MSP430 Addressing Modes

Ad	As	d/s	Register	Syntax	Description
0	00	*ds*	*n ≠ 3*	**R*n***	Register direct. The operand is the contents of R*n*. A_d=0
1	01	*ds*	*n ≠ 0, 2, 3*	**x(R*n*)**	Indexed. The operand is in memory at address R*n*+*x*.
-	10	*s*	*n ≠ 0, 2, 3*	**@R*n***	Register indirect. The operand is in memory at the address held in R*n*.
-	11	*s*	*n ≠ 0, 2, 3*	**@R*n*+**	Indirect auto-increment. As above, then the register is incremented by 1 or 2.
				Addressing modes using R0 (PC)	
1	01	ds	0 (PC)	***LABEL***	Symbolic. x(PC) The operand is in memory at address PC+*x*.
-	11	s	0 (PC)	***#x***	Immediate. @PC+ The operand is the next word in the instruction stream.
				Addressing modes using R2 (SR) and R3 (CG), special-case decoding	
1	01	ds	2 (SR)	***&LABEL***	Absolute. The operand is in memory at address *x*.
-	10	s	2 (SR)	**#4**	Constant. The operand is the constant 4.
-	11	s	2 (SR)	**#8**	Constant. The operand is the constant 8.
-	00	s	3 (CG)	**#0**	Constant. The operand is the constant 0.
-	01	s	3 (CG)	**#1**	Constant. The operand is the constant 1. There is no index word.
-	10	s	3 (CG)	**#2**	Constant. The operand is the constant 2.
-	11	s	3 (CG)	**#–1**	Constant. The operand is the constant –1.

MSP430 Instruction Set

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction
0	**0**	**0**	**1**	**0**	**0**	**opcode**			**B/W**	**As**		**register**				**Single-operand arithmetic**
0	0	0	1	0	0	0	0	0	B/W	As		register				**RRC** Rotate right through carry
0	0	0	1	0	0	0	0	1	0	As		register				**SWPB** Swap bytes
0	0	0	1	0	0	0	1	0	B/W	As		register				**RRA** Rotate right arithmetic
0	0	0	1	0	0	0	1	1	0	As		register				**SXT** Sign extend byte to word
0	0	0	1	0	0	1	0	0	B/W	As		register				**PUSH** Push value onto stack
0	0	0	1	0	0	1	0	1	0	As		register				**CALL** Subroutine call; push PC and move source to PC
0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	**RETI** Return from interrupt; pop SR then pop PC

0	**0**	**1**	**condition**			**10-bit signed offset**	**Conditional jump; PC = PC + 2×offset**
0	0	1	0	0	0	10-bit signed offset	**JNE/JNZ** Jump if not equal/zero
0	0	1	0	0	1	10-bit signed offset	**JEQ/JZ** Jump if equal/zero
0	0	1	0	1	0	10-bit signed offset	**JNC/JLO** Jump if no carry/lower
0	0	1	0	1	1	10-bit signed offset	**JC/JHS** Jump if carry/higher or same
0	0	1	1	0	0	10-bit signed offset	**JN** Jump if negative
0	0	1	1	0	1	10-bit signed offset	**JGE** Jump if greater or equal
0	0	1	1	1	0	10-bit signed offset	**JL** Jump if less
0	0	1	1	1	1	10-bit signed offset	**JMP** Jump (unconditionally)

opcode				**source**	**Ad**	**B/W**	**As**	**destination**	**Two-operand arithmetic**
0	1	0	0	source	Ad	B/W	As	destination	**MOV** Move source to destination
0	1	0	1	source	Ad	B/W	As	destination	**ADD** Add source to destination
0	1	1	0	source	Ad	B/W	As	destination	**ADDC** Add source and carry to destination
0	1	1	1	source	Ad	B/W	As	destination	**SUBC** Subtract source from destination (with carry)
1	0	0	0	source	Ad	B/W	As	destination	**SUB** Subtract source from destination
1	0	0	1	source	Ad	B/W	As	destination	**CMP** Compare (pretend to subtract) source from destination
1	0	1	0	source	Ad	B/W	As	destination	**DADD** Decimal add source to destination (with carry)
1	0	1	1	source	Ad	B/W	As	destination	**BIT** Test bits of source AND destination
1	1	0	0	source	Ad	B/W	As	destination	**BIC** Bit clear (dest &= ~src)
1	1	0	1	source	Ad	B/W	As	destination	**BIS** Bit set (logical OR)
1	1	1	0	source	Ad	B/W	As	destination	**XOR** Exclusive or source with destination
1	1	1	1	source	Ad	B/W	As	destination	**AND** Logical AND source with destination (dest &= src)

Special Registers: PC (Program Counter)=*R0*; SP (Stack Pointer)=*R1*;
SR (Status Register)=*R2*; CG (Constants Generator)=*R3*;

No.	Emulated	Mnemonic	Operand(s)	Description		* changes based on op - not affected V	N	Z	C
1	⊙	ADC(.B)	dst	Add C to destination	dst+C→dst	*	*	*	*
2		ADD(.B)	src,dst	Add source to destination	src+dst→dst	*	*	*	*
3		ADDC(.B)	src,dst	Add source and C to destination	src+dst+C→dst	*	*	*	*
4		AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
5		BIC(.B)	src,dst	Clear bits in destination	not.src .and. dst → dst	-	-	-	-
6		BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	-	-	-	-
7		BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
8	⊙	BR	dst	Branch to destination	dst → PC	-	-	-	-
9		CALL	dst	Call destination	PC+2 → stack, dst → PC	-	-	-	-
10	⊙	CLR(.B)	dst	Clear destination	0 → dst	-	-	-	-
11	⊙	CLRC		Clear C	0→C	-	-	-	0
12	⊙	CLRN		Clear N	0→N	-	0	-	-
13	⊙	CLRZ		Clear Z	0→Z	-	-	0	-
14		CMP(.B)	src,dst	Compare source and destination	dst - src	*	*	*	*
15	⊙	DADC(.B)	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
16		DADD(.B)	src,dst	Add source and C decimally to dst	src + dst + C → dst (decimally)	*	*	*	*
17	⊙	DEC(.B)	dst	Decrement destination	dst - 1 → dst	*	*	*	*
18	⊙	DECD(.B)	dst	Double-decrement destination	dst - 2 → dst	*	*	*	*
19	⊙	DINT		Disable interrupts	0 → GIE	-	-	-	-
20	⊙	EINT		Enable interrupts	1 → GIE	-	-	-	-
21	⊙	INC(.B)	dst	Increment destination	dst +1 → dst	*	*	*	*
22	⊙	INCD(.B)	dst	Double-increment destination	dst+2 → dst	*	*	*	*
23	⊙	INV(.B)	dst	Invert destination	.not.dst → dst	*	*	*	*
24		JC/JHS	label	Jump if C set/Jump if higher or same		-	-	-	-
25		JEQ/JZ	label	Jump if equal/Jump if Z set		-	-	-	-
26		JGE	label	Jump if greater or equal		-	-	-	-
27		JL	label	Jump if less		-	-	-	-
28		JMP	label	Jump	PC + 2 × offset → PC	-	-	-	-
29		JN	label	Jump if N set		-	-	-	-
30		JNC/JLO	label	Jump if C not set/Jump if lower		-	-	-	-
31		JNE/JNZ	label	Jump if not equal/Jump if Z not set		-	-	-	-
32		MOV(.B)	src,dst	Move source to destination	src → dst	-	-	-	-
33	⊙	NOP		No operation		-	-	-	-
34	⊙	POP(.B)	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	-	-	-	-
35		PUSH(.B)	src	Push source onto stack	SP - 2 → SP, src → @SP	-	-	-	-
36	⊙	RET		Return from subroutine	@SP → PC, SP + 2 → SP	-	-	-	-
37		RETI		Return from interrupt		*	*	*	*
38	⊙	RLA(.B)	dst	Rotate left arithmetically		*	*	*	*
39	⊙	RLC(.B)	dst	Rotate left through C		*	*	*	*
40		RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
41		RRC(.B)	dst	Rotate right through C		*	*	*	*
42	⊙	SBC(.B)	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
43	⊙	SETC		Set C	1→C	-	-	-	1
44	⊙	SETN		Set N	1→N	-	1	-	-
45	⊙	SETZ		Set Z	1→Z	-	-	1	-
46		SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
47		SUBC(.B)	src,dst	Subtract source and not(C) from dst	dst + .not.src + C → dst	*	*	*	*
48		SWPB	dst	Swap bytes		-	-	-	-
49		SXT	dst	Extend sign		0	*	*	*
50	⊙	TST(.B)	dst	Test destination	dst+0FFFFh+1	0	*	*	1
51		XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

MSP430 Emulated Instructions

Mnemonic	Operation	Emulation	Description
Arithmetic Instructions			
ADC(.B or .W) dst	**dst+C→dst**	**ADDC(.B or .W) #0,dst**	Add carry to destination
DADC(.B or .W) dst	**dst+C→dst (decimally)**	**DADD(.B or .W) #0,dst**	Decimal add carry to destination
DEC(.B or .W) dst	**dst-1→dst**	**SUB(.B or .W) #1,dst**	Decrement destination
DECD(.B or .W) dst	**dst-2→dst**	**SUB(.B or .W) #2,dst**	Decrement destination twice
INC(.B or .W) dst	**dst+1→dst**	**ADD(.B or .W) #1,dst**	Increment destination
INCD(.B or .W) dst	**dst+2→dst**	**ADD(.B or .W) #2,dst**	Increment destination twice
SBC(.B or .W) dst	**dst+0FFFFh+C→dst** **dst+0FFh→dst**	**SUBC(.B or .W) #0,dst**	Subtract source and borrow /.NOT. carry from dest.

Mnemonic	Operation	Emulation	Description
Logical and Register Control Instructions			
INV(.B or .W) dst	**.NOT.dst→dst**	**XOR(.B or .W) #0(FF)FFh,dst**	Invert bits in destination
RLA(.B or .W) dst	**C←MSB←MSB-1 LSB+1←LSB←0**	**ADD(.B or .W) dst,dst**	Rotate left arithmetically
RLC(.B or .W) dst	**C←MSB←MSB-1 LSB+1←LSB←C**	**ADDC(.B or .W) dst,dst**	Rotate left through carry
Program Flow Control			
BR dst	**dst→PC**	**MOV dst,PC**	Branch to destination
DINT	**0→GIE**	**BIC #8,SR**	Disable (general) interrupts
EINT	**1→GIE**	**BIS #8,SR**	Enable (general) interrupts
NOP	**None**	**MOV #0,R3**	No operation
RET	**@SP→PC SP+2→SP**	**MOV @SP+,PC**	Return from subroutine

Mnemonic	Operation	Emulation	Description
Data Instructions			
CLR(.B or .W) dst	**0→dst**	**MOV(.B or .W) #0,dst**	Clear destination
CLRC	**0→C**	**BIC #1,SR**	Clear carry flag
CLRN	**0→N**	**BIC #4,SR**	Clear negative flag
CLRZ	**0→Z**	**BIC #2,SR**	Clear zero flag
POP(.B or .W) dst	**@SP→temp SP+2→SP temp→dst**	**MOV(.B or .W) @SP+,dst**	Pop byte/word from stack to destination
SETC	**1→C**	**BIS #1,SR**	Set carry flag
SETN	**1→N**	**BIS #4,SR**	Set negative flag
SETZ	**1→Z**	**BIS #2,SR**	Set zero flag
TST(.B or .W) dst	**dst + 0FFFFh + 1 dst + 0FFh + 1**	**CMP(.B or .W) #0,dst**	Test destination

	000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
0xxx																
4xxx																
8xxx																
Cxxx																
1xxx	RRC	RRC.B	SWPB		RRA	RRA.B	SXT		PUSH	PUSH.B	CALL		RETI			
14xx																
18xx																
1Cxx																
20xx	JNE/JNZ															
24xx	JEQ/JZ															
28xx	JNC															
2Cxx	JC															
30xx	JN															
34xx	JGE															
38xx	JL															
3Cxx	JMP															
4xxx	MOV, MOV.B															
5xxx	ADD, ADD.B															
6xxx	ADDC, ADDC.B															
7xxx	SUBC, SUBC.B															
8xxx	SUB, SUB.B															
9xxx	CMP, CMP.B															
Axxx	DADD, DADD.B															
Bxxx	BIT, BIT.B															
Cxxx	BIC, BIC.B															
Dxxx	BIS, BIS.B															
Exxx	XOR, XOR.B															
Fxxx	AND, AND.B															

asm code 0(Rn) = @Rn when source	machine code	opcode	source Reg	Ad (dest)	B/~W	As (Src)	destination	* IF NEEDED Additional Data 1	* IF NEEDED Additional Data 2
mov.w R5, R6	4506	0 1 0 0	5	0	0	0 0	6	-	-
add.b R4, R8	5448	0 1 0 1	4	0	1	0 0	8	-	-
bit.w @R7, R12	B72C	1 0 1 1	7	0	0	1 0	C		
bit.b 0(R7), R15	B76F	1 0 1 1	7	0	1	1 0	F		
bic.b 2(R8), 0(R8)	C8D8 0002 0000	1 1 0 0	8	1	1	0 1	8	0002	0000
bic.w 2(R8), 4(R7)	C897 0002 0004	1 1 0 0	8	1	0	0 1	7		
bis.w #0xAAAA, R11	D03B AAAA	1 1 0 1	0	0	0	1 1	B	AAAA	
rra.b R10	114A	0 0 0 1	0001	0	1	0 0	A		
rrc.w R11	100B	0 0 0 1	0000	0	0	0 0	B		
jmp *LABEL*	3 [11xx xxx xxxx]	0 0 1 1	11xx	x	x	x x	(offset - Δ inst. words)		
cmp R13, 2(R8)	9D88 0002	1 0 0 1	D	1	0	0 0	8	0002	
jnz *LABEL*	2 [00xx xxxx xxxx]	0 0 1 0	00xx	x	x	x x	(offset - Δ inst. words)		
rla.b R10	5A4A								
add.b R10, R10	🡸 (emulated code) 🡹	0 1 0 1	A	0	1	0 0	A		
rlc.w R11	6B0B								
addc.w R11, R11	🡸 (emulated code) 🡹	0 1 1 0	B	0	0	0 0	B		
inv.w R10	E33A								
xor.w #0xFFFF, R10	🡸 (emulated code) 🡹	1 1 1 0	3	0	0	1 1	A		
inc.w R11	531B								
add.w #0x0001, R11	🡸 (emulated code) 🡹	0 1 0 1	3	0	0	0 1	B		

SR Bit — *After the operation is complete, the SR bits are set based on these conditions*

V — oVerflow: "signed" overflow - leading digit (MSB) switched from (+) to (-) or from (-) to (+)

examples in binary

add.w	*inc.w*	*incd.b*	*add.b*
0100 0000 0000 0000	0111 1111 1111 1111	0111 1110	1011 0000
+0111 0100 0010 1011			+ 1000 1000
1011 0100 0010 1011	1000 0000 0000 0000	0000 0000 1000 0000	0000 0000 0011 1000

N — Negative: The leading digit (MSB) is 1 - the number "negative" if you are using signed numbers

examples in binary

cmp.w	*inv.w*	*dec.b*	*sub.b*
0000 0010 0101 0101	0101 0000 1111 1010	0000 0000 0000 0000	0000 0101
1111 1100 1110 0000			– 0000 0111
1111 1111 0011 0101	1010 1111 0000 0101	0000 0000 1111 1111	0000 0000 1111 1110

Z — Zero: All the bits are 0

examples in binary

sub.w	*rra.w*	*and.b*	*xor.b*
0000 0010 0101 0101	0000 0000 0000 0001	1010 1010	0010 1101
–0000 0010 0101 0101		0101 0101	0010 1101
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0000 0000

C — Carry: "unsigned" overflow - there was not enough space - a digit got knocked off ***either end***

examples in binary

add.w	*rra.w*	*rlc.b*	*rrc.b*
1000 0000 0000 0000	0000 0100 0000 0001	0000 0000 1000 0010	0000 0000 1101 1011
1000 0000 0000 0001			
(1) 0000 0000 0000 0001	0000 0010 0000 0000 (1)	(1) 0000 0000 0000 0100	0000 0000 0110 1101 (1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C

12.3.1 TACTL, Timer_A Control Register

15	14	13	12	11	10	9	8
Unused						TASSELx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)

Unused Bits 15-10 Unused

TASSELx Bits 9-8 Timer_A clock source select
- 00 TACLK
- 01 ACLK
- 10 SMCLK
- 11 INCLK (INCLK is device-specific and is often assigned to the inverted TBCLK) (see the device-specific data sheet)

IDx Bits 7-6 Input divider. These bits select the divider for the input clock.
- 00 /1
- 01 /2
- 10 /4
- 11 /8

MCx Bits 5-4 Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.
- 00 Stop mode: the timer is halted.
- 01 Up mode: the timer counts up to TACCR0.
- 10 Continuous mode: the timer counts up to 0FFFFh.
- 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h.

Unused Bit 3 Unused

TACLR Bit 2 Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.

TAIE Bit 1 Timer_A interrupt enable. This bit enables the TAIFG interrupt request.
- 0 Interrupt disabled
- 1 Interrupt enabled

TAIFG Bit 0 Timer_A interrupt flag
- 0 No interrupt pending
- 1 Interrupt pending

Table 12-2. Output Modes

OUTMODx	Mode	Description
000	Output	The output signal OUTx is defined by the OUTx bit. The OUTx signal updates immediately when OUTx is updated.
001	Set	The output is set when the timer *counts* to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected and affects the output.
010	Toggle/Reset	The output is toggled when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
011	Set/Reset	The output is set when the timer *counts* to the TACCRx value. It is reset when the timer *counts* to the TACCR0 value.
100	Toggle	The output is toggled when the timer *counts* to the TACCRx value. The output period is double the timer period.
101	Reset	The output is reset when the timer *counts* to the TACCRx value. It remains reset until another output mode is selected and affects the output.
110	Toggle/Set	The output is toggled when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.
111	Reset/Set	The output is reset when the timer *counts* to the TACCRx value. It is set when the timer *counts* to the TACCR0 value.

12.2.5.2 Output Example — Timer in Up Mode

The OUTx signal is changed when the timer *counts* up to the TACCRx value, and rolls from TACCR0 to zero, depending on the output mode. An example is shown in Figure 12-12 using TACCR0 and TACCR1.

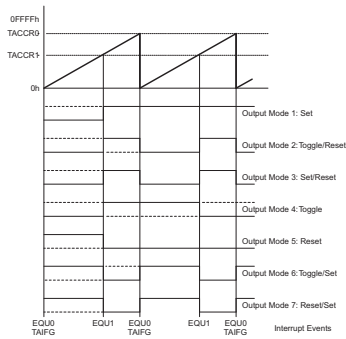


Figure 12-12. Output Example—Timer in Up Mode

Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	-
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC
	Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
	Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
	Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
	Port Select	P1SEL	026h	Read/write	Reset with PUC
	Port Select 2	P1SEL2	041h	Read/write	Reset with PUC
	Resistor Enable	P1REN	027h	Read/write	Reset with PUC

12.3.4 TACCTLx, Capture/Compare Control Register

15	14	13	12	11	10	9	8
CMx		CCISx		SCS	SCCI	Unused	CAP
rw-(0)	rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	r0	rw-(0)

7	6	5	4	3	2	1	0
OUTMODx			CCIE	CCI	OUT	COV	CCIFG
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r	rw-(0)	rw-(0)	rw-(0)

CMx Bit 15-14 Capture mode
- 00 No capture
- 01 Capture on rising edge
- 10 Capture on falling edge
- 11 Capture on both rising and falling edges

CCISx Bit 13-12 Capture/compare input select. These bits select the TACCRx input signal. See the device-specific data sheet for specific signal connections.
- 00 CCIxA
- 01 CCIxB
- 10 GND
- 11 V_{CC}

SCS Bit 11 Synchronize capture source. This bit is used to synchronize the capture input signal with the timer clock.
- 0 Asynchronous capture
- 1 Synchronous capture

SCCI Bit 10 Synchronized capture/compare input. The selected CCI input signal is latched with the EQUx signal and can be read via this bit

Unused Bit 9 Unused. Read only. Always read as 0.

CAP Bit 8 Capture mode
- 0 Compare mode
- 1 Capture mode

OUTMODx Bits 7-5 Output mode. Modes 2, 3, 6, and 7 are not useful for TACCR0, because EQUx = EQU0.
- 000 OUT bit value
- 001 Set
- 010 Toggle/reset
- 011 Set/reset
- 100 Toggle
- 101 Reset
- 110 Toggle/set
- 111 Reset/set

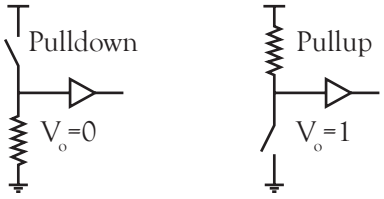
CCIE Bit 4 Capture/compare interrupt enable. This bit enables the interrupt request of the corresponding CCIFG flag.
- 0 Interrupt disabled
- 1 Interrupt enabled

CCI Bit 3 Capture/compare input. The selected input signal can be read by this bit.

OUT Bit 2 Output. For output mode 0, this bit directly controls the state of the output.
- 0 Output low
- 1 Output high

COV Bit 1 Capture overflow. This bit indicates a capture overflow occurred. COV must be reset with software.
- 0 No capture overflow occurred
- 1 Capture overflow occurred

CCIFG Bit 0 Capture/compare interrupt flag
- 0 No interrupt pending
- 1 Interrupt pending



SCG1	SCG0	OSCOFF	CPUOFF	Mode	CPU and Clocks Status
0	0	0	0	Active	CPU is active, all enabled clocks are active
0	0	0	1	LPM0	CPU, MCLK are disabled, SMCLK, ACLK are active
0	1	0	1	LPM1	CPU, MCLK are disabled. DCO and DC generator are disabled if the DCO is not used for SMCLK. ACLK is active.
1	0	0	1	LPM2	CPU, MCLK, SMCLK, DCO are disabled. DC generator remains enabled. ACLK is active.
1	1	0	1	LPM3	CPU, MCLK, SMCLK, DCO are disabled. DC generator disabled. ACLK is active.
1	1	1	1	LPM4	CPU and all clocks disabled

8.2.7 P1 and P2 Interrupts

Each pin in ports P1 and P2 have interrupt capability, configured with the PxIFG, PxIE, and PxIES registers. All P1 pins source a single interrupt vector, and all P2 pins source a different single interrupt vector. The PxIFG register can be tested to determine the source of a P1 or P2 interrupt.

8.2.7.1 Interrupt Flag Registers P1IFG, P2IFG

Each PxIFGx bit is the interrupt flag for its corresponding I/O pin and is set when the selected input signal edge occurs at the pin. All PxIFGx interrupt flags request an interrupt when their corresponding PxIE bit and the GIE bit are set. Each PxIFG flag must be reset with software. Software can also set each PxIFG flag, providing a way to generate a software initiated interrupt.

Bit = 0: No interrupt is pending
Bit = 1: An interrupt is pending

Only transitions, not static levels, cause interrupts. If any PxIFGx flag becomes set during a Px interrupt service routine, or is set after the RETI instruction of a Px interrupt service routine is executed, the set PxIFGx flag generates another interrupt. This ensures that each transition is acknowledged.

NOTE: PxIFG Flags When Changing PxOUT or PxDIR

Writing to P1OUT, P1DIR, P2OUT, or P2DIR can result in setting the corresponding P1IFG or P2IFG flags.

8.2.7.2 Interrupt Edge Select Registers P1IES, P2IES

Each PxIES bit selects the interrupt edge for the corresponding I/O pin.

Bit = 0: The PxIFGx flag is set with a low-to-high transition
Bit = 1: The PxIFGx flag is set with a high-to-low transition

8.2.7.3 Interrupt Enable P1IE, P2IE

Each PxIE bit enables the associated PxIFG interrupt flag.

Bit = 0: The interrupt is disabled.
Bit = 1: The interrupt is enabled.

8.1 Digital I/O Introduction

MSP430 devices have up to eight digital I/O ports implemented, P1 to P8. Each port has up to eight I/O pins. Every I/O pin is individually configurable for input or output direction, and each I/O line can be individually read or written to.

Ports P1 and P2 have interrupt capability. Each interrupt for the P1 and P2 I/O lines can be individually enabled and configured to provide an interrupt on a rising edge or falling edge of an input signal. All P1 I/O lines source a single interrupt vector, and all P2 I/O lines source a different, single interrupt vector.

The digital I/O features include:

- Independently programmable individual I/Os
- Any combination of input or output
- Individually configurable P1 and P2 interrupts
- Independent input and output data registers
- Individually configurable pullup or pulldown resistors
- Individually configurable pin-oscillator function (some MSP430 devices)

NOTE: MSP430G22x0 : These devices feature digital I/O pins P1.2, P1.5, P1.6 and P1.7. The GPIOs P1.0, P1.1, P1.3, P1.4, P2.6, and P2.7 are implemented on this device but not available on the device pin-out. To avoid floating inputs, these GPIOs, these digital I/Os should be properly initialized by running a start-up code. See initialization code below:
```
mov.b #0x1B, P1REN; ; Terminate unavailable Port1 pins properly ; Config as Input with
pull-down enabled
xor.b #0x20, BCSCTL3; ; Select VLO as low freq clock
```
The initialization code configures GPIOs P1.0, P1.1, P1.3, and P1.4 as inputs with pull-down resistor enabled (that is, P1REN.x = 1) and GPIOs P2.6 and P2.7 are terminated by selecting VLOCLK as ACLK – see the Basic Clock System chapter for details. The register bits of P1.0, P1.1, P1.3, and P1.4 in registers P1OUT, P1DIR, P1IFG, P1IE, P1IES, P1SEL and P1REN should not be altered after the initialization code is executed. Also, all Port2 registers are should not be altered.

8.2 Digital I/O Operation

The digital I/O is configured with user software. The setup and operation of the digital I/O is discussed in the following sections.

8.2.1 Input Register PxIN

Each bit in each PxIN register reflects the value of the input signal at the corresponding I/O pin when the pin is configured as I/O function.

Bit = 0: The input is low
Bit = 1: The input is high

NOTE: Writing to Read-Only Registers PxIN

Writing to these read-only registers results in increased current consumption while the write attempt is active.

8.2.2 Output Registers PxOUT

Each bit in each PxOUT register is the value to be output on the corresponding I/O pin when the pin is configured as I/O function, output direction, and the pullup/down resistor is disabled.

Bit = 0: The output is low
Bit = 1: The output is high

If the pin's pullup/pulldown resistor is enabled, the corresponding bit in the PxOUT register selects pullup or pulldown.

Bit = 0: The pin is pulled down
Bit = 1: The pin is pulled up

8.2.3 Direction Registers PxDIR

Each bit in each PxDIR register selects the direction of the corresponding I/O pin, regardless of the selected function for the pin. PxDIR bits for I/O pins that are selected for other functions must be set as required by the other function.

Bit = 0: The port pin is switched to input direction
Bit = 1: The port pin is switched to output direction

8.2.4 Pullup/Pulldown Resistor Enable Registers PxREN

Each bit in each PxREN register enables or disables the pullup/pulldown resistor of the corresponding I/O pin. The corresponding bit in the PxOUT register selects if the pin is pulled up or pulled down.

Bit = 0: Pullup/pulldown resistor disabled
Bit = 1: Pullup/pulldown resistor enabled

8.2.5 Function Select Registers PxSEL and PxSEL2

Port pins are often multiplexed with other peripheral module functions. See the device-specific data sheet to determine pin functions. Each PxSEL and PxSEL2 bit is used to select the pin function - I/O port or peripheral module function.

Table 8-1. PxSEL and PxSEL2

PxSEL2	PxSEL	Pin Function
0	0	I/O function is selected.
0	1	Primary peripheral module function is selected.
1	0	Reserved. See device-specific data sheet.
1	1	Secondary peripheral module function is selected.

Setting PxSELx = 1 does not automatically set the pin direction. Other peripheral module functions may require the PxDIRx bits to be configured according to the direction needed for the module function. See the pin schematics in the device-specific data sheet.

NOTE: Setting PxREN = 1 When PxSEL = 1

On some I/O ports on the MSP430F261x and MSP430F2416/7/8/9, enabling the pullup/pulldown resistor (PxREN = 1) while the module function is selected (PxSEL = 1) does not disable the logic output driver. This combination is not recommended and may result in unwanted current flow through the internal resistor. See the device-specific data sheet pin schematics for more information.

```
;Output ACLK on P2.0 on MSP430F21x1
  BIS.B   #01h,&P2SEL   ; Select ACLK function for pin
  BIS.B   #01h,&P2DIR   ; Set direction to output *Required*
```

NOTE: P1 and P2 Interrupts Are Disabled When PxSEL = 1

When any P1SELx or P2SELx bit is set, the corresponding pin's interrupt function is disabled. Therefore, signals on these pins will not generate P1 or P2 interrupts, regardless of the state of the corresponding P1IE or P2IE bit.

Port P1 input, P1IN: reading returns the logical values on the inputs if they are configured for digital input/output. This register is read-only and volatile. It does not need to be initialized because its contents are determined by the external signals.

Port P1 output, P1OUT: writing sends the value to be driven to each pin if it is configured as a digital output. If the pin is not currently an output, the value is stored in a buffer and appears on the pin if it is later switched to be an output. This register is not initialized and you should therefore write to P1OUT *before* configuring the pin for output.

Port P1 direction, P1DIR: clearing a bit to 0 configures a pin as an input, which is the default in most cases. Writing a 1 switches the pin to become an output. This is for digital input and output; the register works differently if other functions are selected using P1SEL.

Port P1 resistor enable, P1REN: setting a bit to 1 activates a pull-up or pull-down resistor on a pin. Pull-ups are often used to connect a switch to an input as in the section "Read Input from a Switch" on page 80. The resistors are inactive by default (0). When the resistor is enabled (1), the corresponding bit of the P1OUT register selects whether the resistor pulls the input up to V_{CC} (1) or down to V_{SS} (0).

Port P1 selection, P1SEL: selects either digital input/output (0, default) or an alternative function (1). Further registers may be needed to choose the particular function.

Port P1 interrupt enable, P1IE: enables interrupts when the value on an input pin changes. This feature is activated by setting appropriate bits of P1IE to 1. Interrupts are off (0) by default. The whole port shares a single interrupt vector although pins can be enabled individually.

Port P1 interrupt edge select, P1IES: can generate interrupts either on a positive edge (0), when the input goes from low to high, or on a negative edge from high to low (1). It is not possible to select interrupts on both edges simultaneously but this is not a problem because the direction can be reversed after each transition. Care is needed if the direction is changed while interrupts are enabled because a spurious interrupt may be generated. This register is not initialized and should therefore be set up before interrupts are enabled.

Port P1 interrupt flag, P1IFG: a bit is set when the selected transition has been detected on the input. In addition, an interrupt is requested if it has been enabled. These bits can also be set by software, which provides a mechanism for generating a software interrupt (SWI).

6.7 What Happens when an Interrupt Is Requested?

A lengthy chain of operations lies between the cause of a maskable interrupt and the start of its ISR. It starts when a flag bit is set in the module when the condition for an interrupt occurs. For example, TAIFG is set when the counter TAR returns to 0. This is passed to the logic that controls interrupts if the corresponding enable bit is also set, TAIE in this case. The request for an interrupt is finally passed to the CPU if the GIE bit is set. Hardware then performs the following steps to launch the ISR:

1. Any currently executing instruction is completed if the CPU was active when the interrupt was requested. MCLK is started if the CPU was off.
2. The PC, which points to the next instruction, is pushed onto the stack.
3. The SR is pushed onto the stack.
4. The interrupt with the highest priority is selected if multiple interrupts are waiting for service.
5. The interrupt request flag is cleared automatically for vectors that have a single source. Flags remain set for servicing by software if the vector has multiple sources, which applies to the example of TAIFG.
6. The SR is cleared, which has two effects. First, further maskable interrupts are disabled because the GIE bit is cleared; nonmaskable interrupts remain active. Second, it terminates any low-power mode, as explained in the section "Low-Power Modes of Operation" on page 198. (The SCG0 bit is not cleared in the MSP430x4xx family, which means that the frequency-locked loop is not automatically reactivated; see "Frequency-Locked Loop, FLL+" on page 172.)
7. The interrupt vector is loaded into the PC and the CPU starts to execute the interrupt service routine at that address.

This sequence takes six clock cycles in the MSP430 before the ISR commences. The stack at this point is shown in Figure 6.5. The position of SR on the stack is important if the low-power mode of operation needs to be changed.

The delay between an interrupt being requested and the start of the ISR is called the *latency*. If the CPU is already running it is given by the time to execute the current instruction, which might only just have started when the interrupt was requested, plus the six cycles needed to execute the launch sequence. This should be calculated for the slowest instruction to get the worst case. Format I instructions take up to 6 clock cycles so the overall latency is 12 cycles. The time required to start MCLK replaces the duration of the

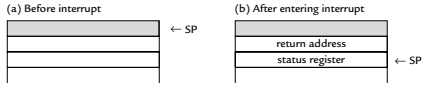


Figure 6.5: Stack before and after entering an interrupt service routine. The return address (PC) and status register (SR) have been saved, with SR on the top of the stack.

current instruction if the device was in a low-power mode. The delay varies on each occasion because the interrupt may be requested at different points during an instruction, whose length may also differ. Thus there is no fixed interval between the request of an interrupt and the start of its ISR. Use the hardware of a timer to read an input or change an output at a precise time. Figure 6.6 shows an example of this and there are many more in Chapter 8.

An interrupt service routine must always finish with the special *return from interrupt* instruction `reti`, which has the following actions:

1. The SR pops from the stack. All previous settings of GIE and the mode control bits are now in effect, regardless of the settings used during the interrupt service routine. In particular, this reenables maskable interrupts and restores the previous low-power mode of operation if there was one.
2. The PC pops from the stack and execution resumes at the point where it was interrupted. Alternatively, the CPU stops and the device reverts to its low-power mode before the interrupt.

This takes a further five cycles in the MSP430. The stack is restored to its state before the interrupt was accepted.

6.8.1 Interrupt Service Routines in Assembly Language

An ISR looks almost identical to a subroutine but with two distinctions:

- The address of the subroutine, for which we can use its name (a label on its first line), must be stored in the appropriate interrupt vector.
- The routine must end with `reti` rather than `ret` so that the correct sequence of actions takes place when it returns.

The other change in the program is that interrupts must be enabled or nothing happens.