag (PSF.6) reflects the current state of the most significant bit of the active accumulator. If signed arithmetic is being used, this flag indicates whether the value in the accumulator is positive or negative.

Since the Sign flag is a dynamic reflection of the high bit of the active accumulator, any instruction that changes the value in the active accumulator can potentially change the value of the Sign flag. Also, any instruction that changes which accumulator is the active one (including AP auto-increment/decrement) can also change the Sign flag.

The following operation uses the Sign flag:

- JUMP S, src (Jump if Sign flag is set)

19.6.2 Zero Flag

The Zero flag (PSF.7) is a dynamic flag that reflects the current state of the active accumulator Acc. If all bits in the active accumulator are zero, the Zero flag equals 1. Otherwise, it equals 0.

Since the Zero flag is a dynamic reflection of (Acc = 0), any instruction that changes the value in the active accumulator can potentially change the value of the Zero flag. Also, any instruction that changes which accumulator is the active one (including AP auto-increment/decrement) can also change the Zero flag.

The following operations use the Zero flag:

- JUMP Z, src (Jump if Zero flag is set)
- JUMP NZ, src (Jump if Zero flag is cleared)

19.6.3 Equals Flag

The Equals flag (PSF.0) is a static flag set by the CMP instruction. When the source given to the CMP instruction is equal to the active accumulator, the Equals flag is set to 1. When the source is different from the active accumulator, the Equals flag is cleared to 0.

The following instructions use the value of the Equals flag. Please note that the 'src' for the JUMP E/NE instructions must be immediate.

- JUMP E, src (Jump if Equals flag is set)
- JUMP NE, src (Jump if Equals flag is cleared)

In addition to the CMP instruction, any instruction using PSF as the destination can alter the Equals flag.

19.6.4 Carry Flag

The Carry flag (PSF.1) is a static flag indicating that a carry or borrow bit resulted from the last ADD/ADDC or SUB/SUBB operation. Unlike the other status flags, it can be set or cleared explicitly and is also used as a generic bit operand by many other instructions.

The following instructions can alter the Carry flag:

- ADD src (Add source to active accumulator)
- ADDC src (Add source and Carry to active accumulator)
- SUB src (Subtract source from active accumulator)
- SUBB src (Subtract source and Carry from active accumulator)
- SLA, SLA2, SLA4 (Arithmetic shift left active accumulator)
- SRA, SRA2, SRA4 (Arithmetic shift right active accumulator)
- SR (Shift active accumulator right)
- RLC/RRC (Rotate active accumulator left / right through Carry)
- MOVE C, Acc.<b> (Set Carry to selected active accumulator bit)
- MOVE C, #i (Explicitly set, i = 1, or clear, i = 0, the Carry flag)
- CPL C (Complement Carry)
- AND Acc.<b>
- OR Acc.<b>
- XOR Acc.<b>
- MOVE C, src.<b> (Copy bit addressable register bit to Carry)
- Any instruction using PSF as the destination

The following instructions use the value of the Carry flag:

- ADDC src (Add source and Carry to active accumulator)
- SUBB src (Subtract source and Carry from active accumulator)
- RLC/RRC (Rotate active accumulator left/right through Carry)
- CPL C (Complement Carry)
- MOVE Acc.<b>, C (Set selected active accumulator bit to Carry)
- AND Acc.<b> (Carry = Carry AND selected active accumulator bit)
- OR Acc.<b> (Carry = Carry OR selected active accumulator bit)
- XOR Acc.<b> (Carry = Carry XOR selected active accumulator bit)
- JUMP C, src (Jump if Carry flag is set)
- JUMP NC, src (Jump if Carry flag is cleared)

19.6.5 Overflow Flag

The Overflow flag (PSF.2) is a static flag indicating that the carry or borrow bit (Carry status Flag) resulting from the last ADD/ADDC or SUB/SUBB operation but did not match the carry or borrow of the high order bit of the active accumulator. The overflow flag is useful when performing signed arithmetic operations.

The following instructions can alter the Overflow flag:

- ADD src (Add source to active accumulator)
- ADDC src (Add source and Carry to active accumulator)
- SUB src (Subtract source from active accumulator)
- SUBB src (Subtract source and Carry from active accumulator)

19.7 Controlling Program Flow

The MAX31782 provides several options to control program flow and branching. Jumps may be unconditional, conditional, relative, or absolute. Subroutine calls store the return address on the hardware stack for later return. Built-in counters and address registers are provided to control looping operations.

19.7.1 Obtaining the Next Execution Address

The address of the next instruction to be executed can be read at any time by reading the Instruction Pointer (IP) register. This can be particularly useful for initializing loops. Note that the value returned is actually the address of the current instruction plus 1, so this will be the address of the next instruction executed as long as the current instruction does not cause a jump.

19.7.2 Unconditional Jumps

An unconditional jump can be relative (IP +127/-128 words) or absolute (to anywhere in program space). Relative jumps must use an

8-bit immediate operand, such as

```
Label1:              ; must be within +127/-128 words of the JUMP
...
jump      Label1
```

Absolute jumps can use a 16-bit immediate operand, a 16-bit register, or an 8-bit register.

```
jump      LongJump     ; assembles to:    move PFX[0], #high(LongJump)
                       ;                  jump         #low(LongJump)
jump      DP[0]        ; absolute jump to the address in DP[0]
```

If an 8-bit register is used as the jump destination, the prefix value is used as the high byte of the address and the register is used as the low byte.

19.7.3 Conditional Jumps

Conditional jumps transfer program execution based on the value of one of the status flags (C, E, Z, S). Except where noted for JUMP E and JUMP NE, the absolute and relative operands allowed are the same as for the unconditional JUMP command.

```
   jump      c, Label1    ; jump to Label1 if Carry is set
   jump      nc, LongJump ; jump to LongJump if Carry is not set
   jump      z, LC[0]     ; jump to 16-bit register destination if
                          ;     Zero is set
   jump      nz, Label1   ; jump to Label1 if Zero is not set (Acc<>0)
   jump      s, A[2]      ; jump to A[2] if Sign flag is set
   jump      e, Label1    ; jump to Label1 if Equal is set
   jump      ne, Label1   ; jump to Label1 if Equal is cleared
```

JUMP E and JUMP NE may only use immediate destinations.

19.7.4 Calling Subroutines

The CALL instruction works the same as the unconditional JUMP, except that the next execution address is pushed on the stack before transferring program execution to the branch address. The RET instruction is used to return from a normal call, and RETI is used to return from an interrupt handler routine.

```
   call      Label1       ; if Label1 is relative,
                          ; assembles to :   call #immediate
   call      LongCall     ; assembles to:    move PFX[0], #high(LongCall)
                          ; call #low(LongCall)
   call      LC[0]        ; call to address in LC[0]
LongCall:
   ret                    ; return from subroutine
```

19.7.5 Looping Operations

Looping over a section of code can be performed by using the conditional jump instructions. However, there is built-in functionality, in the form of the 'DJNZ LC[n], src' instruction, to support faster, more compact looping code with separate loop counters. The 16-bit registers LC[0], and LC[1] are used to store these loop counts. The 'DJNZ LC[n], src' instruction automatically decrements the associated loop counter register and jumps to the loop address specified by src if the loop counter has not reached 0.

To initialize a loop, set the LC[n] register to the count you wish to use before entering the loop's main body.

The desired loop address should be supplied in the src operand of the 'DJNZ LC[n], src' instruction. When the supplied loop address is relative (+127/-128 words) to the DJNZ LC[n] instruction, as is typically the case, the assembler automatically calculates the relative offset and inserts this immediate value in the object code.

```
   move     LC[1], #10h        ; loop 16 times
LoopTop:                       ; loop addr relative to djnz LC[n],src instruction
   call     LoopSub
   djnz     LC[1], LoopTop     ; decrement LC[1] and jump if nonzero
```

When the supplied loop address is outside the relative jump range, the prefix register (PFX[0]) is used to supply the high byte of the loop address as required.

```
   move     LC[1], #10h        ; loop 16 times
LoopTop:                       ; loop addr not relative to djnz LC[n],src
   call     LoopSub
   ...
   djnz     LC[1], LoopTop     ; decrement LC[1] and jump if nonzero
                               ; assembles to:    move PFX[0], #high(LoopTop)
                               ;                  djnz LC[1],  #low(LoopTop)
```

If loop execution speed is critical and a relative jump cannot be used, one might consider preloading an internal 16-bit register with the src loop address for the 'DJNZ LC[n], src' loop. This ensures that the prefix register will not be needed to supply the loop address and always yields the fastest execution of the DJNZ instruction.

```
   move     LC[0], #LoopTop    ; using LC[0] as address holding register
                               ; assembles to:    move PFX[0], #high(LoopTop)
                               ;                  move LC[0],  #low(LoopTop)
   move     LC[1], #10h        ;     loop 16 times
   ...
LoopTop:                       ; loop address not relative to djnz LC[n],src
   call     LoopSub
   ...
   djnz     LC[1], LC[0]       ; decrement LC[1] and jump if nonzero
```

If opting to preload the loop address to an internal 16-bit register, the most time and code efficient means is by performing the load in the instruction just prior to the top of the loop:

```
   move     LC[1], #10h        ; Set loop counter to 16
   move     LC[0], IP          ; Set loop address to the next address
LoopTop:                       ; loop addr not relative to djnz LC[n],src
   ...
```

19.7.6 Conditional Returns

Similar to the conditional jumps, the MAX31782 also supports a set of conditional return operations. Based upon the value of one of the status flags, the CPU can conditionally pop the stack and begin execution at the address popped from the stack. If the condition is not true, the conditional return instruction does not pop the stack and does not change the instruction pointer. The following conditional return operations are supported:

```
   RET C                        ; if C=1, a RET is executed
   RET NC                       ; if C=0, a RET is executed
   RET Z                        ; if Z=1 (Acc=00h), a RET is executed
   RET NZ                       ; if Z=0 (Acc<>00h), a RET is executed
   RET S                        ; if S=1, a RET is executed
```

19.8 Handling Interrupts

Handling interrupts in the MAX31782 is a three-part process. First, the location of the interrupt handling routine must be set by writing the address to the 16-bit Interrupt Vector (IV) register. This register defaults to 0000h on reset, but this will usually not be the desired location since this will often be the location of reset/power-up code.

```
   move      IV, IntHandler     ; move PFX[0], #high(IntHandler)
                                ; move IV, #low(IntHandler)
                                ; PFX[0] write not needed if IntHandler addr=00xxh
```

Next, the interrupt must be enabled. For any interrupts to be handled, the IGE bit in the Interrupt and Control register (IC) must first be set to 1. Next, the interrupt itself must be enabled at the module level and locally within the module itself. The module interrupt enable is located in the Interrupt Mask register, while the location of the local interrupt enable will vary depending on the module in which the interrupt source is located.

Once the interrupt handler receives the interrupt, the Interrupt in Service (INS) bit will be set by hardware to block further interrupts, and execution control is transferred to the interrupt service routine. Within the interrupt service routine, the source of the interrupt must be determined. Since all interrupts go to the same interrupt service routine, the Interrupt Identification Register (IIR) must be examined to determine which module initiated the interrupt. For example, the II0 (IIR.0) bit will be set if there is a pending interrupt from module 0. These bits cannot be cleared directly; instead, the appropriate bit flag in the module must be cleared once the interrupt is handled.

INS is set automatically on entry to the interrupt handler and cleared automatically on exit (RETI).

```
IntHandler:
   push      PSF                ; save C since used in identification process
   move      C, IIR.X           ; check highest priority flag in IIR
   jump      C, ISR_X           ; if IIR.X is set, interrupt from module X
   move      C, IIR.Y           ; check next highest priority int source
   jump      C, ISR_Y           ; if IIR.Y is set, interrupt from module Y
   ...
ISR_X:
   ...
   reti
```

To support high priority interrupts while servicing another interrupt source, the IMR register may be used to create a user-defined prioritization. The IMR mask register should not be utilized when the highest priority interrupt is being serviced because the highest priority interrupt should never be interrupted. This is default condition when a hardware branch is made the Interrupt Vector address (INS is set to 1 by hardware and all other interrupt sources are blocked). The code below demonstrates how to use IMR to allow other interrupts.

```
ISR_Z:
   pop       PSF                ; restore PSF
   push      IMR                ; save current interrupt mask
   move      IMR, #int_mask     ; new mask to allow only higher priority ints move
   INS, #0                      ; re-enable interrupts
   ...
   (interrupt servicing code)
   ...
   pop       IMR                ; restore previous interrupt mask
   ret                          ; back to code or lower priority interrupt
```

Note that configuring a given IMR register mask bit to '0' only prevents interrupt conditions from the corresponding module or system from generating an interrupt request. Configuring an IMR mask bit to '0' does not prevent the corresponding IIR system or module identification flag from being set. This means that when using the IMR mask register functionality to block interrupts, there may be cases when both the mask (IMR.x) and identifier (IIR.x) bits should be considered when determining if the corresponding peripheral should be serviced.

19.8.1 Conditional Return from Interrupt

Similar to the conditional returns, the MAX31782 also supports a set of conditional return from interrupt operations. Based upon the value of one of the status flags, the CPU can conditionally pop the stack, clear the INS bit to 0, and begin execution at the address popped from the stack. If the condition is not true, the conditional return from interrupt instruction leaves the INS bit unchanged, does not pop the stack and does not change the instruction pointer. The following conditional return from interrupt operations are supported:

```
   RETI C                       ; if C=1, a RETI is executed
   RETI NC                      ; if C=0, a RETI is executed
   RETI Z                       ; if Z=1 (Acc=00h), a RETI is executed
   RETI NZ                      ; if Z=0 (Acc<>00h), a RETI is executed
   RETI S                       ; if S=1, a RETI is executed
```

19.9 Accessing the Stack

The hardware stack is used automatically by the CALL, RET and RETI instructions, but it can also be used explicitly to store and retrieve data. All values stored on the stack are 16 bits wide.

The PUSH instruction increments the stack pointer SP and then stores a value on the stack. When pushing a 16-bit value onto the stack, the entire value is stored. However, when pushing an 8-bit value onto the stack, the high byte stored on the stack comes from the pre- fix register. The @++SP stack access mnemonic is the associated destination specifier that generates this push behavior, thus the following two instruction sequences are equivalent:

```
move    PFX[0], IC
push    PSF             ; stored on stack: IC:PSF

move    PFX[0], IC
move    @++SP, PSF      ; stored on stack: IC:PSF
```

The POP instruction removes a value from the stack and then decrements the stack pointer. The @SP-- stack access mnemonic is the associated source specifier that generates this behavior, thus the following two instructions are equivalent:

```
pop     PSF
move    PSF, @SP--
```

The POPI instruction is equivalent to the POP instruction but additionally clears the INS bit to 0. Thus, the following two instructions would be equivalent:

```
popi    IP
reti
```

The @SP-- mnemonic can be used by the MAX31782 so that stack values may be used directly by ALU operations (e.g. ADD src, XOR src, etc.) without requiring that the value be first popped into an intermediate register or accumulator.

```
add     @SP--           ; sum the last three words pushed onto the stack
add     @SP--           ; with Acc, disregarding overflow
add     @SP--
```

The stack pointer SP can be set explicitly, however only the lowest four bits are used and setting SP to 0Fh will return it to its reset state.

Since the stack is 16 bits wide, it is possible to store two 8-bit register values on it in a single location. This allows more efficient use of the stack if it is being used to save and restore registers at the start and end of a subroutine.

```
SubOne:
    move    PFX[0], IC
    push    PSF             ; store IC:PSF on the stack
    ...
    pop     GR              ; 16-bit register
    move    IC, GRH         ; IC was stored as high byte
    move    PSF, GRL        ; PSF was stored as low byte ret
```

19.10 Accessing Data Memory

Data memory is accessed through the data pointer registers DP[0] and DP[1] or the Frame Pointer BP[OFFS]. Once one of these registers is set to a location in data memory, that location can be read or written as follows, using the mnemonic @DP[0], @DP[1] or @BP[OFFS] as a source or destination.

```
move    DP[0], #0000h           ; set pointer to location 0000h
move    A[0], @DP[0]            ; read from data memory
move    @DP[0], #55h            ; write to data memory
```

Either of the data pointers may be post-incremented or post-decremented following any read or may be pre-incremented or predecremented before any write access by using the following syntax.

```
move      A[0], @DP[0]++          ; increment DP[0] after read
move      @++DP[0], A[1]          ; increment DP[0] before write
move      A[5], @DP[1]--          ; decrement DP[1] after read
move      @--DP[1], #00h          ; decrement DP[1] before write
```

The Frame Pointer (BP[OFFS]) is actually composed of a base pointer (BP) and an offset from the base pointer (OFFS). For the frame pointer, the offset register (OFFS) is the target of any increment or decrement operation. The base pointer (BP) is unaffected by increment and decrement operations on the Frame Pointer. Similar to DP[n], the OFFS register may be preincremented/decremented when writing to data memory and may be post-incremented/decremented when reading from data memory.

```
move      A[0], @BP[OFFS--]       ; decrement OFFS after read
move      @BP[++OFFS], A[1]       ; increment OFFS before write
```

All three data pointers support both byte and word access to data memory. Each data pointer has its own word/byte select (WBSn) special-function register bit to control the access mode associated with the data pointer. These three register bits (WBS2, which controls BP[OFFS] access; WBS1, which controls DP[1] access; and WBS0, which controls DP[0] access) reside in the Data Pointer Control (DPC) register. When a given WBSn control bit is configured to 1, the associated pointer is operated in the word access mode. When the WBSn bit is configured to 0, the pointer is operated in the byte access mode. Word access mode allows addressing of 64kWords of memory while byte access mode allows addressing of 64kBytes of memory.

Each data pointer (DP[n]) and Frame Pointer base (BP) register is actually implemented internally as a 17-bit register (e.g., 16:0). The Frame Pointer offset register (OFFS) is implemented internally as a 9-bit register (e.g., 8:0). The WBSn bit for the respective pointer controls whether the highest 16 bits (16:1) of the pointer are in use, as is the case for word mode (WBSn = 1) or whether the lowest 16 bits (15:0) are in use, as will be the case for byte mode (WBSn = 0). The WBS2 bit also controls whether the high 8 bits (8:1) of the offset register are in use (WBS2 = 1) or the low 8 bits (7:0) are used (WBS2 = 0). All data pointer register reads, writes, auto-increment/decrement operations occur with respect to the current WBSn selection. Data pointer increment and decrement operations only affect those bits specific to the current word or byte addressing mode (e.g., incrementing a byte mode data pointer from FFFFh does not carry into the internal high order bit that is utilized only for word mode data pointer access). Switching from byte to word access mode or vice versa does not alter the data pointer contents. Therefore, it is important to maintain the consistency of data pointer address value within the given access mode.

```
move      DPC, #0                 ; DP[0] in byte mode
move      DP[0], #2345h           ; DP[0]=2345h (byte mode)
                                  ; internal bits 15:0 loaded
move      DPC, #4                 ; DP[0] in word mode
move      DP[0], #2345h           ; DP[0]=2345h (word mode)
                                  ; internal bits 16:1 loaded
move      DPC, #0                 ; DP[0] in byte mode
move      GR, DP[0]               ; GR = 468Bh (looking at bits 15:0)
```

The three pointers share a single read/write port on the data memory and thus, the user must knowingly activate a desired pointer before using it for data memory read operations. This can be done explicitly using the data pointer select bits (SDPS1:0; DPC.1:0), or implicitly by writing to the DP[n], BP, or OFFS registers as shown below. Any indirect memory write operation using a data pointer will set the SDPS bits, thus activating the write pointer as the active source pointer.

```
move      DPC, #2           ; (explicit) selection of FP as the pointer
move      DP[1], DP[1]      ; (implicit) selection of DP[1]; set SDPS1:0=01b
move      OFFS, src         ; (implicit) selection of FP; set SDPS1=1
move      @DP[0], src       ; (implicit) selection of DP[0]; set SDPS1:0=00b
```

Once the pointer selection has been made, it remains in effect until:

- The source data pointer select bits are changed via the explicit or implicit methods described above (i.e., another data pointer is selected for use).
- The memory to which the active source data pointer is addressing is enabled for code fetching using the Instruction Pointer. Or,
- A memory write operation is performed using a data pointer other than the current active source pointer.

```
move      DP[1], DP[1]       ; select DP[1] as the active pointer
move      dst, @DP[1]        ; read from pointer
move      @DP[1], src        ; write using a data pointer
                             ; DP[0] is needed
move      DP[0], DP[0]       ; select DP[0] as the active pointer
```

To simplify data pointer increment/decrement operations without disturbing register data, a virtual NUL destination has been assigned to system module 6, sub-index 7 to serve as a bit bucket. Data pointer increment/decrement operations can be done as follows with- out altering the contents of any other register:

```
move      NUL, @DP[0]++      ; increment DP[0]
move      NUL, @DP[0]--      ; decrement DP[0]
```

The following data pointer related instructions are invalid:

```
move @++DP[0], @DP[0]++
move @++DP[1], @DP[1]++
move @BP[++OFFS], @BP[OFFS++]
move @--DP[0], @DP[0]--
move @--DP[1], @DP[1]--
move @BP[--OFFS], @BP[OFFS--]
move @++DP[0], @DP[0]--
move @++DP[1], @DP[1]--
move @BP[++OFFS], @BP[OFFS--]
move @--DP[0], @DP[0]++
move @--DP[1], @DP[1]++
move @BP[--OFFS], @BP[OFFS++]
move @DP[0], @DP[0]++
move @DP[1], @DP[1]++
move @BP[OFFS], @BP[OFFS++]
move @DP[0], @DP[0]--
move @DP[1], @DP[1]--
move @BP[OFFS], @BP[OFFS--]
move DP[0], @DP[0]++
move DP[0], @DP[0]--
move DP[1], @DP[1]++
move DP[1], @DP[1]--
move OFFS, @BP[OFFS--]
move OFFS, @BP[OFFS++]
```

SECTION 20: INSTRUCTION SET SUMMARY

Table 20-1. Instruction Set Summary

	MNEMONIC	DESCRIPTION	16-BIT INSTRUCTION WORD	STATUS BITS AFFECTED	AP INC/ DEC	NOTES
LOGICAL OPERATIONS	AND src	Acc ← Acc AND src	f001 1010 ssss ssss	S, Z	Y	1
	OR src	Acc ← Acc OR src	f010 1010 ssss ssss	S, Z	Y	1
	XOR src	Acc ← Acc XOR src	f011 1010 ssss ssss	S, Z	Y	1
	CPL	Acc ← ~Acc	1000 1010 0001 1010	S, Z	Y	
	NEG	Acc ← ~Acc + 1	1000 1010 1001 1010	S, Z	Y	
	SLA	Shift Acc left arithmetically	1000 1010 0010 1010	C, S, Z	Y	
	SLA2	Shift Acc left arithmetically twice	1000 1010 0011 1010	C, S, Z	Y	
	SLA4	Shift Acc left arithmetically four times	1000 1010 0110 1010	C, S, Z	Y	
	RL	Rotate Acc left (w/o C)	1000 1010 0100 1010	S	Y	
	RLC	Rotate Acc left (through C)	1000 1010 0101 1010	C, S, Z	Y	
	SRA	Shift Acc right arithmetically	1000 1010 1111 1010	C, Z	Y	
	SRA2	Shift Acc right arithmetically twice	1000 1010 1110 1010	C, Z	Y	
	SRA4	Shift Acc right arithmetically four times	1000 1010 1011 1010	C, Z	Y	
	SR	Shift Acc right (0 → msbit)	1000 1010 1010 1010	C, S, Z	Y	
	RR	Rotate Acc right (w/o C)	1000 1010 1100 1010	S	Y	
	RRC	Rotate Acc right (though C)	1000 1010 1101 1010	C, S, Z	Y	
BIT OPERATIONS	MOVE C, Acc.<b>	C ← Acc.<b>	1110 1010 bbbb 1010	C		
	MOVE C, #0	C ← 0	1101 1010 0000 1010	C		
	MOVE C, #1	C ← 1	1101 1010 0001 1010	C		
	CPL C	C ← ~C	1101 1010 0010 1010	C		
	MOVE Acc.<b>, C	Acc.<b> ← C	1111 1010 bbbb 1010	S, Z		
	AND Acc.<b>	C ← C AND Acc.<b>	1001 1010 bbbb 1010	C		
	OR Acc.<b>	C ← C OR Acc.<b>	1010 1010 bbbb 1010	C		
	XOR Acc.<b>	C ← C XOR Acc.<b>	1011 1010 bbbb 1010	C		
	MOVE dst.<b>, #1	dst.<b> ← 1	1ddd dddd 1bbb 0111	C, S, E, Z		2
	MOVE dst.<b>, #0	dst.<b> ← 0	1ddd dddd 0bbb 0111	C, S, E, Z		2
	MOVE C, src.<b>	C ← src.<b>	fbbb 0111 ssss ssss	C		
MATH	ADD src	Acc ← Acc + src	f100 1010 ssss ssss	C, S, Z, OV	Y	1
	ADDC src	Acc ← Acc + (src + C)	f110 1010 ssss ssss	C, S, Z, OV	Y	1
	SUB src	Acc ← Acc – src	f101 1010 ssss ssss	C, S, Z, OV	Y	1
	SUBB src	Acc ← Acc – (src + C)	f111 1010 ssss ssss	C, S, Z, OV	Y	1

Table 20-1. Instruction Set Summary (continued)

	MNEMONIC	DESCRIPTION	16-BIT INSTRUCTION WORD	STATUS BITS AFFECTED	AP INC/ DEC	NOTES
BRANCHING	{L/S}JUMP src	IP ← IP + src or src	f000 1100 ssss ssss			6
	{L/S}JUMP C, src	If C=1, IP ← (IP + src) or src	f010 1100 ssss ssss			6
	{L/S}JUMP NC, src	If C=0, IP ← (IP + src) or src	f110 1100 ssss ssss			6
	{L/S}JUMP Z, src	If Z=1, IP ← (IP + src) or src	f001 1100 ssss ssss			6
	{L/S}JUMP NZ, src	If Z=0, IP ← (IP + src) or src	f101 1100 ssss ssss			6
	{L/S}JUMP E, src	If E=1, IP ← (IP + src) or src	0011 1100 ssss ssss			6
	{L/S}JUMP NE, src	If E=0, IP ← (IP + src) or src	0111 1100 ssss ssss			6
	{L/S}JUMP S, src	If S=1, IP ← (IP + src) or src	f100 1100 ssss ssss			6
	{L/S}DJNZ LC[n], src	If --LC[n] <> 0, IP ← (IP + src) or src	f10n 1101 ssss ssss			6
	{L/S}CALL src	@++SP ← IP+1; IP ← (IP+src) or src	f011 1101 ssss ssss			6,7
	RET	IP ← @SP--	1000 1100 0000 1101			
	RET C	If C=1, IP ← @SP--	1010 1100 0000 1101			
	RET NC	If C=0, IP ← @SP--	1110 1100 0000 1101			
	RET Z	If Z=1, IP ← @SP--	1001 1100 0000 1101			
	RET NZ	If Z=0, IP ← @SP--	1101 1100 0000 1101			
	RET S	If S=1, IP ← @SP--	1100 1100 0000 1101			
	RETI	IP ← @SP-- ; INS ← 0	1000 1100 1000 1101			
	RETI C	If C=1, IP ← @SP-- ; INS ← 0	1010 1100 1000 1101			
	RETI NC	If C=0, IP ← @SP-- ; INS ← 0	1110 1100 1000 1101			
	RETI Z	If Z=1, IP ← @SP-- ; INS ← 0	1001 1100 1000 1101			
	RETI NZ	If Z=0, IP ← @SP-- ; INS ← 0	1101 1100 1000 1101			
	RETI S	If S=1, IP ← @SP-- ; INS ← 0	1100 1100 1000 1101			
DATA TRANSFER	XCH	Swap Acc bytes	1000 1010 1000 1010	S	Y	
	XCHN	Swap nibbles in each Acc byte	1000 1010 0111 1010	S	Y	
	MOVE dst, src	Dst ← src	fddd dddd ssss ssss	C, S, Z, E	(Note 8)	7, 8
	PUSH src	@++SP ← src	f000 1101 ssss ssss			7
	POP dst	Dst ← @SP--	1ddd dddd 0000 1101	C, S, Z, E		7
	POPI dst	Dst ← @SP-- ; INS ← 0	1ddd dddd 1000 1101	C, S, Z, E		7
	CMP src	E ← (Acc = src)	f111 1000 ssss ssss	E		
	NOP	No operation	1101 1010 0011 1010			

Note 1: The active accumulator (Acc) is not allowed as the src in operations where it is the implicit destination.
Note 2: Only module 8 and modules 0-5 (when implemented for a given product) are supported by these single-cycle bit operations. Potentially affects C or E if PSF register is the destination. Potentially affects S and/or Z if AP or APC is the destination.
Note 3: The terms Acc and A[AP] can be used interchangeably to denote the active accumulator.
Note 4: Any index represented by <b> or found inside [ ] brackets is considered variable, but required.
Note 5: The active accumulator (Acc) is not allowed as the dst if A[AP] is specified as the src.
Note 6: The '{L/S}' prefix is optional.
Note 7: Instructions that attempt to simultaneously push/pop the stack (e.g. PUSH @SP--, PUSH @SPI--, POP @++SP, POPI @++SP) or modify SP in a conflicting manner (e.g., MOVE SP, @SP--) are invalid.
Note 8: Special cases: If 'MOVE APC, Acc' sets the APC.CLR bit, AP will be cleared, overriding any auto-inc/dec/modulo operation specified for AP. If 'MOVE AP, Acc' causes an auto-inc/dec/modulo operation on AP, this overrides the specified data transfer (i.e., Acc will not be transferred to AP).

ADD/ADDC src **Add/Add with Carry**

Description: The ADD instruction sums the active accumulator (Acc or A[AP]) and the specified src data and stores the result back to the active accumulator. The ADDC instruction additionally includes the Carry (C) Status Flag in the sum- mation. For the complete list of src specifiers, reference the MOVE instruction. Because the source field is limited to 8 bits, the PFX[n] register is used to supply the high-byte of data for 16 bit sources.

Status Flags: C, S, Z, OV

ADD Operation: Acc ← Acc + src

Encoding:

15			0
f100	1010	ssss	ssss

Example(s):

```
                        ; Acc = 2345h for each example
ADD A[3]                ; A[3]=FF0Fh
                        ; → Acc =2254h,C=1, Z=0, S=0, OV=0
ADD #0C0h               ; → Acc =2405h,C=0, Z=0, S=0, OV=0
ADD A[4]                ; A[4]=C000h
                        ; → Acc = E345h, C=0, Z=0, S=1, OV=0
ADD A[5]                ; A[5]=6789h
                        ; → Acc = 8ACEh, C=0, Z=0, S=1, OV=1
```

ADDC Operation: Acc ← Acc + C + src

Encoding:

15			0
f110	1010	ssss	ssss

Example(s):

```
                        ; Acc = 2345h for each example
ADDC A[3]               ; A[3] = DCBAh, C=1
                        ; → Acc = 0000h, C=1, Z=1, S=0, OV=0
ADDC @DP[0]--           ; @DP[0] = 00EEh, C=1
                        ; → Acc = 2434h, C=0, Z=0, S=0, OV=0
```

Special Notes: The active accumulator (Acc) is not allowed as the src for these operations.

AND src **Logical AND**

Description: Performs a logical-AND between the active accumulator (Acc) and the specified src data. For the complete list of src specifiers, reference the MOVE instruction. Because the source field is limited to 8 bits, the PFX[n] register is used to supply the high-byte of data for 16 bit sources.

Status Flags: S, Z

Operation: Acc ← Acc AND src

Encoding:

15			0
f001	1010	ssss	ssss

Example(s):

```
                        ; Acc = 2345h for each example
AND A[3]                ; A[3]=0F0Fh
                        ; → Acc = 0305h, S=0, Z=0
AND  #33h               ; → Acc = 0001h
AND #2233h              ; generates object code below
                        ; MOVE PFX[0], #22h (smart-prefixing)
                        ; AND #33h
                        ; → Acc = 2201h
MOVE PFX[0], #0Fh
AND M0[8]               ; M0[8]=0Fh (assume M0[8] is an 8-bit register)
                        ; → Acc = 0305h
```

Special Notes: The active accumulator (Acc) is not allowed as the src for this operation.

AND Acc.<b> **Logical AND Carry Flag with Accumulator Bit**

Description: Performs a logical-AND between the Carry (C) status flag and a specified bit of the active accumulator (Acc.<b>) and returns the result to the Carry.

Status Flags: C

Operation: C ← C AND Acc. <b>

Encoding:

15			0
1001	1010	bbbb	1010

Example(s):

```
                        ; Acc = 2345h, C=1 at start
AND Acc.0               ; Acc.0=1 → C=1
AND Acc.1               ; Acc.1=0 → C=0
AND C, Acc.8            ; Acc.8=1 → C=0
```

{L/S}CALL src **{Long/Short} Call to Subroutine**

Description: Performs a call to the subroutine destination specified by src. The CALL instruction uses an 8-bit immediate src to perform a relative short call (IP +127/-128 words). The CALL instruction uses a 16-bit immediate src to perform an absolute long CALL to the specified 16-bit address. The PFX[0] register is used to supply the high byte of a 16-bit immediate address for the absolute long CALL. Using the optional 'L' prefix (i.e., LCALL) results in an absolute long call and use of the PFX[0] register. Using the optional 'S' prefix (i.e., SCALL) attempts to generate a relative short call, but is flagged by the assembler if the destination is out of range. Specifying an internal register src always produces an absolute CALL to a 16-bit address, thus the 'L' and 'S' prefixes should not be used.

Status Flags: None

Operation:

@++SP ← IP + 1	PUSH
IP ← src	Absolute CALL
IP ← IP + src	Relative CALL

Encoding:

15 0

f011	1101	ssss	ssss

Example(s):

```
CALL label1          ; relative call to label1 (must be within IP +127/ -
                     ; 128 address range)
CALL label1          ; absolute call to label1 = 0120h
                     ; MOVE PFX[0], #01h
                     ; CALL #20h.
CALL  DP[0]          ; DP[0] holds 16-bit address of subroutine
LCALL label1         ; label=0120h and is relative to this instruction
                     ; absolute call is forced by use of 'L' prefix
                     ; MOVE PFX[0], #01h
                     ; CALL #20h
SCALL label1         ; relative offset for label1 calculated and used
                     ; if label1 is not relative, assembler will generate an error
SCALL #10h           ; relative offset of #10h is used directly by the CALL
```

CMP src **Compare Accumulator**

Description: Compare for equality between the active accumulator and the least significant byte of the specified src. Because the source is limited to 8 bits, the PFX[n] register is used to supply the high-byte of data for 16 bit sources.

Status Flags: E

Operation: Acc = src: E ← 1

Acc <> src: E ← 0

Encoding:

15			0
f111	1000	ssss	ssss

Example(s):

```
CMP #45h        ; Acc = 0145h, E=0
CMP #145h       ; PFX[0] register used
                ; MOVE PFX[0], #01h (smart-prefixing)
                ; CMP #45h  E=1
```

CPL **Complement Acc**

Description: Performs a logical bitwise complement (1's complement) on the active accumulator (Acc or A[AP]) and returns the result to the active accumulator.

Status Flags: S, Z

Operation: Acc ← ~Acc

Encoding:

15			0
1000	1010	0001	1010

Example(s):

```
                ; Acc = FFFFh, S=1, Z=0
CPL             ; Acc ← 0000h, S=0, Z=1
                ; Acc = 0990h, S=0, Z=0
CPL             ; Acc ← F66Fh, S=1, Z=0
```

CPL C — **Complement Carry Flag**

Description: Logically complements the Carry (C) Flag.

Status Flags: C

Operation: C ← ~C

Encoding:

15			0
1101	1010	0010	1010

Example(s):

```
                    ; C = 0
CPL C               ; C ← 1
```

{L/S}DJNZ LC[n], src — **Decrement Counter, {Long/Short} Jump Not Zero**

Description: The DJNZ LC[n], src instruction performs a conditional branch based upon the associated Loop Counter (LC[n]) reg- ister. The DJNZ LC[n], src instruction decrements the LC[n] loop counter and branches to the address defined by src if the decremented counter has not reached 0000h. Program branches can be relative or absolute depending upon the src specifier and may be qualified by using the 'L' or 'S' prefixes as documented in the JUMP src op code.

Status Flags: None

Operation: LC[n] ← LC[n] -1

LC[n] <> 0: IP ← IP + src (relative) -or- src (absolute)

LC[n] = 0: IP ← IP + 1

Encoding:

15			0
f10n	1101	ssss	ssss

Example(s):

```
MOVE LC[1], #10h    ; counter = 10h
Loop:
ADD @DP[0]++        ; add data memory contents to Acc, post-inc DP[0]
DJNZ LC[1], Loop    ; 16 times before falling through
```

{L/S} JUMP src **Unconditional {Long/Short} Jump**

Description: Performs an unconditional jump as determined by the src specifier. The JUMP instruction uses an 8-bit immediate src to perform a relative jump (IP +127/-128 words). The JUMP instruction uses a 16-bit immediate src to perform an absolute JUMP to the specified 16-bit address. The PFX[0] register is used to supply the high byte of a 16-bit immediate address for the absolute JUMP. Using the optional 'L' prefix (i.e., LJUMP) results in an absolute long jump and use of the PFX[0] register. Using the optional 'S' prefix (i.e., SJUMP) attempts to generate a relative short jump, but will be flagged by the assembler if the destination is out of range. Specifying an internal register src always produces an absolute JUMP to a 16-bit address, thus the 'L' and 'S' prefixes should not be used.

Status Flags: None

Operation:

IP ← src	Absolute JUMP
IP ← IP + src	Relative JUMP

Encoding:

15			0
f000	1100	ssss	ssss

Example(s):

```
JUMP label1          ; relative jump to label1 (must be within range
                     ; IP +127/-128 words)
JUMP label1          ; absolute jump to label1= 0400h
                     ; MOVE PFX[0], #04h
                     ; JUMP #00h
JUMP DP[0]           ; absolute jump to addr16 DP[0]
LJUMP label1         ; label=0120h and is relative to this instruction
                     ; absolute jump is forced by use of 'L' prefix
                     ; MOVE PFX[0], #01h
                     ; JUMP #20h
SJUMP label1         ; relative offset for label1 calculated and used
                     ; if label1 is not relative, assembler will generate an error
SJUMP #10h           relative offset of #10h is used directly by the JUMP
```

{L/S}JUMP C/{L/S}JUMP NC, src, L/S}JUMP Z/{L/S}JUMP NZ, src, {{L/S}JUMP E/{L/S}JUMP NE, src, {L/S}JUMP S, src

Conditional {Long/Short} Jump on Status Flag

Description: Performs conditional branching based upon the state of a specific processor status flag. JUMP C results in a branch if the Carry flag is set while JUMP NC branches if the Carry flag is clear. JUMP Z results in a branch if the Zero flag is set while JUMP NZ branches if the Zero flag is clear. JUMP E results in a branch if the Equal flag is set while JUMP NE branches if the Equal flag is clear. JUMP S results in a branch if the Sign flag is set. Program branches can be relative or absolute depending upon the src specifier and may be qualified by using the 'L' or 'S' prefixes as docu- mented in the JUMP src op code. Special src restrictions apply to JUMP E and JUMP NE.

Status Flags: None

JUMP C
Operation: C=1: IP ← IP + src (relative) -or- src (absolute)
C=0: IP ← IP + 1

Encoding:

15			0
f010	1100	ssss	ssss

Example(s):
```
JUMP C, label1          ; C=0, branch not taken
```

JUMP NC
Operation: C=0: IP ← IP + src (relative) -or- src (absolute)
C=1: IP ← IP +1

Encoding:

15			0
f110	1100	ssss	ssss

Example(s):
```
JUMP NC, label1         ; C=0, branch taken
```

JUMP Z
Operation: Z=1: IP ← IP + src
Z=0: IP ← IP + 1

Encoding:

15			0
f001	1100	ssss	ssss

Example(s):
```
JUMP Z, label1          ; Z=1, branch taken
```

JUMP NZ Z=0: IP ← IP + src (relative) -or- src (absolute)
Operation: Z=1: IP ← IP + 1
Encoding:

15			0
f101	1100	ssss	ssss

Example(s): JUMP NZ, label1 ; Z=1, branch taken

JUMP E E=1: IP ← IP + src (relative) -or- src (absolute)
Operation: E=0: IP ← IP + 1
Encoding:

15			0
0011	1100	ssss	ssss

Example(s): JUMP E, label1 ; E=1, branch taken
Special Notes: The src specifier must be immediate data.

JUMP NE
Operation: E=0: IP ← IP + src (relative) -or- src (absolute)
E=1: IP ← IP + 1
Encoding:

15			0
0111	1100	ssss	ssss

Example(s): JUMP NE, label1 ; E=0, branch taken
Special Notes: The src specifier must be immediate data.

JUMP S S=1: IP ← IP + src (relative) -or- src (absolute)
Operation: S=0: IP ← IP + 1
Encoding:

15			0
f100	1100	ssss	ssss

Example(s): JUMP S, label1 ; S=0, branch not taken

MOVE dst, src **Move Data**

Description: Moves data from a specified source (src) to a specified destination (dst). A list of defined source, destination spec- ifiers is given in the table below. Also, since src can be either 8-bit (byte) or 16-bit (word) data, the rules governing data transfer are also explained below in the encoding section.

Status Flags: S, Z (if dst is Acc or AP or APC) C, E (if dst is PSF)

Operation: dst ← src

Encoding:

Table 20-2. Source Specifier Codes

src	src Bit Encoding (f ssssssss)	WIDTH (16 or 8)	DESCRIPTION
#k	0 kkkk kkkk	8	kkkkkkkk = Immediate (Literal) Data
MN[n]	1 nnnn 0NNN	8/16	nnnn Selects One of First 16 Registers in Module NNN; where NNN= 0 to 5. Access to Second 16 using PFX[n].
AP	1 0000 1000	8	Accumulator Pointer
APC	1 0001 1000	8	Accumulator Pointer Control
PSF	1 0100 1000	8	Processor Status Flag Register
IC	1 0101 1000	8	Interrupt and Control Register
IMR	1 0110 1000	8	Interrupt Mask Register
SC	1 1000 1000	8	System Control Register
IIR	1 1011 1000	8	Interrupt Identification Register
CKCN	1 1110 1000	8	Clock Control Register
WDCN	1 1111 1000	8	Watchdog Control Register
A[n]	1 nnnn 1001	16	nnnn Selects One of 16 Accumulators
Acc	1 0000 1010	16	Active Accumulator = A[AP]. Update AP per APC
A[AP]	1 0001 1010	16	Active Accumulator = A[AP]. No change to AP
IP	1 0000 1100	16	Instruction Pointer
@SP--	1 0000 1101	16	16-Bit Word @SP, Post-Decrement SP
SP	1 0001 1101	16	Stack Pointer
IV	1 0010 1101	16	Interrupt Vector
LC[n]	1 011n 1101	16	n Selects 1 of 2 Loop Counter Registers
@SPI--	1 1000 1101	16	16-bit word @SP, Post-Decrement SP, INS=0
@BP[Offs]	1 0000 1110	8/16	Data Memory @BP[Offs]
@BP[Offs++]	1 0001 1110	8/16	Data memory @BP[Offs]; Post Increment OFFS
@BP[Offs--]	1 0010 1110	8/16	Data Memory @BP[Offs]; Post Decrement OFFS
OFFS	1 0011 1110	8	Frame Pointer Offset from Base Pointer (BP)
DPC	1 0100 1110	16	Data Pointer Control Register
GR	1 0101 1110	16	General Register
GRL	1 0110 1110	8	Low Byte of GR Register
BP	1 0111 1110	16	Frame Pointer Base Pointer (BP)
GRS	1 1000 1110	16	Byte-Swapped GR Register
GRH	1 1001 1110	8	High Byte of GR Register
GRXL	1 1010 1110	16	Sign Extended Low Byte of GR Register
FP	1 1011 1110	16	Frame Pointer (BP[Offs])
@DP[n]	1 0n00 1111	8/16	Data Memory @DP[n]
@DP[n]++	1 0n01 1111	8/16	Data Memory @DP[n], Post-Increment DP[n]
@DP[n]--	1 0n10 1111	8/16	Data Memory @DP[n], Post-Decrement DP[n]
DP[n]	1 0n11 1111	16	n Selects 1 of 2 Data Pointers

Table 20-3. Destination Specifier Codes

dst	dst Bit Encoding (ddd dddd)	WIDTH (16 OR 8)	DESCRIPTION
NUL	111 0110	8/16	Null (Virtual) Destination. Intended as a bit bucket to assist software with pointer increments/decrements.
MN[n]	nnn 0NNN	8/16	nnnn Selects One of First 8 Registers in Module NNN; where NNN= 0 to 5. Access to Next 24 Using PFX[n].
AP	000 1000	8	Accumulator Pointer
APC	001 1000	8	Accumulator Pointer Control
PSF	100 1000	8	Processor Status Flag Register
IC	101 1000	8	Interrupt and Control Register
IMR	110 1000	8	Interrupt Mask Register
A[n]	nnn 1001	16	nnn Selects 1 of First 8 Accumulators: A[0]..A[7]
Acc	000 1010	16	Active Accumulator = A[AP]
PFX[n]	nnn 1011	8	nnn Selects One of 8 Prefix Registers
@++SP	000 1101	16	16-Bit Word @SP, Pre-Increment SP
SP	001 1101	16	Stack Pointer
IV	010 1101	16	Interrupt Vector
LC[n]	11n 1101	16	n Selects 1 of 2 Loop Counter Registers
@BP[Offs]	000 1110	8/16	Data Memory @BP[Offs]
@BP[++Offs]	001 1110	8/16	Data Memory @BP[Offs]; Pre-Increment OFFS
@BP[--Offs]	010 1110	8/16	Data Memory @BP[Offs]; Pre-Decrement OFFS
OFFS	011 1110	8	Frame Pointer Offset from Base Pointer (BP)
DPC	100 1110	16	Data Pointer Control Register
GR	101 1110	16	General Register
GRL	110 1110	8	Low Byte of GR Register
BP	111 1110	16	Frame Pointer Base Pointer (BP)
@DP[n]	n00 1111	8/16	Data Memory @DP[n]
@++DP[n]	n01 1111	8/16	Data Memory @DP[n], Pre-Increment DP[n]
@--DP[n]	n10 1111	8/16	Data Memory @DP[n], Pre-Decrement DP[n]
DP[n]	n11 1111	16	n Selects 1 of 2 Data Pointers
2-CYCLE DESTINATION ACCESS USING PFX[n] REGISTER (See Special Notes)			
SC	000 1000	8	System Control Register
CKCN	110 1000	8	Clock Control Register
WDCN	111 1000	8	Watchdog Control Register
A[n]	nnn 1001	16	nnn Selects 1 of Second 8 Accumulators A[8]..A[15]
GRH	001 1110	8	High Byte of GR Register

Data Transfer Rules

dst (16-bit) ← src (16-bit): dst[15:0] ← src[15:0]
dst (8-bit) ← src (8-bit): dst[7:0] ← src[7:0]
dst (16-bit) ← src (8-bit): dst[15:8] ← 00h *
dst[7:0] ← src[7:0]
dst (8-bit) ← src (16-bit): dst[7:0] ← src[7:0]

*__Note:__ The PFX[0] register may be used to supply a separate high-order data byte for this type of transfer.

Example(s):

```
MOVE A[0], A[3]        ; A[0] ← A[3]
MOVE DP[0], #110h      ; DP[0] ← #0110h  (PFX[0] register used)
                       ; MOVE PFX[0], #01h (smart-prefixing)
                       ; MOVE DP[0], #10h
MOVE DP[0], #80h       ; DP[0] ← #0080h (PFX[0] register not needed)
```

Special Notes: Proper loading of the PFX[n] registers, when for the purpose of supplying 16-bit immediate data or accessing 2-cycle destinations, is handled automatically by the assembler and is therefore an optional step for the user when writing assembly source code. Examples of the automatic PFX[n] code insertion by the assembler are demonstrated below.

Initial Assembly Code	Assembler Output
MOVE DP[0], #0100h	MOVE PFX[0], #01h
MOVE A[15], A[7]	MOVE PFX[2], anysrc
	MOVE A[7], A[7]
MOVE A[8], #3040h	
MOVE PFX[2], #30h	MOVE A[0], #40h

MOVE Acc.<b>, C — Move Carry Flag to Accumulator Bit

Description: Replaces the specified bit of the active accumulator with the Carry bit.

Status Flags: S, Z

Operation: Acc.<b> ← C

Encoding:

15			0
1111	1010	bbbb	1010

Example(s):

```
                       ; Acc = 8000h, S=1, Z=0, C=0
MOVE Acc.15, C         ; Acc = 0000h, S=0, Z=1
```

MOVE C, Acc.<b> **Move Accumulator Bit to Carry Flag**

Description: Replaces the Carry (C) status flag with the specified active accumulator bit.

Status Flags: C

Operation: C ← Acc.<b>

Encoding:

15			0
1110	1010	bbbb	1010

Example(s):

```
                        ; Acc = 01C0h, C=0
MOVE C, Acc.8           ; C =1
```

MOVE C, src.<b> **Move Bit to Carry Flag**

Description: Replaces the Carry (C) status flag with the specified source bit src.<b>.

Status Flags: C

Operation: C ← src.<b>

Encoding:

15			0
fbbb	0111	ssss	ssss

Example(s):

```
                        ; M0[0] = FEh; C=1 (assume M0[0] is an 8-bit register)
MOVE C, M0[0].0         ; C=0
```

MOVE C, #0 **Clear Carry Flag**

Description: Clears the Carry (C) processor status flag.

Status Flag: C ← 0

Operation: C ← 0

Encoding:

15			0
1101	1010	0000	1010

Example(s):

```
                        ; C = 1
MOVE C, #0              ; C ← 0
```

MOVE C, #1 **Set Carry Flag**

Description: Sets the Carry (C) processor status flag.

Status Flag: C ← 1

Operation: C ← 1

Encoding:

15			0
1101	1010	0001	1010

Example(s):

```
                        ; C = 0
MOVE C, #1              ; C ← 1
```

MOVE dst.<b>, #0 **Clear Bit**

Description: Clears the bit specified by dst.<b>.

Status Flags: C, E (if dst is PSF), S, Z

Operation: dst.<b> ← 0

Encoding:

15			0
1ddd	dddd	0bbb	0111

Example(s):

```
                        ; M0[0] = FEh
MOVE M0[0].1, #0        ; M0[0] = FCh
MOVE M0[0].7, #0        ; M0[0] = 7Ch
```

Special Notes: Only system module 8 and peripheral modules (0-5) are supported by MOVE dst.<b>, #0.

MOVE dst.<b>, #1 **Set Bit**

Description: Sets the bit specified by dst.<b>.

Status Flags: C, E (if dst is PSF), S, Z

Operation: dst.<b> ← 1

Encoding:

15			0
1ddd	dddd	1bbb	0111

Example(s):

```
                        ; M0[0] = 00h
MOVE M0[0].1, #1        ; M0[0] = 02h
MOVE M0[0].7, #1        ; M0[0] = 82h
```

Special Notes: Only system module 8 and peripheral modules (0-5) are supported by MOVE dst.<b>, #1.

NEG — **Negate Accumulator**

Description: Performs a negation (two's complement) of the active accumulator and returns the result back to the active accumulator.

Status Flags: S, Z

Operation: Acc ← ~Acc + 1

Encoding:

15			0
1000	1010	1001	1010

Example(s):

```
                    ; Acc = FEEDh, S=1, Z=0
NEG                 ; Acc = 0113h,  S=0, Z=0
```

OR src — **Logical OR**

Description: Performs a logical-OR between the active accumulator (Acc or A[AP]) and the specified src data. For the complete list of src specifiers, reference the MOVE instruction. Because the source is limited to 8 bits, the PFX[n] register is used to supply the high-byte of data for 16 bit sources.

Status Flags: S, Z

Operation: Acc ← Acc OR src

Encoding:

15			0
f010	1010	ssss	ssss

Example(s):

```
                    ; Acc = 2345h for each example
OR A[3]             ; A[3]= 0F0Fh → Acc = 2F4Fh
OR #1133h           ; MOVE PFX[0], #11h (smart-prefixing)
                    ; OR #33h → Acc = 3377h
```

Special Notes: The active accumulator (Acc) is not allowed as the src for this operation.

OR Acc.<b> **Logical OR Carry Flag with Accumulator Bit**

Description: Performs a logical-OR between the Carry (C) status flag and a specified bit of the active accumulator (Acc.<b>) and returns the result to the Carry.

Status Flags: C

Operation: C ← C OR Acc.<b>

Encoding:

15			0
1010	1010	bbbb	1010

Example(s):

```
                    ; Acc = 2345h, C=0 at start
OR Acc.1            ; Acc.1=0 → C=0
OR Acc.2            ; Acc.2=1 → C=1
```

POP dst **Pop Word from the Stack**

Description: Pops a single word from the stack (@SP) to the specified dst and decrements the stack pointer (SP).

Status Flags: S, Z (if dst = Acc or AP or APC) C, E (if dst = PSF)

Operation: dst ← @ SP--

Encoding:

15			0
1ddd	dddd	0000	1101

Example(s):

```
                    ; GR ← 1234h
POP GR              ; @DP[0] ← 76h (WBS0=0)
POP @DP[0]          ; @DP[0] ← 0876h (WBS0=1)
```

Stack Data:

xxxxh	
1234h	← SP (initial)
0876h	← SP (after POP GR)
xxxxh	← SP (after POP @DP[0])
xxxxh	

POPI dst — **Pop Word from the Stack Enable Interrupts**

Description: Pops a single word from the stack (@SP) to the specified dst and decrements the stack pointer (SP). Additionally, POPI returns the interrupt logic to a state in which it can acknowledge additional interrupts.

Status Flags: S, Z (if dst = Acc or AP or APC)

C, E (if dst = PSF)

Operation: dst ← @ SP--

INS ← 0

Encoding:

15			0
1ddd	dddd	1000	1101

Example(s): See POP

PUSH src — **Push Word to the Stack**

Description: Increments the stack pointer (SP) and pushes a single word specified by src to the stack (@SP).

Status Flags: None

Operation: SP ← ++SP

Encoding:

15			0
f000	1101	ssss	ssss

Example(s):

```
PUSH GR                 ; GR=0F3Fh
PUSH #40h
```

Stack Data:

xxxxh	
0040h	← SP (after PUSH #40h)
0F3F	← SP (after PUSH GR)
xxxxh	← SP (initial)
xxxxh	

RET **Return from Subroutine**

Description: RET pops a single word from the stack (@SP) into the Instruction Pointer (IP) and decrements the stack pointer (SP). The decremented SP is saved as the new stack pointer (SP).

Status Flags: None

Operation: IP ← @ SP--

Encoding:

15			0
1000	1100	0000	1101

Example(s): RET

Code Execution:

Addr (IP)	Op Code
0311h	...
0312h	RET
0103h	...

Stack Data:

xxxxh	
xxxxh	
0103h	← SP (before RET)
xxxxh	← SP (after RET)
xxxxh	

RET C/RET NC, RET Z/RET NZ, RET S **Conditional Return on Status Flag**

Description: Performs conditional return (RET) based upon the state of a specific processor status flag. RET C returns if the Carry flag is set while RET NC returns if the Carry flag is clear. RET Z returns if the Zero flag is set while RET NZ returns if the Zero flag is clear. RET S returns if the Sign flag is set. See RET for additional information on the return operation.

Status Flags: None

RET C

Operation: C=1: IP ← @SP--

C=0: IP ← IP + 1

Encoding:

15			0
1010	1100	0000	1101

Example(s):

```
RET C                    ; C=1, return (RET) is performed
```

RET NC

Operation: C=0: IP ← @SP--
C=1: IP ← IP +1

Encoding:

15			0
1110	1100	0000	1101

Example(s): RET NC ; C=1, return (RET) does not occur

RET Z

Operation: Z=1: IP ← @SP--
Z=0: IP ← IP + 1

Encoding:

15			0
1001	1100	0000	1101

Example(s): RET Z ; Z=0, return (RET) does not occur

RET NZ

Operation: Z=0: IP ← @SP--
Z=1: IP ← IP +1

Encoding:

15			0
1101	1100	0000	1101

Example(s): RET NZ ; Z=0, return (RET) is performed

RET S

Operation: S=1: IP ← @SP--
S=0: IP ← IP + 1

Encoding:

15			0
1100	1100	0000	1101

Example(s): RET S ; S=0, return (RET) does not occur

RETI **Return from Interrupt**

Description: RETI pops a single word from the stack (@SP) into the Instruction Pointer (IP) and decrements the stack pointer (SP). Additionally, RETI returns the interrupt logic to a state in which it can acknowledge additional interrupts.

Status Flags: None

Operation: IP ← @SP--

INS ← 0

Encoding:

15			0
1000	1100	1000	1101

Example(s): See RETI

RETI C/RETI NC, RETI Z/RETI NZ, RETI S **Conditional Return from Interrupt on Status Flag**

Description: Performs conditional return (RETI) based upon the state of a specific processor status flag. RETI C returns if the Carry flag is set while RETI NC returns if the Carry flag is clear. RETI Z returns if the Zero flag is set while RETI NZ returns if the Zero flag is clear. RETI S returns if the Sign flag is set. See RETI for additional information on the return operation.

Status Flags: None

RETI C

Operation: C=1: IP ← @SP--

INS ← 0

C=0: IP ← IP + 1

Encoding:

15			0
1010	1100	1000	1101

Example(s):

```
RETI C                  ; C=1, return from interrupt (RETI) is performed
```

RETI NC

Operation: C=0: IP ← @SP--

INS ← 0

C=1: IP ← IP +1

Encoding:

15			0
1110	1100	1000	1101

Example(s):

```
RETI NC                 ; C=1, return from interrupt (RETI) does not occur
```

RETI Z

Operation: Z=1: IP ← @SP--
INS ← 0
Z=0: IP ← IP + 1

Encoding:

15			0
1001	1100	1000	1101

Example(s): RETI Z ; Z=0, return from interrupt (RETI) does not occur

---

RETI NZ

Operation: Z=0: IP ← @SP--
INS ← 0
Z=1: IP ← IP +1

Encoding:

15			0
1101	1100	1000	1101

Example(s): RETI NZ ; Z=0, return from interrupt (RETI) is performed

---

RETI S

Operation: S=1: IP ← @SP--
INS ← 0
S=0: IP ← IP + 1

Encoding:

15			0
1100	1100	1000	1101

Example(s): RETI S ; S=0, return from interrupt (RETI) does not occur

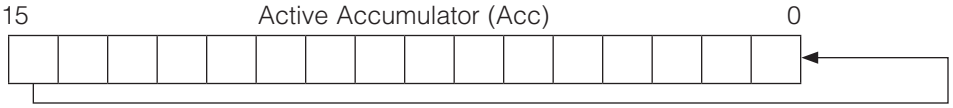
---

RL/RLC — **Rotate Left Accumulator Carry Flag (Ex/In)clusive**

Description: Rotates the active accumulator left by a single bit position. The RL instruction circulates the msb of the accumula- tor (bit 15) back to the lsb (bit 0) while the RLC instruction includes the Carry (C) flag in the circular left shift.

Status Flags: C (for RLC only), S, Z (for RLC only)

RL Operation:



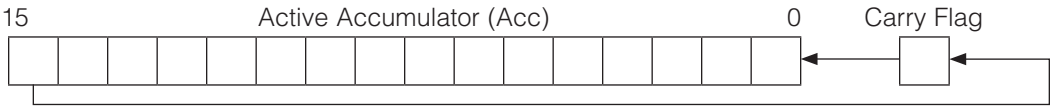
Acc.[15:1] ← Acc.[14:0]; Acc.0 ← Acc.15

Encoding:

Example(s):

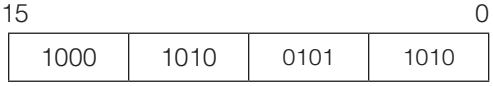
```
                    ; Acc = A345h, S=1, Z=0
RL                  ; Acc = 468Bh, S=0, Z=0
RL                  ; Acc = 8D16h, S=1, Z=0
```

RLC Operation:



Acc.[15:1] ← Acc.[14:0]; Acc.0 ← C; C ← Acc.15

Encoding:



Example(s):

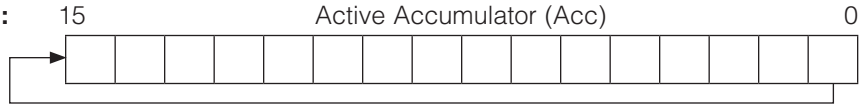
```
                    ; Acc = A345h, C=1, S=1, Z=0
RLC                 ; Acc = 468Bh, C=1, S=0, Z=0
RLC                 ; Acc = 8D17h, C=0, S=1, Z=0
```

RR/RRC — **Rotate Right Accumulator Carry Flag (Ex/In)clusive**

Description: Rotates the active accumulator right by a single bit position. The RR instruction circulates the lsb of the accumula- tor (bit 0) back to the msb (bit 15) while the RRC instruction includes the Carry (C) flag in the circular right shift.

Status Flags: C (for RRC only), S, Z (for RRC only)

RR Operation:



Acc.[14:0] ← Acc.[15:1]; Acc.15 ← Acc.0

Encoding:

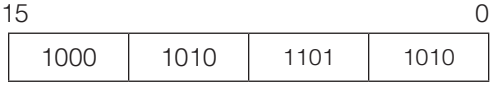
Example(s):

```
                    ; Acc = A345h, S=1, Z=0
RR                  ; Acc = D1A2h, S=1, Z=0
RR                  ; Acc = 68D1h, S=0, Z=0
```

RRC Operation:

Acc.[14:0] ← Acc.[15:1]; Acc.15 ← C; C ← Acc.0

Encoding:



Example(s):

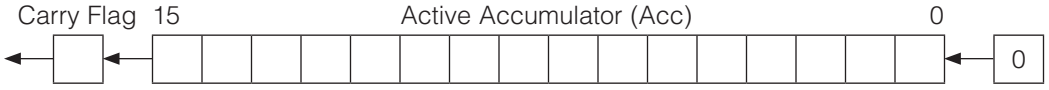
```
                    ; Acc = A345h, C=1, S=1, Z=0
RRC                 ; Acc = D1A2h, C=1, S=1, Z=0
RRC                 ; Acc = E8D1h, C=0, S=1, Z=0
```

SLA/SLA2/SLA4 — **Shift Accumulator Left Arithmetically One, Two, or Four Times**

Description: Shifts the active accumulator left once, twice, or four times respectively for SLA, SLA2, and SLA4. For each shift iter- ation, a 0 is shifted into the lsb, and the msb is shifted into the Carry (C) flag. For signed data, this shifting process effectively retains the sign orientation of the data to the point at which overflow/underflow would occur.

Status Flags: C, S, Z

SLA Operation:



C ← Acc.15; Acc.[15:1] ← Acc.[14:0]; Acc.0 ← 0

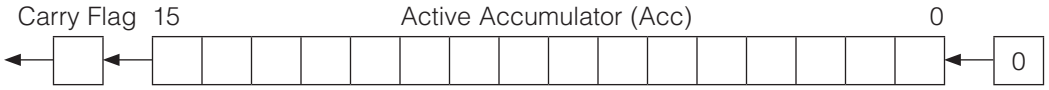
Encoding:

15			0
1000	1010	0010	1010

Example(s):

```
                    ; Acc = E345h, C=0, S=1, Z=0
SLA                 ; Acc = C68h, C=1, S=1, Z=0
SLA                 ; Acc = 8D14h, C=1, S=1, Z=0
```

SLA2 Operation:



C ← Acc.14; Acc.[15:2] ← Acc.[13:0]; Acc.[1:0] ← 0

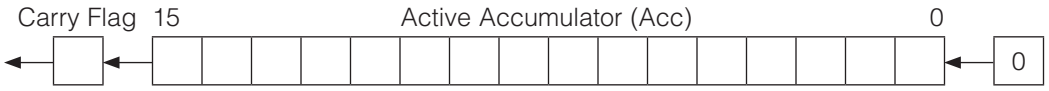
Encoding:

15			0
1000	1010	0011	1010

Example(s):

```
                    ; Acc = E345h, C=0, S=1, Z=0
SLA2                ; Acc = 8D14h, C=1, S=1, Z=0
```

SLA4 Operation:



C ← Acc.12; Acc.[15:4] ← Acc.[11:0]; Acc.[3:0] ← 0

Encoding:

15			0
1000	1010	0110	1010

Example(s):

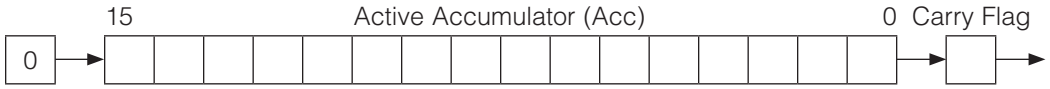
```
                    ; Acc = E345h, C=0, S=1, Z=0
SLA4                ; Acc = 3450h, C=0, S=0, Z=0
```

SR/SRA/SRA2/SRA4 — **Shift Accumulator Right/Shift Accumulator Right Arithmetically One, Two, or Four Times**

Description: Shifts the active accumulator right once for the SR, SRA instructions and 2 or 4 times, respectively, for the SRA2, SRA4 instructions. The SR instruction shifts a 0 into the accumulator msb while the SRA, SRA2, and SRA4 instruc- tions effectively shift a copy of the current msb into the accumulator, thereby preserving any sign orientation. For each shift iteration, the accumulator lsb is shifted into the Carry (C) flag.

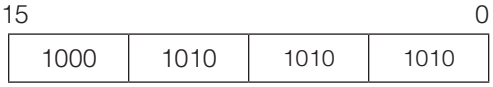
Status Flags: C, S (changes for SR only), Z

SR Operation:



Acc.15 ← 0; Acc.[14:0] ← Acc.[15:1]; C ← Acc.0

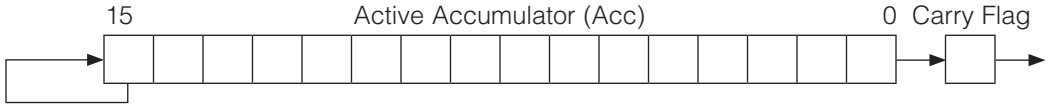
Encoding:



Example(s):

```
                    ; Acc = A345h, C=1, S=1, Z=0
SR                  ; Acc = 51A2h, C=1, S=0, Z=0
SR                  ; Acc = 28D1h, C=0, S=0, Z=0
```

SRA Operation:

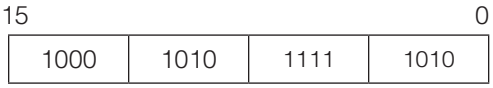


Acc.[14:0] ← Acc.[15:1]

Acc.15 ← Acc.15

C ← Acc.0

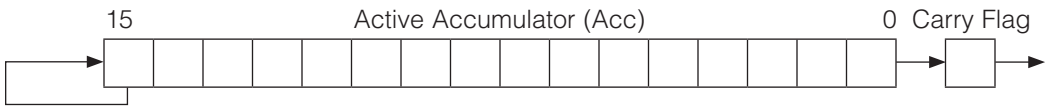
Encoding:



Example(s):

```
                    ; Acc = 0003h, C=0, Z=0
SRA                 ; Acc = 0001h, C=1, Z=0
SRA                 ; Acc = 0000h, C=1, Z=1
```

SRA2 Operation:

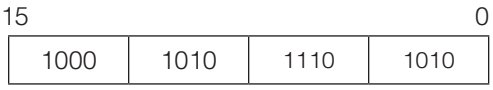


Acc.[13:0] ← Acc.[15:2]

Acc.[15:14] ← Acc.15

C ← Acc.1

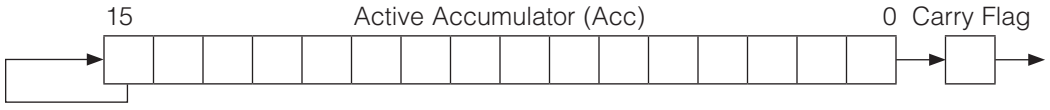
Encoding:



Example(s):

```
                        ; Acc = 0003h, C=0, Z=0
SRA2                    ; Acc = 0000h, C=1, Z=1
```

SRA4 Operation:

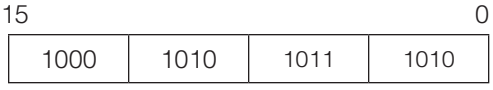


Acc.[11:0] ← Acc.[15:4]

Acc.[15:12] ← Acc.15

C ← Acc.3

Encoding:



Example(s):

```
                        ; Acc = 9878h, C=0, Z=0
SRA4                    ; Acc = F987h, C=1, Z=0
SRA4                    ; Acc = FF98h, C=0, Z=0
```

SUB/SUBB src **Subtract /Subtract with Borrow**

Description: Subtracts the specified src from the active accumulator (Acc) and returns the result back to the active accumula- tor. The SUBB additionally subtracts the borrow (Carry Flag), which may have resulted from previous subtraction. For the complete list of src specifiers, reference the MOVE instruction. Because the source is limited to 8 bits, the PFX[n] register is used to supply the high-byte of data for 16 bit sources.

Status Flags: C, S, Z, OV

SUB Operation: Acc ← Acc - src

Encoding:

15			0
f101	1010	ssss	ssss

Example(s):

```
                    ; Acc = 2345h to start, A[1]= 1250h
SUB A[1]            ; Acc = 10F5h, C=0, S=0, Z=0, OV=0
SUB A[1]            ; Acc = FEA5h, C=1, S=1, Z=0, OV=0
SUB A[2]            ; A[2] =7FFFh
                    ; → Acc = 7EA6h; C=0, S=0, Z=0, OV=1
```

SUBB Operation: Acc ← Acc - (src + C)

Encoding:

15			0
f111	1010	ssss	ssss

Example(s):

```
                    ; Acc = 2345h, A[1]= 1250h, C=1
SUBB A[1]           ; Acc = 10F4h, C=0, S=0, Z=0
SUBB A[1]           ; Acc = FEA4h, C=1, S=1, Z=0
```

Special Notes: The active accumulator (Acc) is not allowed as the src for these operations.

XCH **Exchange Accumulator Bytes**

Description: Exchanges the upper and lower bytes of the active accumulator.

Status Flags: S

Operation: Acc.[15:8] ← Acc.[7:0]

Acc.[7:0] ← Acc.[15:8]

Encoding:

15			0
1000	1010	1000	1010

Example(s):

```
                    ; Acc = 2345h
XCHN                ; Acc = 4523h
```

XCHN **Exchange Accumulator Nibbles**

Description: Exchanges the upper and lower nibbles in the active accumulator byte(s).

Status Flags: S

Operation: Acc.[7:4] ← Acc.[3:0]

Acc.[3:0] ← Acc.[7:4]

Acc.[15:12] ← Acc.[11:8]

Acc.[11:8] ← Acc.[15:12]

Encoding:

15			0
1000	1010	0111	1010

Example(s):

```
                    ; Acc = 2345h
XCHN                ; Acc = 3254h
```

XOR src **Logical XOR**

Description: Performs a logical-XOR between the active accumulator (Acc or A[AP]) and the specified src data. For the complete list of src specifiers, reference the MOVE instruction. Because the source is limited to 8 bits, the PFX[n] register is used to supply the high-byte of data for 16 bit sources.

Status Flags: S, Z

Operation: Acc ← Acc XOR src

Encoding:

15			0
f011	1010	ssss	ssss

Example(s):

```
                        ; Acc = 2345h
XOR A[2]                ; A[2]=0F0Fh; Acc ← 2C4Ah
```

Special Notes: The active accumulator (Acc) is not allowed as the src for this operation.

XOR Acc.<b> **Logical XOR Carry Flag with Accumulator Bit**

Description: Performs a logical-XOR between the Carry (C) status flag and a specified bit of the active accumulator (Acc.<b>) and returns the result to the Carry.

Status Flags: C

Operation: C ← C XOR Acc.<b>

Encoding:

15			0
1011	1010	bbbb	1010

Example(s):

```
                        ; Acc = 2345h, C=1 at start
XOR Acc.1               ; Acc.1=0 → C=1
XOR Acc.2               ; Acc.2=1 → C=0
```

SECTION 21: UTILITY ROM

This section contains the following information:

LIST OF FIGURES

LIST OF TABLES

SECTION 21: UTILITY ROM

21.1 Overview

The MAX31782 utility ROM includes routines that provide the following functions to application software:

- In-application programming routines for flash memory (program, erase, mass erase)
- Single word/byte copy and buffer copy routines for lookup tables in flash
- Entry into stop mode

To provide backwards compatibility among different versions of the utility ROM, a function address table is included that contains the entry points for all user-callable functions. With this table, user code can determine the entry point for a given function as follows:

1) Read the location of the function address table from address 0800Dh in the utility ROM.
2) The entry points for each function listed below are contained in the function address table, one word per function, in the order given by their function numbers.

For example, the entry point for the UROM_flashEraseAll function can be determined by the following procedure.

1) functionTable = romMemory[800Dh]
2) flashEraseAllEntry = romMemory[functionTable + 2]

It is also possible to call utility ROM functions directly, using the entry points given in Table 21-1. Calling a function directly will provide faster code execution.

Table 21-1. MAX31782 Utility ROM Functions

INDEX	FUNCTION NAME	ENTRY POINT	SUMMARY
0	UROM_flashWrite	8449h	Programs a single word of flash memory.
1	UROM_flashErasePage	846Ch	Erases (programs to FFFFh) a 512-byte (256-word) sector of flash memory.
2	UROM_flashEraseAll	8482h	Erases (programs to FFFFh) all flash memory.
3	UROM_moveDP0	8491h	Reads a byte/word at DP[0].
4	UROM_moveDP0inc	8494h	Reads a byte/word at DP[0], then increments DP[0].
5	UROM_moveDP0dec	8497h	Reads a byte/word at DP[0], then decrements DP[0].
6	UROM_moveDP1	849Ah	Reads a byte/word at DP[1].
7	UROM_moveDP1inc	849Dh	Reads a byte/word at DP[1], then increments DP[0].
8	UROM_moveDP1dec	84A0h	Reads a byte/word at DP[1], then decrements DP[0].
9	UROM_moveBP	84A3h	Reads a byte/word at BP[OFFS].
10	UROM_moveBPinc	84A6h	Reads a byte/word at BP[OFFS], then increments OFFS.
11	UROM_moveBPdec	84A9h	Reads a byte/word at BP[OFFS], then decrements OFFS.
12	UROM_copyBuffer	84ACh	Copies LC[0] bytes/words (up to 255) from DP[0] to BP[OFFS].
13	UROM_stopMode	84B2h	Enters stop mode.

21.2 In-Application Programming Functions

21.2.1 UROM_flashWrite

Function	UROM_flashWrite
Summary	Programs a single word of flash memory
Inputs	A[0]: Word address in program flash memory to write. A[1]: Value to write to flash memory.
Outputs	Carry: Set on error and cleared on success
Destroys	PSF, LC[1]

Notes:

- This function uses two stack levels to save and restore values.
- If the watchdog reset function is active, it should be disabled before calling this function.
- Interrupts are disabled while in this function.
- If the flash location has already been programmed to a non-FFFF value, this function returns with an error (Carry set). In order to reprogram a flash location, the location must first be erased by calling UROM_flashErasePage or UROM_flashEraseAll.

21.2.2 UROM_flashErasePage

Function	UROM_flashErasePage
Summary	Erases (programs to FFFFh) a 512-byte page of flash memory.
Inputs	A[0]: Word address located in the page to be erased. (The page number is the upper 8 bits of A[0].)
Outputs	Carry: Set on error and cleared on success.
Destroys	PSF, LC[1], GR, AP, APC, A[0]

Notes:

- If the watchdog reset function is active, it should be disabled before calling this function.
- Interrupts are disabled while in this function.
- When calling this function from flash, care should be taken that the return address is not in the page which is being erased.

21.2.3 UROM_flashEraseAll

Function	UROM_flashEraseAll
Summary	Erases (programs to FFFFh) all locations in flash memory
Inputs	None
Outputs	Carry: Set on error and cleared on success.
Destroys	PSF, GR, LC[1], LC[0], AP, APC, A[0]

Notes:

- If the watchdog reset function is active, it should be disabled before calling this function.
- Interrupts are disabled while in this function.
- This function can only be called by code running from the RAM. Attempting to call this function while running from the flash results in an error.

21.3 Data Transfer Functions

The MAX31782 cannot access data from the same memory segment that is currently being used for instructions. For example, when instructions are executing from FLASH, data in FLASH cannot be accessed. The following utility ROM functions can be used to transfer data from one memory segment to another. For example, if data in FLASH needs to be copied to SRAM, one of these ROM functions can be called to do this transfer. This is useful when code is executing from FLASH and access to lookup tables or non-volatile data that is stored in FLASH is required. These functions can also be used by code running from SRAM to read data that is stored in SRAM.

Since these functions are executed from utility ROM, addresses must be specified correctly to point to the intended memory segments. When executing from utility ROM, the memory map is illustrated in Figure 21-1. For example, data located at word address 0100h in the FLASH must be accessed at word address 8100h (or byte address 8200h) when using any of the functions listed in the following sections.

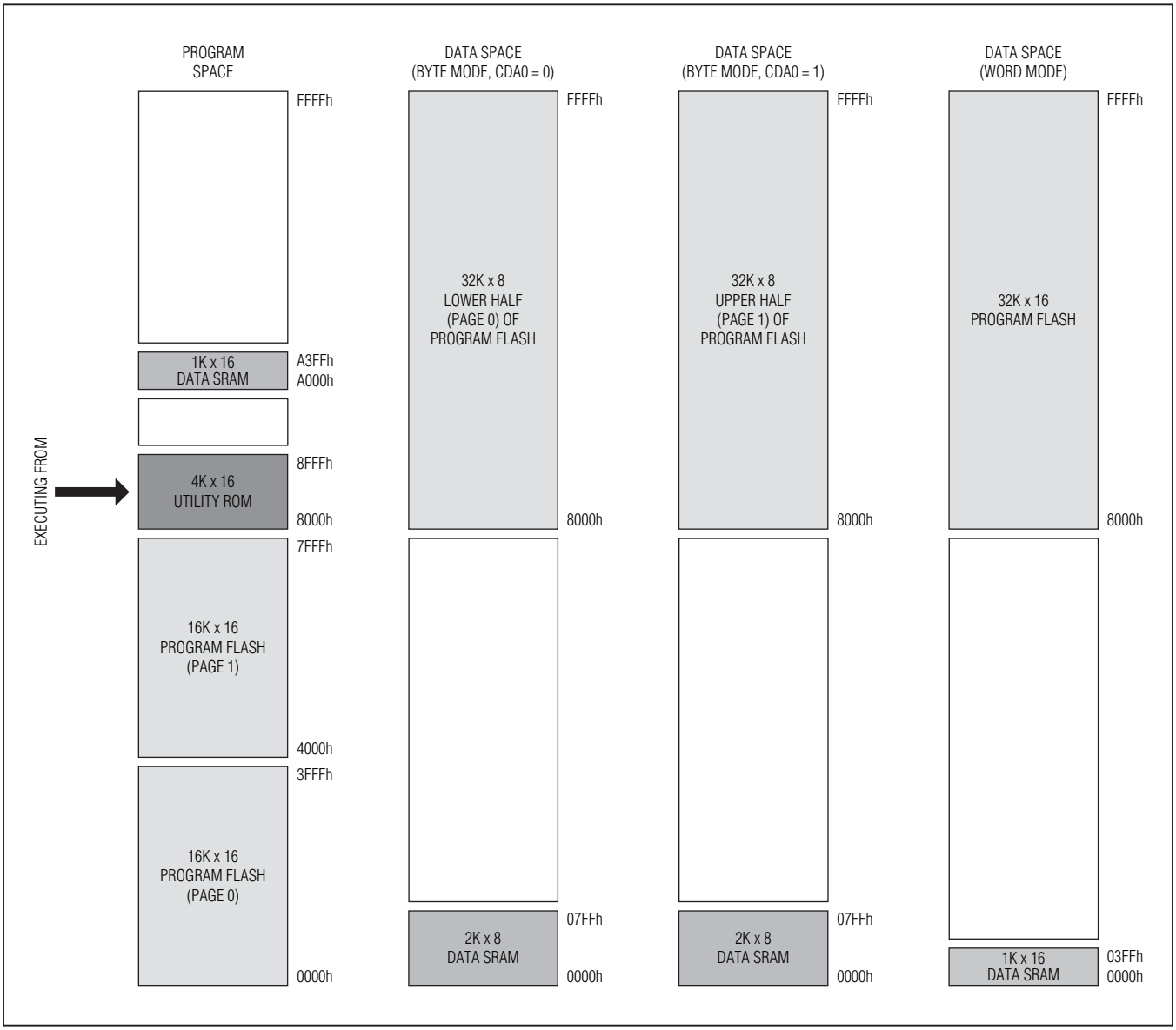


Figure 21-1. Memory Map when Executing from Utility ROM

21.3.1 UROM_moveDP0

Function	UROM_moveDP0
Summary	Reads the byte/word value pointed to by DP[0].
Inputs	DP[0]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure DP[0] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects DP[0] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @DP[0]

21.3.2 UROM_moveDP0inc

Function	UROM_moveDP0inc
Summary	Reads the byte/word value pointed to by DP[0], then increments DP[0].
Inputs	DP[0]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read. DP[0] is incremented.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure DP[0] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects DP[0] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @DP[0]++

21.3.3 UROM_moveDP0dec

Function	UROM_moveDP0dec
Summary	Reads the byte/word value pointed to by DP[0], then decrements DP[0].
Inputs	DP[0]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read. DP[0] is decremented.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure DP[0] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects DP[0] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @DP[0]--

21.3.4 UROM_moveDP1

Function	UROM_moveDP1
Summary	Reads the byte/word value pointed to by DP[1].
Inputs	DP[1]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure DP[1] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects DP[1] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @DP[1]

21.3.5 UROM_moveDP1inc

Function	UROM_moveDP1inc
Summary	Reads the byte/word value pointed to by DP[1], then increments DP[1].
Inputs	DP[1]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read. DP[1] is incremented.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure DP[1] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects DP[1] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @DP[1]++

21.3.6 UROM_moveDP1dec

Function	UROM_moveDP1dec
Summary	Reads the byte/word value pointed to by DP[1], then decrements DP[1].
Inputs	DP[1]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read. DP[1] is decremented.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure DP[1] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects DP[1] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @DP[1]--

21.3.7 UROM_moveBP

Function	UROM_moveBP
Summary	Reads the byte/word value pointed to by BP[OFFS].
Inputs	BP[OFFS]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read
Destroys	None.

Notes:

- Before calling this function, DPC should be set appropriately to configure BP[OFFS] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects BP[OFFS] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @BP[OFFS]

21.3.8 UROM_moveBPinc

Function	UROM_moveBPinc
Summary	Reads the byte/word value pointed to by BP[OFFS], then increments OFFS.
Inputs	BP[OFFS]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read. OFFS is incremented.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure BP[OFFS] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects BP[OFFS] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @BP[OFFS++]

21.3.9 UROM_moveBPdec

Function	UROM_moveBPdec
Summary	Reads the byte/word value pointed to by BP[OFFS], then decrements OFFS.
Inputs	BP[OFFS]: Address to read from data space (include 8000h offset if reading from flash).
Outputs	GR: Data byte/word read. OFFS is decremented.
Destroys	None

Notes:

- Before calling this function, DPC should be set appropriately to configure BP[OFFS] for byte or word mode.
- The address passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects BP[OFFS] as the data pointer before reading the byte/word value.
- Implemented as: move GR, @BP[OFFS--]

21.3.10 UROM_copyBuffer

Function	UROM_copyBuffer
Summary	LC[0] bytes/words (up to 256) from DP[0] to BP[OFFS].
Inputs	DP[0]: Starting address to copy from. BP[OFFS]: Starting address to copy to. LC[0]: Number of bytes/words to copy.
Outputs	OFFS is incremented by LC[0]. DP[0] is incremented by LC[0].
Destroys	LC[0]

Notes:

- This function can be used to copy from program flash to data RAM, or from one part of data RAM to another. It cannot be used to copy data into flash memory; the UROM_writeFlash function should be used for this purpose.
- Before calling this function, DPC should be set appropriately to configure DP[0] and BP[OFFS] for byte or word mode. Both DP[0] and BP[OFFS] should be configured to the same mode (byte or word) for correct buffer copying.
- The addresses passed to this function should be based on the data memory mapping for the utility ROM, as shown in Figure 21-1. When a byte mode address is used, CDA0 must be set appropriately to access either the upper or lower half of program flash memory.
- This function automatically selects the data pointers before reading the byte/word values.

21.3.11 UROM_stopMode

Function	UROM_stopMode
Summary	Enters stop mode.
Inputs	None
Outputs	None
Destroys	None

Notes:

- Stop mode should normally be called from user code.

21.4 Utility ROM Examples

21.4.1 Reading Constant Word Data from Flash

```
UROM_moveDP0inc equ 08494h
move DPC, #1Ch                  ; Set all pointers to word mode
move DP[0], #(table + 8000h)    ; Point to address of data as viewed in the Utility ROM
                                  memory map
lcall #UROM_moveDP0inc
move A[0], GR                   ; A[0] = 1111h
lcall #UROM_moveDP0inc
move A[1], GR                   ; A[1] = 2222h
lcall #UROM_moveDP0inc
move A[2], GR                   ; A[0] = 3333h
lcall #UROM_moveDP0inc
move A[3], GR                   ; A[1] = 4444h
sjump $
org 0100h
table:
      dw 1111h, 2222h, 3333h, 4444h
```

21.4.2 Reading Constant Byte Data from Flash (Indirect Function Call)

```
INDX_moveDP0inc equ 4
move DPC, #1Ch                  ; Set all pointers to word mode
move DP[0], #800Dh              ; Fetch location of function table from Utility ROM
move BP, @DP[0]                 ; Set base pointer to function table location
move Offs, #INDX_moveDP0inc     ; Set offset to moveDP0inc entry in table
move A[7], @BP[Offs]            ; Get address of moveDP0inc function
move DPC, #00h                  ; Set all pointers to byte mode
move DP[0], #((table * 2) + 8000h); Point to address of data as viewed in the Utility ROM memory
                                  map and convert to byte mode pointer
lcall A[7]                      ; moveDP0inc
move A[0], GR                   ; A[0] = 34h
lcall A[7]                      ; moveDP0inc
move A[1], GR                   ; A[1] = 12h
lcall A[7]                      ; moveDP0inc
move A[2], GR                   ; A[2] = 78h
lcall A[7]                      ; moveDP0inc
move A[3], GR                   ; A[3] = 56h
sjump $
org 0100h
table:
      dw 1234h, 5678h
```

REVISION HISTORY

REVISION NUMBER	REVISION DATE	DESCRIPTION	PAGES CHANGED
0	8/11	Initial release	—