Metering Application Report MSP430

AUTHOR:	Lutz Bierl
VERSION:	2.1.00
DATE:	Jan. 96

CONTENTS

LIST OF ILLUSTRATIONS

1 INTRODUCTION

The MSP430 is a 16-bit microcomputer having special features not commonly available with other microcomputers:

- Complete system on chip (LCD, ADC, I/O, ROM, RAM, Watchdog, UART, Basic Timer)
- Extremely low power consumption: only 4.2 nWs/instruction max.
- High speed (300 ns/instruction @ 3.3 MHz with register, register mode)
- RISC structure (27 instructions)
- Orthogonal architecture (any instruction with any addressing mode)
- Seven addressing modes for source operand
- Four (five) addressing modes for destination operand
- Constant generator for the most commonly used constants (-1, 0, 1, 2, 4, 8)
- Only one crystal necessary due to Frequency Locked Loop (FLL)
- Stable MCLK frequency reached after 6 clocks when woken-up from Low Power Mode 3

These features make it very easy to program the MSP430 in assembler or in C-language. For example, despite the low instruction count of only 27, the MSP430 is capable of emulating almost the complete instruction set of the legendary PDP11.

NOTES

> It is advised to have the "MSP430 Architecture User's Guide and Module Library" readily available. This book contains valuable information, illustrations and a detailed description of the MSP430 hardware.
>
> Additionally the "MSP430 Software User's Guide" is recommended. It contains further information regarding the instruction set, besides other more common software information.
>
> All the examples given refer to the "MSP430 Family User's Guide" Revision 0.44 dated 23.12.93. It can not be guaranteed that new revisions will behave exactly in the same manner as described in this User's Guide. See Important Note above.

1.1 Notation

The following abbreviations and special notations are used:

R4|R3	32-bit number. MSB in R4, LSB in R3
AGND	Ground connection for the Analog-to-Digital Converter (V_{SS} resp. AV_{SS})
.or.	Logical Or function
.and.	Logical And function
.xor.	Logical Exclusive Or function
.not.	Logical Inversion

src	Source (location where data is read from)
dst	Destination (location where data is written to)
SP	Stack Pointer
PC	Program Counter
TOS	Top of Stack (data word the Stack Pointer SP points to)
MSB	Most significant bit (or byte)
LSB	Least significant bit (or byte)
DCO	Digitally Controlled Oscillator
BCD	Binary Coded Decimal (numbers 0 to 9 coded by 4 bits)

1.2 The MSP430 Versions

The MSP430 family currently consists of two types:

1. The MSP430x32x
2. The MSP430x31x

Both types are described in depth in the "MSP430 Family Architecture Guide and Module Library". The only differences between the two members are listed below:

Hardware Item	MSP430x32x	MSP430x31x
LCD Select lines	21	23
ADC Measurement Principle	Successive Approximation	Capacity discharge
Package	64QFP	56SSOP

If not mentioned otherwise, the examples and explanations are valid for both members.

1.3 The Operating Modes used for Metering Applications

The MSP430 metering applications fall into two main classes depending on the power supply:

- Electricity meters that are powered from the mains. The micro computer needs to be active all the time, but due to the low current consumption of the MSP430 (max. 1.4 mA (@ 5 V) this is not a problem, despite the need for low power consumption (system consumption < 40 mA).
- Battery driven applications such as gas meter, water flow meter, heat volume counters etc. For these applications the power consumption plays an overwhelming role because these applications have to run from one battery for more than 10 years. The current drawn by the MSP430 needs to be in the range of the self discharge current of the battery, which means 4 to 6 µA.

The MSP430 offers six operating modes, with different current consumption. Three of them are important for battery-driven applications:

1. The Active Mode with running CPU.

2. The Low Power Mode 3: the normal mode for all applications during 99% to 99.9% of the time. This mode is also called Done Mode or Sleep Mode.
3. The Low Power Mode 4: the mode used during storage times. This mode is also called Off Mode.

1.3.1 The Active Mode

This mode is used for calculations, decisions and other activities that make a running CPU necessary. All of the peripherals may be used provided that they are enabled. All of the examples shown in this guide use the Active Mode.

1.3.2 The Low Power Mode 3

The most important mode for all battery driven applications. The CPU is disabled, but enabled peripherals stay active: LCD driver, Basic Timer, I/O-ports, 8-bit Timer. The running Basic Timer allows a precise time base. Enabled interrupts wake-up the CPU, switch on MCLK and start normal operation. The next figure shows the status of the complete MSP430 system when in Low Power Mode 3 (LPM3):

Active	Not Active
RAM	CPU
ACLK	MCLK
32768 Hz Oscillator	Disabled Peripherals
LCD Driver (if enabled)	Disabled Interrupts
Basic Timer (if enabled)	FLL
I/O-Port	
8-bit Timer	
Enabled Peripherals	
RESET Logic	

To enter the Low Power Mode 3 the following code is necessary:

```
;
; Definitions for the Operating Modes
;
GIE       .EQU     008h             ; General Interrupt enable in SR
CPUOFF    .EQU     010h             ; CPU off bit in SR
OSCOFF    .EQU     020h             ; Oscillator off bit in SR
SCG0      .EQU     040h             ; System Clock Generator Bit 0
SCG1      .EQU     080h             ;
;
; Enter Low Power Mode 3, enable interrupts
;
          BIS      #CPUOFF+GIE+SCG1+SCG0,SR  ; Enter LPM3
;
```

After the completion of the interrupt routine the software returns to the instruction that set the CPUoff bit. The normal wake-up for the LPM3 comes from the Basic Timer: it is programmed to wake-up the CPU at regular intervals (ranging from 0.5 Hz to 64 Hz or higher) to maintain a software timer. This software timer controls all necessary system activities.

EXAMPLE: The MSP430 system runs normally in LPM3. The enabled interrupt of the Basic Timer wakes-up the system every second. If one minute elapses, measurements are made and afterwards the system returns to LPM3.

```
;
; Interrupt handler for Basic Timer: Wake-up with 1 Hz
;
BT_HAN   INC.B    SECCNT               ; Counter for seconds +1
         CMP.B    #60,SECCNT           ; 1 minute elapsed?
         JHS      MIN1                 ; Yes, do necessary tasks
         RETI                          ; No return to LPM3
;
; One minute elapsed: Return is removed from stack, a branch to
; the necessary tasks is made. There it is decided how to proceed
;
MIN1     INC      MINCNT               ; Minute counter +1
         CLR      SECCNT               ; 0 -> SECCNT
         ADD      #4,SP                ; House keeping: SR, PC off Stack
         BR       #TASK                ; Do tasks
         ...
;
TASK     ...                           ; Start of necessary tasks
;
; All measurements and calculations are made: Return to LPM3
;
         BIS      #CPUOFF+GIE+SCG0+SCG1,SR  ; Enter LPM3
```

The Low Power Mode 3 is the mode with the lowest current consumption that allows the use of a real time clock: the Basic Timer can interrupt the LPM3 at relatively long time intervals (up to 2 s) and update the real time clock. If the Status Register is not changed during the interrupt routines then the RETI instruction returns to the instruction that set the CPUoff bit (and moved the CPU into LPM3). The Program Counter points to the next instruction but this instruction is not executed unless the interrupt routine resets the CPUoff bit during its run.
If woken up from LPM3 two additional cycles are needed until the PC is loaded with the interrupt vector address and the interrupt handler is started (8 cycles instead of 6 when in Active Mode).

EXAMPLE: The MSP430 system runs normally in LPM3. The enabled interrupt of the Basic Timer wakes-up the system every second. If one minute is over, measurements are made and afterwards the system returns to LPM3. The branch to the task is made by resetting the CPUoff bit inside the interrupt routine.

```
; Interrupt handler for Basic Timer: Wake-up with 1 Hz
;
BT_HAN   INC.B    SECCNT               ; Counter for seconds +1
         CMP.B    #60,SECCNT           ; 1 minute over?
         JHS      MIN1                 ; Yes, do necessary tasks
         RETI                          ; No return to LPM3
;
; One minute elapsed: CPUoff is reset, the program continues
; after the instruction that set the CPUoff bit
;
MIN1     CLR      SECCNT               ; 0 -> SECCNT
         INC      MINCNT               ; Minute counter + 1
         BIC      #CPUOFF,0(SP)        ; Reset CPUoff-bit to continue
         RETI                          ; at DONE+2
;
```

```
; Background part: Return to LPM3
;
DONE      BIS       #CPUOFF+GIE+SCG0+SCG1,SR   ; Enter LPM3
;
; Program continues here if CPUoff bit was reset inside of the
; Basic Timer Handler.
;
TASK      ...                            ; Tasks made every minute
          JMP       DONE                 ; Back to LPM3
```

1.3.3 The Low Power Mode 4

The Low Power Mode 4 (LPM4) is used if the lowest supply current is necessary or if no timing is needed or desired (no change of the RAM content allowed). This is normally the case for storage times preceding or following the calibration process. The next figure shows the status of the complete MSP430 system when in LPM4:

Active	Not Active
RAM	CPU
I/O-Port	MCLK
Enabled Interrupts	ACLK
RESET Logic	FLL
	Disabled Peripherals
	Disabled Interrupts
	Watchdog
	Timers

When woken-up the software has to decide if it is necessary to enter the LPM4 again (if the wake-up was caused by EMI e.g.) or if one of the other operating modes is to be entered. To ensure this decision a code can be given to a port that can be checked by the MSP430 software: only if this code is present is the Active Mode entered.

To enter the Low Power Mode 4 the following code is necessary:

```
;
; Enter LPM4, enable GIE
;
        BIS     #CPUOFF+OSCOFF+GIE+SCG1+SCG0,SR
;
```

The way out of the LPM4 is principally the same as shown with LPM3. The software of the interrupt handler has to decide if the CPU stays active or if a return to a low power mode is necessary.

When entering the LPM4 the information in the control registers of the System Clock Frequency Integrator SCFI0 and SCFI1 remains stored. If at this time the ambient temperature is high, the register SCFI1 contains a relatively high value to compensate the negative temperature dependence of the DCO. If the LPM4 is left afterwards with a very low ambient temperature then it is possible that the resulting DCO frequency is outside of the oscillator's range. Therefore it is a good programming practice to set the System Clock Frequency Integrator to a low value before entering the LPM4.

```
;
         CLRC                                    ; Ensure new MSB is 0
         RRC       &SCFI1                        ; Use halved tap number
         BIC       #CPUOFF+OSCOFF+GIE,SR         ; Enter LPM4, enable GIE
```

1.4 Use of the System Clock Generator

The System Clock Generator of the MSP430 family allows a lot of features not available with other microcomputers. To allow the full use of all the possibilities some basics concerning the function of the oscillator are needed. A detailed description of the hardware is given in the "MSP430 Family Architecture User's Guide and Module Library"; see chapter 6 "Oscillator and System Clock Generator".
The output frequency MCLK of the System Clock Generator is generated in the Digitally Controlled Oscillator (DCO), having 32 "taps". Each of these taps represents an output frequency ranging from 500 kHz to 4 MHz typically. These tap frequencies depend on temperature and supply voltage and referencing to a crystal is necessary therefore.

```
; Software definitions for the programming examples
;
SCG1      .equu     080h      ; System Clock Generator Control Bit 1
SCG0      .equ      040h      ; System Clock Generator Control Bit 0
OSCoff    .equ      020h      ; If 1: Oscillator off
CPUoff    .equ      010h      ; If 1: CPU off
GIE       .equ      008h      ; General Interrupt Enable Bit
SCFI0     .equ      050h      ; System Clock Frequency Integrator Reg.
FN_2      .equ      004h      ; DCO current switch for 2 x fnom
SCFI1     .equ      051h      ; DCO tap register 2^9 to 2^2
TAP       .equ      008h      ; 2^5 bit in SCFI1
SCFQCTL   .equ      052h      ; System Clock Frequency Control Register
M         .equ      080h      ; Modulation Bit in SCFQCTL
;
```

1.4.1 Initialization

After the applying of the supply voltage V_{CC} the system clock frequency f_{SYSTEM} is initialized to 1.024 MHz. This is automatically made by setting of the multiplication factor N to 32 and clearing of the FN_x bits. If the CPU is always on afterwards and 1.024 MHz is the wished frequency, nothing else is to do.

1.4.1.1 First Setting of the DCO Taps during Initialization

The Digitally Controlled Oscillator of the MSP430 starts at the tap 0, which means at the lowest possible frequency. To get from one tap to the next one, 2^{10} (1024) cycles are needed. Thirty-two taps are implemented, so 32 x 1024 cycles are needed worst case to get to the correct DCO tap. The initialization routine should have a length of 32000 cycles therefore. If this is not the case a delay routine should be added to guarantee this length. An example is given below:

```
;
INIT     ...                      ; Loop Control is on (SCG1 = SCG0 = 0)
         MOV       #11000,R4           ; Init delay to allow DCO setting
L$1      DEC       R4                  ; 11000 x 3 cycles = 33000 cycles
         JNZ       L$1                 ;
         BR        #MAINLOOP           ; Start program
;
```

1.4.2 Entering of Low Power Mode 3

The Low Power Mode 3 (LPM3) (crystal on, DCO and loop control off) is the normal mode for battery driven systems. Enabled interrupts (e.g. the Basic Timer) wake-up the CPU. LPM3 is entered with the following source code:

```
;
        BIS     #CPUoff+GIE+SCG1+SCG0,SR            ; Enter LPM3
;
```

1.4.3 Wake-up from Interrupts in Low Power Mode 3

Wake-up from LPM3 clears only bit SCG1. Due to the set bit SCG0 the loop control of the DCO is off. Normal interrupt routines are too short to allow the loop control to adjust the DCO tap: 1024 cycles are necessary to get from one tap to the other one. It is not necessary therefore to switch on the loop control. The CPU uses the DCO tap set during the last adaptation. A normal, short interrupt routine looks this way:

```
;
BT_HAND INC     COUNTER                 ; Loop Control stays off:
        RETI                            ; DCO is on for 15 cycles only
;
```

If woken-up from LPM3 the interrupt latency time (6 cycles) is increased by typ. 2 µs @ 1 MHz resp. 1 µs @ 2 MHz (if FN_2 = 1), this means 8 cycles are needed typically from the interrupt event to the start of the interrupt handler. The time the DCO needs to settle to the nominal frequency is 4 cycles typically.

1.4.4 Adaptation of the DCO Tap during Calculations

The DCO tap of the System clock generator should be updated during longer on-times of the CPU (e.g. during calculations). This is necessary especially if accurate timing of the instructions is needed. During all calculations that exceed 100 cycles in length the loop control of the DCO should be switched on. The way to do this is to reset the SCG0 bit in the Status Register after the wake-up:

```
;
; Calculations are to be made. Allow adaptation of the DCO tap
;
        BIC     #SCG0,SR ; Switch on DCO loop control
        ...              ; Calculate energy (>100 cycles)
        RETI             ; Return to LPM3 with adapted DCO tap
;
```

The RETI instruction restores the CPU mode from the stack as it was when the interrupt occurred.

1.4.5 Wake-up from Interrupts in Low Power Mode 4

The Low Power Mode 4 normally lasts much longer than the Low Power Mode 3: it may last up to months until a stored module is woken-up for calibration. This means that the environment temperature may have changed seriously. If the LPM4 was entered at a high temperature, the used DCO tap will be a relatively high one due to the negative temperature coefficient of the DCO. If then the device is woken-up at a low temperature and the crystal turns on fast, this high DCO tap may lead to a very high DCO frequency the system cannot operate with. Therefore it is a good programming practice, to program a low DCO tap before entering LPM4:

```
;
; Enter Low Power Mode 4: Set DCO tap to 2
;
         MOV.B     #TAP*2,&SCFI1                      ; Set DCO tap to 2
         BIS       #CPUoff+OSCoff+GIE+SCG1+SCG0,SR         ; Enter LPM4
;
```

If woken-up from LPM4 it may last up to seconds until the crystal has reached its nominal frequency. The frequency integrator counts down continuously as long as the crystal oscillator has not started its operation. This lasts until the lowest DCO tap (with the lowest system frequency) is reached. After the start of the crystal oscillator the loop control will set the system frequency to its correct value.

1.4.6 Change of the System Frequency

The system clock frequency f_{system} depends on two values:

$$f_{system} = N \times f_{crystal}$$

with: N Multiplication factor of the DCO loop

$f_{crystal}$ Frequency of the crystal (normally 32768 Hz)

The normal way to change the system clock frequency is to change the multiplication factor N. The System Clock Frequency Control register SCFQCTL is loaded with (N-1) to get the new frequency. To allow the DCO to work always in one of the centered taps (13 to 18), which gives a security not to be at the frequency limits of the DCO, three switches FN_2 to FN_4 are implemented in the register SCFI0. These switches increase the internal current of the DCO and allow higher output frequencies if set. The switch nearest to the programmed DCO output frequency should be used.

The switches FN_x settle typically within ±1 tap if the change is from the nominal frequency of one switch to the nominal frequency of the other one. For example if in the example below the initial system frequency is 1 MHz, then the new tap is one of the neighboring taps. This means, to settle at 2 MHz needs maximum 1024 cycles (0.5 ms) only. If FN_2 is not used, it would take up to 16 x 1024 cycles (8 ms) because the misalignment could be up to 16 taps.

```
;
; Change system frequency to 2.048 MHz (fcrystal = 32 kHz)
; N = 64   : Multiply 32 kHz by 64 to get 2.048 MHz
; FN_2 = 1: Adjust DCO current to 2 MHz
; M = 0    : Switch on modulation
;
         MOV.B    #64-1,&SCFQCTL     ; 64 x 32 kHz = 2.048 MHz
         MOV.B    #FN_2,&SCFI0       ; Adjust DCO current to 2 MHz
;
```

1.4.7 Use of the Modulation Bit M

The modulation bit M switches off and on the influence of the 5 LSBs of the System Clock Frequency Integrator:

M = 0: the modulation is on, this means all 10 bits of the integrator influence the DCO. The used tap of the DCO may be changed with every clock cycle to get the correct system clock frequency. This is the case if the programmed frequency lies exactly between two tap frequencies.

M = 1: the modulation is off, this means only the 5 MSBs of the integrator influence the DCO. The used tap of the DCO is changed only after 1024 clock cycles to get the correct system clock frequency. If the programmed frequency lies exactly between two tap frequencies, then 1024 cycles are output with the lower tap and 1024 cycles are output with the upper tap.

In any case, independent of the modulation status, the integral error of the DCO will be zero.

The modulation may be switched off if a series of MCLK cycles is needed with exactly the same length. To get this the loop control needs to be switched off too.

```
;
; Ensure stable, non regulated output pulses with equal length:
;
          BIS.B     #SCG0,SR            ; Switch off loop control
          BIS.B     #M,&SCFQCTL         ; Switch off modulation
          ...                           ; Use non-regulated MCLK
;
; Return to regulated MCLK
;
          BIC.B     #SCG0,SR ; Switch on loop control
          BIC.B     #M,&SCFQCTL         ; Switch on modulation
;
```

1.4.8 Use without Crystal

If in an application no LCD and no precise timing is necessary, then the crystal may be omitted. If no ACLK is present (due to the missing crystal) then the DCO will run with its lowest frequency which is approximately 500 kHz. No special instructions are necessary to get this behavior.

If this lowest DCO frequency is too low, then a higher DCO tap (e.g. 10) may be used. This tap normally results in a MCLK frequency near 1 MHz. It is important to switch off the FLL loop, otherwise the FLL control will step down to tap 0 slowly. The software for this use of the DCO follows:

```
; Initialization of the DCO for non-crystal mode:
; Loop control off, tap number = 10
;
          BIS.B     #SCG0,SR ; Switch off loop control
          MOV.B     #2,&SCFI0           ; Set bit 2^1 of tap number
          MOV.B     #2,&SCFI1           ; Set bit 2^3 of tap number
;
```

2 THE ANALOG-TO-DIGITAL CONVERTERS

Two completely different Analog-to-Digital Converters (ADCs) are used, depending on the MSP430 device type:

- MSP430C32x contains a successive approximation ADC with 14- and 12-bit resolution
- MSP430C31x contains a capacitor discharge unit which allows comparison of discharge times with measuring resistors (resistive sensors).

2.1 The 14-bit Analog-to-Digital Converter

This ADC of the MSP430 is usable in two different modes:

- 14-bit ADC with an input range of the complete SV_{CC}. 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 SV_{CC}. This conversion mode is used if the voltage range of the input signal is known. The conversion needs 96 µs.

The sampling of the ADC input takes 12 MCLK cycles, this means the sampling gate is open during this time (12 µs @ 1 MHz). The input of an ADC pin can be seen as an RC low pass filter: 2 kΩ in series with 32 pF. The 32-pF 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 R_i of the source to be measured:

$$(R_i + 2k\Omega) \times 32pF < \frac{12\mu s}{\ln 2^{14}}$$

Solved for R_i this results in:

$$R_i < 36.6k\Omega$$

For the full resolution of the ADC the internal resistance of the input signal must be lower than 36.6 kΩ.

If a resolution of n bits is sufficient then the internal resistance R_i of the ADC input source can be higher:

$$R_i < \frac{12\mu s}{\ln 2^n \times 32pF} - 2k\Omega \rightarrow R_i < \frac{375000}{\ln 2^n} - 2k\Omega$$

EXAMPLE: To get a resolution of 13 bits, what is the maximum internal resistance of the input signal?

$$R_i < \frac{375000}{\ln 2^{13}} - 2k\Omega = \frac{375000}{9.0109} - 2k\Omega = 41.6k\Omega - 2k\Omega = 39.6k\Omega$$

The internal resistance of the input signal must be lower than 39.6 kΩ.

The next figure shows different methods how to connect analog signals to the MSP430:

1. Current supply for resistive sensors	(R_{sens1} at A0)
2. Voltage supply for resistive sensors	(R_{sens2} at A1)
3. Direct connection of input signals	(V_{in} at A2)
4. 4-Wire circuitry with current supply	(R_{sens3} at A3 to A5)
5. 4-Wire circuitry with voltage supply	(R_{sens4} at A6 to A7)

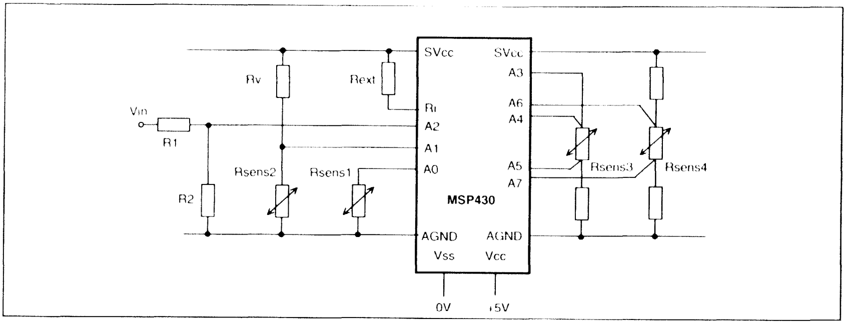


Figure 2.1: Possible Sensor Connections to the MSP430

2.1.1 The Current Source

A stable, programmable Current Source is available at the analog inputs A0 to A3. With a programming resistor R_{ext} between pins SV_{CC} and R_i it is possible to get defined currents out of the programmed analog input An: the current is directly related to the voltage SV_{CC}. The analog input to be measured and the analog 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 (0.5 SV_{CC} in Revision 0.44, which means only ranges A and B).

The current I_{CS} defined by the external resistor R_{ext} is:

$$I_{CS} = \frac{0.25 \times SV_{CC}}{R_{ext}}$$

The input voltage at the analog input with the current I_{CS} and a sensor R_{sens} is:

$$V_{in} = R_{SENS} \times I_{CS} = R_{SENS} \times \frac{0.25 \times SV_{CC}}{R_{ext}}$$

2.1.2 The 14-bit Analog-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 analog ground (V_{SS}) to SV_{CC} (V_{CC}).

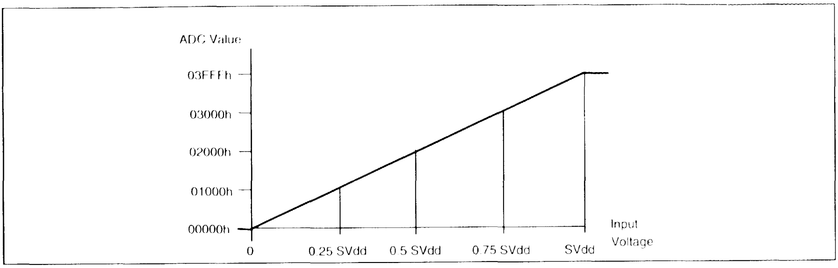


Figure 2.2: Complete ADC Range

The nominal ADC formulas for the 14-bit conversion are:

$$N = \frac{V_{Ax}}{V_{ref}} \times 2^{14} \rightarrow V_{Ax} = \frac{N \times V_{ref}}{2^{14}}$$

with:
N — 14-bit result of the ADC conversion
V_{Ax} — Input voltage at the selected analog input A_x
V_{ref} — Voltage at pin SV_{CC} (external reference or internal V_{CC})

If the current source is used, the above equation changes to:

$$N = \frac{0.25 \times V_{ref}}{R_{ext}} \times \frac{R_x}{V_{ref}} \times 2^{14} = \frac{R_x}{R_{ext}} \times 2^{12}$$

This gives for the resistor R_x:

$$R_x = \frac{N \times R_{ext}}{2^{12}}$$

with: R_{ext} Resistor between SV_{CC} pin and R_I pin (defines current I_{CS})
R_X Resistor to be measured (connected to A_X and A_{GND})

2.1.2.1 ADC with signed signals

The ADC of the MSP430 measures unsigned signals from V_{SS} to V_{CC}. 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 now shown:

- Virtual Ground IC
- Split power supply
- Use of the current source

Virtual Ground IC

With the "Phase Splitter" TLE2426 a common reference is built which lies exactly in the middle of the voltage SV_{CC}. All signed input voltages are connected to this virtual ground with their reference potential (0 V). 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. The Virtual Ground method is used with the electronic electricity meter shown in figure 4.7.

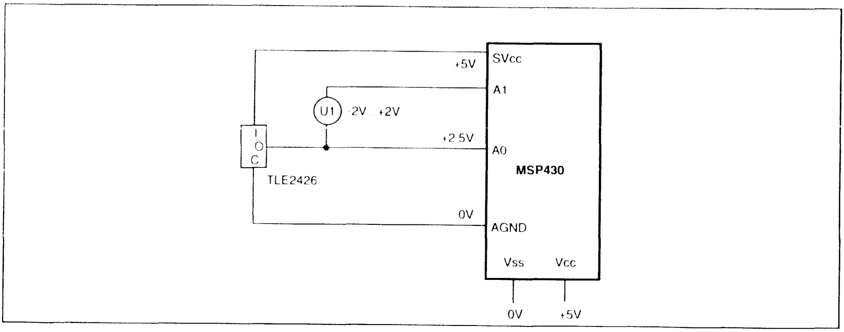


Figure 2.3: Virtual Ground IC for Level Shifting

NOTE

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 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 ANALOG-TO-DIGITAL CONVERTER
;
AIN      .EQU     0110h    ; INPUT REGISTER (FOR DIGITAL INPUTS)
AEN      .EQU     0112h    ; 0: ANALOG 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 ... 0.25 SVCC)
RNGB     .EQU     200h     ; RANGE SELECT B (0.25..0.50 SVCC)
RNGC     .EQU     400h     ; RANGE SELECT C (0.5...0.75 SVCC)
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 +2.5 V and -2.5 V, 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 (0 V). 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. The Split Power Supply method is used with the Electronic Electricity Meters shown in Figures 4.4 and 4.5.

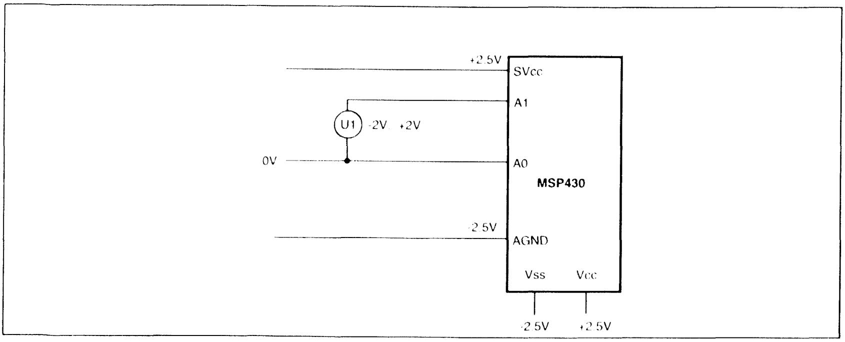


Figure 2.4: Split Power Supply for Level Shifting

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 R_h 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 ranging from -1 V to +1 V. To shift the signal's midpoint (0 V) to the midpoint of the usable ADC range ($SV_{cc}/4$) a current I_{CS} is used. The necessary current I_{CS} to shift the input signal is:

$$I_{CS} = \frac{SV_{CC}/4}{R_h} \rightarrow R_h = \frac{SV_{CC}/4}{I_{CS}}$$

R_h includes the internal resistance of the voltage source V_i.

The current I_{CS} of the current source is defined by:

$$I_{CS} = \frac{0.25 \times SV_{CC}}{R_{ext}}$$

Therefore the necessary shift resistor R_h is:

$$R_h = \frac{SV_{CC}/4 \times R_{ext}}{0.25 \times SV_{CC}} \rightarrow R_h = R_{ext}$$

The voltage V_{A1} at the analog input A1 is:

$$V_{A1} \quad V_1 + R_h \times \frac{0.25 \times SV_{cc}}{R_{ext}}$$

Therefore the unknown voltage V_1 is:

$$V_1 \quad V_{A1} - R_h \times \frac{0.25 \times SV_{cc}}{R_{ext}} = SV_{cc}\left(\frac{N}{2^{14}} - \frac{R_h \times 0.25}{R_{ext}}\right)$$

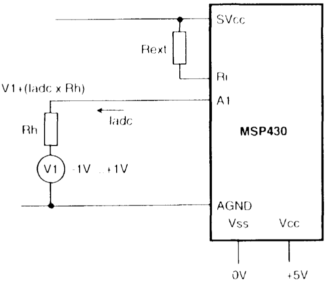


Figure 2.5: Current Source for Level Shifting

The method described is used with the current path of the MSP430 EE-Meter Demo Model shown in Figure 4.6.

2.1.2.2 Four-Wire Circuitry for Sensors

A proven method of eliminating the error coming from the voltage drop on the connection lines to the sensor is the 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 analog inputs is only some nanoamps) which means that no voltage drop falsifies the measured values. The formula for voltage supply is:

$$R_{sens} = \frac{R_1 + R_2}{\frac{2^{14}}{\Delta N} - 1}$$

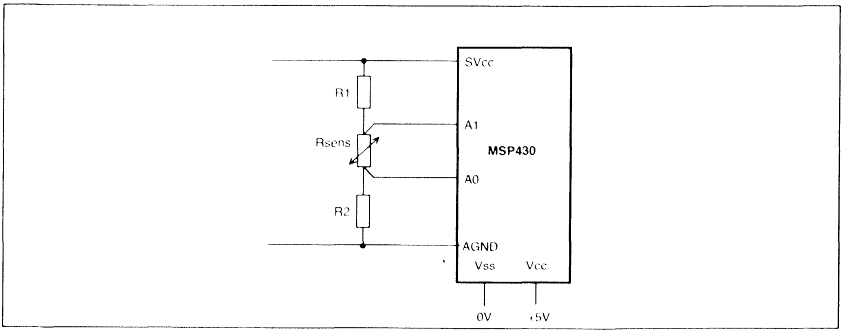


Figure 2.6: 4-Wire Circuitry with Voltage Supply

EXAMPLE: The sensor R_{sens} at A0 and A1 is measured and the ADC value of it computed by the difference of the two results measured at A1 and A0. The result is to be 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:

$$R_{sens} = \frac{\Delta N \times R_{ref}}{2^{12}}$$

The same software as shown before can be used for this hardware too.

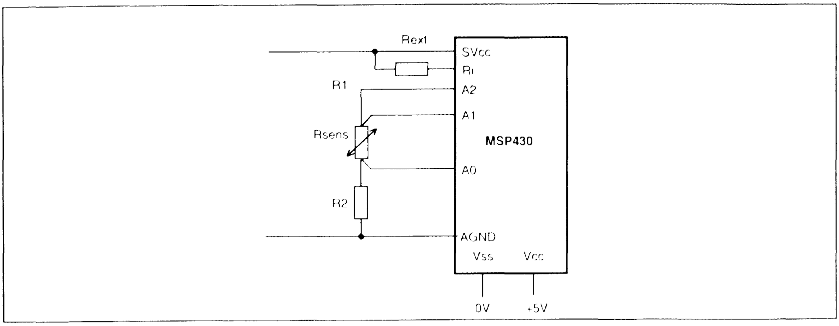


Figure 2.7: 4-Wire Circuitry with Current Supply

2.1.2.3 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 this costly calibration procedure is to make use of reference resistors. Two different methods can be used, depending on the type 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 insufficiently close 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 other well-defined points) then all tolerances are eliminated during calculation.

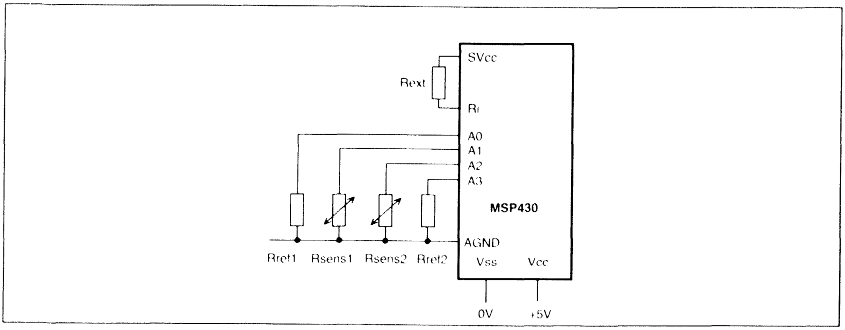


Figure 2.8: Referencing with Precision Resistors

The nominal formulas given in the previous section need to be changed if offset and slope are considered. The ADC value N_x for a given resistor R_x is now:

$$N_x = \frac{0.25 \times R_x}{R_{ext}} \times 2^{14} \times Slope + Offset$$

With two known resistors R_{ref1} and R_{ref2} it is possible to compute slope and offset and to get the values of unknown resistors exactly. The result of the solved equations gives:

$$R_x = \frac{N_x - N_{ref2}}{N_{ref2} - N_{ref1}} \times (R_{ref2} - R_{ref1}) + R_{ref2}$$

with:	N_x	ADC conversion result for R_x
	N_{ref1}	ADC conversion result for R_{ref1}
	N_{ref2}	ADC conversion result for R_{ref2}
	R_{ref1}	Resistance of R_{ref1}
	R_{ref2}	Resistance of R_{ref2}

As shown only known or measurable values are needed for the computation of R_x from N_x. Slope and offset of the ADC disappear completely.

2.1.2.4 Interrupt Handling

The ADC software examples shown above all use polling techniques for the check of the conversion completion. This takes up computing power which can be used otherwise if interrupt techniques are used.

EXAMPLE: Analog 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 analog input.

```
; HARDWARE DEFINITIONS SEE 1st ADC EXAMPLE
;
; ANALOG 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 ANALOG 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
```

2.1.2.5 Enlargement to 16-bit Mode

With the use of two additional output pins (I/O-ports or TP.x) the 14-bit ADC may be enlarged to 16 bits. The principle is simple: the resistor R_{ext} of the Current Source is modified by the paralleling of two additional resistors. These resistors have values that represent one half and one quarter of one ADC-step. Due to the fact that these fractions of a step are accurate only at one point of the ADC-range, this enlargement gives only better resolution, not better accuracy. To get the 16-bit result, four measurements are necessary: one for every combination of the two additional resistors. If these four measurements are added together, a 16-bit result is reached. The following figure shows this.

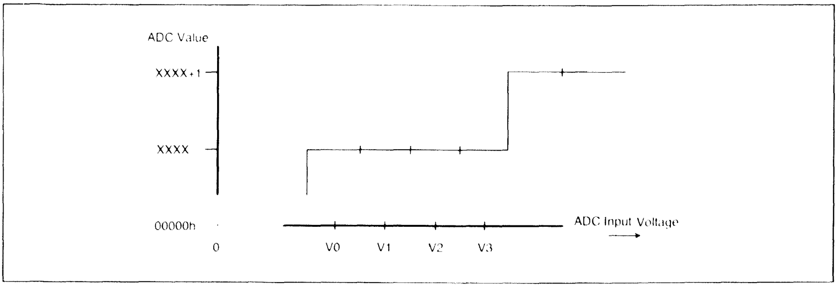


Figure 2.8a: Dividing of an ADC-Step into four Steps

The next table shows the different results of these four measurements depending on the four possible input voltages V_0 to V_3 inside of one ADC-step: the table refers to the hardware shown in figure 2.8b.

Input Voltage	Measurem. 1 TP.1 Hi-Z TP.0 Hi-Z	Measurem. 2 TP.1 Hi-Z TP.0 Hi Out	Measurem. 3 TP.1 Hi Out TP.0 Hi-Z	Measurem. 4 TP.1 Hi Out TP.0 Hi Out	Mean Value (Binary)
V0	XXXX	XXXX	XXXX	XXXX	XXXX.00
V1	XXXX	XXXX	XXXX	XXXX+1	XXXX.01
V2	XXXX	XXXX	XXXX+1	XXXX+1	XXXX.10
V3	XXXX	XXXX+1	XXXX+1	XXXX+1	XXXX.11

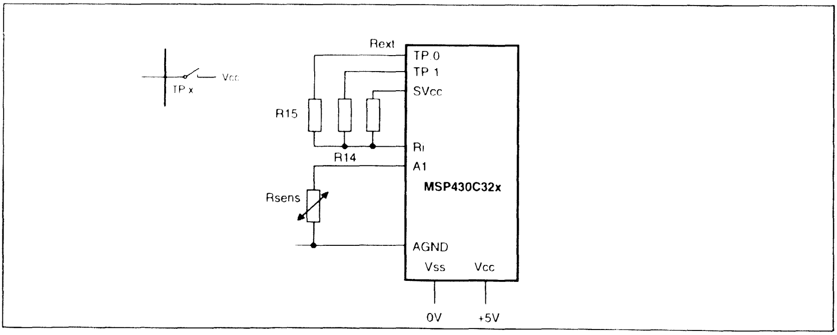


Figure 2.8b: Hardware for a 16-bit ADC

The values for the resistors R_{14} and R_{15} are:

$$R_x = \frac{2^{14} \times 0.25 \times R_{sens}}{n} = \frac{2^{12} \times R_{sens}}{n}$$

with: R_x Parallel resistor to R_{ext}
R_{sens} Value of sensor at point of best fit
n Fraction of ADC step (0.25 or 0.5)

EXAMPLE: With the hardware shown in figure 2.8b a 16-bit measurement is made. The result is placed in R5. The software may be written with a loop too. The software assumes ascending order for the two TP outputs.

```
;
TPD      .EQU      04Eh                 ; Address data register
TPE      .EQU      04Fh                 ; Address of enable register
TP0      .EQU      1                    ; Bit address of TP.0
TP1      .EQU      2                    ; Bit address of TP.1
;
         BIC.B     #TP1+TP0,&TPE        ; TP.0 and TP.1 to Hi-Z
         BIS.B     #TP1+TP0,&TPD        ; Set TPD.0 and TPD.1 to Hi
         CALL      #MEASA1              ; Measure with R14 = R15 = Hi-Z
         MOV       &ADAT,R5             ; 14-bit value to result
         ADD.B     #TP0,&TPE            ; Set R15 to Hi-Out
         CALL      #MEASA1              ; Measure
         ADD       &ADAT,R5             ; Add 14-bit value to result
         ADD.B     #TP0,&TPE            ; Set R14 to Hi-Out, R15 to Hi-Z
         CALL      #MEASA1              ; Measure
         ADD       &ADAT,R5             ; Add 14-bit value to result
         ADD.B     #TP0,&TPE            ; Set R14 and R15 to Hi-Out
         CALL      #MEASA1              ; Measure
         ADD       &ADAT,R5             ; Add 14-bit value to result
         BIC.B     #TP1+TP0,&TPE        ; 16-bit result in R5, TP.n off
;
; Measurement Subroutine for input A1
;
MEASA1   MOV       #RNGAUTO+CSOFF+A1+VREF+CS,&ACTL
L$101    BIT.B     #ADIFG,&IFG2         ; CONVERSION COMPLETED?
         JZ        L$101                ; IF Z=1: NO
         RET                            ; Return with result in ADAT
```

2.1.3 The 14-bit Analog-to-Digital Converter used in 12-bit Mode

This mode is used if the range of the input voltage is known. 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 right selection. The measurement time with MCLK = 1 MHz is only 96 μs compared with 132 μs if the auto ranging mode is used. The following figure shows the four ranges compared to SV_{CC}.

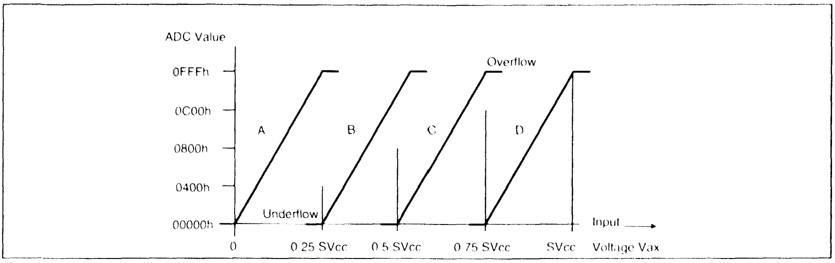


Figure 2.9: The four Single ADC Ranges

NOTE

The ADC results 0000H and 0FFFh mean underflow and overflow: the voltage at the measured analog input is below or above the limits of the addressed range.

The next figure shows how one of the ranges appears:

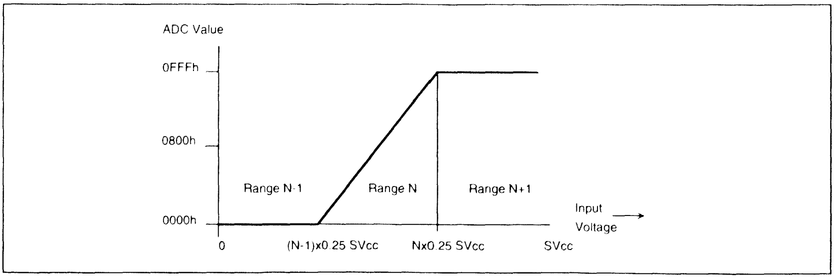


Figure 2.10: Single ADC Range

The possible ways to connect sensors to the MSP430 are the same as shown for the 14-bit ADC:

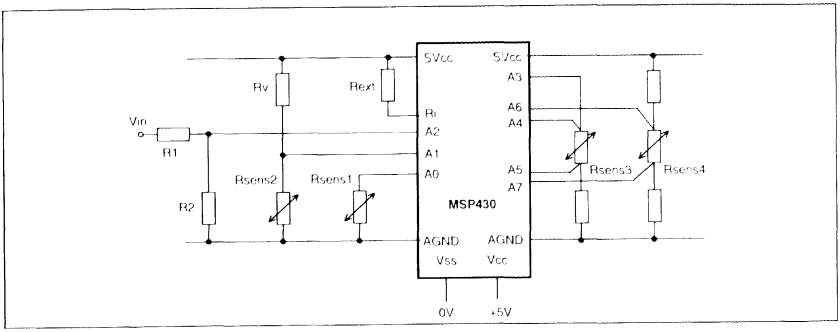


Figure 2.11: Possible Sensor Connections to the MSP430 for 12-bit ADC

The nominal ADC formulas for the 12-bit conversion are:

$$N = \frac{V_{Ax} - n \times 0.25 \times V_{ref}}{V_{ref}} \times 2^{14} \rightarrow V_{Ax} = V_{ref}(\frac{N}{2^{14}} + n \times 0.25)$$

with:	N	12-bit result of the ADC conversion
	V_{Ax}	Input voltage at the selected analog input A_x
	V_{ref}	Voltage at pin SV_{CC} (external reference or internal V_{CC})
	n	Range constant (n = 0,1,2,3 for ranges A,B,C,D)

The ADC formula for a resistor R_x (R_{sens2} in the above figure) which is connected to V_{ref} via a resistor R_v is:

$$N = \frac{\frac{R_x}{R_v + R_x} \times V_{ref} - n \times 0.25 \times V_{ref}}{V_{ref}} \times 2^{14} \rightarrow R_x = R_v \times \frac{\frac{N}{2^{12}} + n}{4 - \frac{N}{2^{12}} + n}$$

If the current source is used (as for R_{sens1} in the above figure), the above equation changes to:

$$N = \frac{\frac{0.25 \times V_{ref}}{R_{ext}} \times R_x - n \times 0.25 \times V_{ref}}{V_{ref}} \times 2^{14} = \left(\frac{R_x}{R_{ext}} - n\right) \times 2^{12}$$

This gives for the unknown resistor R_x:

$$R_x = \left(\frac{N}{2^{12}} + n\right) \times R_{ext}$$

with: R_{ext} Resistor between SV_{CC} pin and R_I pin (defines current I_{CS})
R_x Resistor to be measured (connected to A_x and A_{GND})

2.1.3.1 *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 SV_{CC}) or B (0.375 SV_{CC}), 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 (0.625 V / 4.375 V) or range B (1.875 V / 3.125 V)

For signed signals it is necessary to shift the input signal V_I to the middle of the range A or B. If range B (0.375 SV_{CC}) is used the necessary shift resistor R_h is

$$R_h = \frac{0.375 \times SV_{CC} \times R_{ext}}{0.25 \times SV_{CC}} \rightarrow R_h = 1.5 \times R_{ext}$$

The unknown voltage V_I referred to its zero point in the middle of range n is:

$$V_I = V_{Ax} - R_h \times I_{CS}$$

With the above equations for V_{Ax} this leads to:

$$V_I = 0.25 \times SV_{CC}\left(\frac{N}{2^{12}} + n - \frac{R_h}{R_{ext}}\right)$$

2.1.3.2 *Interrupt Handling*

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 should be included into the initialization part of each measurement.

EXAMPLE: Analog input A0 (without Current Source, always range B, external reference at pin SV_{CC}) 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 analog input.

```
; HARDWARE DEFINITIONS SEE 1st ADC EXAMPLE
;
; ANALOG INPUT                A0              A1
; CURRENT SOURCE              OFF             ON
; RESULT TO                   MEAS0           MEAS1
; RANGE                       B               A
; REFERENCE                   EXTERNAL        SVCC
;
; INITIALIZATION PART FOR THE ADC:
;
        MOV       #RNGB+CSOFF+A0,&ACTL
        MOV.B     #ADIE,&IE2           ; ENABLE ADC INTERRUPT
        MOV       #0FFh-3,&AEN         ; ONLY A0 AND A1 ANALOG INPUTS
        ...                            ; INITIALIZE OTHER MODULES
;
; ADC INTERRUPT HANDLER: A0 AND A1 ARE MEASURED ALTERNATELY
; 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       #RNGB+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
```

2.1.4 Connection of long Sensor Lines

If the distance from the MSP430 to the sensor is long (>30 cm) then it is recommended to use a shielded cable between the microcomputer and the sensor. This is to avoid spikes at the ADC that will cause measurement errors and also gives protection to the ADC input. Figure 2.12 shows this schematic at the left hand side. The same way Four-Wire-Circuitry may be connected to the MSP430.

If a screened cable cannot be used the schematic at the right hand side of Figure 2.12 should be used: the AGND in parallel to the signal line gives a relative good screening. Twisting of the two lines increases the protection.

To protect the measurement against spikes, hum and other unwanted noise see chapter "Signal Averaging and Noise Cancellation". This chapter shows possibilities for the minimization of these influences by software.

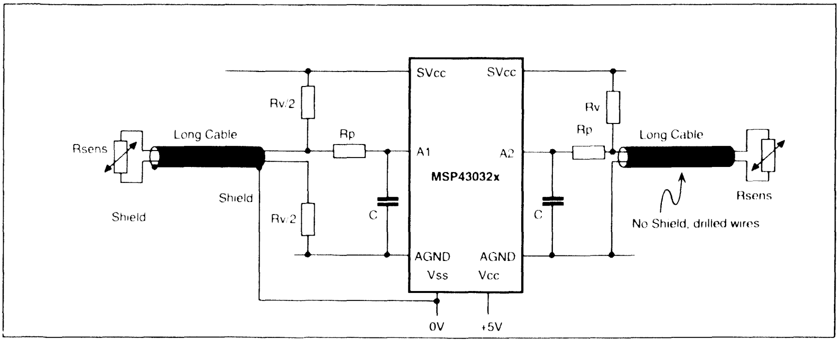


Figure 2.12: Sensor Connection via Long Cable with Voltage Supply

2.1.5 Grounding

The correct grounding is very important for ADCs with high resolution. There are some basic rules that need to be observed.

Rules for common analog and digital ground pins if only the V_{SS} pin exists as a common reference point.

1. Use of a separate analog and digital ground plane wherever possible: no thin connections from battery to pin V_{SS}
2. The V_{SS} pin is a star point for all ground connections
3. Battery and capacitor are connected together at this star point
4. No common path of the analog and the digital signals is allowed

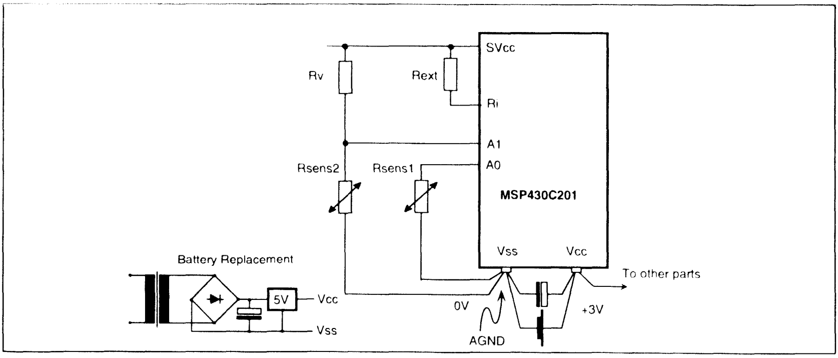


Figure 2.13: 14-bit ADC Grounding (Common Supply Connections)

Figure 2.13 shows also the use of a mains driven power supply: its V_{CC} and V_{SS} connections are connected where the battery is connected normally. The capacitor across the MSP430 supply pins may be smaller if a power supply is used: this is due to the low internal resistance of a power supply compared to the internal resistance of a battery.

Rules for separated analog and digital ground pins: AV_{SS} and DV_{SS} pins are existent

1. Use of a separate analog and digital ground plane wherever possible: no thin connections from battery to pin DV_{SS} and AV_{SS}
2. The AV_{SS} pin is a star point for all analog ground connections. The DV_{SS} pin is a star point for all digital ground connections.
3. Battery and storage capacitor are connected close together (this capacitor is needed for batteries with a relatively high internal resistance). From this capacitor two different paths go to the analog and the digital supply pins. Two small capacitors are connected across the digital (C_d) and the analog (C_a) supply pins. See below.
4. All mentioned points 1 to 3 above are also true for the V_{CC} path
5. The AV_{SS} and DV_{SS} pins must be connected together externally, they are not connected internally. The same is true for the AV_{CC} and DV_{CC} pins.
6. The coil L is needed only in very difficult cases.

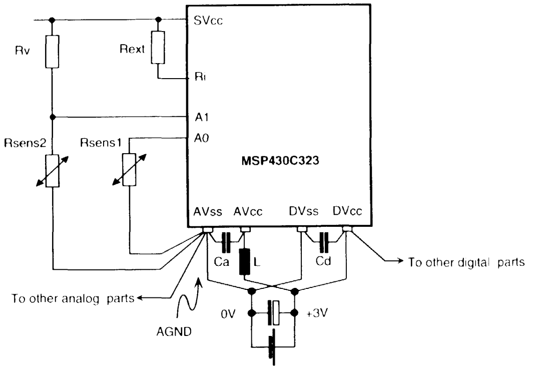


Figure 2.14: 14-bit ADC Grounding (Separate Supply Connections)

2.2 The Universal Timer/Port Module ADC used as ADC

This ADC module is contained in MSP430 versions that do not have the 14-bit ADC. The function is completely different from the 14-bit ADC: the discharge times t_{dc} for different resistors are measured and compared.

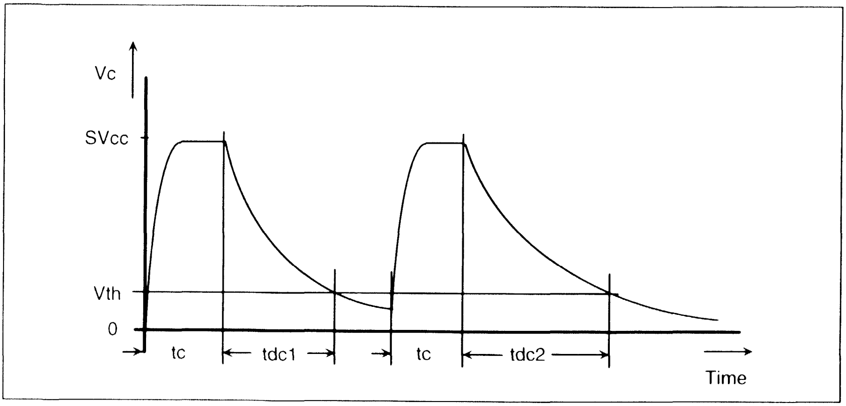


Figure 2.15: Timing for the Universal Timer

with V_{th} Threshold voltage of the comparator
t_{dc1} Discharge time with the reference resistor R_{ref}
t_{dc2} Discharge time with the sensor R_{sens}
t_c Charge time for the capacitor

The solving of the exponential equation leads to the simple equation below:

$$\frac{R_{sens}}{R_{ref}} = \frac{t_{dc2}}{t_{dc1}} \rightarrow R_{sens} = R_{ref} \times \frac{t_{dc2}}{t_{dc1}}$$

To get a resolution of n bits, the capacitor C must have a minimum capacity:

$$C > \frac{-2^n}{R_{x\min} \times f \times \ln \frac{V_{th\max}}{V_{cc}}}$$

With: f Measurement frequency (ACLK or MCLK) in Hertz
R_{xmin} Lowest resistance of sensor or reference resistor in Ohms
V_{thmax} Maximum value for threshold voltage V_{th} in Volts

EXAMPLE: Use of the Universal Timer Port as an ADC without interrupt. The measured t_{dc} values of the two sensors R_1 and R_2 and the reference resistors R_0 and R_3 are stored in RAM starting at label MSTACK (R_3 location). If an error occurs, 0FFFFh is written to the RAM location.

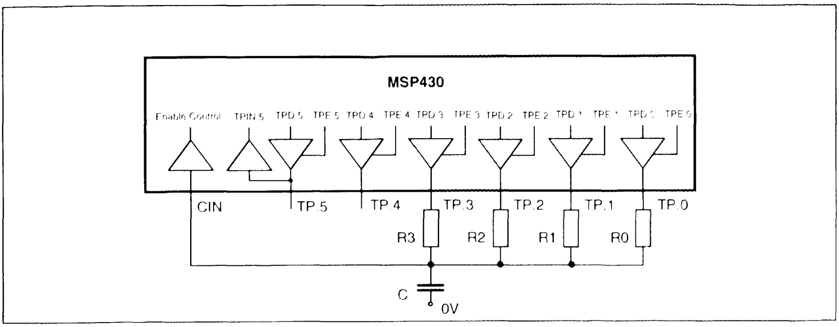


Figure 2.16: Schematic of Example

```
; 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
;
; Call:   CALL      #MEASURE
;
; Return:           Results for TP.3 to TP.0 in MSTACK to MSTACK+6
;                   Result 0FFFFh if error
;
MEASURE  PUSH.B    #TPDMAX              ; START WITH SENSOR R3 TPD.MAX
         CLR       R5                   ; INDEX FOR RESULT STACK
MEASLOP  MOV.B     #(TPSSEL0*3)+ENA,&TPCTL    ; Reset flags
;
; CAPACITOR C IS CHARGED UP FOR > 5 TAU. N-1 OUTPUTS ARE USED
;
         MOV.B     #B16+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
         MOV.B    #(TPSSEL0*3)+ENA+ENB,&TPCTL        ; Reset flags
         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?
         JNZ      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
```

2.2.1 Interrupt Handling

EXAMPLE: Use of the Universal Timer Port as an ADC with interrupt. Everything else is the same as the previous 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
;
```

```
TPE       .EQU     04Fh      ; DATA ENABLE REGISTER
;
          .BSS     MSTACK,8 ; Result stack 1st word (8 bytes)
          .BSS     ADCST,1  ; Status byte
;
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
;
; ADC handler. ADCST contains status
;
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.B    &TPE                ; Outputs disabled
          CLR.B    &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      R5                          ; ADCST + 1
;
ADCST0    BIC.B    #TPIFG,&IFG2                ; Reset ADC flag
          BIC.B    #RC2FG+EN1FG,&TPCTL         ; Reset interrupt flags
          POP      R6                          ; Restore R6
          RETI
;
          .SECT    "INT_VECT",0FFEAh; INTERRUPT VECTORS
          .WORD    ADCINT                      ; ADC INTERRUPT VECTOR;
;
```

2.2.2 Connection of long Sensor Lines

If the distance from the MSP430C31x to the sensor is long (>30 cm) then it is recommended to use a shielded lead between the microcomputer and the sensor. This is to give protection to the ADC input. Figure 2.17 shows the schematic. The protection resistors $R_{v/2}$ need to be included into the calculation: they are connected in series with the sensor. To protect the measurement against spikes, hum and other unwanted noise see chapter "Signal Averaging": here some possibilities for the minimization of these influences are shown.

Depending on the actual application the omission of the two resistors $R_{v/2}$ can give best results: the relatively low internal resistance of the TP.x output and the capacitor alone may get this.
If a shielded cable is not possible then a twisted cable or a three-core cable should be used: the unused wire is to be connected to V_{SS} as shown in figure 2.17 with R_{sens2}.

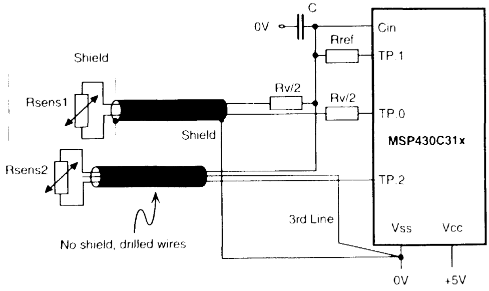


Figure 2.17: Connection of long Sensor Lines

2.2.3 Grounding

The correct grounding is very important if ADCs with high resolution are used. There are some basic rules that need to be observed.
With the MSP430C31x only the V_{SS} pin exists as a common reference point.

1. Use of separate analog and digital ground planes wherever possible: no thin connections from battery to pin V_{SS}
2. The V_{SS} pin is a star point for all V_{SS} connections
3. Battery and capacitor are connected together at this star point
4. No common path of analog and digital signals

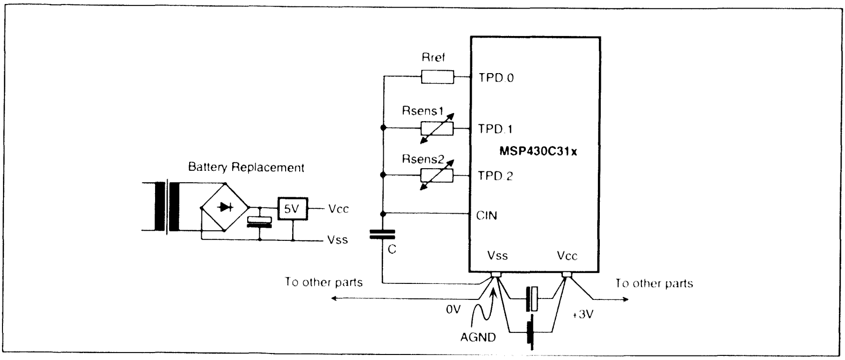


Figure 2.18: Grounding for the Universal Timer/Port ADC

Figure 2.18 shows also the use of a mains driven power supply: its V_{CC} and V_{SS} connections are connected where the battery is connected normally. The capacitor across the MSP430 pins may be smaller if a power supply is used.

3 HARDWARE APPLICATIONS

3.1 I/O-Port Usage

The eight I/O's of Port0 have a very useful feature: each one has interrupt capability for the leading and for the trailing edge of an input signal. This has the following advantages:

1. More than one interrupt input
2. Eight external events can wake-up from Low Power Modes 3 or 4
3. No glue logic necessary for most applications: all inputs can be observed without the need of gates connecting them to a single interrupt input.
4. Wake-up possible out of any input state (high or low)
5. Due to the edge-triggering of the interrupts no external switch-off logic is necessary for input signals that are of long duration.

3.1.1 General Usage

Six peripheral registers control the activities of the I/O-port:

Register	Usage	State after Reset
Input Register	Signals at I/O-pins	---
Output Register	Content of output buffer	Unchanged
Direction Register	0: Input 1: Output	Reset to input direction
Interrupt Flags	0: No interrupt pending 1: Interrupt pending	Reset
Interrupt Edges	0: Low to high causes interrupt 1: High to low causes interrupt	Unchanged
Interrupt Enable	0: Disabled 1: Enabled	Reset

The interrupt vectors, flags and peripheral addresses of I/O-port 0 are:

Name	Mnemonic	Address	Contents	Vector
Input Register	P0IN	010h		---
Output Register	P0OUT	011h		
Direction Register	P0DIR	012h		---
Interrupt Flags	P0FLG	013h	P0FLG.7 ... P0FLG.2	0FFE0h
	IFG1.3	002h	P0.1IFG	0FFF8h
	IFG1.2	002h	P0.0IFG	0FFFAh
Interrupt Edges	P0IES	014h		
Interrupt Enable	P0IE	015h	P0IE.7 ...P0IE.2	
	IE1.3	000h	P0.1IE	
	IE1.2	000h	P0.0IE	

EXAMPLE: The I/O-ports P0.0 to P0.3 are used for input only. P0.4 to P0.7 are outputs, initially at low level. The conditions are:

P0.0	Every change is counted
P0.1	Any Hi-Lo change is counted
P0.2	Any Lo-Hi change is counted
P0.3	Every change is counted

```
;
; RAM definitions
;
        .BSS     P0_0CNT,2          ; Counter for P0.0
        .BSS     P0_1CNT,2          ; Counter for P0.1
        .BSS     P0_2CNT,2          ; Counter for P0.2
        .BSS     P0_3CNT,2          ; Counter for P0.3
;
; Initialization for Port0
;
        MOV.B    #000h,&P0OUT       ; Output register low
        MOV.B    #0F0h,&P0DIR       ; P0.4 to P0.7 outputs
        MOV.B    #00Bh,&P0IES       ; P0.0 to P0.3 Hi-Lo, P0.2 Lo-Hi
        MOV.B    #00Ch,&P0IE        ; P0.2 to P0.3 interrupt enable
        BIS.B    #00Ch,&IE1         ; P0.0 to P0.1 interrupt enable
;
; Interrupt handler for P0.0. Every change is counted
;
P0_0HAN INC      P0_0CNT            ; Flag is reset automatically
        XOR.B    #1,&P0IES          ; Change edge select
        RETI
;
; Interrupt handler for P0.1. Any Hi-Lo change is counted
;
P0_1HAN INC      P0_1CNT            ; Flag is reset automatically
        RETI
;
, Interrupt handler for P0.2 and P0.3
; The flags of all read transitions are reset. Transitions
; occuring during the interrupt routine cause interrupt after
; the RETI
;
P0_23HAN PUSH    R5                 ; Save R5
        MOV.B    &P0FLG,R5          ; Copy interrupt flags
        BIC.B    R5,&P0FLG          ; Reset read flags
        BIT      #4,R5              ; P0.2 flag to carry
        ADC      P0_2CNT            ; Add carry to counter
        BIT      #8,R5              ; P0.3 flag to carry
        JNC      L$304
        INC      P0_3CNT            ; P0.3 changed
        XOR.B    #8,P0IES           ; Change edge select
L$304   POP      R5                 ; Restore R5
        RETI
;
        .SECT    "INT_VECT",0FFF8h
        .WORD    P0_1HAN                      ; P0.1 INTERRUPT VECTOR;
        .WORD    P0_0HAN                      ; P0.0 INTERRUPT VECTOR;
;
        .SECT    "INT_VECT1",0FFE0h
        .WORD    P0_23HAN           ; P0.2/7 INTERRUPT VECTOR
```

3.1.2 Zero Crossing Detection

With the external components shown in figure 3.1 it is possible to build a zero crossing input for the MSP430. The components shown are designed for an external voltage V_{mains} = 230 V. With a circuit capacitance (wiring, diodes) of 30 pF as shown, the following delays will occur (all values for V_{mains} = 230 V, f = 50 Hz, V_{dd} = +5 V) (timing is in μs):

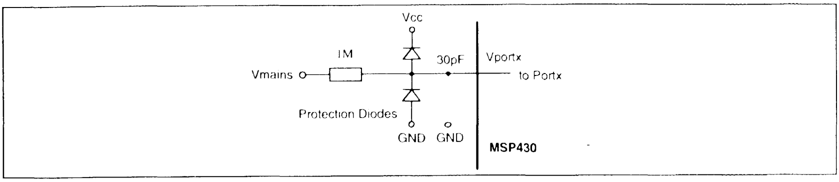


Figure 3.2: MSP430 Input for Zero-Crossing

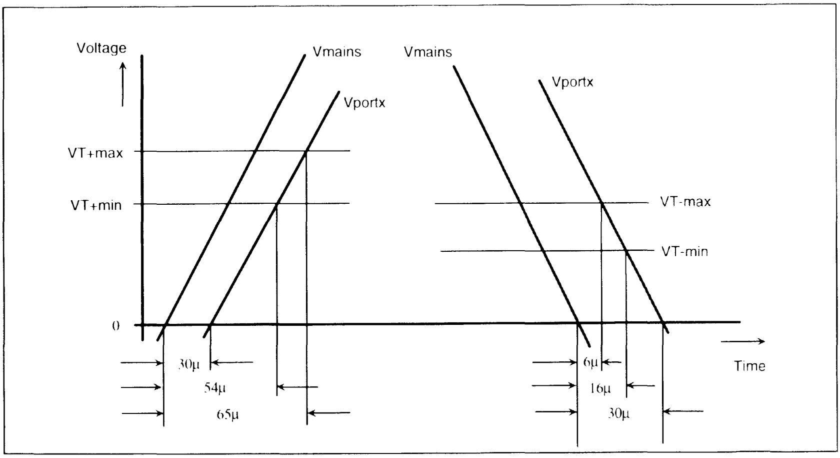


Figure 3.1: Timing for the Zero Crossing

Delay caused by RC (1 MΩ x 30 pF): 0.54° or 30 µs. Same value for leading and trailing edges.

Delay caused by input thresholds: Leading edge: 24 to 35 µs. (V_{T+} = 2.3 to 3.4 V)
Trailing edge: 14 to 24 µs. (V_{T-} = 2.3 to 1.4 V)

The resulting delays are: Leading edge: 54 to 65 µs.
Trailing edge: 6 to 16 µs.

These small deviations do not play a role for 50 Hz or 60 Hz phase control applications with TRIAC's. If other input conditions than 230 V and 50 Hz are used then the resulting delays can be calculated with the following formulas:

$$t_D = \frac{V_T}{S_V}; \quad S_V \quad \frac{d(\hat{U}\sin\omega t)}{dt} \quad \hat{U} \times \omega \times \cos\omega t$$

With	t_D	Delay time caused by input threshold
	V_T	Threshold voltage of input
	S_V	Slope of input voltage
	ω	Angular frequency $2\,\pi f$
	$\hat{U}$	Input voltage $\hat{U}_{mains}$

For t = 0 (zero crossing time) the above equation becomes:

$$t_D = \frac{V_T}{\hat{U} \times \omega \times 1} = \frac{V_T}{\hat{U} \times \omega}$$

3.1.3 Output Buffering

The outputs of the MSP430 (P0.x, Ox) have nominal internal resistances depending on the supply voltage V_{DD}:

V_{DD} =	3 V:	max. 333 Ω	(ΔV = 0.4 V max. @ 1.2 mA)
V_{DD} =	5 V:	max. 266 Ω	(ΔV = 0.4 V max. @ 1.5 mA)

These internal resistances are non-linear and are valid only for small output currents (see above). If larger currents are drawn then saturation effects will limit the output current.

These outputs are intended for driving digital inputs and gates and they normally have too high impedance for other applications such as the driving of relays, lines etc. If output currents greater than the above mentioned ones are needed then output buffering is necessary. The following figure shows some possibilities. The resistors shown for the limitation of the MSP430 output current are minimum values. The design is made for V_{DD}: = 5 V; values in brackets are for V_{DD}: = 3 V.

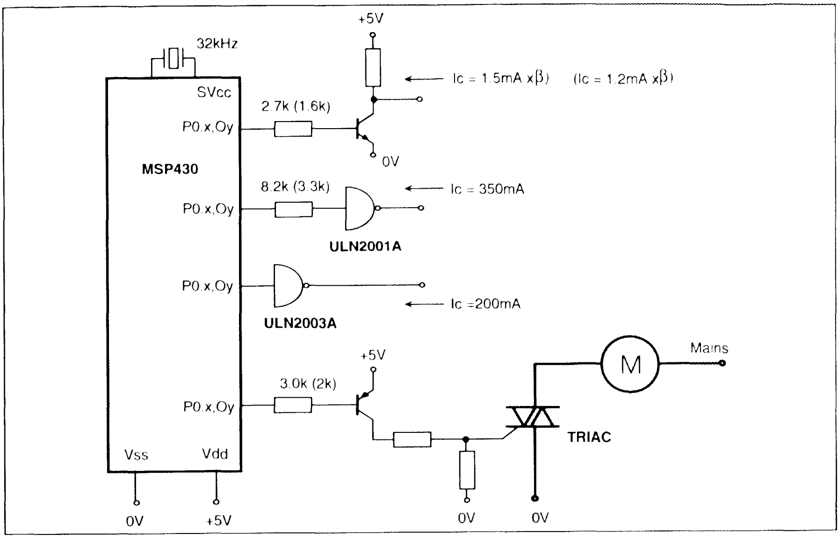


Figure 3.3: Output Buffering

3.1.4 MSP430C31x I/Os

If the Universal Timer/Counter Port is not used for analog-to-digital conversion, or is only partially used, then the unused pins are available as outputs that may be switched to HI-Z. The Universal Timer/Counter Port may be used in three different modes:

- Two 8-bit timers and 6 output pins
- One 16-bit timer and 6 output pins
- Active analog-to-digital conversion that does not need all I/O-pins

The ports TP.0 to TP.5 are completely independent of the analog-to-digital converter: the ports not used for the ADC may be set or reset without disturbing the conversion. Which ports are used for the sensors and reference resistors does not matter.
Power-up resets the data register to zero and switches all ports to HI-Z.

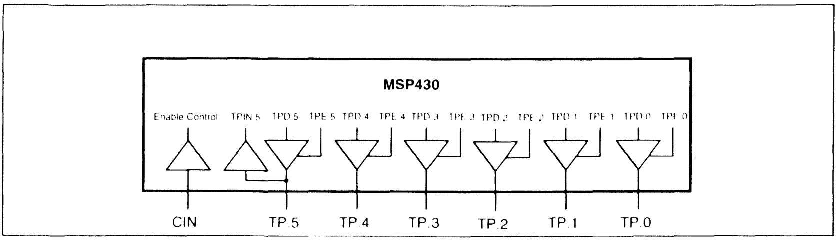


Figure 3.4: MSP430C31x Port

3.1.4.1 I/Os used with the Analog-to-Digital Converter

The analog-to-digital conversion uses the pins CIN and at least two of the TP.x pins (one for the reference and one for the sensor to be measured). Therefore up to 4 outputs are available. It is only possible to use bit instructions (BIS, BIC, XOR) for the modification of the outputs: this is due to the control bits located in the Data Register TPD and Data Enable Register TPE. The programming of the port is the same as described in the next section.

NOTE

> For precise ADC results it is recommended to avoid changes of the ports during the measurement. The board layout and the physical distance of the switched port define the influence of the pin CIN. Spikes coming from the switching of ports can alter the result of a measurement especially if they occur near to the threshold voltage.

3.1.4.2 I/Os used without ADC

This mode allows 5 outputs with the possibility of being switched to HI-Z (TP.0 to TP.4) and one I/O-pin (TP.5). Additionally, two 8-bit timers or one 16-bit timer are available. If one of the timers is used, only bit instructions (BIT, BIS, BIC, XOR) are possible for the manipulation of the port: four control bits of the timers are located within the Data Register TPD and Data Enable Register TPE. If MOV instructions are used, all bits are affected.

EXAMPLE: All six ports are used as outputs. The possibilities of the port are shown:

```
;
; Definitions for the Counter Port
;
TPD     .EQU    04Eh    ; Data Register
TPE     .EQU    04Fh    ; Data Enable Register 1: output enabled
TP0     .EQU    001h    ; TP.0 bit address
TP1     .EQU    002h    ; TP.1 bit address
TP2     .EQU    004h    ; TP.2 bit address
TP3     .EQU    008h    ; TP.3 bit address
TP4     .EQU    010h    ; TP.4 bit address
TP5     .EQU    020h    ; TP.5 bit address
;
; Reset all ports and enable all to outputs
```

```
;
        BIC.B    #TP0+TP1+TP2+TP3+TP4+TP5,&TPD      ; Data to low
        BIS.B    #TP0+TP1+TP2+TP3+TP4+TP5,&TPE      ; Enable outputs
;
; Toggle TP.0 and TP.4, set TP.5 and TP.2 afterwards
;
        XOR.B    #TP0+TP4,&TPD              ; Toggle TP.0 and TP.4
        BIS.B    #TP5+TP2,&TPD              ; Set TP.5 and TP.2
;
; Switch TP.1 and TP.3 to HI-Z state
;
        BIC.B    #TP1+TP3,&TPE        ; HI-Z state for TP.1 and TP.3
;
```

3.1.5 I/Os used for fast serial Transfer

The combination of hardware and software shown below allows the fastest possible serial transfer with the MSP430 family. The data line needs to be P0.0, for the clock line any other Port0 line may be used.

```
;
P0OUT   .EQU    011h     ; Port0 Output register
P0DIR   .EQU    012h     ; Port0 Direction register
P00     .EQU    01h      ; Bit address of P0.0
P01     .EQU    02h      ; Bit address of P0.1
;
        MOV     DATA,R4             ; 1st 16bit data to R4
        CALL    #SERIAL_FAST_INIT;  1st transfer
        MOV     DATA1,R4            ; 2nd 16bit data to R4
        CALL    #SERIAL_FAST        ; 2nd transfer
        ....                        ; aso.
;
; Initialization of the fast serial transfer
;
SERIAL_FAST_INIT         ; Initialization part
        BIC.B   #P00+P01,&P0OUT     ; Reset P0.0 and P0.1
        BIS.B   #P00+P01,&P0DIR     ; P0.0 and P0.1 to output dir.
;
; Part for 2nd and all following transfers
;
SERIAL_FAST                         ; Initialization is made
        RRC     R4                  ; LSB to carry           1 cycle
        ADDC.B  #P01,&P0OUT         ; Data out, set clock    4 cycles
        BIC.B   #P00+P01,&P0OUT     ; Reset data and clock   5 cycles
;
        RRC     R4                  ; LSB+1 to carry 1 cycle
        ADDC.B  #P01,&P0OUT         ; Data out, set clock    4 cycles
        BIC.B   #P00+P01,&P0OUT     ; Reset data and clock   5 cycles
;
        ......                      ; Output all bits the same way
;
        RRC     R4                  ; MSB to carry           1 cycle
        ADDC.B  #P01,&P0OUT         ; Data out, set clock    4 cycles
        BIC.B   #P00+P01,&P0OUT     ; Reset data and clock   5 cycles
        RET
;
```

Each bit needs 10 cycles for the transfer, this results in a maximum Baud rate for the transfer:

$$Baud\ Rate_{max} = \frac{MCLK}{10}$$

This means if MCLK = 1.024 MHz then the maximum Baud rate is 102.4 kBaud.

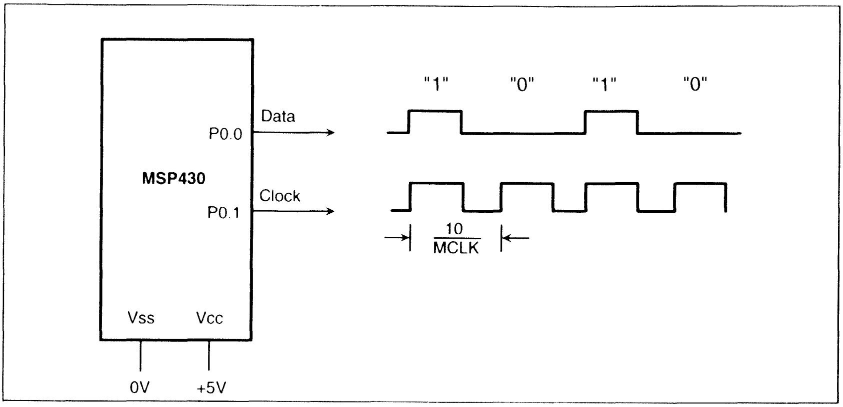


Figure 3.4a: Connections for fast serial Transfer

3.2 Storage of Calibration Constants

Metering devices such as electricity meters, gas meters etc. normally need to store calibration constants (offsets, slopes, limits, addresses, correction factors) for use during the measurements. Depending on the voltage supply (mains, battery) it is necessary or possible to have them stored in the on-chip RAM or in an external EEPROM. Both methods are explained below.

3.2.1 External EEPROM for Calibration Constants

The storage of calibration constants, energy values, meter numbers and device versions in external EEPROM's is necessary if the metering device is supplied by the mains. This is due to the possible power failures that may occur.
The EEPROM is connected to the MSP430 by dedicated inputs and outputs. Three (two) control lines are necessary for proper function:

- 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 V_{CC} is necessary. Three possible solutions are shown:

1. The EEPROM is connected to SV_{CC}. 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 driven by an output port.
3. The EEPROM is connected to +5 V permanently, if its power consumption does not play a role.

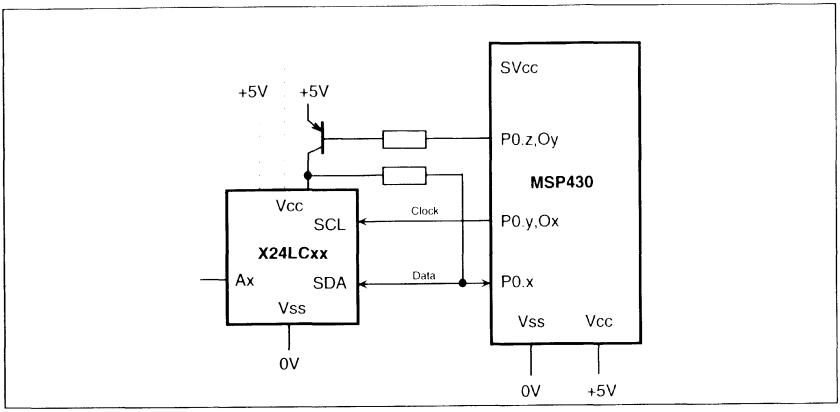


Figure 3.5: External EEPROM Connections

An additional way to connect an EEPROM to the MSP430 is shown in the section describing the I^{2}C-Bus.

NOTE

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 1 MHz each one of the next instructions needs 5 µs.

EXAMPLE: The EEPROM with the dedicated I/O-lines is controlled with normal I/O-instructions. The SCL line is driven by 017, the SDA line is driven by P0.6. The line is driven high by a resistor, and low by the output buffer.

```
P0OUT    .EQU    011h    ; Port0 Output register
P0DIR    .EQU    012h    ; Port0 Direction register
SCL      .EQU    0F0h    ; 017 controls SCL, 039h LCD Address
SDA      .EQU    040h    ; P0.6 CONTROLS SDA
```

```
LCDM      .EQU     030h      ; LCD control byte
;
; INITIALIZE I C-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 IF OUTPUT
        ...
;
; 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 contents 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 I^2C-Bus Connection for more details of such a subroutine.

3.2.2 Internal RAM for Calibration Constants

The internal RAM can be used for the calibration constants if a permanently connected battery is used for the power supply. The use of Low Power Mode 3 or 4 is necessary for such applications to get battery life times of from 5 to 12 years.

3.3 M-BUS Connection

The MSP430 connection is shown in the next figure. Three supply modes are possible when used with the TSS721:

Remote Supply: The MSP430 is fully powered from the TSS721
Remote Supply/Battery support: The MSP430 is supplied normally from the TSS721. In case of a bus power fail the battery powers the MSP430
Battery Supply: The MSP430 is always supplied from its battery

All these operating modes are described in detail in the "TSS721 M-Bus Transceiver Applications Book".

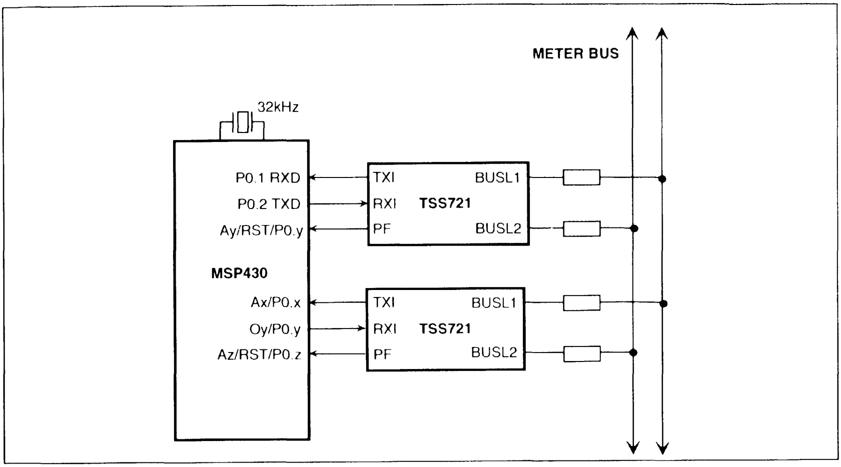


Figure 3.6: TSS721 Connections to MSP430

Two different TSS721 connections are shown in the figure above:

- If the 8-bit Interval Timer with its UART is to be used then the upper connection is necessary. TXI or TX are connected to RXD (P0.1) and RXI or RX are connected to TXD (P0.2).
- If a pure software UART or an individual protocol is to be used, then any input and output combination may be used

3.4 I^{2}C-BUS Connection

If more than one device is to be connected to the I^{2}C-Bus then two I/O-ports are needed for the control of the I^{2}C-peripherals. The reason is the need to switch SDA and SCL to the high impedance state.

The figure below shows the connection of three I^{2}C-peripherals to the MSP430:

- An EEPROM with 128x8-bit data
- An EEPROM with 2048x8-bit data
- An 8-bit DAC/ADC

The bus lines are driven high by the R_p resistors (P0.x is input) and low by the output ports (P0.x is output).

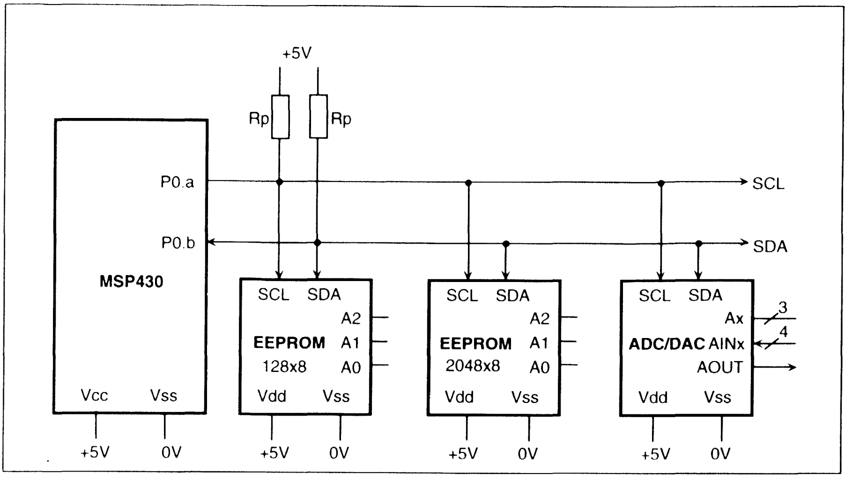


Figure 3.7: I^{2}C-Bus connections

NOTE

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 I^{2}C-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 that can be switched to HI-Z (TP-Port of MSP430C31x e.g.).

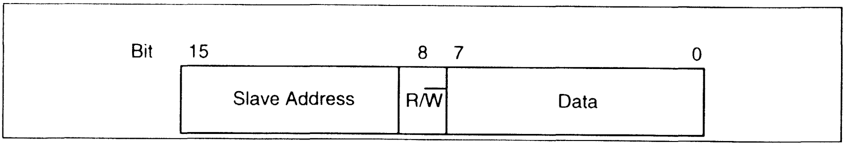


Figure 3.8: Word Format for I^{2}C-Handler Call

```
SCL      .EQU    080h     ; P0.7 CONTROLS SCL
SCLDAT   .EQU    011h     ; P0OUT
SCLEN    .EQU    012h     ; P0DIR
SDA      .EQU    040h     ; P0.6 CONTROLS SDA
SDAIN    .EQU    010      ; P0 input register
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 (MSP430C31x)
;
        BIC.B    #SDA,&SDAEN       ; SDA TO INPUT DIRECTION (HI)
        BIC.B    #SDA,&SDADAT      ; SDA OUTPUT BUFFER LOW
;       BIC.B    #SCL,&SCLEN       ; SCL to input direction (HI)
        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     #I2CACKN
        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
         POP      R5                   ; Restore R5
         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
;
```

3.5 Hardware Optimization

The MSP430 permits using unused analog inputs (A7 to A0) and select lines (S29 to S2) for inputs and outputs respectively. The next two sections explain in detail how to program and use these inputs and outputs.

3.5.1 Use of unused Analog Inputs

Unused Analog-to-Digital-Converter (ADC) inputs can be used as digital inputs or, with some restrictions, as digital outputs.

3.5.1.1 Analog Inputs used for Digital Inputs

Any ADC input A7 to A0 can be used as a digital input. It is only necessary to program it (for example during the initialization) for this function. Two things are important if this feature is used:

- Any activity at these digital inputs has to be stopped during ongoing sensitive ADC measurements. This activity will cause noise which will falsify the ADC results. Activity means in this case:
  - No change of the AEN register (switching between digital and analog mode)
  - No input change at the digital ADC inputs (this allows only rarely changing input signals at these inputs).
- All bits which are switched to ADC inputs will read zero when read. Therefore it is not necessary to clear them by software after the reading.

Software Example: A0 to A4 are used as ADC inputs, A5 to A7 as digital inputs.

```
;
AIN      .EQU    0110h    ; Address DIGITAL INPUT REGISTER
AEN      .EQU    0112h    ; Address DIGITAL INPUT ENABLE REG.
A7EN     .EQU    080h     ; Bits in Dig. Input Enable Reg.:
A6EN     .EQU    040h     ; 0: ADC     1: Digital Input
A5EN     .EQU    020h     ;

; INITIALIZATION: A7 TO A5 ARE SWITCHED TO DIGITAL INPUTS
; A4 TO A0 ARE USED AS ANALOG INPUTS
;
                 MOV      #A7EN+A6EN+A5EN,&AEN      ; A7 TO A5 DIGITAL
MODE
                 ...
; NORMAL PROGRAM EXECUTION:
; CHECK IF A7 OR A5 ARE HIGH. IF YES: JUMP TO LABEL L$100
;
         BIT     #A7EN+A5EN,&AIN            ; A7 .OR. A5 HI?
         JNZ     L$100                      ; YES
                 ...                        ; NO, CONTINUE
;
; CHECK IF ALL DIGITAL INPUTS A7 TO A5 ARE LOW. IF YES: L$200
;
                 TST      &AIN              ; A7 TO A5 LO?
                 JZ       L$200             ; YES, (ANALOG INPUTS READ
ZERO)
;
```

3.5.1.2 Analog Inputs used for Digital Outputs

If outputs are very necessary then the unused ADC inputs with the Current Source connection can be used if the following restrictions are considered:

- Only one ADC input can be high at a given time (1 out of n principle)
- Only the ADC inputs A0 to A3 are usable (only they are connected to the Current Source)
- The outputs can get high only during the time the ADC does not use the Current Source
- The output current is directly related to the supply voltage V_{CC}
- The output voltage is only about 50% of the supply voltage V_{CC}. Logic levels have to be checked carefully therefore. A transistor stage may perhaps be necessary (if not there anyway, e.g. for a relay)
- The output current is given by the current Source's Current. The same considerations as with the point before have to be made. The pull-down resistor has to be high enough to allow the maximum output level.

The example below shows the ADC part which uses the ADC inputs A0 and A1 as digital outputs driving two stages: a transistor stage (energy pulse e.g. with an electricity meter) and a 3 V gate (3 V guarantees that the input levels are sufficient).

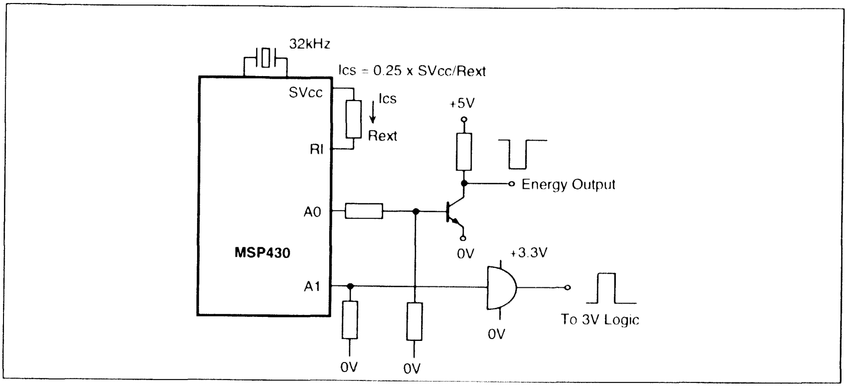


Figure 3.9: Unused ADC inputs used as Outputs

EXAMPLE. To control the two outputs shown above the following software part is necessary:

```
;
ACTL     .EQU    0114h    ; ADC CONTROL REGISTER ACTL
VREF     .EQU    02h      ; 0: Ext. Reference    1: SVCC ON
A0       .EQU    0000h    ; AD INPUT SELECT A0
A1       .EQU    0004h    ;                     A1
CSA0     .EQU    0000h    ; CURRENT SOURCE TO A0
CSA1     .EQU    0040h    ;                     A1
CSOFF    .EQU    0100h    ; CURRENT SOURCE OFF BIT
;
; SET A0 HI FOR 3 ms: SELECT A0 FOR CURRENT SOURCE AND INPUT

         MOV     #VREF+A0+CSA0,&ACTL      ; PD = 0, SVCC = on
         CALL    #WAIT3MS           ; WAIT 3 ms
         BIS     #CSOFF,&ACTL       ; CURRENT SOURCE OFF;
;
; SET A1 HI FOR 3 ms: SELECT A1 FOR CURRENT SOURCE AND INPUT

         MOV     #VREF+A1+CSA1,&ACTL      ; PD = 0, SVCC = on
         CALL    #WAIT3MS           ; WAIT 3 ms
         BIS     #CSOFF,&ACTL       ; CURRENT SOURCE OFF
         ...

;
```

3.5.2 Use of unused Select Lines for Digital Outputs

The LCD-driver of the MSP430 provides additional digital outputs if select lines are not used. Up to 28 digital outputs are possible by 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	

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). Only groups of four select lines can be switched to digital output mode.

NOTES

> 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.B     #(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
```

3.6 Digital-to-Analog Converters

The MSP430 does not contain a Digital-to-Analog Converter (DAC) on-chip in its current versions, but it is relatively simple to implement the DAC function if needed. Three different solutions with distinct hardware and software requirements are shown below:

- The R/2R method
- The Weighted Resistors method
- Integrated Digital-to-Analog Converters connected to the I^2C-Bus

3.6.1 R/2R Method

With a CMOS shift register a Digital-to-Analog Converter can be built with any length. The outputs Q_x of the shift register switch the 2R-resistors to 0 V or V_{CC} according to the digital input. The voltage V_{out} at the non-inverting input and also at the output of the opamp is:

$$Vout = \frac{k}{2^n} \times V_{CC}$$

with:		
	k	Value of the digital input word with n bits length
	n	Number of Q outputs, maximum length of input word
	V_{CC}	Supply voltage

Signed output is possible by level shifting or by splitting of the power supply ($+V_{CC}/2$ and $-V_{CC}/2$). With split power supplies the voltage at the output of the opamp is:

$$V_{out} = \frac{k}{2^n} \times V_{CC} - \frac{V_{CC}}{2} = V_{CC}\left(\frac{k}{2^n} - \frac{1}{2}\right)$$

Advantages of the R/2R-Method:

- Only two different resistors are necessary (R and 2R)
- Absolute monotonicity over the complete output range
- Internal impedance independent of the digital value: impedance is always R
- Expandable to any bit length by adding shift registers

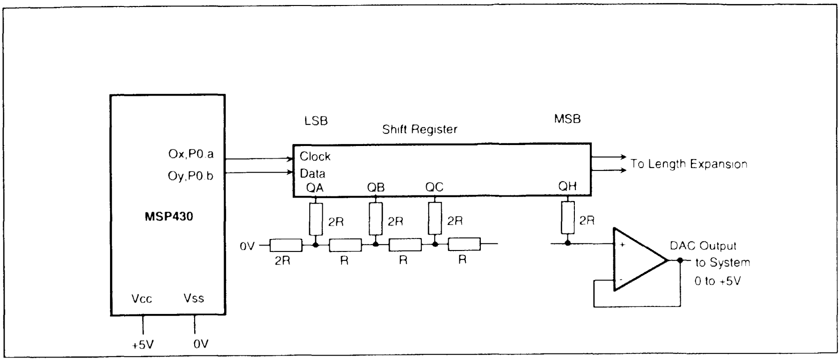


Figure 3.10: R/2R Method for Digital-to-Analog Conversion

3.6.2 Weighted Resistors Method

The simplest Digital-to-Analog Conversion Method: only (n+3) resistors and an opamp are required for an n-bit DAC. This method is used if the performance of the DAC may be low.

The example shown below delivers 2^{n+1} different output voltage steps. They may be seen signed if the voltage $V_{CC}/2$ is seen as zero point. The output voltage at the DAC output is:

$$V_{out} = V_{ninv} - \sum I_n \times R = \frac{V_{CC}}{2} - \left(a \times \frac{1}{R} + b \times \frac{1}{2R} + c \times \frac{1}{4R} + ... + x \times \frac{1}{2^n R} \right)$$

with:

V_{out}	Output voltage of the DAC
V_{ninv}	Voltage at the non inverted input of the opamp ($V_{CC}/2$)
V_{CC}	Supply voltage of the MSP430 and periphery
R	Normalized resistor used with the DAC
a...x	Multiplication factors for the weighted resistors: -1 if port is switched to V_{SS} 0 if port is switched to input direction (HI-Z) +1 if port is switched to V_{CC}

Normally all of the ports are switched to the same potential (V_{SS} or V_{CC}) or are disabled. This allows signed output voltages referenced to $V_{CC}/2$.

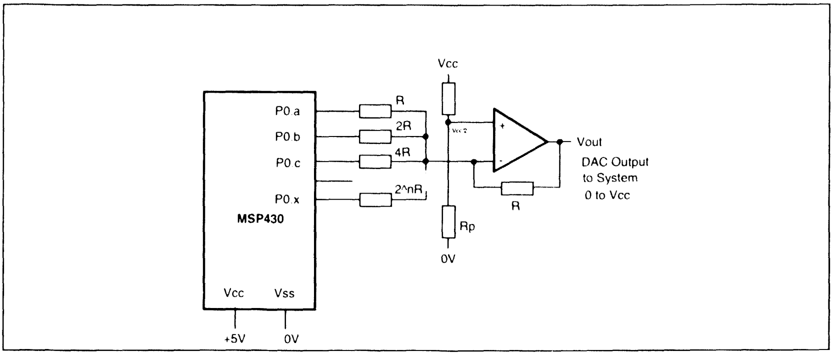


Figure 3.11: Weighted Resistors Method for Digital-to-Analog Conversion

3.6.3 Digital to Analog Converters connected via I²C-Bus

The figure below shows two different DAC's which are connected to the MSP430 via the I²C-Bus:

- A single output 8-bit Digital-to-Analog Converter (with additional 4 ADC inputs): one analog output AOUT is provided.
- An octuple 6-bit DAC: eight analog outputs DAC0 to DAC7 are provided for the system

The generic software to handle these devices is contained in the section explaining the I²C-Bus.

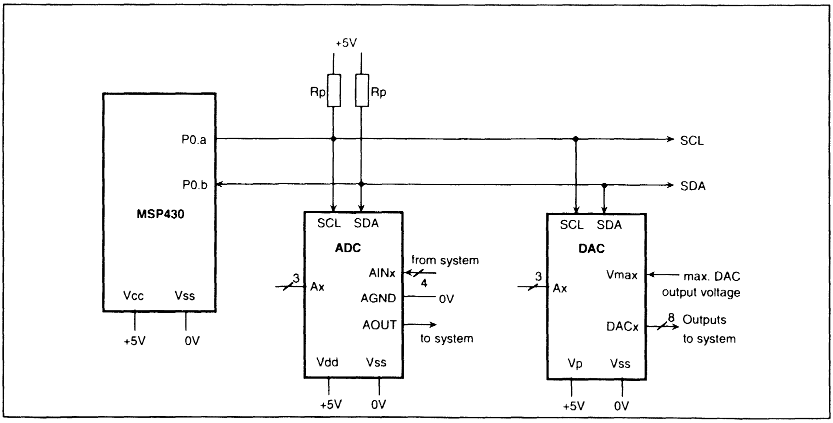


Figure 3.12: I²C-Bus for Digital-to-Analog Converter Connection

4 APPLICATION EXAMPLES

Several metering examples are given in the next sections. Common for nearly all of them is the storage of calibration data, tables, constants etc. in external EEPROM's. External EEPROM's are used for safety reasons: if the microcomputer fails completely then it is relatively easy to read out the accumulated consumption values. This is normally impossible if these values reside in internal EEPROM's.

These EEPROM's can store also tables that describe the principal errors of a given measurement principle dependent on the input value (current, flow, heat etc.). The MSP430 with its excellent table processing capabilities can determine the right starting value out of these tables and calculate the linear, quadratic or cubic approximate value.

The next figure shows the principal error of a meter. The complete range starting at 1% up to 200% is divided into sub ranges of different length. The appertain table contains the starting point, the different distances and the inherent error at the beginning of each range. With this information the MSP430 can calculate the error at any point of the measurement range.

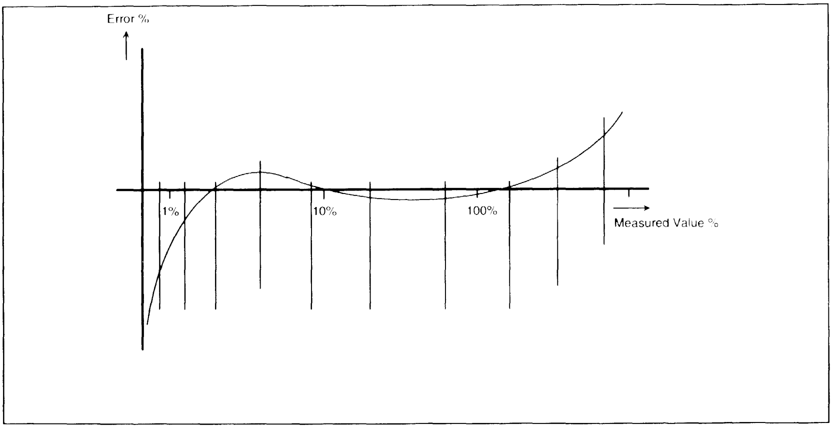


Figure 4.1: Segmentation of Measured Value

4.1 Electricity Meters

The MSP430 can be used in two completely different kinds of electronic electricity meters. The difference of the two methods is mainly where the electrical energy

$$W = \int U \times I \times dt$$

is measured:

1. The electrical energy is measured in a frontend separated from the MSP430. Several methods exist for doing that: Hall effect sensors, Ferraris wheel interfaces, analog multipliers etc. The interface to the MSP430 is normally a train of pulses, where every pulse represents a defined energy (Ws, kWs, Wh). MSP430C32x or MSP430C31x may be used.
2. The electrical energy is calculated by the MSP430 itself, using its 14-bit ADC for the measurement of current and voltage. Only MSP430C32x can be used.

The two different solutions are shown in figure 4.2

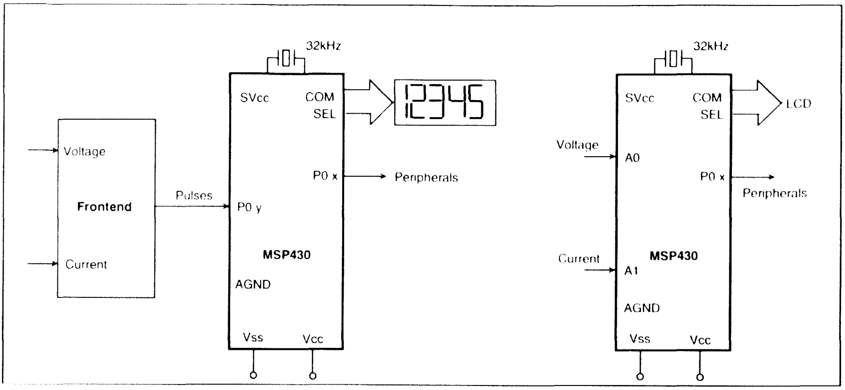


Figure 4.2: Two Methods for Electricity Meters

Only the second method is used with the electricity meters shown: the unnecessary frontend gives a cost advantage when compared to the other solution.

4.1.1 Measurement Principle of the Electricity Meters

The "Reduced Scan Principle" used measures current and voltage at regular time intervals and multiplies the current and voltage samples. The multiplication results are summed up: the sum represents the used power (Ws, kWh). While the normally used method measures voltage and current at the same time, the "Reduced Scan Principle" measures voltage and current samples alternately. Every current sample is used twice: once it is multiplied with the voltage value measured before, and once with the voltage value measured afterwards. (To reduce further the necessary multiplications these two multiplications are reduced to one by using the sum of the two current samples). The measurement principle is shown in Figure 4.3.

This measurement principle is implemented in an evaluation board for a 3-phase meter which has a typical error of 0.2%. See Figure 4.6.

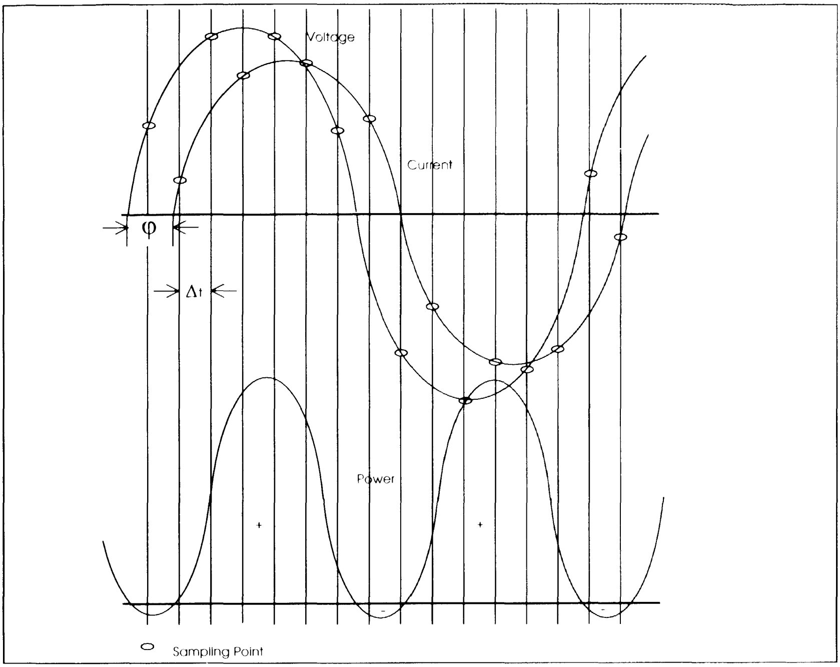


Figure 4.3: Measurement Principle

The measured energy W is:

$$W = \sum_{i=0}^{i=\infty} (u_n \times i_{n-1} + u_n \times i_{n+1}) \times \Delta t \quad = \quad \sum_{i=0}^{i=\infty} u_n \times (i_{n-1} + i_{n+1}) \times \Delta t$$

with:	W	Accumulated energy [Ws]
	u_n	Voltage sample at time t_n
	i_{n-1}	Current sample at time t_{n-1}
	i_{n+1}	Current sample at time t_{n+1}
	Δt	Sampling interval between voltage and current measurements

The "Reduced Scan Principle" has a small inherent error caused by the phase shift, alternately inductive and capacitive, due to the time interval between voltage and current measurements. The value of this error e is:

$$e = [\cos(Dt \times f \times 2p) - 1] \times 100$$

with:	e	Error in per cent

Δt	Sampling interval between voltage and current measurements
f	Mains frequency

For example: for a system with ($f = 50$ Hz, $\Delta t = 150\,\mu s$) the inherent error is 0.111%. This error can be eliminated during runtime by multiplication of the accumulated sum by the correction constant c:

$$c = \frac{1}{\cos(\Delta t \times f \times 2\pi)}$$

The correction factor c is normally included in the calibration constants and not used explicitly.

The advantages of the "Reduced Scan Principle" are:

- Only 50% measurements are necessary because every measured current or voltage value is used twice
- Only 50% multiplications are necessary because two current values are added before multiplication
- Only one Analog-to-Digital-Converter is needed compared to two per phase with the normal method.
- The computing power gained by reducing the number of multiplications can be used by the microcomputer for other system jobs: the MSP430 does the work of the frontend and the host computer.

4.1.2 Single Phase Electricity Meters

The next two Electronic Electricity Meter proposals are made for the measurement of European mains. From the utility one phase and ground are led into the house. In this way a nominal voltage of 230 V is available.
To measure the electric energy consumed a current transformer or a shunt resistor is necessary: both solutions are shown. The voltage of the phase is also measured. With this configuration the energy consumption of the load can be exactly measured.

The Analog-to-Digital-Converter (ADC) of the MSP430 measures the voltage between its V_{SS} and V_{CC} connections with a resolution of 14 bits. To shift the signed voltages coming from the current transformer and voltage divider into the unsigned range of the ADC a split power supply with +2.5 V and -2.5 V is used: the common ground of these two power supplies has a voltage of one half of the voltage SV_{CC}. This voltage is used as a base for the ADC voltages. The MSP430 measures this base voltage at regular intervals and subtracts it from every measured current or voltage sample: in this way signed measurement is possible.

The ultra low current consumption of the MSP430 allows a very small power supply and battery operation:

- Run Mode: 1.4 mA max. @ 5 V (1 MHz, 25°C, MSP430C323)
- Low Power Mode: 5 µA max. @ 5 V (LCD active, CPU halted, 25°C, MSP430C323)

Any customized LCD can be connected to the MSP430 as long as it meets the electrical specifications (max. capacitance per select and common lines, for example). Every segment of the LCD can be controlled independently of the other ones: all 256 (static, 2MUX and 4MUX) and 512 (3MUX) segment combinations are possible.

The EEPROM contains data that must not be lost during power down cycles:

- Calibration data
- Meter number and other device related numbers
- Accumulated energy (stored in regular intervals e.g. every hour)
- Phase error of the current transformer (error = f(I))
- Other data

Depending on the amount of data to be stored an EEPROM with 128 x 8 bit or with 256 x 8 bit is used.

The solution which uses a current transformer for the measurement of the load current is shown in Figure 4.4. The secondary current $I_{secondary}$ of the transformer, which is

$$I_{secondary} = \frac{w_{primary}}{w_{secondary}} \times I_{primary}$$

flows through two paralleled resistors and generates a voltage $U_{secondary}$ which is measured by the MSP430. For currents greater than a certain value the resistor with the lower value is switched on by the analog switch TLC4016I; for low currents this switch is opened to get a higher voltage and therefore a better resolution.
If needed, additional current ranges can be implemented (three analog switches of the TLC4016I are not used).

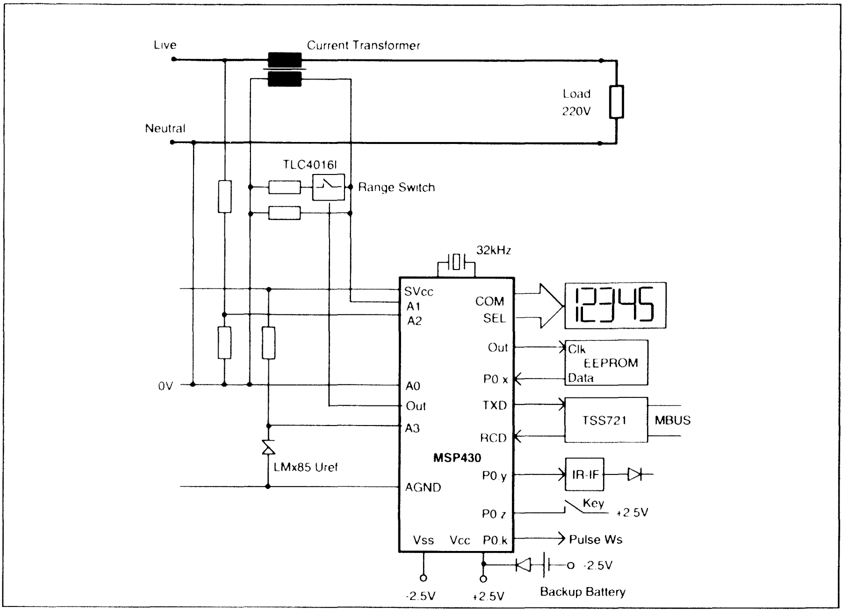


Figure 4.4: Electricity Meter with Current Transformer

The solution which uses a shunt resistor for the measurement of the load current is shown in Figure 4.5. The load current I_{Load} flows through the shunt which has a resistance of approx. 30 mΩ. The voltage drop at the shunt is amplified and measured by the MSP430. The voltage U_{ADC} seen at the ADC of the MSP430 is:

$$U_{ADC} = k \times R_{Shunt} \times I_{Load}$$

with:		
	U_{ADC}	Voltage at the ADC input
	k	Amplification of the operational amplifier
	R_{Shunt}	Resistance of the shunt resistor
	I_{Load}	Load current

If needed, additional current ranges can be implemented (three analog switches of the TLC4016I are not used).

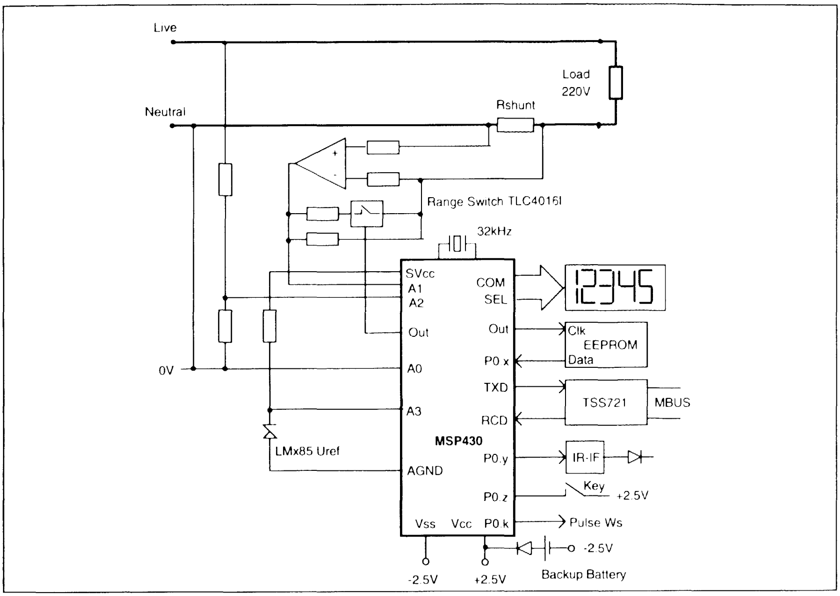


Figure 4.5: Electricity Meter with Shunt Resistor

To have a reference for the measurements a reference diode LMx85 is used. The voltage of this diode is measured in regular intervals and the measured value is used as a base for the SV_{CC} relative ADC measurements.
No reference diode is necessary if voltage regulators are used with the necessary accuracy and long term stability.
The reference used should have a long term stability better than twice the needed accuracy.

Figure 4.6 shows a single phase electricity meter that uses a shunt for the current measurement. The electricity meter shown was built up for demonstration purposes and for measurements. The demonstration board shows an error of less than 1% in the power range from 23 W to 2800 W.
The voltage of the shunt resistor is shifted into the ADC range by the Current Source.
The offset error of the voltage path is eliminated by two analog switches (4066): in regular time intervals (e.g. every minute) one voltage measurement is omitted and the ADC result of the voltage divider R_1/R_2 is measured instead. The load voltage is disconnected by the analog switches during this measurement. The measured ADC result is the zero point and is subtracted from every voltage measurement.

If the voltage and current samples contain offsets then the equation for the measured energy W is:

$$W = \sum_{t=0}^{t=\infty} (u_u + O_u) \times (i_u + O_i) \times \Delta t$$

$$W = \sum_{t=0}^{t=\infty} (u_u \times i_u + u_u \times O_i + i_u \times O_u + O_i \times O_u) \times \Delta t$$

with: O_u Offset of voltage measurement

O_i Offset of current measurement

The terms (u_u x O_i) and (i_u x O_u) get zero when summed-up over one full period (the integral of a sine from 0 to 2π is 0) but the term (O_i x O_u) is added erroneously to the sum buffer with each sample result. If one of the two offsets can be made zero then the error term (O_i x O_u) is eliminated: This is the case due to the regular measurement of the voltage offset value O_u.

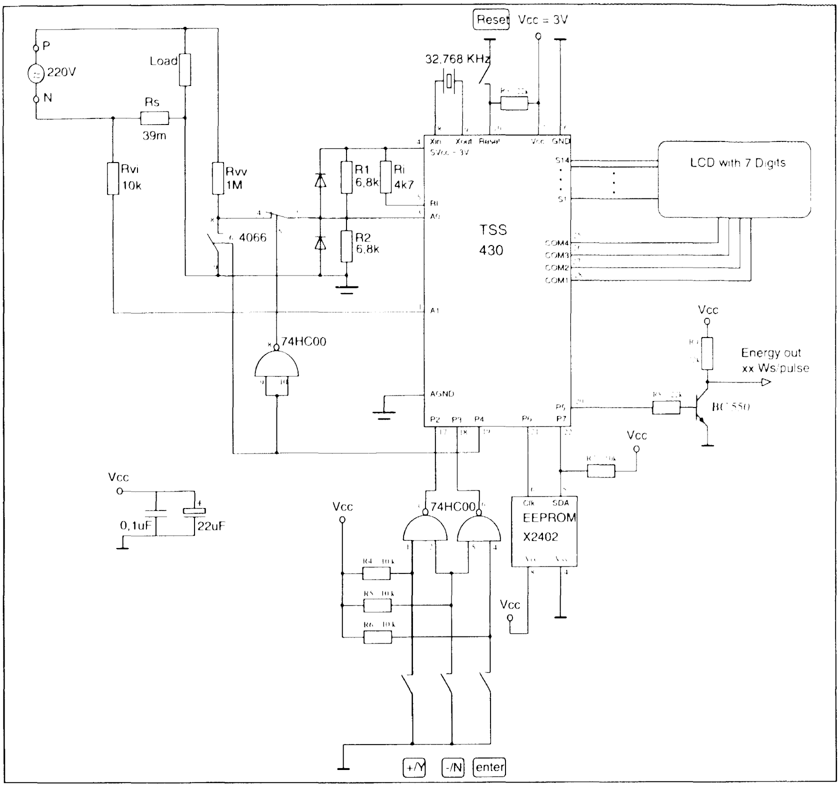


Figure 4.6: Single Phase Electricity Meter

4.1.3 Two Phase Electricity Meter

An Electronic Electricity Meter is shown for the measurement of US domestic mains. As power connections two phases and ground are led into the house. This allows the use of two voltages: 120 V and 240 V.

To measure the electric energy used two current transformers are necessary. The voltage of each phase is measured directly. With this configuration the energy consumption of any load connection can be measured exactly: loads from any phase to ground (120 V) are measured as well as loads connected between the two phases (240 V).

Voltage measurement: the voltage of each phase is adapted to the ADC range by a simple voltage divider.

Power factor measurement: The phase angle φ between voltage and current can be measured as a background task.

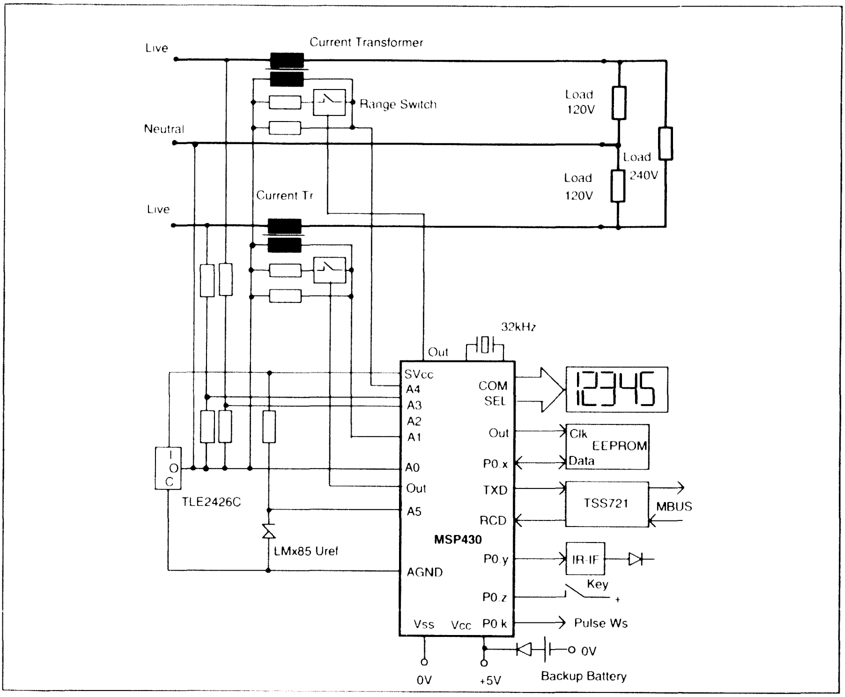


Figure 4.7: Electricity Meter with Current Transformers and virtual Ground

Two current transformers are necessary if loads are possible with all three existing voltages (2 x 120 V, 240 V). The secondary current $I_{secondary}$ of the transformer, which is

$$I_{secondary} = \frac{w_{primary}}{w_{secondary}} \times I_{primary}$$

flows through two parallel resistors and generates a voltage $U_{secondary}$ which is measured by the MSP430. For currents exceeding a certain value the resistor with the lower resistance is switched into the signal path additionally by the analog switch TLC4016I. For low currents this switch is opened to get a higher voltage and therefore a better resolution.
If needed, additional current ranges can be implemented (two analog switches of the TLC4016I are not used).

The "Virtual Ground" IC TLE2426C is used to get a measurement reference in the middle of the ADC range (AGND to SV_{CC}). All current and voltage inputs are referenced to the "Virtual Ground" output of this circuit. The main advantage is the possibility of measuring the ADC value of this reference point without the necessity of switching off the voltage and current inputs.
The measured value (at analog input A0) is subtracted from every measured current or voltage sample which gives signed results.

Instead of the virtual ground circuit TLE2426C two voltage regulators with output voltages of +2.5 V and -2.5 V may be used. In this case the common zero is the reference for all current and voltage measurements and is connected to the analog input A0.
The schematic is shown in Figure 4.8.

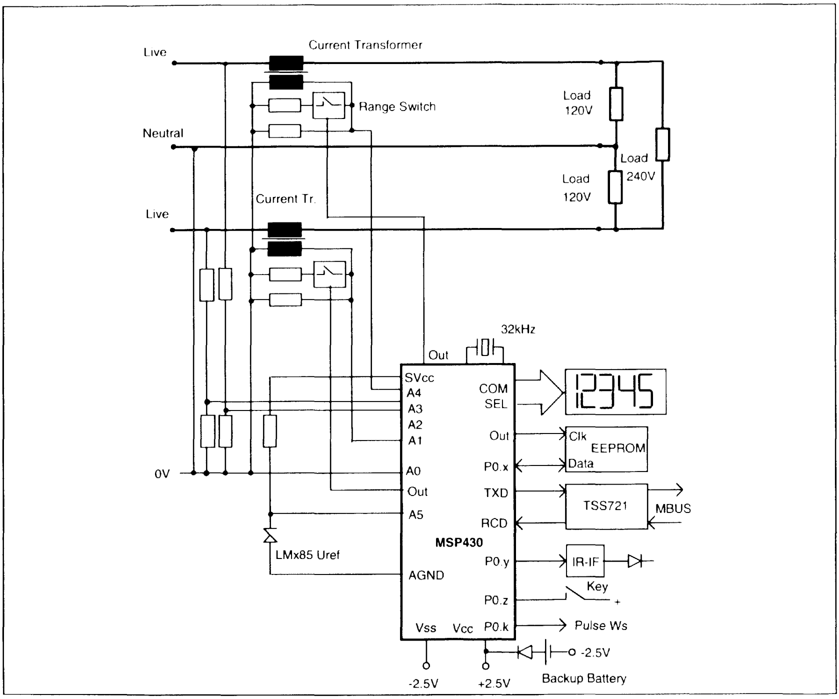


Figure 4.8: Electricity Meter with Current Transformers and split Power Supplies

The M-Bus interface allows the connection of the electricity meter to networks. The M-Bus interface uses the on-chip UART.

Applications of the M-Bus interface:

1. Calibration: Connection to the calibration hardware
2. Automatic readout by a host: The actual consumption and other interesting values may be read out.
3. Tariff switching
4. Test: Start of ROM-based testing routines

Instead of the M-Bus any other bus may be used with the MSP430.

The Infrared Interface IR-IF allows bi-directional data transfer for calibration, test and readout.

To have a reference for the measurements a reference diode LMx85 is used. The voltage of this diode is measured in regular intervals and the measured value is used as a base for the SV_{CC} relative ADC measurements.
No reference diode is necessary if a +5 V voltage regulator is used with the necessary accuracy and long term stability.
The stability of the reference should be better than factor 2 of the desired accuracy of the electricity meter.

Some options are shown for interfacing the MSP430 to other devices:

- Pulse Output: This output changes its state when a certain energy amount is consumed. Usable during calibration or accuracy checks. Mechanical displays can also use this pulse output.
- Key Interface: Keys can be interfaced very simply to the inputs of the MSP430.

4.2 Gas Meter

A gas meter is shown that contains all peripherals which modern gas meters may have. The volume interface is shown for a mechanical meter, and on the left side for an electronic solution:

- The mechanical interface uses contacts to give the volume information to the MSP430. The output Oz is used for scanning, reducing this way the current flow if one or more contacts are closed permanently.
- The electronic interface outputs electrical signals to the MSP430 as long as the enable input is high. The signals V_1 and V_2 are 90° out of phase to allow a reliable distinction of the gas flow direction.

The gas temperature is measured with the ADC of the MSP430: this allows a much better accuracy for the volume measurement, because the dependence of the gas volume to the temperature can be taken into account.

Any combination of the peripherals shown can be used for a given solution: it is not necessary to have all of them implemented.

The MSP430 is normally in Low Power Mode 3 (I = 5 µA nom.), but all enabled interrupt sources will wake it up:

1. Every change of the volume interface if output Oz is high
2. Timing of the Basic Timer: this allows keeping the timing and the scanning if Oz is low due to closed contacts..
3. Actuation of the key
4. M-BUS activity
5. Prepayment interface

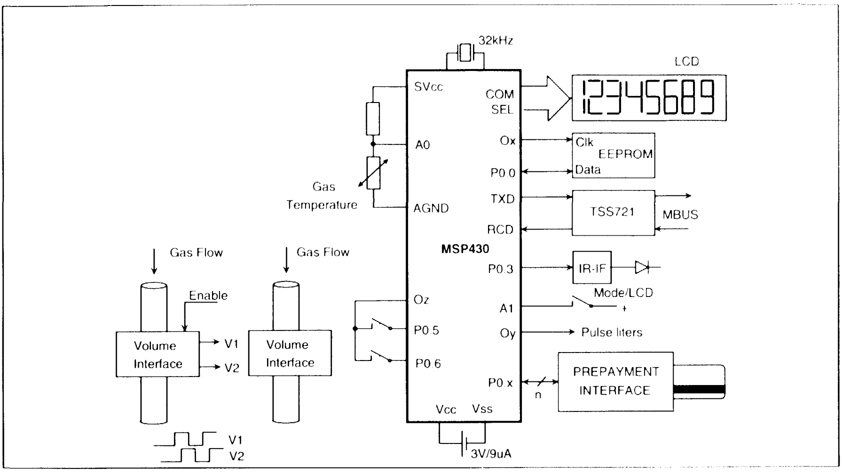


Figure 4.9: Gas Meter with MSP430C32x

The gas meter can be built-up also with the MSP430C31x version. The only difference is the connection of the temperature sensor to the MSP430. The next figure shows this configuration:

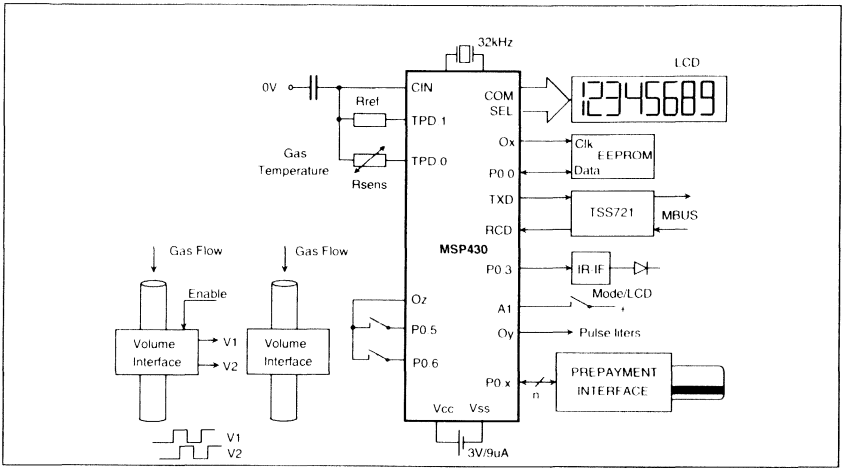


Figure 4.10: Gas Meter with MSP430C31x

4.3 Water Flow Meter

The water flow meter uses an electronic interface to the rotating part of the meter. These signals are 90° out of phase for reliable scanning of direction. The MSP430 is normally in Low Power Mode 3 normally, but every change coming from the volume interface wakes it up.

The water flow meter can be built up also with the MSP430C31x version of the MSP430 family. The only difference is the connection of the sensor for the water temperature. See the above gas meter solution with the MSP430C31x version for details.

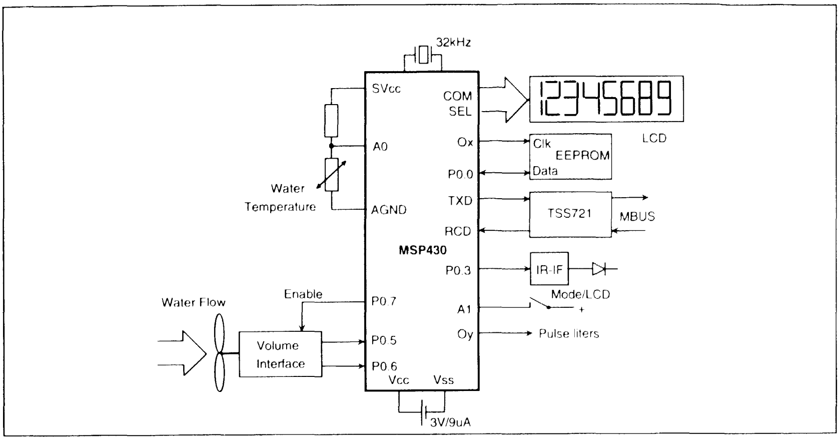


Figure 4.11: Electronic Water Flow Meter

4.4 Heat Allocation Counter

A Heat Allocation Counter with the possibility of sending out the consumption information via RF-frequencies is shown below. The RAM information is scrambled by the DES standard and sent out using the bi-phase code with 19.2 kBaud. The software routines used for the scrambling and the transmission are contained in the section "Data Security".

The heat consumption is computed from the measured room temperature and the heater temperature. The heat consumption is summed up in the RAM and can be read out by the LCD, the M-BUS connection or the RF interface.

The calibration constants and all other important data are contained in the MSP430's RAM. Low Power Mode 3 (CPU off, oscillator on) is used normally; the CPU wakes-up at regular intervals (e.g. 3 minutes), measures the heater and the room temperature, and calculates out of these the actual energy consumption of the radiator. The formulas used take into account the non-linear characteristics given by the thermodynamic theory. This is possible by the use of tables or quadratic or cubic equations.

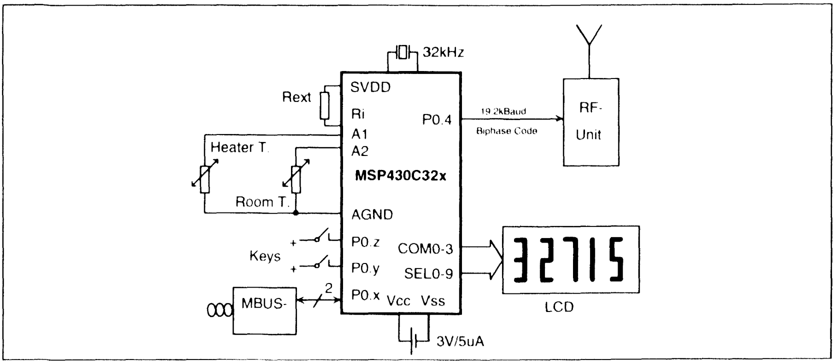


Figure 4.12: Electronic Heat Allocation Meter with MSP430C32x

The heat allocation meter can be built-up also with the MSP430C31x version. Figure 4.13 shows the schematic for this configuration.

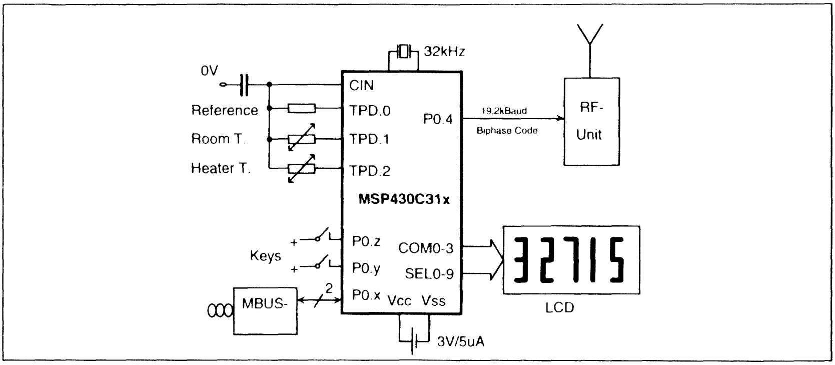


Figure 4.13: Electronic Heat Allocation Meter with MSP430C31x

4.5 Heat Volume Counter

The Heat Volume Counter shown in Figure 4.14 is developed for relatively long sensor lines. An LC-filter is used to prevent spikes and noise at the analog inputs of the MSP430. The system normally runs in Low Power Mode 3 (CPU off, oscillator on) but any change at one of the inputs will wake-up the MSP430.
Every platinum sensor from 100 Ω to 1500 Ω can be used with the MSP430: the Current Source is able to drive them.

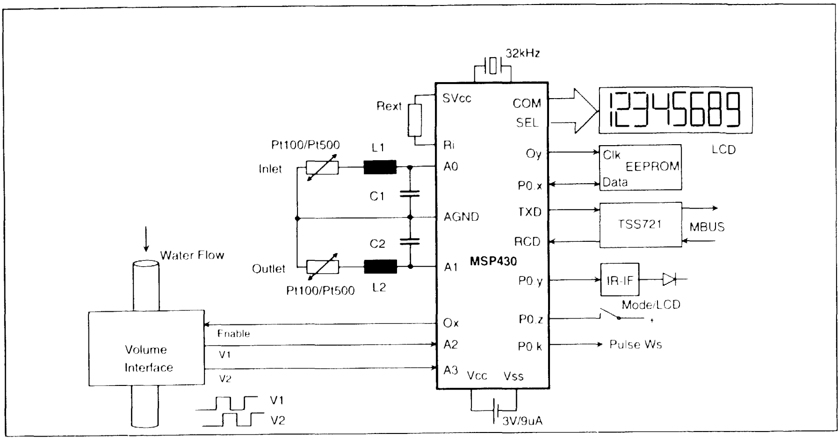


Figure 4.14: Heat Volume Counter MSP430C32x

The Four-Wire circuitry can also be used here. It is possible to use only five analog inputs with the schematic below.

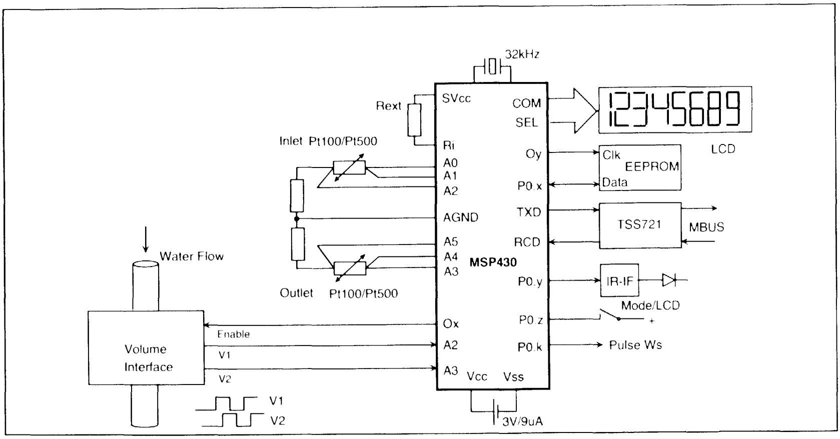


Figure 4.15: Heat Volume Counter with 4-Wire-Circuitry MSP430C32x

Figure 4.15a shows the same heat volume counter as figure 4.15 but with an enlargement of the ADC-resolution to 16 bits. The principle is explained in chapter 2.1.2.5. See there for details of operation.

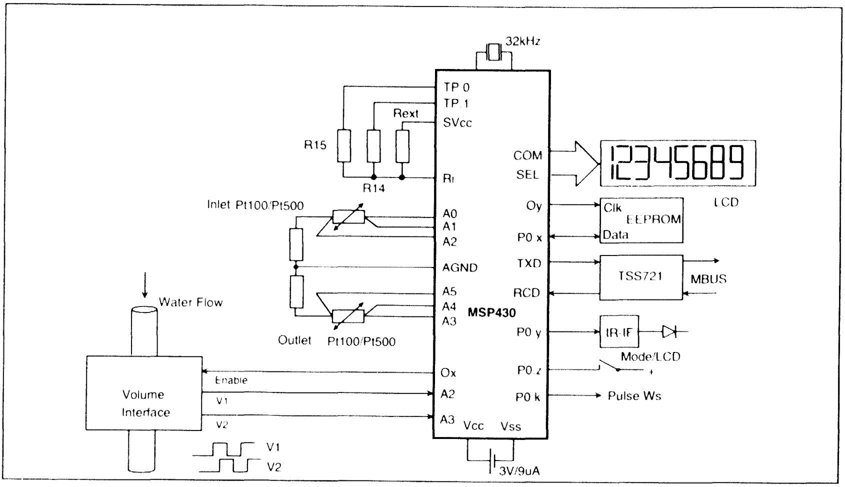


Figure 4.15a: Heat Volume Counter with 16-bits Resolution MSP430C32x

4.6 Battery Charge Meter

The battery charge meter shown below monitors the charge of a battery by means of the measurement of all relevant parameters:

- Battery voltage is measured with the voltage divider R_1/R_2. This voltage is used for the recognition of the end of charge (the battery voltage reduces in a certain manner) and for safety reasons.
- Battery current: the voltage across a shunt gives an exact indication of the current flowing. The low shunt voltage is shifted into the ADC range by a resistor R_3 using the Current Source of the MSP430. The battery current is measured signed (positive sign means charge, negative sign means discharge) to give the possibility of treating charge and discharge currents differently.
- Battery temperature: the resistance of the temperature sensor is measured with the current of the Current Source.

The battery charge meter shown is not restricted concerning the magnitude of voltage, current or capacity of the batteries controlled: these depend only on the design of the shunt resistor, the voltage divider and the calibration constants used. It can be used for cascaded batteries as well as for single ones.

The reference voltage for the system is delivered by the voltage regulator output; the voltage therefore needs to be sufficiently stable. Referencing by a reference diode (LMx85) is also possible. This reference diode may be measured at regular intervals and the result stored. It is not necessary to have the reference always switched on.

The charge indication can be given with a numerical LCD or, as shown below, with a battery symbol showing 20% steps. Other methods for indication are also possible e.g. LED's with different colours that are enabled for a short time by a key stroke.

The voltage regulator needs to have a very low supply current, not exceeding some micro amps. This is necessary due to the long periods the system can be in rest mode (no load). The charge part shown is not necessary for all applications; it can be omitted if, for example, the available space is not provided.

The charge transistor Q_1 is switched on by the MSP430 if a certain charge level is reached. The charge current can be fine tuned by PWM. If the charge current is above the maximum current the transistor is switched off due to safety reasons.

The host connection (for example via RS232 using the MSP430's UART) can be used for the transfer of data: charge, temperature, voltage, current and other system related data. In the other direction the host can transfer instructions: stop or start of charge, start of data transmission etc.

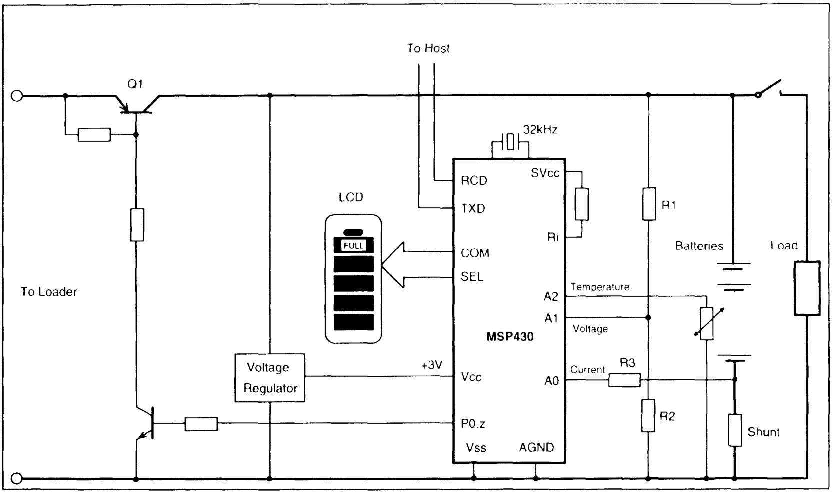


Figure 4.16: Battery Charge Meter MSP430C32x

4.7 Connection of Sensors

4.7.1 Different Ways to connect Sensors

Figure 4.17 shows the connection of simple resistive sensors to the MSP430C32x. The Current Source resistor R_{ext} needs to be calculated in a way that allows its use for both sensor circuits (R_{sens2} and R_{sens3}).

The ways of connection shown in figure 4.17 are described in detail in chapter 2.

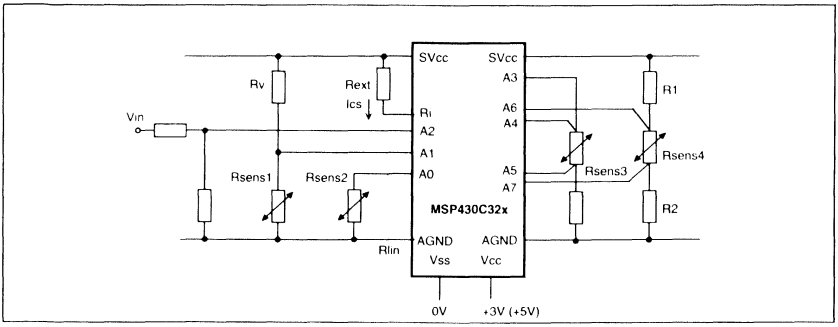


Figure 4.17: Resistive Sensors connected to MSP430C32x

4.7.1.1 Voltage Supply

The sensor Rsens1 in figure 4.17 is connected this way. Resistor R_v supplies the sensor and is used for the linearization too. The optimum value of R_{sens1} is:

$$R_v = \frac{R_m \times (R_u + R_o) - 2 \times R_u \times R_o}{R_u + R_o - 2 \times R_m}$$

Where:

R_u Sensor resistance at the lower temperature limit T_u

R_o Sensor resistance at the upper temperature limit T_o

R_m sensor resistance at the medium temperature $(T_o - T_u)/2$

The ADC values measured are independent of the supply voltage V_{CC} because the measurements are made relative to V_{CC}.

4.7.1.2 Current Supply

Sensor R_{sens2} in figure 4.17 is connected this way. If a linearization of the sensor is wished the same formula used for the voltage supply may be used for the resistor R_{lin}. See above

4.7.1.3 Use of Reference Resistors

Two measurement methods with reference resistors are possible: the use of one reference resistor and the use of two reference resistors:

1. **Measurement with one reference resistor:** the reference resistor is chosen in a way that it equals the sensor resistance at the most important measurement point. Eventually sensor and reference resistor are selected as pairs. The offset error is eliminated completely this way, only the slope error needs to be corrected.
2. **Measurement with two reference resistors:** the two reference resistors represent the sensor resistances at the limits of the measurement range. This method corrects

also the influence of the internal resistance (R_{DSon} of the outputs (MSP430C31x). If sensors and reference resistors are paired, no calibration is necessary with this method.

With two reference resistors R_{ref1} and R_{ref2} it is possible to compute slope and offset and to get the value of an unknown resistors R_x exactly:

$$R_x = \frac{N_x - N_{ref2}}{N_{ref2} - N_{ref1}} \times \left(R_{ref2} - R_{ref1}\right) + R_{ref2}$$

with:	N_x	ADC conversion result for R_x
	N_{ref1}	ADC conversion result for R_{ref1}
	N_{ref2}	ADC conversion result for R_{ref2}
	R_{ref1}	Resistance of R_{ref1}
	R_{ref2}	Resistance of R_{ref2}

As shown only known or measurable values are needed for the computation of R_x from N_x. Slope and offset of the ADC are corrected automatically.

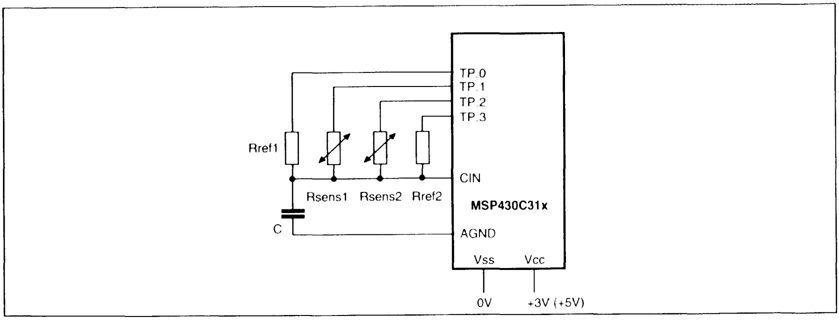


Figure 4.18: Measurement with Reference Resistors (MSP430C31x)

4.7.1.4 Connection of Bridge Assemblies

This kind of sensors is best known for pressure measurement: the voltage difference of the bridge legs changes with the pressure to be measured.

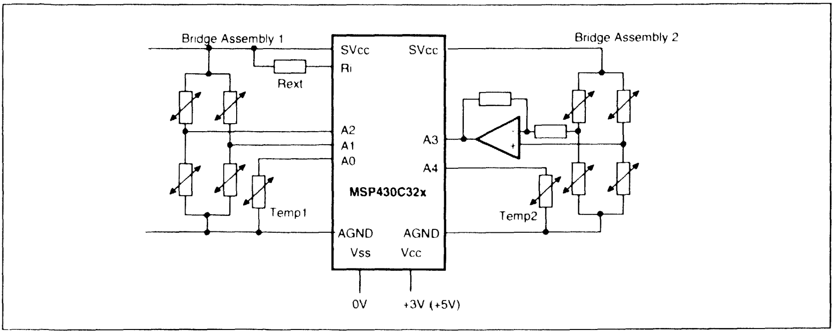


Figure 4.19: Connection of Bridge Assemblies

Figure 4.19 shows in its left part a bridge assembly that creates a voltage difference that is big enough to be measured by the ADC of the MSP430. The measurement result is the difference of the two results of the analog inputs A2 and A1. Due to the temperature dependence of most bridge assemblies a compensation of this dependence is necessary. The sensor Temp1 is used therefore to measure the temperature of the bridge legs (it is integrated in some bridge assemblies).

The used formulae is:

$$P \;=\; MWP \times (Y_e + (T - T_k) \times T_{ke}) + Y_o + (T - T_k) \times T_{ko}$$

where:

P	Pressure to be measured
MWP	Difference of the measured values at A2 and A1
Y_e	Sensitivity of the pressure sensor
T	Temperature of the sensor
T_{ke}	Temperature coefficient of the sensitivity
Y_o	Offset
T_{ko}	Temperature coefficient of the offset
T_k	Temperature during Calibration (e.g. +25°C)

If the difference of the two measurement results is too small to be used then an opamp as shown in the right part of figure 4.19 may be used.

4.7.2 Connection of Special Sensors

Not only analog sensors can be connected to members of the MSP430 family. Nearly all existing sensors can be connected to the MSP430 in a simple way. The examples following will prove this.

4.7.2.1 Gas Sensors

The right part of figure 4.20 shows the connection of two gas sensors (CH_4, hydrogen, alcohol, carbon monoxide, ozone etc.). The gas sensor at the right side (connected to A0) is supplied by the internal current source of the MSP430C32x, where the current flowing through the sensor is defined by the resistor R_{ext}. The gas sensor shown at the left (connected to A1) owns a load resistance R_L where the output voltage can be measured with the ADC input A1.

Both sensors are heated by a pulse-width modulated voltage. The medium current is 133 mA, the power is 120 mW. The measurement of the sensor resistances is made always during the period without current flow.

The temperature dependence of the sensor is corrected by the measurement of the sensor temperature: this is made by sensor Temp2.

Only the MSP430C32x may be used for this kind of sensors: they are not potential free so the MSP430C31 cannot be used.

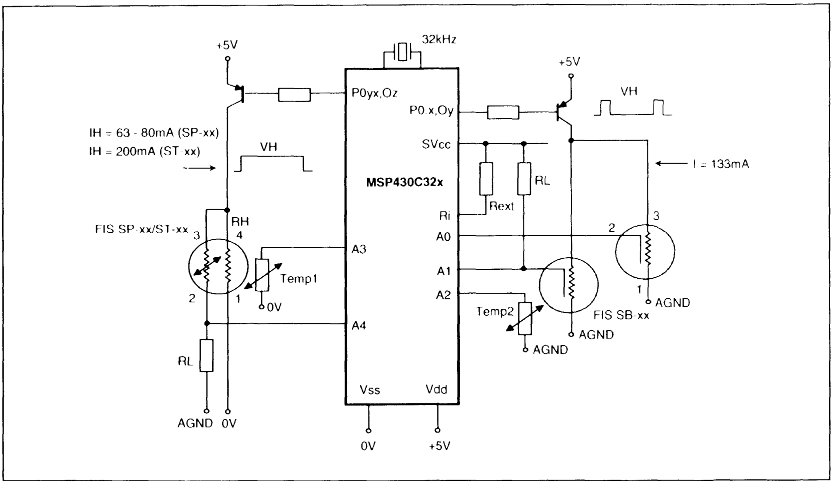


Figure 4.20: Gas Sensor Connection to the MSP430C32x

The left part of figure 4.20 shows the connection of another gas sensor. The heating of the sensor is made here with 5 V DC. The connection is possible only the way shown, therefore the current source cannot be used. Temperature compensation of the measurement result is necessary here too. Sensor Temp1 is used for this purpose.

4.7.2.2 Digital Sensors

Figure 4.21 shows two digital thermometers. They are controlled by instructions via the data bus DQ. The signed measurement result (9 bits) and other internal registers are accessible too via the data bus DQ. The circuit shown left uses a clock line for the data transfer, the right one differs the signals by their length (short is 1, long is 0).

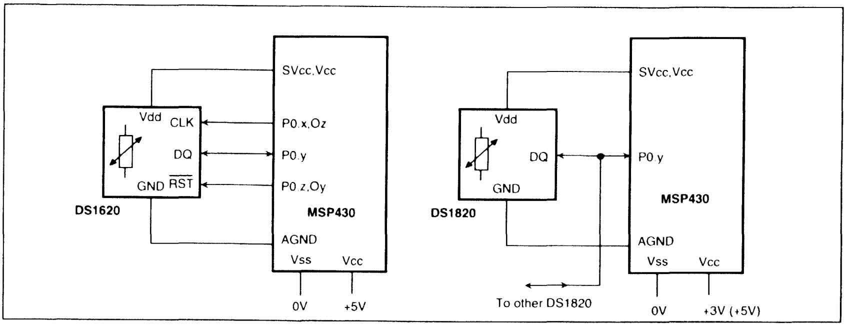


Figure 4.21: Connection of Digital Sensors (Thermometer)

4.7.2.3 Sensors with Frequency Output

The output signal of these sensors is a frequency that is proportional to the measured value. This output frequency can be connected to any of the eight inputs of Port0 and counted via interrupt with a simple software routine. The frequency is the number of interrupts occurring in a one second window defined by the Basic Timer.

If the frequencies to be measured are above 30 kHz then the Universal Timer/Port or the 8-bit Interval Timer/Counter may be used for counting.

The left part of figure 4.22 shows the connection of the linear "Light-Frequency-Converter" TSL220 to the MSP430. The TSL220 outputs a frequency proportional to the incoming light intensity. The range of this output frequency is defined by the capacitor C_f.

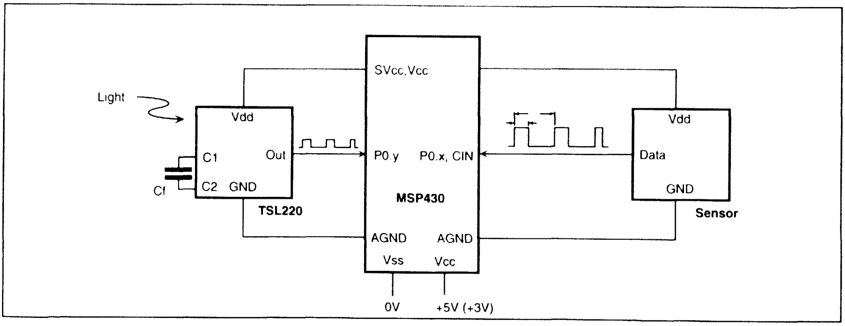


Figure 4.22: Connection of Sensors with Frequency Output resp. Time Output

4.7.2.4 Time Measurements

If the information to be measured is represented by pulse distances or pulse widths then it is also easy to be measured with the MSP430. The right part of figure 4.22 shows how to do this.

The signal to be measured is connected to one of the eight inputs of Port0. Each one of these I/Os allows interrupt on the trailing and on the leading edge. With the Basic Timer an appropriate timing is selected for the needed resolution and the measurement made. The Universal Timer/Port may be used for this purpose too: the pulse to be measured is connected to pin CIN and the time measured from edge to edge.

4.7.2.5 Hall Sensors

Digital hall sensors have an output signal that indicates if the magnetic flux flowing through them is larger or smaller than a certain value. They normally show a hysteresis. Figure 4.23 shows the connection of a revolution counter realized with the TL3101. Every time one of the wings breaks the magnetic flux through the TL3101 a negative pulse is generated and output. These pulses are counted by the MSP430 with interrupt.

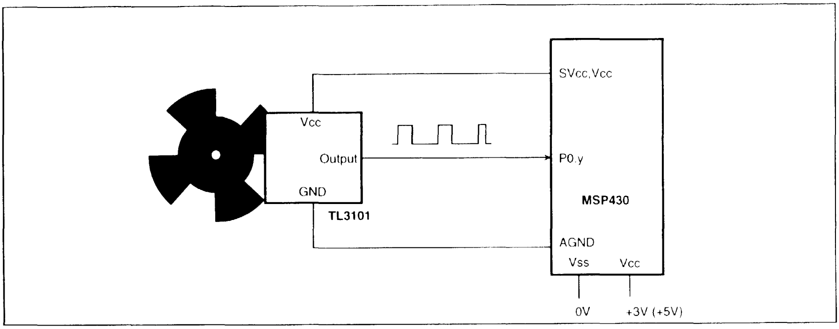


Figure 4.23: Revolution Counter with a Digital Hall Sensor

Analog hall sensors output a signal that is proportional to the magnetic flux through them. For these applications only the MSP430C32x with its 14-bit ADC is usable. During the calibration the ADC value at a known magnetic flux is measured and used for the correction of the slope. The ADC value measured at the magnetic flux zero is subtracted from any measured value. The calculated correction values are stored in the RAM or in an external EEPROM. For the correction of the temperature coefficient of the hall sensor a temperature sensor may be used.

Figure 4.24 shows the connection of an analog hall sensor to the MSP430C32x and the typical output voltage dependent on the magnetic flux.

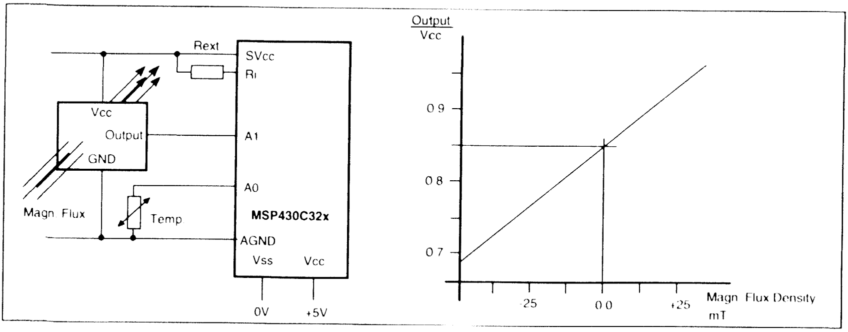


Figure 4.24: Measurement of the magnetic Flux with an Analog Hall Sensor

5 SOFTWARE APPLICATIONS

5.1 Integer Calculation Subroutines

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

1. Speed: Highest speed is possible especially if no loops are used
2. ROM space: Least 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
;
```

All multiplication subroutines shown below permit two different modes:

1. The normal multiplication: the result of the multiplication is placed into the result registers
2. The "Multiplication and Accumulation" function (MAC): the result of the multiplication is added to the previous content of the result registers.

5.1.1 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 use of this subroutine for signed and unsigned multiplication if both are needed. The registers used are shown below:

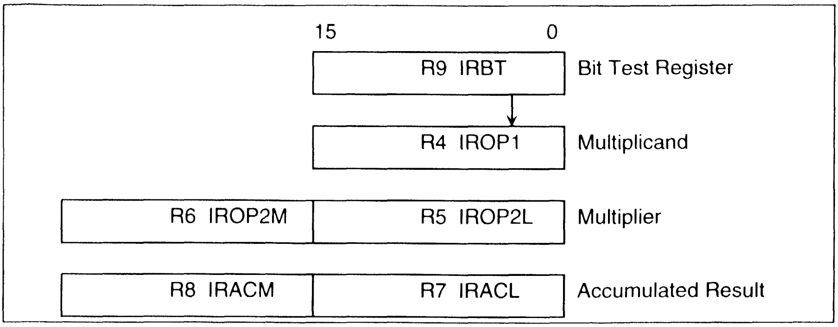


Figure 5.1: 16 x 16-bit Multiplication : Register Use

```
;
; 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
;
```

5.1.2 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; 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 5.1 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
;
```

5.1.3 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:

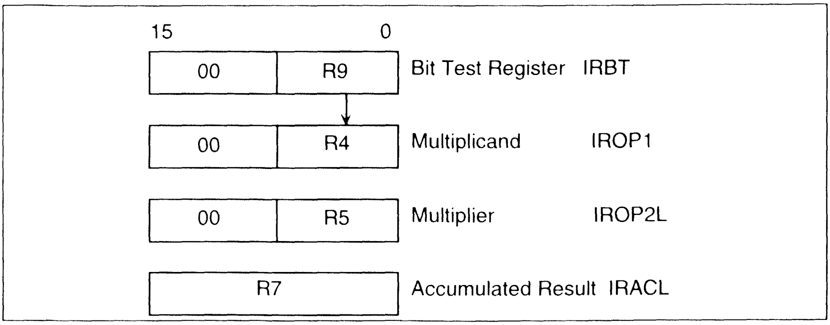


Figure 5.2: 8 x 8-bit Multiplication : Register use

```
;
; 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
;
```

5.1.4 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, 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 5.2 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
;
```

5.1.5 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:

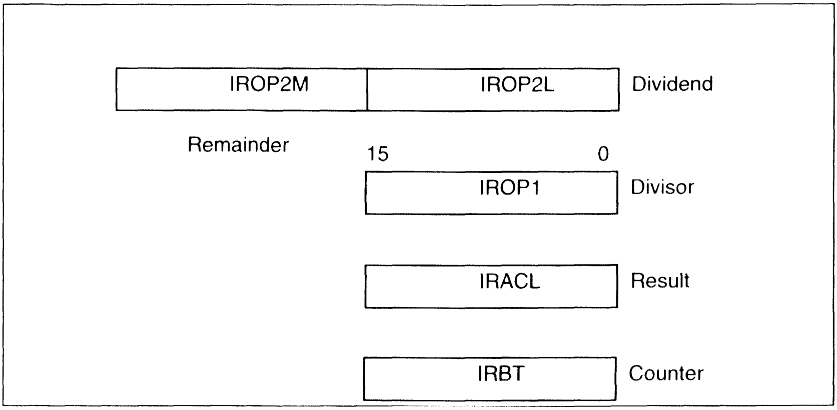


Figure 5.3: Unsigned Division: Register Use

```
;
; DIVISION SUBROUTINE 32-bit BY 16-bit
; IROP2M|IROP2L : IROP1 -> IRACL   REMAINDER IN IROP2M
; RETURN: CARRY = 0: OK    CARRY = 1: QUOTIENT > 16 BITS
;
DIVIDE   CLR      IRACL             ; CLEAR RESULT
         MOV      #17,IRBT          ; INITIALIZE CYCLE COUNTER
DIV1     CMP      IROP1,IROP2M      ;
         JLO      DIV2
         SUB      IROP1,IROP2M
DIV2     RLC      IRACL
         JC       DIV4
         DEC      IRBT              ; Decrement cycle counter
         JZ       DIV4
         RLA      IROP2L
         RLC      IROP2M
         JNC      DIV1
         SUB      IROP1,IROP2M
         SETC
         JMP      DIV2
DIV4     RET
;
```

5.1.6 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 signed
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 unsigned
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
```

5.1.7 Square Root

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

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

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

For m = 2:

$$\sqrt{A} = X$$

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

$$X_0 = {}^{A}\!/_{2}$$

To calculate A/X_n a division is necessary, which is done with the subroutine XDIV. The result of this division has the same integer format as the divisor X_n. 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

;Square Root
;The valid range for the operand is from 0000.0002h to
;7FFF.ffffh
;EXAMPLE:  SQR(2)=1.6a09h
;          SQR(7fff.ffffh)  = B5.04f3h
;          SQR(0000.0002h)  = 0.016ah
;
SQR        .EQU     $
           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, remainder is in R14|R15
;*************************************************************
XDIV
           push     R8        ;save operands onto the stack
           push     R9
           push     R10
           push     R11
           mov      #48,R7    ;counter=48
           clr      R15       ;clear remainder
           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
```

5.1.8 Signed and unsigned 32-bit Compares

The following examples show optimized routines for the comparison of values longer than 16 bits. They can be enlarged to any length (48 bit, 64 bit etc.).

```
; Comparison for unsigned 32-bit numbers: R11|R12 with R13|R14
;
         CMP      R11,R13          ; Compare MSBs
         JNE      L$1              ; MSBs are not equal
         CMP      R12,R14          ; Equality: Compare LSBs too
L$1      JLO      LO               ; Jumps are used for MSBs and LSBs
         JEQ      EQUAL            ;
         ...                       ; R13|R14 > R11,R12
LO       ...                       ; R13|R14 < R11,R12
EQUAL    ...                       ; R13|R14 = R11,R12
```

The shown approach can be adapted to any number length; only additional comparisons have to be added:

```
; Comparison for unsigned 48-bit numbers: R10|R11|R12 with
; R13|R14|R15
;
         CMP      R10,R13          ; Compare MSBs
         JNE      L$1              ; MSBs are not equal
         CMP      R11,R14          ; Equality: Compare MSBs-1 too
         JNE      L$1              ; MSBs-1 are not equal
         CMP      R12,R15          ; Equality: Compare LSBs too
L$1      JLO      LO               ; Jumps are used for all words
         JEQ      EQUAL            ;
         ...                       ; R13|R14|R15 > R10|R11,R12
LO       ...                       ; R13|R14|R15 < R10|R11,R12
EQUAL    ...                       ; R13|R14|R15 = R10|R11,R12
```

```
; Comparison for signed 32-bit numbers: R11|R12 with R13|R14
;
         CMP      R11,R13          ; Compare MSBs signed
         JLT      LO               ; R13 < R11
         JNE      HI               ; Not LO, not EQUAL: only HI rests
         CMP      R12,R14          ; Equality: Compare LSBs too
L$1      JLO      LO               ; LSBs use unsigned jumps!
         JEQ      EQUAL            ; Not LO, not EQUAL: only HI rests
HI       ...                       ; R13|R14 > R11,R12
LO       ...                       ; R13|R14 < R11,R12
EQUAL    ...                       ; R13|R14 = R11,R12
```

```
; Comparison for signed 48-bit numbers: R10|R11|R12 with
; R13|R14|R15
;
         CMP      R10,R13          ; Compare MSBs signed
         JLT      LO
         JNE      HI               ; Not LO, not EQUAL: only HI rests
         CMP      R11,R14          ; Equality: Compare MSBs-1 too
```

```
        JNE     L$1             ; MSBs-1 are not equal
        CMP     R12,R15         ; Equality: Compare LSBs too
L$1     JLO     LO              ; Used for MSBs-1 and LSBs
        JEQ     EQUAL           ; Not LO, not EQUAL: only HI rests
HI      ...                     ; R13|R14|R15 > R10|R11,R12
LO      ...                     ; R13|R14|R15 < R10|R11,R12
EQUAL   ...                     ; R13|R14|R15 = R10|R11,R12
```

5.1.9 Random Number Generation

The linear congruential method is used (introduced by D. Lehmer in 1951). The advantages of this method are speed, simplicity to code, and ease of use. However, if care is not taken in choosing the multiplier and increment values, the results can quickly become degenerate. This algorithm produces 65,536 unique numbers with very good correlation. Therefore the random numbers repeat in the same sequence every 65,536. Within this sequence only the LSB exhibits a repeatable pattern every 16 calls.

The linear congruential method has the following form:

$$Rndnum_n = \left(Rndnum_{n-1} \times MULT \right) + INC(modM)$$

With:	$Rndnum_n$	Current random number
	$Rndnum_{n-1}$	Previous random number
	MULT	Multiplier (unique constant)
	INC	Increment (unique constant)
	M	Modulus (word width of MSP430 = 16 bits = 64K)

Much research has been done to identify the optimal choices for the constants MULT and INC. The constant used in this implementation are based on this research. If changes are made to these numbers, extreme care must be taken to avoid degeneration. Following is a more detailed look at the algorithm and the numbers used:

M M is the modulus value and is typically defined by the word width of the processor. The linear congruential algorithm will return a random number between 0 and 65,535 and is NOT internally bounded. If the user requires a min/max limit, this must be coded externally to this routine. The result is not actually divided by 65,536. The result register is allowed to overflow, thus implementing the modulus.

SEED The first random number in the sequence is called the seed value. This is an arbitrary constant between 0 and 64K. Zero can be used, but the first two results of the generator will be 0 and 1. This is OK if the code is allowed 3 calls to “warm up” before the numbers are taken seriously. The number 21,845 was used in this implementation because it is 1/3 of the modulus (65,536).

MULT Based on random number theory, this number should be chosen such that the last three digits are even-2-1(such as xx821, x421, etc.). The number 31,821 was used in this implementation.

Caution: the generator is extremely sensitive to the choice of this constant!

INC In general, this constant can be any prime number related to M. Two values were actually tested in this implementation: 1 and 13,849. Research shows that INC should be chosen based on the following formula:

$$INC = \left(\frac{1}{2} - \left(\frac{1}{6} \times \sqrt{3}\right)\right) \times M$$

(Using M=65,536 leads to INC=13,849)

The following code describes the first equation. Three subroutines are used to generate random numbers. Furthermore the initialization of corresponding constants and of a RAM-variable storing the random number is included. The symbol names of the 1st equation are strictly used in the code underneath. The first time the initialization routine INIRndnum must be called. Then you can call the subroutine Rndum16 calculating the random numbers as often you want. The necessary code and the description of the subroutine MPYU can be found in "MSP430 Metering Application Guide, Unsigned Multiplication 16 x 16-bit".

```
;
; INITIALIZE CONSTANTS FOR RANDOM NUMBER GENERATION
;
SEED      .set     21845           ;Arbitrary seed value (65536/3)
MULT      .set     31821           ;Multiplier value (last 3
                                   ;digits are even-2-1)
INC       .set     13849           ;1 and 13849 have been tested
;
; ALLOCATION RANDOM NUMBER IN RAM-ADDRESS 200h
;
          .bss     Rndnum,2,0200h
;
; SUBROUTINE: INITIALIZE RANDOM NUMBER GENERATOR:
; LOAD THE SEED VALUE
;
INIRndnum  .equ    $
          mov      #SEED,Rndnum     ; SEED is the first random number
          ret                       ; This RET may be omitted
;
; SUBROUTINE: GENERATES NEXT RANDOM NUMBER
;
Rndnum16   .equ    $
          mov      Rndnum,IROP2L    ; Prepare multiplication
          mov      #MULT,IROP1      ; Prepare multiplication
          call     #MPYU            ; Call unsigned multiplication
          add      #INC,IRACL       ; Add INC to low word of product
;
; Overwrite old random number with low word of new product
;
          mov      IRACL,Rndnum
          ret
;
; SUBROUTINE: UNSIGNED MULTIPLY ROUTINE 16 x 16 bits
; See 5.1.1
;
MPYU      CLR      IRACL            ;Start of multiplication
          ...
```

Algorithm from "TMS320DSP Designer's Notebook Number 43 Random Number Generation on a TMS320C5x". 7/94

5.1.10 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 style 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 to get the result's position.
3. Division: Positions after the decimal point may be different. The two positions are subtracted to get the result's position. (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 shifted appropriately 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     8.8
        MOV     R15,IROP1         ; Operand fetched 2.3
        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
;
```

5.2 Table Processing

One of the development targets of the MSP430 was the capability to process tables. This is due to the fact that software can be written more readably and functionally when using tables. The addressing modes, the instruction set and the word/byte structure make the MSP430 an excellent table processor. The arrangement of information in tables has several advantages:

- Good visibility
- Simple changes: Enlargements and deletions are made easily
- Low software overhead: Short programs
- High speed: Fastest way to access data

Generally, two ways exist of arranging data in tables:

- Data is arranged in blocks, each block containing the complete information of one item
- Data is arranged in several tables, each table containing one or two kinds of information for all items.

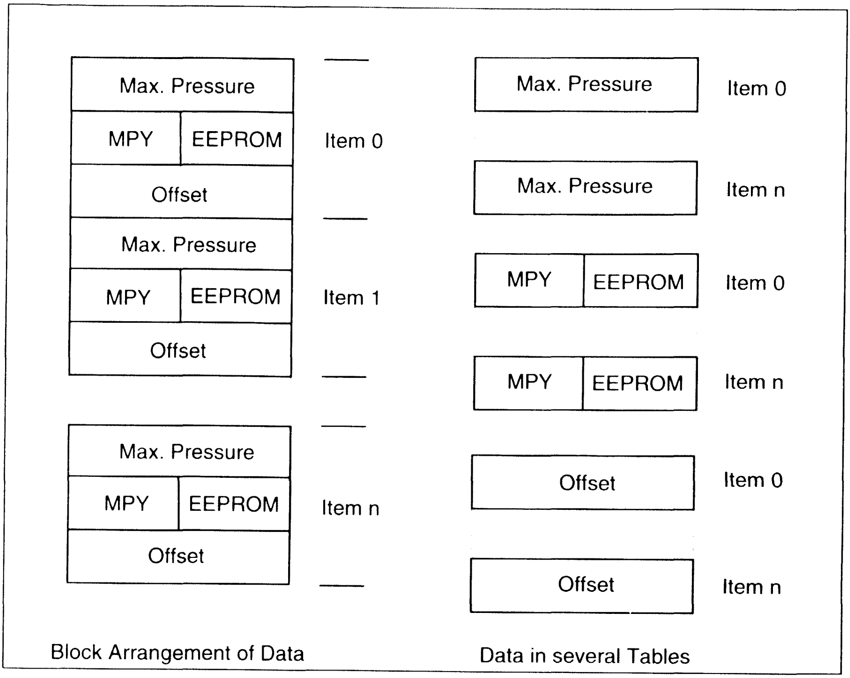


Figure 5.4: Data Arrangement in Blocks

EXAMPLE: A table arranged in blocks is shown. Some examples for random access are given.

```
;
;Block Arrangement of a table
;
TABLE    .WORD    2095         ; Maximum pressure item 0
TEEPR    .BYTE    16           ; EEPROM start address
TMPY     .BYTE    3            ; Multiply constant
TOFFS    .WORD    01456h       ; Offset correction value
;
TABN     .WORD     3084        ; Maximum pressure item 1
         ...
         .WORD    2010         ; Maximum pressure item N
         .BYTE    37           ; EEPROM start address
         .BYTE    3            ; Multiply constant
         .WORD    00456h       ; Offset correction value
;
; Access examples for the above block arrangement:
; R5 points to the 1st word of a block (max. pressure)
; Examples how to access the other values are given:
;
        MOV      @R5,R6                 ; Copy max. pressure to R6
        MOV.B    TEEPR-TABLE(R5),R7     ;EEPROM start to R7
        CMP.B    TMPY-TABLE(R5),R8      ; Same constant as in R8?
```

```
          MOV       &ADAT,R9                        ; ADC result to R9
          ADD       TOFFS-TABLE(R5),R9              ; Correct ADC result
          ADD       #TABN-TABLE,R5                  ; Address next item's block
;
; Copying of block arranged data to registers
;
          MOV       @R5+,R6              ; Copy max. pressure to R6
          MOV.B     @R5+,R7              ; EEPROM start to R7
          MOV.B     @R5+,R8              ; MPY constant to R8
          MOV       @R5+,R9              ; Offset to R9
;
; R5 points to next item's block now
;
; Arrangement of data in several tables
;
TMAXPR    .WORD     2095                 ; Maximum pressure item 0 .WORD
          3084                           ; Maximum pressure item 1 ...
          .WORD     2010                 ; Maximum pressure item N


TEEMPY    .BYTE     16,3                 ; EEPROM start, MPY constant
          .BYTE     37,3                 ; item 1
          ...
          .BYTE     37,114               ; item N
;
TOFFS     .WORD     01456h               ; Offset correction value
          ...
          .WORD     00456h               ; item N
;
;
; Access examples for the above arrangement:
; R5 contains the item number x 2
; Examples with identical functions as for the block arr.
;
          MOV       TMAXPR(R5),R6        ; Copy max. pressure to R6
          MOV.B     TEEMPY(R5),R7        ;EEPROM start to R7
          CMP.B     TMPY+1(R5),R8        ; Same constant as in R8?
          MOV       &ADAT,R9             ; ADC result to R9
          ADD       TOFFS(R5),R9         ; Correct ADC result
          INCD      R5                   ; Address next item
;
```

5.2.1 Two dimensional Tables

Often the output value of a function depends on two (or more) input values. If there is no algorithm for such a function, then a two (or more) dimensional table is needed. Examples of such functions are:

- The entropy of water depends on the inlet temperature and the outlet temperature. An approximation equation of the twelfth order is needed for this problem if no table is used.
- The ignition angle of an Otto-motor depends on the throttle opening and the motor revolutions per minute.

Figure 5.5 shows a function such as described. The output value T depends on the input values X and Y.

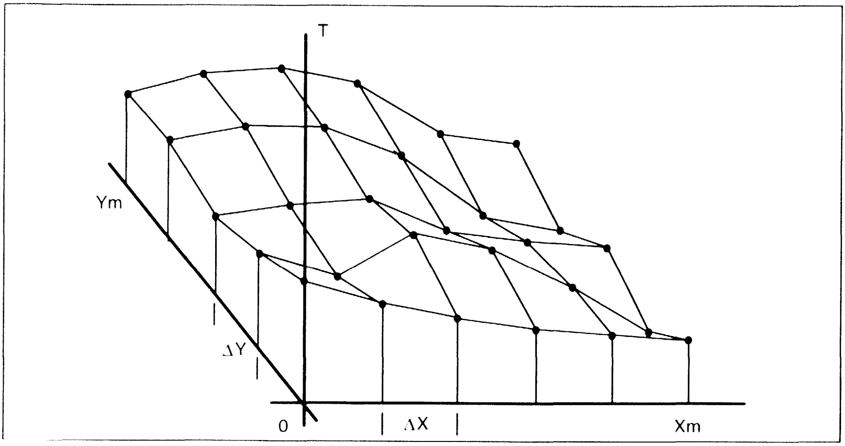


Figure 5.5: Two-dimensional Function

A table contains the output values T for all crossing points of X and Y that have distances of ΔX and ΔY respectively. For every point in between these table points, the output value can be calculated.

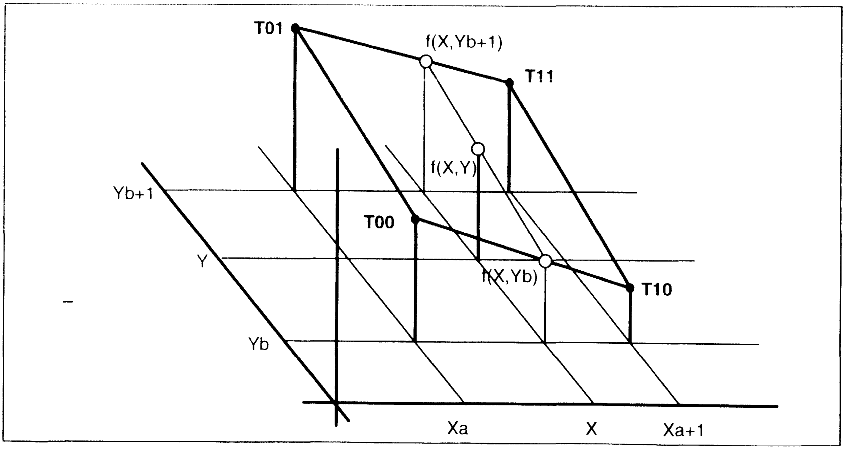


Figure 5.6: Algorithm for two-dimensional Tables

The calculation formulas are:

$$f(X,Y_b) \;=\; \frac{X-X_a}{X_{a+1}-X_a}\times\left(T_{10}-T_{00}\right)+T_{00} \;=\; \frac{X-X_a}{\Delta X}\times\left(T_{10}-T_{00}\right)+T_{00}$$

$$f(X,Y_{b+1}) \;=\; \frac{X-X_a}{\Delta X}\times\left(T_{11}-T_{01}\right)+T_{01}$$

$$f(X,Y) \;=\; \frac{Y-Y_b}{\Delta Y}\times\left(f(X,Y_{b+1})-(f(X,Y_b)\right)+f(X,Y_b)$$

These formulas need division. There are two possible ways of avoiding the division:

- To choose the values for ΔX and ΔY in such a way that simple shifts can do the divisions (ΔX = 0.25, 0.5, 1, 2, 4 etc.)
- To use adapted output values T' within the table

$$T'_{xy} = \frac{T_{xy}}{\Delta X \Delta Y}$$

This adaptation leads to:

$$\frac{f(X,Y_b)}{\Delta Y} = (X-X_a)\times\left(T'_{10}-T'_{00}\right)+T'_{00}$$

$$\frac{f(X,Y_{b+1})}{\Delta Y} = (X-X_a)\times\left(T'_{11}-T'_{01}\right)+T'_{01}$$

$$f(X,Y) = \left(Y-Y_b\right)\times\left(\frac{f(X,Y_{b+1})}{\Delta Y}-\frac{f(X,Y_b)}{\Delta Y}\right)+\frac{f(X,Y_b)}{\Delta Y}\times\Delta Y$$

The output value f(X,Y) is calculable now with multiplications only.

EXAMPLE: A 2-dimensional table is given. ΔX and ΔY are chosen as multiples of 2. The integer subroutines are used for the calculations

NOTE

The software shown is not a generic example; it is tailored to the input values given. If other ΔX and ΔY values are used then the adaptation parts and masks have to be changed.

	X	Y	Comment
Delta	2	4	ΔX and ΔY
Input value format	8.2	7.1	Bits before/after dec.point
Starting value	0	0	X_0 resp. Y_0
End value	42	56	X_M resp. Y_N
Input value (RAM, reg)		X_{IN}	Y_{IN} Assembler mnemonic

```
;
; Two dimensional table processing
;
XIN      .EQU     R15         ; unsigned X value, register or RAM
YIN      .EQU     R14         ; unsigned Y value, register or RAM
XM       .EQU     42          ; Number of X rows
YN       .EQU     56          ; Number of Y columns
XCL      .EQU     7           ; Mask for fraction and dX
YCL      .EQU     7           ; Mask for fraction and dY
XAYB     .EQU     R13         ; Rel. address of (XA,YB), register
ZCFLG    .EQU     0           ; Flag: 0: 2-dim    1: 3-dimensional
;
; Address definitions for the 4 table points:
;
T00      .EQU     TABLE             ; (XA,YB)       TABLE(XAYB)
T01      .EQU     TABLE+2           ; (XA,YB+1)     TABLE+2(XAYB)
T10      .EQU     TABLE+(YN*2)      ; (XA+1,YB)     TABLE+(YN*2)(XAYB)
T11      .EQU     TABLE+(YN*2)+2    ; (XA+1,YB+1)   TABLE+(YM*2)+2(XAYB)
;
; Table for two dimensional processing. Contents are signed
; numbers.
;
TABLE    .WORD    01015h,...073A7h ; (X0,Y0) (X0,Y1)...(X0,YN)
         .WORD    02222h,...08E21h ; (X1,Y0) (X1,Y1)...(X1,YN)
         ...
         .WORD    0A730h,...068D1h ; (XM,Y0) (XM,Y1)...(XM,YN)
;
; Table calculation software 2-dimensional. Approx. 700 cycles
;
; Input value X in XIN, Input value Y in YIN
; Result T in IRACL, same format as TABLE contents
;
; Calculation of YB out of YIN. One less adaptation due to
; word table. Relative address of (X0,YB) to IRACL
;
TABCAL2 CLR       IRACM             ; 0 -> Hi result register
        MOV       YIN,IRACL         ; Y -> Lo result register   7.1
        RRA       IRACL             ; Shift out fraction part   7.0
        RRA       IRACL             ; Adapt to dY = 4     6.0
        BIC       #1,IRACL          ; Word address needed
;
; Calculation of XA out of XIN. One less adaptation due to
; word table. Relative address of (XA,YB) to IRACL (T00)
;
        MOV       XIN,IROP1         ; X -> Multiplicand          8.2
        RRA     IROP1               ; Shift out fraction part  8.1
        RRA     IROP1               ; Adapt to dX = 2     8.0
        BIC     #1,IROP1            ; Word address needed
        MOV     #YN,IROP2L          ; Max. Y (YN) to multipl.  5.0
        CALL    #MACS               ; Rel address (XA,YB)        13.0
        MOV     IRACL,XAYB          ; to storage register        13.0
;
        .IF     ZCFLG               ; If 3-dimensional calculation
        ADD     OFFZC,XAYB          ; Add offset for actual table
        .ENDIF                      ; Rel. address of ZC
;
; Calculation of f(X,YB) = (XIN-XA)/dX x (T10-T00) + T00
;
        MOV     XIN,IROP1           ; build (XIN - XA)            8.2
        AND     #XCL,IROP1          ; Fraction and dX rests     1.2
        MOV     T10(XAYB),IROP2L    ; T10 -> IROP2L      16.0
        SUB     T00(XAYB),IROP2L    ; T10 - T00                  16.0
        CALL    #MPYS               ; (XIN - XA)(T10 - T00)   17.2
        CALL    #SHFTRS3            ; :dX, to integer   15.0
        ADD     T00(XAYB),IRACL     ; (XIN-XA)(T10-T00)+T00  15.0
        PUSH    IRACL               ; Result on stack
;
; Calculation of f(X,YB+1) = (XIN-XA)/dX x (T11-T01) + T01
; (XIN-XA) still in IROP1
;
        MOV     T11(XAYB),IROP2L    ; T11 -> IROP2L      16.0
        SUB     T01(XAYB),IROP2L    ; T11 - T01                  16.0
        CALL    #MPYS               ; (XIN - XA)(T11 - T01)   17.2
        CALL    #SHFTRS3            ; :dX, to integer   15.0
        ADD     T01(XAYB),IRACL     ; (XIN-XA)(T11-T01)+T01  15.0
;
; Calculation of f(X,Y) = (YIN-YB)/dY x (f(X,YB)-f(X,YB+1) + f(X,YB)
;
        MOV     YIN,IROP1           ; build (YIN - XB    7.1
        AND     #YCL,IROP1          ; Fraction and dX rests     2.1
        SUB     @SP,IRACL           ; f(X,YB+1)-f(X,YB)          16.0
        MOV     IRACL,IROP2L        ; Result to multiplier
        CALL    #MPYS               ; (YIN-YB)(f..-f..)          18.1
        CALL    #SHFTRS3            ; :dY, to integer   16.0
        ADD     @SP+,IRACL          ; (YIN-YB)(f..-f..)+f..   15.0
        RET                         ; Result T in IRACL          16.0
```

The table used with the example before uses unsigned values for X and Y (the upper left table of figure 5.6a shows this). If X or Y or both are signed values then the structure of the table and its entry point have to be changed. The following examples in figure 5.6a show how to do that.

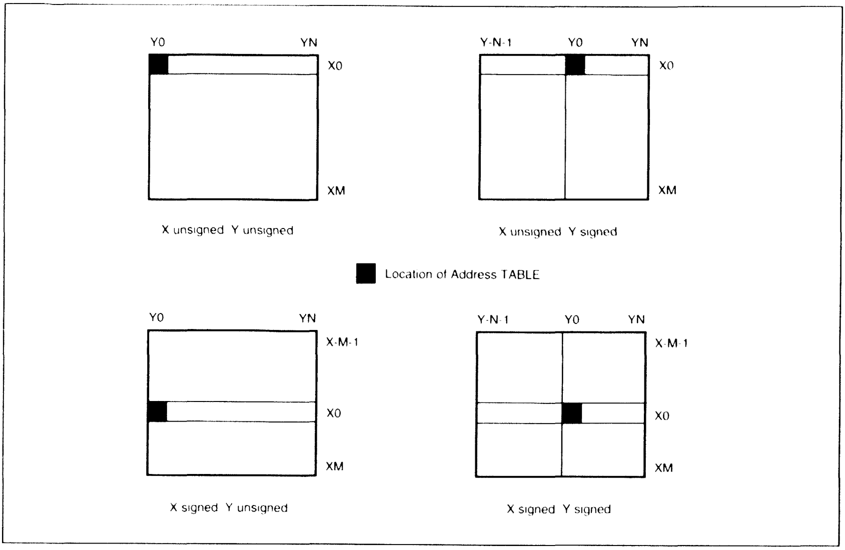


Figure 5.6a: Table Configuration for signed X and Y

The above tables are shown in assembler code:

```
;
; X unsigned, Y unsigned
;
TABLE    .WORD    01015h,...073A7h ;  (X0,Y0)...(X0,YN)
         .WORD    02222h,...08E21h ;  (X1,Y0)...(X1,YN)
         ...
         .WORD    0A73h,...068D1h  ;  (XM,Y0)...(XM,YN)
;
; X unsigned, Y signed
;
         .WORD    03017h,...093A2h ;  (X0,Y-N-1)...(X0,Y-1)
TABLE    .WORD    02233h,...08721h ;  (X0,Y0)...(X0,YN)
         .WORD    03017h,...093A2h ;  (X1,Y-N-1)...(X1,YN)
         ...
         .WORD    00173h,...07851h ;  (XM,Y-N-1)...(XM,YN)
;
; X signed, Y unsigned
;
         .WORD    03017h,...093A2h ;  (X-M-1,Y0)...(X-M-1,YN)
         .WORD    08012h,...0B3C1h ;  (X-M,Y0).....(X-M,YN)
         ...
         .WORD    04019h,...0D3A3h ;  (X-1,Y0)...(X-1,YN)
TABLE    .WORD    02233h,...08721h ;  (X0,Y0)....(X0,YN)
         .WORD    03017h,...093A2h ;  (X1,Y0)....(X1,YN)
         ...
         .WORD    00173h,...07851h ;  (XM,Y0)....(XM,YN)
;
; X signed, Y signed
;
         .WORD    03017h,...093A2h ;  (X-M-1,Y-N-1)(X-M-1,YN)
         .WORD    08012h,...0B3C1h ;  (X-M,Y-N-1) ...(X-M,YN)
         . . .
         .WORD     04019h,...0D3A3h  ;  (X-1,Y-N-1)...(X-1,YN)
         .WORD     02233h,...08721h  ;  (X0,Y-N-1)....(X0,Y-1)
TABLE    .WORD     02233h,...08721h  ;  (X0,Y0).......(X0,YN)
         .WORD     03017h,...093A2h  ;  (X1,Y-N-1)....(X1,YN)
         . . .
         .WORD     00173h,...07851h  ;  (XM,Y-N-1)....(XM,YN)
```

The entry label TABLE always points to the word or byte with the coordinates (X0,Y0).

5.2.2 Three dimensional Tables

If the output value depends on three input variables X, Y and Z, then a three dimensional table is necessary for the crossing points. Eight values T000 to T111 are used for the calculation of the output value T.

The simplest way for the calculation is to calculate the output values for two two-dimensional tables f(X,Y,Z_c) and f(X,Y,Z_{c+1}) with the subroutine TABCAL2 used for the two-dimensional tables. The two results are used for the final calculation:

$$f(X,Y,Z) = \frac{Z - Z_c}{Z_{c+1} - Z_c} \times (f(X,Y,Z_{c+1}) - (f(X,Y,Z_c)) + f(X,Y,Z_c))$$

The next figure shows this method: the output values T are calculated for Z_c and for Z_{c+1}. Out of these two output values the final value is calculated.

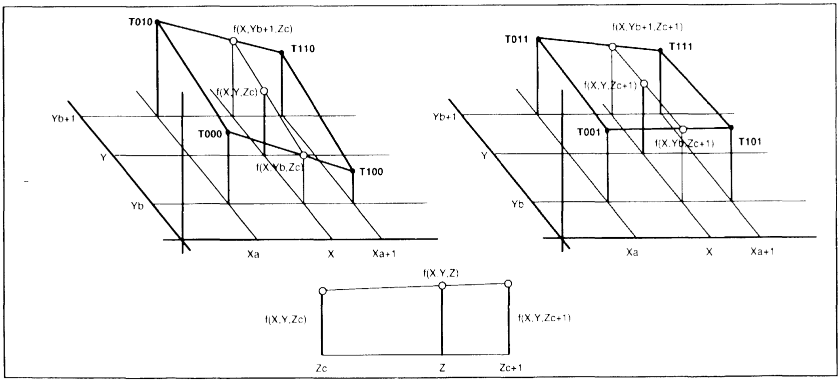


Figure 5.7: Algorithm for a three-dimensional Table

EXAMPLE: A 3-dimensional table is given. ΔX and ΔY and ΔZ are chosen as multiples of 2. The integer subroutines are used for calculations.

	X	Y	Z	
Delta	2	4	256	ΔX, ΔY, ΔZ
Input value format	8.2	7.1	0	Bits after dec.point
Starting value	0	0	0	X_0, Y_0, Z_0
End value	42	56	2^{14}-1	X_M, Y_N, Z_P
Input value (RAM, reg)	XIN	YIN	ZIN	Assembler mnemonic

```
;
XIN      .EQU    R15      ; unsigned X value, register or RAM
YIN      .EQU    R14      ; unsigned Y value, register or RAM
ZIN      .EQU    R13      ; unsigned Z value, register or RAM
XM       .EQU    42       ; Number of X rows
YN       .EQU    56       ; Number of Y columns
XCL      .EQU    7        ; Mask for fraction and dX
YCL      .EQU    7        ; Mask for fraction and dY
ZCL      .EQU    0FFh     ; Mask for deltaZ
XAYB     .EQU    R12      ; Rel. address of (XA,YB), register
ZCFLG    .EQU    1        ; Flag: 0: 2-dim    1: 3-dimensional
OFFZC    .EQU    R11      ; Relative offset to actual (X0,Y0,ZC)
;
; Three dimensional table
;
TABL3D   .WORD   01015h,...073A7h ; (X0,Y0,Z0)...(X0,YN,Z0)
         ...
         .WORD   02222h,...08E21h ; (XM,Y0,Z0)...(XM,YN,Z0)
;
         .WORD   0A730h,...068D1h ; (X0,Y0,Z1)...(X0,YN,Z1)
         ...
         .WORD   010A5h,...09BA7h ; (XM,Y0,Z1)...(XM,YN,Z1)
;
         .WORD   02BC2h,...08E41h ; (X0,Y0,ZP)...(X0,YN,ZP)
         ...
         .WORD   0A980h,...023D1h ; (XM,Y0,ZP)...(XM,YN,ZP)

; Table calculation software 3-dimensional
; Input values: X in XIN, Y in YIN, Z in ZIN
; Result is located in IRACL, same format as TABLE content
;
; Calculation of ZC out of ZIN. One less adaptation due to
; word table.
;
TABCAL3  MOV     ZIN,IROP1          ; Z -> Operand register    14.0
         SWPB    IROP1              ; Use only upper byte (dZ =256)
         MOV.B   IROP1,IROP1        ; Adapt to dZ = 256          6.0
;
; Calculation of relative address of (X0,Y0,ZC) to IRACL
; Corrected for word table
;
         MOV     #YN*2*XM,IROP2L    ; Table length for dZ
         CALL    #MPYU              ; Rel address (X0,Y0,ZC) 13.0
         MOV     IRACL,OFFZC        ; to storage register       13.0
;
; Calculation of f(X,Y,ZC): The table block for ZC is used
;
         CALL    #TABCAL2           ; f(X,Y,ZC) -> IRACL       16.0
         PUSH    IRACL              ; Save f(X,Y,ZC)

; Calculation of f(X,Y,ZC+1): The table block for ZC+1 is used
```

```
;
        ADD      #YN*2*XM,OFFZC     ; Rel. adress (X0,Y0,ZC+1)
        CALL     #TABCAL2           ; f(X,Y,ZC+1) -> IRACL   16.0
;
; Calculation of f(X,Y,Z)
;
        MOV      ZIN,IROP1          ; build (YIN - XB    6.8
        AND      #ZCL,IROP1         ; Fraction and dZ rests    0.8
        SUB      @SP,IRACL          ; f(X,Y,ZC+1)-f(X,Y,ZC)      16.0
        MOV      IRACL,IROP2L       ; Result to multiplier
        CALL     #MPYS              ; (ZIN-ZC)(f..-f..)          16.8
        CALL     #SHFTRS6           ; :dZ, to integer   16.2  CALL
        #SHFTRS2                    ;                            16.0
        ADD      @SP+,IRACL         ; (ZIN-ZC)(f..-f..)+f..   15.0
        RET                         ; Result in IRACL
```

5.3 Signal Averaging and Noise Cancellation

If the measured signals contain noise, spikes and other not wanted signal components then it is necessary to average the ADC results. Four different methods are mentioned here:

1. Oversampling: Several measurements are added-up and the accumulated sum is used for the calculations.
2. Continuous Averaging: A circular buffer is used for the measured samples. With every new sample a new average value can be calculated.
3. Weighted summation: The old value and the new one are added together and are then halved.
4. Wave Digital Filtering: Complex filter algorithms that need only small calculation power are used for the signal conditioning.
5. Rejection of Extremes: the largest and the smallest sample are rejected from the measured values and the remaining ones added-up and averaged.

The advantages and disadvantages of the different methods are shown in the appertaining sections.

5.3.1 Oversampling

Oversampling is the most simple method for the averaging of measurement results: N samples are added-up and the accumulated sum is divided by N afterwards (in time), or is used as it is with the next algorithm steps. It is only necessary to remember that the added-up value is N-times too large. For example the formula below used for a single measurement needs to be modified if N samples are summed-up as shown:

$$V_{normal} = Slope \times ADC + Offset \rightarrow V_{oversample} = \frac{\Sigma(Slope \times ADC + Offset)}{N}$$

EXAMPLE: N measurements have to be summed-up in SUM and SUM+2. The number N is defined in R6

```
;
SUMLO    .EQU    R4            ; LSBs of sum
SUMHI    .EQU    R5            ; MSBs of sum
;
         CLR     SUMLO         ; Init of registers
         CLR     SUMHI
         MOV     #16,R6        ; Sum-up 16 samples of the ADC
OVSLOP   CALL    #MEASURE      ; Result in ADAT
         ADD     &ADAT,SUMLO   ; LSD of accumulated sum
         ADC     SUMHI         ; MSD
         DEC     R6            ; Decr. N counter: 0 reached?
         JNZ     OVSLOP
         ...                   ; Yes, 16 samples in SUMHI|SUMLO
;
```

Disadvantages: - High current consumption due to number of ADC conversions
- Low suppression of spikes etc. (by N)

Advantages: - Simple programming

5.3.2 Continuous Averaging

A very simple and fast way for averaging digital signals is "Continuous Averaging": A circular buffer is fed at one end with the newest sample and the oldest sample is deleted at the other end (both items share the same RAM location). To reduce the calculating time, the oldest sample is subtracted from the actual sum and the new sample is added to the sum. The actual sum (a 32-bit value containing n samples) is used by the background; for calculations it is only necessary to remember that it contains the N samples. The same rule is valid as with oversampling.

The characteristic of this averaging is similar to a "Comb Filter" with relatively good suppression of frequencies that are integral multiples of the scanning frequency. The frequency behaviour is shown in the next figure:

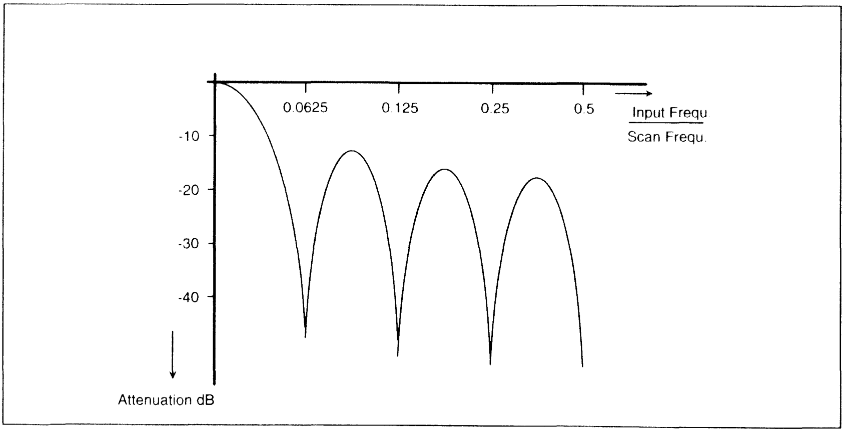


Figure 5.8: Frequency Response of the Continuous Averaging Filter

Disadvantages: – RAM allocation. N words are needed for the circular buffer

Advantages:
- Low current consumption due to one measurement only
- Fast update of buffer
- Good suppression of certain frequencies (multiples of scan frequency)
- Low pass filter characteristic

EXAMPLE: An interrupt driven routine (e.g. from the ADC which is started by the Basic Timer) is shown that updates a circular buffer with N items. The actual sum CFSUM is calculated by subtracting of the oldest sample and adding of the newest one. CFSUM and CFSUM+2 contain the sum of the latest N samples.

```
;
N       .EQU    16                  ; Circular buffer with N items
        .BSS    CFSTRT,N*2          ; Address of 1st item
        .BSS    CFSUM,4             ; Accumulated sum 32 bits
        .BSS    CFPOI,2             ; Points to next (= oldest) item
;
CFHND   PUSH    R5                  ; Save R5
        MOV     CFPOI,R5            ; Actual address to R5
        CMP     #CFSTRT+(N*2),R5    ; Outside circ. buffer?
        JLO     L$300               ; No
        MOV     #CFSTRT,R5          ; Yes, reset pointer
;
; The oldest item is subtracted from the sum. The newest item
; overwrites the oldest one and is added to the sum
;
L$300   SUB     @R5,CFSUM           ; Subtract oldest item from CFSUM
        SBC     CFSUM+2
        MOV     &ADAT,0(R5)         ; Move latest item to buffer
        ADD     @R5+,CFSUM          ; Add latest ADC result to CFSUM
        ADC     CFSUM+2
        MOV     R5,CFPOI            ; Update pointer
        POP     R5                  ; Restore R5
        RETI
;
```

5.3.3 Weighted Summation

The weighted sum of the measurements before and the actual measurement result are added and then divided by two. This gives every measurement a certain weight:

Measurement at t_0:	0.5	Actual measurement
Measurement $t_0 - \Delta t$:	0.25	Last measurement
Measurement $t_0 - 2\Delta t$:	0.125	
Measurement $t_0 - 3\Delta t$:	0.0625	
Measurement $t_0 - 4\Delta t$:	0.03125	
Measurement $t_0 - n\Delta t$:	$2^{-(n+1)}$	

etc.

Disadvantages:
- Suppression of spikes not sufficient (factor 2 only for actual sample)

Advantages:
- Low current consumption due to one measurement only
- Low pass filter characteristic

- Very short code
- Only one RAM word needed

EXAMPLE: The update of the actual sum WSSUM is shown.

```
;
         .BSS     WSSUM,2           ; Accumulated weighted sum
;
WSHND    ADD      &ADAT,WSSUM       ; Add current measurement to sum
         RRA      WSSUM             ; New sum divided by 2
         ...                        ; Continue with value in WSSUM
```

5.3.4 Wave Digital Filtering

Wave Digital Filters (WDFs) have notable advantages:

- Excellent stability properties even under nonlinear operating conditions resulting from overflow and roundoff effects
- Low coefficient wordlength requirements
- Inherently good dynamic range
- Stability under looped conditions

Compared with the often used averaging of measured sensor data, the digital filtering has advantages: Lowpass filtering with sharp cut-off region, notch filtering of noise, ...

For the design of Wave Digital Filter algorithms specialized CAD programs have been designed in order to speed-up the top-down design from filter specification to the machine program for the processor:

- LWDF_DESIGN allows the design of Lattice-WDFs
- LWDF_COMP transforms a Lattice-WDF structure into an assembler program for the MSP430
- DSP430 allows fast transient simulations of the filter algorithms on a model of the MSP430, analysis of frequency response, check of accuracy and stability proof.

The programs enable the users of the MSP430 to solve special measurement problems by means of robust digital filter algorithms.
A complete description of the WDF algorithms and development tools will be given in the "TEXAS INSTRUMENTS Technical Journal" November/December 1994.

Disadvantages:
- Complex algorithm. Support software needed for finding algorithm
- Low current consumption due to one measurement only per time slice

Advantages:
- Good attenuation inside stopband
- Good dynamic stability

5.3.5 Rejection of Extremes

This averaging method measures (N+2) ADC-samples and rejects the largest and the smallest values. The remaining N samples are added-up and the accumulated sum is divided by N afterwards or is used as it is with the next algorithm steps. It is only necessary to remember that the added-up value is N-times too large.

Disadvantages:	– Current consumption due to (N+2) ADC conversions
Advantages:	– Simple programming
	– Very good suppression of spikes (extremes are rejected)
	– Low RAM needs (4 words)

The software example below adds six ADC samples, subtracts the two extremes and returns with the sum of the four medium samples. The constant N may be changed to any number, but the summing-up buffer SESUM needs two words if N exceeds two. It is an advantage to use powers of two for N due to the simple division if needed (right shifts only). Register use is possible too for SESUM, SEHI and SELO.

```
;
N          .EQU     4                  ; Sample count used -2
           .IF      N>2
           .BSS     SESUM,4            ; Summing-up buffer
           .ELSE                       ;
           .BSS     SESUM,2            ; N<=2
           .ENDIF
           .BSS     SEHI,2             ; Largest ADC result
           .BSS     SELO,2             ; Smallest ADC result
           .BSS     SECNT,2            ; Counter for N+2
;
SEHND      CLR      SESUM              ; Initialize buffers
           MOV      #N+2,SECNT         ; Sample count +2 to counter
           MOV      #0FFFFh,SELO       ; ADCmax -> SELO
           CLR      SEHI               ; ADCmin -> SEHI
;
; N+2 measurements are made and summed-up in SESUM
;
SELOOP     CALL     #MEASURE ; ADC result to &ADAT
           MOV      &ADAT,R5 ; Copy ADC result to R5
           ADD      R5,SESUM
           .IF      N>2                ; Use 2nd sum buffer if N>2
           ADDC     SESUM+2
           .ENDIF
           CMP      R5,SEHI            ; Result > SEHI?
           JHS      L$1                ; No
           MOV      R5,SEHI            ; Yes, actualize SEHI
L$1        CMP      R5,SELO            ; Result < SELO?
           JLO      L$2                ; No
           MOV      R5,SELO
L$2        DEC      SECNT              ; Counter - 1
           JNZ      SELOOP             ; N+2 not yet reached
;
; N+2 measurements are made, extremes are subtracted now
; from summed-up result. Return with N-times value in SESUM
;
           SUB      SELO,SESUM         ; Subtract lowest result
           .IF      N>2                ; Necessary if N>2
           SBC      SESUM+2
           .ENDIF
           SUB      SEHI,SESUM         ; Subtract highest result
```

```
          .IF       N>2                 ; Necessary if N>2
          SBC       SESUM+2
          .ENDIF
          RET
;
```

5.3.6 Synchronization of the Measurement to Hum

If hum plays a role during measurements then a synchronization to the power frequency may help to overcome this problem. Fig. 5.8.1 shows the influence of the mains voltage during the measurement of a single sensor. The necessary number of measurements (here 10) is split into two equal parts, the second part is measured after exactly one half of the period T_{MAIN} of the power frequency. The hum introduced to the two parts is equal but has different signs. Therefore the accumulated influence (the sum) is nearly zero.

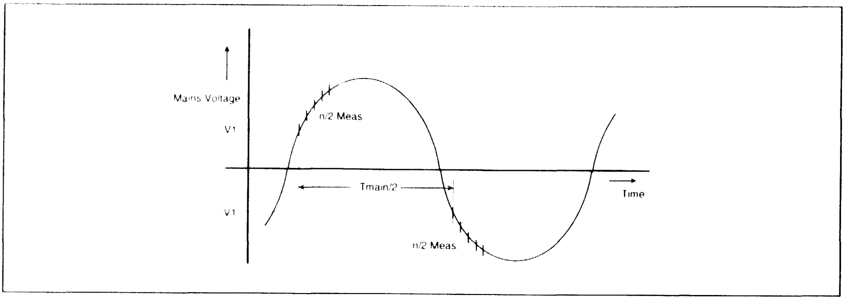


Figure 5.8.1: Reduction of Hum by Synchronizing to the Power Frequency. Single Measurement

If the Basic Timer is used for the timing then the following numbers of Basic Timer interrupts can be used:

Power Frequency f_{main}	Basic Timer Frequency f_{BT}	Number of BT Interrupts k	Time Error e_t max.	Residual Error e_r max.
50 Hz	4096 Hz	41	0.097%	0.61%
60 Hz	2048 Hz	17	-0.39%	-2.45%

The formulas to get the above errors are:

$$e_t = \left(\frac{T_{BT}}{T_{MAIN}} \times 2k - 1 \right) \times 100$$

$$e_r = \sin\left(\frac{T_{BT}}{T_{MAIN}} \times 2k \times 2\pi\right) \times 100$$

with:

e_t	Maximum time error due to fixed Basic Timer frequency in per cent
e_r	Maximum remaining influence of the hum in per cent compared to a measurement without hum cancellation
T_{BT}	Period of Basic Timer frequency ($1/f_{BT}$)
T_{MAIN}	Period of mains ($1/f_{MAIN}$)
k	Number of Basic Timer interrupts to reach $T_{MAIN}/2$ resp. T_{MAIN}

If difference measurements are used, the two measurements to be subtracted should be made with a delay of exactly one mains period: both measurements have the same influence from the hum and the result, the difference of both measurements, does not show the error. This measurement method is used with heat meters, where the temperature difference of the water inlet and the water outlet is used for calculations.

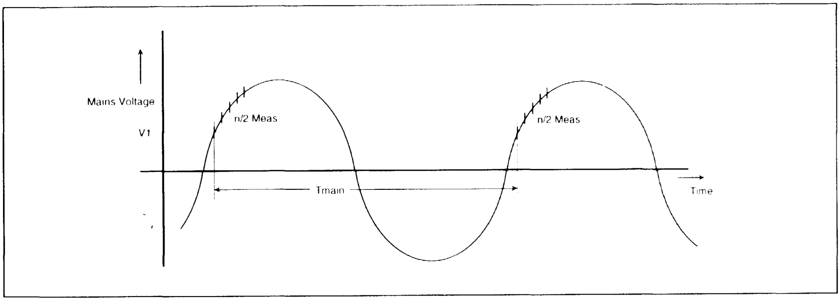


Figure 5.8.2: Reduction of Hum by Synchronizing to the Power Frequency. Differential Measurement

If the Basic Timer is used for the timing then the following numbers of Basic Timer interrupts can be used:

Power Frequency f_{main}	**Basic Timer Frequency f_{BT}**	**Number of Interrupts k**	**Time Error e_t max.**	**Residual Error e_r max.**
50 Hz	2048 Hz	41	0.097%	0.61%
60 Hz	1024 Hz	17	-0.39%	-2.45%

The formulas to get the above results are:

$$e_t = \left(\frac{T_{BT}}{T_{MAIN}} \times k - 1\right) \times 100$$

$$e_r = \sin\left(\frac{T_{BT}}{T_{MAIN}} \times k \times 2\pi\right) \times 100$$

The software needed for the modification of the Basic Timer frequency without the loss of the exact time base is shown in chapter "Change of Basic Timer Frequency"

5.4 Basic Timer Usage

The Basic Timer is normally used as a time base: it is programmed to interrupt the background program at regular time intervals. The following table shows all possible Basic Timer interrupt frequencies in dependence of the control word bits. The values are shown for MCLK = 1.048 MHz:

			SSEL = 0		**SSEL = 1**	
IP2	**IP1**	**IP0**	**DIV = 0**	**DIV = 1**	**DIV = 0**	**DIV = 1**
0	0	0	1634 Hz	64 Hz	(524288 Hz)	64 Hz
0	0	1	8192 Hz	32 Hz	(262144 Hz)	32 Hz
0	1	0	4096 Hz	16 Hz	(131072 Hz)	16 Hz
0	1	1	2048 Hz	8 Hz	(65536 Hz)	8 Hz
1	0	0	1024 Hz	4 Hz	32768 Hz	4 Hz
1	0	1	512 Hz	2 Hz	16348 Hz	2 Hz
1	1	0	256 Hz	1 Hz	8192 Hz	1 Hz
1	1	1	128 Hz	0.5 Hz	4096 Hz	0.5 Hz

The interrupt frequencies in brackets cannot be used by interrupt routines: the frequencies are too high.

```
;
; DEFINITION PART FOR THE BASIC TIMER
;
BTDAT    .EQU     041h     ; BT DATA REGISTER (0.5s)
BTCTL    .EQU     040h     ; BASIC TIMER CONTROL BYTE:
SSEL     .EQU     080h     ; 0: ACLK         1: MCLK
RESET    .EQU     040h     ; 0: RUN 1: RESET BT
DIV      .EQU     020h     ; 0: fBT1=fBT     1: fBT1=128 Hz
FRFQ     .EQU     008h     ; LCD FREQUENCY DIVIDER
IP       .EQU     001h     ; BT FREQUENCY Selection bits
;
ME2      .EQU     005h     ; MODULE ENABLE BYTE 2:
BTME     .EQU     080h     ; BT MODULE ENABLE BIT
;
IE2      .EQU     001h     ; INTERRUPT ENABLE BYTE 2:
BTIE     .EQU     080h     ; BT INTERRUPT ENABLE BIT
;
         .BSS     TIMER,4  ; 0.5s COUNTER
         .BSS     BTDTOL,1 ; LAST READ BT VALUE
;
; INITIALIZATION FOR 1 SECOND TIMING: 32768:(256x128)=1
;
; Input frequency ACLK:             SSEL = 0
; Input division by 256:            DIV = 1
; Add. input division by 128:       IP = 6
; LCD frequency = 128 Hz:          FRFQ = 3
;
; Initialization part
;
HLD         .EQU      040h              ; 1: Disable BT
;
            MOV.B     #(DIV+(6*IP)+(3*FRFQ)),&BTCTL      ; 1s
            BIS.B     #BTIE,&IE2        ; ENABLE INTRPT BASIC TIMER
            ...
;
; INTERRUPT HANDLER BASIC TIMER
; The register BTDAT needs to be read twice
;
BTHAN       PUSH      R5                ; SAVE USED REGISTER
L$300       MOV.B     &BTDAT,R5         ; READ ACTUAL TIMER VALUE
            CMP.B     &BTDAT,R5         ; ENSURE DATA INTEGRITY
            JNE       L$300             ; READ AGAIN IF NOT EQUAL
;
; R5 CONTAINS ACTUAL TIMER VALUE, BTDTOL CONTAINS LAST VALUE
; READ. THE DIFFERENCE IS ADDED TO THE 1S COUNTER
;
            PUSH.B    BTDTOL            ; SAVE LAST TIMER VALUE
            MOV.B     R5,BTDTOL         ; ACTUAL VALUE -> LAST VALUE
            SUB.B     @SP+,R5           ; ACTUAL - LAST VALUE -> R5
            ADD       R5,TIMER          ; 16-bit DIFFERENCE TO COUNTER
            ADC       TIMER+2           ; Carry to high word
            POP       R5                ; Restore R5
            RETI
;
            .SECT     "Int_Vect",0FFE2h
            .WORD     BTHAN             ; Basic Timer Interrupt Vector
```

5.4.1 Change of Basic Timer Frequency

If the Basic Timer is used as a time base (for example as a base for a clock) then it is necessary to do something if the frequency is changed during the normal run. The necessary operations are different for changing from a faster frequency to a slower one than for the reverse operation. The timer register where the interrupts are counted needs to be implemented for the highest used Basic Timer frequency.

Slow to fast change: The change should be done only inside the Basic Timer interrupt routine. The status is to be changed to the new time value.

Fast to slow change: The change should only be done inside the Basic Timer interrupt routine. Afterwards all bits of the software timer register which represent the higher Basic Timer frequencies should be reset to zero. This is the correct time for the lower frequency.

EXAMPLE: A Basic Timer interrupt handler is shown that works with two frequencies, 1 Hz and 8 Hz. All necessary status routines are shown. The handler may be used for all other possible frequency combinations

```
HIF         .EQU      8                 ; Hi frequency is 8 Hz
LOF         .EQU      1                 ; Lo frequency is 1 Hz
LOBIT       .EQU      HIF/LOF           ; Bit position of low frequency
            .BSS      TIMER,2           ; 16-bit timer register
```

```
         .BSS     BTSTAT,1            ; Status byte
;
BT_INT   PUSH     R5                  ; Save R5
         MOV.B    BTSTAT,R5           ; R5 contains status (0, 2, 4, 6)
         BR       BTTAB(R5)           ; Got to appropr. routine
BTTAB    .WORD    BT1HZ               ; ST0: 1 Hz interrupt
         .WORD    BT8HZ               ; ST2: 8 Hz interrupt
         .WORD    CHGT8               ; ST4: Change to 8 Hz interrupt
         .WORD    CHGT1               ; ST6: Change to 1 Hz interrupt
;
BT1HZ    ADD      #LOBIT,TIMER        ; Incr. bit 3 of the 125 ms timer
         POP      R5
         RETI                         ; No change of status
;
BT8HZ    INC      TIMER               ; Incr. bit 0 of the 125 ms timer
         POP      R5
         RETI                         ; No change of status
;
CHGT8    MOV.B    #2,BTSTAT           ; Change to 8 Hz interrupt
         POP      R5                  ; New status: 8 Hz interrupt
         RETI
;
CHGT1    BIC      #LOBIT-1,TIMER      ; Set 8 Hz bits to zero
         MOV.B    #0,BTSTAT           ; New status: 1 Hz interrupt
         POP      R5
         RETI
;
         .SECT    "Int_Vect",0FFE2h
         .WORD    BT_INT              ; Basic Timer Interrupt Vector
```

5.4.2 Elimination of the Quartz Crystal Tolerance

For normal measurement purposes the accuracy of 32768 Hz quartz crystals is more than sufficient. But if highly accurate timing has to be maintained for years, then it is necessary to know the frequency deviation of the quartz crystal used (together with the oscillator) from the exact frequency. An example for such an application is an electricity meter which has to switch the tariff at given times each day without any possibility of synchronizing the internal timer.

The time deviations for two quartz crystal accuracies (±1 Hz and ±10 ppm) are shown in the table below. It shows how long it takes to have a certain time error:

Accuracy	Deviation ±1 s	Deviation ±1 m	Deviation ±1 h
32768 Hz ± 1 Hz	9.10 hours	22.75 days	3.74 years
32768 Hz ± 10 ppm	27.77 hours	69.44 days	11.40 years

If these time deviations are not tolerable then a calibration and correction are necessary:

1. The quartz crystal frequency is measured and the deviation stored in the RAM or EEPROM. All other interrupts have to be disabled during this measurement to get correct results.
2. The measured time deviation of the quartz crystal is used for a correction that takes place at regular time intervals.

The quartz crystal frequency can be measured during the calibration with a timing signal of exactly 10 or 16 seconds at one of the ports with interrupt capability. The MSP430 counts its internal oscillator frequency ACLK during this time with one of the timers (8-bit timer or 16-bit timer) and gets the deviation to 32768 Hz. The deviation measured is added at appropriate time intervals (32768s x10 or 32768s x 16) to the timer register which counts the seconds.

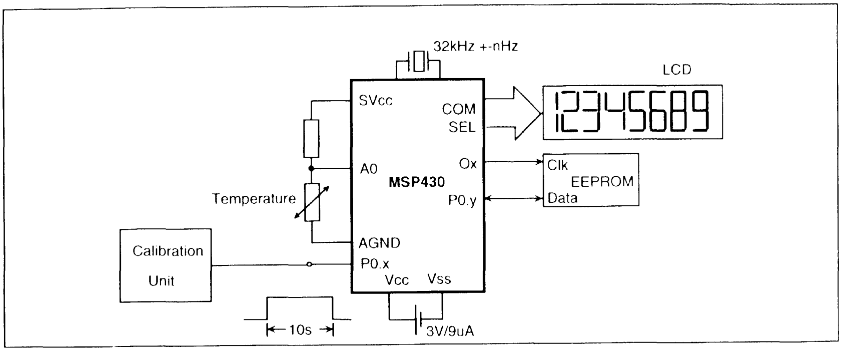


Figure 5.9: Calibration of the Quartz Crystal

If necessary the temperature behaviour of the quartz crystal can also be taken into account. The next figure shows the typical dependence of a quartz crystal in relation to its temperature. The nominal frequency is present at one temperature T_0 (turning point); above and below this temperature the frequency is always lower (negative temperature coefficient). Beside the turning point the frequency deviation increases with the square of the temperature deviation (-0.035 ppm/°C^2 for the example).

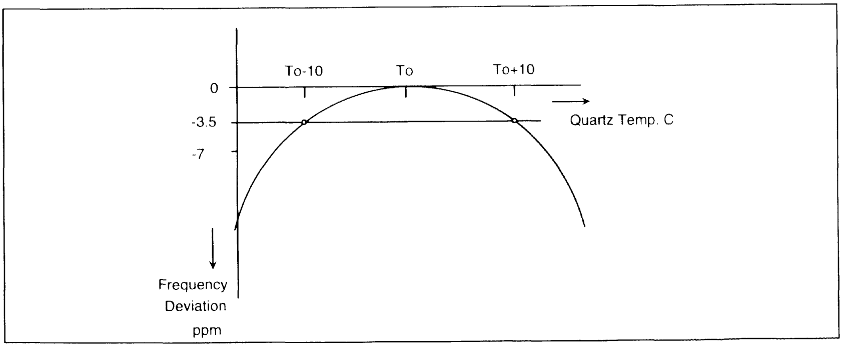


Figure 5.10: Quartz Crystal Frequency Deviation with Temperature

The quadratic equation that describes this temperature behavior is approximately (T_0 = +19°C):

$$\Delta f = -0.035 \times (T - 19)^2$$

with: Δf Frequency deviation in ppm
T Quartz crystal temperature in °C

To use the above equation simply every hour the quartz crystal temperature (PC board temperature) is measured and the frequency deviation computed. These deviations are added-up until an accumulated deviation of one second is reached: the counter for seconds is then incremented by one and one second is subtracted from the accumulated deviation, leaving the remainder in the accumulation register.

EXAMPLE: Quadratic quartz crystal deviation correction. The quartz crystal's temperature is measured each hour (3600s) and computed. The result in ppm/1024 is added-up in RAM location PPMS. If PPMS reaches 1024 one second is added to the seconds counter SECONDS and PPMS is reduced by 1024. The numbers at the right margin show the digits before and after the assumed decimal point.

```
;
; Quadratic temperature compensation after each hour:
; tcorr = -|(T-19)^2 x -0.035 ppm| x t
; Tmax = To+40C, Tmin = To-40C
;
To      .SET    19          ; Turning point of temperature
PPM     .SET    35          ; -0.035 ppm/(T-To)^2
        .BSS    PPMS,2      ; RAM word for adding-up deviation
        .BSS    SECOND,2 ;  RAM word for seconds counting
;
TIMCORR CALL    #MEASTEMP          ;Measure quartz temperature 6.4h
        POP     IROP2L             ; Result to IROP2L          6.4h
        SUB     #(To*10h),IROP2L   ; T - To 6.4h
        MOV     IROP2L,IROP1       ; Copy result
        CALL    #MPYS              ; |T-To|^2  (always pos.) 12.8
        CALL    #SHFTRS6           ; Adapt |T-To|^2   12.2
        ADC     IRACL              ; Rounding
        MOV     IRACL,IROP2L       ; |T-To|^2 -> IROP2L       12.2
;
; tcorr = 3600 x -0.035 x 1E-6 x (T-19)^2  s/h
;
L$006   MOV     #(36*PPM),IROP1    ; 36 x PPM/1E4      ms/h
        CALL    #MPYS              ; Signed multiplication
;
; IRAC contains: 36s x PPM x 4 (To-T)^2 x 1E-7  s/h
; = 36s x PPM x 4 (To-T)^2 x 1E-4  ms/h
ms/h
;
        CALL    #SHFTLS6 ; to IRACM
;
; IRACM contains: tcorr = 4 x dT x 36 x PPM/1024
; Correction: 0.25 x 1E-7 x 1024 = 1/39062.5
;
        ADD     IRACM,PPMS         ; Add-up deviation
        CMP     #39062,PPMS        ; One second deviation reached?
```

```
          JLO       L$200
          INC       SECONDS                   ; Yes, add one second
          SUB       #39062,PPMS               ; and adjust deviation counter
L$200     RETS
;
```

5.4.3 Clock Subroutines

The following two subroutines provide 24-hour clocks: one using decimal counting (RTCLKD) and one using hexadecimal counting (RTCLK). These subroutines are called every second from the Basic Timer handler. They may be enlarged to include the date easily.

```
;
SEC       .EQU      0200H     ; Byte for counting of seconds
MIN       .EQU      0201H     ; Byte for counting of minutes
HOURS     .EQU      0202H     ; Byte for counting of hours
;
; Subroutine provides a decimal clock: 00.00.00 to 23.59.59
;
RTCLKD    SETC                          ; Entry every second
          DADC.B    SEC                 ; Increment seconds
          CMP.B     #060H,SEC           ; One minute elapsed?
          JLO       RTRETD              ; No, return (C = 0)
          CLR.B     SEC                 ; Yes, clear seconds (C = 1)
          DADC.B    MIN                 ; Increment minutes with set carry
          CMP.B     #060H,MIN           ;
          JLO       RTRETD
          CLR.B     MIN
          DADC.B    HOURS
          CMP.B     #024H,HOURS
          JLO       RTRETD
          CLR.B     HOURS               ; 00.00.00
RTRETD    RET                           ; Return to caller
;
; Subroutine provides a hex clock: 00.00.00 to 17.3B.3B
;
RTCLK     INC.B     SEC                 ; Entry point every second
          CMP.B     #60,SEC             ; Increment seconds
          JLO       RTRET               ; One minute elapsed?
          CLR.B     SEC                 ; No, return to caller
          INC.B     MIN                 ; Yes, clear seconds
          CMP.B     #60,MIN             ; Increment minutes
          JLO       RTRET
          CLR.B     MIN
          INC.B     HOURS
          CMP.B     #24,HOURS
          JLO       RTRET
          CLR.B     HOURS               ; 00.00.00
RTRET     RET
;
```

5.5 General Purpose Subroutines

Following tested software examples are shown which may be of help during software development. The examples may not fit into any application, but they can be modified to the user's needs.

5.5.1 Initialization

For the first power-on it is necessary to clear the internal RAM to get a defined basis. If the MSP430 is battery powered and contains calibration factors or other important data in its RAM, it is necessary to distinguish between Cold Start and Warm Start. The reason is the possibility of initializations caused by electromagnetic interference (EMI). If such an erroneous initialization is not checked for legality, EMI influence could destroy the RAM content by clearing the RAM with the initialization software routine. Testing can be made by comparing RAM bytes with known content to their nominal value. These RAM bytes could be identification codes or extra written test patterns (e.g. A5h, F0h). If the tested RAM locations contain the right pattern, a spurious signal causes the initialization and the normal program can continue. If the tested RAM bytes differ from the nominal value, the RAM content is destroyed (e.g. by loss of power) and the initialization routine is invoked: the RAM is cleared and the peripherals are initialized.
The Cold Start software contains the waiting loop for the DCO which is needed to set it to the correct frequency. See chapter 1.4 "Use of the System Clock Generator".

```
; Initialization part: Check if Cold Start or Warm Start:
; RAM location 0200h decides kind of initialization:
; Cold Start: content differs from 0A5F0h
; Warm Start: content is 0A5F0h
;
INIT     CMP      #0A5F0h,&0200h     ; Test content of &200h
         JEQ      EMIINI             ; Correct content: No reset
;
; Control RAM content differs from 0A5F0: RAM needs to be
; cleared, peripherals needs to be initialized
;
         CALL     #RAMCLR            ; Clear complete RAM
         MOV      #0A5F0h,&0200h     ; Insert test word
;
; Waiting loop for the DCO of the FLL to settle: 130 ms
;
         CLR      R5        ; 2 x 65536 us = 131 ms
L$1      INC      R5
         JNZ      L$1
         ...
;
; EMI caused initialization: Periphery needs to be initialized:
; Interrupts need to be enabled again
;
EMINI    ...
```

5.5.2 RAM clearing Routine

```
; Definitions for the RAM block (depend 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.B   RAMSTRT(R4)           ; 1st RAM address
         INC     R4                    ; Next address
         CMP     #RAMEND-RAMSTRT+1,R4      ; RAM cleared?
         JLO     RCL                   ; No, once more
         RET                           ; Yes, return
;
```

5.5.3 Binary to BCD Conversion

The conversion of binary to BCD and vice versa is normally a time consuming task: 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 R12 IS CONVERTED TO A 5-DIGIT BCD
; NUMBER CONTAINED IN R14 AND R13: R14|R13
;
BINDEC   MOV     #16,R15               ; LOOP COUNTER
         CLR     R14                   ; 0 -> RESULT MSD
         CLR     R13                   ; 0 -> RESULT LSD
L$1      RLA     R12                   ; Binary MSB to carry
         DADD    R13,R13               ; RESULT x2 LSD
         DADD    R14,R14               ;           MSD
         DEC     R15                   ; THROUGH?
         JNZ     L$1
         RET                           ; YES, RESULT IN R14|R13
```

The above subroutine may be enlarged to any length of the binary part simply by adding of registers for the storage of the BCD number (a binary number with n bits needs approx. 1.2 x n bits for BCD format).
If numbers containing fractions have to be converted to BCD the following algorithm may be used:

1. Multiply the binary number as often with 5 as there are fractional bits. For example if the number looks like MMM.NN, then multiply it with 25. Ensure that no overflow will take place.
2. Convert the result of step 1 to BCD with the (eventually enlarged) subroutine BINDEC. The BCD result is a number with the same number of fractional digits as the binary number has fractional bits.

EXAMPLE: The binary number 0A8Bh has the format MMM.NNN. The decimal value is therefore 337.375. The steps to get the BCD number are:

1. 0A8Bh is to be multiplied by 5^3 or 125 due to its 3 fractional bits.
   0A8Bh x 125 =0525DFh
2. 0525DFh has the decimal equivalent 337375: the correct number with 3 fractional digits

To convert the above example the basic subroutine BINDEC needs to be enlarged: two binary registers are necessary to hold the input number.

```
; THE BINARY NUMBER IN R12|R11 IS CONVERTED TO AN 8-DIGIT BCD
; NUMBER CONTAINED IN R14 AND R13: R14|R13
; Max. hex number in R12|R11: 05F5E0FFh (99999999)
;
BINDEC  MOV     #32,R15         ; LOOP COUNTER
        CLR     R14             ; 0 -> RESULT MSD
        CLR     R13             ; 0 -> RESULT LSD
L$1     RLA     R11             ; MSB of LSBs to carry
        RLC     R12             ; Binary MSB to carry
        DADD    R13,R13         ; RESULT x2 LSD
        DADD    R14,R14         ;           MSD
        DEC     R15             ; THROUGH?
        JNZ     L$1
        RET                     ; YES, RESULT IN R14|R13
```

5.5.4 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 contained in R4 is converted to binary
; number contained in R5
;
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
```

5.5.5 Keyboard Scan

A lot of possibilities exist for the scanning of a keyboard, which also includes jumpers and digital input signals. If more input signals exist than free inputs, then scanning is necessary. The scanning outputs can be: I/O-ports and unused select outputs On. The scanning input can be I/O-ports and analog inputs An switched to digital inputs. If I/O-

ports are used for inputs then wake-up by input changes is possible: the select line(s) of the interesting inputs (keys, gates etc.) are set high and the interrupt(s) are enabled for the interesting signal edges. If one of the interesting input signal changes occurs, interrupt is given and wake-up takes place.
The figure below shows a keyboard with 16 keys.

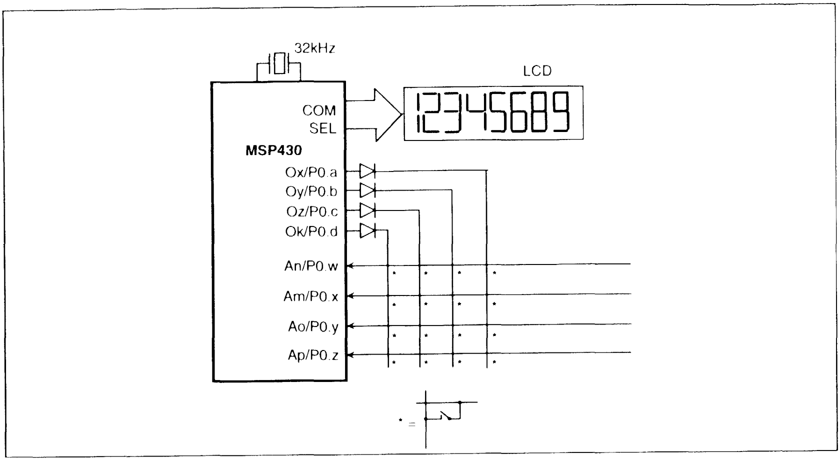


Figure 5.11: Keyboard Connection

The following figure shows some possibilities for connecting external signals to the MSP430:

- The first row contains keys. The decoupling diode in the row selection line prevents that pressed keys shortening other signals. If more than one key can be activated simultaneously then any key needs to have a decoupling diode.
- The second row contains diodes. This is a simple way to tell a system which version is used.
- The third row selects digital signals coming from peripherals with outputs that can be switched to HI-Z mode.
- The fourth row uses an analog switch to connect digital signals to the MSP430. Shown is the output of a CMOS gate and the output of a comparator.

The rows containing keys need to be debounced: if a change is seen at these inputs, the information is read in and stored. A second read is made after 10 to 100 ms, and the information read then compared to the first one. If both reads are equal the information is used; otherwise, the procedure is repeated. The Basic Timer can be used for this purpose.

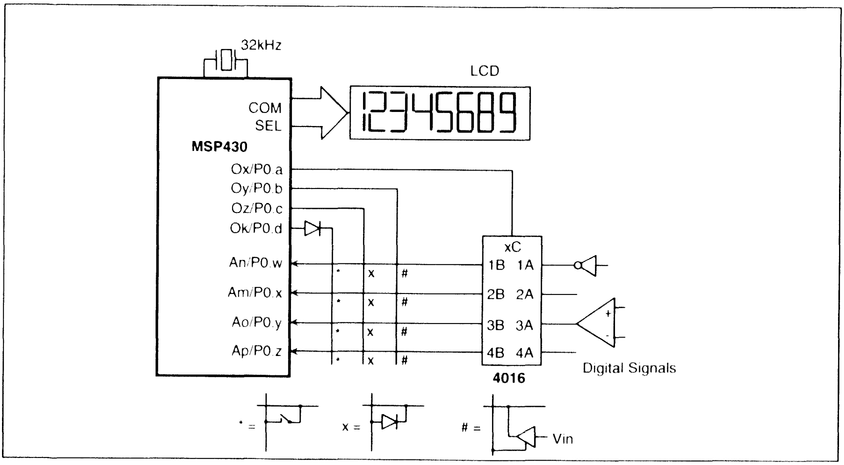


Figure 5.12: Connection of different Input Signals

5.5.6 Temperature Calculations for Sensors

Several sensors can be connected to the MSP430. The section concerning the ADC describes the different possible ways of doing this. Independently of the ADC or sensor type used, a binary number n is finally delivered from the ADC that represents the measured value K:

$$K = f(n)$$

with: K Measured value (temperature, pressure etc.)
n Result of ADC

The function f(n) is not normally linear for sensors, and therefore a calculation is needed to get the measured value K. The linearization of sensors by resistors is described in Application Report "SENSOR COMPONENT".

Two methods are described of how to represent the function f(n):

1. Table processing
2. Algorithms (linear, quadratic, cubic or hyperbolic equations)

5.5.6.1 Table Processing for Sensor Calculations

The ADC measurement range used is divided into parts, each of them having a length of 2^M bits. For any multiple of 2^M the output value K is calculated and stored in a one-dimensional table.

This table is used for linear interpolation to get the values for ADC results between two table values. The next figure shows such a non-linear sensor characteristic.

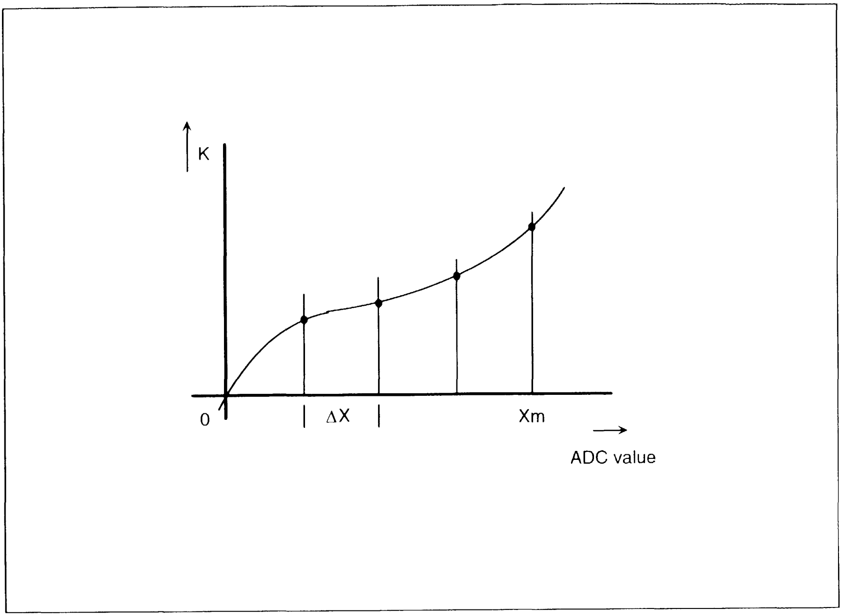


Figure 5.13: Nonlinear Function

Steps for the development of a sensor table:

1. Definition of the external circuitry used at the ADC input (See section "The Analog-to Digital Converters")
2. Definition of the output format of the table contents (bits after decimal point, M.N)
3. Calculation of the voltage at the analog input Ax for equally spaced (Δn) ADC values n
4. Calculation of the sensor resistances for the above calculated analog input voltages
5. Calculation of the input values K (temperature, pressure etc.) that cause these sensor resistances
6. Insertion of the calculated input values K in the format defined with 2. into the table

EXAMPLE: A sensor characteristic is described in a table TABLE. The ADC results are divided in distances Δn = 128 starting at value n0 = 256. The output value K is content of this table. The ADC result is corrected with offset and slope coming from the calibration procedure.

```
;
        .BSS    OFFSET,2        ; Offset from calibration 10.0
        .BSS    SLOPE,2         ; Slope from calibration   1.10
DN      .EQU    128             ; Delta N
;
```

```
; Table contains signed values. The decimal point may be anywhere
;
TABLE     .WORD    02345h, ...,00F3h; N0, N1, ...NM
;
TABCAL1  MOV      &ADAT,IROP1         ; ADC result to IROP1      14.0
         ADD      OFFSET,IROP1        ; Correct offset           10.0
         MOV      SLOPE,IROP2L        ; Slope                     1.10
         CALL     #MPYS               ; (ADC+OFFSET)xSLOPE       15.10
;
; Corrected ADC value in IRACM|IRACL.
;
         CALL     #SHFTLS6            ; Result to IRACM   15.0
         MOV      IRACM,XIN           ; Copy it
;
; Calculation of NA address. One less adaptation due to
; word table (2 bytes/item).
;
         MOV      XIN,IROP1           ; N -> Multiplicand        15.0
         SWPB     IROP1               ; Adapt to deltaN = 128    14.0
         BIC.B    #1,IROP1 ; Even word address needed             8.0
         SUB      #2,IROP1 ; Adapt to N0 = 256 (2 x deltaN)
         MOV      TABLE(IROP1),R15 ; NA from table
         MOV      TABLE+2(IROP1),R14        ; NA+1 from table
;
; K = XIN-NA/(deltaN) x (NA+1 - NA) + NA
;
         SUB      R15,XIN             ; XIN - NA
         MOV      R14,IROP2L          ; NA+1
         SUB      R15,IROP2L          ; NA+1 - NA)
         MOV      XIN,IROP1           ; XIN - NA
         CALL     #MPYS               ; (XIN - NA) x (NA+1 - NA)
         CALL     #SHFTRS6 ; /deltaN
         CALL     #SHFTRS1 ; deltaN = 2^7
         ADD      R15,IRACL           ; + NA, result in IRACL
         RET
```

5.5.6.2 Algorithms for Sensor Calculations

If the function K = f(n) can be described by an algorithm of the form Linear Equation

$$K = b \times n + a$$

or Quadratic Equation

$$K = c \times n^2 + b \times n + a$$

or Cubic Equation

$$K = d \times n^3 + c \times n^2 + b \times n + a$$

or Root Equation

$$K = a \pm \sqrt{b + c \times n}$$

or Hyperbolic Equation

$$K = \frac{c}{b+n} + a$$

then no table is necessary: the output value K can be calculated out of the ADC result n. The coefficients a, b, c, d can be found with PC computer software (e.g. MATHCAD) or with formulas by hand.

Steps for the development of a sensor algorithm:

1. Definition of the hardware circuitry used at the ADC input (See section „The Analog-to- Digital Converters“ for the different possibilities)
2. Definition of the output format of the algorithm (bits after decimal point: M.N)
3. Definition of an input value K to be measured (temperature, pressure etc.)
4. Calculation of the nominal sensor resistance for the above chosen input value
5. Calculation of the voltage at the analog input Ax for this sensor resistance (See section „The Analog-to-Digital Converters“ for the formulas used with the different circuits)
6. Calculation of the ADC result n for this input voltage at Ax
7. Repetition of steps 3 to 6 depending on the algorithm used: twice for linear equations, three times for quadratic equations, four times for cubic, hyperbolic and root equations.
8. Decision of the sensor characteristic
9. Calculation of the coefficients a, b, c and d out of the calculated pairs of input value K and ADC result n

EXAMPLE: A quadratic behaviour is given for a sensor characteristic:

$$K = c \times n^2 + b \times n + a$$

with n representing the ADC result. The corrected ADC result (see above) is stored in XIN; the three terms are stored in ROM locations A, B and C.

```
;
C        .WORD    07FE3h          ; Quadratic term    +-0.14
B        .WORD    00346h          ; Linear term                    +-0.14
A        .WORD    01234h          ; Constant term                  +-15.0
;
QUADR    MOV      XIN,IROP1       ; Corrected ADC result             14.0
         MOV      C,IROP2L        ; Factor c                       +-0.14
         CALL     #MPY            ; XIN x C                         14.14
         ADD      B,IRACL         ; (XIN x C) + B                  +-0.14
         ADC      IRACM           ; Carry to HI reg
         CALL     #SHFTL3         ; To IRACM                         14.1
         MOV      IRACM,IROP2L    ; (XIN x C) + B -> IROP2L   14.1
         CALL     #MPYS           ; (XIN x C) + B) x XIN             28.1
         CALL     #SHFTL2         ; Result to IRACM     15.0
         ADD      A,IRACM         ; Add a                            15.0
;
; The signed 16-bit result is located in IRACM.
;
         RET
```

The HORNER-scheme used above can be expanded to any level; it is only necessary to shift the multiplication results to the right to ensure that the numbers always fit into the 32-bit result buffer. The terms A, B, and C may also be located in RAM.
If lots of calculations need to be done then the use of the floating point package should be considered. See chapter 5.6 for details.

5.5.7 Battery Check

Due to the ratiometric measurement principle of the ADC, the measured digital value of a constant voltage is an indication of the supply voltage of the MSP430. The measured value is inversly proportional to the supply voltage V_{CC}. To get the reference for later battery tests a measurement is made with $V_{CC} = V_{CCmin}$. 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 $V_{CC}=V_{CCmin}$ determines the status of the battery. If the measured value exceeds the stored one, then $V_{CC}<V_{CCmin}$ and a Battery Low indication can be given by software.
Figure 3.14 shows the connecting of the voltage reference.

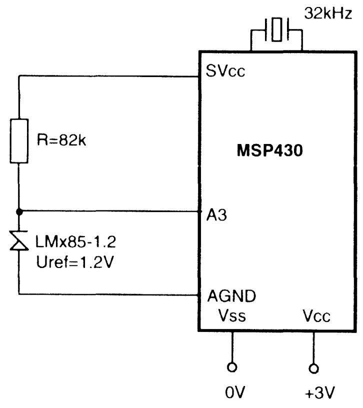


Figure 3.14: Connection of a Voltage Reference

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

$$N = INT\left|\frac{V_{IN} \times 2^{14}}{SV_{CC}}\right|$$

The reference voltage can be calculated as follows:

$SV_{CC} = SV_{CCmin} = 2.8$ Volt
VIN = 1.235 ± 0,012 Volt

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

To ensure that the voltage of the battery is above SV_{CCmin}, 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 given.

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

```
ASOC     .set    1       ;bit position for Conv. 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 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

;*******************************************************************
;Channel A3 is measured with the polling method for one time
;The result will be contained 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
```

5.5.8 Data Security Applications

If consumption data is transmitted via telephone lines or sent by RF then it is normally necessary to encrypt this data to make it completely unreadable. For these purposes the DES (Data Encryption Standard) is used more and more, and is becoming the standard in Europe too. The next two sections show how to implement the algorithms of this standard and how the encrypted data can be sent by the MSP430.

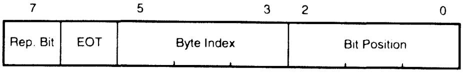
5.5.8.1 Data Encryption Standard (DES) Routines

The DES works on blocks of 64 bits: these blocks are modified in several steps and the output is also a block with totally scrambled 64 bits. It is not the intention of this section to show the complete DES algorithm; instead, a subroutine is shown that is able to do all of the necessary permutations in a very short time. The subroutine mentioned can do the following permutations (the tables mentioned refer to the booklet "Data Encryption Algorithm" of the ANSI):

1. Initial Permutation: 64 bits plain text to 64 bit encrypted text via table IP
2. 32 bit to 48 bit permutation via table E
3. 48 bit to 32 bit permutation via tables S1 to S8
4. 32 bit to 32 bit permutation via table P
5. Inverse initial Permutation: 64 bits to 64 bit via table IP^{-1}

The permutation subroutine is written in a code and time optimized manner to get the highest data throughput with the lowest ROM space requirements.

For each kind of permutation a description table is necessary that contains the following information for every bit to be permuted:



with: Rep. Bit Repetition Bit: The actual bit is contained twice in the

	output table. The next byte (with Rep. = 0) contains the address for the second insertion. This bit is only used during the 32-bit to 48-bit permutation.
EOT	End of Table Bit: This bit is set in the last byte of a permutation table
Byte Index	The byte address 0 to 7 inside the output block
Bit Position	The bit address 0 to 7 inside the output byte

The following figure shows the permutation of bit i. The description table contains at address i the information:

Repetition Bit = 0:	The bit i is to be inserted into the output table only once
EOT = 0	Bit i is not the last bit in the description table
Byte Index = 3:	The relative byte address inside the output table is 3 (PTOUT+3)
Bit Position = 5:	The bit position inside the output byte is 5 (020h)

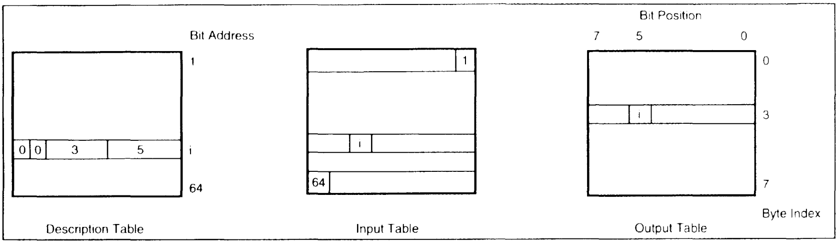


Figure 5.15: DES Encryption Subroutine

NOTE

The bit numbers used in the DES specification range from 1 to 64. The MSP430 subroutines use addresses from 0 to 63 due to the computer architecture.

The software subroutines for the above described permutations follow. The subroutines PERMUT and PERM_BIT are used for all necessary permutations (see above). The subroutines shown have the following needs:

1. The initialization of the subroutine PERMUT decides which permutation takes place. The address of the actual description table is written to pointer register PTPOI.
2. Permutations are always made from table PTIN (input table) to table PTOUT (output table).
3. Only "Ones" are processed during the permutation. This saves 50% of processing time. The output buffer is therefore cleared initially by the PERMUT subroutine.
4. The output buffer must start with an even address (word instructions are used for clearing)

```
; Main loop for a permutation run. Tables with up to 64 bits are
; permuted to other tables.
```

```
;
; Definitions for the permutation software
;
PTPOI     .EQU     R6              ; Pointer to description table
PTBYTP    .EQU     R7              ; Byte index input table
PTBITC    .EQU     R8              ; Bit counter inside input byte
          .BSS     PTIN,8          ; Input table 64 bits
          .BSS     PTOUT,8         ; Output table 64 bits
EOT       .EQU     040h            ; End of table indication bit
REP       .EQU     080h            ; Repetition bit
;
; Call for the "Initial Permutation". Description table is
; starting at label IP (64 bytes for 64 bits).
;         ...
          MOV      #IP,PTPOI       ; Load description table pointer
          CALL     #PERMUT         ; Process Initial Permutation
;         ...
;
; Permutation subroutine. Table PTIN is permuted to table PTOUT
;
PERMUT    CLR      PTBYTP          ; Clear byte index input table
          CLR      PTOUT           ; Clear output table 8 bytes
          CLR      PTOUT+2
          CLR      PTOUT+4
          CLR      PTOUT+6
PERML     CLR      PTBITC          ; Bit counter (bits inside byte)
L$502     RRA.B    PTIN(PTBYTP)    ; Next input bit to Carry
          JNC      L$500           ; If bit = 0: No activity nec.
L$501     CALL     #PERM_BIT       ; Bit = 1: Insert bit to output
L$500     INC      PTPOI           ; Incr. description table pointer
          TST.B    -1(PTPOI)       ; REP bit set for last bit?
          JN       L$501           ; Yes, process 2nd output bit
;
; One input table bit is processed. Check if byte limit reached
;
          INC.B    PTBITC          ; Incr. bit counter
          CMP.B    #8,PTBITC       ; Bit 8 (outside byte) reached?
          JLO      L$502
          INC.B    PTBYTP          ; Yes, address next byte
          BIT.B    #EOT,-1(PTPOI)  ; End of desc. table reached?
          JZ       PERML           ; No, proceed with next byte
          RET

; Permutation subroutine for one bit: A set bit of the input is
; set in the output depending on the information of a
; description table pointed too by pointer PTPOI
; 20 cycles + CALL (5 cycles)
;
PERM_BIT .EQU      $
          MOV.B    @PTPOI,R4       ; Fetch description word
          MOV      R4,R5           ; Copy it
          BIC.B    #REP+EOT,R4     ; Clear Repetition bit and EOT
          RRA.B    R4              ; Move Index Bits to LSBs
          RRA.B    R4              ; to form byte index to PTBIT
          RRA.B    R4
          AND.B    #07h,R5         ; Mask out index for output table
          BIS.B    PTBIT(R5),PTOUT(R4)     ; Set bit in output table
          RET
;
PTBIT     .BYTE    1,2,4,8,10h,20h,40h,80h         ; Bit table
;
; Description Table for the Initial Permutation. 64 bits of
; the input table are permuted to 64 bits in the output table
; (IP-1 table contains these numbers)
;
IP        .BYTE    40-1               ; Bit 1 -> position 40
          .BYTE    8-1                ; Bit 2 -> position 8
;         ...
          .BYTE    EOT+25-1           ; Bit 64 -> pos. 25, End of table
;
; Description Table for the Expansion Function E. 32 bits of
; the input table are permuted to 48 bits in the output table
;
E         .BYTE    REP+2-1            ; Bit 1 -> position 2 and 48
          .BYTE    48-1               ; Bit 1 -> position 48
          .BYTE    3-1                ; Bit 2 -> pos. 3
;         ...
          .BYTE    REP+1-1            ; Bit 32 -> position 1 and 47
          .BYTE    EOT+47-1           ; Bit 32 -> pos. 47, End of table
;
```

Processing time for a 64-bit block: The most time consuming parts for the encryption are the permutations. All other operations are simple moves or exclusive OR's (XOR). This means that the number of permutations multiplied with the number of cycles per bit gives an estimation of the needed processing time. Every bit needs 43 cycles to be permuted.
The necessary number of permutations is:

1.	Initial Permutation:	64
2.	32 bit to 48 bit permutation	16 x 48
3.	48 bit to 32 bit permutation	16 x 32
4.	32 bit to 32 bit permutation	16 x 32
5.	Inverse initial Permutation:	64
6.	Key permutations choice 1	56
7.	Key permutations choice 2	16 x 48

	Sum of permutations	2744

Number of cycles

typically (2744 x 43 x 0.5)	58996 cycles	32 ones in block
maximum (2744 x 43)	117992 cycles	64 ones in block

For a block with 64 bits approximately 59 ms are needed with an MCLK of 1 MHz.

ROM space: The needed ROM space can be divided into the following parts:

1.	Main program (approx.)	400 bytes
2.	Subroutines	100 bytes
3.	Tables for permutations	570 bytes

	Sum of bytes	1070 bytes

The complete DES encryption software fits into 1K of bytes.

5.5.8.2 Output Sequence for 19.2 kHz Bi-Phase Space Code

The encrypted information is output normally with a Bi-Phase Code: Figure 5.16 shows such a modulation. At the beginning of a bit a level change occurs. A zero bit "0" has an additional level change in the middle of the bit, a one bit "1" has the same information during the whole bit.

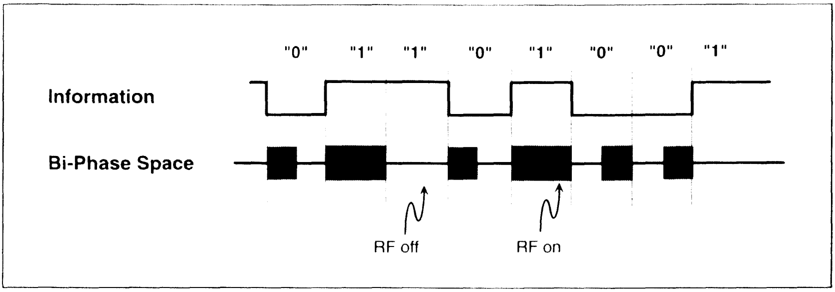


Figure 5.16: Bi-Phase Space Code

The output sequence is written for P0.4 (as shown in Figure 4.12 Heat Allocation Meter). This means that no constant of the Constant Generator may be used. If P0.0, P0.1,
P0.2 or P0.3 are used, the instructions which address the ports are one cycle shorter and the delay subroutines have to be adapted.

```
; OUT192 OUTPUTS THE RAM STARTING AT "RAMSTART" BITWISE
; IN BI-PHASE-CODE. EVERY 040h ADDRESSES A SCAN IS MADE
; TO READ P0.1 WHERE THE WATER FLOW COUNTER IS LOCATED. THE
; 4 SCAN RESULTS ARE ON THE STACK AFTER RETURN FOR CHECKS
; NOPs ARE INCLUDED TO ENSURE EQUAL LENGTH OF EACH BRANCH.
; All interrupts must be disabled during this output subroutine!
; CALL #NOPx MEANS x CYCLES OF DELAY
;
OUTPUT   .EQU    010h              ; P0.4:
PORT     .EQU    011h              ; PORT0
RAMSTART .EQU    0200h             ; Start of output info
RAMEND   .EQU    0300h             ; End of output info
SCAND    .EQU    040h              ; Scan delta (addresses)
Rw       .EQU    R15
Rx       .EQU    R14
Ry       .EQU    R13
Rz       .EQU    R12
;
OUT192   BIC.B   #OUTPUT,&PORT     ; Reset output port
         MOV     #RAMSTART,Ry      ; WORD POINTER
         MOV     #RAMSTART+SCAND,Rw         ; NEXT SCAN ADDRESS

; FETCH NEXT WORD AND OUTPUT IT                         CYCLES

WORDLP   MOV     #16,Rz            ; BIT COUNTER             2
         MOV     @Ry,Rx            ; FETCH WORD              5

; OUTPUT NEXT BIT: Change output state

BITLOP   XOR.B   #OUTPUT,&PORT     ; CHANGE OUTPUT PORT      5
```

```
;
; CHECK IF NEXT SCAN OF WATER FLOW IS NECESSARY: Ry >= Rw
;
        CMP     Rw,Ry           ;                           1
        JHS     SCAN            ; YES                       2
        NOP                     ; NO                        5
        NOP
        NOP
        NOP
        NOP
        JMP     BITT            ;                           2
SCAN    ADD     #SCAND,Rw       ; NEXT SCAN ADDRESS         2
        PUSH    &PORT           ; PUSH INFO OF PORT         5
;
BITT    RRC     Rx              ; NEXT BIT TO CARRY         1
        JNC     OUT0            ; BIT = 0                   2
;
; BIT = 1: OUTPUT PORT IS CHANGED IN THE MIDDLE OF BIT
;
        CALL    #NOP9           ;                           9
        XOR.B   #OUTPUT,&PORT   ; CHANGE OUTPUT PORT        5
        JMP     CHECK           ;                           2
;
; BIT = 0: OUTPUT PORT STAYS DURING COMPLETE BIT
;
OUT0    CALL    #NOP16          ; OUTPUT STAYS HI16
;
; END OF LOOP: CHECK IF COMPLETE WORD OR END OF INFO
;
CHECK   DEC     Rz              ; 16 BITS OUTPUT?1
        JZ      L$1             ; YES                       2
        CALL    #NOP15          ; NO, NEXT BIT              15
        JMP     BITLOP          ;                           2
;
; COMPLETE WORD OUTPUT: ADDRESS NEXT WORD

L$1     ADD     #2,Ry           ; POINTER TO NEXT WORD      2
        CMP     #RAMEND,Ry      ; RAM OUTPUT?               2
        JEQ     COMPLET ; YES   ;                           2
        NOP                     ; NO, NEXT WORD             2
        NOP
        JMP     WORDLP          ;                           2
;
COMPLET ....                    ; 4 SCANS ON STACK

; NOP Subroutine: The Subroutine inserts defined numbers of
; cycles when called. The number xx of the called label defines
; the number of cycles including CALL (5 cycles) and RETS
;
NOP16   NOP             ; CALL #NOPxx needs 5 cycles
NOP15   NOP
NOP14   NOP
NOP13   NOP
NOP12   NOP
NOP11   NOP
NOP10   NOP
NOP9    NOP
NOP8    RET             ; RET needs 3 cycles
;
```

5.6 Floating Point Package

Floating point arithmetic is necessary if the range of the used numbers is very large. When using a floating point package it is normally not necessary to take care if the limits of the number range are exceeded. This is due to a number ratio of about 10^{78} if comparing the largest to the smallest possible number (remember: the number of smallest particles in the whole universe is estimated to 10^{84}). The disadvantages are the slower calculation speed and the ROM space needed.

A Floating Point Package with 24-bit and 40-bit mantissa exists for the MSP430. The number range, resolution and error indication are explained as well as the conversion subroutines used as the interface to binary and binary-coded-decimal (BCD) numbers. Examples are given for a lot of subroutines and applications like the square root are included in a software example chapter.

5.6.1 General

This Floating Point Package (FPP) consists of 3 files supporting the .FLOAT format (32 bits) and the .DOUBLE format (48 bits):

- FPP03.ASM: the Basic Arithmetic Operations add, subtract, multiply, divide and compare
- CNV03.ASM: the conversions from and to the binary and the BCD format
- FPPDEF.ASM: the definitions used with the other two files

NOTE

> The file FPP03.ASM may be used without the conversions, but the conversion subroutines need the FPP03.ASM file. This is due to the common completion parts contained in FPP03.ASM.

The assembly time variable DOUBLE defines which format is to be used:

DOUBLE = 0:	Two word format .FLOAT with 24-bit mantissa
DOUBLE = 1:	Three word format .DOUBLE with 40-bit mantissa

The assembly time variable SW_UFLOW defines the reaction after a software underflow:

SW_UFLOW = 0:	Software underflow (result is zero) is not treated as an error
SW_UFLOW = 1:	Software underflow is treated as an error (N is set)

The FPP supports the four basic arithmetic operations, comparison, conversion subroutines and two register save/restore functions:

FLT_ADD	Addition
FLT_SUB	Subtraction
FLT_MUL	Multiplication
FLT_DIV	Division
FLT_CMP	Comparison
FLT_SAV	Saving of all used registers on the stack

FLT_REC	Restoring of all used registers from the stack
CNV_BINxxx	Binary to floating point conversion
CNV_BCD_FP	BCD to floating point conversion
CNV_FP_BIN	Floating point to binary conversion
CNV_FP_BCD	Floating point to BCD conversion

5.6.2 Common Conventions

The use of registers containing the addresses of the arguments saves time and memory space. The arguments are not affected by the operations and can be located either in ROM or in RAM. Before the call for an operation the two pointers RPARG and RPRES are loaded with the address(es) of the most significant word MSW of the argument(s). After the return from the call both pointers and also the stackpointer SP point to the result (on the stack) for an easy continuation of arithmetical expressions.

NOTE

The result of a floating point operation is always written to the address the stack pointer SP pointed to when the subroutine was called. The address contained in register RPRES is used only for the addressing of Argument 1.

The registers which hold the pointers are called:

RPRES	Pointer to Argument 1 and Result
RPARG	Pointer to Argument 2 and Result

$$1.\ \mathrm{RESULT}_{\mathrm{NEW}} = @(\mathrm{RPRES})\ <\mathrm{operator}>\ @(\mathrm{RPARG})$$

$$2.\ \mathrm{RESULT}_{\mathrm{NEW}} = @(\mathrm{RPRES})\ <\mathrm{operator}>\ \mathrm{RESULT}_{\mathrm{OLD}}$$

$$3.\ \mathrm{RESULT}_{\mathrm{NEW}} = \mathrm{RESULT}_{\mathrm{OLD}}\ <\mathrm{operator}>\ @(\mathrm{RPARG})$$

To 1. RPRES and RPARG both point to the arguments for the next operation. This is the common form that is always valid independent where the two pointers point to (new arguments or result). The result of the operation is written to the address the stack pointer SP points to.

To 2. RPRES points to the argument 1, RPARG still points to the result of the last operation residing on the top of the stack (TOS). This calling form allows the operations (argument 2 - result) and (argument 2 / result).

To 3. RPARG points to the argument 2, RPRES still points to the result of the last operation residing on the top of the stack. This calling form allows the operations (result - argument 2) and (result / argument 2).

NOTE

The formulas 2 and 3 are not equal, they allow to use the result on the TOS in two ways with the division and the subtraction. No time and ROM-consuming moves are necessary if the result is the divisor or the subtrahend.

Common to these subroutines is:

1. The pointers RPARG and RPRES point to the addresses of the input numbers. They always point to the MSBs of these numbers.
2. The input numbers are not modified, except the last result on the stack was used as an operand.
3. The result is located on the top of the stack (TOS), the stack pointer SP, RPARG and RPRES point to the most significant word of the result
4. Every floating point number represents a valid value. No invalid combinations like "Not a Number", "De normalized Number" or "Infinity" do exist. This way the MSP430 FPP has a larger range than other FPPs have and allows a higher speed with smallest memory usage.
5. Every floating point operation outputs a valid floating point number that can be used immediately by the other operations.
6. If a result is too large (exceeds the number range) then the signed, maximum number is output. An error indication is given in this case.

5.6.3 The Basic Arithmetic Operations

The FPP is designed for fast and memory saving computations. So register instructions are the ideal fit for this target. A common save and recall routine for the registers used at the beginning and the end of an arithmetical expression is an additional optimization. The subroutines FLT_SAV and FLT_REC should be applied as shown in the examples below.

5.6.3.1 Addition

FLT_ADD The floating point number pointed to by the register RPARG is added to the floating point number pointed to by the register RPRES. The 25th bit (41st bit in case of DOUBLE format) of the calculated mantissa is used for rounding: it is added to the result.

$$\text{RESULT on TOS} = @(\text{RPRES}) + @(\text{RPARG})$$

Errors: Normal error handling. See chapter Error Handling for a detailed description.

Output: The floating point sum of the two arguments is placed on the top of the stack. The stack pointer SP points to the same location as it did before the subroutine call.

The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point sum. If an error occurred (N = 1 after return) then the result is the number that represents the correct result best: 0 resp. $\pm 3.4 \times 10^{38}$.

EXAMPLE: The floating point number (.FLOAT format) contained in the ROM locations starting at address NUMBER is added to the RAM locations pointed to by R4. The result is written to the RAM addresses RES, and RES+2 (LSBs).

```
DOUBLE      .EQU     0
            MOV      R4,RPRES          ; Address of Argument 1 in R4
            MOV      #NUMBER,RPARG     ; Address of Argument 2
            CALL     #FLT_ADD          ; Call add subroutine
            JN       ERR_HND           ; Error occurred, check reason
            MOV      @RPRES+,RES       ; Store FPP result (MSBs)
            MOV      @RPRES+,RES+2     ; LSBs
            ...                        ; Continue with program
```

5.6.3.2 Subtraction

FLT_SUB The floating point number pointed to by the register RPARG is subtracted from the floating point number pointed to by the register RPRES. By proper loading of the two input pointers it is possible to calculate (Argument1 - Argument2) and (Argumnet2 - Argument1). The 25th bit (41st bit in case of DOUBLE format) of the calculated mantissa is used for rounding: it is subtracted from the result.

RESULT on TOS = @(RPRES) - @(RPARG)

Errors: Normal error handling. See chapter Error Handling for a detailed description.

Output: The floating point difference of the two arguments is placed on the top of the stack. The stack pointer SP points to the same location as it did before the subroutine call.
The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point difference. If an error occurred (N = 1 after return) then the result is the number that represents the correct result best: 0 resp. ±3.4 x 10^{38}.

EXAMPLE: The floating point number (.DOUBLE format) contained in the ROM locations starting at address NUMBER is subtracted from the RAM locations pointed to by R4. The result is written to the RAM addresses pointed to by R4.

```
DOUBLE      .EQU     1
            MOV      R4,RPRES          ; Address of Argument1 in R4
            MOV      #NUMBER,RPARG     ; Address of Argument2
            CALL     #FLT_SUB          ; ((R4)) - (NUMBER) -> TOS
            JN       ERR_HND           ; Error occurred, check reason
            MOV      @RPRES+,0(R4)     ; Store FPP result (MSBs)
            MOV      @RPRES+,2(R4)
            MOV      @RPRES,4(R4)      ; LSBs
            ...                        ; Continue with program
```

5.6.3.3 Multiplication

FLT_MUL The floating point number pointed to by the register RPARG is multiplied by the floating point number pointed to by the register RPRES. The 25th and 26th bit (41st and 42nd bit in case of DOUBLE format) of the calculated mantissa are used for rounding:
If a shift is necessary to get the MSB of the mantissa set then the LSB-1 is shifted into the mantissa and the LSB-2 is added to the result.
If the mantissa is yet one then only the LSB-1 is added to the result.

$$\text{RESULT on TOS} = @(\text{RPRES}) \times @(\text{RPARG})$$

Errors: Normal error handling. See chapter Error Handling for a detailed description.

Output: The floating point product of the two arguments is placed on the top of the stack. The stack pointer SP points to the same location as it did before the subroutine call.
The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point product. If an error occurred (N = 1 after return) then the result is the number that represents the correct result best: 0 resp. $\pm 3.4 \times 10^{38}$.

Special Cases: **0 x 0 = 0 0 x X = 0 X x 0 = 0**

EXAMPLE: The result of the last operation, a floating point number (.FLOAT format) on the top of the stack, is multiplied by the constant π.

```
DOUBLE     .EQU     0
           MOV      #PI,RPARG        ; Address of constant PI
           CALL     #FLT_MUL         ; (RPRES) x (PI) -> TOS
           JN       ERR_HND          ; Error occurred, check reason
           ...                       ; Continue with program
PI         .FLOAT   3.1415926535     ; Constant PI
```

5.6.3.4 Division

FLT_DIV The floating point number pointed to by the register RPRES is divided by the floating point number pointed to by the register RPARG. By proper loading of the two input pointers it is possible to calculate (Argument1 / Argument2) and (Argument2 / Argument1). The 25th bit (41st bit in case of DOUBLE format) of the calculated mantissa is used for rounding: it is added to the result.

$$\text{RESULT on TOS} = \frac{@(\text{RPRES})}{@(\text{RPARG})}$$

Errors: Normal error handling. See chapter Error Handling for a detailed description. Division by zero is indicated too.

Output: The floating point quotient of the two arguments is placed on the top of the stack. The stack pointer SP points to the same location as it did before the subroutine call.
The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point quotient. If an error occurred (N = 1 after return) then the result is the number that represents the correct result best for example the largest number that can be represented if a division by zero was made.

Special Cases: **0/0 = 0 0/X = 0 -X/0 = max. neg. number**
+X/0 = max. pos. number

EXAMPLE: The floating point number (.DOUBLE format) contained in the ROM locations starting at address NUMBER is divided by the RAM locations pointed to by R4. The result is written to the RAM addresses pointed to by R4.

```
DOUBLE    .EQU    1
          MOV     R4,RPARG          ; Address of dividend
          MOV     #NUMBER,RPRES     ; Address of divisor
          CALL    #FLT_DIV          ; (NUMBER) / ((R4)) -> TOS
          JN      ERR_HND           ; Error occurred, check reason
          MOV     @RPRES+,0(R4)     ; Store FPP result (MSBs)
          MOV     @RPRES+,2(R4)
          MOV     @RPRES,4(R4)      ; LSBs
          ...                       ; Continue with program
```

Examples for the Basic Arithmetic Operations

The example below shows the following program steps for the .FLOAT format:

1. The used registers R5 to R12 are saved on the stack.
2. Four bytes are allocated on the stack to hold the results of the operations.
3. The pointer to a 12-digit BCD-buffer is loaded into pointer RPARG and the BCD-to-floating point conversion is called. The resulting floating point number is written to the result space allocated before.
4. The resulting floating point number is multiplied with a number residing in the memory address VAL3. RPARG points to this address.
5. To the last result a floating point number contained in the memory address VAL4 is added
6. The final result is converted back to BCD format (6 bytes) that can be displayed nearly directly in the LCD.
7. The final result is copied to the RAM addresses BCDMSD, BCDMID and BCDLSB. The three necessary POP instructions correct the stack pointer SP to the value after the "Save Register" subroutine.
8. The used registers R5 to R12 are restored from the stack. The system environment is exactly the same now as before the floating point calculations.

```
;
DOUBLE    .EQU    0                 ; Use .FLOAT format
;
          ......                    ; Normal program
          CALL    #FLT_SAV          ; Save registers R5 to R12
          SUB     #4,SP             ; Allocate stack for result
```

```
                MOV       #BCDB,RPARG      ; Load address of BCD-buffer
                CALL      #CNV_BCD_FP      ; Convert BCD number to FP
;
; Calculate (BCD-number x VAL3) + VAL4
;
                MOV       #VAL3,RPARG      ; Load address of slope
                CALL      #FLT_MUL         ; Calculate next result
                MOV       #VAL4,RPARG      ; Load address of offset
                CALL      #FLT_ADD         ; Calculate next result
                CALL      #CNV_FP_BCD      ; Convert final FP result to BCD
                JN        CNVERR           ; Result too big for BCD buffer
                POP       BCDMSD           ; BCD number MSDs and sign
                POP       BCDMID           ; BCD digits MSD-4 to LSD+4
                POP       BCDLSD           ; BCD digits LSD+3 to LSD
                                           ; Stack is corrected by POPs
                CALL      #FLT_REC         ; Restore registers R5 to R12
                                           ; Continue with program
VAL3            .FLOAT    -1.2345          ; Slope
VAL4            .FLOAT    14.4567          ; Offset
CNVERR          ...                        ; Start error handler
```

The next example shows the following program steps for the .DOUBLE format:

1. The used registers R5 to R15 are saved on the stack.
2. Six bytes are allocated on the stack to hold the results of the operations.
3. The ADC buffer address of the MSP430C32x (14 bit result) is written to RPARG and the last ADC result converted into a floating point number. The resulting floating point number is written to the result space allocated before.
4. The resulting floating point number is multiplied with a number located at the memory address VAL3. RPARG points to this address.
5. To the last result a floating point number contained in the memory address VAL4 is added.
6. The final result is converted back to binary format (6 bytes) that can be used for integer calculations.
7. The resulting binary number is copied to the RAM addresses BINMSD, BINMID and BINLSB. The three necessary POP instructions correct the stack pointer SP to the value after the „Save Register“ subroutine.
8. The used registers R5 to R15 are restored from the stack. The system environment is now exactly the same as it was before the floating point calculations.

```
;
DOUBLE          .EQU      1                ; Use .DOUBLE format
;
                ......                     ; Normal program
                CALL      #FLT_SAV         ; Save registers R5 to R15
                SUB       #6,SP            ; Allocate stack for result
                MOV       #ADAT,RPARG      ; Load address of ADC data buffer
                CALL      #CNV_BIN16U      ; Convert unsigned result to FP
;
; Calculate (ADC-Result x VAL3) + VAL4
;
                MOV       #VAL3,RPARG      ; Load address of slope
                CALL      #FLT_MUL         ; Calculate next result
                MOV       #VAL4,RPARG      ; Load address of offset
                CALL      #FLT_ADD         ; Calculate next result
                CALL      #CNV_FP_BIN      ; Convert final FP result to binary
                POP       BINMSD           ; Store MSBs of result and sign
```

```
          POP       BINMID          ;
          POP       BINLSD          ; Stack is corrected by POPs
          CALL      #FLT_REC        ; restore registers R5 to R15
                                    ; Continue with program
VAL3      .DOUBLE   1.2E-3          ; Slope 0.0012
VAL4      .DOUBLE   1.44567E1       ; Offset 14.4567
```

5.6.3.5 Error Handling

Errors during the operation affect the status bits in the status register SR: if the N-bit contained in the Status Register SR is set to zero, no error occurred. If the N-bit is set to one, an error occurred. The kind of error can be seen in the Error Indication Table below. The columns .FLOAT and .DOUBLE show the returned results for each error.

Error Indication Table

Error	Status	.FLOAT	.DOUBLE
Overflow positive	N=1, C=1, Z=1	FF7F,FFFF	FF7F,FFFF,FFFF
Overflow negative	N=1, C=1, Z=0	FFFF,FFFF	FFFF,FFFF,FFFF
Underflow	N=1, C=0, Z=0	0000,0000	0000,0000,0000
Divide by zero	N=1, C=0, Z=1	FF7F,FFFF or FFFF,FFFF	FF7F,FFFF,FFFF or FFFF,FFFF,FFFF

Software underflow is only treated as an error if the variable SW_UFLOW is set to one during assembly.

5.6.3.6 Stack Allocation

Before calling an operation 4 (resp. 6) bytes on the stack have to be reserved for the result. The following return address of the operation occupies another 2 bytes. The subroutines need one subroutine level during the calculations for the common initialization subroutine.

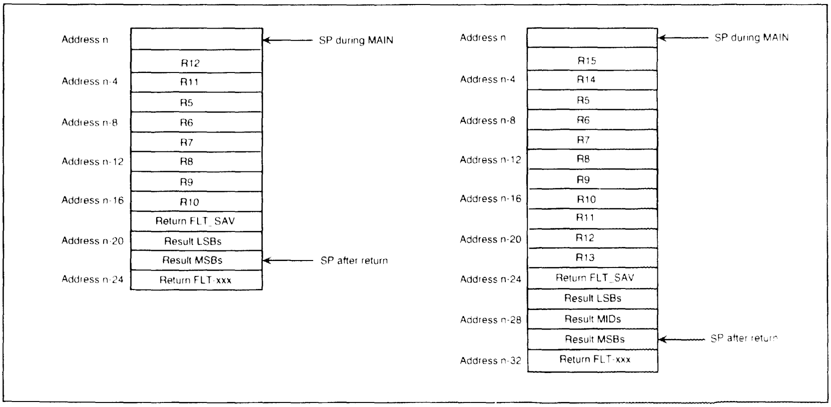


Figure 5.17: Stack Allocation for .FLOAT and .DOUBLE Formats

Note that it is strongly recommended to provide conscientious housekeeping for the stack pointer SP to avoid stack overflow.

5.6.3.7 Number Range and Resolution

E = exponent of the floating point number. See chapter 5.6.5 for explanation.

5.6.3.7.1 FLOAT Format

Most positive number	FF7F,FFFF	$2^{127} \times (2 - 2^{-23})$	$= 3.402823 \times 10^{38}$
Least positive number	0000,0001	$2^{-128} \times (1 + 2^{-23})$	$= 2.938736 \times 10^{-39}$
Zero	0000,0000	0	= 0.0
Least negative number	0080,0000	-2^{-128}	$= -2.938736 \times 10^{-39}$
Most negative number	7FFF,FFFF	$-2^{127} \times (2 - 2^{-23})$	$= -3.402823 \times 10^{38}$
Resolution		$2^{-23} \times 2^{E}$	$= 119.2093 \times 10^{-9}\ 2^{E}$

5.6.3.7.2 DOUBLE Format

Most positive number	FF7F,FFFF,FFFF	$2^{127} \times (2 - 2^{-39})$	$= 3.402824 \times 10^{38}$
Least positive number	0000,0000,0001	$2^{-128} \times (1 + 2^{-39})$	$= 2.938736 \times 10^{-39}$
Zero	0000,0000,0000	0	= 0.0
Least negative number	0080,0000,0000	-2^{-128}	$= -2.938736 \times 10^{-39}$
Most negative number	7FFF,FFFF,FFFF	$-2^{127} \times (2 - 2^{-39})$	$= -3.402824 \times 10^{38}$
Resolution		$2^{-39} \times 2^{E}$	$= 1.818989 \times 10^{-12} \times 2^{E}$

5.6.4 Calling Conventions for the Comparison

The Comparison subroutine works much faster than a floating subtraction: only the exponents and signs are compared in a first step to find out the relation of the two arguments. Only if exponents and signs are equal, than the mantissas are compared. After the comparison the status bits of the status register (SR) hold the result:

Comparison Results

Comparison	Status
Argument 1 > Argument 2	C=1, Z=0
Argument 1 < Argument 2	C=0, Z=0
Argument 1 = Argument 2	C=1, Z=1

The calling and the use of the returned status bits is shown in the next example:

```
                  ..........
                  MOV      #ARG1,RPRES      ; Point to Argument 1 MSBs
                  MOV      #ARG2,RPARG      ; Point to Argument 2 MSBs
                  CALL     #FLT_CMP         ; Comparison: result to SR
                  JEQ      EQUAL            ; Condition for program flow
                  JC       ARG1_GT_ARG2     ; @RPRES is greater than @RPARG
                  ......                    ; ARG1 is less than ARG2
EQUAL             ......                    ; ARG1 and ARG2 are equal
ARG1_GT_ARG2      ..                        ; ARG1 is greater than ARG2
```

5.6.5 Internal Data Representation

The description shows both the FLOAT and the DOUBLE formats. The two floating point formats consist of a floating point number whose

- 8 most significant bits represent the exponent
- and the 24 resp. 40 least significant bits hold the sign and the mantissa.

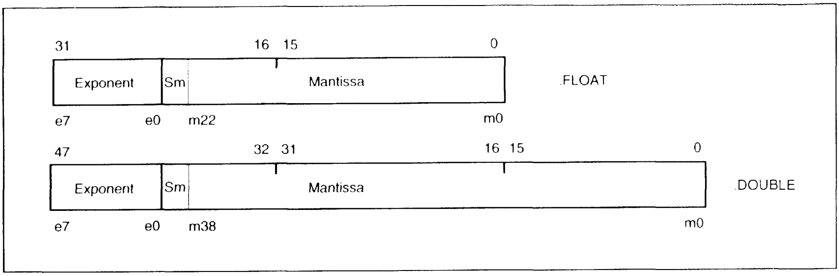


Figure 5.18: Floating Point Formats for the MSP430 FPP

with	Sm	Sign of floating point number (sign of mantissa)
	mx	Mantissa bit x
	ex	Exponent bit x
	x	Valence of bit

The value N of a floating point number is

$$N = (-1)^{Sm} \times M \times 2^{E}$$

NOTE

The only exception to the above equation is the floating zero: it is represented by all zeroes (32 resp. 48 zeroes). No negative zero exists, the corresponding number (0080,0000) is a valid non-zero number.

5.6.5.1 Computation of the Mantissa M

$$M = 1 + \sum_{i=0}^{22}(m_i \times 2^{i-23})$$ .Float Format

$$M = 1 + \sum_{i=0}^{38}(m_i \times 2^{i-39})$$ .Double Format

The result of the above calculation is always: $2 > M \geq 1$
For the MSB of the normalized mantissa is always 1 a most significant non-sign bit is implied providing an additional bit of precision. This bit is hidden and therefore called 'Hidden Bit'. The sign bit is located at this place instead:

Sm = 0:	positive Mantissa
Sm = 1:	negative Mantissa

NOTE

Note that a negative mantissa is NOT represented as a two's-complement number, only the sign Sm decides if the floating-point number is positive or negative.

5.6.5.2 Computation of the Exponent E

$$E = \sum_{i=0}^{7}(e_i \times 2^i) - 128$$

The MSB of the exponent indicates whether the exponent is positive or negative.

MSB of exponent = 0:	Exponent is negative
MSB of exponent = 1:	Exponent is positive

The reason for this convention is the representation of the number zero: this number is represented by all zeroes.

5.6.6 Execution Cycles

In the following evaluation the variables

```
X          .float  3.1416         ; resp. .double 3.1416
Y          .float  3.1416*100     ; resp. .double 3.1416*100
```

are the base for the calculations. The shown cycles include the addressing of the operands and the subroutine call itself:

```
MOV      #X,RPRES        ; Address 1st operand
MOV      #Y,RPARG        ; Address 2nd operand
CALL     #FLT_xxx        ; X <op> Y
....                     ; Result on TOS
```

The following table shows the necessary number of cycles needed for the above shown calculations:

Operation		.FLOAT	.DOUBLE
Addition	X + Y	185	207
Subtraction	X - Y	178	200
Multiplication	X x Y	399	691
Division	X / Y	407	754

5.6.7 Conversion Routines

5.6.7.1 General

To allow the conversion of integer numbers to floating point numbers and vice versa the following subroutines are provided (both for .FLOAT and .DOUBLE format):

CNV_BINxxx	Convert 16-bit, 32-bit or 40-bit signed and unsigned integer binary numbers to the floating point format
CNV_BCD_FP	Convert a signed 12-digit BCD number to the floating point format
CNV_FP_BIN	Convert a floating point number to a signed 5-byte integer (40 bits)
CNV_FP_BCD	Convert a floating point number to a signed 12-digit BCD number

Common to these subroutines is:

1. The pointer RPARG points to the address of the input number
2. The input number is not modified except it is the result of the previous operation on the TOS
3. The result is located on the top of the stack (TOS), the stack pointer SP, RPARG and RPRES point to the most significant word of the result
4. Only integers are converted. See chapter 5.6.7.3 for the handling of non-integer numbers
5. The result is calculated using truncation normally, except rounding is specified. The assembly time variable SW_RND defines which mode is to be used:

   SW_RND = 0: Truncation is used, the trailing bits are cut off
   SW_RND = 1: Rounding is used, the first unused bit is added to the number

   See chapter 5.6.7.4 for details.

6. The subroutines may be used for 2-word (.FLOAT format) and 3-word (.DOUBLE format) floating point numbers. The assembly time variable DOUBLE defines which mode is to be used:

   DOUBLE = 0:   Two word format .FLOAT
   DOUBLE = 1:   Three word format .DOUBLE

7. All conversion subroutines need two resp. three allocated words on the top of the stack. These words contain the result after the completed operation. A simple

```
   SUB  #4,SP            ; .FLOAT format allocation
   SUB  #6,SP            ; .DOUBLE format allocation
or SUB  #(ML/8)+1,SP     ; For both formats
```

   instruction is used for this allocation. It is the same allocation that is necessary anyway for the Basic Arithmetic Operations.

8. The FPP03.ASM package is needed: the completion routines of this file are used too

5.6.7.2 Conversions

The possible conversions are described in detail in the following sections. Input and output formats, error handling and number range are given for each conversion.

5.6.7.2.1 Binary to Floating Point Conversions

Binary numbers, 16 bit, 32 bit and 40 bit in length, are converted to floating point numbers. The used subroutine call defines if the binary number is treated as a signed or an unsigned number. No errors are possible, the N-bit of the Status Register is always cleared on return. Six different conversion calls are provided:

CNV_BIN16 The 16-bit number, RPARG points to, is treated as a 16-bit signed number.
Range: -32768 to + 32767 (08000h to 07FFFh)

CNV_BIN16U The 16-bit number, RPARG points to, is treated as a 16-bit unsigned number.
Range: 0 to + 65535 (00000h to 0FFFFh)

CNV_BIN32 The 32-bit number, RPARG points to, is treated as a 32-bit signed number.
Range: -2^{31} to $+2^{31} - 1$ (08000,0000h to 07FFF,FFFFh)

CNV_BIN32U The 32-bit number, RPARG points to, is treated as a 32-bit unsigned number.
Range: 0 to $2^{32} - 1$ (00000,0000h to 0FFFF,FFFFh)

CNV_BIN40 The 48-bit number, RPARG points to, is treated as a 40-bit signed resp. unsigned number.

Range signed: -2^{40} +1 to $+2^{40}$ - 1
(0FF00,0000,0001h to 000FF,FFFF,FFFFh)
Range unsigned: 0 to $+2^{40}$ - 1
(00000,0000,0000h to 000FF,FFFF,FFFFh)

The above conversion subroutines convert the 16-bit, 32-bit or 48-bit numbers to a sign extended 48-bit number contained in the registers BIN_MSB, BIN_MID and BIN_LSB. Depending on the used call (signed or unsigned) the leading bits are sign extended or cleared. The resulting 48-bit number is converted afterwards. This allows an additional subroutine call:

CNV_BIN The 48-bit signed number contained in the registers BIN_MSB to BIN_LSB (3 words) is converted to a floating point number.
Range signed: -2^{40} +1 to $+2^{40}$ - 1
(0FF00,0000,0001h to 000FF,FFFF,FFFFh)
Range unsigned: 0 to $+2^{40}$ - 1
(00000,0000,0000h to 000FF,FFFF,FFFFh)

NOTE

Input values outside of the 40-bit range shown above do not generate error messages. The leading bits are truncated and only the trailing 40-bits are converted to the floating point format.

Errors: No error is possible, the N-bit of the Status Register is always cleared on return.

Output: The output depends on the chosen floating point format, selected with the assembly time variable DOUBLE.

.FLOAT The two-word floating point result is written to the top of the stack. The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point number.

.DOUBLE The three-word floating point result is written to the top of the stack. The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point number.

EXAMPLE: The 32-bit signed binary number contained in the RAM locations BINLO and BINHI (MSBs) is to be converted to a three word floating point number. The result is to be written to the RAM addresses RES, RES+2 and RES+4 (LSBs).

```
DOUBLE      .EQU     1
            MOV      #BINHI,RPARG      ; Address of binary MSBs
            CALL     #CNV_BIN32        ; Call conversion subroutine
            MOV      @RPRES+,RES       ; Store MSBs of result
            MOV      @RPRES+,RES+2     ;
            MOV      @RPRES,RES+4      ; Store LSBs of result
            ...
```

5.6.7.2.2 Binary Coded Decimal to Floating Point Conversion

Binary coded decimal numbers (BCD numbers), 12 digits in length, are converted to floating point numbers. The MSB of the MSD word contains the sign of the BCD number:

MSB = 0: positive BCD number
MSB = 1: negative BCD number

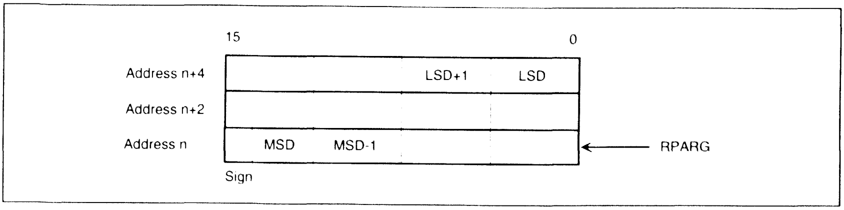


Figure 5.19: BCD Buffer Format

CNV_BCD_FP The 12-digit number (contained in 3 words), RPARG points to, is converted to a floating point number.

Range: $-8 \times 10^{11} + 1$ to $+8 \times 10^{11} - 1$

Errors: No error is possible, the N-bit of the Status Register is always cleared on return. If non-BCD numbers are contained in the BCD-buffer, the result will be erroneous. If the MSB of the input number is greater than 7, then the input number is treated as a negative number.

Output: A floating point number on the top of the stack

.FLOAT The two-word floating point result is written to the top of the stack. The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point number.

.DOUBLE The three-word floating point result is written to the top of the stack. The stack pointer SP, RPRES and RPARG point to the MSBs of the floating point number.

EXAMPLE: The signed BCD number contained in the RAM locations starting at label BCDHI (MSDs) is to be converted to a two word floating point number. The result is to be written to the RAM addresses RES, and RES+2 (LSBs).

```
DOUBLE    .EQU    0
          MOV     #BCDHI,RPARG     ; Address of BCD MSDs
          CALL    #CNV_BCD_FP      ; Call conversion subroutine
          MOV     @RPRES+,RES      ; Store FP result (MSBs)
          MOV     @RPRES,RES+2     ; LSBs
          ...                      ; Continue with program
```

5.6.7.2.3 Floating Point to Binary Conversion

The floating point number pointed to by the register RPARG is converted to a 40-bit signed binary number located on the top of the stack after conversion. See figure 5.20.

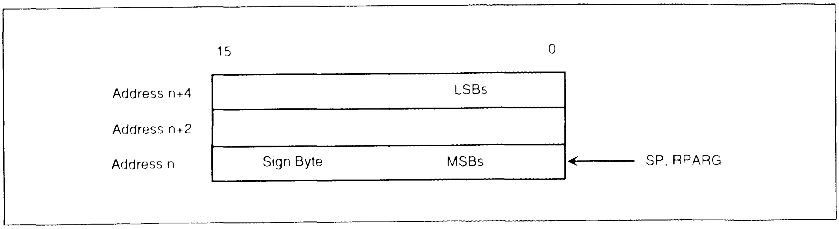


Figure 5.20: Binary Number Format

CNV_FP_BIN The floating point number, RPARG points to, is converted to a 40-bit signed binary number.
Range signed: $-2^{40}+1$ to $+2^{40}-1$
(0FF00,0000,0001h to 000FF,FFFF,FFFFh)

Errors: If the absolute value of the floating point number is greater than 2^{40}- 1, then the N bit in the status register is set to one. Otherwise the N bit is cleared.
The result on top of the stack is the largest signed binary number (saturation mode).

Output: A 40-bit signed, binary number at the top of the stack.

.FLOAT The stack pointer SP, RPRES and RPARG point to the MSBs of the three word binary result: an additional word is inserted. It is the responsibility of the calling software to correct the stack by one level upwards after the reading of the result.

.DOUBLE The stack pointer SP, RPRES and RPARG point to the MSBs of the three word binary result.

EXAMPLE: The floating point number (.DOUBLE format) contained in the RAM locations starting at label FPHI (MSBs) is converted to a 40-bit signed binary number. The result is written to the RAM addresses RES, and RES+2 and RES+4 (LSBs).

```
DOUBLE     .EQU    1
           MOV     #FPHI,RPARG      ; Address of FP MSBs
           CALL    #CNV_FP_BIN      ; Call conversion subroutine
           JN      ERR_HND          ; |FP number| is too big
           MOV     @RPRES+,RES      ; Store binary result (MSBs)
           MOV     @RPRES+,RES+2    ;
           MOV     @RPRES,RES+4     ; LSBs
           ...                      ; Continue with program
```

5.6.7.2.4 Floating Point to Binary Coded Decimal Conversion

The floating point number pointed to by the register RPARG is converted to a signed 12-digit BCD number located on the top of the stack after conversion. See figure 3. The MSD of the result has a maximum value of 7 due to the sign bit that uses the MSB position.

CNV_FP_BCD The floating point number, RPARG points to, is converted to a 12-digit signed BCD number.

Range: $-8 \times 10^{11} + 1$ to $+8 \times 10^{11} - 1$

Errors: Three errors at different stages of the conversion are possible that will set the N-bit in the status register:

1. The exponent value of the floating point number is greater than 39 which represents an absolute value greater than 1.0995×10^{12}
2. The absolute value of the floating point number is greater than $8 \times 10^{11} - 1$
3. The absolute value is greater than 1×10^{12}

Otherwise the N bit is cleared.

The result on top of the stack is the largest signed BCD number in case of an error.

Output: A 12-digit signed BCD number at the top of the stack.

.FLOAT The stack pointer SP, RPRES and RPARG point to the MSDs of the three word BCD result: an additional word is inserted. It is the responsibility of the calling software to correct the stack by one level upwards after the reading of the result.

.DOUBLE The stack pointer SP, RPRES and RPARG point to the MSDs of the three word BCD result.

EXAMPLE: The floating point number (.FLOAT format) contained in the RAM locations starting at label FPHI (MSBs) is to be converted to a 12-digit BCD number. The result is to be written to the RAM addresses RES, and RES+2 and RES+4 (LSDs).

```
DOUBLE     .EQU    0
           MOV     #FPHI,RPARG      ; Address of FP MSBs
           CALL    #CNV_FP_BCD      ; Call conversion subroutine
           JN      ERR_HND          ; |FP number| is too big
           MOV     @SP+,RES         ; Store BCD result (MSDs)
           MOV     @SP,RES+2        ; SP is corrected by last instr.
           MOV     2(SP),RES+4      ; LSDs
           ...                      ; Continue with program
ERR_HND    ...                      ; Correct error here
```

5.6.7.3 Handling of non-integer Numbers

The conversion subroutines allow only the handling of integer numbers when converting to or from floating point numbers. The reasons for this restriction are:

1. The stack grows if non-integer handling is included
2. The necessary program code of the conversion software grows strongly
3. The integration of non-integer numbers is easier outside of the conversion subroutines
4. The execution time grows strongly due to the necessary successive divisions by 10 or multiplications with 10.

5.6.7.3.1 Binary to Floating Point Conversion

If the location of the decimal point in the binary or hexadecimal number is known, then the correction of the result is as follows:
The resulting floating point number is divided by the constant 2^n for binary numbers resp. 16^m for hexadecimal numbers (with m = 0.25n). This is made simply by subtracting of n from the exponent of the floating point number. Overflow or underflow is not possible due to the restricted range of the binary input (-2^{40} +1 to $+2^{40}$ -1) compared to the range of the floating point numbers (-10^{32} to $+10^{32}$).

EXAMPLE: The binary 32-bit signed number contained in the RAM locations starting at label BINHI (MSBs) is converted to a floating point number (.DOUBLE format). The virtual decimal point of the binary input number is 5 bits left to the LSB (this means the integer input number is 32 times too large). For example: The binary buffer contains 1011000 (88_{10}) but the real number is 10.11000 (2.75_{10}; 88 / 32 = 2.75)

```
        MOV     #BINHI,RPARG    ; Address of binary buffer MSBs
        CALL    #CNV_BIN32      ; Call conversion subroutine
        SUB.B   #5,1(SP)        ; Correct result's exp. by 2^5
        ...                     ; Continue with corrected number
```

5.6.7.3.2 Binary Coded Decimal to Floating Point Conversion

If the location of the decimal point in the BCD number is known, then the correction of the result is as follows:
The resulting floating point number is divided by the constant 10^n after the conversion. Overflow or underflow is not possible due to the restricted range of the BCD input number (-8×10^{11} to $+8 \times 10^{11}$) compared to the range of the floating point numbers (-10^{32} to $+10^{32}$).

EXAMPLE: The BCD number contained in the RAM locations starting at label BCDHI (MSDs) is to be converted to a floating point number (.FLOAT format). The virtual decimal point of the BCD input number is 3 digits left to the LSD (this means the integer input number is 1000-times too large). For example: The BCD buffer, containing 123456 represents the number 123.456

```
DOUBLE      .EQU    0
            MOV     #BCDHI,RPARG    ; Address of BCD buffer MSDs
            CALL    #CNV_BCD_FP     ; Call conversion subroutine
            MOV     #FLT1000,RPARG  ; Address of constant 1000
            CALL    #FLT_DIV        ; Correct result by 1000
            ...                     ; Continue with corrected input
FLT1000     .FLOAT  1000            ; Correction constant 1000
```

If the location of the decimal point relative to the number's end is contained in a byte DPL (content > 0) the following code may be used:

```
DOUBLE      .EQU    1
            MOV     #BCDHI,RPARG    ; Address of BCD buffer MSDs
            CALL    #CNV_BCD_FP     ; Call conversion subroutine
LOOP        MOV     #DBL10,RPARG    ; Divide result by 10 as often -
            CALL    #FLT_DIV        ; as DPL defines
            DEC.B    DPL          ; DPL - 1
            JNZ      LOOP         ; Repeat as often as necessary
            ...                   ; Continue with corrected input
DBL10       .DOUBLE  10           ; Succ. correction constant 10
```

5.6.7.3.3 Floating Point to Binary Conversion

If the binary result should contain n binary digits after the decimal point then the following procedure may be used:
The floating point number is multiplied by the constant 2^n before the conversion call. This is made simply by adding of n to the exponent of the floating point number. Overflow may occur if the floating point number is very large and cannot be converted to binary anyway.

EXAMPLE: The floating point number contained in the RAM locations starting at label FPHI (MSBs) is to be converted to a binary number (.FLOAT format). Four fractional bits of the resulting binary number should be included in the result (this means the result needs to be 16-times larger). For example: The floating point number is 12.125, the resulting binary number is 11000010_2 ($C2_{16}$) not only 1100_2 (C_{16}) .

```
DOUBLE      .EQU     0
            MOV      FPHI,0(SP)     ; MSBs of FP number to TOS
            MOV      FPHI+2,2(SP)   ; LSBs to TOS+2
            ADD.B    #4,1(SP)       ; Correct exponent by 2^4
            MOV      SP,RPARG       ; Act. pointer (if not yet done)
            CALL     #CNV_FP_BIN    ; Call conversion subroutine
            ...                     ; Result includes 4 add. bits
```

If the floating point number to be converted may be modified then a simplified code may be used:

```
            MOV      #FPHI,RPARG    ; Address of FP number MSBs
            ADD.B    #4,1(RPARG)    ; Correct exponent by 2^4
            CALL     #CNV_FP_BIN    ; Call conversion subroutine
            ...                     ; Result includes 4 add. bits
```

5.6.7.3.4 Floating Point to Binary Coded Decimal Conversion

If the BCD result of this conversion should contain n digits after the decimal point then the following procedure may be used:
The floating point number is multiplied by the constant 10^n before the conversion call. Overflow may occur if the floating point number is very large and cannot be converted to BCD anyway due to the buffer length (12 digits max.).

EXAMPLE: The floating point number contained in the RAM locations starting at label FPHI (MSBs) is to be converted to a BCD number (.DOUBLE format). Two fractional digits should be included in the BCD result (this means the BCD result needs to be 100-times larger).
For example: The floating point number is 12.125_{10}, the resulting BCD number written to the TOS is 1212_{10} (SW_RND = 0) respective 1213_{10} (SW_RND = 1) not only 12_{10}.

```
DOUBLE      .EQU     1
            MOV      #FPHI,RPARG    ; Address of FP number (MSBs)
```

```
              MOV     #DBL100,RPRES   ; Address of constant 100
              CALL    #FLT_MUL        ; FP number x 100 -> TOS
              CALL    #CNV_FP_BIN     ; Call conversion subroutine
              ...                     ; Result includes 2 add. digits
DBL100        .DOUBLE 100             ; Constant 100
;
```

5.6.7.4 Rounding and Truncation

Two different modes for the conversions can be selected during the assembly of the conversion subroutines:

Truncation: Intermediate results of the conversion process are used as they are, independent of the status of the next lower bits. This is the case if SW_RND = 0 is selected during assembly.

Rounding: Intermediate results of the conversion process are rounded depending on the status of the 1st bit not included in the current result (LSB-1). If this bit is set (1) then the intermediate result is incremented, otherwise the result is not affected. If a carry occurs during the incrementing, then the exponent is corrected too. Rounding is used if SW_RND = 1 is selected during assembly.

Rounding is applied (if specified) at the following conversion steps:

Binary to Floating Point: .FLOAT: the MSB of the truncated word is added to the 24-bit mantissa
.DOUBLE: all 40 input bits are included, no rounding is possible

BCD to Floating Point: like with the binary to floating point conversion

Floating Point to Binary: the 2^{-1} bit (the bit representing 0.5) of the floating point number is added to the binary integer result

Floating Point to BCD: the 2^{-1} bit (the bit representing 0.5) of the floating point number is added to the binary integer that is converted to a BCD number.

If rounding is specified during assembly, then the ROM-code of the conversion subroutines is approximately 26 bytes larger than with truncation selected.

5.6.7.5 Execution Cycles

To give an impression how long conversions will take, the needed cycles for each conversion are given for the converted values 1 and the largest possible value (8 x 10^{11} -1 for BCD conversions and 2^{40} -1 for binary conversions). The cycle count is given for the .FLOAT and for the .DOUBLE format. Rounding is used.

The cycle count for each conversion includes the loading of the pointer RPARG, the subroutine call and the conversion itself.

Conversion	.FLOAT 1	.FLOAT max	.DOUBLE 1	.DOUBLE max
CNV_BIN40	418	67	422	71
CNV_BCD_FP	1223	890	1227	894
CNV_FP_BIN	535	67	531	63
CNV_FP_BCD	1174	706	1170	701

5.6.8 Memory Requirements for the complete Floating Point Package

The memory requirements of an implemented Floating Point Package are depending on the routines used and the precision applied. The following values refer to a completely implemented package. Truncation is used with the Conversion Routines. The given numbers indicate bytes.

Package	.FLOAT	.DOUBLE
Basic Arithmetic Operations	632	720
Conversion Subroutines	344	340
Complete FPP	976	1060

5.6.9 Inclusion of the Floating Point Package into the Customer Software

This chapter shows how to insert the Floating Point Package into the user's written software. The symbolic definition of the working registers makes it necessary to include the FPP-definition file (FPPDEF.ASM) before the customer's software, otherwise the assembler allocates an address word for every use of one of the working registers during the first pass of the assembler. During the second assembler pass this proofs to be wrong and the assembler run will fail. The two files FPP03.ASM and CNV03.ASM need to be located together as shown in the examples below. This is due to the common parts that are connected with jumps.
The constant DOUBLE decides which FPP version will be generated.

```
;
           .text    0E000h        ; ROM/EPROM start address
STACK      .equ     0300h         ; Initial value for SP
;
DOUBLE     .equ     1             ; Insert .DOUBLE format FPP
SW_UFLOW   .equ     0             ; Underflow is no error
SW_RND     .equ     1             ; Use rounding for conversions
;
           .copy    c:\fpp\fppdef.asm      ; Definitions
           .copy    c:\fpp\fpp03.asm       ; FPP file
           .copy    c:\fpp\cnv03.asm       ; Conversions
;
; Customer software starts here
;
START      MOV      #STACK,SP              ; Allocate stack
           .....                           ;
```

```
; Power-up start address:
;
          .sect     "RstVect",0FFFEh
          .word     START                      ; Reset vector
```

A second possibility is shown below: the FPP i s located after the user's software:

```
;
          .text     0E000h          ; ROM start address
STACK     .equ      0300h ; Initial value for SP
;
DOUBLE    .equ      0               ; Insert .FLOAT format FPP
SW_UFLOW  .equ      1               ; Underflow is an error
SW_RND    .equ      0               ; No rounding for conversions
;
          .copy     c:\fpp\fppdef.asm          ; Definitions
;
; Customer software starts here
;
START     MOV       #STACK,SP                  ; Allocate stack
          .....                                ;
          .....                                ; End of user's software
          .copy     c:\fpp\fpp03.asm           ; Copy FPP file
          .copy     c:\fpp\cnv03.asm           ; Copy conversions
;
; Power-up start address:
;
          .sect     "RstVect",0FFFEh
          .word     START           ; Reset vector
```

5.6.10 Software Examples

5.6.10.1 Square Root Subroutines

The following two subroutines show the use of the Floating Point Package for the calculation of the square root out of a number. The NEWTONIAN approach is used:

$$x_{n+1} = 0.5 \times \left(x_n + \frac{A}{x_n} \right)$$

The subroutines use the same approach as the FPP subroutines: the input and the result are located on the top of the stack. A stack location is used for the counting of the approximation loops.
The used algorithm for the 1st estimation leads to worst case errors of +41% and -29%. The table below shows the maximum errors for each approximation step:

Step	Max. Error	Max. Error
1st estimation x0	+41%	-29%
1st approximation x_1	+6%	+6%
2nd approximation x_2	+0.17%	+0.17%

3rd approximation x_3	+1.5 ppm	+1.5 ppm
4th approximation x_4	$<2x10^{-12}$	$<2x10^{-12}$
5th approximation x_5	$<2x10^{-24}$	$<2x10^{-24}$

```
; Square Root Subroutine for .FLOAT format
; Calculate the square root out of A. A is located on TOS, where
; otherwise the results are located. The square root overwrites A
; For input RPARG and RPRES are not relevant
; SP, RPARG and RPRES point to the result on TOS
;
FLT_SQRT   TST.B   2(SP)           ; Argument negative?
           JN      SQRET           ; Yes, return with N = 1
           PUSH    #5              ; Loop count
           PUSH    8(SP)           ; A lsbs
           PUSH    8(SP)           ; A msbs to xn
;
; The 1st estimation x0 with halved exponent creates an error of
; max. 41% (1.414):
; this means 5 loops are sufficient for max. accuracy
;
           XOR.B   #080h,1(SP)     ;
           RRA.B   1(SP)           ; Exponent/2
           XOR.B   #080h,1(SP)     ; Back to exponent format
SQLOOP     MOV     SP,RPARG        ; Pointer to xn
           MOV     SP,RPRES
           ADD     #8,RPRES        ; Pointer to A
           SUB     #4,SP           ; Allocate stack for result
           CALL    #FLT_DIV        ; A/xn
           ADD     #4,RPARG        ; Point to xn
           CALL    #FLT_ADD        ; A/xn + xn
           DEC.B   1(RPRES)        ; 0.5 x (A/xn + xn) = xn+1
           MOV     @SP+,2(SP)      ; xn+1 -> xn
           MOV     @SP+,2(SP)
           DEC     4(SP)           ; Decr. loop count
           JNZ     SQLOOP
           MOV     @SP+,6(SP)      ; N = 0
           MOV     @SP+,6(SP)      ; Root to result space
           ADD     #2,SP           ; Skip loop count
SQRET      MOV     SP,RPARG        ; Set RPARG and RPRES to result
           ADD     #2,RPARG
           MOV     RPARG,RPRES
           RET
;
; Square Root Subroutine for .DOUBLE format
; Calculate the square root out of A. A is located on TOS, where
; otherwise the results are located. The square root overwrites A
; For input RPARG and RPRES are not relevant
; SP, RPARG and RPRES point to the result on TOS

DBL_SQRT   TST.B   2(SP)           ; Argument negative?
           JN      SQRET           ; Yes, return with N = 1
           PUSH    #5              ; Loop count
           PUSH    10(SP)          ; A lsbs
           PUSH    10(SP)          ; A mids
           PUSH    10(SP)          ; A msbs for 1st estimation xn
;
; The 1st estimation x0 with halved exponent creates an error of
; max. 41% (1.414):
; this means 5 loops are sufficient for max. accuracy
;
```

```
          XOR.B    #080h,1(SP)     ;
          RRA.B    1(SP)           ; Exponent/2
          XOR.B    #080h,1(SP)     ; Back to exponent format
SQLOOP    MOV      SP,RPARG        ; Pointer to xn
          MOV      SP,RPRES
          ADD      #10,RPRES       ; Pointer to A
          SUB      #6,SP           ; Allocate stack for result
          CALL     #FLT_DIV        ; A/xn
          ADD      #6,RPARG        ; Point to xn
          CALL     #FLT_ADD        ; A/xn + xn
          DEC.B    1(RPRES)        ; 0.5 x (A/xn + xn) = xn+1
          MOV      @SP+,4(SP)      ; xn+1 -> xn
          MOV      @SP+,4(SP)
          MOV      @SP+,4(SP)
          DEC      6(SP)           ; Decr. loop counter
          JNZ      SQLOOP
          MOV      @SP+,8(SP)      ; N = 0
          MOV      @SP+,8(SP)      ; Root to result space
          MOV      @SP+,8(SP)
          ADD      #2,SP           ; Skip loop count
SQRET     MOV      SP,RPARG        ; Set RPARG and RPRES to result
          ADD      #2,RPARG        ; Correct for return address
          MOV      RPARG,RPRES
          RET
```

5.6.10.2 Cubic Root Subroutines

The same way as shown for the square root the cubic root may be calculated using the NEWTONIAN approach. The formula for the cubic root out of A is:

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{A}{x_n^2}\right)$$

The used algorithm for the 1st estimation of the cubic root leads to worst case errors of +58% and -37%. The table below shows the maximum errors for each approximation step:

Step		Max. Error	Max. Error
1st estimation	x_0	+58%	-37%
1st approximation	x_1	+19%	+25%
2nd approximation	x_2	+3%	+4.7%
3rd approximation	x_3	+0.08%	+0.2%
4th approximation	x_4	+0.7 ppm	4.6 ppm
5th approximation	x_5	$<1.4\text{x}10^{-13}$	$2\text{x}10^{-11}$

```
; The cubic root is calculated for the .FLOAT number on the top
; of the stack. The result is written there too.
; For input RPARG and RPRES are not relevant
; SP, RPARG and RPRES point to the result on TOS
;
FLT_CUB      .EQU     $
             PUSH     #5            ; Loop count
```

```
            PUSH     8(SP)            ; A lsbs
            PUSH     8(SP)            ; A msbs
;
; The 1st estimation x0 needs to be calculated very close to the
; final result: the exponent is divided by 3.
;
            MOV.B    1(SP),RPARG      ; Exponent of A   00xx
            MOV.B    #080h,1(SP)      ; Set exponent of A to 2^0
            TST.B    RPARG            ; Exponent's sign?
            JN       DCL$2            ; positive
DCL$1       DEC.B    1(SP)            ; Neg. exp.: exponent - 1
            ADD.B    #3,RPARG         ; Add 3 until 080h is reached
            JN       CBLOOP           ; 080h is reached,
            JMP      DCL$1            ; Continue
DCL$3       INC.B    1(SP)            ; Pos. exp.: exponent + 1
DCL$2       SUB.B    #3,RPARG         ; Subtr. 3 until 080h is reached
            JN       DCL$3            ; Continue
;
CBLOOP      MOV      SP,RPARG         ; Point to xn
            MOV      SP,RPRES
            SUB      #4,SP            ; Allocate stack for result
            CALL     #FLT_MUL         ; xn^2
            ADD      #12,RPRES        ; Point to A
            CALL     #FLT_DIV         ; A/xn^2
            INC.B    5(SP)            ; xn x 2
            ADD      #4,RPARG         ; Point to 2xn
            CALL     #FLT_ADD         ; A/xn^2 + 2xn
            MOV      #FLT3,RPARG      ; 1/3 x (A/xn^2 + 2xn) = xn+1
            CALL     #FLT_DIV
            MOV      @SP+,2(SP)       ; xn+1 -> xn
            MOV      @SP+,2(SP)
            DEC      4(SP)            ; Decr. loop count
            JNZ      CBLOOP
            MOV      @SP+,6(SP)       ; N = 0
            MOV      @SP+,6(SP)       ; Root to result space
            ADD      #2,SP            ; Skip loop count
            MOV      SP,RPARG         ; Set RPARG and RPRES to result
            ADD      #2,RPARG         ; Skip return address
            MOV      RPARG,RPRES
            RET
FLT3        .FLOAT   3.0              ; Constant for cubic root

; The cubic root is calculated for the .DOUBLE number on the top
; of the stack. The result is written there too.
; For input RPARG and RPRES are not relevant
; SP, RPARG and RPRES point to the result on TOS
;
DBL_CUB     .EQU     $
            PUSH     #5               ; Loop count
            PUSH     10(SP)           ; A LSBs -> xn
            PUSH     10(SP)
            PUSH     10(SP)           ; A MSBs
;
; The 1st estimation x0 needs to be calculated very close to the
; final result: the exponent is divided by 3.
;
            MOV.B    1(SP),RPARG      ; Exponent of A   00xx
            MOV.B    #080h,1(SP)      ; Set exponent of A to 2^0
            TST.B    RPARG            ; Exponent's sign?
            JN       DCL$2            ; positive
DCL$1       DEC.B    1(SP)            ; Neg. exp.: exponent - 1
            ADD.B    #3,RPARG         ; Add 3 until 080h is reached
            JN       CBLOOP           ; 080h is reached,
         JMP      DCL$1          ; Continue
DCL$3    INC.B    1(SP)          ; Pos. exp.: exponent + 1
DCL$2    SUB.B    #3,RPARG       ; Subtr. 3 until 080h is reached
         JN       DCL$3          ; Continue
;
CBLOOP   MOV      SP,RPARG       ; Point to xn
         MOV      SP,RPRES
         SUB      #6,SP          ; Allocate stack for result
         CALL     #FLT_MUL       ; xn^2
         ADD      #16,RPRES      ; Point to A
         CALL     #FLT_DIV       ; A/xn^2
         INC.B    7(SP)          ; xn x 2
         ADD      #6,RPARG       ; Point to 2xn
         CALL     #FLT_ADD       ; A/xn^2 + 2xn
         MOV      #3,RPARG       ; 1/3 x (A/xn^2 + 2xn) = xn+1
         CALL     #FLT_DIV
         MOV      @SP+,4(SP)     ; xn+1 -> xn
         MOV      @SP+,4(SP)
         MOV      @SP+,4(SP)
         DEC      6(SP)          ; Decr. loop count
         JNZ      CBLOOP
         MOV      @SP+,8(SP)     ;
         MOV      @SP+,8(SP)     ; Cubic root to result space
         MOV      @SP+,8(SP)
         ADD      #2,SP          ; Skip loop count
         MOV      SP,RPARG       ; Set RPARG and RPRES to result
         ADD      #2,RPARG       ; Skip return address
         MOV      RPARG,RPRES
         RET
DBL3     .DOUBLE  3.0            ; Constant for cubic root
```

5.6.10.3 Fourth Root Subroutine

The fourth root of a number is calculated by calling the square root subroutine twice. EXAMPLE: the fourth root is calculated for a number residing in RAM at address NUMBER (MSBs). The fourth root is written to RESULT. The previous result on TOS must not be overwritten.

```
;
         PUSH     NUMBER+2       ; LSBs of NUMBER to new space
         PUSH     NUMBER         ; MSBs of NUMBER
         CALL     #FLT_SQRT      ; Square root on TOS
         CALL     #FLT_SQRT      ; Fourth root on TOS
         MOV      @SP+,RESULT    ; 4th root MSBs
         MOV      @SP+,RESULT+2  ; SP to previous result
```

5.6.10.4 Other Root Subroutines

The same way as shown above higher roots may be calculated using the NEWTONIAN approach. The generic formula for the mth root out of A is:

$$x_{n+1} = \frac{1}{m}\left((m-1)x_n + \frac{A}{x_n^{m-1}}\right)$$

5.6.10.5 Calculations with Intermediate Results

If a calculation cannot be executed straight forward but has intermediate results then simply a new result space is used. This is done by subtracting 4 (.FLOAT) resp. 6 (.DOUBLE) from the stack pointer SP.

EXAMPLE: The function for e shown below is to be calculated. The example is shown for the .FLOAT format, for the .DOUBLE format 6 is used for the constants where 4 is used now.

$$e = a \times b - \frac{c}{d}$$

```
;
        SUB     #4,SP           ; Allocate result space 0 (RS0)
        MOV     #a,RPRES        ; Address a
        MOV     #b,RPARG        ; Address b
        CALL    #FLT_MUL        ; a x b -> RS0
        SUB     #4,SP           ; Allocate result space 1 (RS1)
        MOV     #c,RPRES        ; Address c
        MOV     #d,RPARG        ; Address d
        CALL    #FLT_DIV        ; c/d -> RS1
        ADD     #4,RPRES        ; Address (a x b) in RS0
        CALL    #FLT_SUB        ; e = (a x b) - c/d -> RS1
        MOV     @SP+,2(SP)      ; Result e to RS0
        MOV     @SP+,2(SP)      ; Overwrite (a x b) with e
;
; Housekeeping is made, SP points to RS0 again, but not RPARG and ;
RPRES
;
```

EXAMPLE: The multiply-and-add (MAC) function for e shown below is calculated. The example is written for the .DOUBLE format otherwise 4 is used for the constants where 6 is used now.

$$e_{n+1} = a \times b + e_n$$

```
;
        SUB     #6,SP           ; Allocate result space
        MOV     #a,RPRES        ; Address a
        MOV     #b,RPARG        ; Address b
        CALL    #FLT_MUL        ; a x b
        MOV     #e,RPARG        ; Address e
        CALL    #FLT_ADD        ; (a x b)+ e
        MOV     @RPARG+,e       ; Actualize e with result
        MOV     @RPARG+,e+2     ; MIDs
        MOV     @RPARG,e+4      ; LSBs
;
; SP and RPRES still point to the result, RPARG may be used
; for the next argument address.
```

5.6.10.6 Absolute Value of a Number

If the absolute value of a number is needed, this is simply done by resetting the sign bit of this number.

EXAMPLE: the absolute value of the result on the top of the stack is needed.

```
;
            BIC       #080h,0(SP)       ; |result| on TOS
;
```

5.6.10.7 Change of the Sign of a Number

If a sign change is necessary (multiplication by -1), this is simply done by inverting the sign bit of this number.

EXAMPLE: the sign of the result on the top of the stack is changed.

```
;
            XOR       #080h,0(SP)       ; Negate result on TOS
;
```

5.6.10.8 Integer Value of a Number

The integer value of a floating point number can be calculated with the subroutine FLT_INTG below. The pointer RPARG is loaded with the address of the number, the result is placed on the top of the stack. No error is possible. Numbers lower than one are returned as zero. The subroutine can handle .FLOAT and .DOUBLE formats.

```
;
; Calculate the integer value of the number RPARG points to.
; Result: on top of the stack. RPARG, RPRES and SP point to it
;
FLT_INTG MOV.B    1(RPARG),COUNTER         ; Exponent to COUNTER
           MOV      @RPARG+,2(SP)            ; MSBs and Exponent
           MOV      @RPARG+,4(SP)            ; LSBs .FLOAT
           .if      DOUBLE=1
           MOV      @RPARG,6(SP)             ; LSBs .DOUBLE
           .endif
           MOV      #0FFFFh,ARG2_MSB         ; Mask for fractional part
           .if      DOUBLE=1
           MOV      #0FFFFh,ARG2_MID
           .endif
           MOV      #0FFFFh,ARG2_LSB
           JMP      L$30
;
INTGLP     CLRC                      ; Shift 0 in always
           RRC.B    ARG2_MSB         ; Shift mask to next lower bit
           .if      DOUBLE=1
           RRC      ARG2_MID
           .endif
           RRC      ARG2_LSB
           DEC      COUNTER          ; Shift as often as:
L$30       CMP      #080h,COUNTER    ; SHIFT COUNT = EXPONENT - 07Fh
           JHS      INTGLP
           BIC      ARG2_MSB,2(SP)   ; Mask out fract. part
           .if      DOUBLE=1
           BIC      ARG2_MID,4(SP)   ; For .DOUBLE format
           BIC      ARG2_LSB,6(SP)
           .else
           BIC      ARG2_LSB,4(SP)   ; For .FLOAT format
           .endif
           MOV      SP,RPARG         ; Both pointer to result`s MSBs
           ADD      #2,RPARG
           MOV      RPARG,RPRES
           RET                       ; Return with Integer on TOS
;
```

EXAMPLE: the integer value of the floating point number residing at address VOL1 is placed on TOS.

```
            MOV       #VOL1,RPARG       ; Load pointer with address
            CALL      #FLT_INTG         ; Calculate integer of VOL1
;
```

5.6.10.9 Fractional Part of a Number

The fractional part of a floating point number can be calculated with the subroutine FLT_FRCT below. The pointer RPARG is loaded with the address of the number, the result is placed on the top of the stack. No error is possible. The subroutine can handle both floating point formats. The subroutine calls the subroutine FLT_INTG shown above. Integer values or very large numbers return a zero value due to the missing resolution:

.DOUBLE format:	numbers > 1.099512×10^{12}	($>2^{40}$)
.FLOAT format:	numbers > 1.6777216×10^{7}	($>2^{24}$)

```
;
;
; Calculate the fractional part of the number RPARG points to.
; Result: on top of the stack. RPARG, RPRES and SP point to it
; Subroutine FLT_INTG is called
;
FLT_FRCT MOV        RPARG,RPRES       ; Copy operand's address
         .if        DOUBLE=1
         PUSH       4(RPARG)          ; Copy operand to allow the use
         .endif                       ; of the value on TOS
         PUSH       2(RPARG)
         PUSH       @RPARG
         CALL       #FLT_INTG         ; Integer part of operand to TOS
         MOV        SP,RPARG          ; Integer part address
         CALL       #FLT_SUB          ; Operand - Integer part to TOS
         .if        DOUBLE=1          ; Housekeeping:
         MOV        @SP+,6(SP)        ; Fractional part back
         MOV        @SP+,6(SP)        ; .DOUBLE format
         MOV        @SP+,6(SP)
         .else
         MOV        @SP+,4(SP)        ; .FLOAT format
         MOV        @SP+,4(SP)
         .endif
         MOV        SP,RPARG          ; Both pointer to result's MSBs
         ADD        #2,RPARG
         MOV        RPARG,RPRES
         RET
;
```

EXAMPLE: the fractional part of the floating point number R4 points to is placed on TOS.

```
         MOV        R4,RPARG          ; Load pointer with address
         CALL       #FLT_FRCT         ; Calculate fractional part
         ....                         ; Fractional part on TOS
```

6 HINTS AND RECOMMENDATIONS

During the software development for the first MSP430 projects a lot of experience was acquired. The following hints and recommendations are conceived for all programmers and system designers having more experience with 4- and 8-bit microcomputers than with 16-bit systems. Also mentioned are deviations 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).

- **Frequently used Bits:** 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 of BCD arithmetic:** if simple up/down counters are used that are to be displayed: this saves time and ROM space due to the 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
```

- **Conditional Assembly:** this feature of the MSP430 assembler allows to get 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".

- **Usage of Bytes:** Use bytes wherever appropriate. The MSP430 allows using every instruction with bytes. (exceptions are only SWPB, SXT and CALL)

- **Use of Status Bytes or Words:** Use status bytes or words, not flags for remembering of states. This allows 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 the Addressing Modes:**
  Use the Symbolic Mode for random accesses
  Use the Absolute Mode for fixed addresses such as peripheral addresses
  Use the Indexed Mode for random accesses in tables
  Use the Register Mode for time critical processing and as the normal one
  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

> The above mentioned possibilities make rigid "house keeping" necessary: every program 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 advised 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 Program Counter PC:** The PC can be accessed as 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:** it 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.

- **Enabling of the General Interrupt:** The instruction following the enabling of the interrupt is executed before an interrupt is accepted:

```
EINT                ; Enable interrupt (GIE)
CLRC                ; This instruction is executed before
ADC      R5         ; the 1st interrupt is accepted
```

- **High Speed Multiplication:** 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              CYCLES          EXAMPLE
;-------------------------------------------------------------------
; MINIMUM            80             00000h x 00000h = 000000000h
; MEDIUM  96                        0A5A5h x 05A5Ah = 03A763E02h
; MAXIMUM           112             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
;
```

- **Special Use of the Carry Bit:** 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 if the result of an operation is zero then the carry is reset and vice versa. The instructions having this feature 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 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             ; P0.0 high ? .NOT.Zero -> carry
        RLA     R5                   ; Shift bit into R5
```

- **The Carry Bit used for Increments:** 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
```

- **Immediate Addition of the Carry Bit:** 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" Programming:** 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 program falls through it. This saves 6 bytes of ROM: the CALL itself and the RET instruction. The I^2C 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                ; Fall through to SUBR2
;
SUBR2   ...                          ; Start of  subroutine SUBR2
        RET
```

- **Shift Operations for 32-bit Numbers:** If shifts with numbers greater than 16 bits are necessary the shift operations for the upper words must be RLC or RRC. If RLA or RRA are used then only zeroes are shifted in

```
        RLA     R11                  ; MSB of low byte to carry
        RLC     R12                  ; RLA is wrong here!

        RRA     R12                  ; LSB of high byte to carry
        RRC     R11                  ; RRA is wrong here!

                                     ; R13|R14|R15 = R10|R11,R12
```

- **Interrupt Handlers:** the length of interrupt handlers should be kept as short as possible. All necessary computations should be made in the background program (main program). The activation and control can be made easily with status bytes.

6.1 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 to be done (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 software.
7. Coding of the software with an editor
8. Assembling of the program 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

6.2 Most often occurring Software Errors

During software development the same errors are made by nearly all assembler programrs. The following list contains the errors which are most often heard of and experienced.

- **Missing "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 program crash, due to wrong data being written into the Program Counter.
  The Stack Pointer needs to be initialized before the EINT instruction is executed.

- **Use of the wrong Jump Instructions:** the conditional jump instructions JLO and JLT, or JHS and JGE, give different results if used for numbers above 07FFFh. It is therefore necessary to distinguish always between signed and unsigned comparisons.

- **Wrong Completion Instructions.** Despite their virtual similarity, subroutines and interrupt handlers need completely different actions for completion.
  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 Program 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 Working Registers: byte instructions always clear the upper byte of the used working register (except CMP.B, TST.B, BIT.B). It is necessary therefore to use word instructions if operations in working registers can exceed the byte range.

- **Use of Byte Instructions with the Program 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 is to 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, and 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.

- **Counters and Timers longer than 16 bits:** 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 simply 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 read 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 that a timer interrupt that occurs between the reading of TIMLO and TIMHI can falsify 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
```

- **Counters used by Foreground and Background:** if counters are modified by the foreground and read and cleared by the background care is to be taken that no counts are lost. With the following example it is possible to loose a count if the interrupt occurs between the MOV and the CLR instruction: the additional count is not recognized because CNTR (with its content 1) is cleared.

```
INT_HAN INC     CNTR                ; Incr. counter CNTR
        RETI                        ; by interrupts

        ...                         ; Background program
        MOV     CNTR,STORE          ; Read CNTR
        CLR     CNTR                ; Counts may be lost!
        ...
```

To avoid the loosing of counts the following solutions are possible for the background part:

```
;
; Background part switches off the interrupt during reading
;
        DINT                        ; GIE <- 0 (inactive after MOV)
        MOV     CNTR,STORE          ; Read CNTR
        CLR     CNTR                ; Clear unmodified CNTR
        EINT                        ; Enable interrupt again
;
; Background part uses difference of contents. If interrupt occurs
; after the PUSH instruction, 1 remains in CNTR.
;
        PUSH    CNTR                ; Copy CNTR
        SUB     *SP,CNTR            ; Subtract read number from CNTR
        POP     STORE               ; Place read info to STORE
```

- **Use of the PUSH Instruction:** 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
```

EXAMPLE: The stored Stack Pointer SP does not point to the same stack address after the restoring: it points to the address -2 afterwards.

```
;
        PUSH     SP          ; Store SP-2 on stack
;
        POP      SP          ; Restore SP-2 to SP !!
```

- **Register Overflow:**: if registers do not have the necessary length negative numbers (MSB = 1) or too small numbers (register is reset to zero by overflow) may result. The length of registers needs to be evaluated with "worst case" methods.

- **Interrupt Blocking:** long interrupt routines should be avoided. If they are necessary then the GIE bit located in the Status Register should be set at the start of these routines. Otherwise the disabled interrupt blocks all other interrupt sources.

- **Real Time Processing:** if the used algorithm is longer than the time slot that is available then errors will occur. "Worst case" evaluations are necessary to guarantee the fitting of the algorithm.

- **Open Inputs:** every inputs needs to have a defined potential. Otherwise hum and noise will influence the program flow.

- **Crystal turn-on Time:** if woken-up from the Low Power Mode 4 the crystal needs a relatively long time until it runs with the correct frequency. This may last up to three seconds. No correct timing is possible until the crystal reached its nominal frequency. Up to this the MCLK generator runs with its lowest frequency.

- "**Frequency Locked Loop**" **Considerations**:.

- **FLL turn-on Time:** if woken-up from LPM3 the FLL needs approximately 6 cycles to reach the nominal frequency. This time needs to be added to the 6 cycles of the interrupt latency time.
- **Setting Time:** the FLL needs a certain noninterrupted time to set the control value of the DCO. If this time is not provided no control for the DCO is possible, it remains on the same point. This time is spent best during initialization by a software loop with a worst case length of 28 x 32 x 30.5 μs = 27.3 ms. To allow the system clock the adaptation of the "Digitally Controlled Oscillator:" to the eventually changed tap, the FLL-loop should be closed during longer calculations. This is simply done with the instruction:

```
;
  BIC    #SCG0,SR            ; Turn on FLL-loop control
```

- **Supply Voltage for Battery driven Systems:** if certain batteries are used the supply voltage may go below the lower limit during Active Mode (especially if the ADC is used) due to the high resistance of these batteries. A capacitor is necessary then in parallel to the battery.

- **Supply Voltage for Mains driven Systems:** no hum, noise and spikes are allowed. If present the reliability of the system and the accuracy of the ADC will decrease.

- **EEPROM clocking:** for some EEPROMs the minimum clock duration is longer than one MSP430 instruction . This means that NOPs have to be included into the clock timing.

6.3 Run Time Estimation

To get a quick overview concerning the speed of a given piece of software, the following estimations may be used:

- If the code contains all addressing modes then the estimation for the needed runtime t_{run} is:

$$t_{run} = 0.75 \text{ cycles/byte}$$

- If the code contains only or predominant register mode addressing then the estimation for the needed runtime t_{run} is:

$$t_{run} = 0.5 \text{ cycles/byte}$$

7 INSTRUCTION SET

NOTE

All marked instructions (*) are emulated instructions. The emulated instructions use core instructions. Emulated single operand instructions (e.g. RLA) can not use all seven addressing modes for the operand: only the four addressing modes usable for the destination are possible. The branch instruction BR is the only exception to this rule.

@Z stands for: the Carry bit has the inverted information of the Zero bit.

				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	*	*	@Z
	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	*	*	@Z
*	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	*	*	*	@Z
	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	-	-	-	-
	JLT	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	-	-	-	-
	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	*	*	@Z
*	TST[.W];TST.B	dst	Test destination	*	*	*	*
	XOR[.W];XOR.B	src,dst	src .xor. dst → dst	*	*	*	@Z

APPENDIX

A1 CPU REGISTERS

All of the MSP430 CPU-registers can be used with all instructions.

A1.1 The Program Counter R0

One of the main differences to other microcomputer architectures relates to the Program 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 Program Counter too. A branch, for example, 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

The Program Counter always points to even addresses: this means that the LSB is always zero. The software has to ensure that no odd addresses are used if the Program Counter is involved. Odd PC addresses will result in non-predictable results.

A1.2 Stack Processing

A1.2.1 Usage of the System Stack Pointer R1

The system stack pointer (SP) is a normal register like the others. 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 too for interrupts and subroutine calls. To ensure the error free running of the program it is necessary to do exact "housekeeping" for the system stack.

NOTE

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 in 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
```

A1.2.2 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 those 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
```

A1.3 Byte and Word Handling

Every word is addressable by three addresses as shown in the figure below:
- 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

> Registers R0 to R15 do not have an address but are treated in a special way: Byte addressing always uses the lower byte of the register. The upper byte is set to zero if the instruction modifies the destination: therefore all instructions clear the upper byte of a register except CMP.B, TST.B, BIT.B and PUSH.B. The source is never affected.

The way an instruction treats data is defined with its extension:

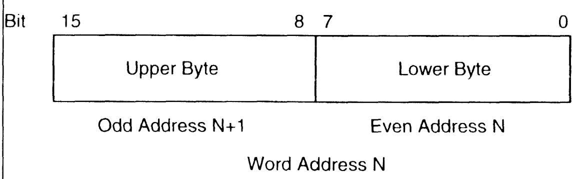
- The extension .B means byte handling
- The extension .W (or none) means word handling

EXAMPLES: The next two software 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.

```
ADD.B    &050h,R5          ; ADD 8-bit VALUE TO R5
```



Word/Byte Configuration

If registers are used with byte instructions the upper byte of the destination register is set to zero for all instructions except CMP.B, TST.B, BIT.B and PUSH.B. 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
;
```

A1.4 Constant Generator

A statistical look at 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	Field 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

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 often used bits of the peripheral registers are located in the bits addressable by the Constant Generator bits whenever possible.

A1.5 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. autoincrement	---	*R5+
11	Immediate	---	#TABLE

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 be placed always into the source. Only in a very few cases it will 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 Autoincrement Mode: With table computing a method is usable that saves ROM-space and the number of used registers additionally:

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 can 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 program 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, addresses or constants within the instruction,

With the use 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 is to be done is made immediately without any overhead.

A1.6 Program Flow Control

A1.6.1 Computed Branches and Calls

The Branch instruction is an emulated instruction which moves the destination address into the Program Counter:

```
         MOV      dst,PC          ; EMULATION FOR BR dst
```

The possibility to access the Program Counter in the same way as all other registers gives interesting options:

1. The destination address can be taken from tables
2. The destination address may be computed
3. The destination address may be a constant

A1.6.2 Nesting of Subroutines

Due 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.

A1.6.3 Nesting of Interrupts

Nesting of interrupts gives no problem at all, provided there is enough RAM for the stack. For every occurring interrupt two words on the stack are needed for the storage of the Status Register and the return address. To enable nested interrupts it is only necessary

to include an EINT instruction into the interrupt handler. If the interrupt handlers are as short as possible (a good real-time practice) then nesting may not be necessary.
EXAMPLE: The Basic Timer interrupt handler is woken-up with 1 Hz only but has to do a lot of things. The interrupt nesting is used therefore. The latency time is 8 clock cycles only.

```
;
; Interrupt handler for Basic Timer: Wake-up with 1 Hz
;
BT_HAN   EINT                          ; Enable interrupt for nesting
         INC.B     SECCNT              ; Counter for seconds +1
         CMP.B     #60,SECCNT          ; 1 minute elapsed?
         JHS       MIN1                ; Yes, do necessary tasks
         RETI                          ; No return to LPM3
;
; One minute elapsed: Return is removed from stack, a branch to
; the necessary tasks is made. There it is decided how to proceed
;
MIN1     INC       MINCNT              ; Minute counter +1
         CLR       SECCNT              ; 0 -> SECCNT
         ...                           ; Start of necessary tasks
         RETI                          ; Tasks completed
```

A1.6.4 Jumps

Jumps allow the conditional or unconditional leaving of the linear program flow. The Jumps cannot reach every address of the address map but they have the advantage to need 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 a few cases is the 2-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	Program 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	After signed comparisons
JLT label	Jump if N .XOR. V = 1	After signed comparisons
JN label	Jump if N = 1	Test for sign of a result

NOTE

> It is important to use the appropriate conditional Jump for signed and unsigned data. For positive data (0 to 07FFFh resp. 0 to 07Fh) both signed and unsigned conditional jumps behave similarly. This changes completely when used with negative data (08000h to 0FFFFh resp. 080h to 0FFh): the signed conditional jumps treat negative data as smaller numbers than the positive ones, and 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 be only 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 within 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

> If addresses are computed only the unsigned jumps are adequate: addresses are always unsigned, positive numbers.

No "Jump if Overflow" is provided because the Overflow Bit located in the Status Register is used primarily for the signed jumps. If the status of the Overflow Bit is needed from the software a simple bit test can be used:

```
;
        OV      .EQU    0100h    ; Bit address in SR
;
        BIT     #OV,SR           ; Test Overflow Bit
        JNZ     OVFL             ; If OV = 1 branch to label OVFL
        ...                      ; If OV = 0 continue here
```

A2 SPECIAL CODING TECHNIQUES

The flexibility of the MSP430 CPU together with a powerful assembler allows coding techniques not available with every microcomputer. The most important ones are explained below.

A2.1 Conditional Assembly

For a detailed description of the syntax please refer to "MSP430 Family Assembler Language Tools".

Conditional assembly provides the possibility to compile different lines of source into the object file depending on the value of an expression that is defined in the source of the program. This makes it easy to alter the behaviour of the code with modifying one single line in the source.

The following example shows how to use conditional assembly. The example will allow easy debugging of a program 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 can be achieved. The following piece of code shows more built-in ways to debug the written 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
```

Conditional Assembly is not restricted to the debug phase of software development. The main use is normally to get different software versions out of one source. For every version only the necessary software parts are assembled and the not needed parts are left out by Conditional Assembly. The big advantage is the single source that is to be maintained.
An example is a Floating Point Package with different number lengths (32, 48 and 64 bits) contained in one source. Before assembly the desired length is defined by an .EQU directive.

A2.2 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 necessary. PIC is possible with the MSP430 mainly due to the allocation of the PC inside the register bank. The availability of the PC is made much use of. 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. The loaded code may have several purposes:

- Application specific software that is different for some devices
- Additional code that was not anticipated before mask generation
- Test routines for manufacturing purposes

A2.2.1 Referencing of Code inside Position Independent Code

The referenced code or data is located in the same block of PIC as the program resides.

Jumps

Jumps are position independent anyway: 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
```

A2.2.2 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 or 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 program examples may be implemented as MACROs if needed. This would ease the usage and transparency.

A2.3 Reentrant Code

If the same subroutine is used by the background program 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 for several tasks is called "Reentrancy".
Reentrancy allows the calling of a subroutine despite the fact that the current using task has not yet finished the subroutine.
The main difference of a reentrant subroutine to a normal one is that the reentrant routine contains only "pure code": that is, no part of the routine is modified during the 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
          ...
```

A2.4 Recursive Code

Recursive subroutines are subroutines that call themselves. This is not possible with normal architectures; stack processing is necessary for this often used feature. A simple example for recursive code is a lineprinter handler that calls itself for inserting of a "Form Feed" after a certain number of printed lines. This self-calling allows using all of the existent checks and features of the handler without the need to write it 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?
          JLT       L$500              ; No, proceed
          CALL      #LPHAND            ;
          .BYTE     CR,FF              ; Yes, output Carriage Return
          ...                            ; and Form Feed
L$500     ...
```

A2.5 Flag Replacement by Status Usage

Flags have several disadvantages if used for program control:

- Missing transparency (flags may depend on other flags)
- Possibility of nonexistent flag combinations if not handled very carefully
- Slow speed: The flags can be tested only serially

The MSP430 allows the use of a status (contained in a RAM byte or register) which defines the current program 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 program parts that are used currently define the new status dependent on the actual conditions: normally the status is only incremented, but it may change more randomly too.

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 program 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 next 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
;
```

A2.6 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 passed information itself may be numbers, addresses, counter contents, upper and lower limits etc. It only depends on the application.

A2.6.1 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).

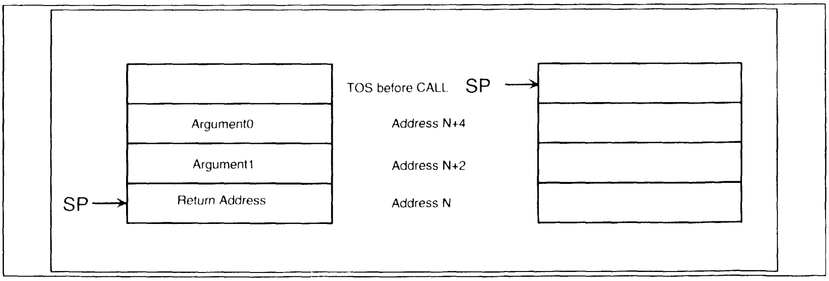
Advantages:
- Usable generally; no registers have to be freed for argument passing
- Variable arguments are possible

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
```

After the subroutine call the stack looks as follows: After the RET, it looks like this:

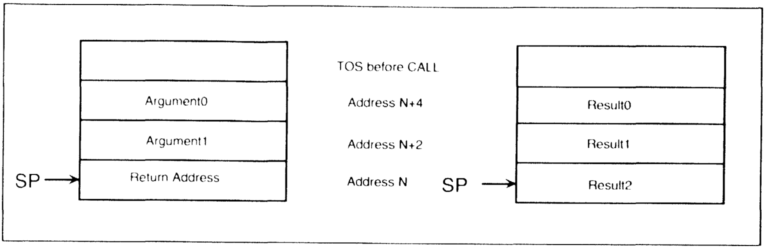


EXAMPLE: The subroutine SUBR gets its information from two arguments pushed onto the stack before the calling. Three result words are returned on the stack: it is the responsibility of the calling program to pop the results from the stack.

```
        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 the stack looks as follows: After the RET, it looks like this:



NOTE

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 to move the SP above the last item on the stack; indexed addressing is needed instead.

A2.6.2 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 easily 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
```

A2.6.3 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
        RETS                    ; NORMAL RETURN
```

A2.7 Interrupt Vectors in RAM

If the destination address of an interrupt changes with the program 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
```

REFERENCES

TSS721 M-Bus Transceiver Application Guide.		1994	SLAAE03
MSP430 Family Assembly Language Tools User's Guide		1994	SLAUE12
MSP430 Family Software User's Guide		1994	SLAUE11
MSP430 Family Architecture Guide and Module Library		1994	SLAUE10A
Data Encryption Algorithm	ANSI	1981	ANSI X3.92-1981
"TMS320DSP Designer's Notebook Number 43		1994	

TI European Sales Offices

BELGIQUE/BELGÏE

Texas Instruments S.A./N.V.
Avenue Jules Bordetlaan 11
1140 Bruxelles/Brussels
Tel: (02) 745 54 00
Fax: (02) 745 54 10

DEUTSCHLAND

Texas Instruments GmbH
Haggertystr.1
85356 Freising
Tel: (0 81 61) 80-0 od. Nbst.
Fax: (0 81 61) 80 45 16

Kirchhorster Straße 2
30659 Hannover
Tel: (0511) 90 49 60
Fax: (0511) 6 49 03 31

Maybachstraße 11
73760 Stuttgart - Ostfildern
Tel: (0711) 3 40 30
Fax: (0711) 3 40 32 57

ITALIA

Texas Instruments S.p.A.
Centro Direzionale Colleoni
Palazzo Perseo – Via Paracelso, 12
20041 Agrate Brianza (Mi)
Tel: (039) 6 84 21
Fax: (039) 6 84 29 12

ESPAÑA

Texas Instruments S.A.
c/Gobelas 43
28023 Madrid
Tel: (1) 3 72 80 51
Fax: (1) 3 72 82 66

FRANCE, MIDDLE EAST & AFRICA

Texas Instruments
8-10 Avenue Morane-Saulnier
B.P. 67
78141 Vélizy Villacoublay
Cedex
Tel: (1) 30 70 10 01
Fax: (1) 30 70 10 54

HOLLAND

Texas Instruments B.V.
Amsterdamseweg 204
1182 HL Amstelveen
Tel: (020) 5 46 98 00
Fax: (020) 6 46 31 36

HUNGARY

TI Technical Information Office
Hvsvlgyi #t. 54
1021 Budapest
Tel: (1) 176 3733/ext.
Fax: (1) 202 6219

EIRE

Texas Instruments Ltd.
7/8 Harcourt Street
Dublin 2
Tel: (01) 4 75 52 33
Fax: (01) 4 78 14 63

SUOMI/FINLAND

Texas Instruments OY
Tekniikantie 12
02150 Espoo
Tel: (0) 43 54 20 33
Fax: (0) 46 73 23

SVERIGE, DANMARK & NORGE

Texas Instruments
Halsingegatan 40
113 85 Stockholm
Tel: (08) 587 555 00
Fax: (08) 587 555 90

UNITED KINGDOM

Texas Instruments Ltd.
800 Pavilion Drive
Northampton Business Park,
Northampton NN47YL
Tel: (01604) 663000
Fax: (01604) 663001
Technical Enquiry Service
Tel: 0033-130 70 11 65

E-PIC
TI SC European Product Information Center

MULTILINGUAL TECHNICAL HOTLINE:

Francais:	+33 (0)1 30 70 11 64
English:	+33 (0)1 30 70 11 65
Italiano:	+33 (0)1 30 70 11 67
Deutsch:	+49 (0)8161 80 33 11
E-mail	epic@ti.com
24 Hours FAXLINE:	+33 (0)1 30 70 10 32

APPLICATION SUPPORT (Via MODEM):

BBS:	+33 (0)1 30 70 11 99
INTERNET:	http://www.ti.com

If you have access to the Internet, you may like to visit TI's World Wide Web server at:

http://www.ti.com

EXTENDING YOUR REACH™

TEXAS INSTRUMENTS