User's Guide

MSP430 Family Software

1994

TEXAS INSTRUMENTS

MSP430 Family

Software User's Guide

1994

IMPORTANT NOTICE

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

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

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

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

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

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

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

MSP430 Family
Software Users Guide

Topics

1 Introduction

This section discusses the features of the MSP430 family of controllers with special capabilities for analog processing control. All family members are software compatible, allowing easy migration within the MSP430 family by maintaining a software base, design expertise and development tools.

The concept of a CPU designed for various applications with a 16-bit structure is presented. It uses a "von-Neumann Architecture" and hence has RAM, ROM and all peripherals in one address space.

Topics

Notes

2 Instruction set

The MSP430 Core CPU architecture evolved from the idea of using a reduced instruction set and highly transparent instruction formats. There are instructions that are implemented into hardware and instructions that use the present hardware construction and emulate instructions with high efficiency. The emulated instructions use core instructions with the additional built-in constant generators CG1 and CG2. Both the core instructions (hardware implemented instructions) and the emulated instructions are described in this part. The mnemonics of the emulated instructions are used with the examples.

The words in programme memory used by an instruction vary from 1 to 3 words depending on the combination of addressing modes.
Each instruction uses a minimum of one word (two bytes) in the programme memory. The indexed, symbolic, absolute and immediate modes need in one additional word in the programme memory. These four modes are available for the source operand. The indexed, symbolic and absolute mode can be used for the destination operand.
The instruction combination for source and destination consumes one to three words of code memory.

2.1 Instruction Set Overview

				Status Bits			
				V	N	Z	C
*	ADC[.W];ADC.B	dst	dst + C -> dst	*	*	*	*
	ADD[.W];ADD.B	src,dst	src + dst -> dst	*	*	*	*
	ADDC[.W];ADDC.B	src,dst	src + dst + C -> dst	*	*	*	*
	AND[.W];AND.B	src,dst	src .and. dst -> dst	0	*	*	*
	BIC[.W];BIC.B	src,dst	.not.src .and. dst -> dst	-	-	-	-
	BIS[.W];BIS.B	src,dst	src .or. dst -> dst	-	-	-	-
	BIT[.W];BIT.B	src,dst	src .and. dst	0	*	*	*
*	BR	dst	Branch to	-	-	-	-
	CALL	dst	PC+2 -> stack, dst -> PC	-	-	-	-
*	CLR[.W];CLR.B	dst	Clear destination	-	-	-	-
*	CLRC	Clear carry bit		-	-	-	0
*	CLRN	Clear negative bit		-	0	-	-
*	CLRZ	Clear zero bit		-	-	0	-
	CMP[.W];CMP.B	src,dst	dst - src	*	*	*	*
*	DADC[.W];DADC.B	dst	dst + C -> dst (decimal)	*	*	*	*
	DADD[.W];DADD.B	src,dst	src + dst + C -> dst (decimal)	*	*	*	*
*	DEC[.W];DEC.B	dst	dst - 1 -> dst	*	*	*	*
*	DECD[.W];DECD.B	dst	dst - 2 -> dst	*	*	*	*
*	DINT	Disable interrupt		-	-	-	-
*	EINT	Enable interrupt		-	-	-	-
*	INC[.W];INC.B	dst	Increment destination, dst +1 -> dst	*	*	*	*

*	INCD[.W];INCD.B	dst	Double-Increment destination, dst+2->dst	*	*	*	*
*	INV[.W];INV.B	dst	Invert destination	*	*	*	*
	JC/JHS	Label	Jump to Label if Carry-bit is set	-	-	-	-
	JEQ/JZ	Label	Jump to Label if Zero-bit is set	-	-	-	-
	JGE	Label	Jump to Label if (N .XOR. V) = 0	-	-	-	-
	JL	Label	Jump to Label if (N .XOR. V) = 1	-	-	-	-
	JMP	Label	Jump to Label unconditionally	-	-	-	-
	JN	Label	Jump to Label if Negative-bit is set	-	-	-	-
	JNC/JLO	Label	Jump to Label if Carry-bit is reset	-	-	-	-
	JNE/JNZ	Label	Jump to Label if Zero-bit is reset	-	-	-	-

Note: Marked instructions are emulated instructions

All marked instructions (*) are emulated instructions. The emulated instructions use core instructions combined with the architecture and implementation of the CPU for higher code efficiency and faster execution.

				Status Bits			
				V	N	Z	C
	MOV[.W];MOV.B	src,dst	src -> dst	-	-	-	-
*	NOP		No operation	-	-	-	-
*	POP[.W];POP.B	dst	Item from stack, SP+2 → SP	-	-	-	-
	PUSH[.W];PUSH.B	src	SP - 2 → SP, src → @SP	-	-	-	-
	RETI		Return from interrupt TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SZP	*	*	*	*
*	RET		Return from subroutine TOS → PC, SP + 2 → SP	-	-	-	-
*	RLA[.W];RLA.B	dst	Rotate left arithmetically	*	*	*	*
*	RLC[.W];RLC.B	dst	Rotate left through carry	*	*	*	*
	RRA[.W];RRA.B	dst	MSB → MSB → ....LSB → C	0	*	*	*
	RRC[.W];RRC.B	dst	C → MSB → .....LSB → C	*	*	*	*
*	SBC[.W];SBC.B	dst	Subtract carry from destination	*	*	*	*
*	SETC		Set carry bit	-	-	-	1
*	SETN		Set negative bit	-	1	-	-
*	SETZ		Set zero bit	-	-	1	-
	SUB[.W];SUB.B	src,dst	dst + .not.src + 1 → dst	*	*	*	*
	SUBC[.W];SUBC.B	src,dst	dst + .not.src + C → dst	*	*	*	*
	SWPB	dst	swap bytes	-	-	-	-
	SXT	dst	Bit7 → Bit8 ..... Bit15	0	*	*	*
*	TST[.W];TST.B	dst	Test destination	0	*	*	1
	XOR[.W];XOR.B	src,dst	src .xor. dst → dst	*	*	*	*

Note: Marked instructions

All marked instructions (*) are emulated instructions. The emulated instructions use core instructions combined with the architecture and implementation of the CPU for higher code efficiency and faster execution.

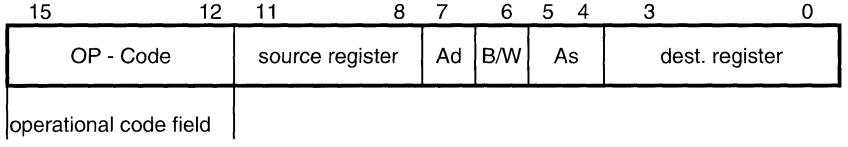
2.2 Instruction Formats

Double operand instructions (core instructions)

The instruction format using double operands consists of four main fields, in total a 16bit code:

- operational code field, 4bit [OP-Code]
- source field, 6bit [source register + As]
- byte operation identifier, 1bit [BW]
- destination field, 5bit [dest. register + Ad]

The source field is composed of two addressing bits and the 4bit register number (0....15); the destination field is composed of one addressing bit and the 4bit register number (0....15).
The byte identifier B/W indicates whether the instruction is executed as a byte (B/W=1) or as a word instruction (B/W=0)



			Status Bits			
			V	N	Z	C
ADD[.W[;ADD.B	src,dst	src + dst -> dst	*	*	*	*
ADDC[.W];ADDC.B	src,dst	src + dst + C -> dst	*	*	*	*
AND[.W];AND.B	src,dst	src .and. dst -> dst	0	*	*	*
BIC[.W];BIC.B	src,dst	.not.src .and. dst -> dst	-	-	-	-
BIS[.W];BIS.B	src,dst	src .or. dst -> dst	-	-	-	-
BIT[.W];BIT.B	src,dst	src .and. dst	0	*	*	*
CMP[.W];CMP.B	src,dst	dst - src	*	*	*	*
DADD[.W];DADD.B	src,dst	src + dst + C -> dst (dec)	*	*	*	*
MOV[.W];MOV.B	src,dst	src -> dst	-	-	-	-
SUB[.W];SUB.B	src,dst	dst + .not.src + 1 -> dst	*	*	*	*
SUBC[.W];SUBC.B	src,dst	dst + .not.src + C -> dst	*	*	*	*
XOR[.W];XOR.B	src,dst	src .xor. dst -> dst	*	*	*	*

Note: Operations using Status Register SR for destination

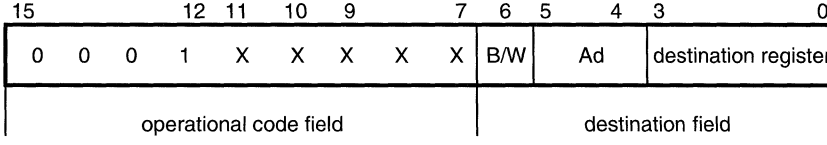
All operations using Status Register SR for destination overwrite the content of SR with the result of that operation: the status bits are not affected as described in that operation.

Example: ADD #3,SR ; Operation: (SR) + 3 --> SR

Single operand instructions (core instructions)

The instruction format using a single operand consists of two main fields, in total 16bit:
- operational code field, 9bit with 4MSB equal '1h'
- byte operation identifier, 1bit [BW]
- destination field, 6bit [destination register + Ad]

The destination field is composed of two addressing bits and the 4bit register number (0....15). The bit position of the destination field is located in the same position as the two operand instructions.
The byte identifier B/W indicates whether the instruction is executed as a byte (B/W=1) or as a word instruction (B/W=0)



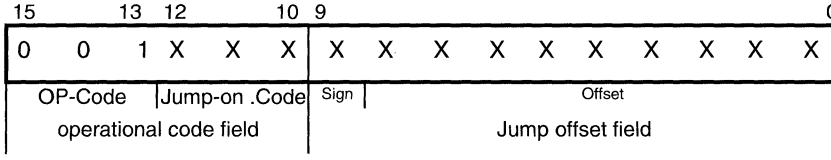
			Status Bits			
			V	N	Z	C
RRA[.W];RRA.B	dst	MSB → MSB → ...LSB → C	0	*	*	*
RRC[.W];RRC.B	dst	C → MSB → .....LSB → C	*	*	*	*
PUSH[.W];PUSH.B	dst	SP - 2 → SP, src → @SP	-	-	-	-
SWPB	dst	swap bytes	-	-	-	-
CALL	dst	PC+2 → @SP, dst → PC	-	-	-	-
RETI		TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SP	*	*	*	*
SXT	dst	Bit7 -> Bit8 ..... Bit15	0	*	*	*

Conditional and unconditional Jumps (core instructions)

The instruction format for (un-)conditional jumps consists of two main fields, in total 16bit :

- operational code (OP-Code) field, 6bit
- jump offset field, 10bit

The operational code field is composed of OP-Code (3bits) and 3 bits according to the following conditions.



The conditional jumps allow jumps to addresses in the range -511 to +512 words relative to the current address. The assembler computes the signed offsets and inserts them into the opcode.

JC/JHS	Label	Jump to Label if Carry-bit is set
JEQ/JZ	Label	Jump to Label if Zero-bit is set
JGE	Label	Jump to Label if (N .XOR. V) = 0
JL	Label	Jump to Label if (N .XOR. V) = 1
JMP	Label	Jump to Label unconditionally
JN	Label	Jump to Label if Negative-bit is set
JNC/JLO	Label	Jump to Label if Carry-bit is reset
JNE/JNZ	Label	Jump to Label if Zero-bit is reset

Note: Conditional and unconditional Jumps

The conditional and unconditional Jumps do not effect the status bits.

A Jump which has been taken alters the PC with the offset: PCnew=PCold + 2 + 2*offset.
A Jump which has not been taken continues the programme with the ascending instruction.

Emulation of instructions without ROM penalty

The following instructions can be emulated with the reduced instruction set without additional ROM words. The assembler accepts the mnemonic of the emulated instruction and inserts the opcode of the suitable core instruction.

Note: Emulation of the following instructions

The emulation of the following instructions is possible using the contents of R2 and R3:
The register R2(CG1) contains the immediate values 2 and 4; the register R3(CG2) contains -1 or 0FFFFh, 0, +1 and +2 depending on the addressing bits As. The assembler sets the addressing bits according to the used immediate value.

Short form of emulated instructions

Mnemonic		Description	Statusbits V	N	Z	C	Emulation	
Arithmetical instructions								
ADC[.W]	dst	Add carry to destination	*	*	*	*	ADDC	#0,dst
ADC.B	dst	Add carry to destination	*	*	*	*	ADDC.B	#0,dst
DADC[.W]	dst	Add carry decimal to destination	*	*	*	*	DADD	#0,dst
DADC.B	dst	Add carry decimal to destination	*	*	*	*	DADD.B	#0,dst
DEC[.W]	dst	Decrement destination	*	*	*	*	SUB	#1,dst
DEC.B	dst	Decrement destination	*	*	*	*	SUB.B	#1,dst
DECD[.W]	dst	Double-Decrement destination	*	*	*	*	SUB	#2,dst
DECD.B	dst	Double-Decrement destination	*	*	*	*	SUB.B	#2,dst
INC[.W]	dst	Increment destination	*	*	*	*	ADD	#1,dst
INC.B	dst	Increment destination	*	*	*	*	ADD.B	#1,dst
INCD[.W]	dst	Increment destination	*	*	*	*	ADD	#2,dst
INCD.B	dst	Increment destination	*	*	*	*	ADD.B	#2,dst
SBC[.W]	dst	Subtract carry from destination	*	*	*	*	SUBC	#0,dst
SBC.B	dst	Subtract carry from destination	*	*	*	*	SUBC.B	#0,dst
Logical instructions								
INV[.W]	dst	Invert destination	*	*	*	*	XOR	#0FFFFh,dst
INV.B	dst	Invert destination	*	*	*	*	XOR.B	#0FFFFh,dst
RLA[.W]	dst	Rotate left arithmetically	*	*	*	*	ADD	dst,dst
RLA.B	dst	Rotate left arithmetically	*	*	*	*	ADD.B	dst,dst
RLC[.W]	dst	Rotate left through carry	*	*	*	*	ADDC	dst,dst
RLC.B	dst	Rotate left through carry	*	*	*	*	ADDC.B	dst,dst
Data instructions (common use)								
CLR[.W]		Clear destination	-	-	-	-	MOV	#0,dst
CLR.B		Clear destination	-	-	-	-	MOV.B	#0,dst
CLRC		Clear carry bit	-	-	-	0	BIC	#1,SR
CLRN		Clear negative bit	-	0	-	-	BIC	#4,SR
CLRZ		Clear zero bit	-	-	0	-	BIC	#2,SR
POP	dst	Item from stack	-	-	-	-	MOV	@SP+,dst
SETC		Set carry bit	-	-	-	1	BIS	#1,SR
SETN		Set negative bit	-	1	-	-	BIS	#4,SR
SETZ		Set zero bit	-	-	1	-	BIS	#2,SR
TST[.W]	dst	Test destination	0	*	*	1	CMP	#0,dst
TST.B	dst	Test destination	0	*	*	1	CMP.B	#0,dst
Programme flow instructions								
BR	dst	Branch to	-	-	-	-	MOV	dst,PC
DINT		Disable interrupt	-	-	-	-	BIC	#8,SR
EINT		Enable interrupt	-	-	-	-	BIS	#8,SR
NOP		No operation	-	-	-	-	MOV	#0h,#0h
RET		Return from subroutine	-	-	-	-	MOV	@SP+,PC

2.3 Instruction set description - alphabetical order

This section catalogues and describes all core and emulated instructions. Some examples are given for explanation and as application hints.
The suffix .W or no suffix in the instruction memonic will result in a word operation.
The suffix .B at the instruction memonic will result in a byte operation.

*** ADC[.W]** Add carry to destination

Syntax ADC dst or ADC.W dst

Operation dst + C -> dst

Emulation ADDC #0,dst

Description The carry C is added to the destination operand. The previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if dst was incremented from 0FFFFh to 0000, reset otherwise
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 16-bit counter pointed to by R13 is added to a 32-bit counter pointed to by R12.

```
ADD     @R13,0(R12)     ; Add LSDs
ADC     2(R12)          ; Add carry to MSD
```

* ADC.B Add carry to destination

Syntax ADC.B dst

Operation dst + C -> dst

Emulation ADDC.B #0,dst

Description The carry C is added to the destination operand. The previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if dst was incremented from 0FFh to 00, reset otherwise
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 8-bit counter pointed to by R13 is added to a 16-bit counter pointed to by R12.

```
ADD.B    @R13,0(R12)    ; Add LSDs
ADC.B    1(R12)         ; Add carry to MSD
```

ADD[.W] Add source to destination

Syntax ADD src,dst or ADD.W src,dst

Operation src + dst -> dst

Description The source operand is added to the destination operand. The source operand is not affected, the previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the result, cleared if not.
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R5 is increased by 10. The 'Jump' to TONI is performed on a carry

```
ADD     #10,R5
JC      TONI        ; Carry occurred
.....               ; No carry
```

ADD.B

Add source to destination

Syntax ADD.B src,dst

Operation src + dst -> dst

Description The source operand is added to the destination operand. The source operand is not affected, the previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the result, cleared if not.
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R5 is increased by 10. The 'Jump' to TONI is performed on a carry

```
ADD.B   #10,R5      ; Add 10 to Lowbyte of R5
JC      TONI        ; Carry occurred, if (R5) ≥ 246 [0Ah+0F6h]
.....               ; No carry
```

ADDC[.W]

Add source and carry to destination.

Syntax ADDC src,dst or ADDC.W src,dst

Operation src + dst + C -> dst

Description The source operand and the carry C are added to the destination operand. The source operand is not affected, the previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the MSB of the result, reset if not
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 32-bit counter pointed to by R13 is added to a 32-bit counter eleven **words** (20/2 + 2/2) above pointer in R13.

```
ADD     @R13+,20(R13)   ; ADD LSDs with no carryin
ADDC    @R13+,20(R13)   ; ADD MSDs with carry
...                     ; resulting from the LSDs
```

ADDC.B

source and carry to destination.

Syntax ADDC.B src,dst

Operation src + dst + C -> dst

Description The source operand and the carry C are added to the destination operand. The source operand is not affected, the previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the MSB of the result, reset if not
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 24-bit counter pointed to by R13 is added to a 24-bit counter eleven words above pointer in R13.

```
ADD.B    @R13+,10(R13)    ; ADD LSDs with no carryin
ADDC.B   @R13+,10(R13)    ; ADD medium Bits with carry
ADDC.B   @R13+,10(R13)    ; ADD MSDs with carry
...                              ; resulting from the LSDs
```

AND[.W] source AND destination

Syntax AND src,dst or AND.W src,dst

Operation src .AND. dst -> dst

Description The source operand and the destination operand are logically AND'ed. The result is placed into the destination.

Status Bits
N: Set if MSB of result is set, reset if not set
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (= .NOT. Zero)
V: Reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The bits set in R5 are used as a mask (#0AA55h) for the word addressed by TOM. If the result is zero, a branch is taken to label TONI

```
        MOV     #0AA55h,R5      ; Load mask into register R5
        AND     R5,TOM          ; mask word addressed by TOM with R5
        JZ      TONI            ;
        ......                  ; Result is not zero
;
;
;       or
;
;
        AND     #0AA55h,TOM
        JZ      TONI
```

AND.B source AND destination

Syntax AND.B src,dst

Operation src .AND. dst -> dst

Description The source operand and the destination operand are logically AND'ed. The result is placed into the destination.

Status Bits
N: Set if MSB of result is set, reset if not set
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (= .NOT. Zero)
V: Reset

Mode Bits OscOff, CPUOff and GIE are not affected

Example The bits of mask #0A5h are logically AND'ed with the Lowbyte TOM. If the result is zero, a branch is taken to label TONI

```
AND.B   #0A5h,TOM       ; mask Lowbyte TOM with R5
JZ      TONI            ;
.....                   ; Result is not zero
```

BIC[.W] Clear bits in destination

Syntax BIC src,dst or BIC.W src,dst

Operation .NOT.src .AND. dst -> dst

Description The inverted source operand and the destination operand are logically AND'ed. The result is placed into the destination. The source operand is not affected.

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 6 MSBs of the RAM word LEO are cleared.

```
BIC #0FC00h,LEO  ; Clear 6 MSBs in MEM(LEO)
```

BIC.B

Clear bits in destination

Syntax BIC.B src,dst

Operation .NOT.src .AND. dst -> dst

Description The inverted source operand and the destination operand are logically AND'ed. The result is placed into the destination. The source operand is not affected.

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 5 MSBs of the RAM byte LEO are cleared.

```
BIC.B   #0F8h,LEO       ; Clear 5 MSBs in Ram location LEO
```

Example The Portpins P0 and P1 are cleared.

```
P0OUT   .equ    011h    ;Definition of the Portaddress
P0      .equ    01h
P1      .equ    02h

BIC.B   #P0+P1,&P0OUT   ;Set P0 and P1 to low
```

BIS[.W] Set bits in destination

Syntax BIS src,dst or BIS.W src,dst

Operation src .OR. dst -> dst

Description The source operand and the destination operand are logically OR'ed. The result is placed into the destination. The source operand is not affected.

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 6 LSB's of the RAM word TOM are set.

```
BIS     #003Fh,TOM  ; set the 6 LSB's in RAM location TOM
```

Example Start an A/D-conversion

```
ASOC    .equ    1       ;Start of Conversion bit
ACTL    .equ    114h    ;ADC-Control Register

BIS     #ASOC,&ACTL     ;Start A/D-conversion
```

BIS.B

Set bits in destination

Syntax BIS.B src,dst

Operation src .OR. dst -> dst

Description The source operand and the destination operand are logically OR'ed. The result is placed into the destination. The source operand is not affected.

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 3 MSBs of the RAM byte TOM are set.

```
BIS.B   #0E0h,TOM   ; set the 3 MSBs in RAM location TOM
```

Example The Portpins P0 and P1 are set to high

```
P0OUT   .equ    011h
P0      .equ    01h
P1      .equ    02h

BIS.B   #P0+P1,&P0OUT
```

BIT[.W] Test bits in destination

Syntax BIT src,dst or BIT.W src,dst

Operation src .AND. dst

Description The source operand and the destination operand are logically AND'ed. The result affects only the Status Bits. The source and destination operands are not affected.

Status Bits
N: Set if MSB of result is set, reset if not set
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (.NOT. Zero)
V: Reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example If bit 9 of R8 is set, a branch is taken to label TOM.

```
BIT     #0200h,R8       ; bit 9 of R8 set ?
JNZ     TOM             ; Yes, branch to TOM
...                     ; No, proceed
```

Example Determine which A/D-Channel is configured by the MUX

```
ACTL    .equ    114h    ;ADC Control Register

BIT     #4,&ACTL        ;Is Channel 0 selected ?
jnz     END             ;Yes, branch to END
```

BIT.B Test bits in destination

Syntax BIT.B src,dst

Operation src .AND. dst

Description The source operand and the destination operand are logically AND'ed. The result affects only the Status Bits: the source and destination operands are not affected.

Status Bits
N: Set if MSB of result is set, reset if not set
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (.NOT. Zero)
V: Reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example If bit 3 of R8 is set, a branch is taken to label TOM.

```
BIT.B   #8,R8
JC      TOM
```

Example

```
; The receive bit RCV of a serial communication is tested. Since while using the BIT
        ; instruction to test a single bit the carry is equal to the state of the
        tested bit, the carry is ; used by the subsequent instruction: the read info
        is shifted into the register RECBUF.
;
; Serial communication with LSB is shifted first:
                                        ; xxxx  xxxx    xxxx    xxxx
        BIT.B   #RCV,RCCTL              ; Bit info into carry
        RRC     RECBUF                  ; Carry -> MSB of RECBUF
                                        ; cxxx  xxxx
        ......                          ; repeat previous two instructions
        ......                          ; 8 times
                                        ; cccc  cccc
                                        ; ^         ^
                                        ; MSB       LSB

; Serial communication with MSB is shifted first:
        BIT.B   #RCV,RCCTL              ; Bit info into carry
        RLC.B   RECBUF                  ; Carry -> LSB of RECBUF
                                        ; xxxx  xxxc
        ......                          ; repeat previous two instructions
        ......                          ; 8 times
                                        ; cccc      cccc
                                        ; |         LSB
                                        ; MSB
```

* BR, BRANCH

Branch to ..... destination

Syntax BR dst

Operation dst -> PC

Emulation MOV dst,PC

Description An unconditional branch is taken to an address anywhere in the 64 K address space. All source addressing modes may be used. The branch instruction is a word instruction.

Status Bits Status bits are not affected

Examples Examples for all addressing modes are given

```
;
        BR    #EXEC     ; Branch to label EXEC or direct branch (e.g #0A4h)
                        ; Core instruction  MOV  @PC+,PC
;
        BR    EXEC      ; Branch to the address contained in EXEC
                        ; Core instruction  MOV  X(PC),PC
                        ; Indirect address
;
        BR    &EXEC     ; Branch to the address contained in absolute
                        ; address
                        ; EXEC
                        ; Core instruction  MOV  X(0),PC
                        ; Indirect address
;
        BR    R5        ; Branch to the address contained in R5
                        ; Core instruction  MOV  R5,PC
                        ; Indirect R5
;
        BR    @R5       ; Branch to the address contained in the word R5
                        ; points
                        ; to. Core instruction  MOV  @R5,PC
                        ; Indirect, indirect R5
;
        BR    @R5+      ; Branch to the address contained in the word R5
                        ; points to and increments pointer in R5 afterwards.
                        ; The next time - S/W flow uses R5 pointer - it can
                        ; alter the programme execution due to access to
                        ; next address in a table, pointed by R5
                        ; Core instruction  MOV  @R5,PC
                        ; Indirect, indirect R5 with autoincrement
```

```
BR      X(R5)       ; Branch to the address contained in the address
                    ; pointed to  by R5 + X  (e.g table with address start-
                    ; ing at X). X can be an address or a label
                    ; Core instruction MOV  X(R5),PC
                    ; Indirect indirect R5 + X
```

CALL Subroutine

Syntax CALL dst

Operation

dst	-> tmp	dst is evaluated and stored
SP - 2	-> SP	
PC	-> @SP	updated PC to TOS
tmp	-> PC	saved dst to PC

Description A subroutine call is made to an address anywhere in the 64-K-address space. All addressing modes may be used. The return address (the address of the following instruction) is stored on the stack. The call instruction is a word instruction.

Status Bits Status bits are not affected

Example Examples for all addressing modes are given

```
CALL   #EXEC   ; Call on label EXEC or immediate address (e.g
               ; #0A4h)
               ; SP-2 → SP, PC+2 → @SP, @PC+ → PC
CALL   EXEC    ; Call on the address contained in EXEC
               ; SP-2 → SP, PC+2 → @SP, X(PC) → PC
               ; Indirect address
CALL   &EXEC   ; Call on the address contained in absolute address
               ; EXEC
               ; SP-2 → SP, PC+2 → @SP, X(PC) → PC
               ; Indirect address
CALL   R5      ; Call on the address contained in R5
               ; SP-2 → SP, PC+2 → @SP, R5 → PC
               ; Indirect R5
CALL   @R5     ; Call on the address contained in the word R5
               ; points
               ; to
               ; SP-2 → SP, PC+2 → @SP, @R5 → PC
               ; Indirect, indirect R5
CALL   @R5+    ; Call on the address contained in the word R5 points
               ; to and increments pointer in R5. The next time -
               ; S/W flow uses R5 pointer - it can alter the
               ; programme execution due to access to next address
               ; in a table, pointed ; to by R5
               ; SP-2 → SP, PC+2 → @SP, @R5 → PC
               ; Indirect, indirect R5 with autoincrement
CALL   X(R5)   ; Call on the address contained in the address pointed
               ; to by R5 + X (e.g table with address starting at X)
               ; X can be an address or a label
               ; SP-2 → SP, PC+2 → @SP, X(R5) → PC
               ; Indirect indirect R5 + X
```

* CLR[.W] Clear destination

Syntax CLR dst or CLR.W dst

Operation 0 -> dst

Emulation MOV #0,dst

Description The destination operand is cleared.

Status Bits Status bits are not affected

Example RAM word TONI is cleared

```
CLR     TONI    ; 0 -> TONI
```

Example Register R5 is cleared

```
CLR     R5
```

*** CLR.B** Clear destination

Syntax CLR.B dst

Operation 0 -> dst

Emulation MOV.B #0,dst

Description The destination operand is cleared.

Status Bits Status bits are not affected

Example RAM byte TONI is cleared

```
CLR.B   TONI    ; 0 -> TONI
```

* CLRC Clear carry bit

Syntax CLRC

Operation 0 -> C

Emulation BIC#1,SR

Description The Carry Bit C is cleared. The clear carry instruction is a word instruction.

Status Bits
N: Not affected
Z: Not affected
C: Cleared
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 16bit decimal counter pointed to by R13 is added to a 32bit counter pointed to by R12.

```
CLRC                    ; C=0: Defines start
DADD    @R13,0(R12)     ; add 16bit counter to Lowword of 32bit
                        ; counter
DADC    2(R12)          ; add carry to Highword of 32bit counter
```

* CLRN Clear Negative bit

Syntax CLRN

Operation 0 → N
or
(.NOT.src .AND. dst -> dst)

Emulation BIC#4,SR

Description The constant 04h is inverted (0FFFBh) and the destination operand are logically AND'ed. The result is placed into the destination. The clear negative bit instruction is a word instruction.

Status Bits
N: Reset to 0
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The Negative bit in the status register is cleared. This avoids the special treatment of the called subroutine with negative numbers.

```
        CLRN
        CALL    SUBR
        ......
        ......
SUBR    JN      SUBRET          ; If input is negative: do nothing and return
        ......
        ......
        ......
SUBRET  RET
```

* CLRZ Clear Zero bit

Syntax CLRZ

Operation 0 → Z
or
(.NOT.src .AND. dst -> dst)

Emulation BIC #2,SR

Description The constant 02h is inverted (0FFFDh) and the destination operand are logically AND'ed. The result is placed into the destination. The clear zero bit instruction is a word instruction.

Status Bits
N: Not affected
Z: Reset to 0
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The Zero bit in the status register is cleared.

```
CLRZ
```

CMP[.W] compare source and destination

Syntax CMP src,dst or CMP.W src,dst

Operation dst + .NOT.src + 1
or
(dst - src)

Description The source operand is subtracted from the destination operand. This is made by adding of the 1's complement of the source operand plus 1. The two operands are not affected, the result is not stored, only the status bits are affected.

Status Bits
N: Set if result is negative, reset if positive (src >= dst)
Z: Set if result is zero, reset otherwise (src = dst)
C: Set if there is a carry from the MSB of the result, reset if not
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R5 and R6 are compared. If they are equal, the programme continues at the label EQUAL

```
CMP   R5,R6     ; R5 = R6 ?
JEQ   EQUAL     ; YES, JUMP
```

Example Two RAM blocks are compared. If they not equal, the programme branches to the label ERROR

```
      MOV   #NUM,R5              ;number of words to be compared
L$1   CMP   &BLOCK1,&BLOCK2      ;Are Words equal ?
      JNZ   ERROR                ;No, branch to ERROR
      DEC   R5                   ;Are all words compared?
      JNZ   L$1                  ;No, another compare
```

CMP.B compare source and destination

Syntax CMP.B src,dst

Operation dst + .NOT.src + 1
or
(dst - src)

Description The source operand is subtracted from the destination operand. This is made by adding of the 1's complement of the source operand plus 1. The two operands are not affected, the result is not stored, only the status bits are affected.

Status Bits
N: Set if result is negative, reset if positive (src >= dst)
Z: Set if result is zero, reset otherwise (src = dst)
C: Set if there is a carry from the MSB of the result, reset if not
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The RAM bytes addressed by EDE and TONI are compared. If they are equal, the programme continues at the label EQUAL

```
CMP.B       EDE,TONI     ; MEM(EDE) = MEM(TONI) ?
JEQ     EQUAL        ; YES, JUMP
```

Example Check two Keys, which are connected to the Portpin P0 and P1. If key1 is pressed, the programme branches to the label MENU1, if key2 is pressed, the programme branches to MENU2.

```
P0IN    .EQU    010h
KEY1    .EQU    01h
KEY2    .EQU    02h

CMP.B           #KEY1,&P0IN
JEQ     MENU1
CMP.B           #KEY2,&P0IN
JEQ     MENU2
```

* DADC[.W] Add carry decimally

Syntax DADC dst o DADC.W src,dst

Operation dst + C -> dst (decimally)

Emulation DADD #0,dst

Description The Carry Bit C is added decimally to the destination

Status Bits
N: Set if MSB is 1
Z: Set if dst is 0, reset otherwise
C: Set if destination increments from 9999 to 0000, reset otherwise
V: Undefined

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 4-digit decimal number contained in R5 is added to an 8-digit decimal number pointed to by R8

```
CLRC            ; Reset carry
                ; next instruction's start condition is defined
DADD  R5,0(R8)  ; Add LSDs + C
DADC  2(R8)     ; Add carry to MSD
```

* DADC.B Add carry decimally

Syntax DADC.B dst

Operation dst + C -> dst (decimally)

Emulation DADD.B #0,dst

Description The Carry Bit C is added decimally to the destination

Status Bits
N: Set if MSB is 1
Z: Set if dst is 0, reset otherwise
C: Set if destination increments from 99 to 00, reset otherwise
V: Undefined

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 2-digit decimal number contained in R5 is added to an 4-digit decimal number pointed to by R8

```
CLRC                    ; Reset carry
                        ; next instruction's start condition is
                        ; defined
DADD.B   R5,0(R8)       ; Add LSDs + C
DADC     1(R8)          ; Add carry to MSDs
```

DADD[.W] source and carry added decimally to destination

Syntax DADD src,dst or DADD.W src,dst

Operation src + dst + C -> dst (decimally)

Description The source operand and the destination operand are treated as four binary coded decimals (BCD) with positive signs. The source operand and the carry C are added decimally to the destination operand. The source operand is not affected, the previous contents of the destination are lost. The result is not defined for non-BCD numbers.

Status Bits
N: Set if the MSB is 1, reset otherwise
Z: Set if result is zero, reset otherwise
C: Set if the result is greater than 9999.
V: Undefined
OscOff, **CPUOff** and **GIE** are not affected

Example The 8-digit-BCD-number contained in R5 and R6 is added decimally to a 8-digit-BCD-number contained in R3 and R4 (R6 and R4 contain the MSDs).

```
CLRC                ; CLEAR CARRY
DADD    R5,R3       ; add LSDs
DADD    R6,R4       ; add MSDs with carry
JC      OVERFLOW    ; If carry occurs go to error handling routine
```

DADD.B source and carry added decimally to destination

Syntax DADD.B src,dst

Operation src + dst + C -> dst (decimally)

Description The source operand and the destination operand are treated as two binary coded decimals (BCD) with positive signs. The source operand and the carry C are added decimally to the destination operand. The source operand is not affected, the previous contents of the destination are lost. The result is not defined for non-BCD numbers.

Status Bits **N:** Set if the MSB is 1, reset otherwise
Z: Set if result is zero, reset otherwise
C: Set if the result is greater than 99.
V: Undefined
OscOff, **CPUOff** and **GIE** are not affected

Example The 2-digit decimal counter in RAMbyte CNT is incremented by one.

```
CLRC                    ; clear Carry
DADD.B    #1,CNT        ; increment decimal counter
```

or

```
SETC
DADD.B    #0,CNT        ; =DADC.B    CNT
```

* DEC[.W] Decrement destination

Syntax DEC dst or DEC.W dst

Operation dst - 1 -> dst

Emulation SUB #1,dst

Description The destination operand is decremented by one. The original contents are lost.

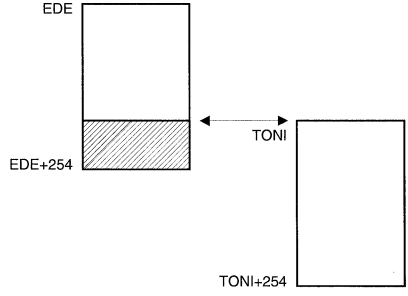
Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 1, reset otherwise
C: Reset if dst contained 0, set otherwise
V: Set if an arithmetic overflow occurs, otherwise reset.
Set if initial value of destination was 08000h, otherwise reset.

Mode Bits **OscOff, CPUOff** and **GIE** are not affected

Example R10 is decremented by 1

```
        DEC     R10         ; Decrement R10
```

```
; Move a block of 255 bytes from memory location starting with EDE to memory location
; starting with TONI
; Tables should not overlap: start of destination address TONI must not be within the
range ; EDE to EDE+0FEh
;
                MOV     #EDE,R6
                MOV     #255,R10
L$1             MOV.B   @R6+,TONI-EDE-1(R6)
                DEC     R10
                JNZ     L$1

;               Do not transfer tables with the routine above with this ovelap:
```



* DEC.B Decrement destination

Syntax DEC.B dst

Operation dst - 1 -> dst

Emulation SUB.B #1,dst

Description The destination operand is decremented by one. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 1, reset otherwise
C: Reset if dst contained 0, set otherwise
V: Set if an arithmetic overflow occurs, otherwise reset.
Set if initial value of destination was 080h, otherwise reset.

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Memory byte at address LEO is decremented by 1

```
        DEC.B   LEO  ; Decrement MEM(LEO)
```

```
; Move a block of 255 bytes from memory location starting with EDE to memory location
; starting with TONI
; Tables should not overlap: start of destination address TONI must not be within the
range ; EDE to EDE+0FEh
;
        MOV     #EDE,R6
        MOV.B   #255,LEO
L$1     MOV.B   @R6+,TONI-EDE-1(R6)
        DEC.B   LEO
        JNZ     L$1
```

* DECD[.W] Double-Decrement destination

Syntax DECD dst or DECD.W dst

Operation dst - 1 -> dst

Emulation SUB #2,dst

Description The destination operand is decremented by two. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 2, reset otherwise
C: Reset if dst contained 0 or 1, set otherwise
V: Set if an arithmetic overflow occurs, otherwise reset.
Set if initial value of destination was 08001 or 08000h, otherwise reset.

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R10 is decremented by 2

```
DECD    R10     ; Decrement R10 by two
```

```
; Move a block of 255 words from memory location starting with EDE to memory location
; starting with TONI
; Tables should not overlap: start of destination address TONI must not be within the
range ; EDE to EDE+0FEh
;
        MOV     #EDE,R6
        MOV     #510,R10
L$1     MOV     @R6+,TONI-EDE-2(R6)
        DECD    R10
        JNZ     L$1
```

* DECD.B Double-Decrement destination

Syntax DECD.B dst

Operation dst - 2 -> dst

Emulation SUB.B #2,dst

Description The destination operand is decremented by two. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 2, reset otherwise
C: Reset if dst contained 0 or 1, set otherwise
V: Set if an arithmetic overflow occurs, otherwise reset.
Set if initial value of destination was 081 or 080h, otherwise reset.

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Memory at location LEO is decremented by 2

```
DECD.B      LEO  ; Decrement MEM(LEO)
```

Decrement status byte STATUS by 2

```
DECD.B      STATUS
```

* DINT

Disable (general) interrupts

Syntax DINT

Operation 0 → GIE
or
(0FFF7h .AND. SR → SR / .NOT.src .AND. dst -> dst)

Emulation BIC #8,SR

Description All interrupts are disabled.
The constant #08h is inverted and logically AND'ed with the status register SR. The result is placed into the SR.

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **GIE** is reset.
OscOff and **CPUOff** are not affected

Example The general interrupt enable bit GIE in the status register is cleared to allow a non disrupted move of a 32bit counter. This ensures that the counter is not modified during the move by any interrupt.

```
DINT                    ; All interrupt events using the GIE bit are
                        ; disabled
MOV   COUNTHI,R5        ; Copy counter
MOV   COUNTLO,R6
EINT                    ; All interrupt events using the GIE bit are
                        ; enabled
```

Note: Disable Interrupt

The instruction following the disable interrupt instruction DINT is executed when the interrupt request becomes active during execution of DINT. If any code sequence needs to be protected from being interrupted the DINT instruction should be executed at least one instruction before this sequence.

* EINT

Enable (general) interrupts

Syntax EINT

Operation 1 → GIE
or
(0008h .OR. SR -> SR / .NOT.src .OR. dst -> dst)

Emulation BIS #8,SR

Description All interrupts are enabled.
The constant #08h and the status register SR are logically OR'ed. The result is placed into the SR.

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits
GIE is set.
OscOff and **CPUOff** are not affected

Example The general interrupt enable bit GIE in the status register is set.

```
; Interrupt routine of port P0.2 to P0.7
; The interrupt level is the lowest in the system
; P0IN is the address of the register where all port bits are read. P0IFG is the address of
; the register where all interrupt events are latched.
;
            PUSH.B   &P0IN
            BIC.B    @SP,&P0IFG ; Reset only accepted flags
            EINT                ; Preset port 0 interrupt flags stored on stack
                                ; other interrupts are allowed
            BIT      #Mask,@SP
            JEQ      MaskOK     ; Flags are present identically to mask: Jump
            ......
MaskOK      BIC      #Mask,@SP
            ......
            INCD     SP         ; Housekeeping: Inverse to PUSH instruction
                                ; at the start of interrupt subroutine. Corrects
                                ; the  stack pointer.
            RETI
```

Note: Enable Interrupt

The instruction following the enable interrupt instruction EINT is executed anyway even on a pending interrupt service request

* INC[.W] Increment destination

Syntax INC dst or INC.W dst

Operation dst + 1 -> dst

Emulation ADD #1,dst

Description The destination operand is incremented by one. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FFFFh, reset otherwise
C: Set if dst contained 0FFFFh, reset otherwise
V: Set if dst contained 07FFFh, reset otherwise

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The item on the top of a software stack (not the system stack) for byte data is removed.

```
SSP     .EQU  R4
;
        INC   SSP   ; Remove TOSS (top of SW stack) by increment
                    ; Do not use INC.B since SSP is a word register
```

* INC.B

Increment destination

Syntax INC.B dst

Operation dst + 1 -> dst

Emulation ADD #1,dst

Description The destination operand is incremented by one. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FFh, reset otherwise
C: Set if dst contained 0FFh, reset otherwise
V: Set if dst contained 07Fh, reset otherwise

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The status byte of a process STATUS is incremented. When it is equal to eleven, a branch to OVFL is taken.

```
INC.B     STATUS
CMP.B               #11,STATUS
JEQ       OVFL
```

* INCD[.W] Double-Increment destination

Syntax INCD dst or INCD.W dst

Operation dst + 2 -> dst

Emulation ADD #2,dst

Description The destination operand is incremented by two. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FFFEh, reset otherwise
C: Set if dst contained 0FFFEh or 0FFFFh, reset otherwise
V: Set if dst contained 07FFEh or 07FFFh, reset otherwise

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The item on the top of the stack is removed without the use of a register.

```
SUB
    .......
    PUSH  R5  ;R5 is the result of a calculation, which is stored in the
              ;Stack
    INCD  SP  ;Remove TOS by double-increment from stack
              Do not use INCD.B, SP is a word aligned register
    RET
```

* INCD.B Double-Increment destination

Syntax INCD.B dst

Operation dst + 2 -> dst

Emulation ADD.B #2,dst

Description The destination operand is incremented by two. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FEh, reset otherwise
C: Set if dst contained 0FEh or 0FFh, reset otherwise
V: Set if dst contained 07Eh or 07Fh, reset otherwise

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The byte on the top of the stack is incremented by two.

```
INCD.B    0(SP)    ; Byte on TOS is increment by two
```

*** INV[.W]** Invert destination

Syntax INV dst

Operation .NOT.dst -> dst

Emulation XOR #0FFFFh,dst

Description The destination operand is inverted. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FFFFh, reset otherwise
C: Set if result is not zero, reset otherwise (= .NOT. Zero)
V: Set if initial destination operand was negative, otherwise reset
OscOff, **CPUOff** and **GIE** are not affected

Example Content of R5 is negated (two's complement).

```
MOV   #00Aeh,R5   ;                    R5 = 000AEh
INV   R5          ; Invert R5,         R5 = 0FF51h
INC   R5          ; R5 is now negated, R5 = 0FF52h
```

* INV.B Invert destination

Syntax INV.B dst

Operation .NOT.dst -> dst

Emulation XOR #0FFh,dst

Description The destination operand is inverted. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FFh, reset otherwise
C: Set if result is not zero, reset otherwise (= .NOT. Zero)
V: Set if initial destination operand is negative, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Content of memory byte LEO is negated.

```
MOV.B   #0AEh,LEO                   MEM(LEO) = 0AEh
INV.B   LEO     ; Invert LEO,       MEM(LEO) = 051h
INC.B   LEO     ; MEM(LEO) is now negated,  MEM(LEO) = 052h
```

JC
JHS

Jump if carry set
Jump if higher or same

Syntax

```
JC      label
JHS     label
```

Operation

if C = 1: PC + 2*offset -> PC
if C = 0: execute following instruction

Description The Carry Bit C of the Status Register is tested. If it is set, the 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter. If C is reset, the next instruction following the jump is executed. JC (jump if carry/higher or same) is used for the comparison of unsigned numbers (0 to 65536).

Status Bits Status bits are not affected

Example The signal of input P0IN.1 is used to define or control the programme flow.

```
BIT     #10h,&P0IN  ; State of signal -> Carry
JC      PROGA       ; If carry=1 then execute programme routine A
.....               ; Carry=0, execute programme here
```

Example R5 is compared to 15. If the content is higher or same branch to LABEL.

```
CMP     #15,R5
JHS     LABEL       ; Jump is taken if R5 ≥ 15
.....               ; Continue here if R5 < 15
```

JEQ, JZ

Jump if equal, Jump if zero

Syntax JEQ label, JZ label

Operation
if Z = 1: PC + 2*offset -> PC
if Z = 0: execute following instruction

Description The Zero Bit Z of the Status Register is tested. If it is set, the 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter. If Z is not set, the next instruction following the jump is executed.

Status Bits Status bits are not affected

Example Jump to address TONI if R7 contains zero.

```
TST     R7          ; Test R7
JZ      TONI        ; if zero: JUMP
```

Example Jump to address LEO if R6 is equal to the table contents.

```
CMP     R6,Table(R5)  ; Compare content of R6 with content of
                      ; MEM(Table address + content of R5)
JEQ     LEO           ; Jump if both data are equal
.....                 ; No, data are not equal, continue here
```

Example Branch to LABEL if R5 is 0.

```
TST     R5
JZ      LABEL
.....
```

JGE

Jump if greater or equal

Syntax JGE label

Operation if (N .XOR. V) = 0 then jump to label: PC + 2*offset -> PC
if (N .XOR. V) = 1 then execute following instruction

Description The negative bit N and the overflow bit V of the Status Register are tested. If both N and V are set or reset, the 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter. If only one is set, the next instruction following the jump is executed.
This allows comparison of signed integers.

Status Bits Status bits are not affected

Example When the content of R6 is greater or equal the memory pointed to by R7 the programme continues at label EDE.

```
CMP     @R7,R6      ; R6 ≥ (R7)?, compare on signed numbers
JGE     EDE         ; Yes, R6 ≥ (R7)
.....               ; No, proceed
.....
.....
```

JL

Jump if less

Syntax

JL label

Operation

if (N .XOR. V) = 1 then jump to label: PC + 2*offset -> PC
if (N .XOR. V) = 0 then execute following instruction

Description

The negative bit N and the overflow bit V of the Status Register are tested. If only one is set, the 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter. If both N and V are set or reset, the next instruction following the jump is executed. This allows comparison of signed integers.

Status Bits

Status bits are not affected

Example

When the content of R6 is less than the memory pointed to by R7 the programme continues at label EDE.

```
CMP     @R7,R6      ; R6 < (R7)?, compare on signed numbers
JL      EDE         ; Yes, R6 < (R7)
.....               ; No, proceed
.....
.....
```

JMP Jump unconditionally

Syntax JMP label

Operation PC + 2*offset -> PC

Description The 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter.

Status Bits Status bits are not affected

Hint This 1word instruction replaces the BRANCH instruction in the range of -511 to +512 words relative to the current programme counter.

JN

Jump if negative

Syntax JN label

Operation if N = 1: PC + 2*offset -> PC
if N = 0: execute following instruction

Description The negative bit N of the Status Register is tested. If it is set, the 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter. If N is reset, the next instruction following the jump is executed.

Status Bits Status bits are not affected

Example The result of a computation in R5 is to be subtracted from COUNT. If the result is negative, COUNT is to be cleared and the programme continues execution in another path.

```
        SUB   R5,COUNT    ; COUNT - R5 -> COUNT
        JN    L$1         ; If negative continue with COUNT=0at PC=L$1
        ......            ; Continue with COUNT≥0
        ......
        ......
        ......
L$1     CLR   COUNT
        ......
        ......
        ......
```

JNC
JLO

Jump if carry not set
Jump if lower

Syntax

JNC label
JNC label

Operation

if C = 0: PC + 2*offset -> PC
if C = 1: execute following instruction

Description

The Carry Bit C of the Status Register is tested. If it is reset, the 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter. If C is set, the next instruction following the jump is executed. JNC (jump if no carry/lower) is used for the comparison of unsigned numbers (0 to 65536).

Status Bits

status bits are not affected

Example

The result in R6 is added in BUFFER. If an overflow occurs an error handling routine at address ERROR is going to be used.

```
        ADD     R6,BUFFER       ; BUFFER + R6 -> BUFFER
        JNC     CONT            ; No carry, jump to CONT
ERROR   ......                  ; Error handler start
        ......
        ......
        ......
CONT    ......                  ; Continue with normal programme flow
        ......
        ......
```

Example

Branch to STL2 if byte STATUS contains 1 or 0.

```
        CMP.B   #2,STATUS
        JLO     STL2            ; STATUS < 2
        ......                  ; STATUS ≥ 2, continue here
```

JNE, JNZ

Jump if not equal, Jump if not zero

Syntax JNE label, JNZ label

Operation if Z = 0: PC + 2*offset -> PC
if Z = 1: execute following instruction

Description The Zero Bit Z of the Status Register is tested. If it is reset, the 10-bit signed offset contained in the LSB's of the instruction is added to the Programme Counter. If Z is set, the next instruction following the jump is executed.

Status Bits Status bits are not affected

Example Jump to address TONI if R7 and R8 have different contents

```
CMP   R7,R8      ; COMPARE R7 WITH R8
JNE   TONI       ; if different: Jump
.....            ; if equal, continue
```

MOV[.W] Move source to destination

Syntax MOV src,dst or MOV.W src,dst

Operation src -> dst

Description The source operand is moved to the destination.
The source operand is not affected, the previous contents of the destination are lost.

Status Bits Status bits are not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The contents of table EDE (word data) are copied to table TOM. The length of the tables should be 020h locations.

```
        MOV   #EDE,R10                ; Prepare pointer
        MOV   #020h,R9                ; Prepare counter
Loop    MOV   @R10+,TOM-EDE-2(R10)    ; Use pointer in R10 for both tables
        DEC   R9                      ; Decrement counter
        JNZ   Loop                    ; Counter ≠ 0, continue copying
        ......                        ; Copying completed
        ......
        ......
```

MOV.B Move source to destination

Syntax MOV.B src,dst

Operation src -> dst

Description The source operand is moved to the destination.
The source operand is not affected, the previous contents of the destination are lost.

Status Bits Status bits are not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The contents of table EDE (byte data) are copied to table TOM. The length of the tables should be 020h locations.

```
        MOV   #EDE,R10                    ; Prepare pointer
        MOV   #020h,R9                    ; Prepare counter
Loop    MOV.B @R10+,TOM-EDE-1(R10)        ; Use pointer in R10 for
                                          ; both tables
        DEC   R9                          ; Decrement counter
        JNZ   Loop                        ; Counter ≠ 0, continue
                                          ; copying
        ......                            ; Copying completed
        ......
        ......
```

*** NOP** No operation

Syntax NOP

Operation None

Emulation MOV #0,#0

Description No operation is performed. The instruction may be used for the elimination of instructions during the software check or for defined waiting times.

Status Bits Status bits are not affected

The NOP instruction is mainly used for two purposes:
- hold one, two or three memory words
- adjust software timing

Note: Other instructions can be used to emulate no operation

Other instructions can be used to emulate no-operation instruction using different numbers of cycles and different numbers of code words.

Examples:

```
MOV   0(R4),0(R4)     ; 6 cycles, 3 words
MOV   @R4,0(R4)       ; 5 cycles, 2 words
BIC   #0,EDE(R4)      ; 4 cycles, 2 words
JMP   $+2             ; 2 cycles, 1 word
BIC   #0,R5           ; 1 cycles, 1 word.
```

*** POP[.W]** Pop word from stack to destination

Syntax POP dst

Operation @SP -> dst
SP + 2 -> SP

Emulation MOV @SP+,dst or MOV.W @SP+,dst

Description The stack location pointed to by the Stack Pointer (TOS) is moved to the destination. The Stack Pointer is incremented by two afterwards.

Status Bits Status bits are not affected

Example The contents of R7 and the Status Register are restored from the stack.

```
POP   R7          ; Restore R7
POP   SR          ; Restore status register
```

Note: The system Stack Pointer SP, Note 1

The system Stack Pointer SP is always incremented by two, independent of the byte suffix. This is mandatory since the system Stack Pointer is used not only by POP instructions; it is also used by the RETI instruction.

*** POP.B** Pop byte from stack to destination

Syntax POP.B dst

Operation
@SP -> dst
SP + 2 -> SP

Emulation MOV.B @SP+,dst

Description The stack location pointed to by the Stack Pointer (TOS) is moved to the destination. The Stack Pointer is incremented by two afterwards.

Status Bits Status bits are not affected

Example The content of RAM byte LEO is restored from the stack.

```
POP.B   LEO     ; The Low byte of the stack is moved to LEO.
```

Example The content of R7 is restored from the stack.

```
POP.B   R7      ; The Low byte of the stack is moved to R7,
                ; the High byte of R7 is 00h
```

Example The contents of the memory pointed to by R7 and the Status Register are restored from the stack.

```
POP.B   0(R7)   ; The Low byte of the stack is moved to the
                ; the byte which is pointed to by R7
                : Ex1: R7 = 203h
                ;      Mem(R7) = Low Byte of system stack
                ;
                : Ex2: R7 = 20Ah
                ;      Mem(R7) = Low Byte of system stack
POP     SR
```

Note: The system Stack Pointer, Note 2

The system Stack Pointer SP is always incremented by two, independent of the byte suffix. This is mandatory since the system Stack Pointer is used not only by POP instructions; it is also used by the RETI instruction.

PUSH[.W] Push word onto stack

Syntax PUSH src or PUSH.W src

Operation SP - 2 → SP
src → @SP

Description The Stack Pointer is decremented by two, then the source operand is moved to the RAM word addressed by the Stack Pointer (TOS).

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The contents of the Status Register and R8 are saved on the stack.

```
PUSH    SR      ; save status register
PUSH    R8      ; save R8
```

Note: The system Stack Pointer, Note 3

The system Stack Pointer SP is always decremented by two, independent of the byte suffix. This is mandatory since the system Stack Pointer is used not only by PUSH instruction; it is also used by the interrupt routine service.

PUSH.B

Push byte onto stack

Syntax PUSH.B src

Operation SP - 2 → SP
src → @SP

Description The Stack Pointer is decremented by two, then the source operand is moved to the RAM byte addressed by the Stack Pointer (TOS).

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The content of the peripheral TCDAT is saved on the stack.

```
PUSH.B  &TCDAT   ; save data from 8bit peripheral module,
                 ; address TCDAT, onto stack
```

Note: The system Stack Pointer, Note 4

The system Stack Pointer SP is always decremented by two, independent of the byte suffix. This is mandatory since the system Stack Pointer is used not only by PUSH instruction; it is also used by the interrupt routine service.

* RET

Return from subroutine

Syntax RET

Operation @SP→ PC
SP + 2 → SP

Emulation MOV @SP+,PC

Description The return address pushed onto the stack by a CALL instruction is moved to the Programme Counter. The programme continues at the code address following the subroutine call.

Status Bits Status bits are not affected

RETI

Return from Interrupt

Syntax

RETI

Operation

TOS → SR
SP + 2 → SP
TOS → PC
SP + 2 → SP

Description

1. The status register is restored to the value at the beginning of the interrupt service routine. This is performed by replacing present the contents of SR with the contents of TOS memory. The stack pointer SP is incremented by two.
2. The programme counter is restored to the value at the beginning of interrupt service. This is the consecutive step after the interrupted programme flow. Restore is performed by replacing present contents of PC with the contents of TOS memory. The stack pointer SP is incremented.

Status Bits

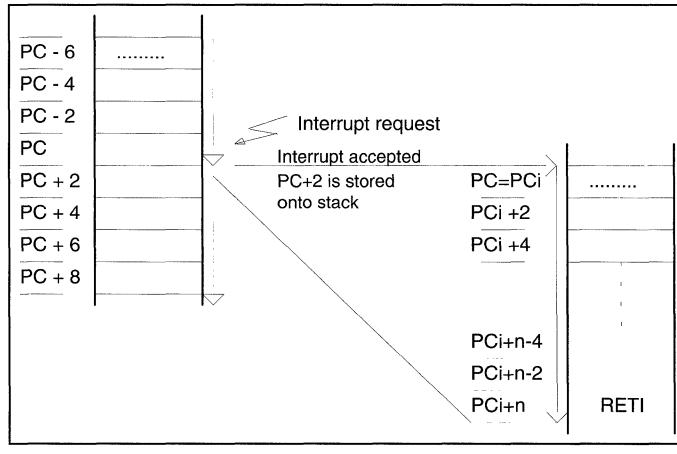
N: restored from system stack
Z: restored from system stack
C: restored from system stack
V: restored from system stack

Mode Bits

OscOff, **CPUOff** and **GIE** are restored from system stack

Example

Main programme is interrupted



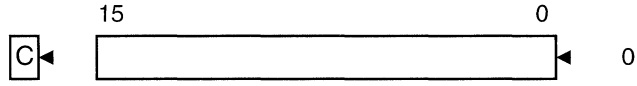
* RLA[.W] Rotate left arithmetically

Syntax RLA dst or RLA.W dst

Operation C <- MSB <- MSB-1 .... LSB+1 <- LSB <- 0

Emulation ADD dst,dst

Description The destination operand is shifted left one position. The MSB is shifted into the carry C, the LSB is filled with 0. The RLA instruction acts as a signed multiplication with 2. An overflow occurs if dst ≥ 04000h and dst < 0C000h before operation is performed: the result has changed sign.



Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the MSB
V: Set if an arithmetic overflow occurs -
the initial value is 04000h ≤ dst < 0C000h;
otherwise it is reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R7 is multiplied by 4.

```
RLA    R7      ; Shift left R7  (x 2) - emulated by  ADD  R7,R7
RLA    R7      ; Shift left R7  (x 4) - emulated by  ADD  R7,R7
```

Note: RLA substitution

The Assembler does not recognize the instruction

RLA @R5+.

It must be substituted by

ADD @R5+,-2(R5).

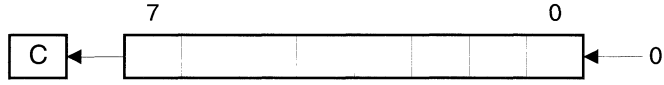
* RLA.B Rotate left arithmetically

Syntax RLA.B dst

Operation C <- MSB <- MSB-1 .... LSB+1 <- LSB <- 0

Emulation ADD.B dst,dst

Description The destination operand is shifted left one position. The MSB is shifted into the carry C, the LSB is filled with 0. The RLA instruction acts as a signed multiplication with 2. An overflow occurs if dst ≥ 040h and dst < 0C0h before operation is performed: the result has changed sign.



Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the MSB
V: Set if an arithmetic overflow occurs:
the initial value is 040h ≤ dst < 0C0h;
otherwise it is reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Lowbyte of R7 is multiplied by 4.

```
RLA.B   R7      ; Shift left Lowbyte of R7  (x 2) - emulated by
                ; ADD.B  R7,R7
RLA.B   R7      ; Shift left Lowbyte of R7  (x 4) - emulated by
                ; ADD.B  R7,R7
```

Note: RLA.B substitution

The Assembler does not recognize the instruction

RLA.B @R5+.

It must be substituted by

ADD.B @R5+,-1(R5).

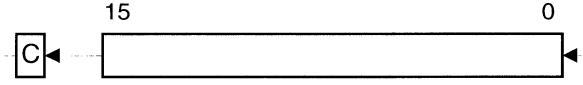
* RLC[.W] Rotate left through carry

Syntax RLC dst or RLC.W dst

Operation C <- MSB <- MSB-1 .... LSB+1 <- LSB <- C

Emulation ADDC dst,dst

Description The destination operand is shifted left one position. The carry C is shifted into the LSB, the MSB is shifted into the carry C.



Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the MSB
V: Set if arithmetic overflow occurs otherwise reset
Set if 03FFFh < $dst_{initial}$ < 0C000h, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R5 is shifted left one position.

```
RLC     R5          ; (R5 x 2) + C -> R5
```

Example The information of input P0IN.1 is to be shifted into LSB of R5.

```
BIT.B   #2,&P0IN    ; Information -> Carry
RLC     R5          ; Carry=P0in.1 -> LSB of R5
```

Note: RLC substitution

The Assembler does not recognize the instruction

```
RLC     @R5+.
```

It must be substituted by

```
ADDC    @R5+,-2(R5).
```

* RLC.B

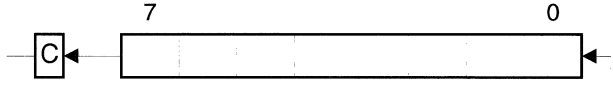
Rotate left through carry

Syntax RLC.B dst

Operation C <- MSB <- MSB-1 .... LSB+1 <- LSB <- C

Emulation ADDC.B dst,dst

Description The destination operand is shifted left one position. The carry C is shifted into the LSB, the MSB is shifted into the carry C.



Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the MSB
V: Set if arithmetic overflow occurs otherwise reset
Set if 03Fh < $dst_{initial}$ < 0C0h otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Content of MEM(LEO) is shifted left one position.

```
RLC.B   LEO             ; Mem(LEO) x 2 + C -> Mem(LEO)
```

Example The information of input P0IN.1 is to be shifted into LSB of R5.

```
BIT.B   #2,&P0IN        ; Information -> Carry
RLC.B   R5              ; Carry=P0in.1 -> LSB of R5
                        ; High byte of R5 is reset
```

Note: RLC.B emulated

The Assembler does not recognize the instruction

RLC.B @R5+.

It must be substituted by

ADDC.B @R5+,-1(R5).

RRA[.W] Rotate right arithmetically

Syntax RRA dst or RRA.W dst

Operation MSB -> MSB, MSB -> MSB-1, MSB-1 -> MSB-2 .... LSB+1 -> LSB, LSB -> C

Description The destination operand is shifted right one position. The MSB is shifted into the MSB, the MSB is shifted into the MSB-1, the LSB+1 is shifted into the LSB.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the LSB
V: Reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R5 is shifted right one position. The MSB remains with the old value. It operates equal to an arithmetic division by 2.

```
                RRA     R5          ; R5/2 -> R5

;               The value in R5 is multiplied by 0.75 (0.5 + 0.25)
;
                PUSH    R5          ; hold R5 temporarily using stack
                RRA     R5          ; R5 x 0.5  ->  R5
                ADD     @SP+,R5     ; R5 x 0.5 + R5 = 1.5 x R5  -> R5
                RRA     R5          ; (1.5 x R5) x 0.5 = 0.75 x R5  -> R5
                ......
                ......
                ......
; OR
;
                RRA     R5          ; R5 x 0.5  ->  R5
                PUSH    R5          ; R5 x 0.5  ->  TOS
                RRA     @SP         ; TOS x 0.5 = 0.5 x R5 x 0.5 = 0.25 x R5  -> TOS
                ADD     @SP+,R5     ; R5 x 0.5 + R5 x 0.25 = 0.75 x R5  -> R5
                ......
                ......
                ......
```

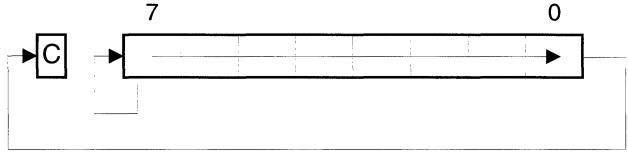
RRA.B

Rotate right arithmetically

Syntax RRA.B dst

Operation MSB -> MSB, MSB -> MSB-1, MSB-1 -> MSB-2 .... LSB+1 -> LSB, LSB -> C

Description The destination operand is shifted right one position. The MSB is shifted into the MSB, the MSB is shifted into the MSB-1, the LSB+1 is shifted into the LSB.



Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the LSB
V: Reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The Lowbyte of R5 is shifted right one position. The MSB remains with the old value. It operates equal to an arithmetic division by 2.

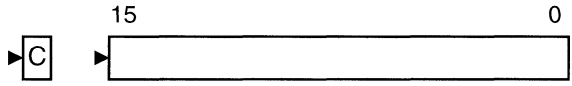
```
                RRA.B   R5          ; R5/2 -> R5: Operation is on Low byte only
                                    ; High byte of R5 is reset

;               The value in R5 - Low byte only! - is multiplied by 0.75 (0.5 + 0.25)
;
                PUSH.B  R5          ; hold Low byte of R5 temporarily using stack
                RRA.B   R5          ; R5 x 0.5  ->  R5
                ADD.B   @SP+,R5     ; R5 x 0.5 + R5 = 1.5 x R5  -> R5
                RRA.B   R5          ; (1.5 x R5) x 0.5 = 0.75 x R5  -> R5
                ......
                ......
                ......
; OR
;
                RRA.B   R5          ; R5 x 0.5  ->  R5
                PUSH.B  R5          ; R5 x 0.5  ->  TOS
                RRA.B   @SP         ; TOS x 0.5 = 0.5 x R5 x 0.5 = 0.25x R5  -> TOS
                ADD.B   @SP+,R5     ; R5 x 0.5 + R5 x 0.25 = 0.75 x R5  -> R5
                ......
                ......
                ......
```

RRC[.W] Rotate right through carry

Syntax RRC dst or RRC.W dst

Operation C -> MSB -> MSB-1 .... LSB+1 -> LSB -> C

Description The destination operand is shifted right one position. The carry C is shifted into the MSB, the LSB is shifted into the carry C.



Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the LSB
V: Set if initial destination is positive and initial Carry is set, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R5 is shifted right one position. The MSB is loaded with 1.

```
SETC            ; PREPARE CARRY FOR MSB
RRC   R5        ; R5/2 + 8000h -> R5
```

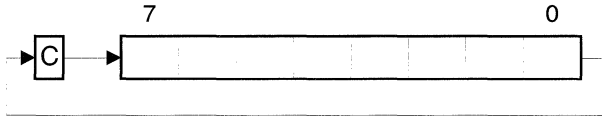
RRC.B

Rotate right through carry

Syntax RRC dst

Operation C -> MSB -> MSB-1 .... LSB+1 -> LSB -> C

Description The destination operand is shifted right one position. The carry C is shifted into the MSB, the LSB is shifted into the carry C.



Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the LSB
V: Set if initial destination is positive and initial Carry is set, otherwise reset

OscOff, **CPUOff** and **GIE** are not affected

Example R5 is shifted right one position. The MSB is loaded with 1.

```
SETC            ; PREPARE CARRY FOR MSB
RRC.B   R5      ; R5/2 + 80h -> R5; Low byte of R5 is used
```

*** SBC[.W]** Subtract borrow*) from destination

Syntax SBC dst or SBC.W dst

Operation dst + 0FFFFh + C -> dst

Emulation SUBC #0,dst

Description The carry C is added to the destination operand minus one. The previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Reset if dst was decremented from 0000 to 0FFFFh, set otherwise
V: Set if initially C=0 and dst=08000h

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 16-bit counter pointed to by R13 is subtracted from a 32-bit counter pointed to by R12.

```
SUB    @R13,0(R12)  ; Subtract LSDs
SBC    2(R12)       ; Subtract carry from MSD
```

Note: Borrow is treated as a .NOT. carry 1

The borrow is treated as a .NOT. carry:

Borrow	Carry bit
Yes	0
No	1

* SBC.B

Subtract borrow*) from destination

Syntax SBC.B dst

Operation dst + 0FFh + C -> dst

Emulation SUBC.B #0,dst

Description The carry C is added to the destination operand minus one

the borrow is subtracted from the destination operand.
The previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Reset if dst was decremented from 0000 to 0FFFFh, set otherwise
V: Set if initially C=0 and dst=080h

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 8bit counter pointed to by R13 is subtracted from a 16bit counter pointed to by R12.

```
SUB.B   @R13,0(R12)     ; Subtract LSDs
SBC.B   1(R12)          ; Subtract carry from MSD
```

Note: Borrow is treated as a .NOT. carry 2

The borrow is treated as a .NOT. carry:

Borrow	Carry bit
Yes	0
No	1

* SETC Set carry bit

Syntax SETC

Operation 1 -> C

Emulation BIS #1,SR

Description The Carry Bit C is set, an often necessary operation.

Status Bits
N: Not affected
Z: Not affected
C: Set
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Emulation of the decimal subtraction:
Subtract R5 from R6 decimally
Assume that R5=3987 and R6=4137

```
DSUB    ADD     #6666h,R5       ; Move content R5 from 0-9 to 6-0Fh
                                ; R5 = 03987 + 6666 = 09FEDh
        INV     R5              ; Invert this(result back to 0-9)
                                ; R5 = .NOT. R5 = 06012h
        SETC                    ; Prepare carry = 1
        DADD    R5,R6           ; Emulate subtraction by adding of:
                                ; (10000 - R5 - 1)
                                ; R6 = R6 + R5 + 1
                                ; R6 = 4137 + 06012 + 1 = 1 0150 = 0150
```

*** SETN** Set Negative bit

Syntax SETN

Operation 1 -> N

Emulation BIS #4,SR

Description The Negative bit N is set.

Status Bits
N: Set
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

* SETZ Set Zero bit

Syntax SETZ

Operation 1 -> Z

Emulation BIS #2,SR

Description The Zero bit Z is set.

Status Bits
N: Not affected
Z: Set
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

SUB[.W] subtract source from destination

Syntax SUB src,dst or SUB.W src,dst

Operation dst + .NOT.src + 1 -> dst
or
[(dst - src -> dst)]

Description The source operand is subtracted from the destination operand. This is made by adding of the 1's complement of the source operand and the constant 1. The source operand is not affected, the previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the MSB of the result, reset if not
Set to 1 if no borrow, reset if borrow.
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example See example at the SBC instruction

Note: Borrow is treated as a .NOT. carry 3

The borrow is treated as a .NOT. carry:

Borrow	Carry bit
Yes	0
No	1

SUB.B

subtract source from destination

Syntax SUB.B src,dst

Operation dst + .NOT.src + 1 -> dst
or
(dst - src -> dst)

Description The source operand is subtracted from the destination operand. This is made by adding of the 1's complement of the source operand and the constant 1. The source operand is not affected, the previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the MSB of the result, reset if not
Set to 1 if no borrow, reset if borrow.
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example See example at the SBC.B instruction

Note: Borrow is treated as a .NOT. carry 4

The borrow is treated as a .NOT. carry:

Borrow	Carry bit
Yes	0
No	1

SUBC[.W]SBB[.W] subtract source and borrow/.NOT. carry from destination

Syntax

SUBC src,dst or SUBC.W src,dst or
SBB src,dst or SBB.W src,dst

Operation

dst + .NOT.src + C -> dst
or
(dst - src - 1 + C -> dst)

Description The source operand is subtracted from the destination operand. This is made by adding of the 1's complement of the source operand and the carry C. The source operand is not affected, the previous contents of the destination are lost.

Status Bits

N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the MSB of the result, reset if not
Set to 1 if no borrow, reset if borrow.
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Two floating point mantissas (24bits) are subtracted .
LSB's are in R13 resp. R10, MSB's are in R12 resp. R9.

```
SUB.W   R13,R10     ; 16bit part, LSB's
SUBC.B  R12,R9      ;  8bit part, MSB's
```

Note: Borrow is treated as a .NOT. carry 5

The borrow is treated as a .NOT. carry:	Borrow	Carry bit
	Yes	0
	No	1

SUBC.B,SBB.B subtract source and borrow/.NOT. carry from destination

Syntax SUBC.B src,dst or SBB.B src,dst

Operation dst + .NOT.src + C -> dst
or
(dst - src - 1 + C -> dst)

Description The source operand is subtracted from the destination operand. This is made by adding of the 1's complement of the source operand and the carry C. The source operand is not affected, the previous contents of the destination are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the MSB of the result, reset if not
Set to 1 if no borrow, reset if borrow.
V: Set if an arithmetic overflow occurs, otherwise reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The 16-bit counter pointed to by R13 is subtracted from a 16-bit counter in R10 and R11(MSD).

```
SUB.B   @R13+,R10   ; Subtract LSDs without carry
SUBC.B  @R13,R11    ; Subtract MSDs with carry
...                 ; resulting fron the LSDs
```

Note: Borrow is treated as a .NOT. carry 6

The borrow is treated as a .NOT. carry:

Borrow	Carry bit
Yes	0
No	1

SWPB Swap bytes

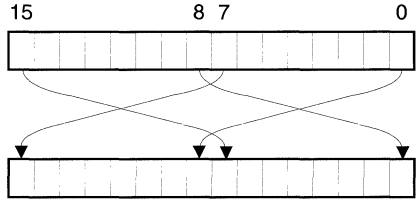
Syntax SWPB dst

Operation bits 15 to 8 <-> bits 7 to 0

Description The high and the low bytes of the destination operand are exchanged.

Status Bits
N: Not affected
Z: Not affected
C: Not affected
V: Not affected

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected



Example

```
MOV   #040BFh,R7      ; 0100000010111111 -> R7
SWPB  R7              ; 1011111101000000 in R7
```

Example The value in R5 is multiplied by 256. The result is stored in R5,R4

```
SWPB  R5              ;
MOV   R5,R4           ;Copy the swapped value to R4
BIC   #0FF00h,R5      ;Correct the result
BIC   #00FFh,R4       ;Correct the result
```

SXT Extend Sign

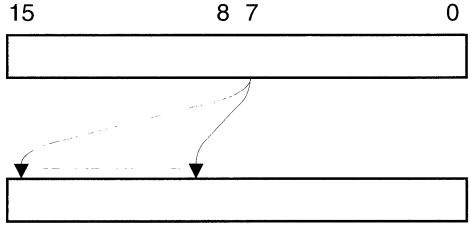
Syntax SXT dst

Operation Bit 7 -> Bit 8 ..... Bit 15

Description The sign of the Low byte is extended into the High byte.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (.NOT. Zero)
V: Reset

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected



Example R7 is loaded with Timer/Counter value. The operation of the sign extend instruction expands the bit8 to bit15 with the value of bit7.
R7 is added then to R6 where it is accumulated.

```
MOV.B   &TCDAT,R7   ; TCDAT = 080h: .... .... 1000 0000
SXT     R7          ; R7 = 0FF80h:   1111 1111 1000 0000
ADD     R7,R6       ; add value of EDE to 16bit ACCU
```

* TST[.W] Test destination

Syntax TST dst or TST.W dst

Operation dst + 0FFFFh + 1

Emulation CMP #0,dst

Description The destination operand is compared to zero. The status bits are set according to the result. The destination is not affected.

Status Bits
N: Set if destination is negative, reset if positive
Z: Set if destination contains zero, reset otherwise
C: Set
V: Reset.

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example R7 is tested. If it is negative continue at R7NEG; if it is positive but not zero continue at R7POS.

```
        TST   R7        ; Test R7
        JN    R7NEG     ; R7 is negative
        JZ    R7ZERO    ; R7 is zero
R7POS   .....           ; R7 is positive but not zero
        .....
        .....
R7NEG   .....           ; R7 is negative
        .....
        .....
R7ZERO  .....           ; R7 is zero
        .....
        .....
```

* TST.B Test destination

Syntax TST.B dst

Operation dst + 0FFh + 1

Emulation CMP.B #0,dst

Description The destination operand is compared to zero (R15). The status bits are set according to the result. The destination is not affected.

Status Bits
N: Set if destination is negative, reset if positive
Z: Set if destination contains zero, reset otherwise
C: Set
V: Reset.

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example Lowbyte of R7 is tested. If it is negative continue at R7NEG; if it is positive but not zero continue at R7POS.

```
        TST.B  R7          ; Test Low byte of R7
        JN     R7NEG       ; Low byte of R7 is negative
        JZ     R7ZERO      ; Low byte of R7 is zero
R7POS   .....              ; Low byte of R7 is positive but not zero
        .....
        .....
R7NEG   .....              ; Lowbyte of R7 is negative
        .....
        .....
R7ZERO  .....              ; Lowbyte of R7 is zero
        .....
        .....
```

XOR[.W] Exclusive OR of source with destination

Syntax XOR src,dst or XOR.W src,dst

Operation src .XOR. dst -> dst

Description The source operand and the destination operand are OR'ed exclusively. The result is placed into the destination. The source operand is not affected.

Status Bits
N: Set if MSB of result is set, reset if not set
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (= .NOT. Zero)
V: Set if both operands are negative

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The bits set in R6 toggle the bits in the RAM word TONI.

```
XOR     R6,TONI    ; Toggle bits of word TONI on the bits set in R6
```

XOR.B

Exclusive OR of source with destination

Syntax XOR.B src,dst

Operation src .XOR. dst -> dst

Description The source operand and the destination operand are OR'ed exclusively. The result is placed into the destination. The source operand is not affected.

Status Bits
N: Set if MSB of result is set, reset if not set
Z: Set if result is zero, reset otherwise
C: Set if result is not zero, reset otherwise (= .NOT. Zero)
V: Set if both operands are negative

Mode Bits **OscOff**, **CPUOff** and **GIE** are not affected

Example The bits set in R6 toggle the bits in the RAM byte TONI.

```
XOR.B R6,TONI  ; Toggle bits in word TONI on bits set in Low byte of
R6,
```

Example Reset bits in Lowbyte of R7 to 0 that are different to bits in RAM byte EDE.

```
XOR.B   EDE,R7    ; Set different bit to '1s'
INV.B   R7        ; Invert Lowbyte, Highbyte is 0h
```

2.4 Macro instructions emulated with several instructions

The following table shows the instructions which need more words if emulated by the reduced instruction set. This is not of big concern due to the rare use of them. The immediate values -1, 0, +1, 2, 4 and 8 are provided by the Constant Generator Registers R2/CG1 and R3/CG2.

Emulated instruction			Instruction flow		Comment
ABS	**dst**		TST	dst	; Absolute value of destination
			JN	L$0	; Destination is negative
		L$1	...		; Destination is positive
			...		
			...		
		L$0	INV	dst	; Convert negative destination
			INC	dst	; to positive
			JMP	L$1	
DSUB	**src,dst**		ADD	#6666h,src	; Decimal subtraction
			INV	src	; Source is destroyed!
			SETC		
			DADD	src,dst	; DST - SRC (dec)
NEG	**dst**		INV	dst	; Negation of destination
			INC	dst	
RL	**dst**		ADD	dst,dst	; Rotate left circularly
			ADDC	#0,dst	
RR	**dst**		CLRC		; Rotate right circularly
			RRC	dst	
			JNC	L$1	
			BIS	#8000h,dst	
		L$1	...		

2.5 Stack pointer addressing

The placement of the Stack Pointer inside the register space allows a lot of features not possible with the normal allocation outside the register space.

```
MOV   Rn,SP      ; Load SP with the contents of Rn

MOV   @Rn,SP     ; Load SP with the contents of the word pointed to by
                 ; Rn

MOV   @Rn+,SP    ; Same as above with autoincrement of Rn

MOV   X(Rn),SP   ; Load SP with the contents of a table pointed to by Rn.
                 ; X defines the offset relative to the table start

MOV   #n,SP      ; Load SP with a constant n (e.g. for initialization)

MOV   ADDR,SP    ; Load SP with the contents of word ADDR

MOV   &ADDR,SP   ; Load SP with the contents of absolute address ADDR

MOV   SP,Rn      ; Copy SP to Rn (e.g. for later restoring)

MOV   @SP,Rn     ; Move top of stack (TOS) to Rn

MOV   @SP+,Rn    ; Pop stack item to Rn

MOV   X(SP),Rn   ; Move a stack item relative to the SP to Rn

MOV   Rn,0(SP)   ; Replace TOS by contents of Rn

MOV   Rn,X(SP)   ; Replace item on the stack. X defines the offset relative
                 ; to the SP (TOS)

INCD  SP         ; Remove TOS item
```

The Stack Pointer allows the transfer of arguments in several ways. The following example shows a CALL with arguments and the handling inside of the subroutine:

```
CALL    #SUBROUT
.BYTE   MODE,CODE    ; Control bytes
.WORD   ERRADD       ; Error address, if ERROR occurs
.WORD   ARG1         ; ARGUMENT #1
.WORD   ARG2         ; ARGUMENT #2
...                  ; Continue here after RETURN
;
```

```
SUBROUT  ...                    ; prepare registers
         MOV     @SP,Rn         ; TOS points to control bytes
         ADD     #8,0(SP)       ; Ajust return address
         MOV     @Rn+,Rm        ; Control bytes -> Rm
         MOV     @Rn+,Rx        ; Error address -> Rx
         MOV     @Rn+,Ry        ; ARGUMENT #1 -> Ry
         MOV     @Rn+,Rz        ; ARGUMENT #2 -> Rz
         ...
         RETN                   ; Normal RETURN
ERROR    MOV     Rx,PC          ; Error occured: return address to PC
```

The same subroutine can be called in different ways. The arguments following the call are read by the subroutine and the information is handled appropriately.

2.6 Branch operation

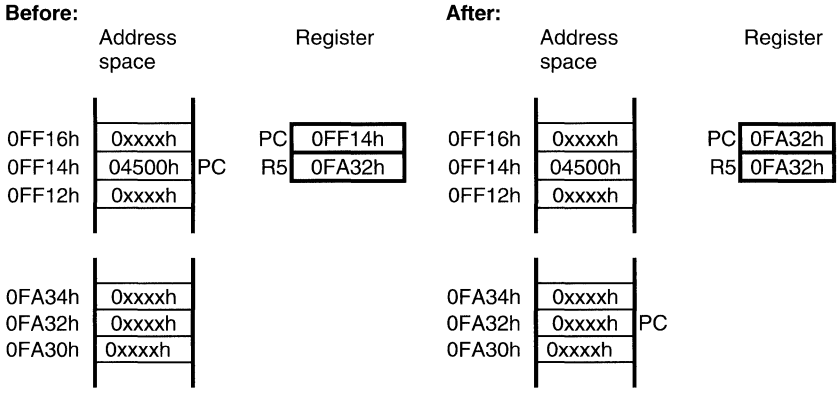
All seven addressing modes can be applied to the Branch instruction. The Branch instruction is emulated by the core instruction MOV source,PC.

Branch and call instructions operate within one segment; both do not manipulate the code segment information.

2.6.1 Indirect Branch, CALL

Indirect Branch

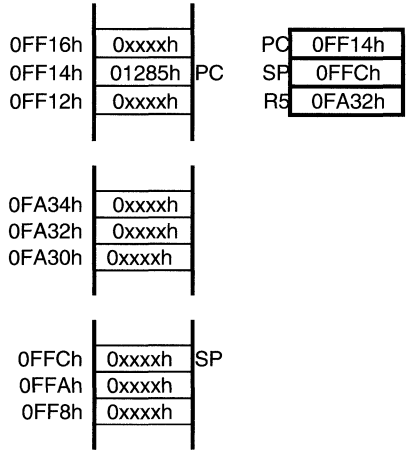
```
BR      R5

MOV     R5,PC           ; Core instruction
```



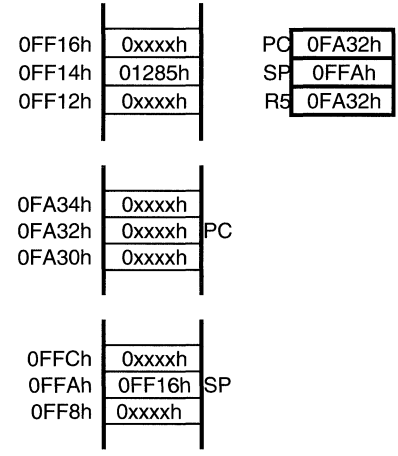
Indirect CALL

CALL R5

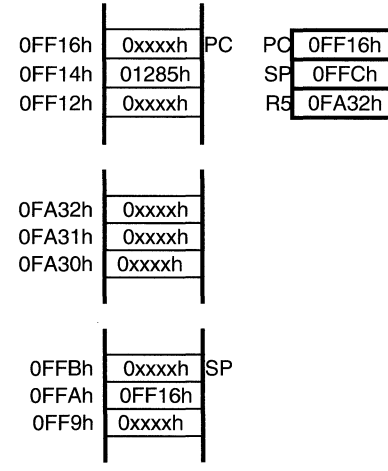
Before:



After : CALL



After : RET



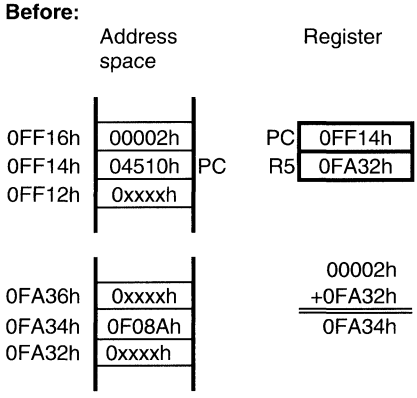
2.6.2 Indirect indexed Branch, CALL

Indirect indexed Branch

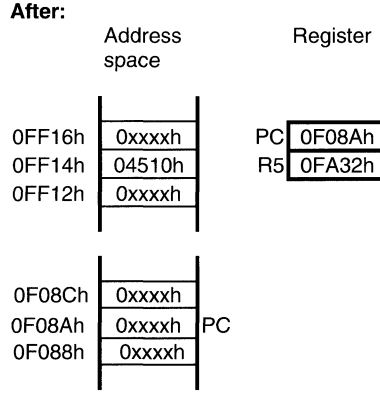
BR 2(R5)

```
MOV     2(R5),PC      ;Core instruction
```

Before:

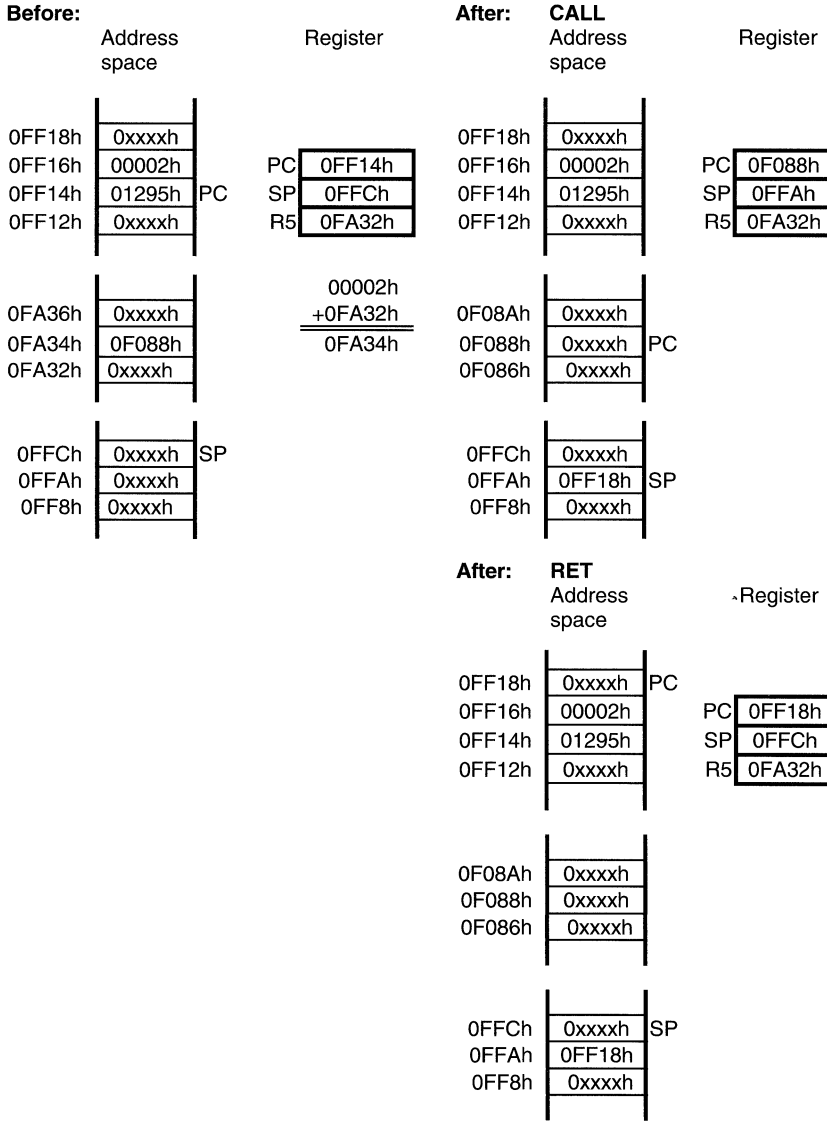


After:



Indirect indexed CALL

CALL 2(R5)

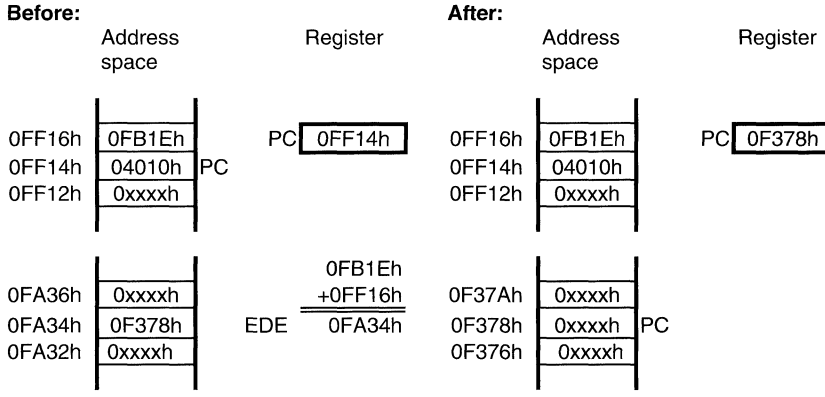


2.6.3 Indirect symbolic Branch, CALL

Indirect symbolic Branch

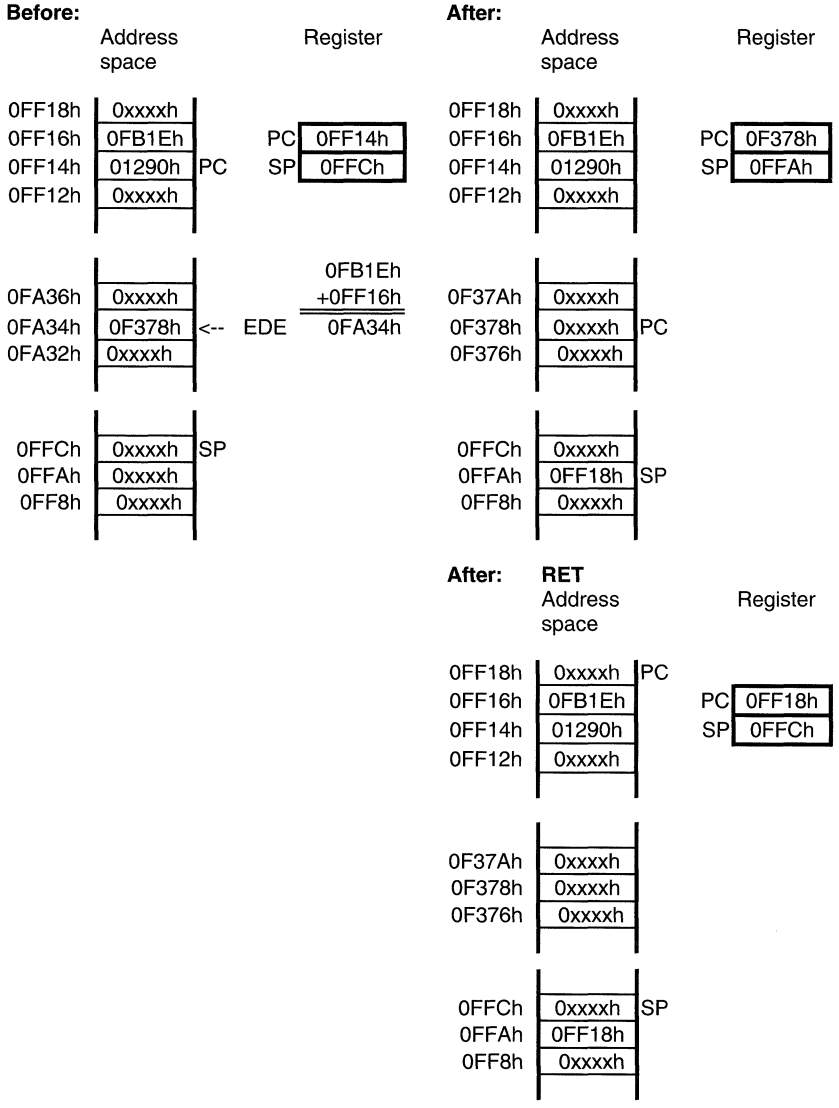
BR EDE

MOV EDE,PC ; Core instruction



Indirect symbolic CALL

CALL EDE



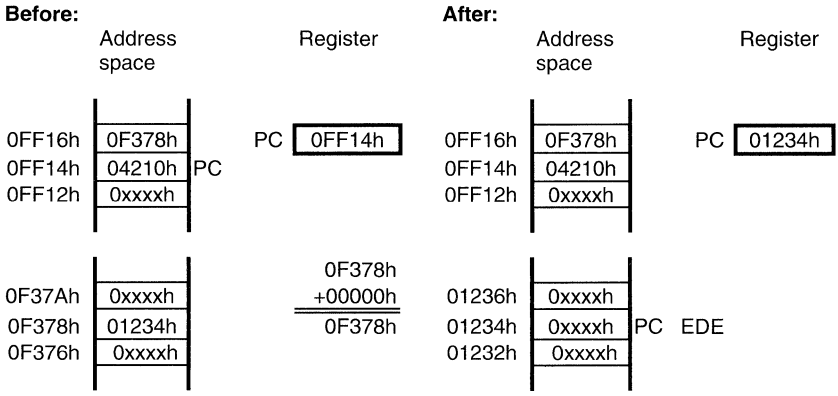
2.6.4 Indirect absolute Branch, CALL

The absolute branch and call instruction in the segmented memory model will result in a branch or call to code segment 0.

Indirect absolute Branch

BR &EDE

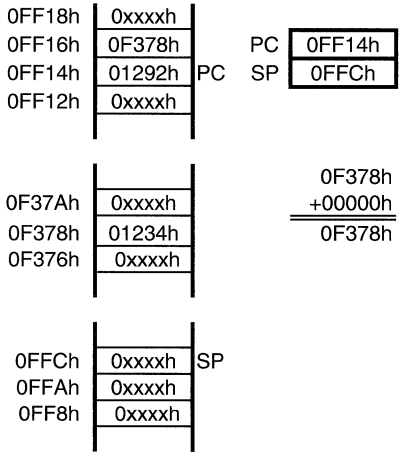
```
MOV     &EDE,PC         ; Core instruction
```



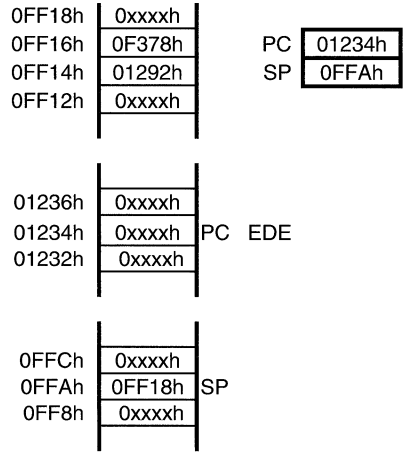
Indirect absolute CALL

CALL &EDE

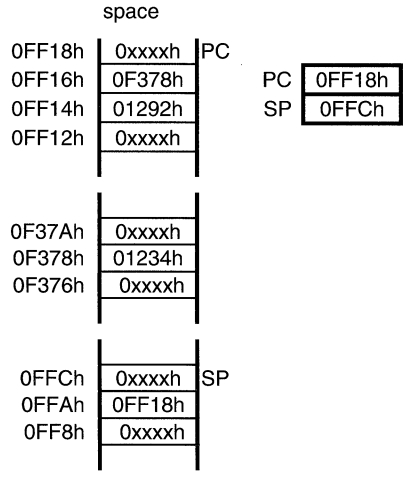
Before:



After:



After: **RET**



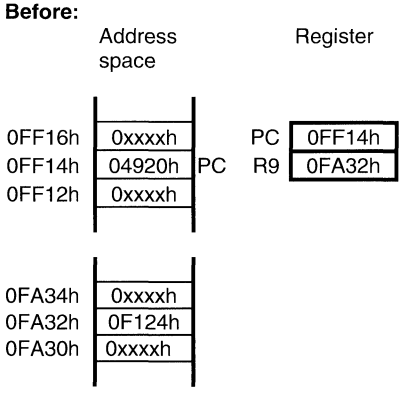
2.6.5 Indirect indirect Branch, CALL

Indirect indirect Branch

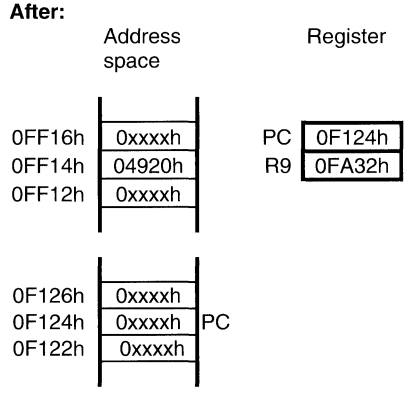
BR @R9

```
MOV     @R9,PC          ; Core instruction
```

Before:

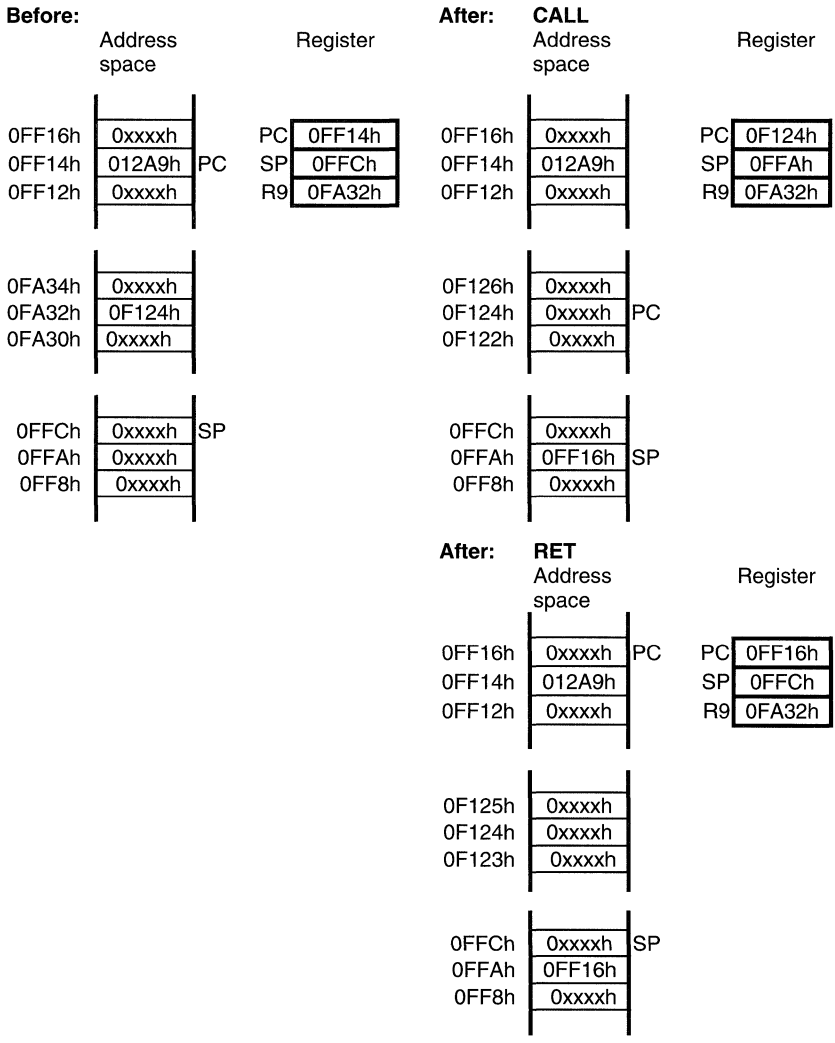


After:



Indirect indirect CALL

CALL @R9



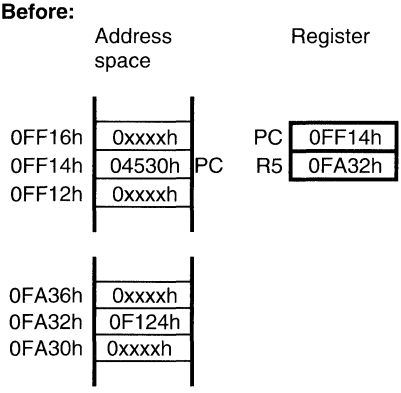
2.6.6 Indirect indirect Branch, CALL with autoincrement

Indirect indirect Branch with autoincrement

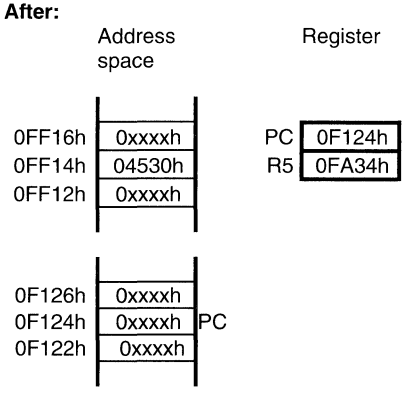
BR @R5+

```
MOV     @R5+,PC         ; Core instruction
```

Before:

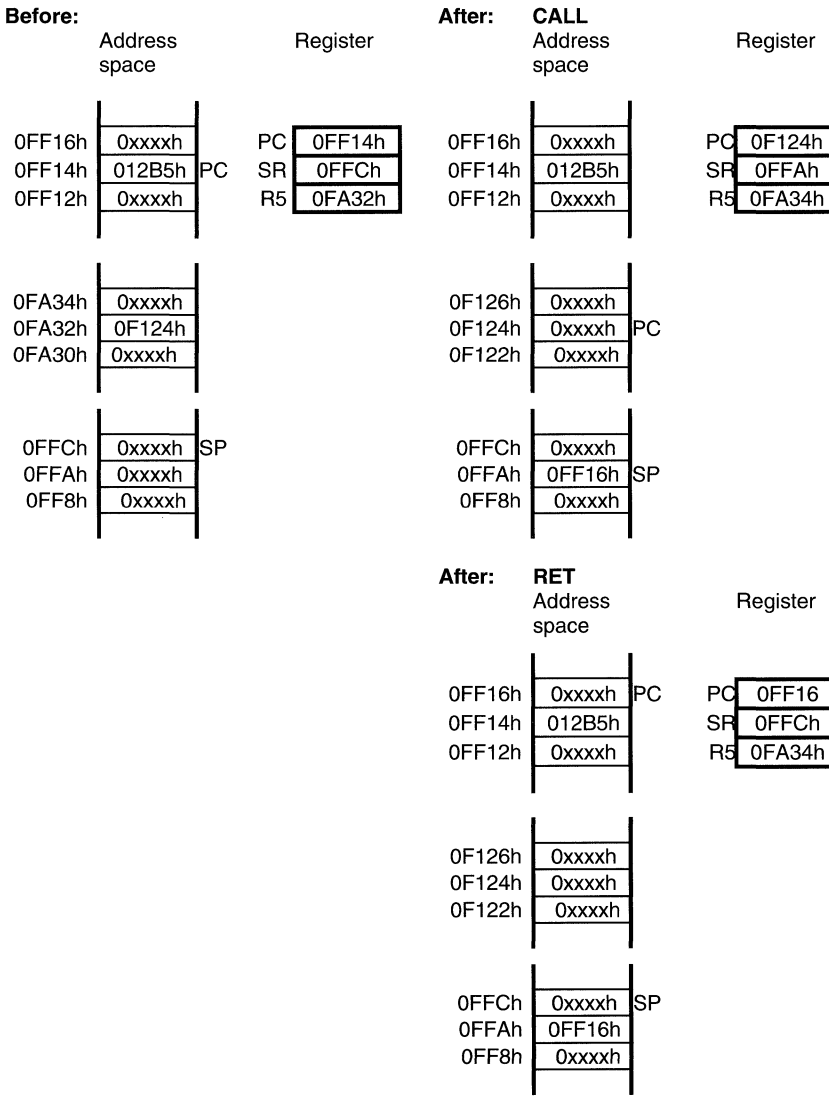


After:



Indirect indirect CALL with autoincrement

CALL @R5+



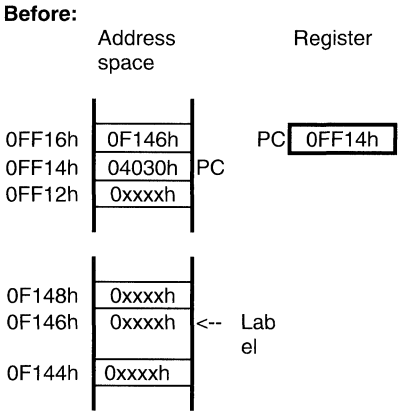
2.6.7 Direct Branch, direct CALL

Branch immediate #N, Branch Label

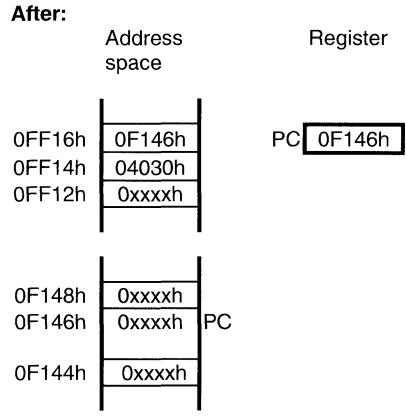
BR #0F146h OR BR #Label

MOV @PC+,PC ; Core instruction

Before:

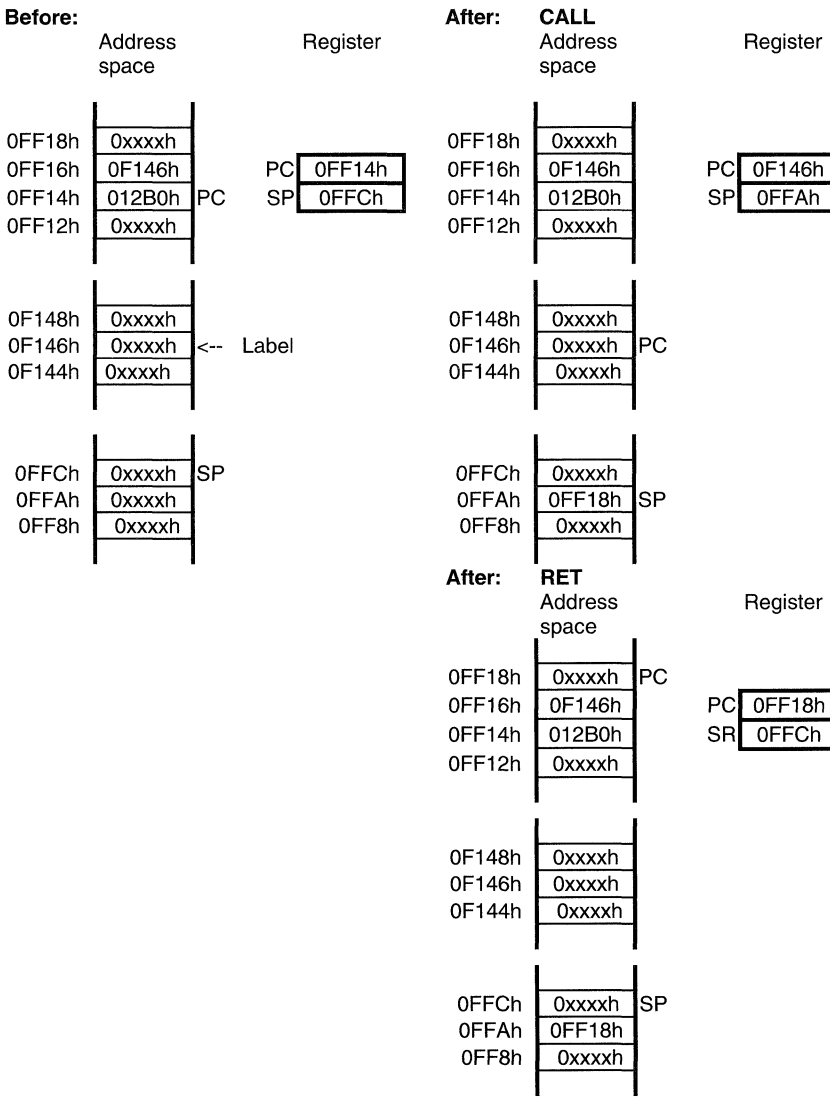


After:



Direct CALL

CALL #0F146h OR CALL #Label



Topics

Figures

3 General Initialization

The most important thing to initialize the processor is the reset vector, which is located at address FFFEh and must point to the starting address of the programme code. The initialization of the stack in the RAM area is important as well. This is done by simple MOV instructions as shown in the following example. To operate with the proper system frequency the system clock generator must be initialized.

3.1 System Clock Generator

The first thing to do after the power up reset is to initialize the system clock generator. The following waiting loop is used to get the initialized system frequency. Then the stack is defined at the RAM address 300h and can range down to 200h (depending on the MSP430 type). Therefore, the first word which is pushed on the stack will be located at 2FEh. If the RAM is used for storing variables the space for the stack will be smaller.

```
STACK     .EQU     300H
SCFQCTL   .EQU     052H
SCFI1     .EQU     051H


          .SECT    "INIT",0F100H  ;STARTADDRESS OF THE RAM VERSION
START                             ;SPECIAL STARTUP FOR FLL
          MOV.B    #1FH,&SCFQCTL  ;LOAD FLL TO RUN WITH 32KHZ*20H
          MOV.B    #80H,&SCFI1    ;LOAD FLL TO RUN WITH 1MHZ
          MOV      #STACK,SP      ;INITIALIZE STACK
          CALL     #RAMCLR        ;USED FOR WAITING, TOO
          ......

          .SECT    "RES_VECT",0FFFEH
          .WORD    START          ; POR, EXT. RESET, WATCHDOG
```

3.2 RAM Clearing Routine

This subroutine sets all of the RAM to zero and is called after the initialization of the system clock generator. The size of the RAM depends on the type of the MSP430. For the following example the RAM starts at address 200h and the size is assumed to be 100h.

```
; DEFINITIONS FOR THE RAM BLOCK (DEPENDS ON MSP430 TYPE)
RAMSTRT  .EQU     0200H           ; START OF RAM
RAMEND   .EQU     02FFH           ; LAST RAM ADDRESS

; SUBROUTINE FOR THE CLEARING OF THE RAM BLOCK
RAMCLR   CLR      R4              ; PREPARE INDEX REGISTER
RCL      CLR      RAMSTRT(R4)     ; 1ST RAM ADDRESS
         INCD     R4              ; NEXT ADDRESS
         CMP      #RAMEND-RAMSTRT+1,R4        ; RAM CLEARED?
         JLO      RCL             ; NO, ONCE MORE
         RET
```

3.3 RAM Self-Test

This routine performs a simple alternating 0/1 test on the RAM. The RAM is tested by writing a AAh,55h pattern to the entire RAM and checking the RAM for this patten. The inverted pattern is then written to RAM and rechecked. Finally, the entire RAM is cleared. If an error is found, the negative bit is set.

```
RAMSTRT  .EQU     0200H           ; START OF RAM
RAMEND   .EQU     02FFH           ; LAST RAM ADDRESS

;SUBROUTINE TO CHECK ENTIRE RAM
;USE REGISTER: R4,R5
RAMCHECK
         MOV      #55AAH,R4       ;FIRST TESTPATTERN
FILLS    CLR      R5              ;POINTER TO RAM
FILLR    MOV      R4,RAMSTRT(R5);FILL RAM WITH R4
         INCD     R5              ;NEW RAM POINTER
         CMP      #RAMEND-RAMSTRT+1,R5        ;IS RAM FILLED ?
         JLO      FILLR
         CLR      R5              ;NEW RAM POINTER
COMPAR   CMP      R4,RAMSTRT(R5);COMPARE RAM WITH R4
         JNE      ERROR           ;EXIT IF VALUES DON'T MATCH
         DECD     R5              ;NEXT RAM WORD
         CMP      #RAMEND-RAMSTRT+1,R5        ;ALL OF RAM TESTED
?
         JLO      COMPAR
         SWPB     R4              ;NEW TESTPATTERN
         TST      R4
         JN       FILLS           ;=AA55H, NEW TEST
         JZ       EXIT            ;=0000H, FINISCHED
         CLR      R4              ;TESTPATTERN = 0000
```

```
        JMP     FILLS
ERROR   SETN
EXIT    RET
```

3.4 ROM Checksum

This routine checks the integrity of the ROM by performing a checksum on the entire ROM. All ROM words from ROMSTRT+2 to ROMEND are added together in a 16-bit word. This sum is checked against the value at the beginning of the ROM (ROMSTRT). If these values do not match, then an error has occurred and the negative bit is set.

```
STACK   .SET    02E0H           ;START OF SYSTEM STACK

ROMSTRT .SECT   "PROG",0F000H
        .WORD   CHECKSUM        ;PUT CORRECT CHECKSUM INTO ROM
START   MOV     #STACK,SP       ;INITIALIZE SYSTEM STACK
                                ;POINTER
        .....                   ;OTHER INITIALIZATION PROGRAM
                                ;HERE


;SUBROUTINE TO CHECK THE INTEGRITY OF THE ROM
;USE REGISTER: R4,R5
;OUTPUT: ROM OK.          N=0
;       ROM CHECK FAILED: N=1

ROMCHECK
        CLR     R5
        MOV     #ROMEND,R4
        SUB     #ROMSTRT,R4     ;R4 CONTAINS THE LENGTH OF ROM
ROML    ADD     ROMSTRT(R4),R5  ;MAKE CHECKSUM
        DECD    R4
        JNZ     ROML
        CMP     R5,&ROMSTRT     ;IF MATCH, N-BIT IS CLEARED
        JEQ     ROMEXIT
        SETN
ROMEXIT RET


;INTERRUPT VECTOR ADDRESSES:

        .SECT   "RSTVECT",0FFFEH; PUC/RESET ADDRESS
ROMEND  .WORD   START
```

3.5 Battery Check

Due to the ratiometric measurement principle of the ADC, the measured digital value is an indication of the supply voltage of the MSP430. The measured value is inversly proportional to the supply voltage Vcc. To get the reference for later battery tests a measurement is made with Vcc = Vccmin. The result is stored in the RAM. If the battery should be tested, another measurement has to be made, and the result compared to the stored value measured with Vcc=Vccmin determines the status of the battery. If the measured value exceeds the stored one, then Vcc<Vccmin and a Battery low indication can be given by software.

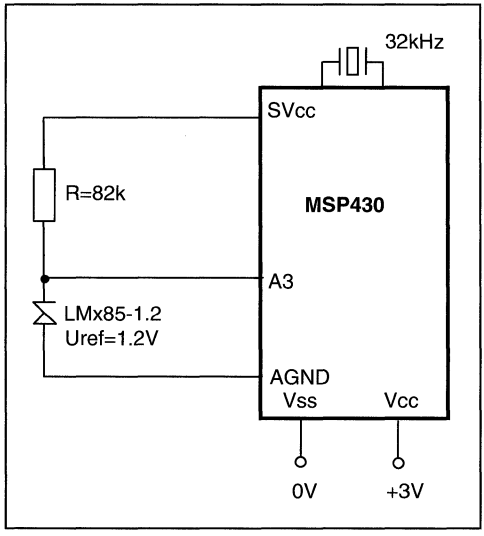


Figure 3.0: Battery Check

If no reference measurement has to be done, the value for the comparison can be determined by calculation.

According to the data sheet of the LMx85-1.2 the typical reference voltage is 1.235 Volt with a maximal deviation of ±0.012 Volt. Using the Auto-Mode of the A/D-Converter, the digital value is

$$N = INT\left|\frac{V_{IN} \cdot 2^{14}}{SVcc}\right|$$

The reference voltage can be calculated as follows:

$$SV_{CC}=SV_{CCmin}=2.8\ Volt$$

$$VIN = 1.235 \pm 0{,}012\ Volt$$

$$N_{REF} = INT\left|\frac{(1.235 \pm 0{,}012\ Volt)\cdot 2^{14}}{2.8\ Volt}\right| = 7226 \pm 70$$

To ensure that the voltage of the battery is above SVccmin, the reference value should be set to:

$$N_{REF}=7156$$

Every measured value above 7156 indicates that the battery voltage is lower than the calculated value, and a battery low signal should be sent.

The software for making a reference measurement and a resulting comparison with a new measured value is shown below.

```
ASOC      .SET      1             ;BIT POSITION FOR CONVERSION
START
                                  ;IN BTCTL
ADAUTO    .SET      800H          ;BIT POSITION TO SELECT AUTO MODE
ADNOI     .SET      100H          ;BIT POSITION TO SELECT NO
CURRENT
                                  ;SOURCE
ADA3      .SET      0CH           ;BIT POSITION TO SELECT INPUT TO
                                  ;A3
ADVREF    .SET      2H            ;SVCC=VCC


;FIRST THE VCCMIN VALUE HAS TO BE MEASURED
;AND IS STORED IN THE RAM VARIABLE BATREF
          CALL      #MEAS_A3      ;MEASURE VCCMIN
          MOV       R10,&BATREF   ;AND STORE VALUE IN RAM


;MAIN PROGRAM:
.......

;NOW THE BATTERY SHOULD BE CKECKED. IF THE BATTERY IS LOW, THE
;PROGRAM JUMPS TO THE LABEL BATLOW
          CALL      #MEAS_A3      ;MEASURE INPUT A3
          CMP       &BATREF,R10   ;IS VBATT <= VMIN ?
          JLO       BATOK
BATLOW    .....                   ;BATTERY IS LOW !
BATOK     .....                   ;BATTERY IS OK, NORMAL OPERATION
```

```
;*****************************************************************
*
;SUBROUTINE TO MEASURE CHANNEL A3 WITH THE POLLING METHOD FOR
ONE
;TIME. THE RESULT WILL BE CONTAINED IN R10
;OUTPUT: ADC VALUE OF A3 IN R10
;*****************************************************************
*

MEAS_A3  BIC.B    #ADIE,&IE2      ;DISABLE ADC INTERRUPT
         MOV      #ADVREF+ADA3+ADNOI+ADAUTO+ASOC,&ACTL
                                  ;SVCC=VCC
                                  ;INPUT=A3
                                  ;NO CURRENT SOURCE
                                  ;RANGE=AUTO
MEAS_1   BIT.B    #ADIFG,&IFG2    ;WAIT FOR EOC-SHOULD BE IFG2
(IE2)
         JZ       MEAS_1
         BIC.B    #ADIFG,&IFG2    ;CLEAR EOC FLAG
         MOV      &ADAT,R10
         BIS.B    #ADIE,&IE2      ;ENABLE ADC INTERRUPT
         RET
```

3.6 Interrupt Management

Using Interrupts is a very good method for achieving fast response with several events: for example, a transition at the I/O port initiating a communication (Start Bit). Another reason for using interrupts instead of the polling method is that the time during the occurrence of interrupts can be used for further calculations: e.g. during an A/D-conversion, a multiplication can be performed. By entering an interrupt service routine, the GIE bit will be set and therefore no other interrupt request can be handled. After leaving the interrupt service routine by executing the RETI instruction, the status word including the GIE-bit will be restored and every occurring interrupt request can now be handled. If an interrupt request should be handled while executing another interrupt service routine, the GIE-bit has to be set explicitly by software in the dedicated interrupt service routine. The handling of the interrupts is easy, as shown in the following example.

```
START    .SECT    "PROG",0F000H
         CLR.B    &IE1              ;CLEAR ALL INTERRUPT ENABLE
                                    ;FLAGS
         CLR.B    &IE2
         CLR.B    &IFG              ;CLEAR ALL INTERRUPT FLAG
                                    ;REGISTER
         CLR.B    &IFG2
         BIS.B    #P0_0IE+..,&IE1   ;ENABLE USED INTERRUPTS
         BIS.B    #ADIE+BTIE+..,&IE2
         EINT                       ;ENABLE INTERRUPTS
```

```
          .....

;INTERRUPT SERVICE ROUTINES
P0_0ISR   EINT      ;SET GIE-BIT TO ALLOW INTERRUPT NESTING
          .....
          RETI
ADCISR
          .....
          RETI
BTISR
          .....
          RETI
;INERRUPT VECTORS
          .SECT     "INT_VECT",0FFE0H
          .WORD     P0_27ISR        ;PORT0, BIT 2 TO BIT 7
          .WORD     BTISR           ;BASIC TIMER
          .WORD     START           ;NO SOURCE
          .WORD     START           ;NO SOURCE
          .WORD     START           ;NO SOURCE
          .WORD     ADCISR          ;EOC FORM ADC
          .WORD     START           ;NO SOURCE
          .WORD     START           ;NO SOURCE
          .WORD     START           ;NO SOURCE
          .WORD     START           ;NO SOURCE
          .WORD     WDTISR          ;WATCHDOG/TIMER, TIMER MODE
          .WORD     START           ;NO SOURCE
          .WORD     UARTISR         ;ADDRESS OF UART HANDLER
          .WORD     P0_0ISR         ;PORT0 BIT 0
          .WORD     START           ;NMI, OSCILLATOR FAULT
          .WORD     START           ;POWER UP RESET, WATCHDOG
```

Topics

Figures

4 Integer Calculation Subroutines

Integer routines have important advantages compared to all other calculation subroutines:

1. Speed:
   The highest speed is possible, especially if no loops are used.
2. ROM space:
   The minimum of ROM space is needed for these subroutines.
3. Adaptability:
   With the following definitions it is very easy to adapt the subroutines to the actual needs. The necessary calculation registers can be located in the RAM or in registers.

The following definitions are valid for all of the following Integer Subroutines

```
; INTEGER SUBROUTINES DEFINITIONS

IRBT     .EQU     R9              ; BIT TEST REGISTER MPY
IROP1    .EQU     R4              ; FIRST OPERAND
IROP2L   .EQU     R5              ; SECOND OPERAND LOW WORD
IROP2M   .EQU     R6              ; SECOND OPERAND HIGH WORD
IRACL    .EQU     R7              ; RESULT LOW WORD
IRACM    .EQU     R8              ; RESULT HIGH WORD
```

4.1 Unsigned Multiplication 16 x 16 bits

The following subroutine performs an unsigned 16 x 16-bit multiplication (label MPYU) or "Multiplication and Accumulation" (label MACU). The multiplication subroutine clears the result registers IRACL and IRACM before the start; the MACU subroutine adds the result of the multiplication to the contents of the result registers.

The multiplication loop starting at label MACU is the same one as the one used for the signed multiplication. This allows the usage of this subroutine for signed and unsigned multiplication, if both are needed. The used registers are shown below:

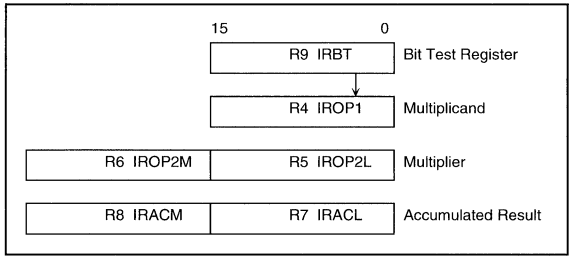


Figure 4.1: 16 x 16 Bit Multiplication : Register Use

```
; EXECUTION TIMES FOR REGISTERS USED (CYCLES @ 1MHZ):
; TASK            MACU  MPYU     EXAMPLE
;-----------------------------------------------------------------
; MINIMUM         132   134      00000H X 00000H = 000000000H
; MEDIUM          148   150      0A5A5H X 05A5AH = 03A763E02H
; MAXIMUM         164   166      0FFFFH X 0FFFFH = 0FFFE0001H

; UNSIGNED MULTIPLY SUBROUTINE: IROP1 X IROP2L -> IRACM/IRACL
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRACM, IRBT

MPYU     CLR      IRACL          ; 0 -> LSBS RESULT
         CLR      IRACM          ; 0 -> MSBS RESULT

; UNSIGNED MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 X IROP2L) + IRACM|IRACL -> IRACM|IRACL

MACU     CLR      IROP2M         ; MSBS MULTIPLIER
         MOV      #1,IRBT        ; BIT TEST REGISTER
L$002    BIT      IRBT,IROP1     ; TEST ACTUAL BIT
         JZ       L$01           ; IF 0: DO NOTHING
         ADD      IROP2L,IRACL   ; IF 1: ADD MULTIPLIER TO RESULT
         ADDC     IROP2M,IRACM
L$01     RLA      IROP2L         ; MULTIPLIER X 2
         RLC      IROP2M         ;

         RLA      IRBT           ; NEXT BIT TO TEST
         JNC      L$002          ; IF BIT IN CARRY: FINISHED
         RET
```

4.2 Signed Multiplication 16 x 16 bits

The following subroutine performs a signed 16 x 16-bit multiplication (label MPYS) or "Multiplication and Accumulation" (label MACS). The multiplication subroutine clears the result registers IRACL and IRACM before the start, and the MACS subroutine adds the result of the multiplication to the contents of the result registers. The register use is the same as with the unsigned multiplication; Figure 4.1 is therefore also valid.

```
; EXECUTION TIMES FOR REGISTERS USED (CYCLES @ 1MHZ):

; TASK            MACS  MPYS     EXAMPLE
;-----------------------------------------------------------------
; MINIMUM         138   140      00000H X 00000H = 000000000H
; MEDIUM          155   157      0A5A5H X 05A5AH = 0E01C3E02H
; MAXIMUM         172   174      0FFFFH X 0FFFFH = 000000001H

; SIGNED MULTIPLY SUBROUTINE: IROP1 X IROP2L -> IRACM|IRACL
```

```
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRACM, IRBT

MPYS    CLR     IRACL           ; 0 -> LSBS RESULT
        CLR     IRACM           ; 0 -> MSBS RESULT

;SIGNED MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 X IROP2L) + IRACM|IRACL -> IRACM|IRACL

MACS    TST     IROP1           ; MULTIPLICAND NEGATIVE ?
        JGE     L$001
        SUB     IROP2L,IRACM    ; YES, CORRECT RESULT REGISTER
L$001   TST     IROP2L          ; MULTIPLIER NEGATIVE ?
        JGE     MACU
        SUB     IROP1,IRACM     ; YES, CORRECT RESULT REGISTER

; THE REMAINING PART IS EQUAL TO THE UNSIGNED MULTIPLICATION

MACU    CLR     IROP2M          ; MSBS MULTIPLIER
        MOV     #1,IRBT         ; BIT TEST REGISTER
L$002   BIT     IRBT,IROP1      ; TEST ACTUAL BIT
        JZ      L$01            ; IF 0: DO NOTHING
        ADD     IROP2L,IRACL    ; IF 1: ADD MULTIPLIER TO RESULT
        ADDC    IROP2M,IRACM
L$01    RLA     IROP2L          ; MULTIPLIER X 2
        RLC     IROP2M          ;

        RLA     IRBT            ; NEXT BIT TO TEST
        JNC     L$002           ; IF BIT IN CARRY: FINISHED
        RET
```

4.3 Unsigned Multiplication 8 x 8 bits

The following subroutine performs an unsigned 8 x 8-bit multiplication (label MPYU8) or "Multiplication and Accumulation" (label MACU8). The multiplication subroutine clears the result register IRACL before the start, the MACU subroutine adds the result of the multiplication to the contents of the result register. The upper bytes of IROP1 and IROP2L must be zero when the subroutine is called. The register use is shown below:

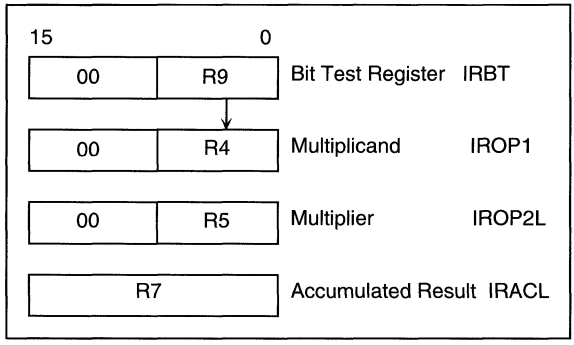


Figure 4.2: 8 x 8 Bit Multiplication : Register Use

```
; EXECUTION TIMES FOR REGISTERS USED (CYCLES @ 1MHZ):

; TASK            MACU8  MPYU8   EXAMPLE
;-------------------------------------------------------------
; MINIMUM         58     59      000H X 000H = 00000H
; MEDIUM          62     63      0A5H X 05AH = 03A02H
; MAXIMUM         66     67      0FFH X 0FFH = 0FE01H

; UNSIGNED BYTE MULTIPLY SUBROUTINE: IROP1 X IROP2L -> IRACL
;
; USED REGISTERS IROP1, IROP2L, IRACL, IRBT
;
MPYU8    CLR      IRACL          ; 0 -> RESULT
;
; UNSIGNED BYTE MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 X IROP2L) +IRACL -> IRACL
;
MACU8    MOV      #1,IRBT        ; BIT TEST REGISTER
L$002    BIT      IRBT,IROP1     ; TEST ACTUAL BIT
         JZ       L$01           ; IF 0: DO NOTHING
         ADD      IROP2L,IRACL   ; IF 1: ADD MULTIPLIER TO RESULT
L$01     RLA      IROP2L         ; MULTIPLIER X 2
         RLA.B    IRBT           ; NEXT BIT TO TEST
         JNC      L$002          ; IF BIT IN CARRY: FINISHED
         RET
```

4.4 Signed Multiplication 8 x 8 bits

The following subroutine performs a signed 8 x 8-bit multiplication (label MPYS8) or "Multiplication and Accumulation" (label MACS8). The multiplication subroutine clears the result register IRACL before the start, and the MACS8 subroutine adds the result of the multiplication to the contents of the result register. The register usage is the same as with the unsigned 8 x 8 multiplication; Figure 4.2 is therefore also valid.

The part starting with label MACU8 is the same as used with the unsigned multiplication.

```
; EXECUTION TIMES FOR REGISTER USED (CYCLES @ 1MHZ):

; TASK            MACS8  MPYS8   EXAMPLE
;-------------------------------------------------------------
; MINIMUM         64     65      000H X 000H = 00000H
; MEDIUM          75     76      0A5H X 05AH = 0E002H
; MAXIMUM         86     87      0FFH X 0FFH = 00001H

; SIGNED BYTE MULTIPLY SUBROUTINE: IROP1 X IROP2L -> IRACL

; USED REGISTERS IROP1, IROP2L, IRACL, IRBT

MPYS8    CLR      IRACL          ; 0 -> RESULT

; SIGNED BYTE MULTIPLY AND ACCUMULATE SUBROUTINE:
; (IROP1 X IROP2L) +IRACL -> IRACL

MACS8    TST.B    IROP1          ; MULTIPLICAND NEGATIVE ?
         JGE      L$101          ; NO
         SWPB     IROP2L         ; YES, CORRECT RESULT
         SUB      IROP2L,IRACL
         SWPB     IROP2L         ; RESTORE MULTIPLICATOR

L$101    TST.B    IROP2L         ; MULTIPLICATOR NEGATIVE ?
         JGE      MACU8
         SWPB     IROP1          ; YES, CORRECT RESULT
         SUB      IROP1,IRACL
         SWPB     IROP1

; THE REMAINING PART IS THE UNSIGNED MULTIPLICATION

MACU8    MOV      #1,IRBT        ; BIT TEST REGISTER
L$002    BIT      IRBT,IROP1     ; TEST ACTUAL BIT
         JZ       L$01           ; IF 0: DO NOTHING
         ADD      IROP2L,IRACL   ; IF 1: ADD MULTIPLIER TO RESULT
L$01     RLA      IROP2L         ; MULTIPLIER X 2
         RLA.B    IRBT           ; NEXT BIT TO TEST
         JNC      L$002          ; IF BIT IN CARRY: FINISHED
         RET
```

4.5 Unsigned Division 32/16 bits

The subroutine performs an unsigned 32-bit by 16-bit division. If the result does not fit into 16-bit, then the carry is set after return. If a valid result is obtained, then the carry is reset after return. The register usage is shown in the next figure:

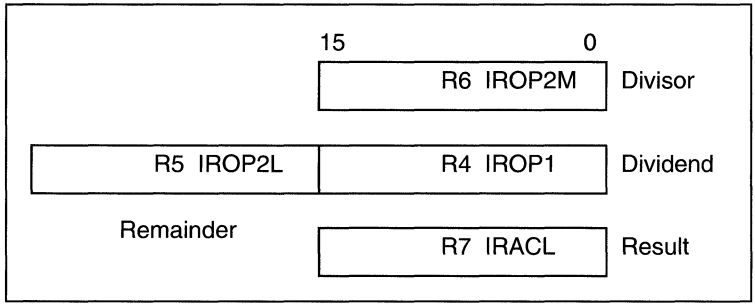


Figure 4.3: Unsigned Division: Register use

```
; DIVISION SUBROUTINE 32-BIT BY 16-BIT
; IROP2L/IROP1 : IROP2M -> IRACL   REMAINDER IN IROP2L
; RETURN: CARRY = 0: OK    CARRY = 1: QUOTIENT > 16 BITS

DIVIDE   CLR      IRACL           ; CLEAR RESULT
         MOV      #17,IRACM       ; INITIALIZE CYCLE COUNTER
DIV1     CMP      IROP2M,IROP2L   ; DIVIDEND > DIVISOR ?
         JLO      DIV2            ; NO
         SUB      IROP2M,IROP2L   ;YES, DIVIDEND = DIVIDEND -
                                  ;DIVISOR
DIV2     RLC      IRACL           ;C = 0, IF DIVIDEND < DIVISOR
         JC       DIV4            ;IF CARRY, END
         DEC      IRACM           ;OPERATION AT THE END ?
         JZ       DIV3            ;YES,
         RLA      IROP1           ;DOUBBLE DIVIDEND
         RLC      IROP2L
         JNC      DIV1            ;NO CARRY BY DOUBBLING THE
                                  ;DIVIDEND ?
         SUB      IROP2M,IROP2L   ;IF CARRY, DIVIDEND = DIVIDEND-
                                  ;DIVISOR
         SETC                     ;CARRY = 1 IF DIVIDEND > DIVISOR
         JMP      DIV2
DIV3     CLRC                     ;CLEAR CARRY
DIV4     RET
```

4.6 Shift Routines

The results of the above subroutines (MPY, DIV) accumulated in IRACM/IRACL have to be adapted to different numbers of bits after the decimal point, or because they are getting too large to fit into 32 bits. The following subroutines can do these jobs. If other numbers of shifting are necessary they may be constructed as shown for the 6-bit shifts. No tests are made for overflow!

```
; SIGNED SHIFT RIGHT SUBROUTINE FOR IRACM/IRACL
; DEFINITIONS SEE ABOVE

SHFTRS6  CALL     #SHFTRS3        ; SHIFT 6 BITS RIGHT
SHFTRS3  RRA      IRACM           ; SHIFT MSBS, BIT0 -> CARRY
         RRC      IRACL           ; SHIFT LSBS, CARRY -> BIT15
SHFTRS2  RRA      IRACM
         RRC      IRACL
SHFTRS1  RRA      IRACM
         RRC      IRACL
         RET

; UNSIGNED SHIFT RIGHT SUBROUTINE FOR IRACM/IRACL

SHFTRU6  CALL     #SHFTRU3        ; SHIFT 6 BITS RIGHT
SHFTRU3  CLRC                     ; CLEAR CARRY
         RRC      IRACM           ; SHIFT MSBS, BIT0 -> CARRY,
                                  ; 0 -> BIT15
         RRC      IRACL           ; SHIFT LSBS, CARRY -> BIT15
SHFTRU2  CLRC
         RRC      IRACM
         RRC      IRACL
SHFTRU1  CLRC
         RRC      IRACM
         RRC      IRACL
         RET

; SIGNED/UNSIGNED SHIFT LEFT SUBROUTINE FOR IRACM/IRACL

SHFTL6   CALL     #SHFTL3         ; SHIFT 6 BITS LEFT
SHFTL3   RLA      IRACL           ; SHIFT LSBS, BIT0 -> CARRY
         RLC      IRACM           ; SHIFT MSBS, CARRY -> BIT15
SHFTL2   RLA      IRACL
         RLC      IRACM
SHFTL1   RLA      IRACL
         RLC      IRACM
         RET
```

4.7 Rules for the Integer Subroutines

Despite the fact that the subroutines shown above can only handle integer numbers, it is possible to use numbers with fractional parts. It is only necessary to define for each number where the "virtual" decimal point is located. Relatively simple rules define where the decimal point is located for the result.

For calculations with the integer subroutines it is almost impossible to remember where the virtual decimal point is located. It is therefore a good programming practice to indicate, in the comment part of the software listing, where the decimal point is currently located . The indication can have the following form:

N.M

with: N Worst case bit count of integer part (allows additional assessments)
M Number of bits after the virtual decimal point

The rules for determining the location of the decimal point are easy:

1. Addition and subtraction: Positions after the decimal point have to be equal. The position is the same for the result.
2. Multiplication: Positions after the decimal point may be different. The two positions are added for the result.
3. Division: Positions after the decimal point may be different. The two positions are subtracted for the result. (Dividend - divisor)

EXAMPLES:

First Operand	Operation	Second Operand	Result
NNN.MMM	+	NNNN.MMM	NNNN.MMM
NNN.M	x	NN.MMM	NNNNN.MMMM
NNN.MM	-	NN.MM	NNN.MM
NNNN.MMMM	:	NN.MMM	NN.M
NNN.M	+	NNNN.M	NNNN.M
NNN.MM	x	NN.MMM	NNNNN.MMMMM
NNN.M	-	NN.M	NNN.M
NNNN.MMMMM	:	NN.M	NN.MMMM

If two numbers have to be divided and the result should have n digits after the decimal point, the dividend has to be loaded with the number appropriately shifted to the left, and zeroes filled into the lower bits. The same procedure may be used if a smaller number is to be divided by a larger one.

EXAMPLES for the division:

First Operand (shifted)	Operation	Second Operand	Result
NNNN.000	:	NN	NN.MMM
NNNN.000	:	NN.M	NN.MM
NNNN.000	:	N.MM	NNN.M
0.MMM000	:	NN.M	0.MMMMM

EXAMPLE for a source using the number indication:

```
MOV     #01234H,IROP2L ; CONSTANT 12.34 LOADED   8.8H
MOV     R15,IROP1      ; OPERAND FETCHED         2.3H
CALL    #MPYS          ; SIGNED MPY             10.11
CALL    #SHFTRS3       ; DIVIDE BY 2^3          10.8
ADD     #00678H,IRACL  ; ADD CONSTANT 6.78      10.8
ADC     IRACM          ; ADD CARRY              10.8
```

TOPICS

Figures

5 General Purpose Subroutines

5.1 Saving Power Consumption

The following software routine generates a square-wave at the port pin P0.0. The low to high ratio is 1:1. The time for the high and the low period is determined by the 8bit Timer Preload Register, which is set to f0h. This means that every 512 MCLK cycles (= 16 ACLK cycles) an interrupt occurs. In the time between these interrupts the processor is switched to Low Power Mode LPM3, to save power consumption. A wake up is initiated by the 8bit Timer. In the corresponding interrupt service routine the level of the port pin is determined.

```
;*************************************************************
;EVERY 512 MCLK CYCLES A TC8-INTERRUPT OCCURS AND WAKES UP
;THE MSP430 FROM LOW POWER MODE 3
;*************************************************************

          MOV.B     #P0_1IFG,&IE1    ;ENABLE TC8 INTERRUPT
          CLR.B     &IE2             ;AND DISABLE ALL OTHER
                                     ;INTERRUPTS
          CLR.B     &P0IE            ;DISABLE I/O INTERRUPT
          BIS.B     #P0_0,&P0DIR     ;SET PORTPIN P0 TO OUTPUT
          MOV.B     #P0_0,&P0OUT     ;SET PORTPIN P0 TO LOW

          EINT                       ;SET GIE BIT IN SR
          MOV.B     #0F0H,&TCPLD     ;LOAD PRELOAD REGISTER
                                     ;(0100H-16)
          CLR.B     &TCDAT           ;LOAD COUNTER WITH PRELOAD
                                     ;REGISTER
          MOV.B     #SSEL0+ISCTL+ENCNT,&TCCTL
                                     ;SET TC8 TO ACLK CLOCK SOURCE,
                                     ;INTERRUPT FROM COUNTER AND
                                     ;ENABLE COUNTER

          BIS       #SCG0+SCG1+CPUOFF,SR ;ENTER LP-MODE 3
LOOP      JMP       LOOP             ;NEVER ENDING LOOP

;INTERRUPT SERVICE ROUTINE FOR INTERRUPT CAUSED BY TC8:
P0_1INT   XOR.B     #P0_0,&P0OUT     ;TOGGLE OUTPUT
          RETI                       ;RETURN FROM INTERRUPT

;INTERRUPT VECTOR ADDRESSES:
          .SECT     "P0_1VECT",0FFF8H     ;ADDRESS FOR TC8 INTERRUPT
          .WORD     P0_1INT
          .SECT     "RSTVECT",0FFFEH      ;PUC/RESET ADDRESS
          .WORD     START
```

5.2 Calculated Branch

The following software example shows a small menu system. Two keys control the operations: the Enter key calls the displayed subroutine, and the Next key selects the subroutine. The following subroutines are assumed:

DSP_TXT	displays the text, which is pointed to by R10
WAIT10MS	waits 10 ms
KEYIN	reads the keyboard and stores the pressed key into R6

This example should demonstrate the capability of branches controlled by pointer.

```
MENU_1    CLR      R15                ;TABLE POINTER
MENU      MOV      TXT_TAB(R15),R10   ;TEXT POINTER
          CALL     #DSP_TXT           ;DISPL. THE TEXT POINTED BY
                                      ;R10
          CALL     #WAIT10MS          ;WAITS 10 MS
MENU_3    CALL     #KEYIN             ;READS THE KEYS INTO R6
          CMP.B    #KEYE,R6           ;WAS ENTER KEY PRESSED ?
          JEQ      MENU_E             ;YES, JUMP TO PROPER SUBROUTINE
          CMP.B    #KEYN,R6           ;WAS NEXT KEY PRESSED ?
          JEQ      MENU_N             ;YES, POINTER TO NEXT MENU
                                      ;ENTRY
          JMP      MENU_3             ;NO
MENU_E    BR       SUB_TAB(R15)       ;JUMP TO SELECTED ROUTINE
MENU_N    INCD     R15                ;UPDATE TABLE POINTER
          TST      SUB_TAB(R15)       ;IS POINTER AT THE END OF
                                      ;TABLE
          JNZ      MENU               ;NO
          JMP      MENU_1             ;YES, RESET POINTER

;TABLE FOR ALL TEXT DISPLAYED ON THE LCD
TXT_TAB   .WORD    TEXT1              ;POINTER TO TEXT TEXT1
          .WORD    TEXT2              ;POINTER TO TEXT TEXT2
          .WORD    TEXT3              ;POINTER TO TEXT TEXT3
;TABLE FOR ALL SUBROUTINES, WHICH CAN BE CALLED
SUB_TAB   .WORD    SUB1               ;POINTER TO SUBROUTINE SUB1
          .WORD    SUB2               ;POINTER TO SUBROUTINE SUB1
          .WORD    SUB3               ;POINTER TO SUBROUTINE SUB1
          .WORD    0                  ;END OF TABLE

;DEFINITION OF THE TEXT, WHICH IS DISPLAYED
TEXT1     .BYTE    "SUB1",255         ;255 IS END OF TEXT
TEXT2     .BYTE    "SUB2",255
TEXT3     .BYTE    "SUB3",255
```

```
;SUBROUTINES WHICH ARE CALLED BY THE MENU
SUB1
          .....
          JMP       MENU
SUB2
          .....
          JMP       MENU
SUB3
          .....
          JMP       MENU
```

The above program example use word-tables to branch to the appropriate location. To reduce program space, the word tables can be substituted by byte-tables. The following example use byte-tables to branch to the apropriate program location.

```
MEASINIT MOV.B #0,ADCST            ; STATUS 0 IS THE INITIALIZATION
                                   ; VALUE
.....

;SUBROUTINE TO DEMONSTRATE THE BRANCHES IN REGARD OF BYTE-TABLES

ADCINT    PUSH      R6             ; WORKING REGISTER
          MOV.B     ADCST,R6       ; ADC STATUS BYTE
          MOV.B     ADCIT(R6),R6   ; REL. ADDRESS OF CURRENT
                                   ; HANDLER
          ADD       R6,PC          ; BRANCH TO HANDLER

ADCIT     .BYTE     ADCST0-ADCIT   ; STATUS0: ADC INACTIVE
          .BYTE     ADCST1-ADCIT   ; 1: INIT 1ST CHARGE
          .BYTE     ADCST2-ADCIT   ; 2: CHARGE, INIT 1ST MEASUREMENT
          .BYTE     ADCST3-ADCIT   ; 3: 1ST MEAS., INIT 2ND CHARGE
          .....

ADCST0    .....
          JMP       L$402

ADCST1    .....
          JMP       L$402

ADCST2    .....
          JMP       L$402

ADCST3    .....
          JMP       L$402
.....
```

```
L$402     INC.B     ADCST          ; ADCST + 1

          POP       R6             ; RESTORE R6
          RET
```

5.3 Binary to BCD

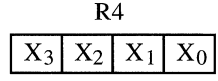
The conversion of binary to BCD and vice versa is normally a time-consuming task. For example, five divisions by ten are necessary to convert a 16-bit binary number to BCD. The DADD instruction reduces this to a loop with five instructions.

```
; THE BINARY NUMBER IN R4 IS CONVERTED TO A 5-DIGIT
; BCD NUMBER CONTAINED IN R5 AND R6

BINDEC    MOV       #16,R7         ; LOOP COUNTER
          CLR       R6             ; 0 -> RESULT MSD
          CLR       R5             ; 0 -> RESULT LSD
L$1       RLA       R4
          DADD      R5,R5          ; RESULT X2 LSD
          DADD      R6,R6          ;           MSD
          DEC       R7             ; THROUGH?
          JNZ       L$1
          RET                      ; YES, RESULT IN R5|R6
```

The above subroutine may be enlarged to any length of the binary part simply by adding registers for the storage of the BCD number.

5.4 BCD to Binary

This subroutine converts a packed 16 bit BCD word to a 16 bit binary word by multiplying the digit with its valency. To reduce code length, the horner scheme is used as follows:



$$R5=X_0+10(X_1+10(X_2+10X_3))$$

```
;THE PACKED BCD NUMBER IN R4 IS CONVERTED INTO A BINARY NUMBER
;CONTAINED IN R5
;INPUT:  R4 = BCD NUMBER
;OUTPUT: R5 = BINARY NUMBER
;EXECUTION TIME: 79 CYCLES
```

```
BCDBIN    MOV     #4,R8          ;LOOP COUNTER ( 4 DIGITS )
          CLR     R5
          CLR     R6
SHFT4     RLA     R4             ;SHIFT LEFT DIGIT INTO R6
          RLC     R6             ;THROUGH CARRY
          RLA     R4
          RLC     R6
          RLA     R4
          RLC     R6
          RLA     R4
          RLC     R6
          ADD     R6,R5          ;X_N+10X_N+1
          CLR     R6
          DEC     R8             ;THROUGH ?
          JZ      END            ;YES
MPY10     RLA     R5             ;NO, MULTIPLICATION WITH 10
          MOV     R5,R7
          RLA     R5
          RLA     R5
          ADD     R7,R5
          JMP     SHFT4          ;NEXT DIGIT
END       RET                    ;RESULT IS IN R5
```

5.5 Bubble Sort

The following routine is sorting a word-array in falling sequence by using the Bubble Sort Algorithm, which is the most efficient algorithm if the array is less than 20 elements. If up to 100 elements are contained in the array, the execution time of this algorithm is acceptable with regard to the code-length. The number of loops which are necessary to check the entire word array is as follows:

$$N = \frac{n \cdot (n-1)}{2}$$

N = number of loops
n = number of elements to be sorted

The absolute execution time depends on the number of changes to be done.

Example:

Number of words to be sorted:	20
Number of cycles, if words are sorted :	7412
Number of cycles, if words are sorted inversely:	9122

The software to implement the Bubble Sort algorithm is as follows.

```
VARSTRT   .SECT    "VAR",0200H
TABST     .WORD    10,20,30,4,5,6,7   ;TABLE OF WORDS TO BE SORTED
TABEND    .WORD    8                  ;END OF TABLE
.....                                 ;INITIALISATION, ASO.
```

```
;SUBROUTINE TO SORT A LIST OF WORDS
;THE LIST TO SORT BEGINS AT ADDRESS TABST AND ENDS WITH ADDRESS
;TABEND.
;USE REGISTER: R5,R6,R7

BUBBLE    MOV       #TABEND-TABST,R6    ;LENGTH OF LIST
L$20      MOV       #TABST,R5           ;START OF LIST
L$30      MOV       @R5+,R7             ;FETCH 1ST ITEM
          CMP       @R5,R7              ;COMPARE TWO ITEMS
          JHS       L$12                ;RIGHT ORDER
          MOV       @R5,-2(R5)          ;WRONG ORDER:
          MOV       R7,0(R5)            ;EXCHANGE ITEMS
L$12      CMP       #TABEND,R5          ;ALL THROUGH ?
          JNE       L$30                ;NO
          DEC       R6                  ;N-TIMES MADE ?
          JNE       L$20                ;NO
          RET
```

5.6 Table Search

Table searches are efficiently performed by using the indexed mode (X(Rn)) to address the tables. In the following example, a table of 31 bytes is searched for a match with a 5-byte string. The used index mode has the capability to search a 65535-byte string in an 65535-byte table, if needed. If the search-string is found, the address of the first character will be TABST(R6).

```
EOS       .EQU      0FFH              ;END OF STRING

VARSTRT   .SECT     "VAR",0200H
TABST     .BYTE     "MUEHLHOFERANTONTEXASINSTRUMENTS"
TABEND    .BYTE     EOS

ROMSTRT   .SECT     "PROG",0F000H
.....                               ;INITIALIZATION AND OTHER SOFTWARE

SEARCH
STRLEN    MOV       #0FFFFH,R4        ;DETERMINES THE LENGTH OF A STRING
STR_1     INC       R4                ;RESULT WILL BE CONTAINED IN R4
          CMP.B     #EOS,STRING(R4)
          JNE       STR_1             ;R4 IS LENGHT OF STRING

          DEC       R4                ;POINTER TO END OF STRING
          MOV       #TABEND-TABST,R6      ;LENGHT OF TABLE
L1        MOV       R4,R7             ;RESET POINTER
L2        DEC       R6                ;NEXT CHARACTER TO COMPARE
          JNC       NOFIND            ;THE SEARCH STRING WAS NOT FOUND
          CMP.B     STRING(R7),TABST(R6)
```

```
        JNE     L1              ;COMPARE NEXT
        DEC     R7              ;ONE CHARACTER WAS FOUND
        JC      L2              ;IS NEXT CHARACTER THE SAME ?

MATCH   .....                   ;TABSTRT(R6) IS THE BEGINNING
                                ;OF THE FOUND STRING IN THE TABLE
NOFIND  .....                   ;THE TEXT WAS NOT FOUND

STRING  .BYTE   "TEXAS",EOS     ;STRING TO FIND
```

5.7 Parity

This routine provides a quick way of determining the parity of the number of 1's in a byte. By exclusive OR'ing all the bits of the byte together, a single bit will be derived which is the even parity of the word. When exclusive OR'ing, an even number of 1's will combine to form a 0, leaving either an odd 1 or 0 bit. This routine keeps splitting the byte in half, and exclusive OR'ing the two halves. The algorithm is shown below:

```
STEP 1
            7654 3210     R4
       XOR       7654     R5
            ---------
            XXXX ABCD     R4
STEP 2
                 AB CD    R4
             XOR    AB    R5
             ---------
            XXXX XX AB    R4
STEP3
                   A B    R4
              XOR    A    R5
                   ---
            XXXX XX X P   (RESULT)  R4-> CARRY
```

```
;*****************************************************************
;SUBROUTINE TO FIND EVEN PARITY IN R4
;CARRY = 0 = EVEN NUMBER OF 1S
;CARRY = 1 = ODD NUMBER OF 1S
;R4 IS CLEARED AFTERWARDS
;USE REGISTER : R5
;EXECUTION TIME : 17 CYCLES INCLUDING RET
;CODE LENGTH    : 30 BYTES
;*****************************************************************
```

```
PARITY    MOV       R4,R5          ;DUBLICATE TARGET BYTE
          RRA       R5             ;LINE UP THE MS NIBBLE WITH
                                   ;THE LS NIBBLE
          RRA       R5
          RRA       R5
          RRA       R5
          XOR       R5,R4          ;EXCLUSIVE OR THE NIBBLES TO GET A
                                   ;NIBBLE ANSWER
          MOV       R4,R5          ;DUBLICATE THE NIBBLE ANSWER
          RRA       R5             ;LINE UP BITS 0, 1 OF THE ANSWER
          RRA       R5             ;2, 3 OF THE ANSWER
          XOR       R5,R4          ;GET A NEW 2-BIT ANSWER
          MOV       R4,R5          ;DUBLICATE THIS 2-BIT ANSWER
          RRA       R5             ;LINE UP BIT 0 WITH BIT 1
          XOR       R5,R4          ;GET FINAL EVEN PARITY ANSWER
          RRA       R4             ;ROTATE ANSWER INTO THE CARRY BIT
          RET
```

The next possibility to find the parity of a byte is less program memory consuming, but needs more execution time. All bits of the byte whose parity has to be determined are shifted into the carry and added to as a sum. If the lowest significant bit is zero, the number of 1's is even; if it is one, the number of 1's is odd. This bit is shifted into the carry to identify the parity.

```
;*************************************************************
;SUBROUTINE TO FIND EVEN PARITY IN R4
;CARRY = 0 = EVEN NUMBER OF 1S
;CARRY = 1 = ODD NUMBER OF 1S
;R4 IS CLEARED AFTERWARDS
;USE REGISTER : R5,R6
;EXECUTION TIME : 46 CYCLES INCLUDING RET
;CODE LENGTH    : 16 BYTES
;*************************************************************

PARITY2   MOV       #8,R5
          CLR       R6
L1        RRA       R4
          ADC       R6
          DEC       R5
          JNZ       L1
          RRA       R6
          RET
```

5.8 Realtime Clock with 8 bit Timer

To programme a realtime clock, in this example the 8bit Timer is used to get the appropriate timing. The ACLK frequency (32.768 kHz) is divided by 256 by the 8 bit Timer, and an interrupt is generated. This interrupt occurs every 1/128 second. The corresponding interrupt handler has to accumulate these interrupts to be able to calculate the time of it. The current time is stored in the 16 bit registers RTCLSW and RTCMSW as follows:

These registers are defined as R4 and R5, but they can also be located in the RAM. The routine to display the time is assumed to exist and is named DSP_CLK.

This routine can be extended to a complete calendar. For this purpose, the days can be accumulated to weeks, and further to months and years.

The accuracy of the calendar and the clock depends only on the accuracy of the crystal frequency.

```
;*************************************************************
;             8BIT TIMER/COUNTER AS REALTIME CLOCK
;*************************************************************

RTCLSW    .EQU      R4                  ;LO WORD OF REALTIME CLOCK
RTCMSW    .EQU      R5                  ;HI WORD OF REALTIME CLOCK
CALEN     .EQU  0   ;0 = NO CALENDAR IS IMPLEMENTED
                                        ;1 = CALENDAR IS IMPLEMENTED
                                        ;INITIALIZATION:

          CLR       RTCLSW              ;CLEAR REALTIME CLOCK
          CLR       RTCMSW
          MOV.B     #P0_1IFG,&IE1       ;ENABLE TC8 INTERRUPT
          CLR.B     &IE2                ;AND DISABLE ALL OTHER
                                        ;INTERRUPTS
          CLR.B     &TCPLD              ;CLEAR PRELOAD REGISTER
          CLR.B     &TCDAT              ;LOAD COUNTER WITH PRELOAD
                                        ;REGISTER
          MOV.B     #SSEL1+ISCTL+ENCNT,&TCCTL
                                        ;SET TC8 TO ACLK CLOCK
                                        ;SOURCE,
                                        ;INTERRUPT FROM COUNTER AND
```

```
                                    ; ENABLE COUNTER
          EINT                      ;SET GIE BIT IN SR
;INTERRUPT SERVICE ROUTINE FOR INTERRUPT CAUSED BY TC8:
TC8_INT
          INC       RTCLSW          ;INC 1/128 SEC COUNTER
          BIT       #80H,RTCLSB     ;1 SEC OVER?
          JZ        TC8_END         ;NO
          CLRC                      ;YES
          DADD      #20H,RTCLSW     ;ADJUST LSW WITHOUT CARRY
          CMP       #6000H,RTCLSW   ;60 SEC OVER?
          JLO       TC8_1           ;NO, DISPLAY NEW TIME
          CLRC                      ;YES
          DADD      #4000H,RTCLSW   ;ADJUST MSW WITHOUT CARRY
          DADC      RTCMSW          ;AND ADJUST MSW
          CMP.B     #0060H,RTCMSW   ;1 HOUR OVER ?
          JNE       TC8_1           ;NO
          CLRC                      ;YES
          DADD      #40H,RTCMSW     ;ADJUST MSW WITHOUT CARRY
          CMP       #2400H,RTCMSW   ;1 DAY OVER ?
          JNE       TC8_1           ;NO
          CLR       RTCMSW          ;YES, ADJUST MSW
          .IF       CALEN
          CALL      #CALDAR         ;ONLY IF THE CALENDAR
                                    ;FUNCTION
                                    ;IS IMPLEMENTED
          .ENDIF
TC8_1     CALL      #DSP_CLK        ;DISPLAY THE TIME
TC8_END   RETI

;INTERRUPT VECTOR ADDRESSES:

          .SECT     "P0_1VECT",0FFF8H  ;ADDRESS FOR TC8 INTERRUPT
          .WORD     TC8_INT
          .SECT     "RSTVECT",0FFFEH   ;PUC/RESET ADDRESS
          .WORD     START
```

5.9 Realtime Clock with Basic Timer

The appropriate timing for a Realtime Clock can also be generated by the Basic Timer. The initialization routine has to be substituted by the following routine:

```
;INITIALIZATION:
CALEN     .EQU      0               ;0 = NO CALENDAR IS IMPLEMENTED
                                    ;1 = CALENDAR IS IMPLEMENTED

          CLR       RTCLW           ;CLEAR REALTIME CLOCK
          CLR       RTCMSW
          ;LCD-TIMING CONFIGURATION STANDS HERE
```

```
          BIS.B     #IP2+DIV,&BTCTL;TIMER INTERRUPT OCCURS EVERY
                                   ;SEC.
          BIS.B     #BTIE,&IE2     ;ENABLE BASIC TIMER INTERRUPT
          CLR.B     &IE1           ;AND DISABLE ALL OTHER
                                   ;INTERRUPTS
          EINT                     ;SET GIE BIT IN SR
```

Because of the possibility of configuring the Basic Timer in such a way that the interrupt occurs only every second (not every 1/128 sec as above), the counter for the 1/128 seconds is unnecessary and will be set to zero. For that reason the interrupt service routine becomes shorter as follows:

```
;INTERRUPT SERVICE ROUTINE FOR INTERRUPT CAUSED BY BT:
BT_INT
          CLRC
          DADD      #100H,RTCLSW       ;ADJUST LSW WITHOUT CARRY
          CMP       #6000H,RTCLSW      ;60 SEC OVER?
          JLO       BT_1               ;NO, DISPLAY NEW TIME
          CLRC                         ;YES
          DADD      #4000H,RTCLSW      ;ADJUST MSW WITHOUT CARRY
          DADC RTCMSW                  ;AND ADJUST MSW
          CMP.B     #0060H,RTCMSW      ;1 HOUR OVER ?
          JNE       BT_1               ;NO
          CLRC                         ;YES
          DADD      #40H,RTCMSW        ;ADJUST MSW WITHOUT CARRY
          CMP       #2400H,RTCMSW      ;1 DAY OVER ?
          JNE       BT_1               ;NO
          CLR       RTCMSW             ;YES, ADJUST MSW
          .IF       CALEN
          CALL      #CALDAR            ;ONLY IF THE CALENDAR
                                       ;FUNCTION
                                       ;IS IMPLEMENTED
          .ENDIF
BT_1      CALL      #DSP_CLK           ;DISPLAY THE TIME
BT_END    RETI

;INTERRUPT VECTOR ADDRESSES:
          .SECT     "P0_1VECT",0FFE2H;ADDRESS FOR BT INTERRUPT
          .WORD     BT_INT
          .SECT     "RSTVECT",0FFFEH ;PUC/RESET ADDRESS
          .WORD     START
```

5.10 Optional Calendar

This code enhances the previous realtime clock to a calendar, which will keep track of days, months, and years including leap years. To implement these functions, it is necessary to set the assembler variable CALEN =1. The result will be in the following registers in the BCD Format:

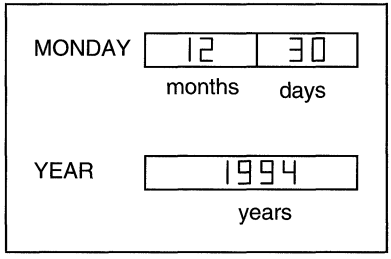


Figure 5.1: Format of the Calendar

```
;*****************************************************************
;SUBROUTINE: CALDAR
;THIS ROUTINE IS CALLED EVERY DAY AND CALCULATES THE DATE INCLUDING
;LEAP YEARS.
;INPUT: IS CALLED EVERY DAY
;OUTPUT: WORD MODAY : MSB = MONTHS, LSB = DAYS
;        WORD YEAR  : YEARS
;USED REGISTER: TEMP .EQU R8
;*****************************************************************

TEMP      .EQU     R8              ;TEMPORARY REGISTER
LEAP0     .EQU     0FFECH          ;MASKS
LEAP1     .EQU     12H
LEAP2     .EQU     0H

DAYTAB    .BYTE    31H,28H,31H,30H,31H,30H   ;MAX DAYS EACH MONTH
          .BYTE    31H,31H,30H,31H,30H,31H   ;IN BCD FORMAT

CALDAR    MOV      MODAY,TEMP      ;ONLY MONTHS TO TEMP
          SWPB     TEMP
          BIC      #0FF00H,TEMP    ;CLEAR DAYS OF TEMP
          CMP.B    #10,TEMP        ;ADJUST BCD -> BIN
          JLO      CALDAR_1
          SUB.B    #6,TEMP
CALDAR_1
          CMP.B    DAYTAB-1(TEMP),MODAY ;IS ONE MONTH OVER ?
          JLO      CALDAR_6        ;NO
          CMP      #0228H,MODAY    ;WAS IT 28.FEB. ?
```

```
          JNE       CALDAR_5        ;NO, NORMAL OPERATION

; CALCULATION OF THE LEAP YEAR
          MOV       YEAR,TEMP
          BIC       #LEAP0,TEMP
          CMP       #LEAP1,TEMP
          JEQ       CALEAP
          CMP       #LEAP2,TEMP
          JEQ       CALEAP
          JMP       CALDAR_5        ;NO LEAP YEAR, NORMAL OPERATION
CALEAP    INC       MODAY           ;TODAY IS 29.FEB
          RET

;NORMAL OPERATION WITHOUT LEAP YEAR
CALDAR_5
          CLRC
          DADD      #0100H,MODAY    ;YES, ONE MORE MONTH
          BIC       #00FFH,MODAY    ;DAYS TO ZERO
CALDAR_6
          CLRC
          DADD      #1,MODAY        ;ONE DAY IS OVER
          CMP       #1300H,MODAY    ;IS ONE YEAR OVER ?
          JLO       CALDAR_7        ;NO
          MOV       #0101H,MODAY    ;YES, ADJUST MONTHS AND DAYS
          CLRC
          DADD      #1,YEAR         ;ONE MORE YEAR
CALDAR_7
          RET
```

5.11 Square Root

The square root is often needed in computations. The following subroutine uses the NEWTONIAN approximation for this calculation. The number of iterations depends on the length of the operand. The general formula is:

$$\sqrt[m]{A} = X$$

$$X_{n+1} = \frac{1}{m}\left((m-1)\cdot X_n + \frac{A}{X_n^{m-1}}\right)$$

With the substitution of m=2 it follows:

$$\sqrt{A} = X$$

$$X_{n+1} = \frac{1}{2}\cdot\left(X_n + \frac{A}{X_n}\right)$$

$$X_0 = A/2$$

To calculate A/Xn a division is necessary, which is done in the subroutine XDIV. The result of this division has the same integer format as the divisor Xn. This makes an easy operation possible.

```
AH          .EQU      R8              ;HIGH WORD OF A
AL          .EQU      R9              ;LOW WORD OF A
XNH         .EQU      R10             ;HIGH WORD OF RESULT
XNL         .EQU      R11             ;LOW WORD OF RESULT

;THE RANGE FOR THE OPERAND EXTENDS FROM 0000.0002H TO 7FFF.FFFFH
;INPUT: OPERAND IN AH.AL
;OUTPUT: RESULT IN XNH.XNL
;EXAMPLE: SQR(2)=1.6A09H
;          SQR(7FFF.FFFFH) = B5.04F3H
;          SQR(0000.0002H) = 0.016AH
SQR         MOV       AH,XNH          ;SET X0 TO A/2 FOR THE FIRST
            MOV       AL,XNL          ;APPROXIMATION
            RRA       XNH             ;X0=A/2
            RRC       XNL
SQR_1       CALL      #XDIV           ;R12XR13=A/XN
            ADD       R13,XNL         ;XN+1=XN+A/XN
            ADDC      R12,XNH
            RRA       XNH             ;XN+1=1/2(XN+A/XN)
            RRC       XNL
            CMP       XNH,R12         ;IS HIGH WORD OF XN+1 = XN
            JNE       SQR_1           ;NO, ANOTHER APPROXIMATION
            CMP       XNL,R13         ;YES, IS LOW WORD OF XN+1 = XN
            JNE       SQR_1           ;NO, ANOTHER APPROXIMATION
SQR_3       RET                       ;YES, RESULT IS XNH.XNL

;*************************************************************
;EXTENDED UNSIGNED DIVISION
;R8|R9 / R10|R11 = R12|R13, REST IS R14|R15
;THIS DIVISION ROUTINE IS WRITTEN EXCLUSIVELY FOR THE SQUARE
;ROOT ROUTINE TO OPTIMIZE THE EXECUTION TIME FOR THE DIVISION
;OF A/XN.
;*************************************************************

XDIV
            PUSH      R8              ;SAVE OPERANDS ONTO THE STACK
            PUSH      R9
            PUSH      R10
            PUSH      R11
            MOV       #48,R7          ;ONLY 48 LOOPS ARE NECESSARY
            CLR       R15             ;CLEAR REST
            CLR       R14
            CLR       R12             ;CLEAR RESULT
            CLR       R13
```

```
L$361   RLA     R9              ;SHIFT ONE BIT OF R8|R9 TO
                                ;R14|R15
        RLC     R8
        RLC     R15
        RLC     R14
        CMP     R10,R14         ;IS SUBTRACTION NECESSARY?
        JLO     L$364           ;NO
        JNE     L$363           ;YES
        CMP     R11,R15         ;R11=R15
        JLO     L$364           ;NO
L$363   SUB     R11,R15         ;YES, SUBTRACT
        SUBC    R10,R14

L$364   RLC     R13             ;SHIFT RESULT TO R12|R13
        RLC     R12
        DEC     R7              ;ARE 48 LOOPS OVER ?
        JNZ     L$361           ;NO
        POP     R11             ;YES, RESTORE OPERANDS
        POP     R10
        POP     R9
        POP     R8
        RET
```

5.12 Trigonometric Calculation

As a matter of principle there are three methods of calculating trigonometric functions:

1. Power series
   e.g. Series of Taylor
2. Straight line approximation
   Fetch the gradient and offset of a straight line equation from a table
   e.g. $\sin(x) = m_x \cdot x = t_x$; m_x and t_x are stored in a table.
3. Fetching the calculated values from a table.

The first method is the most time-consuming method, since there is no hardware multiplier implemented in the processor. The advantage of this method is the optional accuracy. The higher the accuracy the higher is the ordinal number of the power series. The multiplication can be reduced by using the horner scheme.

As opposed to the first method, the following two methods determine the value indirectly from tables. For this purpose the trigonometric curves are divided into sections and therefore the accuracy of the result is limited. The straight line approximation needs one table to determine the gradient and one table to determine the offset of a straight line.

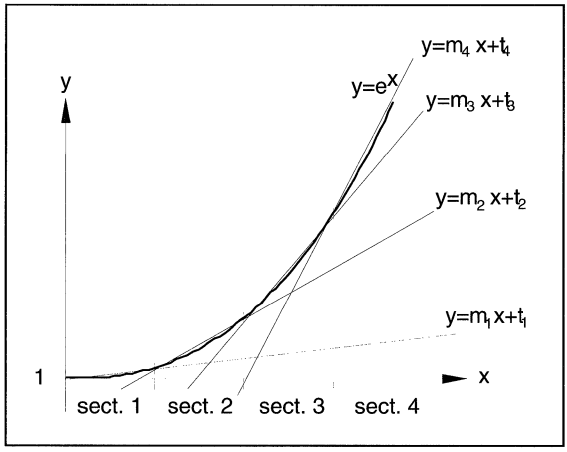


Figure 5.2: Straight Line Approximation

```
; IN THE FOLLOWING EXAMPLE THE SUBROUTINE MPYU IS USED.
; THIS ROUTINE IS DEFINED IN THE CHAPTER INTEGER SUBROUTINES.
; USING THE MPYU SUBROUTINE, THE CORRESPONDING EQUATIONS MUST
; BE INSERTED IN THE PROGRAM OF COURSE.
; X IS IN IROP1

SECT1     .EQU      2
SECT2     .EQU      5
SECT3     .EQU      8

IROP1     .EQU      R4
PTAB      .EQU      R5

GRA_TAB   .WORD     1,2,3,4
OFF_TAB   .WORD     11,22,33,44


;SUBROUTINE TO CALCULATE TRIGONOMETRIC FUNCTIONS WITH THE STRAIGHT
;LINE APPORXIMATION
;INPUT: OPERAND IN IROP1
;OUTPUT: RESULT IN IRACL

EX        CLR       PTAB                ;CLEAR TABLE POINTER
          CMP       #SECT1,IROP1        ;IS X IN SECT. 1 ?
          JLO       EX_1                ;YES
          INCD      PTAB                ;NO, ACTUALIZE POINTER
          CMP       #SECT2,IROP1        ;IS X IN SECT. 2 ?
```

```
        JLO     EX_1                    ;YES
        INCD    PTAB                    ;NO, ACTUALIZE POINTER
        CMP     #SECT3,IROP1            ;IS X IN SECT. 3 ?
        JLO     EX_1                    ;YES
        INCD    PTAB                    ;NO, X IS IN SECT. 4
EX_1    MOV     GRA_TAB(PTAB),IROP2L    ;FETCH GRADIENT
        CALL    #MPYU                   ;IRACL=M*X
        ADD     OFF_TAB(PTAB),IRACL     ;IRACL=M*X+T
        RET
```

The straight line approximation method is advantageous, if the trigonometric curve can be divided into a few sections. If many sections are needed, the search for the right section will take a lot of time and code space. A solution is to calculate the pointer by the x-value immediately.

For example:

Range of x-values : 0.0 ... 10.0
Sections needed at least : 8

To get the right pointer from the x-value only, the input value is used. This means that if the trigonometric curve is divided into 10 sections, the input value is already the pointer to the tables.

The third method to calculate a trigonometric function is to get the value directly from a table. This is suitable if only a few operands are possible. For example the operands of a sine wave can only be one of 46 possible values of one period.

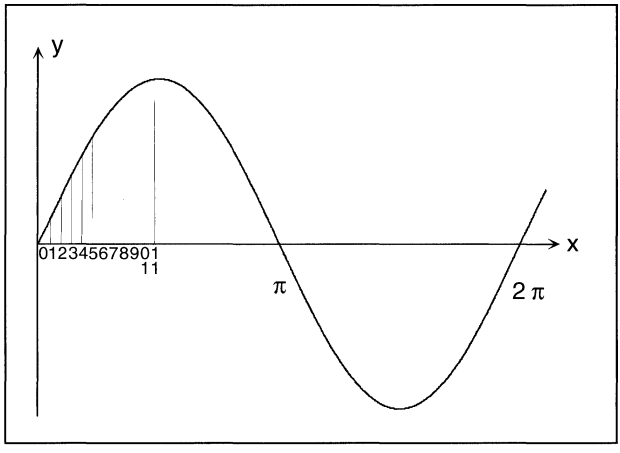


Figure 5.3: Sine Wave Approximation

Only the values of the quarter of the sinus wave are stored in the table. All values can be determined as follows:

$0 < x < \pi/2$	$y=y(x)$
$\pi/2 < x < \pi$	$y = y(INV(x)+\pi)$
$\pi < x < 2\pi$	$y = -y(x-\pi)$

The accuracy of the values that are stored in the table is 8 bit and can be increased to 16 bit. The table length can be up to 65536 bytes or words. In this example 12 values are stored (including 0) for one quarter of the sine wave.

```
HALFW     .EQU      24              ;NUMBER OF VALUES FOR THE HALF
                                    ;WAVE
QUARTW    .EQU      12              ;NUMBER OF VALUES FOF THE
                                    ;QUARTER

;*************************************************************
;MAKE_SIN;    SIN(R4)=R5
;IN THE SIN_TAB ONLY THE NUMBERS AFTER THE POINT ARE PERFORMED.
;THE OPERAND HAS TO BE LOADED INTO R4 BEFORE CALLING SIN AND
;MUST BE A NUMBER BETWEEN 0 AND 45. THE RESULT WILL BE IN R5
;*************************************************************
SIN_TAB   .BYTE     0,22H,45H,65H,85H,0A1H,0BBH,0D1H,0E3H,0F1H
          .BYTE     0FAH,0FFH
SIN       PUSH      R4
          CMP       #HALFW,R4       ;IS X IN THE NEGATIVE HALF WAVE?
          JLO       SIN_1           ;NO
          SUB       #HALFW-1,R4     ;YES, CORRECT X
SIN_1     CMP       #QUARTW,R4      ;IS X IN THE 2TH HALF OF THE
                                    ;HALF WAVE
          JLO       SIN_2           ;NO
          INV       R4
          ADD       #HALFW,R4       ;YES, ADJUST X
SIN_2     MOV.B     SIN_TAB(R4),R5  ;GET RESULT FROM TABLE
          POP       R4
          CMP       #HALFW,R4       ;IS X IN THE NEGATIVE HALF WAVE?
          JLO       SIN_END         ;NO RESULT IS OK
          INV       R5              ;YES, RESULT HAS TO BE NEGATED
          INC       R5
SIN_END RET
```

Topics

Figures

Notes

Tables

6 I/O Module Programming Examples

6.1 Initialization

One common task is getting information from the connected keys, jumpers and digital signals. Only the methods for connecting a keyboard to the MSP430 are described below. Other digital signals can be read in the same way.

The input pins can be the I/O-port, and unused analogue inputs can be switched to digital inputs. If the I/O-port is used for inputs, then wake-up by input changes is possible. If one of the input signal changes of interest occurs, an interrupt is given and wake-up occurs.

There are several methods of building a keyboard. The easiest way is to connect the keys directly to input pins. The software to get the pressed key is also easy to implement. Only the test of one register is necessary to get information about the keys. This method is possible if only a few keys are necessary to control the programme flow.

If more input signals exists than free inputs, then scanning is necessary. The scanning outputs can be the I/O-port and unused select outputs On. The scanning input can be I/O-ports and analogue inputs as described above.

The interrupt handler for the I/O pins P0.2 to P0.7 is as follows.

```
IOINTR    PUSH      R5              ;SAVE R5
          MOV.B     &IOFLAG,R5      ;READ INTERRUPT FLAGS
          AND.B     P0IE,R5         ;MASK ALLOWED INTERRUPTS
          BIC.B     R5,&IOFLAG      ;ADDITIONAL SET BITS ARE NOT
                                    ;CLEARED
          EINT                      ;ALLOW INTERRUPT NESTING

          ;R5 CONTAINS INFORMATION WHICH I/O PIN CAUSED INTERRUPT
          .....
          POP       R5              ;RESTORE R5
          RETI

          .SECT     "IO_VECT",0FFE0H
          .WORD     IOINTR
          .SECT     "RST_VEC".0FFFEH
          .WORD     START
```

6.2 Keyboard-Matrix of 4 x 4 Keys

This is the most commonly used method of connecting a keyboard to a microprocessor, because of the minimal hardware requirement. Only the diodes are necessary to protect the outputs P0.0 to P0.3 if more than one key is pressed.

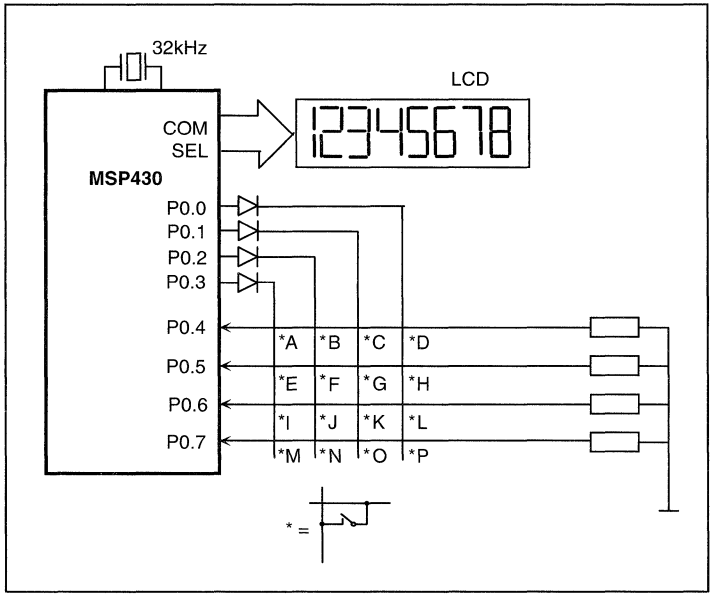


Figure 6.1: 4x4 Keyboard matrix

To check the keyboard, one output line (P0.0 to P0.3) has to be high. If a key in this line is pressed, the corresponding input line will be high too. To scan all keys, the outputs have to be high, one after the other, and the input lines can then be read in.

The following table shows the key-code which is on the I/O port, if the correspondig key was pressed.

Key	inp outp	Hex
A	0001 1000	18
B	0001 0100	14
C	0001 0010	12
D	0001 0001	11
E	0010 1000	28
F	0010 0100	24
G	0010 0010	22
H	0010 0001	24
I	0100 1000	48
J	0100 0100	44
K	0100 0010	42
L	0100 0001	41
M	1000 1000	88
N	1000 0100	84
O	1000 0010	82
P	1000 0001	81

Table 6.1: Configuration of the MUX Modes

The Routine to scan the 4x4 keyboard now follows. The pressed key will be stored in KEYOUT. If more than one key was pressed, the result is the binary OR of the keycodes shown in the table above. If no key was pressed, KEYOUT becomes zero.

```
KEYMASK   .EQU      R4                ;MASK FOR THE OUTPUT
KEYOUT    .EQU      R6                ;RESULT OF THE SCAN


          MOV.B     #0FH,&P0DIR       ;INITIALIZATION OF THE I/O PORT

;SUBROUTINE TO GET THE SCANCODE FROM THE KEYBOARD WITH THE
;4X4 MATRIX
;OUTPUT: SCANCODE AS SHOWN IN TABLE 6.1 IN KEYOUT
;USE REGISTER: R7

KEYSCAN   MOV       #4,R7             ;LOOP COUNTER
          MOV       #1,KEYMASK
SCAN_1    MOV.B     KEYMASK,&P0OUT    ;SCAN ONE LINE
          TST.B     &P0IN             ;ANY KEY PRESSED ?
          JZ        SCAN_2            ;NO
```

```
            BIS.B     KEYMASK,KEYOUT    ;YES, SET CORRESPONDING BITS
            BIS.B     &P0IN,KEYOUT
SCAN_2      RLA.B     KEYMASK           ;NEXT SCANNING LINE
            DEC       R7                ;ALL LINES TESTED ?
            JNZ       SCAN_1            ;NO, NEXT LINE
            RET
```

6.3 Keyboard-Matrix of 3 x 4 Keys

The next scheme describes a keyboard matrix which needs only 4 I/O pins to handle 12 keys. This is possible because of the capability of switching the direction of the I/O pins independently.

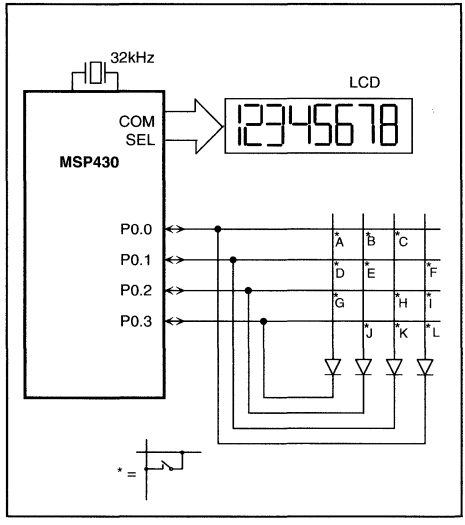


Figure 6.2: 3 x 4 Keyboard Matrix

Key	Outp Inp	Hex
A	0001 1000	18
B	0001 0100	14
C	0001 0010	12
D	0010 1000	28
E	0010 0100	24
F	0010 0001	21
G	0100 1000	48
H	0100 0010	42
I	0100 0001	41
J	1000 0100	84
K	1000 0010	82
L	1000 0001	81

Table 6.2:
Configuration of the MUX Modes

```
KEYMASK   .EQU     R4                ;MASK FOR THE OUTPUT
KEYOUT    .EQU     R6                ;RESULT OF THE SCAN


;SUBROUTINE TO GET THE SCANCODE AS SHOWN IN TABLE 6.2
;OUTPUT: KEYOUT = SCANCODE
;USE REGISTER: R7,R8,R9

KEYSCAN   MOV      #4,R7             ;LOOP COUNTER
          MOV      #1,KEYMASK        ;FIRST SCANNING LINE
          BIC.B    #0FH,&P0OUT       ;INITIALISATION VALUE
          BIC.B    #0FH,&P0DIR       ;SET ONLY KEYBOARD LINES TO
                                     ;INPUT
SCAN_1    BIS.B    KEYMASK,&P0OUT    ;SCAN ONE LINE
          BIS.B    KEYMASK,&P0DIR    ;SET THE OUTPUT LINE
          MOV.B    &P0IN,R8          ;WHOLE INPUT REGISTER TO R8
          BIC.B    KEYMASK+0F0H,R8   ;R8 CONTAINS THE PRESSED KEY
          TST.B    R8                ;WAS ANY KEY IN THIS LINE
                                     ;PRESSED ?
          JZ       SCAN_2            ;NO, TEST ANOTHER LINE
          MOV.B    KEYMASK,R9        ;YES, SET THE CORRESPONDING
                                     ;BITS
          RLA.B    R9                ;SHIFT THE OUTPUT MASK 4 BITS
                                     ;LEFT
          RLA.B    R9
          RLA.B    R9
          RLA.B    R9
          BIS.B    R9,KEYOUT         ;SET THE MASKED OUTPUT BITS
          BIS.B    R8,KEYOUT         ;SET THE MASKED INPUT BITS
SCAN_2    BIC.B    KEYMASK,&P0OUT    ;CLEAR OUTPUT REGISTER
          BIC.B    KEYMASK,&P0DIR    ;CLEAR DIRECTION REGISTER
          RLA.B    KEYMASK           ;NEW SCANNING LINE
          DEC      R7                ;ALL LINES TESTED ?
          JNZ      SCAN_1            ;NO, NEXT LINE
          RET                        ;YES, RETURN
```

6.4 Noise Generator

The Noise Generator is constructed with a 32bit shift register having feedback which is generated by a logical XNOR of bit 1 and bit 14 of the 16bit register NREG+1. Therefore, the shift register receives about 2 billion different conditions before the values are repeated.

The output can be a loudspeaker with amplifier and low-pass filter. If a random number is necessary, the value of the register NREG or NREG+1 has simply to be used.

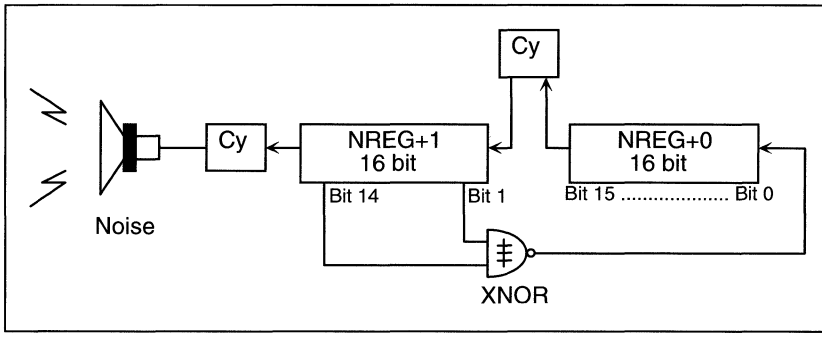


Figure 6.3: Noise Generator

```
LS          .EQU     1              ;OUTPUT PIN FOR THE LOUDSPEAKER
NREG0       .EQU     R4
NREG1       .EQU     R5

            BIS.B    #LS,&P0DIR     ;PREPARE I/O PIN FOR OUTPUT
NOISE       CLR      NREG0          ;STARTING VALUE FOR THE 32 BIT
                                    ;SHIFT
            CLR      NREG1          ;REGISTER, CAN BE EVERY OTHER
                                    ;NUMBER

NLOOP1      BIC.B    #LS,&P0OUT     ;SET OUTPUT TO LOW
NLOOP2      MOV      NREG1,R6       ;DUPLICATE NREG1
            RRC      R6             ;SHIFT BIT 14 TO BIT 1
            RRC      R6
            RRC      R6
            RRC      R6
            XOR      NREG1,R6       ;XOR BIT 14 AND BIT 1
            BIT      #4000H,R6      ;AND MOVE RESULT TO CARRY
            XOR      #1,SR          ;INVERT CARRY
                                    ;NOW CARRY IS BIT 14 XNOR BIT 1
            RLC      NREG0          ;SHIFT 32BIT REGISTER FOR ONE TIME
            RLC      NREG1
            JNC      NLOOP1         ;AND OUTPUT CARRY TO LS
            BIS.B    #LS,&P0OUT
            JMP      NLOOP2
```

One loop needs 17 cycles, so the sampling rate of the generated signal on the output is about 59 kHz and therefore the signal repeats after about 9 hours. This should be acceptable for a noise.

6.5 External EEPROM for Setup Values

To save important values in a nonvolatile storage, a serial EEPROM can be connected to the I/O port of the MSP430. This memory keeps its values even if the supply voltage is disconnected. In this way it is possible to store setup values which are necessary to initialize the system after the Powerup Reset.

The EEPROM is connected to the MSP430 by dedicated inputs and outputs. Three (two) control lines are necessary for proper operation:

- Data line SDA: an I/O-port is needed for this bi-directional line. Data can be read from and written to the EEPROM
- Clock line SCL: an output line is sufficient for the clock line. This clock line may be used for other peripheral devices too, if it is ensured that no data is present on the data line during use.
- Supply line: if the current consumption of the EEPROM when not in use is too high, then switching of the EEPROM's Vcc is necessary. Three possible solutions are shown:
  1. The EEPROM is connected to SVcc. This is a very simple way to have the EEPROM switched off when not in use
  2. The EEPROM is switched on and off by an external PNP transistor.
  3. The EEPROM is connected permanently to +5V , if its power consumption does not play a role.

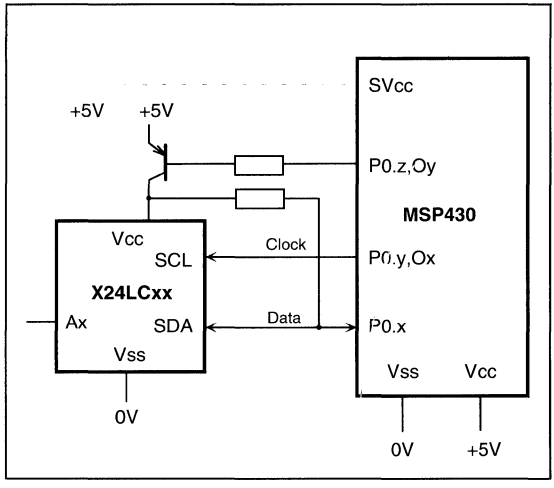


Figure 6.4: External EEPROM Connection

An additional way to connect an EEPROM to the MSP430 is shown in the chapter describing the I2C-Bus.

Note: Next example does not contain the necessary delay times

The next example does not contain the necessary delay times between the setting and resetting of the clock and data bits. These delay times can be seen in the specifications of the EEPROM device. With a processor frequency of 1MHz each one of the next instructions needs 5µs.

EXAMPLE: The EEPROM with the dedicated I/O-lines is controlled with normal I/O-instructions. The SCL line is driven by O17; the SDA line is driven by P0.6:

```
P0OUT     .EQU     011H            ; PORT0 OUTPUT REGISTER
P0DIR     .EQU     012H            ; PORT0 DIRECTION REGISTER
SCL       .EQU     0F0H            ; O17 CONTROLS SCL, 039H LCD
                                   ; ADDRESS
SDA       .EQU     040H            ; P0.6 CONTROLS SDA
LCDM      .EQU     030H            ; LCD CONTROL BYTE

; INITIALIZE I2C BUS PORTS:
; INPUT DIRECTION:   BUS LINE GETS HIGH
; OUTPUT BUFFER LOW: PREPARATION FOR LOW SIGNALS
          BIC.B    #SDA,&P0DIR     ; SDA TO INPUT DIRECTION
          BIS.B    #SCL,&LCDM+9    ; SET CLOCK HI
          BIC.B    #SDA,&P0OUT     ; SDA LOW
          ...

; START CONDITION: SCL AND SDA ARE HIGH, SDA IS SET LOW,
; AFTERWARDS SCL GOES LO
          BIS.B    #SDA,&P0DIR     ; SET SDA LO (SDA GETS OUTPUT)
          BIC.B    #SCL,&LCDM+9    ; SET CLOCK LO

; DATA TRANSFER: OUTPUT OF A "1"
          BIC.B    #SDA,&P0DIR     ; SET SDA HI
          BIS.B    #SCL,&LCDM+9    ; SET CLOCK HI
          BIC.B    #SCL,&LCDM+9    ; SET CLOCK LO

; DATA TRANSFER: OUTPUT OF A "0"
          BIS.B    #SDA,&P0DIR     ; SET SDA LO
          BIS.B    #SCL,&LCDM+9    ; SET CLOCK HI
          BIC.B    #SCL,&LCDM+9    ; SET CLOCK LO

; STOP CONDITION: SDA IS LOW, SCL IS HI, SDA IS SET HI
          BIC.B    #SDA,&P0DIR     ; SET SDA HI
          BIS.B    #SCL,&LCDM+9    ; SET SCL HI
```

The examples shown above for the different conditions can be implemented into a subroutine which outputs the content of a register. This shortens the necessary ROM code significantly. Instead of line Ox for the SCL line, another I/O-port P0.x may be used. See section I2C Bus Connection for more details of such a subroutine.

6.6 I2C BUS Connection

If more than one device is to be connected to the I2C-Bus, then two I/O-ports are needed for the control of the I2C-peripherals. The reason for this is the need to switch SDA and SCL to the high impedance state.

The figure below shows the connection of three I2C-peripherals to the MSP430:
- An EEPROM PCF8581 with 128x8-bit data
- An EEPROM X24LCxx with 2048x8-bit data
- An 8-bit DAC/ADC

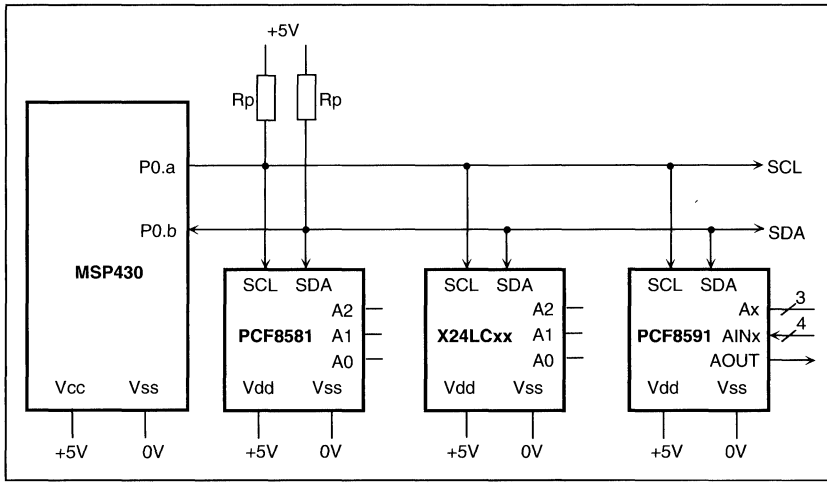
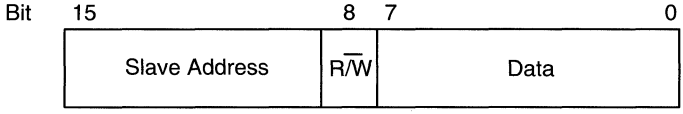


Figure 6.5: I2C-Bus Connections

Note: Next example does not contain the necessary delay times

The next example does not contain the necessary delay times between the setting and resetting of the clock and data bits. These delay times can be seen in the specifications of the peripheral device.

The complete I2C-Handler for one byte of data follows. The data pin SDA needs an I/O-pin (Port0); the clock pin SCL may be an I/O-pin or an output pin.




```
SCLDAT    .EQU     011H                ; P0OUT
SCLEN     .EQU     012H                ; P0DIR
SDA       .EQU     040H                ; P0.6 CONTROLS SDA
SCL       .EQU     080H                ; P0.7 CONTROLS SCL
SDADAT    .EQU     011H                ; P0 OUTPUT DIRECTION
                                       ; REGISTER
SDAEN     .EQU     012H                ; P0 DIRECTION REGISTER

; INITIALIZATION FOR THE I2C BUS PORTS:
; INPUT DIRECTION:    BUS LINES GET HIGH BY PULL-UPS
; OUTPUT BUFFERS LOW: PREPARATION FOR LOW ACTIVE SIGNALS
; INITIALIZATION FOR SDA AND SCL FROM PORT0

          BIC.B    #SCL+SDA,&SDAEN     ; SCL AND SDA TO INPUT
                                       ; DIRECTION
          BIC.B    #SCL+SDA,&SDADAT    ; SCL AND SDA OUTPUT BUFFER
                                       ; LOW
          ...

; INITIALIZATION FOR SDA AT PORT0, SCL AT TP.X (EVEOPT)

          BIC.B    #SDA,&SDAEN         ; SDA TO INPUT DIRECTION
          BIC.B    #SDA,&SDADAT        ; SDA OUTPUT BUFFER LOW
          BIC.B    #SCL,&SCLEN         ; SCL TO INPUT DIRECTION
          BIC.B    #SCL,&SDADAT        ; SCL OUTPUT BUFFER LOW
          ...
          ...

;*******************************************************************
; I2C-HANDLER: OUTPUTS OR READS 8-BIT DATA
;
; WRITE: R/@W = 0. R6 CONTAINS SLAVE ADDRESS AND 8-BIT DATA
;          RETURN: C = 0: TRANSFER OK (R6 UNCHANGED)
;                  C = 1: ERROR (R6 UNCHANGED)
;CALL     MOV.B    DATA,R6             ; 8-BIT DATA TO R6
;         BIS      (2*ADDR)*0100H,R6   ; ADDRESS AND FUNCTION
;         CALL     #I2CHND             ; CALL HANDLER
;         JC       ERROR               ; C = 1: ERROR OCCURED
;
;READ:     R/@W = 1. R6 CONTAINS SLAVE ADDRESS , LOW BYTE UNDEFINED
```

```
;          RETURN: R6 CONTAINS 8-BIT DATA IN LOW BYTE, HI BYTE = 0
;CALL     MOV       (2*ADDR+1)*0100H,R6  ; ADDRESS AND FUNCTION
;         CALL      #I2CHND              ; CALL HANDLER
;         ...                            ; 8-BIT INFO IN R6 LO
;*****************************************************************

I2CHND    .PUSH     R5                   ; SAVE REGISTERS

; I2C START CONDITION: SCL AND SDA ARE HIGH, SDA GOES LOW
; THEN SCL GOES LOW

          BIS.B     #SDA,&SDAEN          ; SET SDA LO
          BIS.B     #SCL,&SCLEN          ; SET SCL LINE LO

; SENDING OF THE ADDRESS BITS (7) AND R/@W-BIT

          MOV       #8000H,R5            ; BIT MASK MSB
I2CCL     BIT       R5,R6                ; BIT -> CARRY
          CALL      #I2CSND              ; SEND CARRY
          CLRC
          RRC       R5                   ; NEXT ADDRESS BIT
          CMP       #080H,R5             ; R/@W SENT?
          JNE       I2CCL                ; NO, CONTINUE

; ADDRESS AND R/@W SENT: RECEIVE OF ATHEN CKNOWLEDGE BIT,
; DECISION IF READ OR WRITE

          CALL      #I2ACKN
          JC        I2CERR               ; NO ACKNOWLEDGE, ERROR
          BIT       #100H,R6             ; READ OR WRITE?
          JNZ       I2CRI

; WRITE: CONTINUE WITH 8-BIT DATA IN LOW BYTE OF R6

I2CWL     BIT       R5,R6                ; WRITE: CONTINUE WITH DATA
          CALL      #I2CSND
          CLRC
          RRC       R5                   ; IF TESTBIT IN CARRY:
                                         ; FINISHED
          JNC       I2CWL
          CALL      #I2CACKN             ; ACKNOWLEDGE BIT -> CARRY

; CARRY INFORMATION: 0: OK,  1: ERROR

I2CEND    .EQU      $
I2CERR    BIC.B     #SCL,&SCLEN          ; STOP CONDITION
          BIC.B     #SDA,&SDAEN          ; SET SDA HI
          RET                            ; CARRY INFO UNDESTROYED

; READ: READ 8 DATA BITS TO R6 LOW BYTE. R5 = 080H
```

```
I2CRI     CALL      #I2CRD              ; READ BIT -> CARRY
          RLC.B     R6                  ; CARRY TO LSB R6
          RRA       R5                  ; BIT MASK USED FOR COUNT
          JNC       I2CRI               ; BIT MASK IN CARRY:
                                        ; FINISHED
          CALL      #I2C0               ; ACKNOWLEDGE BIT = 0
          JMP       I2CEND              ; CARRY = 0

; SUBROUTINES FOR I2C-HANDLER

; SENDROUTINE: INFO IN CARRY IS SENT OUT.
; ACKNOWLEDGE BIT SUBROUTINE IS USED FOR CLOCK OUTPUT

I2CSND    JNC       I2C0                ; INFO IN CARRY
          BIC.B     #SDA,&SDAEN         ; INFO = 1
          JMP       I2CACKN
I2C0      BIS.B     #SDA,&SDAEN         ; INFO = 0

; READING OF ACKNOWLEDGE (OR DATA) BIT TO CARRY

I2CACKN   .EQU      $
I2CRD     BIC.B     #SCL,&SCLEN         ; SET CLOCK HI
          BIT.B     #SDA,&SDAIN         ; READ DATA TO CARRY
          BIS.B     #SCL,&SCLEN         ; CLOCK LO
          RET
```

Topics

Figures

7 Timer Examples

7.1 Watchdog

The Watchdog timer can be used as Watchdog, or as a normal timer. It is even possible to switch between these functions. The following example shows the correct method. The watchdog mode is first selected and a time interval of 0.5 ms is performed. The Watchdog Timer then works as a normal timer with an time interval of 250 ms. The corresponding interrupt service routine can be programmed as usual.

```
WDTCTL    .EQU     0120H          ;ADDRESS OF WATCHDOG TIMER
WDTPW     .EQU     05A00H         ;PASSWORD
T250MS    .EQU     5              ;INTERVAL IS SET TO 250 MS
T05MS     .EQU     2              ;INTERVAL IS SET TO 0.5 MS
CNTCL     .EQU     8              ;BITPOSITION TO RESET WDTCNT
TNSEL     .EQU     010H           ;BITPOSITION TO SELECT TIMER MODE
IE1       .EQU     0              ;ADDRESS OF IE1


;FOLLOWING INSTRUCTION SELECTS THE WATCHDOG MODE AND A TIME
;INTERVAL OF 0,5MS IS PERFORMED.
          MOV      #WDTPW+CNTCL,&WDTCTL     ;RESET WDT COUNTER
          MOV      #WDTPW+T05MS,&WDTCTL     ;WATCHDOG MODE AND
                                            ;0.5 MS TIME INTERVAL
          .....

;TO CHANGE TO TIMER MODE AND A TIMER INTERVAL OF 250 MS, THE
;FOLLOWING INSTRUCTION SEQUENCE CAN BE USED:
          MOV      #WDTPW+CNTCL,&WDTCTL     ;CLEAR WDT COUNTER
          MOV      #WDTPW+T250MS+TMSEL,&WDTCTL  ;SELECT 250 MS AND
                                            ;TIMER MODE
          BIS.B    #1,&IE1                  ;ENABLE WDT INTERRUPT
          EINT                              ;ALLOW INTERRUPT
          .....

;IN THE TIMER MODE A INTERRUPT SERVICE ROUTINE IS NECESSARY TO
;HANDLE THE INTERRUPT CAUSED BY THE WATCHDOG TIMER
WDTISR                                      ;INTERRUPT SERVICE
          .....                             ;ROUTINE FOR THE WDT
          RETI                              ;IN THE TIMER MODE
          .....
          .SECT    "WDTVECT",0FFF4H         ;INTERRUPT VECTOR OF
          .WORD    WDTISR                   ;WDT IN TIMER MODE
```

7.2 8 bit Timer

7.2.1 Measuring the Pulse Width

The following example shows the performance of the 8bit Timer / Counter to measure the pulse width of a signal on port pin P0.1. In the 8bit Counter Control Register there is an option to select the source of the clock input to the signal on pin P0.1 AND MCLK. Therefore, if the signal at pin P0.1 is high, the MCLK pulses are counted. The maximum length of the pulse can be up to 4096 seconds. The result of the following pulse will be 13.

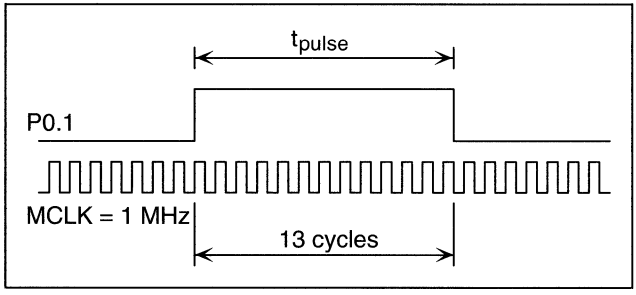


Figure 7.1: Measuring the Pulse Width

```
;***************************************************************
;THIS ROUTINE MEASURES THE TIME DURING A PULSE AT THE I/O PIN
;P0.1 IS HIGH. THE RESULT IN 1/(2^20) SEC IS TO BE STORED IN ;R5|R6.
;***************************************************************
;CONFIGURE 8BIT TIMER TO COUNT THE MCLK'S DURING THE SIGNAL
;AT THE I/O PIN P0.1 IS HIGH.
          MOV.B     #ENCNT+ISCTL+SSEL0+SSEL1,&TCCTL
          CLR.B     &TCPLD
          CLR.B     &TCDAT              ;CLEAR COUNTER
          BIC.B     #P0_1,&P0DIR        ;P0.1 IS INPUT
          BIS.B     #8,&IE1             ;ALLOW COUNTER INTERRUPT
          EINT

LOOP      BIT.B     #P0_1,&P0IN         ;IS SIGNAL HIGH ?
          JNZ       LOOP                ;YES
          BIS.B     &TCDAT,R5           ;RESULT IS R5|R6
          JMP       LOOP

TC8ISR
          ADD       #100H,R6
          ADC       R5
          RETI
```

```
;INTERRUPT VECTOR ADDRESSES:
          .SECT     "TC8VECT", 0FFF8H
          .WORD     TC8ISR
          .SECT     "RSTVECT", 0FFFEH    ;PUC/RESET ADDRESS
          .WORD     START
```

7.2.2 Output Pulse Responding to Input Signal

Output a 1ms pulse on every positive edge of an input signal.

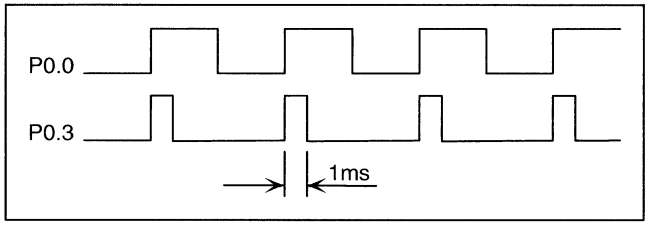


Figure 7.2: Output a 1 ms Signal

In this example, a rising edge on the P0.2 input pin causes a 1 ms pulse to be outputted on the P0.3 pin. To give a simple application, this could be used in a 50 Hz lamp dimmer or motor speed controller, where the input is the 50 Hz signal and the output connects to the output driver.

```
;INITIALIZATION OF THE COUNTER:
          MOV.B     #-32,&TCPLD      ;LOAD PRELOAD REGISTER
          CLR.B     &TCDAT           ;LOAD COUNTER WITH PRELOAD
          MOV.B     #SSEL0+ISCTL,&TCCTL
                                     ;SET TC8 TO ACLK CLOCK SOURCE,
                                     ;INTERRUPT FROM COUNTER AND
                                     ;ENABLE COUNTER
          BIS.B     #P0_1IFG,&IE1    ;ENABLE TC8 INTERRUPT

;INITIALIZATION OF THE I/O PORT
          BIS.B     #P0_3,&P0DIR     ;P0_3 TO OUTPUT
          BIS.B     #P0_0IE,&IE1     ;ENABLE P0_0 INTERRUPT
          BIC.B     #P0_0,&P0IES     ;SET INTERRUPT EDGE TO LO/HI
          EINT                       ;SET GIE BIT IN SR
LOOP      JMP       LOOP             ;ENDLESS LOOP

P0INT     BIS.B     #ENCNT,&TCCTL    ;ENABLE COUNTER
          BIS.B     #P0_3,&P0OUT     ;START OF 1 MS PULSE
          RETI
```

```
TC8INT    BIC.B     #ENCNT,&TCCTL    ;DISABLE COUNTER
          BIC.B     #P0_3,&P0OUT     ;END OF 1 MS PULSE
          RETI

;INTERRUPT VECTOR ADDRESSES:
          .SECT     "P0_1VECT",0FFF8H    ;ADDRESS FOR TC8 INTERRUPT
          .WORD     TC8INT
          .SECT     "P0_0VECT",0FFFAH    ;ADDRESS FOR P0_0 INTERRUPT
          .WORD     P0INT
          .SECT     "RSTVECT",0FFFEH     ;PUC/RESET ADDRESS
          .WORD     START
```

7.3 Basic Timer

7.3.1 Generating Interrupts Sequentially

The Basic timer is well suited for generating interrupts periodically. The following software routine generates a one second interrupt sequence. In the corresponding interrupt service routine, the LCD can be updated or a new measurement cycle can be initiated, for example.

```
;INITIALISATION

START     BIC.B     #0E7H,&BTCTL        ;CLEAR CONFIGURATION BITS FOR
                                        ;BT
          BIS.B     #IP2+DIV,&BTCTL     ;NOW THE BT IS CONFIGURED
          BIS.B     #BTIE,&IE2          ;ALLOW INTERRUPT CAUSED BY BT
          ...
          EINT                          ;ALLOW ALL INTERRUPTS
          .....                         ;PROGRAM

;INTERRUPT SERVICE ROUTINES

BT_ISR
          CALL      #LCD_UPD            ;UPDATE LCD
          CALL      #NEW_MEAS           ;INITIATE A NEW MEASUREMENT
          .....
          RETI

;INTERRUPT VECTOR ADDRESS

          .SECT     "BTVECT",0FFE2H     ;ADDRESS FOR BASIC TIMER INT.
          .WORD     BT_ISR
          .SECT     "RSTVECT",0FFFEH    ;PUC/RESET ADDRESS
          .WORD     START
```

7.3.2 PWM Modulation

Output a 1 kHz signal with a varying duty cycle

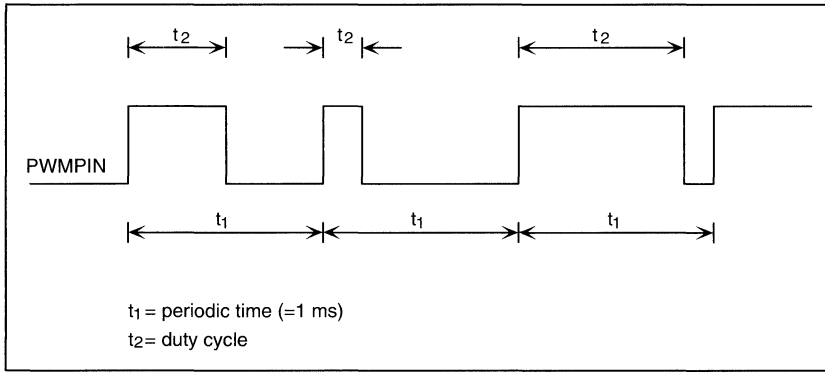


Figure 7.3: PWM Modulation

The Basic Timer controls the period of the signal (t1) and is not changed in this routine, while the 8 bit Timer / Counter controls the varying duty cycle (t2). The Basic Timer Service Routine will be entered each time the interrupt request flag BTIFG is set. The main programme is required to load any new values for the PWM duty cycle into the working Register, which is loaded into the Pre-load Register by the interrupt routine.

The discrepancy between the value in the PWM register and the corresponding duty cycle is caused by the interrupt service rotine, which takes about 34 cycles, if the value in the PWM regiser is lower than 100h. If it is higher than 100h, the 8 bit Timer service routine is called more times, and therefore it takes additional cycles (14 cycles per additional interrupt request).

The periodic time can be increased by choosing another time devision factor for the Basic Timer. Therefore the difference between the value in the PWM register and the duty cycle must not be considered.

```
;*******************************************************
;PWM
;      T1 = 939 CYCLES = 1117 HZ ( MCLK = 2^20 HZ )
;           CAUSED BY BASIC TIMER WITH DEVIDING FACTOR 32
;      T2,MIN = 40 CYCLES (PWM=6)
;      T2,MAX = 908 CYCLES (PWM=830)
;*******************************************************

RAMSTART  .SECT     "RAM",0200H
PWM       .WORD     0                   ;WORKING REGISTER
PWMLOAD   .WORD     0                   ;MAX=990, MIN=10


START     .SECT "PROG", 0F000H


;INITIALIZATION OF THE BASIC TIMER FOR THE 1 MS INTERRUPT INTERVAL
          BIC.B     #0E7H,&BTCTL        ;CLEAR CONFIGURATION BITS FOR
                                        ;BT
          BIS.B     #IP2,&BTCTL         ;INTERRUPT INTERVAL IS 1 MS
          BIS.B     #BTIE,&IE2          ;ENABLE BT INTERRUPT
          BIS.B     #BTME,&ME2          ;ENABLE BT MODULE


;INITIALZATION OF THE 8 BIT TIMER / COUNTER
          CLR.B     &TCCTL              ;CLEAR CONFIGURATION OF 8 BIT
                                        ;TIMER
          BIS.B     #SSEL1+ISCTL        ;CARRY IS INTERRUPT SOURCE
                                        ;CLOCK INPUT IS MCLK
          BIS.B     #P0_1IE,&IE1        ;ENABLE 8 BIT TIMER INTERRUPT


;INITIALIZATION OF THE I/O PORT
          BIS.B     #PWMPIN,&P0DIR      ;SET DEDICATED PIN TO OUTPUT
          BIC.B     #PWMPIN,&P0OUT      ;PWM OUTPUT = LOW

          EINT                          ;ENABLE SELECTED INTERRUPTS
;MAIN PROGRAM CHANGES THE DURY CYCLE BY CHANGING THE VALUE IN
;THE RAM WORD PWM

MAIN      MOV       #1,&PWM
          CALL      #WAIT
          MOV       #100,&PWM
          CALL      #WAIT
          MOV       #1000H,&PWM
          CALL      #WAIT
          JMP       MAIN
```

```
WAIT      MOV       #500,R4
WAITL$    DEC       #R4
          JNZ       WAITL$
          RET


;BASIC TIMER INTERRUPT SERVICE ROUTINE
BTISR     MOV.B     &PWM,&TCPLD       ;LOAD PWM VALUE INTO PRELOAD
                                      ;REG
          INV.B     &TCPLD            ;8 BIT TIMER = UPCOUNTER !
          CLR.B     &TCDAT            ;LOAD PRELOAD REGISTER INTO
                                      ;COUNTER
          BIS.B     #PWMPIN,&P0OUT    ;OUTPUT PIN = HIGH
          BIS.B     #ENCNT,&TCCTL     ;ENABLE 8 BIT TIMER
          RETI

;8 BIT TIMER / COUNTER INTERRUPT SERVICE ROUTINE
TC8ISR    CLR.B     &TCPLD            ;CLEAR PRE-LOAD REGISTER
          BIC.B     #P0_1IFG,&IFG1    ;NECESSARY, IF PRE-LOAD VALUE
                                      ;IS FFH
          CLR.B     &TCDAT            ;AND LOAD NEW VALUE INTO
                                      ;COUNTER
          SUB       #100H,&PWM        ;ACTUALIZE WORKING REGISTER
          JNC       TC8_END           ;T2 OVER ?
          BIC.B     #PWMPIN,&P0OUT    ;YES, SET OUTPUT TO LOW
          BIC.B     #ENCNT,&TCCTL     ;DISABLE 8 BIT TIMER
TC8_END   RETI

;INTERRUPT VECTOR ADDRESSES:
          .SECT     "BTVECT",0FFE2H    ;ADDRESS FOR BT INTERRUPT
          .WORD     BTISR
          .SECT     "P0_1VECT",0FFF8H  ;ADDRESS FOR TC8 INTERRUPT
          .WORD     TC8ISR
          .SECT     "RSTVECT",0FFFEH   ;PUC/RESET ADDRESS
          .WORD     START
```

7.3.3 Software UART

The following software routines for implementing a Software UART use the 8 bit Timer/ Counter for generating the appropriate timing, and support half duplex protocols. The baudrate is up to 2400 bps with the ACLK of 32768 Hz. The transmit and receive routines are written as interrupt service routines, to get high performance. The communication parameters (Stop bit, Data bits, Parity) can be defined at the setting part at the beginning of the programme.

```
;*****************************************************************
; SOFTWARE UART FOR MSP430
; CONDITIONS: HALF DUPLEX, ACLK IS CLOCK SOURCE OF 8-BIT-
; TIMER/COUNTER
; REQUIRED RAM-SPACE FOR VARIABLES: 5 BYTES
;*****************************************************************

; DEFINED BY USER

BAUD                .SET  600           ;600, 1200, 2400
DATABITS            .SET  8             ;7, 8
PARITY              .SET  "NONE"        ;"EVEN", "ODD", "NONE"
STOPBITS            .SET  1             ;1, 2

;PROTOCOL DEFINITIONS

                    .IF PARITY = "NONE"
FRAME_END           .SET  2*(1+DATABITS+STOPBITS) ;# OF BITS (*2)
                    .ELSE
FRAME_END           .SET  2*(2+DATABITS+STOPBITS) ;# OF BITS (*2)
                    .ENDIF

                    .IF BAUD = 600
BITIME1             .SET  0100H - 55    ;TWICE USED    54.6133
BITIME2             .SET  0100H - 54    ;SINGLE USED
BITIME1_2           .SET  0100H - 27    ;HALFBIT       27.3067
                    .ENDIF

                    .IF BAUD = 1200
BITIME1             .SET  0100H - 27    ;TWICE USED    27.3067
BITIME2             .SET  0100H - 28    ;SINGLE USED
BITIME1_2           .SET  0100H - 14    ;HALFBIT       13.6533
                    .ENDIF

                    .IF BAUD = 2400
BITIME1             .SET  0100H - 14    ;TWICE USED    13.6533
BITIME2             .SET  0100H - 13    ;SINGLE USED
BITIME1_2           .SET  0100H -  7    ;HALFBIT        6.8267
                    .ENDIF

;I/O DEFINITION
P0IES               .SET  14H
;TIMER DEFINITIONS
TCDAT               .SET  44H
TCPLD               .SET  43H

; RAM DEFINITIONS
                    .SECT "RAM",200H
TXDATA              .BYTE 0,0           ;TRANSMIT SHIFT REGISTER
TXSTATUS            .BYTE 0             ;ACTUAL STATUS OF TRANSMIT
```

```
                                        ;SEQUENCE
TXFLAGS             .BYTE 0             ;FLAGS DURING TRANSMITTING
RXBUF               .BYTE 0             ;RECEIVE BUFFER FOR COMPLETED
                                        ;CHARS

RXDATA              .SET  TXDATA        ;RECEIVE SHIFT REGISTER
RXSTATUS            .SET  TXSTATUS      ;ACTUAL STATUS OF RECEIVE
                                        ;SEQUENCE
RXFLAGS             .SET  TXFLAGS       ;FLAGS DURING RECEIVING

REC_BIT             .SET  01H           ;FLAG INDICATES 'CHAR RECEIVED'
PAR_BIT             .SET  02H           ;PARITY BIT
ERR_BIT             .SET  04H           ;RECEIVE ERROR

; CONTROL DEFINITIONS
P0IES_1             .SET  02H           ;EDGE SELECT FOR P0.1 INTRPT
P0IFG_1             .SET  08H           ;INT. FLAG FOR P0.1

TCCTL               .SET  042H          ;TIMER/COUNTER CONTROL
                                        ;REGISTER
RXD                 .SET  1             ;RECEIVE DATA BIT IN TCCTL
TXD                 .SET  2             ;TRANSMIT DATA BIT IN TCCTL
RXACT               .SET  4             ;EDGE DETECT LOGIC BIT
ENCNT               .SET  8             ;COUNTER ENABLE BIT IN TCCTL
TXE                 .SET  010H          ;1: TX BUF ACTIVE, 0: TX BUF
                                        ;3-STATE
ISCTL               .SET  020H          ;INTERRUPT SOURCE BIT
IE1                 .SET  0H            ;INT ENABLE 1 REGISTER
                                        ;ADDRESS (SFR)
P0IE_1              .SET  08H           ;BIT IN INT ENABLE 1 REGISTER
                                        ;(SFR)
IFG1                .SET  02H           ;INTERRUPT FLAG REGISTER 1
TEMP                .SET  R4

;-----------------------------------------------------------------
; SUBROUTINE : INITIALIZE UART CONTROL REGISTERS
; (CALLED ONCE AFTER RESET)
;-----------------------------------------------------------------

INIT_RXTX
          MOV.B     #072H,&TCCTL      ;ACLK IS SOURCE FOR 8-BIT-
                                      ;TIMER/COUNT
          BIS.B     #P0IES_1,&P0IES   ;SELECT NEGATIVE EDGE FOR
                                      ;P0.1
                                      ;INTRPT
          BIC.B     #P0IFG_1,&IFG1    ;AND RXACT_FF, RESET INT.
                                      ;FLAG
          CLR.B     RXFLAGS           ; CLEAR RX/TX REGISTERS
          CLR.B     RXSTATUS          ; CLEAR RX STATUS REGISTER
          RET
```

```
;-----------------------------------------------------------------
; SUBROUTINE : PREPARE TRANSMIT CYCLE
;-----------------------------------------------------------------

PREP_TX
          CLR.B     TXSTATUS            ;INITIALIZE TRANSMIT STATUS
          MOV.B     #072H,&TCCTL        ;TXD = 1, TXE = 1

          MOV.B     #0F0H,TCPLD         ;LOAD TIME UNTIL START BIT
                                        ;STARTS
          MOV.B     #0,&TCDAT           ;DUMMY WRITE TO LOAD
                                        ;COUNTER/TIMER
          MOV.B     #BITIME1,TCPLD      ;LOAD PRELOAD REG. WITH
                                        ;BITTIME 1

          BIS.B     #ENCNT,&TCCTL       ;SET TRANSMIT START CONDITION
          JMP       PREP_RXTX           ;FALL INTO COMMON PART

;-----------------------------------------------------------------
; SUBROUTINE : PREPARE RECEIVE CYCLE
;-----------------------------------------------------------------

PREP_RX   CLR.B     RXSTATUS            ;INITIALIZE RECEIVE STATUS
          MOV.B     #072H,&TCCTL        ;SSEL1 = 0, SSEL0 = ISCTL = 1

          MOV.B     #BITIME1_2,&TCPLD   ;SET PRELOAD REGISTER WITH
                                        ;T1-2
          MOV.B     #0,&TCDAT           ;DUMMY WRITE TO LOAD COUNTER/
                                        ;TIMER
          MOV.B     #BITIME1,&TCPLD     ;LOAD PRELOAD REG. WITH
                                        ;BITTIME 1

          BIS.B     #RXACT,&TCCTL       ;ACTIVATE NEG. EDGE DETECT OF
                                        ;P0.1(RX)

PREP_RXTX           ;COMMON PART ALSO FOR
                                        ;PREP_TX
          .IF PARITY = "EVEN"
          BIC.B     #PAR_BIT,RXFLAGS    ;PRESET PAR_BIT = 0
          .ELSE
          BIS.B     #PAR_BIT,RXFLAGS    ;PRESET PAR_BIT = 1
          .ENDIF
          BIS.B     #P0IE_1,&IE1        ;ENABLE P0.1 / 8BIT COUNTER
                                        ;INTRPT
          RET                           ;ACCORDING TO STATE OF ISCTL
```

```
;*******************************************************************
; INTERRUPT HANDLER OF SOFTWARE UART
;*******************************************************************

INT_P0_1
          BIT       #RXACT,TCCTL          ;RX/TX INTRPT HANDLER ?
          JNZ       RXINTRPT              ;RECEIVE MODE IS ACTIVE ->
                                          ;JUMP

;-------------------------------------------------------------------
; TRANSMIT INTERRUPT HANDLER : DATA IS IN TXDATA
; INPUT : DATA TO TRANSMIT IN TXDATA
; OUTPUT: IF TRANSMIT IS COMPLETED, TXSTATUS WILL BE #FRAME_END
;-------------------------------------------------------------------

TXINTRPT
          PUSH      R5                    ;RXACT = 0 -->  TRANSMIT
          MOV.B     TXSTATUS,R5           ;USE TXSTATUS FOR
          BR        TXTAB(R5)             ;TRANSMIT PROCESS TABLE
TXTAB     .WORD     TXSTAT0               ;STARTBIT
          .WORD     TXSTAT2               ;BIT 0, LSB  BITIME2
          .WORD     TXSTAT1               ;BIT 1       BITIME1
          .WORD     TXSTAT1               ;BIT 2       BITIME1
          .WORD     TXSTAT2               ;BIT 3       BITIME2
          .WORD     TXSTAT1               ;BIT 4       BITIME1
          .WORD     TXSTAT1               ;BIT 5       BITIME1
          .WORD     TXSTAT2               ;BIT 6       BITIME2
          .IF DATABITS = 8
          .WORD     TXSTAT1               ;BIT 7       BITIME1
          .ENDIF
          .IF PARITY != "NONE"
          .WORD     TXPAR                 ;PARITY BIT  BITIME1
          .ENDIF
          .IF STOPBITS = 2
          .WORD     TXSTOP                ;STOPBIT     BITIME1
          .ENDIF
          .WORD     TXSTOP                ;STOPBIT     BITIME1
          .WORD     TXRET                 ;FRAME TRANSMITTED

TXSTAT0   MOV.B     #BITIME1,&TCPLD       ;LOAD BITTIME OF NEXT BIT
          JMP       TX_LO                 ;STARTBIT, OUT=LO
TXSTAT2   MOV.B     #BITIME2,&TCPLD       ;LOAD BITTIME OF NEXT BIT
          JMP       TX_BIT
TXSTAT1   MOV.B     #BITIME1,&TCPLD
TX_BIT    RRA       &TXDATA               ;LSB -> CARRY
          JNC       TX_LO
          XOR.B     #PAR_BIT,TXFLAGS      ;1 : TOGGLE PARITY
TX_HI     BIS.B     #TXD,&TCCTL           ;    OUT=HI
          JMP       TXRET
TX_LO     BIC.B     #TXD,&TCCTL           ;0 : OUT=LO
```

```
TXRET     INCD.B    &TXSTATUS          ;TXSTATUS + 2
          POP       R5
          RETI                         ;TRANSMISSION OF ONE BIT
                                       ;COMPLETED

;PARITY BIT CHECK: PAR_BIT + PARITY BIT MUST BE EVEN
TXPAR     MOV.B     #BITIME1,&TCPLD
          BIT.B     #PAR_BIT,TXFLAGS   ;CHECK PARITY BIT VALUE
          JNZ       TX_HI              ;PARITY BIT SHOULD BE MARK
          JMP       TX_LO              ;PARITY BIT SHOULD BE SPACE

;OUTPUT OF STOP BIT(S)
TXSTOP    MOV.B     #BITIME1,&TCPLD
          JMP       TX_HI              ;SEND STOP BIT 1 OR 2

;-------------------------------------------------------------------
; RECEIVE INTERRUPT HANDLER
; OUTPUT: RECEIVED DATA IN RXDATA
; RECEIVE FINISHED: #REC_BIT IS SET IN STATUSBYTE RXFLAGS
;-------------------------------------------------------------------

RXINTRPT
          PUSH      R5                 ;RECEIVER INTERRUPT ROUTINE
          MOV.B     &RXSTATUS,R5       ;R5 IS USED TEMPORARY AS
                                       ;POINTER OF
          BR        RCTAB(R5)          ;RECEIVE PROCESS TABLE

RCTAB     .WORD     RCSTAT0            ;START BIT
          .WORD     RCSTAT1            ;BIT 0
          .WORD     RCSTAT1            ;BIT 1
          .WORD     RCSTAT2            ;BIT 2
          .WORD     RCSTAT1            ;BIT 3
          .WORD     RCSTAT1            ;BIT 4
          .WORD     RCSTAT2            ;BIT 5
          .WORD     RCSTAT1            ;BIT 6
          .IF DATABITS = 8
          .WORD     RCSTAT1            ;BIT 7
          .ENDIF
          .IF PARITY != "NONE"
          .WORD     RCSTAT2            ;PARITY BIT     BIT 1
          .ENDIF
          .IF STOPBITS = 2
          .WORD     RCSTOP1            ;STOP BIT       BIT 2
          .ENDIF
          .WORD     RCSTOP2            ;STOP BIT       BIT 2
```

```
RCSTAT0   BIT.B     #RXD,&TCCTL         ;CHECK START BIT
          JC        RCERROR             ;ERROR: START BIT IS MARK NOT
                                        ;SPACE
          MOV.B     #BITIME2,&TCPLD     ;START BIT FINE, LOAD PRE-
                                        ;LOAD REG.
          JMP       RCRET0
RCSTAT2   MOV.B     #BITIME2,&TCPLD     ;LOAD PRELOAD REG. WITH BIT
                                        ;TIME 2
          JMP       RCBIT
RCSTAT1   MOV.B     #BITIME1,&TCPLD     ;LOAD PRELOAD REG. WITH BIT
                                        ;TIME 1
RCBIT     BIT.B     #RXD,&TCCTL         ;RXD BIT->CARRY BIT
          JNC       RCRET               ;RXD BIT=CARRY BIT=0 ? YES,
                                        ;JUMP
          RRC       &RXDATA             ;RXD BIT -> MSB, NEGATIVE BIT
          XOR.B     #PAR_BIT,RXFLAGS;  RXD BIT = 1, TOGGLE PAR_BIT
          JMP       RCRET0
RCRET     RRC       &RXDATA             ;RXD=0, RXD BIT -> MSB,
                                        ;NEGATIVE BIT
RCRET0    INCD.B    &RXSTATUS
RCCMPL    POP       R5
          RETI
; PARITY BIT WAS RECEIVED JUST LIKE ALL OTHER BITS

RCSTOP1   MOV.B     #BITIME1,&TCPLD     ;LOAD PRELOAD REG. WITH BIT
                                        ;TIME 1
          BIT.B     #RXD,&TCCTL         ;CHECK STOP BIT FOR MARK
          JZ        RCERROR             ;STOP BIT IS MARK -> OK
          JMP       RCRET0

RCSTOP2
          .IF PARITY != "NONE"
          BIT.B     #PAR_BIT,RXFLAGS    ;CHECK PARITY BIT. BIT MUST
                                        ;BE ZERO
          JNZ       RCERROR             ;PARITY BIT FALSE.
          .ENDIF
          BIT.B     #RXD,&TCCTL         ;CHECK STOP BIT FOR MARK
          JZ        RCERROR             ;STOP BIT IS MARK -> OK
          POP       R5

; PREPARE CHARACTER RECEIVED AND STORE IT

          .IF PARITY != "NONE"
          RLA       RXDATA              ;SHIFT PARITY OUT
          .ENDIF
          .IF DATABITS = 7
          CLRC                          ;CLEAR CARRY
          RRC       RXDATA              ;SHIFT 0 INTO BIT 7
          .ENDIF
          MOV.B     &RXDATA+1,&RXBUF    ;STORE RECEIVED CHARACTER
```

```
          BIS.B     #REC_BIT,RXFLAGS    ;SET 'RECEIVED' BIT = 1
          CALL      #PREP_RX            ;PREPARE NEXT FRAME
          RETI

; ERROR HANDLING: A NEW START IS TRIED
RCERROR   BIS.B     #ERR_BIT,RXFLAGS
          CALL      #PREP_RX
          JMP       RCCMPL

; INTERRUPT VECTOR ADDRESS
          .SECT     "P0_1VECT",0FFF8H   ;ADDRESS FOR TC8 INTERRUPT
          .WORD     INT_P0_1
          .SECT     "RST_VECT",0FFFEH   ;PUC / RESET ADDRESS
          .WORD     START
```

The following subroutines shows the capability of the software UART described above. First, the timer must be initialized by calling the subroutine INIT_RXTX.

The direction of the communication has to be selected by calling the subroutines PREP_PX respectively PREP_TX. This subroutine call is performed in the following example.

RX_CHAR initiates the timer to receive one byte, that will be stored in the RAMbyte RXBUF after the complete receive cycle. If an error occurs during the serial communication, the error bit in the RXFLAGS byte will be set.

```
;---------------------------------------------------------------
; SUBROUTINE : RECEIVE 1 CHARACTER INTO RXBUF
; OUTPUT : RECEIVED CHARACTER IN RXBUF
;---------------------------------------------------------------

RX_CHAR   BIC.B     #ERR_BIT,RXFLAGS      ;CLEAR ERROR BIT FROM
PREVIOUS CALL
          BIT.B     #REC_BIT,RXFLAGS      ;TEST 'RECEIVED' FLAG
          JZ        RX_CHAR               ;0 : WAIT FOR CHAR.
          BIC.B     #REC_BIT,RXFLAGS      ;1 : CLEAR FLAG AND EXIT
          RET                             ;RXBUF HOLDS DATA
```

The next example called TX_SPACE transmits one space character, that is located in the RAM byte TXDATA. Every other character can be transmitted, of course. Saving code space, this routine uses the part TX-END, that is used by the subroutined TX_TABLE, too. The routine TX_TABLE transmits a whole string, which is pointed to by TEMP. The end of the string is indicated by the value 00.

```
;-------------------------------------------------------------------
; SUBROUTINE : TRANSMIT 1 CHARACTER FROM TXDATA
;-------------------------------------------------------------------

TX_SPACE
          MOV       #' ',TXDATA         ;TRANSMIT <SPACE>
TX_CHAR
          CALL      #PREP_TX            ;INITIALIZE TRANSMISSION

TX_CHAR1
          CMP.B     #FRAME_END,TXSTATUS
                                        ;OUTPUT OF ONE FRAME
                                        ;COMPLETED?
          JNE       TX_CHAR1            ;NO : WAIT FOR COMPLETION
          JMP       TX_END

;-------------------------------------------------------------------
;SUBROUTINE : TRANSMIT DATA FROM TEXT TABLE, STARTADDRESS IN TEMP
;-------------------------------------------------------------------

TX_TABLE
          CALL      #PREP_TX            ;INITIALIZE TRANSMISSION
TX_TABLE1
          MOV.B     @TEMP+,TXDATA       ;CHAR TO SEND TO TXDATA
          TST.B     TXDATA              ;ALL CHARS TRANSMITTED?
                                        ;(CHAR = 0)
          JZ        TX_END              ;YES, STOP TX AND CONTINUE
                                        ;PROGRAM
          CLR.B     TXSTATUS            ;CLEAR TRANSMIT STATUS

          .IF PARITY = "EVEN"
          BIC.B     #PAR_BIT,TXFLAGS    ;PRESET PAR_BIT = 0
          .ELSE
          BIS.B     #PAR_BIT,TXFLAGS    ;PRESET PAR_BIT = 1
          .ENDIF

TX_TABLE2
          CMP.B     #FRAME_END,TXSTATUS
                                        ;OUTPUT OF ONE FRAME
                                        ;COMPLETED?
          JEQ       TX_TABLE1           ;YES, TRANSMIT NEXT DATA OF
                                        ;TABLE!
          JMP       TX_TABLE2           ;NO, WAIT FOR COMPLETION

; ----------- OUTPUT OF ONE STRING COMPLETED --------------------
TX_END    CMP.B     #FRAME_END+2,TXSTATUS
          JNE       TX_END              ;WAIT FOR OUTPUT OF LAST
                                        ;STOPBIT
```

```
          BIC.B     #P0IE_1,&IE1        ;INTERPT DISABLED FOR P0.1/TC
                                        ;IN SFR
          BIC.B     #ENCNT,&TCCTL       ;STOP COUNTER TO SAVE POWER
                                        ;CONS.
          RET                           ;TRANSMISSION OF TABLE IS
                                        ;COMPLETED


;EXAMPLE FOR TRANSMITTING THE TEXT "TONI"
STRING    .BYTE     "TONI",0            ;0 IS THE INDICATION OF THE
                                        ;END OF
                                        ;THE STRING
TEST      MOV       STRING,TEMP
          CALL      #TX_TABLE
```

7.3 8 bit PWM Timer

This module is integrated in the EVE_OPT-Version of the MSP430 family, and generates a rectangular output pulse with a duty factor of 0% to 100 %. The period of the PWM signal can be selected from 242 µs up to 992,2 ms, as shown in the table below. The resolution of the duty factor is 1/254 .

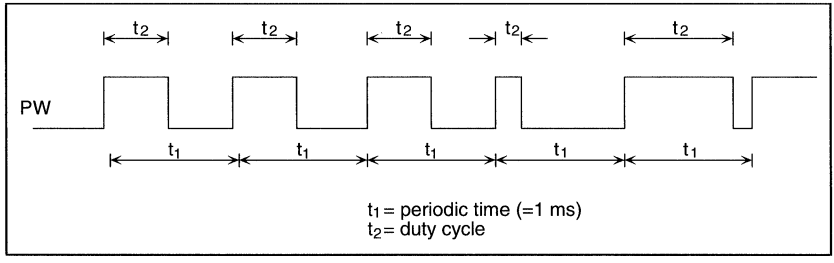


Figure 7.4: PWM Modulation

Clock Source	Period Time t1	SSEL2	SSEL1	SSEL0
MCLK	$1/2^{20} \cdot 254 = 242\ \mu s$	0	0	0
MCLK/4	$4/2^{20} \cdot 254 = 969\ \mu s$	0	0	1
MCLK/16	$16/2^{20} \cdot 254 = 3{,}88\ ms$	0	1	0
ACLK	$1/2^{15} \cdot 254 = 7{,}75\ ms$	0	1	1
ACLK/4	$4/2^{15} \cdot 254 = 31{,}01\ ms$	1	0	0
ACLK/8	$8/2^{15} \cdot 254 = 62{,}01\ ms$	1	0	1
ACLK/16	$16/2^{15} \cdot 254 = 124{,}02\ ms$	1	1	0
ACLK/128	$128/2^{15} \cdot 254 = 992{,}2\ ms$	1	1	1

$$t2 = t1 \cdot \frac{PWMDT}{254} \qquad ; PWMDT \in [0,FEh]$$

The following software example shows the capability of this peripheral module. After configuration of the PWM registers, no software control is necessary to generate the output pulses, unless the duty cycle or the PWM period have to be modified. These modifications will affect the output after the end of the actual PWM period.

```
; INITIALIZATION
          CLR.B     &PWMCTL          ;T1=242 US
                                     ;PWM OUTPUT IS DISABLED
                                     ;POSITIVE LOGIC
          MOV.B     #7FH,&PWMCTL     ;DUTY CYCLE = 50 %
.....

; CHANGE DUTY CYCLE T2
          MOV.B     #3FH,&PWMDT      ;DUTY CYCLE = 25 %
.....

; CHANGE PWM PERIOD T1
          BIC.B     #SSEL0+SSEL1+SSEL2, &PWMCTL
                                     ;CLEAR ALL CLOCK SOURCES
          BIS.B     #SSEL0+SSEL1,&PWMCTL
                                     ;NEW CLOCK SOURCE IS ACLK,
                                     ;T1=7,75MS
.....
```

```
;STOP PWM SIGNAL AT LOW STATE
LOOP      BIT.B     #CMPM,&PWMCTL    ;WAIT FOR PWM OUTPUT = LOW
          JZ        LOOP
          BIC.B     #0EH,&PWMCTL     ;STOP PWM SIGNAL
          CLR.B     &PWMCNT          ;RESET COUNTER
.....

          BIS.B     #0EH,&PWMCTL     ;CONTINUE WITH PWM
```

7.4 Universal Timer / Port Module

The Universal Timer / Port Module is implemented in the EVE-Opt version of the MSP430 family and contains up to six independent outputs (TP.0 .. TP.5), two 8 bit counters which are cascadeable for 16 bit mode, and a comparator for A/D-conversion of the slope-converter type. The use of this module as an A/D-Converter is described in the section *The Analog to Digital Converters*.

If no A/D-conversion is needed, this module can be configured as an Universal Timer / Counter with interrupt capability.

The following example shows the use of this module in the timer mode. The Register Equate table for the software examples is shown below.

```
;UNIVERSAL TIMER / PORT REGISTER DEFINITIONS
TPCTL     .EQU      4BH
TPCNT1    .EQU      4CH
TPCNT2    .EQU      4DH
TPD       .EQU      4EH
TPE       .EQU      4FH

;TPCTL BIT DEFINITIONS
EN1FG     .EQU      1H
RC1FG     .EQU      2H
RC2FG     .EQU      4H
EN1       .EQU      8H
ENA       .EQU      10H
ENB       .EQU      20H
TPSSEL0   .EQU      40H
TPSSEL1   .EQU      80H

;TPD BIT DEFINITIONS
CPON      .EQU      40H
TP16B     .EQU      80H

;TPE BIT DEFINITIONS
TPSSEL2   .EQU      40H
TPSSEL3   .EQU      80H
```

```
IE1       .EQU      00H
IE2       .EQU      01H
IFG1      .EQU      02H
IFG2      .EQU      03H

TPIE      .EQU      4H
```

7.4.1 Initialization

The initalization for using the two 8 bit counters in 16 bit mode is shown in the following example. Every other configuration can be achieved easily by modifying the appropriate bits in the configuration registers described in the MSP430 Architecture Guide.

```
;INITIALIZATION OF THE TIMER / PORT REGISTERS
          MOV.B     #TPSSEL0,&TPCTL
                                    ;CLEAR EN1 FLAG
                                    ;CLEAR RC1 AND RC2 FLAG
                                    ;EN1=0 : DISABLES COUNTER
                                    ;CLOCK SOURCE IS ACLK

          CLR.B     &TPCNT1         ;CLEAR COUNTER 1
          CLR.B     &TPCNT2         ;CLEAR COUNTER 2

          MOV.B     #TP16B,&TPD     ;16 BIT COUNTER MODE
                                    ;TPCNT1 = LOW BYTE
                                    ;TPCNT2 = HIGH BYTE
                                    ;RESETS OUTPUTS TP.0 TO TP.5

          CLR.B     &TPE            ;T0.0 TO TP.5 ARE SET TO 3-STATE
                                    ;CLOCK SOURCE = CLK1 = ACLK

          MOV.B     #0AAH,&TPCNT1   ;LOAD LOW BYTE WITH APPROPRIATE
                                    ;VALUE
          MOV.B     #0BBH,&TPCNT2   ;LOAD HIGH BYTE WITH APPROPRIATE
                                    ;VALUE
          BIS.B     #TPIE,&IE2      ;ENABLE TIMER/PORT INTERRUPT
          GIE                       ;ENABLE ALL SELECTED INTERRUPTS
.....

;*************************************************************
; TIMER / PORT COUNTER INTERRUPT SERVICE ROUTINE
;FOLLOWING INTERRUPT SERVICE ROUTINE CAN BE USED FOR EVERY MODE
;OF
;THIS MODULE. IN THE 8 BIT MODE THREE DIFFERENT SOURCES CAN CAUSE
;INTERRUPTS CAN OCCUR
;         - NEGATIVE EDGE OF EN1
;         - OVERFLOW FROM TPCNT1 ( RC1 SIGNAL )
;         - OVERFLOW FROM TPCNT2 ( RC2 SIGNAL )
```

```
;IN THE 16 BIT MODE OF THE COUNTER TWO DIFFERENT SOURCES CAN
;CAUSE
;INTERRUPTS:
;         - NEGATIVE EDGE OF EN1
;         - OVERFLOW FROM TPCNT2
;*****************************************************************

TP_ISR    BIT.B     #EN1FG,&TPCTL   ;NEG.EDGE ON EN1 CAUSES INTERRUPT
?
          JNZ       EN1_ISR         ;YES, EXECUTE CORRESPONDING ISR
          BIT.B     #RC1FG,&TPCTL   ;RIPPLE CARRY OF COUNTER1 CAUSES
                                    ;INT ?
          JNZ       TPA8_ISR        ;YES, EXECUTE CORRESPONDIG ISR
          BIT.B     #TP16B,&TPD     ;IS 16 BIT MODE SELECTED ?
          JNZ       TP16_ISR        ;YES, EXECUTE 16 BIT ISR

          BIC.B     #RC2FG,&TPCTL   ;8BIT MODE, ISR FOR COUNTER 2
          .....
          RETI

TP16_ISR            ;16 BIT MODE
          BIC.B     #RC2FG,&TPCTL
          .....
          RETI

TPA8_ISR            ;8 BIT MODE, COUNTER 1
          BIC.B     #RC1FG,&TPCTL
          .....
          RETI

EN1_ISR                             ;EN1 ISR
          BIC.B     #EN1FG,&TPCTL
          .....
          RETI

;INTERRUPT VECTOR ADDRESSES FOR THE UNIVERSAL COUNTER

          .SECT     "TPVECT",0FFEAH ;UNIVERSAL COUNTER
          .WORD     TP_ISR
```

7.4.2 Measuring the Revolutions of a Toothed Wheel

The Figure below shows the set-up for determining the number of revolutions and the angular resolution of the number of the tooths of the wheel. If the toothed wheel has 72 teeth, the resolution is in steps of 5 degrees.

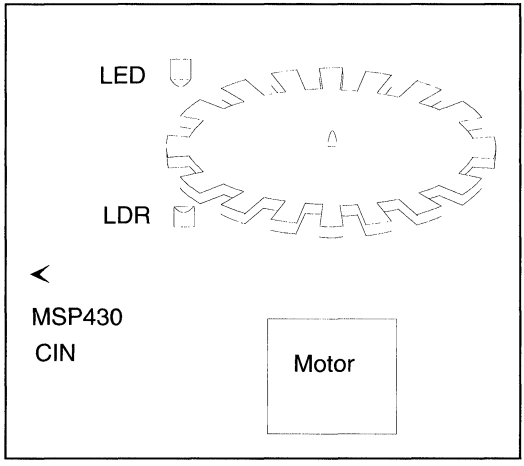


Figure 7.5: Measuring the Revolutions of a Toothed-Wheel

When the light from the LED impinges on the LDR, the pulse which occurs increments the 8 bit counter by 1. If one revolution of the disk is performed, the counter TPCNT1 will generate an overflow, and an interrupt occurs. The corresponding ISR increments the revolution register; the actual angle is contained in the PTCNT1 register.

```
NUT       .EQU     72                ;NUMBER OF TOOTHS
REV       .EQU     R10               ;CONTAINING THE NUMBER OF
                                     ;REVOLUTIONS


;INITIALIZATION OF THE UNIVERSAL COUNTER

          CLR.B    &TPCTL            ;EN1=0
                                     ;CLK1 = CIN
                                     ;INTERRUPT FLAGS = 0

          MOV.B    #0-NUT,&TPCNT1    ;GENERATE OVERFLOW, IF ONE
                                     ;REVOLUTION IS OVER
                                     ;IS OVER

          CLR.B    &TPD              ;SET OUTPUTS TP.0 TO TP.5 TO
                                     ;LOW
                                     ;8 BIT COUTER MODE
                                     ;COMPARATOR IS SWITCHED OFF

          CLR.B    &IFG1             ;CLEAR INTERRUPT REQUEST
                                     ;FLAG 1
          CLR.B    &IFG2             ;CLEAR INTERRUPT REQUEST
                                     ;FLAG 2
          BIS.B    #TPIE,&IE2        ;ENABLE UNIVERSAL COUNTER
                                     ;INTERRUPT
          BIS.B    #ENA,&TPCTL       ;START COUNTER


;INTERRUPT SERVICE ROUTINE

TP_ISR    BIC.B    #RC1FG,&TPCTL     ;CLEAR INTERRUPT REQUEST FLAG
          INC      REV               ;ONE MORE REVOLUTION
          MOV.B    #0-NUT,&TPCNT1    ;RELOAD COUNTER WITH
APPROPRIATE
                                     ;VALUE
          RETI


;INTERRUPT VECTOR ADDRESSES FOR THE UNIVERSAL COUNTER

          .SECT    "TPVECT",0FFEAH;UNIVERSAL COUNTER
          .WORD    TP_ISR
```

Topics

Figures

Notes

Tables

8 LCD Display

In many applications the result of an operation must be visually displayed. For this purpose, the integrated LCD Driver can be used. In the 4 MUX mode, up to 8 user-defined characters can be displayed at once.

8.1 Initialization

First, the right display mode has to be selected. In the following example the 4MUX mode is selected, because this mode allows displaying up to 8 digits with only 20 lines (4xCOM + 16xSelect). The correct timing for the selected mode is generated by the basic timer, which has to be initialized accordingly. The number of digits on the LCD used can be defined by LCD_DIG.

```
;*************************************************************
;BASIC-TIMER DEFINITIONS
;*************************************************************
BTCTL     .SET      040H            ;BASIC TIMER CONTROL REGISTER


;*************************************************************
;LCD DRIVER DEFINITIONS FOR ALL MUX MODES
;*************************************************************
LCD0      .SET      00030H          ;ADDRESS OF LCD CONTROL
LCDM      .SET      0031H           ;START OF LCD DIGIT MEMORY
LCD_DIG   .SET      8               ;LCD WITH 8 DIGITS


;*************************************************************
;PREPARE LCD AND BASIC TIMER FOR 4 MUX MODE
;*************************************************************
          MOV.B     #-1H,&LCD0      ;SELECTED FUNCT. ANALOG
                                    ;GENERATOR ON
                                    ;LOW IMPEDANCE OF AG
                                    ;4MUX ACTIVE
                                    ;ALL OUTPUTS ARE SEG

          MOV.B    #077H,&BTCTL     ;SELECTED FUNCTION BASIC TIMER:
                                    ;ACLK
                                    ;RESET
                                    ;HIGHEST DIVISION FACTOR
                                    ;LCD FRAME FREQUENCY @4MUX: 64HZ
          BIS.B     #80H,&ME2       ;ENABLE BASIC TIMER MODULE
          BIC.B     #040H,&BTCTL    ;BASIC TIMER RESET DISABLED
```

8.2 Definition of the Characters

To define the characters, the allocation of the segments is assumed to be as shown in Figure 8.1. In the first table of the following software routine, the 8 segments (a to h) are defined. In the LCDTAB the characters which can be built with the segments are arranged with the ASCII code. Therefore, it is possible to get the segment code in a simple manner.

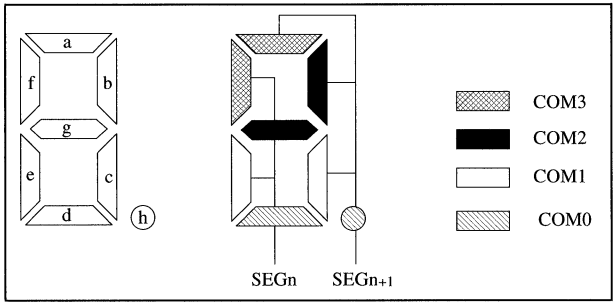


Figure 8.1: Allocation of the Segments, 4 MUX Mode

The allocation of the segments depens on the model of the used LC Display. Figure 8.1 shows only a possible allocation. if a different display is used, the proper segment allocation has to be defined by the tables shown in the MSP430 Family Architecture Guide. For the 4MUX Mode LCD's the table of the segment allocation is as follows.

bit position	7	6	5	4	3	2	1	0
3Fh	-				-			
...	-				-			
32h	-				-	Y	h	g
31h	-	f	e	d	-	c	b	a
	COM3	COM2	COM1	COM0	COM3	COM2	COM1	COM0
	select n+1				select n			

Table 8.1: Allocation of the LCD Digits in 4MUX Mode

For example the a-segment can be selected by COM3 and odd select (select n+1), the appropriate equation is:

```
A       .EQU    80H             ;= 1000 0000 B
                                ; BIT 7 MEANS COM3 AND SELECT N+1
```

If this pattern is written in the RAM Memory, the a-segment will shine. Writing 0FFh into the LCD RAM byte, all segments will shine. If the segments are defined correctly by the 8 equations, all of the described subroutines will work.

```
A           .EQU      80H             ;DEFINITION OF THE 7 SEGMENTS
B           .EQU      40H             ;BY THE USER.
C           .EQU      20H             ;DEPENDS ON THE USED LC DISPLAY
D           .EQU      01H             ;THIS EXAMPLE IS IN RELATION TO
E           .EQU      02H             ;THE LCD CONFIGURATION SHOWN IN
F           .EQU      08H             ;FIGURE 8.1
G           .EQU      04H
H           .EQU      10H

;THE FOLLOWING ASCII TABLE CAN BE USED FOR ALL MUX MODES
LCDTAB      .BYTE     A+B+C+D+E+F     ;"0"
            .BYTE     B+C             ;"1"
            .BYTE     A+B+G+E+D       ;"2"
            .BYTE     A+B+G+C+D       ;"3"
            .BYTE     F+G+B+C         ;"4"
            .BYTE     A+F+G+C+D       ;"5"
            .BYTE     A+F+E+D+C+G     ;"6"
            .BYTE     A+B+C           ;"7"
            .BYTE     A+B+C+D+E+F+G   ;"8"
            .BYTE     A+B+C+D+G+F     ;"9"
            .BYTE     0               ;":"
            .BYTE     0               ;";"
            .BYTE     0               ;"<"
            .BYTE     G               ;"="
            .BYTE     0               ;">"
            .BYTE     0               ;"?"
            .BYTE     0               ;"@"
            .BYTE     E+F+A+B+C+G     ;"A"
            .BYTE     F+E+D+C+G       ;"B"
            .BYTE     G+E+D           ;"C"
            .BYTE     G+E+D+C+B       ;"D"
            .BYTE     A+F+E+D+G       ;"E"
            .BYTE     A+F+E+G         ;"F"
            .BYTE     A+F+E+D+C+G     ;"G"
            .BYTE     F+E+B+C+G       ;"H"
            .BYTE     B+C             ;"I"
            .BYTE     B+C+D           ;"J"
            .BYTE     0               ;"K"
            .BYTE     F+E+D           ;"L"
            .BYTE     E+F+A+B+C       ;"M"
            .BYTE     0               ;"N"
            .BYTE     E+D+C+G         ;"O"
            .BYTE     F+E+A+B+G       ;"P"
            .BYTE     0               ;"Q"
            .BYTE     E+G             ;"R"
            .BYTE     A+F+G+C+D       ;"S"
```

```
            .BYTE     F+E+D+G           ;"T"
            .BYTE     F+E+D+C+B         ;"U"
            .BYTE     0                 ;"V"
            .BYTE     0                 ;"W"
            .BYTE     0                 ;"X"
            .BYTE     F+G+B+C+D         ;"Y"
            .BYTE     0                 ;"Z"
T_LCD       .BYTE     "LCDTEST",255     ;TESTSTRING WITH END OF
                                        ;TEXT = 255
```

8.3 Display Text

The following subroutines describe how easy it is to handle the display driver. The text which is to be displayed is simply moved into the LCD RAM. The allocation of the LCD RAM is shown in Figure 8.2.

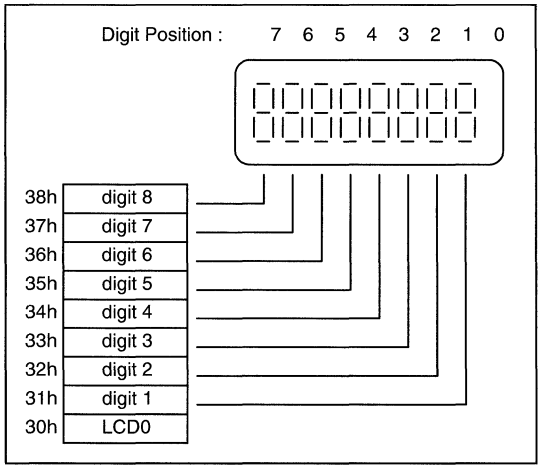


Figure 8.2: Allocation of the LCD Digits in the Memory

```
;*************************************************************
;LCD DISPLAY CHARACTER USING 4 MUX MODE
;THE LSDIGIT OF REGISTER R12(000M) IS DISPLAYED ON
;DIGIT R13(0..LCD_DIG)
;*************************************************************

DSP_CHR
          SUB       #030H,R12       ;R12 IS ASCII FORMAT
          CMP       #42,R12         ;ABOVE TABLE
          JLO       DSP_MUX4        ;NO
          MOV       #11,R12         ;YES PRINT SPACE

DSP_MUX4
          MOV.B     LCDTAB(R12),LCDM(R13)
          RET

;*************************************************************
;DSP_TXT, CAN BE USED FOR ALL MUX MODES
;SHOWS THE TEXT, WHICH IS POINTED TO BY R10 ON THE BEGINING OF
;THE DISPLAY
;EXAMPLE :
;         MOV       #T_LCD,R10
;         CALL      #DSP_TXT
;*************************************************************

DSP_TXT
          PUSH      R13
          MOV       #LCD_DIG-1,R13  ;R13 IS MAX LCD POSITION

DSP_L$1 MOV.B       @R10+,R12       ;CHAR TO REGISTER
          CMP.B     #0FFH,R12       ;END OF TEXT
          JZ        DSP_L$2
          CALL      #DSP_CHR        ;ONE CHARACTER TO THE LCD
          DEC       R13             ;NEW POSITION OF THE POINTER
          JHS       DSP_L$1         ;NEXT CHARACTER

DSP_L$2   POP       R13
          RET



;*************************************************************
;CLRSCR : WRITE BLANKS TO THE LCD BY CLEARING THE LCD MEMORY
;*************************************************************

CLRSCR    PUSH      R5
          MOV       #LCD_DIG,R10    ;NUMER OF LCD DIGITS TO R10

CLR_1     CLR.B     LCDM(R10)       ;CLEAR ONE DIGIT
          DEC       R10             ;NEXT LCD POSITION
```

```
          JNZ       CLR_1           ;ALL DIGITS CLEARED ?
          POP       R5              ;YES
          RET


;*************************************************************
;BINTOLCD : PUT THE INTEGER IN R10 ON THE LCD BY USING DSP_CHR
;R13 MUST CONTAIN THE POSITION ON THE LCD(4..7)
;*************************************************************

BINTOLCD
          PUSH      R12             ;SAVE USED REGISTERS
          PUSH      R10
          PUSH      R4
          MOV       #4,R4           ;COUNTER OF DIGITS
          SUB       #3,R13          ;FIRST MEMORY POSITION

BINL$1    MOV       R10,R12         ;STORE VALUE
          BIC       #0FFF0H,R12     ;ONLY LAST DIGIT
          CMP       #10,R12         ;VALUE ABOVE 10 (A) ?
          JLO       BINL$2          ;NO
          ADD       #7,R12          ;YES, SELECT A..F

BINL$2    ADD       #30H,R12        ;ADJUST TO ASCII CODE
          CALL      #DSP_CHR        ;ONE CHARACTER TO LCD
          INC       R13             ;NEW POSITION ON LCD
          RRA       R10             ;NEXT DIGIT INTO LAST POSITION
          RRA       R10             ;OF REGISTER
          RRA       R10
          RRA       R10
          DEC       R4              ;ALL DIGITS DISPLAYED ?
          JNZ       BINL$1          ;NO
          POP       R4              ;YES, RESTORE USED REGISTERS
          POP       R10
          POP       R12
          RET
```

8.4 Adaption to other MUX Modes

The routines described above can also be used for Displays using other Modes (3 MUX, 2 MUX or static). For these purposes the initialization of the LCD Mode Register has to be modified as follows:

MUX Mode	Mode Register		
	LCDM 4	LCDM 3	LCDM 2
MUX 4	1	1	1
MUX 3	1	0	1
MUX 2	0	1	1
Static	0	0	1

Table 8.2:Configuration of the MUX Modes

Additionally, the equations for the segments have to be adjusted.
Furthermore, the software routines for displaying one character on the dedicated LCD position (DSP_CHR) have to be substituted.

The DSP_CHR routine for the 3 MUX Mode is the most complicated, because the determination of the LCD position in the LCD memory needs a lot of code.

8.4.1 Adaption to 3MUX Mode

The DSP_CHR routine for the 3 MUX Mode is the most complicated, because the determination of the LCD position in the LCD memory needs a lot of code.
The eight segments of the digits are located in 1½ display memory bytes. In the 3MUX Mode an additional segment Y can be selected.

Example for a layout of a 3MUX driven LCD digit:

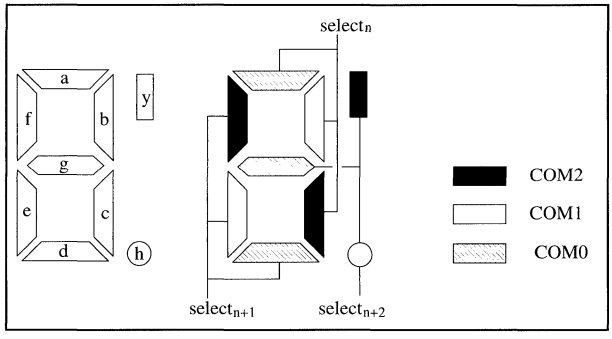


Figure 8.3: Allocation of the Segments, 3 MUX Mode

Using a LCD with the segment allocation as shown above, the corresponding RAM memory should be as follows:

bit position	7	6	5	4	3	2	1	0
3Fh	-				-			
...	-				-			
32h	-				-	Y	h	g
31h	-	f	e	d	-	c	b	a
		COM2	COM1	COM0		COM2	COM1	COM0
		select n+1 (odd)				select n (even)		

Table 8.3: Allocation of the LCD Digits in 3MUX Mode

The following equations shows the allocation between segments and memory location.

```
a     .equ    001h
b     .equ    002h
c     .equ    004h
d     .equ    010h
e     .equ    020h
f     .equ    040h
g     .equ    100h
h     .equ    200h
y     .equ    400h
```

```
;*******************************************************
;LCD DISPLAY CHARACTER USING 3 MUX MODE
;THE LSDIGIT OF REGISTER R12(000M) IS DISPLAYED ON
;DIGIT R13(0..LCD_DIG)
;*******************************************************

DSP_CHR
                                    ;FIRST THE POSITON OF THE DIGIT IN
                                    ;THE LCD MEMORY MUST BE LOCATED

          BIT.B     #8,R13          ;IS LCD POSITION 6 OR 7 ?
          JZ        NEXT1           ;NO
          ADD       #3,R13          ;YES, ADJUST MEMORY POSITION
          JMP       DSP_STRT        ;WRITE CHARACTER INTO MEMORY
NEXT1     BIT.B     #4,R13          ;IS LCD POSITION 4 OR 5 ?
          JZ        NEXT2           ;NO
          ADD       #2,R13          ;YES, ADJUST MEMORY POSITON
          JMP       DSP_STRT        ;WRITE CHARACTER INTO MEMORY
NEXT2     BIT.B     #2,R13          ;IS LCD POSITION 2 OR 3 ?
          JZ        DSP_STRT        ;NO, LCD POSITION IS 0 OR 1
          INC       R13             ;YES, ADJUST MEMORY POSITION
```

```
DSP_STRT
          SUB       #030H,R12       ;R12 IS ASCII
          CMP       #42,R12         ;ABOVE TABLE
          JLO       DSP_MUX3        ;NO
          MOV       #11,R12         ;YES PRINT SPACE
DSP_MUX3
          MOV.B     LCDTAB(R12),R4
          BIT.B     #1,R13          ;POSITION ODD OR EVEN ?
          JNZ       DIG_1
DIG_0     MOV.B     R4,LCDM(R13)
          SWPB      R4
          BIC.B     #07H,LCDM+1(R13)
          BIS.B     R4,LCDM+1(R13)
          RET
DIG_1     RLA       R4
          RLA       R4
          RLA       R4
          RLA       R4
          BIC.B     #07H,LCDM(R13)
          BIS.B     #R4,LCDM(R13)
          SWPB      R4
          MOV.B     R4,LCDM+1(R13)
          RET
```

8.4.2 Adaption to 2MUX Mode

The DSP_CHR routines for the 2 Mux and the static Mode are simpler to implement than the 3 MUX Mode, because of the easy to determine coherence between the LCD position and the corresponding memory position.

The eight segments of one digit are located in 2 bytes of the display memory.

Example of a possible layout of the segjments of a 2 MUX driven LCD

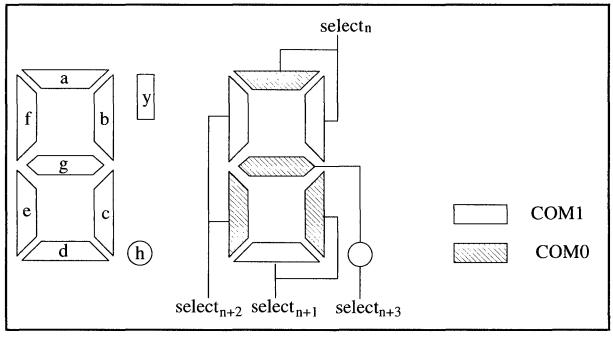


Figure 8.4: Allocation of the Segments, 2 MUX Mode

The corresponding RAM memory should be as follows:

bit position	7	6	5	4	3	2	1	0
3Fh	-	-			-	-		
...	-	-			-	-		
32h	-	-	h	g	-	-	f	e
31h	-	-	d	c	-	-	b	a
			COM1	COM0			COM1	COM0
	select n+1 (odd)				select n (even)			

Table 8.4: Allocation of the LCD Digits in 2MUX Mode

```
a   .equ    001h
b   .equ    002h
c   .equ    010h
d   .equ    020h
e   .equ    004h
f   .equ    008h
g   .equ    040h
h   .equ    080h
```

```
;*******************************************************
;LCD DISPLAY CHARACTER USING 2 MUX MODE
;THE LSDIGIT OF REGISTER R12(000M) IS DISPLAYED ON
;DIGIT R13(0..LCD_DIG)
;*******************************************************
DSP_CHR
          SUB       #030H,R12       ;R12 IS ASCII
          CMP       #42,R12         ;ABOVE TABLE
          JLO       DSP_MUX2        ;NO
          MOV       #11,R12         ;YES PRINT SPACE
DSP_MUX2
          RRA       R13
          MOV.B     LCDTAB(R12),R4
          MOV.B     R4,LCDM(R13)
          RRA       R4
          RRA       R4
          MOV.B     R4,LCDM+1(R13)
          RET
```

8.4.3 Adaption to Static Mode

The eight segments of one digit are located in four display memory bytes.

Example for a layout of a static driven LCD digit:

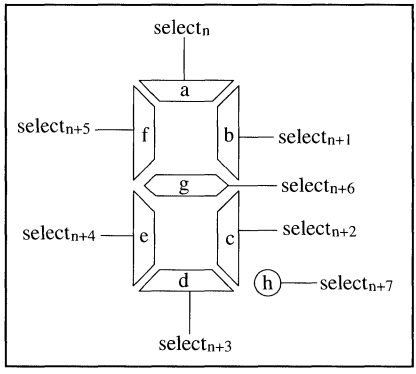


Figure 8.5: Allocation of the Segments, Static Mode

The corresponding display-RAM is shown below:

bit position	7	6	5	4	3	2	1	0
3Fh	-	-	-		-	-	-	
	-	-	-	h	-	-	-	g
33h	-	-	-	f	-	-	-	e
32h	-	-	-	d	-	-	-	c
31h	-	-	-	b	-	-	-	a
		COM2	COM1	COM0		COM2	COM1	COM0
		select n+1 (odd)				select n (even)		

Table 8.5: Allocation of the LCD Digits in Static Mode

If the static driven LCD is connected as shown above, following equation are valid:

```
a    .equ   001h
b    .equ   010h
c    .equ   002h
d    .equ   020h
e    .equ   004h
f    .equ   040h
g    .equ   008h
h    .equ   080h
```

```
;*******************************************************
;LCD DISPLAY CHARACTER USING STATIC MODE
;THE LSDIGIT OF REGISTER R12(000M) IS DISPLAYED ON
;DIGIT R13(0..LCD_DIG)
;*******************************************************
DSP_CHR
          SUB       #030H,R12       ;R12 IS ASCII
          CMP       #42,R12         ;ABOVE TABLE
          JLO       DSP_MUX1        ;NO
          MOV       #11,R12         ;YES PRINT SPACE
DSP_MUX1
          RRA       R13
          RRA       R13
          MOV.B     LCDTAB(R12),R4
          MOV.B     R4,LCDM(R13)
          RRA       R4
          MOV.B     R4,LCDM+1(R13)
          RRA       R4
          MOV.B     R4,LCDM+2(R13)
          RRA       R4
          MOV.B     R4,LCDM+3(R13)
          RET
```

8.5 Use of Unused Select Lines for Digital Outputs

The LCD Driver of the MSP430 allows the use of additional digital outputs if select lines are not used. Up to 28 digital outputs are possible in the hardware design, but not all of them will be implemented for a given chip. The addressing scheme for the digital outputs O2 to O29 is as follows:

Address	7 6 5 4	3 2 1 0	Digit Nr.	LCDP
03Fh	O29	O28	Digit 15	6 to 0
03Eh	O27	O26	Digit 14	6 to 0
03Dh	O25	O24	Digit 13	5 to 0
03Ch	O23	O22	Digit 12	5 to 0
03Bh	O21	O20	Digit 11	4 to 0
03Ah	O19	O18	Digit 10	4 to 0
039h	O17	O16	Digit 9	3 to 0
038h	O15	O14	Digit 8	3 to 0
037h	O13	O12	Digit 7	2 to 0
036h	O11	O10	Digit 6	2 to 0
035h	O09	O08	Digit 5	1 to 0
034h	O07	O06	Digit 4	1 to 0
033h	O05	O04	Digit 3	0
032h	O03	O02	Digit 2	0
031h	h g f e	d c b a	Digit 1	

Table 8.6: Dependence of the Select Lines on LCDP

The above table shows the dependence of the select/output lines on the 3-bit value LCDP. Only if LCDP = 7 are all lines switched to the LCD Mode (select lines).

Note: Restrictions using select lines as outputs

The above table shows the digit environment for a 4MUX LCD display. The outputs O0 and O1 are not available: S0 and S1 are always implemented. (digit 1).

The digital outputs Ox have always to be addressed with all four bits. This means that 0Fh is to be used for the addressing of one output.

Only byte addressing is allowed for the addressing of the LCD controller bytes.

- Software example: S0 to S13 drive a 4MUX LCD (7 digits), O14 to O17 are digital outputs.

```
;LCD DRIVER DEFINITIONS:

LCDM      .EQU      030H              ; ADDRESS LCD CONTROL BYTE
LCDM0     .EQU      001H              ; 0: LCD OFF   1: LCD ON
LCDM1     .EQU      002H              ; 0: HIGH      0: LOW IMPEDANCE
MUX       .EQU      004H              ; MUX: STATIC, 2MUX, 3MUX, 4MUX
LCDP      .EQU      020H              ; SELECT/OUTPUT DEFINITION
                                      ; LCDM7/6/5
O14       .EQU      00FH              ; O14 CONTROL DEFINITION
O15       .EQU      0F0H              ; O15
O16       .EQU      00FH              ; O16
O17       .EQU      0F0H              ; O17

; INITIALIZATION:DISPLAY ON:          LCDM0 = 1
;                HI IMPEDANCE         LCDM1 = 0
;                4MUX:                LCDM4/3/2 = 7
;         O14 TO O17 ARE OUTPUTS:     LCDM7/6/5 = 3

          MOV       #(LCDP*3)+(MUX*7)+LCDM0,&LCDM      ; INIT LCD
          ...

; NORMAL PROGRAM EXECUTION:
; SOME EXAMPLES HOW TO MODIFY THE DIGITAL OUTPUTS O14 TO O17:

          BIS.B     #O14,&LCDM+8        ; SET O14, O15 UNCHANGED
          BIC.B     #O15+O14,&LCDM+8    ; RESET O14 AND O15
          MOV.B     #O15+O14,&LCDM+8    ; SET O14 AND O15
          MOV.B     #O17,&LCDM+9        ; RESET O16, SET O17
          XOR.B     #O17,&LCDM+9        ; TOGGLE O17, O16 STAYS
                                        ; UNCHANGED
```

Topics

Figures

Notes

9 The Analogue-to-Digital Converters

Two completely different Analogue-to-Digital Converters (ADCs) are in use, depending on the MSP430 device type:
- EVE contains a successive approximation ADC with 14 and 12-bit resolution
- EVE_OPT contains a capacitor discharge unit which allows comparison of discharge times with measurement resistors (resistive sensors).

9.1 The (12+2)-bit Analogue-to-Digital Converter

The ADC of the MSP430 is usable in two different modes:

- (12+2)-bit ADC with an input range of the complete SVcc. The ADC searches automatically which one of the four ranges is currently appropriate to the input voltage. This searching adds 30 MCLK cycles to the conversion time. The complete conversion time for a 14-bit conversion is 132 MCLK cycles.
- 12-bit ADC with four ranges. Each range covers one fourth of the SVcc. This conversion mode is used, if the voltage range of the input signal is known. The conversion needs 102 µs.

The sampling of the ADC input takes 12 MCLK cycles; this means the sampling gate is open during this time (12µs@1MHz). The input of an ADC pin can be seen as an RC low pass filter: 2kΩ in series with 32pF. The 32pF capacitor must be charged during the 12 MCLK cycles to the final value to be measured. This means within 2^{-14} of this value. This time limits the internal resistance Ri of the source to be measured:

$$(Ri + 2k\Omega) \times 32pF < \frac{12\mu s}{\ln 2^{14}}$$

Solved for Ri this results in:

$$Ri < 36.6k\Omega$$

For the full resolution of the ADC the internal resistance of the input signal must be lower than 36.6kΩ.

If a resolution of n bits is sufficient then the internal resistance Ri of the ADC input source can be higher:

$$Ri < \frac{12\mu s}{\ln 2^n \times 32pF} - 2k\Omega \rightarrow Ri < \frac{375000}{\ln 2^n} - 2k\Omega$$

EXAMPLE: To get a resolution of 13 bits, what is the maximum internal resistance of the input signal?

$$Ri < \frac{375000}{\ln 2^{13}} - 2k\Omega = \frac{375000}{9.0109} - 2k\Omega = 41.6k - 2k = 39.6k\Omega$$

The internal resistance of the input signal must be lower than 39.6kΩ.

The next figure shows different methods of connecting analogue signals to the MSP430:

1. Current supply for resistive sensors (Rsens1 at A0)
2. Voltage supply for resistive sensors (Rsens2 at A1)
3. Direct connection of input signals (Vin at A2)
4. 4-Wire circuitry with current supply (Rsens3 at A3 to A5
5. 4-Wire circuitry with voltage supply (Rsens4 at A6 to A7

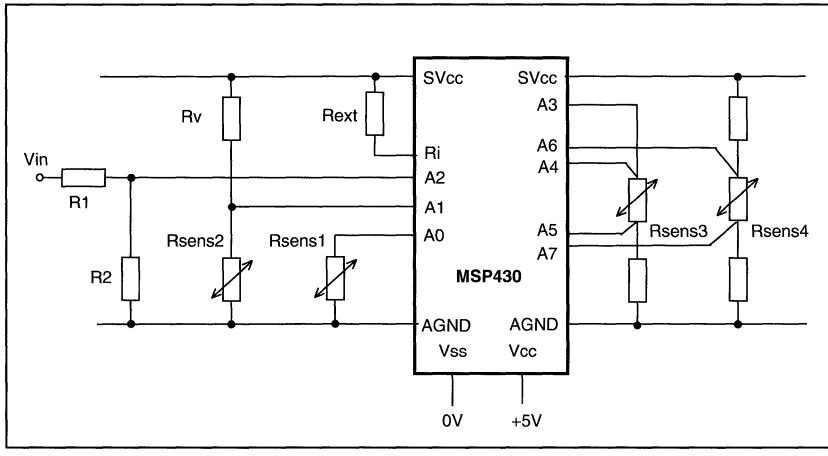


Figure 9.1: Possible Sensor Connections to the MSP430

9.1.1 The Current Source

A stable, programmable Current Source is available at the analogue inputs A0 to A3. With a programming resistor Rext between pins SVcc and Ri it is possible to get defined currents out of the programmed analogue input An: the current is directly related to the voltage SVcc. The analogue input to be measured and the analogue input for the Current Source are independent of each other. This means that the Current Source may be programmed to A3 and the measurement taken from A4 as shown in the example above.

When using the Current Source it is not possible to use the full range of the ADC: only the range defined with "Load Compliance" in the Electrical Description is usable (0.5SVcc in Revision 0.44, which means only ranges A and B).

The current I_{CS} defined by the external resistor Rext is:

$$ICS = \frac{0.25 \times SVcc}{Rext}$$

The input voltage at the analogue input with the current I_{CS} is then:

$$Vin = RSENS \times ICS = RSENS \times \frac{0.25 \times SVcc}{Rext}$$

9.1.2 The (12+2)-bit Analogue-to-Digital Converter used in 14-bit Mode

The 14-bit mode is used if the range of the input voltage exceeds one ADC range. The input signal range is from analogue ground (Vss) to SVcc (Vcc).

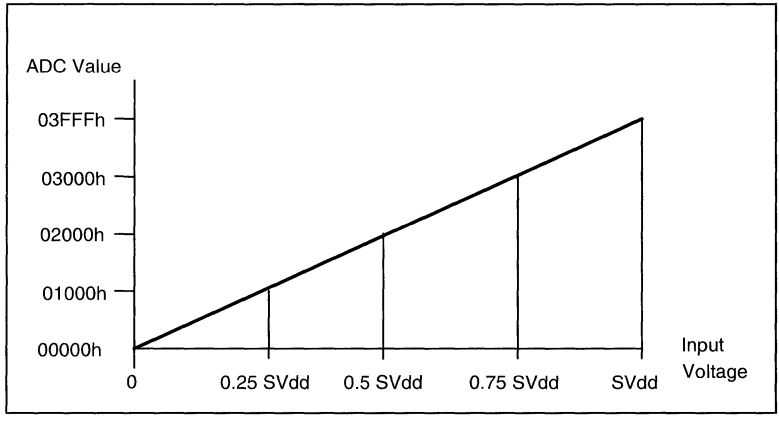


Figure 9.2: Complete ADC Range

The nominal ADC formulas for the 14-bit conversion are:

$$N = \frac{VAx}{Vref} \times 2^{14} \rightarrow VAx = \frac{N \times Vref}{2^{14}}$$

with: N 14-bit result of the ADC conversion
V_{Ax} Input voltage at the selected analogue input Ax
Vref Voltage at pin SVcc (external reference or internal Vcc)

If the current source is used, the above equation changes to:

$$N = \frac{0.25 \times Vref}{Rext} \times \frac{Rx}{Vref} \times 2^{14} = \frac{Rx}{Rext} \times 2^{12}$$

This gives for the resistor Rx:

$$Rx = \frac{N \times Rext}{2^{12}}$$

with: Rext Resistor between SVcc pin and Ri pin (defines current Ics)
Rx Resistor to be measured (connected to Ax and AGND)

9.1.2.1 ADC with Signed Signals

The ADC of the MSP430 measures unsigned signals from Vss to Vcc. If signed measurements are necessary, then a virtual zero-point has to be provided. Signals above this zero-point are treated as positive signals; signals below it are treated as negative ones.

Three possibilities for a virtual zero-point are shown:
- Virtual Ground IC
- Split power supply
- Use of the current source

Virtual Ground IC

With the "Phase Splitter" TLE2426 a common reference is created which lies exactly in the middle of the voltage SVcc. All signed input voltages are connected to this virtual ground with their reference potential (0V). The virtual ground voltage (at A0) is measured at regular time intervals and the measured ADC value is stored and subtracted from the measured signal (at A1). This gives a signed result for the input A1.

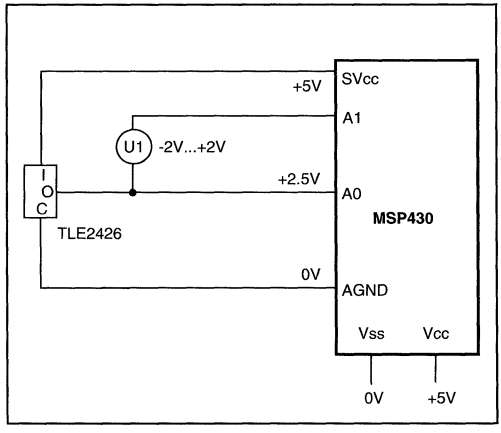


Figure 9.3: Virtual Ground IC for Level Shifting

Note: ADC Definitions are Valid for all ADC Examples

The ADC definitions given in the next example are valid for all ADC examples which follow. They are in accordance with the "MSP430 Family User's Guide Preliminary Specification".

EXAMPLE: The virtual ground voltage at A0 is measured and stored in RAM cell VIRTGR. The value of VIRTGR is subtracted from the ADC value measured at input A1. This gives the signed value for the A1 input.

```
; HARDWARE DEFINITIONS FOR THE ANALOGUE-TO-DIGITAL CONVERTER

AIN       .EQU      0110H          ; INPUT REGISTER (FOR DIGITAL
                                   ; INPUTS)
AEN       .EQU      0112H          ; 0: ANALOGUE INPUT   1: DIGITAL
                                   ; INPUT
ACTL      .EQU      0114H          ; ADC CONTROL REGISTER
CS        .EQU      01H            ; CONVERSION START
VREF      .EQU      02H            ; 0: EXT. REFERENCE    1: SVCC ON
A0        .EQU      00H            ; INPUT A0
A1        .EQU      04H            ; INPUT A1
A2        .EQU      08H            ; INPUT A2
CSA0      .EQU      00H            ; CURRENT SOURCE TO A0
CSA1      .EQU      40H            ; CURRENT SOURCE TO A1
CSOFF     .EQU      100H           ; CURRENT SOURCE OFF
CSON      .EQU      000H           ; CURRENT SOURCE ON
RNGA      .EQU      00H            ; RANGE SELECT A (0 ... 0.25SVCC)
RNGB      .EQU      200H           ; RANGE SELECT B (0.25..0.50SVCC)
RNGC      .EQU      400H           ; RANGE SELECT C (0.5...0.75SVCC)
RNGD      .EQU      600H           ; RANGE SELECT D (0.75..SVCC)
RNGAUTO   .EQU      800H           ; 1: RANGE SELECTED AUTOMATICALLY
PD        .EQU      1000H          ; 1: ADC POWERED DOWN

ADAT      .EQU      0118H          ; ADC DATA REGISTER (12 OR 14-BIT)
IFG2      .EQU      03H            ; INTERRUPT FLAG REGISTER 2
ADIFG     .EQU      04H            ; ADC "EOC" BIT (IFG2.2)

IE2       .EQU      01H            ; INTERRUPT ENABLE REGISTER 2
ADIE      .EQU      02H            ; ADC INTERRUPT ENABLE BIT

VIRTGR    .EQU      R4             ; VIRTUAL GROUND ADC VALUE


; MEASURE VIRTUAL GROUND INPUT A0 AND STORE VALUE FOR REFERENCE
          MOV       #RNGAUTO+CSOFF+A0+VREF+CS,&ACTL
```

```
L$101     BIT.B     #ADIFG,&IFG2   ; CONVERSION COMPLETED?
          JZ        L$101          ; IF Z=1: NO
          MOV       &ADAT,VIRTGR   ; STORE A0 14-BIT VALUE
          ...

; MEASURE INPUT A1 (0 ...03FFFH) AND COMPUTE SIGNED VALUE
; (02000H ...01FFFH).

          MOV       #RNGAUTO+CSOFF+A1+VREF+CS,&ACTL
L$102     BIT.B     #ADIFG,&IFG2   ; CONVERSION COMPLETED?
          JZ        L$102          ; IF Z=1: NO

          MOV       &ADAT,R5       ; READ ADC VALUE FOR A1
          SUB       VIRTGR,R5      ; R5 CONTAINS SIGNED ADC VALUE
```

Split Power Supply

With two power supplies, for example +2.5V and -2.5V, a potential in the middle of the ADC range of the MSP430 can be created. All signed input voltages are connected to this voltage with their reference potential (0V). The mid range voltage (at A0) is measured at regular time intervals and the measured ADC value is stored and subtracted from the measured signal (at A1). This gives a signed result for the input A1.

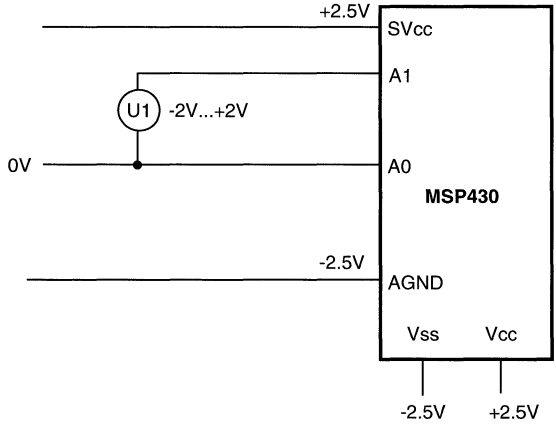


Figure 9.4: Split Power Supply for Level Shifting

The same software can be used as shown with the Virtual Ground IC.

Use of the Current Source

With the current source a voltage which is partially or completely below the AGND potential can be shifted to the middle of the usable ADC range of the MSP430. This is accomplished by a resistor Rh whose voltage drop shifts the input voltage accordingly. This method is useful especially if differential measurements are necessary, because the ADC value of the signal's midpoint is not available as easily as with the methods shown previously.

The example below shows an input signal V1 reaching from -1V to +1V. To shift the signal's midpoint (0V) to the midpoint of the usable ADC range (SVcc/4) a current I_{CS} is used. The necessary current I_{CS} to shift the input signal is:

$$Ics = \frac{SVcc/4}{Rh} \quad \rightarrow \quad Rh = \frac{SVcc/4}{Ics}$$

Rh includes the internal resistance of the voltage source Vi.

The current Ics of the current source is defined by:

$$Ics = \frac{0.25 \times SVcc}{Rext}$$

Therefore, the necessary shift resistor Rh is

$$Rh = \frac{SVcc/4 \times Rext}{0.25 \times SVcc} \rightarrow Rh = Rext$$

The voltage V_{A1} at the analogue input A1 is:

$$V_{A1} = V1 + Rh \times \frac{0.25 \times SVcc}{Rext}$$

Therefore, the unknown voltage V1 is:

$$V1 = V_{A1} - Rh \times \frac{0.25 \times SVcc}{Rext} = SVcc\left(\frac{N}{2^{14}} - \frac{Rh \times 0.25}{Rext}\right)$$

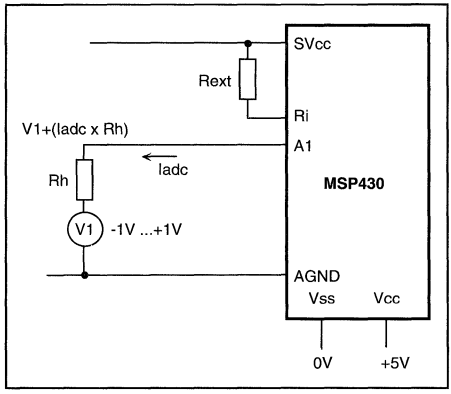


Figure 9.5: Current Source for Level Shifting

9.1.2.2 Four-Wire Circuitry for Sensors

A proven method for eliminating the error coming from the voltage drop on the connection lines to the sensor is the use of 4-wire circuitry. Instead of 2 lines, 4 lines are used: 2 for the measurement current, and 2 for the sensor voltages. These 2 sensor lines do not carry current (the input current of the analogue inputs is only some nanoamps), and this means that no voltage drop falsifies the measured values. The formula for voltage supply is:

$$Rsens = \frac{R1+R2}{\frac{2^{14}}{\Delta N}-1}$$

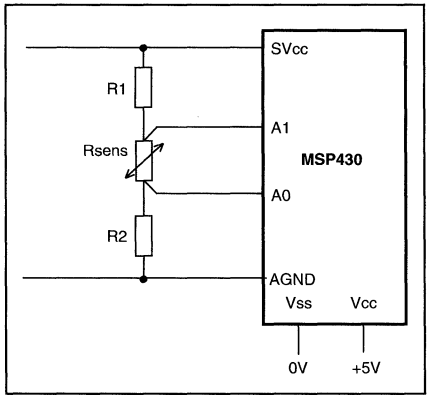


Figure 9.6: 4-Wire Circuitry with Voltage Supply

EXAMPLE: The sensor Rsens at A0 and A1 is measured, and the ADC value of it is computed by the difference of the two results measured at A1 and A0. The result is stored in R5.

```
; MEASURE UPPER VALUE OF RSENS AT INPUT A1 AND STORE VALUE

          MOV       #RNGAUTO+CSOFF+A1+VREF+CS,&ACTL
L$103     BIT.B     #ADIFG,&IFG2        ; CONVERSION COMPLETED?
          JZ        L$103               ; IF Z=1: NO

          MOV       &ADAT,R5            ; STORE A1 VALUE

; MEASURE INPUT A0 AND COMPUTE ADC VALUE OF RSENS

          MOV       #RNGAUTO+CSOFF+A0+VREF+CS,&ACTL
L$104     BIT.B     #ADIFG,&IFG2        ; CONVERSION COMPLETED?
          JZ        L$104               ; IF Z=1: NO

          SUB       &ADAT,R5            ; R5 CONTAINS RSENS ADC VALUE
```

The next figure shows the more common 4-wire circuitry with Current Supply:

$$Rsens = \frac{\Delta N \times Rext}{2^{12}}$$

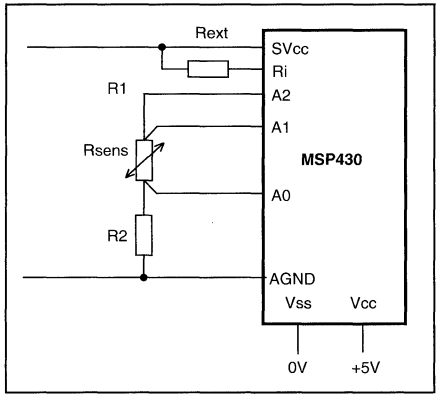


Figure 9.7: 4-Wire Circuitry with Current Supply

9.1.2.3 Referencing with Reference Resistors

A system that uses sensors normally needs to be calibrated, due to tolerances of the sensors themselves, and of the ADC. A way to omit the costly calibration procedure is the use of reference resistors. Two different methods can be used, depending on the kind of sensor:

1. Platinum sensors. These are sensors with a precisely known temperature-resistance characteristic. Precision resistors are used with the sensor values of the temperatures at the two limits of the range.
2. Other sensors. Nearly all other sensors have tolerances. This makes it necessary to group sensors with similar characteristics and to select the two reference resistors according to the upper and lower limits of these groups.

If the two reference resistors have precisely the values of the sensors at the range limits (or at another well-defined point) then all tolerances are eliminated during calculation:

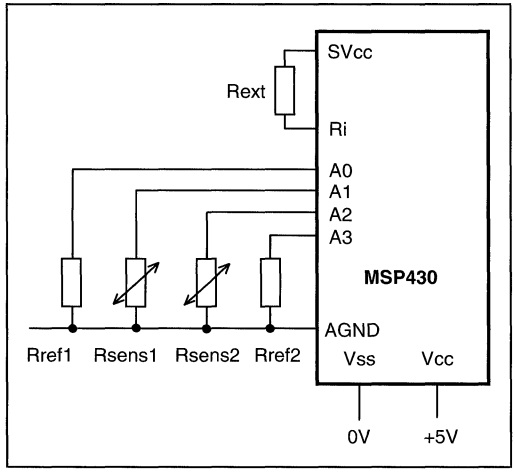


Figure 9.8: Referencing with Precision Resistors

The nominal formulas, given in the preceding sections, need to be changed if offset and slope are considered. The ADC value Nx for a given resistor Rx is now:

$$Nx = \frac{0.25 \times Rx}{Rext} \times 2^{14} \times Slope + Offset$$

With two known resistors Rref1 and Rref2 it is possible to compute slope and offset and to get the values of unknown resistors exactly. The result of the solved equations gives:

$$Rx = \frac{Nx - Nref2}{Nref2 - Nref1} \times (Rref2 - Rref1) + Rref2$$

with:
- Nx ADC conversion result for Rx
- Nref1 ADC conversion result for Rref1
- Nref2 ADC conversion result for Rref2
- Rref1 Resistance of Rref1
- Rref2 Resistanceof Rref2

As shown, only known or measurable values are needed for the computation of Rx from Nx. The slope and offset of the ADC disappear completely.

9.1.2.4 Interrupt Handling using the 14-bit-Mode

The examples shown above all use polling techniques for checking the completion of conversion. This takes up computing power which can be used otherwise if interrupt techniques are used.

EXAMPLE: Analogue input A0 (without Current Source) and A1 (with Current Source) are measured alternately. The measured 14-bit results are stored in address MEAS0 for A0 and MEAS1 for A1. The background software uses these measured values and sets them to 0FFFFh after use. The time interval between two measurements is defined by the 8-bit timer: every timer interrupt starts a new conversion for the prepared analogue input.

```
; HARDWARE DEFINITIONS SEE 1ST ADC EXAMPLE
; ANALOGUE INPUT                    A0        A1
; CURRENT SOURCE                    OFF       ON
; RESULT TO                         MEAS0     MEAS1
; RANGE SELECTION                   AUTO      AUTO
; REFERENCE                         SVCC      SVCC

; INITIALIZATION PART FOR THE ADC:

          MOV       #RNGAUTO+CSOFF+A0+VREF,&ACTL
          MOV.B     #ADIE,&IE2        ; ENABLE ADC INTERRUPT
          MOV       #0FFH-3,&AEN      ; ONLY A0 AND A1 ANALOGUE INPUTS
          ...                         ; INITIALIZE OTHER MODULES

; ADC INTERRUPT HANDLER: A0 AND A1 ARE MEASURED ALTERNATIVELY
; THE NEXT MEASUREMENT IS PREPARED BUT NOT STARTED.

AD_INT    BIT       #A1,&ACTL         ; A1 RESULT IN ADAT?
          JNZ       ADI               ; YES
          MOV       &ADAT,MEAS0       ; A0 VALUE IS ACTUAL
          MOV       #RNGAUTO+CSON+A1+VREF,&ACTL   ; A1 NEXT MEAS.
          RETI
```

```
ADI       MOV       &ADAT,MEAS1      ; A1 VALUE
          MOV       #RNGAUTO+CSOFF+A0+VREF,&ACTL    ; A0 NEXT MEAS.
          RETI

; 8-BIT TIMER INTERRUPT HANDLER: THE ADC CONVERSION IS STARTED
; FOR THE PREPARED ADC INPUT

T8BINT    BIS       #CS,&ACTL        ; START CONVERSION FOR THE ADC
          ...
          RETI

          .SECT     "INT_VEC0",0FFEAH     ; INTERRUPT VECTORS
          .WORD     AD_INT                ; ADC INTERRUPT VECTOR;
          .SECT     "INT_VEC1",0FFF8H
          .WORD     T8BINT                ; 8-BIT TIMER INTERRUPT
                                          ; VECTOR
```

9.1.3 The (12+2)-bit Analogue-to-Digital Converter used in 12-bit Mode

This mode is used if it is known in which range the input voltage is. If, for example, a temperature sensor is used whose signal range always fits into one range (for example range C), then the 12-bit mode is the correct selection. The measurement time with MCLK = 1MHz is only 102 µs, compared to 132 µs if the auto ranging mode is used. The following Figure shows the four ranges compared to SVcc.

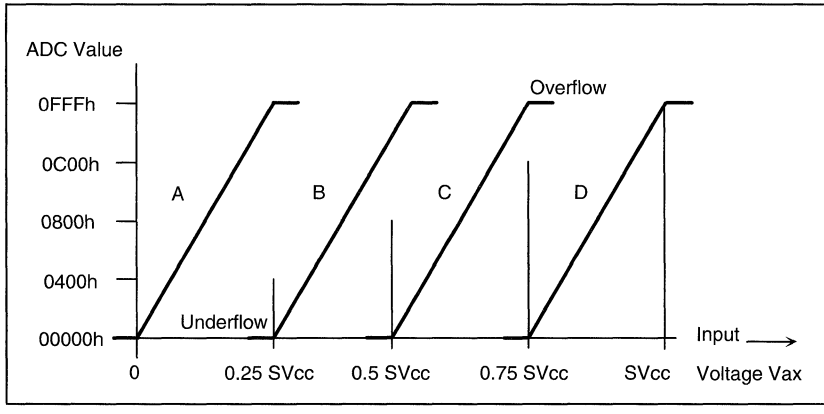


Figure 9.9: The four Single ADC Ranges

Note: ADC Ranges

The ADC results 0000H and 0FFFh mean underflow and overflow: the voltage at the measured analogue input is below or above the limits of the addressed range respectively.

The next figure shows how one of the ranges can be seen:

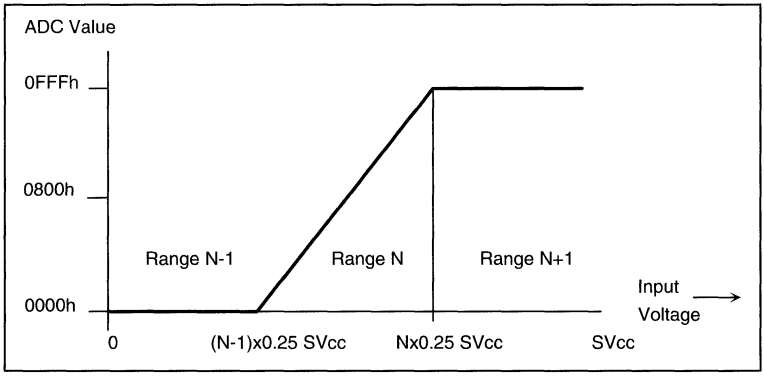


Figure 9.10: Single ADC Range

The possible ways to connect sensors to the MSP430 are the same as shown for the (12+2)-bit ADC:

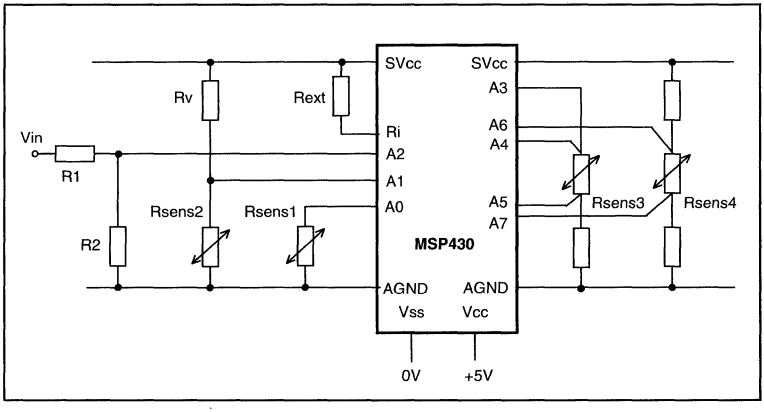


Figure 9.11: Possible Sensor Connections to the MSP430

The nominal ADC formulas for the 12-bit conversion are:

$$N = \frac{V_{Ax} - n \times 0.25 \times Vref}{Vref} \times 2^{14} \rightarrow V_{Ax} = Vref \left(\frac{N}{2^{14}} + n \times 0.25 \right)$$

with:

N		12-bit result of the ADC conversion
V_{Ax}		Input voltage at the selected analogue input Ax
Vref		Voltage at pin SVcc (external reference or internal Vcc)
n		Range constant (n = 0,1,2,3 for ranges A,B,C,D)

The ADC formula for a resistor Rx (Rsens2 in the above figure) which is connected to Vref via a resistor Rv is:

$$N = \frac{\frac{Rx}{Rv + Rx} \times Vref - n \times 0.25 \times Vref}{Vref} \times 2^{14} \rightarrow Rx = Rv \times \frac{\frac{N}{2^{12}} + n}{4 - \left(\frac{N}{2^{12}} + n \right)}$$

If a current source is used (as for Rsens1 in the above figure), the above equation changes to:

$$N = \frac{\frac{0.25 \times Vref}{Re\,xt} \times Rx - n \times 0.25 \times Vref}{Vref} \times 2^{14} = \left(\frac{Rx}{Re\,xt} - n \right) \times 2^{12}$$

This gives for the unknown resistor Rx:

$$Rx = \left(\frac{N}{2^{12}} + n \right) \times Rext$$

with:

Rext		Resistor between SVcc pin and Ri pin (defines current Ics)
Rx		Resistor to be measured (connected to Ax and AGND)

9.1.3.1 ADC with Signed Signals

Only the Current Source method is applicable if signed signals have to be measured:
- Normal phase splitter circuits are not able to shift the virtual ground into the middle of range A (0.125 SVcc) or B (0.375 SVcc), as is necessary here.
- The split power supply method would need two different voltages to get the zero point into the middle of range A (0.625V/4.375V) or range B (1.875V/3.125V)

For signed signals it is necessary to shift the input signal V1 to the middle of the range A or B. If range B (0.375 SVcc) is used the necessary shift resistor Rh is

$$Rh = \frac{0.375 \times SVcc \times Re\,xt}{0.25 \times SVcc} \rightarrow Rh = 1.5 \times Re\,xt$$

The unknown voltage V1 referred to its zero point in the middle of range n is:

$$V1 = V_{Ax} - Rh \times Ics$$

With the above equations for Vax this leads to:

$$V1 = 0.25 \times SVcc\left(\frac{N}{2^{12}} + n - \frac{Rh}{Rext}\right)$$

9.1.3.2 Interrupt Handling using the 12-bit-Mode

The software is the same as for the 14-bit conversion. The only difference is the omission of the RNGAUTO bit during the initialization of ACTL. Instead, the desired range is to be included into the initialization part of each measurement.

EXAMPLE: Analogue input A0 (without Current Source, always range C, external reference at pin SVcc) and A1 (with Current Source, always range A) have to be measured alternately. The measured 12-bit results have to be stored in address MEAS0 for A0 and MEAS1 for A1. The background software uses these measured values and sets them to 0FFFFh after use. The time interval between two measurements is defined by the 8-bit timer: every timer interrupt starts a new conversion for the prepared analogue input.

```
; HARDWARE DEFINITIONS SEE 1ST ADC EXAMPLE
; ANALOGUE INPUT                          A0          A1
; CURRENT SOURCE                          OFF         ON
; RESULT TO                               MEAS0       MEAS1
; RANGE                                   C           A
; REFERENCE                               EXTERNAL    SVCC

; INITIALIZATION PART FOR THE ADC:

          MOV       #RNGC+CSOFF+A0,&ACTL
          MOV.B     #ADIE,&IE2            ; ENABLE ADC INTERRUPT
          MOV       #0FFH-3,&AEN          ; ONLY A0 AND A1 ANALOGUE
                                          ; INPUTS
          ...                             ; INITIALIZE OTHER MODULES

; ADC INTERRUPT HANDLER: A0 AND A1 ARE MEASURED ALTERNATIVELY
; THE NEXT MEASUREMENT IS PREPARED BUT NOT STARTED

AD_INT    BIT       #A1,&ACTL             ; A1 MEASURED ?
          JNZ       ADI                   ; YES
          MOV       &ADAT,MEAS0           ; A0 VALUE IS ACTUAL
          MOV       #RNGA+CSA1+A1+VREF,&ACTL     ; A1 NEXT MEAS.
          RETI
```

```
ADI       MOV       &ADAT,MEAS1           ; A1 VALUE
          MOV       #RNGC+CSOFF+A0,&ACTL  ; A0 NEXT MEASUREMENT
          RETI

; 8-BIT TIMER INTERRUPT HANDLER: THE ADC CONVERSION IS STARTED
;  FOR THE ADDRESSED ADC INPUT

T8BINT    BIS       #CS,&ACTL             ; START CONVERSION
          ...
          RETI

          .SECT     "INT_VECT",0FFEAH     ; INTERRUPT VECTORS
          .WORD     AD_INT                ; ADC INTERRUPT VECTOR;
          .SECT     "INT_VECT",0FFF8H
          .WORD     T8BINT                ; 8-BIT TIMER INTERRUPT
                                          ; VECTOR
```

9.2 The Universal Timer/Port Module used as ADC

This ADC module is contained in MSP430 versions that do not have the (12+2)-bit ADC. The function is completely different from the (12+2)-bit ADC: the discharge times tdc for different resistors are measured and compared.

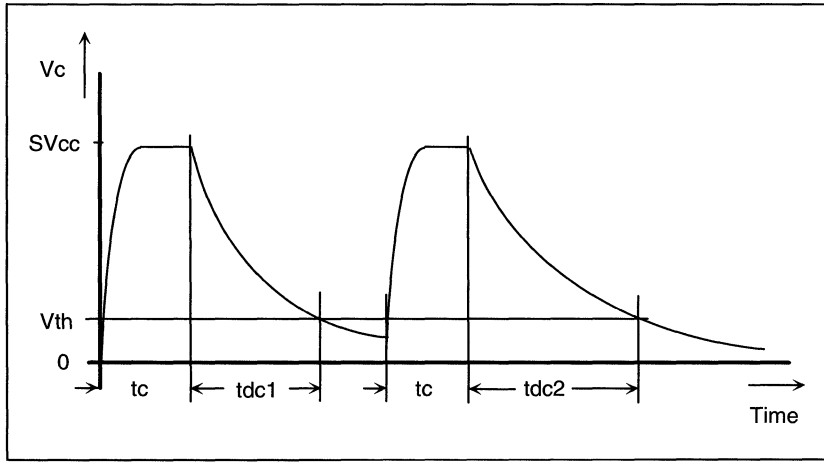


Figure 9.12: Timing for the Universal Timer

with	Vth	Threshold voltage of the comparator
	tdc1	Discharge time with the reference resistor
	tdc2	Discharge time with the sensor
	tc	Charge time for the capacitor

EXAMPLE: Use of the Universal Timer Port as an ADC without interrupt

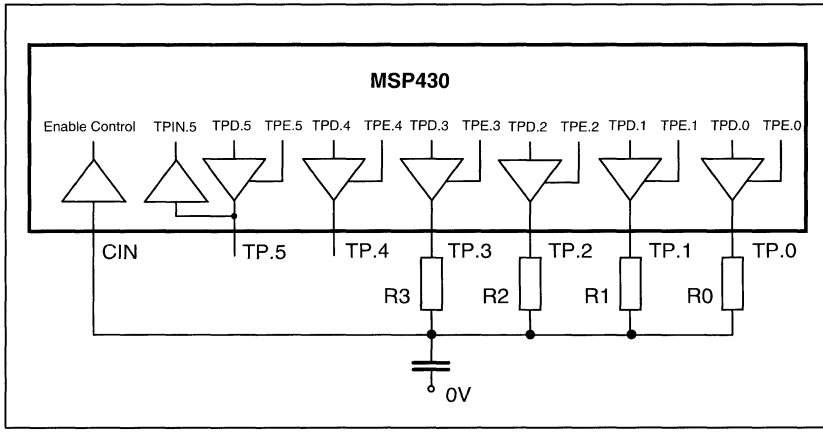


Figure 9.13: Schematic of Example

```
; DEFINITION PART FOR THE UT/PM ADC

TPCTL     .EQU     04BH              ; TIMER PORT CONTROL REGISTER
TPSSEL0   .EQU     040H              ; TPSSEL.0
ENB       .EQU     020H              ; CONTROLS EN1 OF TPCNT1
ENA       .EQU     010H              ; AS ENB
EN1       .EQU     008H              ; ENABLE INPUT FOR TPCNT1
RC2FG     .EQU     004H              ; RIPPLE CARRY TPCNT2
EN1FG     .EQU     001H              ; EN1 FLAG BIT

TPCNT1    .EQU     04CH              ; LO 8-BIT COUNTER/TIMER
TPCNT2    .EQU     04DH              ; HI 8-BIT COUNTER/TIMER

TPD       .EQU     04EH              ; DATA REGISTER
B16       .EQU     080H              ; 0: SEPARATE TIMERS  1: 16-BIT
                                     ; TIMER
CPON      .EQU     040H              ; 0: COMP OFF       1: COMP ON
TPDMAX    .EQU     008H              ; BIT POSITION OUTPUT TPD.MAX
```

```
TPE       .EQU      04FH              ; DATA ENABLE REGISTER

MSTACK    .EQU      0240H             ; RESULT STACK 1ST WORD
NN        .EQU      011H              ; TPCNT2 VALUE FOR CHARGING OF C

; MEASUREMENT SUBROUTINE WITHOUT INTERRUPT. TPD.4 AND TPD.5
; ARE NOT USED AND THEREFORE OVERWRITTEN
; INITIALIZATION: STACK INDEX <- 0, START WITH TPD.3
; 16-BIT TIMER, MCLK, CIN ENABLES COUNTING

MEASURE   PUSH.B#TPDMAX             ; START WITH SENSOR R3 TPD.MAX
          CLR       R5              ; INDEX FOR RESULT STACK
MEASLOP   MOV.B     #(TPSSEL0*3)+ENB+ENA,&TPCTL   ; RESET FLAGS

; CAPACITOR C IS CHARGED UP FOR > 5 TAU. N-1 OUTPUTS ARE USED

          MOV.B     #B16+CPON+TPDMAX-1,&TPD       ; SELECT CHARGE
OUTPUTS
          MOV.B     #TPDMAX-1,&TPE; ENABLE CHARGE OUTPUTS
          MOV.B     #NN,&TPCNT2       ; LOAD NEG. CHARGE TIME

MLP0      BIT.B     #RC2FG,&TPCTL     ; CHARGE TIME ELAPSED?
          JZ        MLP0              ; NO CONTINUE WAITING

          MOV.B     @SP,&TPE          ; ENABLE ONLY ACTUAL SENSOR
          CLR.B     &TPCNT2           ; CLEAR HI BYTE TIMER


; SWITCH ALL INTERRUPTS OFF, TO ALLOW NON-INTERRUPTED START
; OF TIMER AND CAPACITY DISCHARGE

          DINT                        ; ALLOW NEXT 2 INSTRUCTIONS
          CLR.B     &TPCNT1           ; CLEAR LO BYTE TIMER
          BIC.B     @SP,&TPD          ; SWITCH ACTUAL SENSOR TO
                                      ; LO
          EINT                        ; COMMON START TOOK PLACE
; WAIT UNTIL EOC (EN1 = 1) OR OVERFLOW ERROR (RC2FG = 1)

MLP1      BIT.B     #RC2FG,&TPCTL     ; OVERFLOW (BROKEN SENSOR)?
          JNZ       MERR              ; YES, GO TO ERROR HANDLING
          BIT.B     #EN1,&TPCTL       ; CIN < UCOMP?
          JZ        MLP1              ; NO, WAIT

; EN1 = 0: END OF CONVERSION: STORE 2 X 8 BIT RESULT ON MSTACK
; ADDRESS NEXT SENSOR, IF NO ONE ADDRESSED: END REACHED

          MOV.B     &TPCNT1,MSTACK(R5)  ; STORE RESULT ON STACK
          MOV.B     &TPCNT2,MSTACK+1(R5) ; HI BYTE
L$301     INCD      R5                        ; ADDRESS NEXT WORD
          RRA.B     @SP                       ; NEXT OUTPUT TPD.X
```

```
          JNC       MEASLOP                    ; IF C=1: FINISHED
          INCD      SP                         ; HOUSEKEEPING:
                                               ; TPDMAX OFF STACK

          RET

; ERROR HANDLING: ONLY OVERFLOW POSSIBLE (BROKEN SENSOR ?)
; 0FFFFH IS WRITTEN FOR RESULT AND SUBROUTINE CONTINUED

MERR      MOV       #0FFFFH,MSTACK(R5)    ; OVERFLOW
          JMP       L$301
```

9.2.1 Interrupt Handling

EXAMPLE: Use of the Universal Timer Port as an ADC with interrupt. This has the same function as the example without interrupt.

```
; DEFINITION PART FOR THE UT/PM ADC

TPCTL     .EQU      04BH            ; TIMER PORT CONTROL REGISTER
TPSSEL0   .EQU      040H            ; TPSSEL.0
ENB       .EQU      020H            ; CONTROLS EN1 OF TPCNT1
ENA       .EQU      010H            ; AS ENB
EN1       .EQU      008H            ; ENABLE INPUT FOR TPCNT1
RC2FG     .EQU      004H            ; RIPPLE CARRY TPCNT2
EN1FG     .EQU      001H            ; EN1 FLAG BIT

TPCNT1    .EQU      04CH            ; LO 8-BIT COUNTER/TIMER
TPCNT2    .EQU      04DH            ; HI 8-BIT COUNTER/TIMER

TPD       .EQU      04EH            ; DATA REGISTER
B16       .EQU      080H            ; 0: SEPARATE TIMERS  1: 16-BIT
                                    ; TIMER
CPON      .EQU      040H            ; 0: COMP OFF 1: COMP ON

TPE       .EQU      04FH            ; DATA ENABLE REGISTER

MSTACK    .EQU      0240H           ; RESULT STACK 1ST WORD
                                    ; (8 BYTES)
ADCST     .EQU      MSTACK+8
NN        .EQU      011H            ; TPCNT2 VALUE FOR CHARGING OF C

IFG2      .EQU      003H            ; INTERRUPT FLAG REGISTER 2
TPIFG     .EQU      008H            ; ADC INTERRUPT FLAG

IE2       .EQU      001H            ; INTERRUPT ENABLE REGISTER 2
ADIE      .EQU      004H            ; ADC INTERRUPT ENABLE BIT
```

```
TP0       .EQU      01H             ; TP.0 BIT ADDRESS
TP1       .EQU      02H             ; TP.1 BIT ADDRESS
TP2       .EQU      04H             ; TP.2 BIT ADDRESS
TP3       .EQU      08H             ; TP.3 BIT ADDRESS


; MEASUREMENT SUBROUTINE WITH INTERRUPT. TPD.4 AND TPD.5
; ARE NOT USED AND THEREFORE OVERWRITTEN

; RETURN: RESULTS FOR TP.3 TO TP.0 IN MSTACK TO MSTACK+6
;                  ADCST = 10: RESULTS OK
;                  ADCST = 11: ERROR

; INITIALIZATION: ADCST<- 1, 16-BIT TIMER, MCLK
; CIN ENABLES COUNTING
; ADCST IS SET: CAUSES INTERRUPT FOR CHARGE INITIALIZATION

MEASINIT MOV.B #1,ADCST           ; STATUS TO INIT. OF CHARGE
         BIS.B    #TPIFG,&IFG2    ; CAUSES INTERRUPT FOR INIT.
         BIS.B    #ADIE,&IE2      ; ENABLE ADC INTERRUPT
         EINT                     ; GIE ON
         ...                      ; CONTINUE MAIN PROGRAM

ADCINT   PUSH     R6              ; WORKING REGISTER
         MOV.B    ADCST,R6        ; ADC STATUS BYTE
         MOV.B    ADCIT(R6),R6    ; REL. ADDRESS OF CURRENT
                                  ; HANDLER
         ADD      R6,PC           ; BRANCH TO HANDLER
ADCIT    .BYTE    ADCST0-ADCIT    ; STATUS0: ADC INACTIVE
         .BYTE    ADCST1-ADCIT    ; 1: INIT 1ST CHARGE
         .BYTE    ADCST2-ADCIT    ; 2: CHARGE, INIT 1ST MEASUREMENT
         .BYTE    ADCST3-ADCIT    ; 3: 1ST MEAS., INIT 2ND CHARGE
         .BYTE    ADCST4-ADCIT    ; 4: CHARGE, INIT 2ND MEASUREMENT
         .BYTE    ADCST3-ADCIT    ; 5: 2ND MEAS., INIT 3RD CHARGE
         .BYTE    ADCST6-ADCIT    ; 6: CHARGE, INIT 3RD MEASUREMENT
         .BYTE    ADCST3-ADCIT    ; 7: 3RD MEAS., INIT 4TH CHARGE
         .BYTE    ADCST8-ADCIT    ; 8: CHARGE, INIT 4TH MEASUREMENT
         .BYTE    ADCST3-ADCIT    ; 9: 4TH MEAS.
         .BYTE    ADCST0-ADCIT    ; 10: COMPLETED, NO ERROR
ADCERR   .BYTE    ADCST0-ADCIT    ; 11: ERROR OCCURED
```

```
; MEASUREMENT COMPLETED?                EN1FG = 1: YES, OK
;                                       RC2FG = 1: OVERFLOW BY BROKEN
;                                                  SENSOR

ADCST3    MOV.B    ADCST,R6          ; STATUS X 2
          RLA      R6                ; FOR RESULT ADDRESSING
          BIT.B    #EN1FG,&TPCTL     ; EN1 OR RC2FG?
          JNZ      L$401
          MOV.B    #ADCERR-ADCIT-1,ADCST     ; ERROR CODE-1 TO
                                             ; STATUS
          JMP      ADCCMPL                   ; SWITCH OFF ADC

L$401     MOV.B    &TPCNT1,MSTACK-6(R6)      ; STORE RESULT ON STACK
          MOV.B    &TPCNT2,MSTACK-5(R6)      ; HI BYTE

; IF LAST MEASUREMENT (ADCST = 9): SWITCH OFF ADC

          CMP.B    #9,ADCST
          JNE      ADCST1                    ; ADCST # 9: INIT NEXT
                                             ; MEAS.

ADCCMPL   CLR      &TPE                      ; OUTPUTS DISABLED
          CLR      &TPD                      ; ADC OFF, OUTPUTS LO
          JMP      L$402                     ; ADCST =10 AFTER
                                             ; RETURN

; CAPACITOR C CHARGE-UP FOR > 5 TAU. TP.2 TO TP.0 ARE USED

ADCST1    MOV.B    #(TPSSEL0*3)+ENB+ENA,&TPCTL   ; RESET FLAGS

          MOV.B    #B16+CPON+TP0+TP1+TP2,&TPD    ; SELECT OUTPUTS
          MOV.B    #TP0+TP1+TP2,&TPE         ; ENABLE CHARGE OUTPUTS
          MOV.B    #NN,&TPCNT2               ; LOAD NEG. CHARGE TIME
          JMP      L$402

; CHARGE IS MADE, INIT MEASUREMENT

ADCST8    MOV.B    #TP0,&TPE         ; ENABLE TP.0
          BIC.B    #TP0,&TPD         ; SET TP.0 LOW
          JMP      L$403

ADCST6    MOV.B    #TP1,&TPE         ; ENABLE TP.1
          BIC.B    #TP1,&TPD         ; SET TP.1 LOW
          JMP      L$403

ADCST4    MOV.B    #TP2,&TPE         ; ENABLE TP.2
          BIC.B    #TP2,&TPD         ; SET TP.2 LOW
          JMP      L$403
```

```
ADCST2    MOV.B    #TP3,&TPE          ; ENABLE TP.3
          BIC.B    #TP3,&TPD          ; SET TP.3 LOW
L$403     CLR.B    &TPCNT2            ; CLEAR HI BYTE TIMER
          CLR.B    &TPCNT1            ; CLEAR LO BYTE TIMER
L$402     INC.B    ADCST              ; ADCST + 1

ADCST0    BIC.B    #TPIFG,&IFG2       ; RESET ADC FLAG
          POP      R6                 ; RESTORE R6
          RETI

          .SECT    "INT_VECT",0FFEAH     ; INTERRUPT VECTORS
          .WORD    ADCINT             ; ADC INTERRUPT VECTOR;
```

Topics

Notes

10 Hints and Recommendations

10.1 Hints for Programmers

During the software development of the first MSP430 projects, a great deal of experience was acquired. The following hints and recommendations are intended for all programmers and system designers having more experience with 4- and 8-bit microcomputers than with 16-bit systems. Also mentioned are differences which the MSP430 family has when compared with other 16-bit architectures (e.g. the function of the carry bit as an inverted zero bit with some instructions).

- Bits to be used frequently should be located always in bit positions 0, 1, 2, 3, 7, 15. The first four bits can be set, reset and tested with constants coming from the Constant Generator (1, 2, 4, 8) and the last two ones can be tested easily with the conditional jump instructions JN and JGE:

```
TST.B     RSTAT              ; TEST BIT7 (OV <- 0)
JGE       BIT7LO             ; JUMP IF MSB OF BYTE IS 0

TST       MSTAT              ; TEST BIT15 (OV <- 0)
JN        BIT15HI            ; JUMP IF MSB OF WORD IS 1
```

- Use BCD arithmetic if simple up/down counters are used that are to be displayed. This saves time and ROM space due to unnecessary binary-BCD conversion.

EXAMPLE: Counter1 (4 BCD digits) is incremented, Counter2 (8 BCD digits) is decremented by one:

```
CLRC                         ; DADD ADDS CARRY BIT TOO!
DADD      #0001,COUNTER1     ; INCREMENT COUNTER1
                             ; DECIMALLY
CLRC
DADD      #9999,COUNTER2     ; DECREMENT 8 DIGIT COUNTER2
DADD      #9999,COUNTER2+2   ;DECIMALLY
```

- The Conditional Assembly feature of the MSP430 assembler allows obtaining more than one version out of one source. This reduces the effort to maintain software drastically: only one version needs to be updated if changes are necessary. See section "Conditional Assembly".
- Use bytes wherever appropriate. The MSP430 allows using every instruction with bytes. (exceptions are only SWPB, SXT and CALL)
- Use status bytes or words, not flags, for remembering states. This allows the extremely fast branching in one instruction to the appropriate handler. Otherwise a time (and ROM) consuming skip chain is necessary.

- Computing software. Use integer routines if speed is essential; use FPP if complex computing is necessary.

- Bit Test Instructions:
  With the bit handling instructions (BIS, BIT and BIC) more than one bit can be handled simultaneously: up to 16 bits can be handled inside one instruction.
  The BIS instruction is equivalent to the logical OR and can be used this way
  The BIC instruction is equivalent to the logical AND with the inverted source and can be used this way.

- Use of Addressing Modes:
  Use the Symbolic Mode for random accesses
  Use the Absolute Mode for fixed addresses like peripherals
  Use the Indexed Mode for random accesses in tables
  Use the Register Mode for time critical processing and as the normal mode
  Use assigned registers for extremely critical purposes: if a register contains always the same information, then it is not necessary to save it and to load it afterwards. The same is true for the restoring of the register when the task is done.

- Stack Operations:
  All items on the stack can be accessed directly with the Indexed Mode: this allows completely new applications compared with architectures that have only simple hardware stacks.
  The stack size is limited only by the available RAM, not by hardware register limitations.

Note: Handling the Stack

The above mentioned possibilities make careful "house keeping" necessary: every programme part which uses the stack has to ensure that only relevant information remains on the stack, and that all irrelevant data is removed. If this rule is not used consequently, the stack will overflow or underflow. If complex stack handling is used it is advisable to draw the stack with its items and the stack pointer as shown with the examples "Argument Transfer with Subroutine Calls" in the appendix.

- The Programme Counter PC can be accessed like every other register with all instructions and all addressing modes. Be very careful when using this feature! Do not use byte instructions when accessing the PC, due to the clearing of the upper byte when used.

- The Status Register SR can be accessed in register Mode only. Every status bit can be set or reset alone or together with other ones. This feature may be used for status transfer in subroutines.

- If highest possible speed is necessary for multiplications then two possibilities exist. Straight through programming: the effort used for the looping can be saved if the shifts and adds are programmed straight through. The routine ends at the known MSB of the multiplicand (here at bit 13 due to an ADC result [14 bits] that is multiplied):

```
; EXECUTION TIMES FOR REGISTER USE (CYCLES @ 1MHZ, 16 BITS):

; TASK              MACUF           EXAMPLE
;-------------------------------------------------------------
; MINIMUM  80       00000H X 00000H = 000000000H
; MEDIUM             96             0A5A5H X 05A5AH = 03A763E02H
; MAXIMUM 132       0FFFFH X 0FFFFH = 0FFFE0001H


; FAST MULTIPLICATION ROUTINE: PART USED BY SIGNED AND UNSIGNED
; MULTIPLICATION

MACUF     CLR       R6              ; MSBS MULTIPLIER

          RRA       R4              ; LSB TO CARRY
          JNC       L$01            ; IF ZERO: DO NOTHING
          ADD       R5,R7           ; IF ONE: ADD MULTIPLIER TO
                                    ; RESULT
          ADDC      R6,R8
L$01      RLA       R5              ; MULTIPLIER X 2
          RLC       R6              ;

          RRA       R4              ; LSB TO CARRY
          JNC       L$02            ; IF ZERO: DO NOTHING
          ADD       R5,R7           ; IF ONE: ADD MULTIPLIER TO
                                    ; RESULT
          ADDC      R6,R8
L$02      RLA       R5              ; MULTIPLIER X 2
          RLC       R6              ;
          ....                      ; SAME WAY FOR BITS 2 TO 12

          RRA       R4              ; LSB TO CARRY
          JNC       L$014           ; IF ZERO: DO NOTHING
          ADD       R5,R7           ; IF ONE: ADD MULTIPLIER TO
                                    ; RESULT
          ADDC      R6,R8
L$014     RET
```

- The following instructions have a special feature that is valuable during serial to parallel conversion: the carry acts as an inverted zero bit. This means that if the result of an operation is zero, then the carry is reset and vice versa. The instructions involved are:

  XOR, SXT, INV, BIT, AND.

  Without this feature a typical sequence for the conversion of an I/O-port bit to a parallel word would look like as follows:

```
        RLA     R5              ; FREE BIT 0 FOR NEXT INFO
        BIT     #1,&IOIN        ; PO.0 HIGH ?
        JZ      L$111
        INC     R5              ; YES, SET BIT 0
L$111   ...                     ; INFO IN BIT 0
```

  With this feature the above sequence is shortened to two instructions:

```
        BIT     #1,&IOIN        ; PO.0 HIGH ? .NOT.ZERO -> CARRY
        RLA     R5              ; SHIFT BIT INTO R5
```

- The carry bit can be used if increments by one are used:

  EXAMPLE: If the RAM word COUNT is greater than or equal to the value 1000 then a word COUNTER is to be incremented by one

```
        CMP     #1000,COUNT     ; COUNT >= 1000
        ADC     COUNTER         ; IF YES, CARRY = 1
```

- The carry bit can be added immediately. No conditional jumps are necessary for counters longer than 16 bits:

```
        ADD     R5,COUNT        ; LOW PART OF COUNT
        ADC     COUNT+2         ; MEDIUM PART
        ADC     COUNT+4         ; HIGH PART OF 48-BIT COUNTER
```

- "Fall Through" Usage: ROM space is saved if a subroutine call that is located immediately before a RET instruction is changed: The called subroutine is located after the instruction before the CALL and the programme falls through it. This saves 6 bytes of ROM: the CALL itself and the RET instruction. The I2C handler uses this mode.

```
; NORMAL WAY: SUBR2 IS CALLED, AFTERWARDS RETURNED
SUBR1   ...
        MOV     R5,R6
        CALL    #SUBR2          ; CALL SUBROUTINE
        RET
```

```
; "FALL THROUGH" SOLUTION: SUBR2 IS LOCATED AFTER SUBR1

SUBR1     ...
          MOV       R5,R6           ;GO TO SUBR2

SUBR2     ...                       ;START OF  SUBROUTINE SUBR2
          RET
```

10.2 Design Checklist

Several steps are necessary to complete a system consisting of an MSP430 and its peripherals with the necessary performance. Typical and recommended development steps are shown below. All of the tasks mentioned should be done carefully in order to prevent trouble later on.

1. Definition of the tasks to be performed by the MSP430 and its peripherals.
2. Worst case timing considerations for all of the tasks (interrupt timing, calculation times, I/O etc.).
3. Drawing of a complete hardware schematic. Decision which hardware options are used (Supply voltage, pull-downs at the I/O-ports ?)
4. Worst case design for all of the external components.
5. Organization of the RAM and if present of the EEPROM.
6. Flowcharting of the complete programme.
7. Coding of the software with an editor
8. Assembling of the programme with the ASM430 Assembler
9. Removing of the logical errors found by the ASM430 Assembler
10. Testing of the software with the SIM430 Simulator and EMU430 Emulator
11. Repetition of the steps 7 to 10 until the software is error free

10.3 Most frequently Occurring Software Errors

During software development the same errors are made by nearly all assembler programmers. The following list contains the errors most often heard of and experienced.

- A lack of "housekeeping" during stack operations: if items are removed from or placed onto the stack during subroutines or interrupt handlers, it is mandatory to keep track of these operations. Any wrong positioning of the stack pointer will lead to a programme crash due to wrong data which is written into the Programme Counter.

- Use of the wrong jump instructions: the conditional jump instructions JLO and JL, and JHS and JGE, respectively, give different results if used for numbers above 07FFFh. It is therefore necessary always to distinguish between signed and unsigned comparisons.

- Wrong completion instructions: dDespite their virtual similarity, subroutines and interrupt handlers need completely different actions when completed.
  Subroutines end with the RET instruction: only the address of the next instruction (the one following the subroutine call) is popped from the stack.
  Interrupt handlers end with the RETI instruction: two items are popped from the stack, first the Status Register is restored and afterwards the address (the address of the next instruction after the interrupted one) is popped from the stack to the Programme Counter.
  If RETI and RET are used wrongly, then a wrong item is written into the PC anyway. This means that the software will continue at random addresses and will therefore hang-up.

- Addition and subtraction of numbers with differently located decimal points: if numbers with virtual decimal points are used the addition or subtraction of numbers with different fractional bits leads to errors. It is necessary to shift one of the operands in a way to achieve equal fractional parts. See "Rules for the Integer Subroutines".

- Byte instructions applied to registers always clear the upper byte of the register. It is necessary therefore to use word instructions if operations in registers can exceed the byte range.

- Use of byte instructions with the Programme Counter as destination register: if the PC is the destination register, byte instructions do not make sense. The clearing of the PC's high byte is certainly wrong in any case. Instead a register should be used before the modification of the PC with the byte information.

- Use of falsely addressed branches and subroutine calls. The destination of branches and calls is used indirectly; this means the content of the destination is used as the address. These errors occur most often with the symbolic mode and the absolute mode:

```
CALL      MAIN      ;SUBROUTINE'S ADDRESS IS STORED IN MAIN
CALL      #MAIN     ;SUBROUTINE STARTS AT ADDRESS MAIN
```

  The real behaviour is seen easily when looking at the branch instruction. It is an emulated instruction, using the MOV instruction:

```
BR        MAIN      ;EMULATED INSTRUCTION BR
MOV       MAIN,PC   ;EMULATION BY MOV INSTRUCTION
```

  The addressing for the CALL instruction is exactly the same as for the BR instruction.

- If counters or timers longer than 16 bits are modified by the foreground (interrupt routines) and used by the background, it is necessary to disable the timer interrupt (most simple with the GIE bit in SR) during the reading of these words. If this is not done, the foreground can modify these words between the reading of two words. This would mean that one word contains the old value and the other one the modified one.

EXAMPLE: The timer interrupt handler increments a 32-bit timer. The background software uses this timer for calculations. The disabling of the interrupts avoids a timer interrupt that occurs between the reading of TIMLO and TIMHI falsifying the read information. This is the case if TIMLO overflows from 0FFFFh to 0000h during the interrupt routine: TIMLO was read with the old information 0FFFFh and TIMHI contains the new information x+1.

```
BT_HAN    INC       TIMLO           ;INCR. LO WORD
          ADC       TIMHI           ;INCR. HI WORD
          RETI
          ...

; BACKGROUND PART COPIES TIMXX FOR CALCULATIONS

          DINT                      ;GIE <- 0
          NOP                       ;DINT NEEDS 2 CYCLES
          MOV       TIMLO,R4        ;COPY LSDS
          MOV       TIMHI,R5        ;COPY MSDS
          EINT                      ;ENABLE INTERRUPT AGAIN
```

- When using sophisticated stack processing it is often overlooked that the PUSH instruction decrements the stack pointer first and moves the item afterwards.

EXAMPLE: The return address stored at TOS is to be moved one word down to free space for an argument.

```
          PUSH      @SP             ;WRONG! 1ST FREE WORD (TOS-2) IS
                                    ; COPIED
                                    ;ON ITSELF

          PUSH      2(SP)           ;CORRECT, OLD TOS IS PUSHED
```

Topics

Figures

Tables

Notes

Appendixes

A 1 CPU Registers and Features

All of the MSP430 CPU-registers can be used with all instructions.

A 1.1 The Programme Counter R0

One of the main differences to other microcomputer architectures relates to the Programme Counter (PC) that may be used as a normal register with the MSP430. This means that all of the instructions and addressing modes may be used with the Programme Counter too. For example, a branch is made by simply moving an address into the PC:

```
MOV     #LABEL,PC       ;JUMP TO ADDRESS LABEL
MOV     &LABEL,PC       ;JUMP TO ADDRESS CONTAINED
                        ; IN ADDRESS LABEL
MOV     @R14,PC         ;JUMP INDIRECT INDIRECT R14
```

Note: Use no Odd Address, if the Programme Counter is involved

The Programme Counter always points to even addresses: this means the LSB is always zero. The software has to ensure that no odd addresses are used if the Programme Counter is involved. Odd PC addresses will end up with non-predictable results.

A 1.2 Stack Processing

A 1.2.1 Usage of the Stack Pointer R1

The system stack pointer (SP) is a normal register like the other ones. This means it can use the same addressing modes. This gives good access to all items on the stack, not only to the one on the top of the stack.

The system stack pointer SP is used for the storage of the following items:
- Interrupt return addresses and Status Register contents
- Subroutine return addresses
- Intermediate results
- Variables for subroutines, floating point package etc.

When using the system stack, one should bear in mind that the microcomputer hardware uses the stack pointer for interrupts and subroutine calls too. To ensure the error free

running of the programme it is necessary to do exact "housekeeping" for the system stack.

Note: Use no Odd Address, if the Stack Pointer is involved

The Stack Pointer always points to even addresses: this means the LSB is always zero. The software has to ensure that no odd addresses are used if the Stack Pointer is involved. Odd SP addresses will end up with non-predictable results.

If bytes are pushed on the system stack, only the lower byte is used; the upper byte is not modified.

```
PUSH    #05H            ; 0005H -> TOS
PUSH.B  #05H            ; XX05H -> TOS
```

A 1.2.2 Software Stacks

Every register from R4 to R15 may be used as a software stack pointer. This allows independent stacks for jobs that have a need for this. Every part of the RAM may be used for these software stacks.

EXAMPLE: R4 is to be used as a software stack pointer.

```
MOV     #SW_STACK,R4    ;INIT. SW STACK POINTER
...
DECD    R4              ;DECREMENT STACK POINTER
MOV     ITEM,0(R4)      ;STORE ITEM ON STACK
...                     ;PROCEED
MOV     @R4+,ITEM2      ;POP ITEM FROM STACK
```

Software stacks may be organized as byte stacks. This is not possible for the system stack which always uses 16-bit words. The example shows R4 used as a byte stack pointer:

```
MOV     #SW_STACK,R4    ;INIT. SW STACK POINTER
...
DEC     R4              ;DECREMENT STACK POINTER
MOV.B   ITEM,0(R4)      ;STORE ITEM ON STACK
...                     ;PROCEED
MOV.B   @R4+,ITEM2      ;POP ITEM FROM STACK
```

A 1.3 Byte and Word Handling

Every word is addressable by three addresses as shown in Figure A1:

- The word address: an even address N
- The lower byte address: an even address N

- The upper byte address: an odd address N+1

If byte addressing is used, only the addressed byte is affected: no carry or overflow can affect the other byte.

> **Note: Byte Addressing and R0 to R15**
>
> Registers R0 to R15 do not have an address: they are treated in a special way. Byte addressing always uses the lower byte of the register; the upper byte is set to zero.

The way an instruction treats data is defined with its extension:
- The extension .B means byte handling
- The extension .W (or none) means word handling

Examples: The first two lines are equivalent. The 16-bit values, read in absolute address 050h, are added to the value in R5.

```
ADD       &050H,R5          ; ADD 16-BIT VALUE TO R5
ADD.W     &050H,R5          ; ADD 16-BIT VALUE TO R5
```

The 8-bit value, read in the lower byte of absolute address 050h, is added to the value contained in the lower byte of R5. The upper byte of R5 is set to zero. If the addressed byte 050h contains 078h, then R5 will contain 00078h afterwards, regardless of its former contents.

```
ADD.B     &050H,R5          ; ADD 8-BIT VALUE TO R5
```

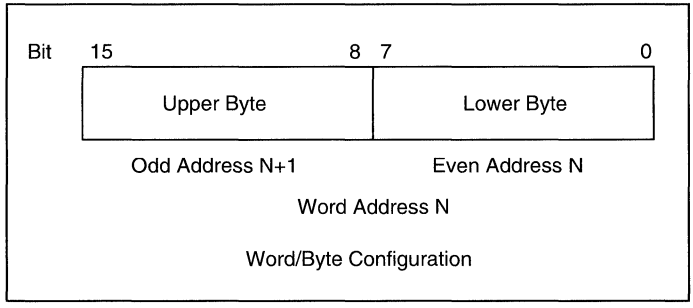


Figure A1: Word/Byte Configuration

If registers are used with byte instructions the upper byte of the destination register is always set to zero. It is necessary therefore to use word instructions if the range of calculations can exceed the byte range.

EXAMPLE: The two signed bytes OP1 and OP2 have to be added and the result stored in word OP3.

```
MOV.B     &OP1,R4          ; FETCH 1ST OPERAND
SXT       R4               ; CHANGE TO WORD FORMAT
MOV.B     &OP2,R5          ; SECOND OPERAND
SXT       R5
ADD.W     R4,R5            ; ADD WORDS
MOV.W     R5,&OP3          ; 16-BIT RESULT TO OP3
```

A 1.4 Constant Generator

A statistical look to the numbers used with the Immediate Mode shows that a few small numbers are in use most often. The six most often used numbers can be addressed with the four addressing modes of R3 (Constant Generator 2) and with the two not usable addressing modes of R2 (Status Register). The six constants that do not need an additional 16-bit word when used with the immediate mode are:

Number		Hexadecimal	Register	Ad
+0	Zero	(0000h)	R3	00
+1	positive one	(0001h)	R3	01
+2	positive two	(0002h)	R3	10
+4	positive four	(0004h)	R2	10
+8	positive eight	(0008h)	R2	11
-1	negative one	(FFFFh)	R3	11

Table A1: Constantsof the Constant Generator

The assembler inserts these ROM-saving addressing modes automatically if one of the above immediate constants is encountered. But only immediate constants are replaceable this way, not (for example) index values.
If an immediate constant out of the Constant Generator is used then the execution time is equal to the execution time of the Register Mode.
The most commonly used bits of the peripheral registers are, whenever possible, located in the bits addressable by the Constant Generator bits.

A 1.5 Addressing

The MSP430 allows seven addressing modes for the source operand, and four or five addressing modes for the destination. The addressing modes used are:

Address Bits	Source Modes	Destination Modes	Example
00	Register	Register	R5
01	Indexed	Indexed	TAB(R5)
01	Symbolic	Symbolic	TABLE
01	Absolute	Absolute	&BTCTL
10	Indirect	---	@R5
11	Ind. autoinkr.	---	@R5+
11	Immediate	----	#TABLE

Table A2: Addressing Modes

The three missing addressing modes for the destination operand are not of much concern for the programming:
Immediate Mode: Not necessary for the destination; immediate operands can always be placed into the source. Only in a very few cases will it be necessary to have two immediate operands in one instruction

Indirect Mode: if necessary the Indexed Mode with an index of zero is usable. For example:

```
ADD     #16,0(R6)       ; @R6 + 16 -> @R6
CMP     R5,0(SP)        ; R5 EQUAL TO TOS?
```

The second example above can be written in the following way saving 2 bytes of ROM:

```
CMP     @SP,R5          ; R5 EQUAL TO TOS? (R5-TOS)
```

Indirect Auto increment Mode: with table computing a method is usable that saves ROM-space, and also the number of registers used:

Example: The content of TAB1 is to be written into TAB2. TAB1 ends at the word preceding TAB1END.

```
        MOV     #TAB1,R5                ; INITIALIZE POINTER
LOOP    MOV.B   @R5+,TAB2-TAB1-1(R5)    ; MOVE TAB1 -> TAB2
        CMP     #TAB1END,R5             ; END OF TAB1 REACHED?
        JNE     LOOP                    ; NO, PROCEED
        ...                             ; YES, FINISHED
```

The above example uses only one register instead of two and saves three words due to the smaller initialization part. The normally written, longer loop is shown below

```
        MOV     #TAB1,R5        ;INITIALIZE POINTERS
        MOV     #TAB2,R6
LOOP    MOV.B   @R5+,0(R6)      ;MOVE TAB1 -> TAB2
        INC     R6
        CMP     #TAB1END,R5     ;END OF TAB1 REACHED?
        JNE     LOOP            ;NO, PROCEED
        ...                     ;YES, FINISHED
```

In other cases it may be possible to exchange source and destination operands to have the auto increment feature available for a pointer.

Each of the seven addressing modes has its own features and advantages:

Register Mode:
Fastest mode, least ROM requirements

Indexed Mode:
Random access to tables

Symbolic Mode:
Access to random addresses without overhead by loading of pointers

Absolute Mode:
Access to absolute addresses independent of current programme address

Indirect Mode:
Table addressing via register, code saving access to often referenced addresses

Indirect Autoincrement Mode:
Table addressing with code saving automatic stepping, for transfer routines

Immediate Mode:
Loading of pointers, 16-bit-constants within the instruction.

With the usage of the Symbolic Mode, an interrupt routine can be as short as possible. An interrupt routine is shown which has to increment a RAM word COUNTER, and to do a comparison if a status byte STATUS has reached the value 5. If this is the case, the status byte is cleared; otherwise, the interrupt routine terminates:

```
INTRPT  INC     COUNTER         ;INCREMENT COUNTER
        CMP.B   #5,STATUS       ;STATUS = 5?
        JNE     INTRET          ;
        CLR.B   STATUS          ;STATUS = 5: CLEAR IT
INTRET  RETI
```

No loading of pointers or saving and restoring of registers is necessary. What needs to be done is performed immediately without any overhead.

A 1.6 Programme Flow Control

A 1.6.1 Computed Branches and Calls

The Branch instruction is an emulated instruction which moves the destination address into the Programme Counter:

```
MOV     DST,PC          ; EMULATION FOR BR DST
```

The possibility to access the Programme Counter in the same way as all other registers gives interesting possibilities:
1. The destination address can be taken from tables
2. The destination address may be computed
3. The destination address may be a constant

A 1.6.2 Nesting of Subroutines

Thanks to the stack orientation of the MSP430, one of the main problems of other architectures does not play a role at all: subroutine nesting can proceed as long as RAM is available. There is no need to keep track of the subroutine calls as long as all subroutines terminate with a "Return from Subroutine" instruction. If subroutines are left without the RET instruction, then some housekeeping is necessary: popping of the return address or addresses from the stack.

A 1.6.3 Jumps

Jumps allow the conditional or unconditional leaving of the linear programme flow. The Jumps cannot reach every address of the address map, but they have the advantage of needing only one word and only two oscillator cycles. The 10-bit offset field allows Jumps of 512 words maximum in the forward direction, and 511 words maximum backwards. This is four times the normal reach of a Jump: only in few cases is the two word branch necessary.

Eight Jumps are possible with the MSP430. Four of them have two mnemonics, to allow better readability:

Mnemonic	Condition	Purpose
JMP label	Unconditional Jump	Programme control transfer
JEQ label	Jump if Z = 1	After comparisons
JZ label	Jump if Z = 1	Test for zero contents
JNE label	Jump if Z = 0	After comparisons
JNZ label	Jump if Z = 0	Test for non zero contents
JHS label	Jump if C = 1	After unsigned comparisons
JC label	Jump if C = 1	Test for set Carry
JLO label	Jump if C = 0	After unsigned comparisons
JNC label	Jump if C = 0	Test for reset Carry
JGE label	Jump if N .XOR. V = 0	
JL label	Jump if N .XOR. V = 1	
JN label	Jump if N = 1	Test for sign of a result

Table A3: Possible Jumps

Note: Conditional Jumps for Signed and Unsigned Data

It is important to use the appropriate conditional Jump for signed and unsigned data. For positive data (0 to 07FFFh and 0 to 07Fh) both signed and unsigned conditional jumps behave similiarly. This changes completely when used with negative data (08000h to 0FFFFh and 080h to 0FFh): the signed conditional jumps treat negative data as smaller numbers than the positive ones; the unsigned conditional jumps treat them as larger numbers than the positive ones.

No "Jump if Positive" is provided, only a "Jump if Negative". But after several instructions it is possible to use the "Jump if Greater Than or Equal" for this purpose. It must only be ensured that the instruction preceding the JGE resets the overflow bit V. The following instructions ensure this:

```
AND     SRC,DST         ; V <- 0
BIT     SRC,DST         ; V <- 0
RRA     DST             ; V <- 0
SXT     DST             ; V <- 0
TST     DST             ; V <- 0
```

If this feature is used it should be noted in the comment for later software modifications. For example:

```
MOV     ITEM,R7         ; FETCH ITEM
TST     R7              ; V <- 0, ITEM POSITIVE?
JGE     ITEMPOS         ; V=0: JUMP IF >= 0
```

Note: Only Unsigned Jumps are Adequate for Computed Addresses

If addresses are computed only the unsigned jumps are adequate: addresses are always unsigned, positive numbers.

A 2 Special Coding Techniques

A 2.1 Conditional Assembly

The Syntax for conditional assembly is described in detail in the *MSP430 Family Assembler Tools User's Guide*. Another example for conditional assembly is shown in the section *Software UART*.

Conditional assembly provides the possibility of compiling different lines of source into the object file, depending on the value of an expression that is defined in the source of the programme. This makes it easy to alter the behaviour of the code by modifying one single line in the source.

The following example shows how to use conditional assembly. The example will allow easy debugging of a programme that processes input from the ADC, by pretending that the input of the ADC is always 07FFh. The following is the routine used for reading the input of the ADC. It returns the value read from ADC input A0 in R8.

```
DEBUG     .SET      1              ;1= DEBUGGING MODE; 0= NORMAL MODE
ACTL      .SET      0114H
ADAT      .SET      0118H
IFG2      .SET      3
ADIFG     .SET      4

; GET_ADC_VALUE:

          .IF       DEBUG=1
          MOV       #07FFH,R8
          .ELSE
          BIC       #60,&ACTL      ; INPUT CHANNEL IS A0
          BIC.B     #ADIFG,&IFG2
          BIS       #1,&ACTL       ; START CONVERSION
WAIT      BIT.B     #ADIFG,&IFG2
          JZ        WAIT           ; WAIT UNTIL CONVERSION READY
          MOV       &ADAT,R8
          .ENDIF
          RET
```

With a little further refining of the code, better results may be achieved. The following piece of code shows more built-in ways to debug the code. The second 'debug code', where debug=2, returns 0700h and 0800h alternately.

```
DEBUG     .SET      1               ; 1= DEBUG MODE 1; 2= DEB. MODE 2;
0=                                  ; NORMAL MODE
ACTL      .SET      0114H
ADAT      .SET      0118H
IFG2      .SET      3
ADIFG     .SET      4

; GET_ADC_VALUE:

VAR       .SECT     "VAR"'0200H
OSC       .WORD     0700H

          .IF       DEBUG=1         ; RETURNING CONSTANT VALUE
          MOV      #07FFH,R8
          .ELSEIF   DEBUG=2         ; RETURNING ALTERNATING VALUE
          MOV       #0F00H,R8
          SUB       OSC,R8
          MOV       R8,OSC
          .ELSE
          BIC       #60H,&ACTL      ; INPUT CHANNEL IS A0
          BIC       #ADIFG,&IFG2
          BIS       #1,&ACTL        ; START CONVERSION
WAIT      BIT       #ADIFG,&IFG2
          JZ        WAIT            ; WAIT UNTIL CONVERSION READY
          MOV       &ADAT,R8
          .ENDIF
          RET
```

A 2.2 Position Independent Code

The architecture of the MSP430 allows the easy implementation of "Position Independent Code" (PIC). This is a code which may run anywhere in the address space of a computer, without any relocation being necessary. PIC is possible with the MSP430 mainly due to the allocation of the PC inside the register bank. Great use is made of the availability of the PC. Links to other PIC-blocks are possible only by references to absolute addresses (pointers).

EXAMPLE: Code is transferred to the RAM from an outside storage (EPROM, ROM, EEPROM) and executed there with full speed. This code needs to be PIC.

A 2.2.1 Referencing of Code Inside of PIC

The referenced code or data is located in the same block of PIC as that in which the programme resides.

Jumps
Jumps are anyway position independent: their address information is an offset to the destination address.

Branches

```
        ADD     @PC,PC          ;BRANCH TO LABEL DESTINATION
        .WORD   DESTINATION-$
```

Subroutine Calls
Calling a subroutine starting at the label SUBR:

```
SC      MOV     PC,RN           ;ADDRESS SC+2 -> AUX. REG
        ADD     #SUBR-$,RN      ;ADD OFFSET (SUBR - (SC+2))
        CALL    RN              ;SC+2+SUBR-(SC+2)) = SUBR
```

Operations on Data
The symbolic addressing mode is position independent: an offset to the PC is used. No special addressing is necessary

```
        MOV     DATA,RN         ;DATA IS ADDRESSED
        CMP     DATA1,DATA2     ;SYMBOLICALLY
```

Jump Tables
The status contained in Rstatus decides where the SW continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n). Range: +512 words, -511 words

```
        ADD     RSTATUS,PC      ;RSTATUS = (2X STATUS)
        JMP     STATUS0         ;CODE FOR STATUS = 0
        JMP     STATUS1         ;CODE FOR STATUS = 2
        ...
        JMP     STATUSN         ;CODE FOR STATUS = 2N
```

Branch Tables
The status contained in Rstatus decides where the SW continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n). Range: complete 64K

```
        ADD     TABLE(RSTATUS),PC   ;RSTATUS = STATUS
TABLE   .WORD   STATUS0-TABLE       ;OFFSET FOR STATUS = 0
        .WORD   STATUS1-TABLE       ;OFFSET FOR STATUS = 2
        ...
        .WORD   STATUSN-TABLE       ;OFFSET FOR STATUS = 2N
```

A 2.2.2 Referencing of Code Outside of PIC (Absolute)

The referenced code or data is located outside the block of PIC. These addresses can be absolute addresses only, e.g. for linking to other blocks and peripheral addresses.

Branches
Branching to the absolute address DESTINATION:

```
        BR      #DESTINATION    ;#DESTINATION -> PC
```

Subroutine Calls
Calling a subroutine starting at the absolute address SUBR:

```
        CALL    #SUBR           ;#SUBR -> PC
```

Operations on Data
Absolute mode (indexed mode with Reg = 0)

```
        CMP     &DATA1,&DATA2   ;DATA1 + 0 = DATA1
        ADD     &DATA1,RN
        PUSH    &DATA2          ;DATA2 -> STACK
```

Branch Tables
The status contained in Rstatus decides where the SW continues. Rstatus steps in increments of 2. Table is located in absolute address space:

```
        MOV     TABLE(RSTATUS),PC  ;RSTATUS = STATUS
        ...
        .SECT XXX ;TABLE IN ABSOLUTE ADDRESS
                                   ;SPACE
TABLE   .WORD   STATUS0            ;CODE FOR STATUS = 0
        .WORD   STATUS1            ;CODE FOR STATUS = 2
        ...
        .WORD   STATUSN            ;CODE FOR STATUS = 2N
```

Table is located in PIC address space, but addresses are absolute:

```
        MOV     RSTATUS,RHELP   ;RSTATUS CONTAINS STATUS
        ADD     PC,RHELP        ;STATUS + L$1 -> RHELP
L$1     ADD     #TABLE-L$1,RHELP  ;STATUS+L$1+TABLE-L$1
        MOV     @RHELP,PC       ;COMPUTED ADDRESS TO PC
TABLE   .WORD   STATUS0         ;CODE FOR STATUS = 0
        .WORD   STATUS1         ;CODE FOR STATUS = 2
        ...
        .WORD   STATUSN         ;CODE FOR STATUS = 2N
```

The above shown programme examples may be implemented as MACRO's if needed. This would simplify usage and improve transparency.

A 2.3 Reentrant Code

If the same subroutine is used by the background programme and interrupt routines, then two copies of this subroutine are necessary with normal computer architectures. The stack gives a method of programming that allows many tasks to use a single copy of the same routine. This ability of sharing a subroutine between several tasks is called "Reentrancy".

Reentrancy allows the calling of a subroutine despite the fact that the current task in use has not yet finished the subroutine.

The main difference between a reentrant subroutine and a normal one is that the reentrant routine contains only "pure code": that is, no part of the routine is modified during usage. The linkage between the routine itself and the calling software part is possible only via the stack i.e. all arguments during calling, and all results after completion, have to be placed on the stack and retrieved from there. The following conditions must be met for "Reentrant Code":

- No usage of dedicated RAM, only stack usage
- If registers are used, they need to be saved on the stack and restored from there.

EXAMPLE: A conversion subroutine "Binary to BCD" needs to be called from the background and the interrupt part. The subroutine reads the input number from TOS and places the 5-digit result also on TOS (two words): the subroutines save all used registers on the stack and restore them from there, or they compute directly on the stack.

```
        PUSH    R7              ; R7 CONTAINS THE BINARY VALUE
        CALL    #BINBCD         ; TO BE CONVERTED TO BCD
        MOV     @SP+,LSD        ; BCD-LSDS FROM STACK
        MOV     @SP+,MSD        ; BCD-MSD  FROM STACK
        ...
```

A 2.4 Recursive Code

Recursive subroutines are subroutines that call themselves. This is not possible with normal architectures: stack processing is necessary for this frequently used feature. A simple example of recursive code is a lineprinter handler that calls itself for inserting a "Form Feed" after a certain number of printed lines. This self-calling allows the use of all of the existing checks and features of the handler without the need to write them once more.

The following conditions must be met for "Recursive Code":

- No usage of dedicated RAM; only stack usage
- A termination item must exist to avoid infinite nesting (e.g. the lines per page must be greater than 1 with the above line printer example)
- If registers are used they need to be saved and restored on the stack

EXAMPLE: The line printer handler inserts a Form Feed after 70 printed lines

```
LPHAND  PUSH    R4              ; SAVE R4
        ...
        CMP     #70,LINES       ; 70 LINES PRINTED?
        JL      L$500           ; NO, PROCEED
        CALL    #LPHAND         ;
        .BYTE   CR,FF           ; YES, OUTPUT CARRIAGE RETURN
        ...                     ; AND FORM FEED
L$500   ...
```

A 2.5 Flag Replacement by Status Usage

Flags have several disadvantages if used for programme control:
- Missing transparency (flags may depend on other flags)
- Possibility of nonexistent flag combinations, if not handled very carefully
- Slow speed: the flags can only be tested serially

The MSP430 allows the use of a status (contained in a RAM byte or register) which defines the current programme part to be used. This status is very descriptive and prohibits "nonexistent" combinations. A second advantage is the high speed of the decision: one instruction only is needed to get to the start of the appropriate handler. See Branch Tables.

The programme parts that are used currently define the new status dependent on the actual conditions: normally the status is only incremented, but it may also change more randomly.

EXAMPLE: The status contained in register Rstatus decides where the software continues. Rstatus contains a multiple of 2 (0, 2, 4 ... 2n)

```
; RANGE: COMPLETE 64K

          MOV       TABLE(RSTATUS),PC   ;RSTATUS = STATUS
TABLE     .WORD     STATUS0             ; ADDRESS HANDLER FOR STATUS
                                        ; = 0

          .WORD     STATUS1             ; ADDRESS HANDLER FOR STATUS
                                        ; = 2
          ...
          .WORD     STATUSN             ; ADDRESS HANDLER FOR STATUS
                                        ; = 2N

STATUS0   ....                          ; START HANDLER STATUS 0
          INC       RSTATUS             ; NEXT STATUS IS 1
          JMP       HEND                ; COMMON END
```

The above solution has the disadvantage to use words even if the distances to the different programme parts are small. The next example shows the use of bytes for the branch table. The SXT instruction allows backward references (handler starts on lower addresses than TABLE4).

```
; BRANCH TABLES WITH BYTES: STATUS IN R5 (0, 1, 2, ..N)
; USABLE RANGE: TABLE4-128 TO TABLE4+126

          PUSH.B    TABLE4(R5)          ; STATUSX-TABLE4 -> STACK
          SXT       @SP                 ; FORWARD/BACKWARD REFERENCES
          ADD       @SP+,PC             ; TABLE4+STATUSX-TABLE4 -> PC
TABLE4    .BYTE     STATUS0-TABLE4      ; DIFFERENCE TO START OF HANDLER
          .BYTE     STATUS1-TABLE4
          ....
```

```
          .BYTE     STATUSN-TABLE4    ; OFFSET FOR STATUS = N
```

If only forward references are possible (normal case) the addressing range can be doubled. The following example shows this:

```
; STEPPING IS FORWARD ONLY (WITH DOUBLED FORWARD RANGE)
; STATUS IS CONTAINED IN R5 (0, 1, ..N)
; USABLE RANGE: TABLE5 TO TABLE5+254


          PUSH.B    TABLE5(R5)        ;STATUSX-TABLE -> STACK
          CLR.B     1(SP)             ; HI BYTE <- 0
          ADD       @SP+,PC           ;TABLE+STATUSX-TABLE -> PC
TABLE5    .BYTE     STATUS0-TABLE5    ;DIFFERENCE TO START OF HANDLER
          .BYTE     STATUS1-TABLE5
          ....
          .BYTE     STATUSN-TABLE5    ;OFFSET FOR STATUS = N
```

The above example can be made shorter and faster if a register can be used:

```
; STEPPING IS FORWARD ONLY (WITH DOUBLED FORWARD RANGE)
; STATUS IS CONTAINED IN R5 (0, 1, 2..N)
; USABLE RANGE: TABLE5 TO TABLE5+254

          MOV.B     TABLE5(R5),R6     ;STATUSX-TABLE5 -> R6
          ADD       R6,PC             ;TABLE5+STATUSX-TABLE5 -> PC
TABLE5    .BYTE     STATUS0-TABLE5    ;DIFFERENCE TO START OF HANDLER
          .BYTE     STATUS1-TABLE5
          ....
          .BYTE     STATUSN-TABLE5    ;OFFSET FOR STATUS = N
```

The addressable range can be doubled once more with the following code; the status (0, 1, 2, ..n) is doubled before its use.

```
; THE ADDRESSABLE RANGE MAY BE DOUBLED WITH THE FOLLOWING CODE:
; THE "FORWARD ONLY" VERSION WITH AN AVAILABLE REGISTER (R6) IS
; SHOWN: STATUS 0, 1, 2 ...N
; USABLE RANGE: TABLE6 TO TABLE6+510

          MOV.B     TABLE6(R5),R6        ;(STATUSX-TABLE6)/2
          RLA       R6                   ;STATUSX-TABLE6
          ADD       R6,PC                ;TABLE6+STATUSX-TABLE6 -> PC
TABLE6    .BYTE     (STATUS0-TABLE6)/2
          .BYTE     (STATUS1-TABLE6)/2
          ...
          .BYTE    (STATUSN-TABLE6)/2    ;OFFSET FOR STATUS = N
```

A 2.6 Argument Transfer with Subroutine Calls

Subroutines often have arguments to work with. Several methods exist for the passing of these arguments to the subroutine:
- On the stack
- In the words (bytes) after the subroutine call
- In registers
- Address is contained in the word after the subroutine call

The information passed may itself consist of numbers, addresses, counter contents, upper and lower limits etc. It depends only on the application.

A 2.6.1 Arguments on the Stack

The arguments are pushed on the stack and read afterwards by the called subroutine. The subroutine is responsible for the necessary housekeeping (here, the transfer of the return address to the top of the stack).

Advantages:
- Usable generally; no registers have to be freed for argument passing
- Variable arguments are possible

Disadvantages:
- Overhead due to necessary housekeeping
- Not easy to understand

EXAMPLE: The subroutine SUBR gets its information from two arguments pushed onto the stack before the calling. No information is given back; normal return from subroutine is used.

```
        PUSH    ARGUMENT0       ; 1ST ARGUMENT FOR SUBROUTINE
        PUSH    ARGUMENT1       ; 2ND ARGUMENT
        CALL    #SUBR           ; SUBROUTINE CALL
        ...
SUBR    MOV     4(SP),RX        ; COPY ARGUMENT0 TO RX
        MOV     2(SP),RY        ; COPY ARGUMENT1 TO RY
        MOV     @SP,4(SP)       ; RETURN ADDRESS TO CORRECT LOC.
        ADD     #4,SP           ; PREPARE SP FOR NORMAL RETURN
        ...                     ; PROCESSING OF DATA
        RET                     ; NORMAL RETURN
```

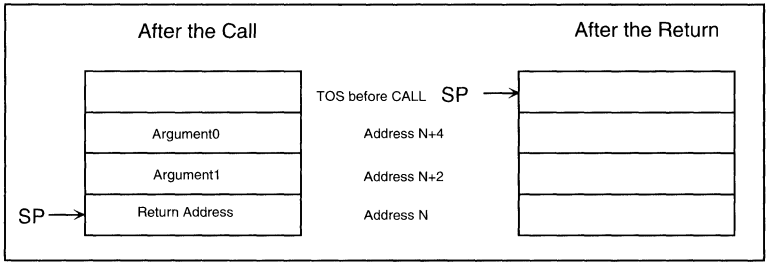


Figure A2: Arguments on the Stack

EXAMPLE: The subroutine SUBR gets its information from two arguments pushed onto the stack before the calling. Three result words are returned on the stack: it is the responsibility of the calling programme to pop the results from the stack.

```
        PUSH    ARGUMENT0       ; 1ST ARGUMENT FOR SUBROUTINE
        PUSH    ARGUMENT1       ; 2ND ARGUMENT
        CALL    #SUBR           ; SUBROUTINE CALL
        POP     R15             ; RESULT2 -> R15
        POP     R14             ; RESULT1 -> R14
        POP     R13             ; RESULT0 -> R13
        ...
SUBR    MOV     4(SP),RX        ; COPY ARGUMENT0 TO RX
        MOV     2(SP),RY        ; COPY ARGUMENT1 TO RY
        ...                     ; PROCESSING CONTINUES
        PUSH    2(SP)           ; SAVE RETURN ADDRESS
        MOV     RESULT0,6(SP)   ; 1ST RESULT ON STACK
        MOV     RESULT1,4(SP)   ; 2ND RESULT ON STACK
        MOV     RESULT2,2(SP)   ; 3RD RESULT ON STACK
        RET
```

After the subroutine call and the RET, the stack looks as follows:

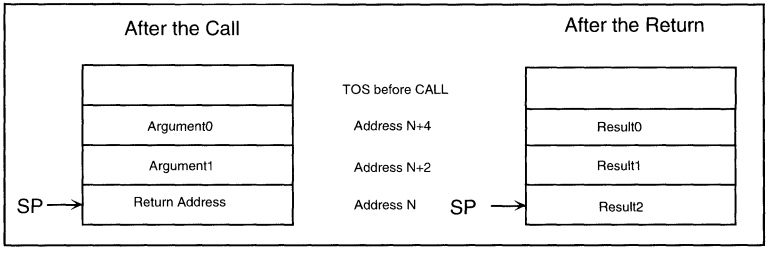


Figure A3: Arguments on the Stack

Note: Only Data at or above the Top Of Stack is Protected Against Overwriting

If the stack is involved during data transfers it is very important to have in mind that only data at or above the top of stack (TOS, the word the SP points to) is protected against overwriting by enabled interrupts. This does not allow moving the SP above the last item on the stack; indexed addressing is needed instead.

A 2.6.2 Arguments following the Subroutine Call

The arguments follow the subroutine call and are read by the called subroutine. The subroutine is responsible for the necessary housekeeping; here, the adaptation of the return address on the stack to the 1st word after the arguments.

Advantages:
- Very clear and good readable interface

Disadvantages:
- Overhead due to necessary housekeeping
- Only fixed arguments possible

EXAMPLE: The subroutine SUBR gets its information from two arguments following the subroutine call. Information can be given back on the stack or in registers.

```
        CALL    #SUBR           ; SUBROUTINE CALL
        .WORD   START           ; START OF TABLE
        .BYTE   24,0            ; LENGTH OF TABLE, FLAGS
        ...
SUBR    MOV     @SP,R5          ; COPY ADDRESS 1ST ARGUMENT TO R5
        MOV     @R5+,R6         ; MOVE 1ST ARGUMENT TO R6
        MOV     @R5+,R7         ; MOVE ARGUMENT BYTES TO R7
        MOV     R5,0(SP)        ; ADJUST RETURN ADDRESS ON STACK
        ...                     ; PROCESSING OF DATA
        RET                     ; NORMAL RETURN
```

A 2.6.3 Arguments in Registers

The arguments are moved into defined registers and used afterwards by the subroutine.

Advantages:
- Simple interface and easy to understand
- Very fast
- Variable arguments are possible

Disadvantages:
- Registers have to be freed

EXAMPLE: The subroutine SUBR gets its information inside two registers which are loaded before the calling. Information can be given back or not, with the same registers.

```
        MOV     ARG0,R5         ; 1ST ARGUMENT FOR SUBROUTINE
        MOV     ARG1,R6         ; 2ND ARGUMENT
        CALL    #SUBR           ; SUBROUTINE CALL
        ...
SUBR    ...                     ; PROCESSING OF DATA
        RET                     ; NORMAL RETURN
```

A 2.7 Interrupt Vectors in RAM

If the destination address of an interrupt changes with the programme run it is valuable to have the possibility to modify the pointer. The vector itself (which resides in ROM) is not changeable but a second pointer residing in RAM may be used for this purpose:

EXAMPLE: The interrupt handler for the Basic Timer starts at location BTHAN1 after initialization and at BTHAN2 when a certain condition is met (for example calibration is made).

```
; BASIC TIMER INTERRUPT GOES TO ADDRESS BTVEC. THE INSTRUCTION
; "MOV @PC,PC" WRITES THE ADDRESS IN BTVEC+2 INTO THE PC: PROGRAM
; CONTINUES AT THAT ADDRESS

        .SECT   "VAR",0200H     ; RAM START
BTVEC   .WORD   0               ; OPCODE "MOV @PC,PC"
        .WORD   0               ; ACTUAL HANDLER START ADDR.

; THE SOFTWARE VECTOR BTVEC IS INITIALIZED:

INIT    MOV     #04020H,BTVEC   ; OPCODE "MOV @PC,PC
        MOV     #BTHAN1,BTVEC+2 ; START WITH HANDLER BTHAN1
        ...                     ; INITIALIZATION CONTINUES

; THE CONDITION IS MET: THE BASIC TIMER INTERRUPT IS HANDLED
; AT ADDRESS BTHAN2 STARTING NOW

        MOV     #BTHAN2,BTVEC+2 ; CONT. WITH ANOTHER HANDLER
        ...

; THE INTERRUPT VECTOR FOR THE BASIC TIMER CONTAINS THE RAM
; ADDRESS OF THE SOFTWARE VECTOR BTVEC:

        .ORG    0FFE2H          ; VECTOR ADDRESS BASIC TIMER
        .WORD   BTVEC           ; FETCH ACTUAL VECTOR THERE
```

A 3 References

MSP430 Family Architecture Guide and Module Library	1994
MSP430 Family Assembly Language Tools User's Guide	1994
MSP430 Family Metering User's Guide	1994
The Art of Electronics, Cambridge University Press	1989

Notes

TI SC Sales Offices in Europe

Belgium
Texas Instruments S.A./N.V.
Brussels
Tel.: (02) 7 26 75 80
Fax: (02) 7 26 72 76

Finland
Texas Instruments OY
Espoo
Tel.: (0) 43 54 20 33
Fax: (0) 46 73 23

France,
Middle-East & Africa
Texas Instruments
Velizy Villacoublay
Tel.: (1) 30 70 10 01
Fax: (1) 30 70 10 54

Germany
Texas Instruments GmbH
Freising
Tel.: (0 81 61) 80-0
Fax: (0 81 61) 80 45 16

Hannover
Tel.: (05 11) 90 49 60
Fax: (05 11) 6 49 03 31

Ostfildern
Tel.: (07 11) 3 40 30
Fax: (07 11) 3 40 32 57

Holland
Texas Instruments B.V.
Amstelveen
Tel.: (0 20) 6 40 04 16
Fax: (0 20) 5 45 06 60
(0 20) 6 40 38 46

Hungary
TI Representation:
Budapest
Tel.: (1) 1 76 37 33
Fax: (1) 2 02 62 56

Italy
Texas Instruments S.p.A.
Agrate Brianza (Mi)
Tel.: (0 39) 6 84 21
Fax: (0 39) 6 84 29 12

Republic of Ireland
Texas Instruments Ltd.
Dublin
Tel.: (01) 4 75 52 33
Fax: (01) 4 78 14 63

Spain
Texas Instruments S.A.
Madrid
Tel.: (1) 3 72 80 51
Fax: (1) 3 72 82 66

Sweden
Texas Instruments
International Trade Corporation
Kista
Tel.: (08) 7 52 58 00
Fax: (08) 7 51 97 15

United Kingdom
Texas Instruments Ltd.
Northampton
Tel.: (0 16 04) 66 30 00
Fax: (0 16 04) 66 30 01

TEXAS INSTRUMENTS

TI Technology Centres

France
Texas Instruments
Velizy Villacoublay
Tel.: Standard:
(1) 30 70 10 01
Technical Service:
(1) 30 70 11 33

Holland
Texas Instruments B.V.
Amstelveen
Tel.: (0 20) 5 45 06 00
Fax: (0 20) 6 40 38 46

Italy
Texas Instruments S.p.A.
Agrate Brianza (Mi)
Tel.: (0 39) 6 84 21
Fax: (0 39) 6 84 29 12

Sweden
Texas Instruments
International Trade Corporation
Kista
Tel.: (08) 7 52 58 00
Fax: (08) 7 51 97 15

European SC Information Centre

Telephone:
Dutch (33) 1 30 70 11 66
English (33) 1 30 70 11 65
French (33) 1 30 70 11 64
German (33) 1 30 70 11 68
Italian (33) 1 30 70 11 67

Fax: (33) 1 30 70 10 32

Printed in Germany
by Sellier Druck, Angerstr. 54, 85354, Freising

Chlorine free paper – to protect our environment

Texas Instruments

SLAUE11