Chapter 5

Software Applications

5.1 Integer Calculation Subroutines

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

- ❏ Speed: Highest speed is possible especially when no loops are used
- ❏ ROM space: Least amount of ROM space is needed for these subroutines
- ❏ 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. They can be changed as needed.

```
; Integer Subroutines Definitions: Software Multiply
;
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
;
; Hardware Multiplier
;
ResLo    .EQU     013Ah            ; HW_MPYer: Result reg. LSBs
ResHi    .EQU     013Ch            ; Result register MSBs
SumExt   .EQU     013Eh            ; Sum Ext. Register
```

All multiplication subroutines shown in the following section permit two different modes:

- ❏ The normal multiplication: the result of the multiplication is placed into the result registers
- ❏ 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 in the Figure 5–1:

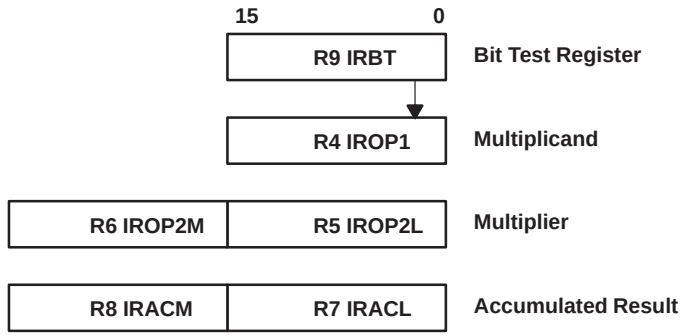

Figure 5–1. 16 x 16 Bit Multiplication – Register Use

```
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; 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
```

If the hardware multiplier is implemented then the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, a NOP is necessary after the MACRO call to allow the completion of the multiplication. The SumExt Register contains the carry after the MAC instruction; 0 (no carry) or 1 (carry occurred).

```
; Macro Definition for the unsigned multiplication 16 x 16 bits
;
MPYU    .MACRO  arg1,arg2               ; Unsigned MPY 16x16
        MOV     arg1,&0130h
        MOV     arg2,&0138h
        .ENDM                           ; Result in ResHi|ResLo
;
; Multiply the contents of two registers
;
        MPYU    IROP1,IROP2L            ; CALL the MPYU macro
        MOV     ResLo,R6                ; Fetch LSBs of result
        MOV     ResHi,R7                ; Fetch MSBs of result
        ...
;
; Multiply the operands located in a table, R6 points to
;
        MOV     #ResLo,R5               ; Pointer to LSBs of result
        MPYU    @R6+,@R6                ; CALL the MPYU macro
        NOP                             ; NOP: allow completion of MPYU
        MOV     @R5+,R7                 ; Fetch LSBs of result
```

```
        MOV     @R5,R8                  ; Fetch MSBs of result
;
; Macro Definition for the unsigned multiplication and
; accumulation 16 x 16 bits
;
MACU    .MACRO  arg1,arg2               ; Unsigned MAC 16x16
        MOV     arg1,&0134h             ; Carry in SumExt
        MOV     arg2,&0138h
        .ENDM                           ; Result in SumExt|ResHi|ResLo
;
; Multiply and accumulate the contents of two registers
;
        MPYU    R5,R6                   ; Initialize SumExt|ResHi|ResLo
        MACU    IROP1,IROP2L            ; Add IROP1 x IROP2 to result
        ADC     &SumExt,RAM             ; Add carry to RAM extension
;
```

5.1.1.1 Run Time Optimized Unsigned Multiplication 16 x 16-Bits

If the operands of the multiplication subroutine are shorter than 16 bits, the previous multiplication subroutine MPYU can be optimized during run time

The multiplication stops immediately after the operand IROP1 equals zero. This indicates that the operand with leading zeroes should be in IROP1. This run time optimized subroutine can be used instead of the normal subroutine. (The subroutine was developed by Leslie Mable/UK).

```
;
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; TASK            MACU              MPYU      IROP1    IROP2
;-------------------------------------------------------------
; MINIMUM          18                20      00000h x 00000h = 000000000h
; MEDIUM           90                92      000FFh x 0FFFFh = 000FEFF01h
; MAXIMUM         170               172      0FFFFh x 0FFFFh = 0FFFE0001h
; UNSIGNED MULTIPLY SUBROUTINE (Run time optimized):
; IROP1 x IROP2L –> IRACM|IRACL
;
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRACM
;
```

```
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
L$002   BIT     #1,IROP1                ; TEST ACTUAL BIT (LSB)
        JZ      L$01                    ; IF 0: DO NOTHING
        ADD     IROP2L,IRACL            ; IF 1: ADD MULTIPLIER TO RESULT
        ADDC    IROP2M,IRACM
L$01    RLA     IROP2L                  ; Double MULTIPLIER IROP2
        RLC     IROP2M                  ;
;
        RRC     IROP1                   ; Next bit of IROP1 to LSB
        JNZ     L$002                   ; If IROP1 = 0: finished
        RET
```

5.1.1.2 Fast Unsigned Square Function

For some applications, a fast square function is necessary. Two different solutions are given:

- ❏ For 16-bit unsigned numbers without rounding
- ❏ For 14-bit unsigned numbers with rounding. This version is adapted to the output of the ADC of the MSP430C32x family.

Both use table processing; an offset to a table containing the squared input numbers is built. The given cycles include the move of the operand into R5.

```
; Fast unsigned squaring for a 16 bit number. The upper 16 bits
; of the result are moved to R5. No rounding is used. 7 cycles
;
        MOV.B   DATA+1,R5               ; MSBs to R5
        RLA     R5                      ; Number x 2 (word table address)
        MOV     SQTAB(R5),R5            ; MSBs^2 to R5
        ...                             ; Squared value in R5
;
; Fast unsigned squaring for a 14 bit number. The upper 16 bits of
; the result are added to a buffer SQSUM. Rounding is used.
```

```
; 18 cycles. If registers are used for the sum: 12 cycles
;
        MOV     &ADAT,R5                ; ADC result to R5
        ADD     #80h,R5                 ; Round high byte
        SWPB    R5                      ; MSBs to LSBs
        RLA.B   R5                      ; Number x 2 (word table address)
        ADD     SQTAB(R5),SQSUM         ; Add MSBs^2 to SQSUM
        ADC     SQSUM+2                 ; Add carry
        ...                             ; Continue
;
; Table with squared values. Length may be adapted to the maximum
; possible input number.
;
SQTAB   .word   ($–SQTAB)*($–SQTAB)/4   ; 0 x 0 = 0
        .word   ($–SQTAB)*($–SQTAB)/4   ; 1 x 1 = 1
        .word   ($–SQTAB)*($–SQTAB)/4   ; 2 x 2 = 4
        ...
        .word   ($–SQTAB)*($–SQTAB)/4   ; 0FFh x 0FFh = 0FE01h
        .word   0FFFFh                  ; Max. for 0100h x 0100h
```

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 used is the same as with the unsigned multiplication. Therefore, Figure 5–1 is also valid.

```
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; 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
```

If the hardware multiplier is implemented then the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, then a NOP is necessary after the MACRO call to allow the completion of the multiplication. The SumExt Register contains the sign of the result in ResHi and ResLo; 0000h (positive result) or 0FFFFh (negative result).

```
; Macro Definition for the signed multiplication 16 x 16 bits
;
MPYS     .MACRO   arg1,arg2                ; Signed MPY 16x16
         MOV      arg1,&0132h
         MOV      arg2,&0138h
```

```
        .ENDM                           ; Result in SumExt|ResHi|ResLo
;
; Multiply the contents of two registers
;
        MPYS    IROP1,IROP2             ; CALL the MPYS macro
        MOV     &ResLo,R6               ; Fetch LSBs of result
        MOV     &ResHi,R7               ; Fetch MSBs of result
        MOV     &SumExt,R8              ; Fetch Sign of result
;
; Multiply the operands located in a table, R6 points to
;
        MOV     #ResLo,R5               ; Pointer to LSBs of result
        MPYS    @R6+,@R6                ; CALL the MPYS macro
        NOP                             ; NOP: allow completion of MPYS
        MOV     @R5+,R7                 ; Fetch LSBs of result
        MOV     @R5+,R8                 ; Fetch MSBs of result
        MOV     @R5,R9                  ; Fetch sign of result
;
; Macro Definition for the signed multiplication and
; accumulation 16 x 16 bits. The accumulation is made in the
; RAM: MACHi, MACmid and MAClo. If more than 48 bits are used
; for the accumulation, the SumExt register is added to all
; further RAM extensions (here shown for only one).
;
MACS    .MACRO  arg1,arg2               ; Signed MAC 16x16
        MOV     arg1,&0132h             ; Signed MPY is used
        MOV     arg2,&0138h
        ADD     &ResLo,MAClo            ; Add LSBs to result
        ADDC    &ResHi,MACmid           ; Add MSBs to result
        ADDC    &SumExt,MAChi           ; Add SumExt to MSBs
        .ENDM                           ;
;
; Multiply and accumulate signed the contents of two tables
;
        MACS    2(R6),@R5+              ; CALL the MACS macro
        ....                            ; Accumulation is yet made
```

5.1.2.1 Fast Signed Square Function

For some applications, a fast signed square function is necessary (e.g. if the RMS value of an input signal needs to be calculated). Two different solutions are given:

- For 16-bit signed numbers without rounding
- For 14-bit signed numbers with rounding. This version is adapted to the output of the ADC of the MSP430C32x family.

Both use table processing; an offset to a table containing the squared input numbers is built. The given cycles include the move of the operand into R5.

```
; Fast signed squaring for a 16 bit number. The upper 16 bits
; of the result are moved to R5. No rounding is used. 10–12 cycles
;
        MOV.B   DATA+1,R5               ; MSBs of number to R5
        TST.B   R5                      ; Check sign of input number
        JGE     L$1                     ; Positive sign
        INV.B   R5                      ; Negative sign:
        INC.B   R5                      ; Use absolute value
L$1     RLA     R5                      ; Number x 2 (word table address)
        MOV     SQTAB(R5),R5            ; MSBs^2 from table to R5
        ...                             ; Squared value in R5
;
; Squaring for a signed 14 bit value:
; Change the unsigned ADC value (0 to 3FFFh) to a signed value
; by the subtraction of the measured zero point of the system:
;
        MOV     &ADAT,R5                ; ADC result to R5
        SUB     VAL0,R5                 ; Subtract measured 0–point
;
; Fast signed squaring for a 14 bit number. The upper 16 bits of
; the result are added to a buffer SQSUM. Rounding is used.
; If registers are used for the sum: 15–17 cycles
;
        RLA     R5                      ; One bit more resolution
        ADD     #80h,R5                 ; Round to high byte
        BIC     #0FFh,R5                ; Delete lower byte
```

```
        JGE     L$1                     ; Sign?
        INV     R5                      ; Absolute value of ADC result
        INC     R5                      ; Complement + increment
L$1     SWPB    R5                      ; MSBs to LSBs
        RLA.B   R5                      ; Number x 2 (word table address)
        ADD     SQTAB(R5),SQSUM         ; Add MSBs^2 to SQSUM
        ADC     SQSUM+2                 ; Add carry
        ...                             ; Continue
;
; Table with squared values. Length may be adapted to the maximum
; possible input number.
;
SQTAB   .word   ($–SQTAB)*($–SQTAB)/4   ; 0 x 0 = 0
        .word   ($–SQTAB)*($–SQTAB)/4   ; 1 x 1 = 1
        .word   ($–SQTAB)*($–SQTAB)/4   ; 2 x 2 = 4
        ...
        .word   ($–SQTAB)*($–SQTAB)/4   ; 07Fh x 07Fh
        .word   ($–SQTAB)*($–SQTAB)/4   ; 080h x 080h
```

The errors for a single squaring are in the range of 1%. But, if rounding is used and several squared inputs are summed-up, the resulting error gets much smaller. For example, if a sinusoidal input voltage is measured in distances of 15°, then an error of less than 0.24% results.

If the previous method is used for the measurement of RMS values, then for a decision, it usually is not necessary to calculate the square root out of the accumulated squared inputs. It is much faster to use the accumulated value itself .

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 MOV.B instruction used for the loading ensures these bits are cleared. The registers used are shown in the Figure 5–2:

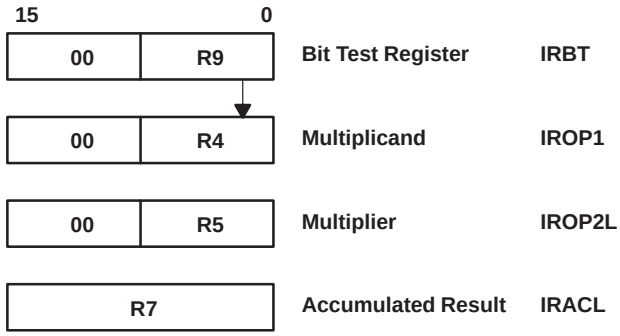

Figure 5–2. 8 x 8 Bit Multiplication – Register use

```
; EXECUTION TIMES FOR REGISTERS CONTENTS (CYCLES) without CALL:
; 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
```

If the hardware multiplier is implemented, the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, a NOP is necessary after the MACRO call to allow the completion of the multiplication. If byte instructions are used for loading the multiplier registers, the high byte is cleared like a CPU register.

```
; Macro Definition for the unsigned multiplication 8 x 8 bits
;
MPYU8    .MACRO   arg1,arg2                ; Unsigned MPY 8x8
         MOV.B    arg1,&0130h              ; 00xx to 0130h
         MOV.B    arg2,&0138h              ; 00yy to 0138h
         .ENDM                             ; Result in ResLo. ResHi = 0
;
; Multiply the contents of two registers (low bytes)
;
         MPYU8    IROP1,IROP2L             ; CALL the MPYU8 macro
         MOV      &ResLo,R6                ; Fetch result (16 bits)
         ...
;
; Macro Definition for the unsigned multiplication and
; accumulation 8 x 8 bits
;
MACU8    .MACRO   arg1,arg2                ; Unsigned MAC 8x8
         MOV.B    arg1,&0134h              ; 00xx
         MOV.B    arg2,&0138h              ; 00yy
         .ENDM                             ; Result in SumExt|ResHi|ResLo
;
; Multiply and accumulate the low bytes of two registers
;
         MACU8    IROP1,IROP2              ; CALL the MACU8 macro
```

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. Therefore, Figure 5–2 is also valid.

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

```
; EXECUTION TIMES FOR REGISTER CONTENTS (CYCLES) without CALL:
; 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
```

If the hardware multiplier is implemented, the previous subroutines can be substituted by MACROs. For source and destination, all seven addressing modes are possible. If register indirect or register indirect with autoincrement addressing modes are used to address the result, a NOP is necessary after the MACRO call to allow the completion of the multiplication. If byte instructions are used for loading the multiplier registers, the high byte is cleared like a CPU register.

```
; Macro Definition for the signed multiplication 8 x 8 bits
;
MPYS8   .MACRO  arg1,arg2               ; Signed MPY 8x8
        MOV.B   arg1,&0132h             ; 00xx
        SXT     &0132h                  ; Extend sign: 00xx or FFxx
        MOV.B   arg2,&0138h             ; 00yy
        SXT     &0138h                  ; Extend sign: 00yy or FFyy
        .ENDM                           ; Result in SumExt|ResHi|ResLo
;
; Multiply the contents of two registers signed (low bytes)
;
        MPYS8   IROP1,IROP2             ; CALL the MPYS8 macro
        MOV     &ResLo,R6               ; Fetch result (16 bits)
        MOV     &ResHi,R7               ; Only sign: 0000 or FFFF
;
; Macro Definition for the signed multiplication and
; accumulation 8 x 8 bits. The accumulation is made in the
; RAM: MACHi, MACmid and MAClo. If more than 48 bits are used
; for the accumulation, the SumExt register is added to all
; further RAM extensions
;
MACS8   .MACRO  arg1,arg2               ; Signed MAC 8x8
        MOV.B   arg1,&0132h             ; MPYS is used
```

```
        SXT     &0132h                  ; Extend sign: 00xx or FFxx
        MOV.B   arg2,&0138h             ; 00yy
        SXT     &0138h                  ; Extend sign
        ADD     &ResLo,MAClo            ; Accumulate LSBs 16 bits
        ADDC    &ResHi,MACmid
        ADDC    &SumExt,MAChi           ; Add SumExt to MSBs
        .ENDM                           ;
;
; Multiply and accumulate signed the contents of two byte tables
;
        MACS8   2(R6),@R5+              ; CALL the MACS8 macro
        ....                            ; Accumulation is yet made
```

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 bits, the carry is then set after return. If a valid result is obtained, the carry is reset after a return. The register usage is shown in Figure 5–3. The subroutine was developed by Mr. Leipold/L&G.

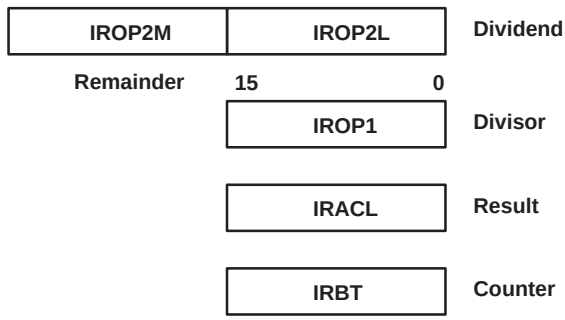

Figure 5–3. Unsigned Division – Register Use

```
; EXECUTION CYCLES FOR REGISTER CONTENTS (without CALL):
;DIVIDE          CYCLES          EXAMPLE
;-------------------------------------------------------------------------------
;               242             0xxxxxxxxh : 00000h = 0FFFFh      C = 1
;               237             03A763E02h : 05A5Ah = 0A5A5h      C = 0
;               240             0FFFE0001h : 0FFFFh = 0FFFFh      C = 0
;
```

```
; USED REGISTERS IROP1, IROP2L, IRACL, IRBT, IROP2M
;
; UNSIGNED 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 LOOP COUNTER
DIV1     CMP      IROP1,IROP2M             ;
         JLO      DIV2
         SUB      IROP1,IROP2M
DIV2     RLC      IRACL
         JC       DIV4                     ; Error: result > 16 bits
         DEC      IRBT                     ; Decrement loop counter
         JZ       DIV3                     ; Is 0: terminate w/o error
         RLA      IROP2L
         RLC      IROP2M
         JNC      DIV1
         SUB      IROP1,IROP2M
         SETC
         JMP      DIV2
DIV3     CLRC                              ; No error, C = 0
DIV4     RET                               ; Error indication in C
```

A 32-bit divided by 32-bit numbers (XDIV) is given in the square root section.

5.1.6 Signed Division 32/16-Bits

The subroutine performs a signed 32-bit by 16-bit division. If the result does not fit into 16 bits, the carry is then set after a return. If a valid result is obtained, the carry is reset after a return. The register IRACM contains the extended sign (0000h or 0FFFFh) of the signed result in IRACL. The register usage is shown in the Figure 5–4:

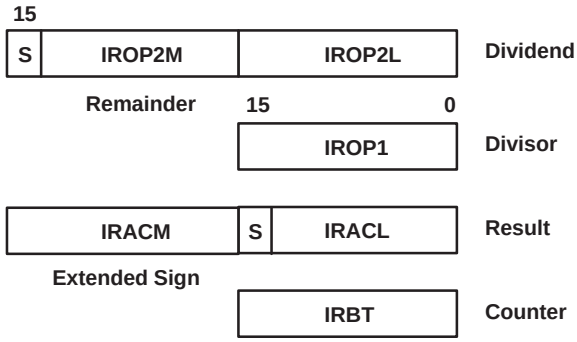

Figure 5–4. Signed Division – Register Use

```
; EXECUTION CYCLES FOR REGISTER CONTENTS (without CALL):
;DIVIDE           CYCLES            EXAMPLE
;-------------------------------------------------------------------------------
; MINIMUM          15               0xxxxxxxxh : 00000h = 0yyyyh      C = 1
;                 268               0E01C3E02h : 05A5Ah = 0A5A5h      C = 0
;                 258               000000001h : 0FFFFh = 0FFFFh      C = 0
;
; USED REGISTERS IROP1, IROP2L, IROP2M, IRACL, IRBT
;
; SIGNED DIVISION SUBROUTINE 32–BIT BY 16–BIT
; IROP2M|IROP2L : IROP1 –> IRACL   REMAINDER IN IROP2M
; RETURN: CARRY = 0: OK    CARRY = 1: QUOTIENT > 16 BITS
;
DIVS     CLR      IRACM                    ; Sign of result
         TST      IROP2M                   ; Check sign of dividend
         JGE      DIVS1
         INV      IROP2M                   ; Is neg.: |dividend|
         INV      IROP2L
         INC      IROP2L
         ADC      IROP2M
         INV      IRACM                    ; Invert sign of result
DIVS1    TST      IROP1                    ; Check sign of divisor. C = 1
         JEQ      DIVSERR                  ; Divisor is 0: error. C = 1
         JGE      DIVS2                    ; Sign is neg.: |divisor|
         INV      IROP1
```

```
         INC     IROP1
         INV     IRACM                   ; Invert sign of result
DIVS2    CALL    #DIVIDE                 ; Call unsigned division
         JC      DIVSERR                 ; C = 1: error
         TST     IRACM                   ; Test sign of result
         JZ      DIVS3
         INV     IRACL                   ; Is neg.: negate result
         INC     IRACL
DIVS3    CLRC                            ; No error occured: C = 0
DIVSERR  RET                             ; Error: C = 1
```

5.1.7 Shift Routines

The results of the previous subroutines (MPY, DIV) accumulated in IRACM/IRACL have to be adapted to different numbers of bits after the decimal point because they are too large to fit into 32 bits. The following subroutines can do this function. If other types of number shifting is necessary, the subroutines can be constructed as shown for the 6-bit shifts (subroutine SHFTRS6). 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
```

5.1.8 Square Root Routines

The square root of a number is often needed in computations. Two different methods are given:

- ❏ A very fast method for 32-bit integer numbers
- ❏ A normal method for 32-bit numbers that can have a fractional part

5.1.8.1 Square Root for 32-Bit Integer Numbers

The square root of a 30-bit integer number is calculated. The result contains 15 correct fractional bits. The subroutine uses the method known from the finding of a square root by hand. This method is much faster than the widely known NEWTONIAN method and only 720 cycles are needed. This subroutine was developed by Jürg Müller Software–Art GmbH/Zurich. The C program code needed is also shown:

```
{ unsigned long y, h;
  int  i;
  h = x;
  x = y = 0;
```

```
  for (i = 0; i < 32; i++)
  {                    // x ist eigentlich 2*x
    x <<= 1; x++;      // 4*x + 1
    if (y < x)
    {
      x -= 2;
    } else
      y -= x;
    x++;
    y <<= 1;           // <y, h> <<= 2
    if (h & Minus) y++;
    h <<= 1;
    y <<= 1;
    if (h & Minus) y++;
    h <<= 1;
  }
  return x;
}
; Square Root of a 32-bit number.
;
x_MSB    .equ   R4
x_LSB    .equ   R5
y_MSB    .equ   R6
y_LSB    .equ   R7
h_MSB    .equ   R8
h_LSB    .equ   R9
i        .equ   R10
;
; Call:    32-bit-Integer in x_MSB, x_LSB
; Result:  32-bit-number  in x_MSB  (16 bit integer part)
;                            x_LSB  (16 bit fraction)
;
; Range for x:      0 <= x    <= 40000000h
; Range for result: 0 <= SQRT <= 8000.0000h
; Max. Error:       0000.0002h
; Calculation Time: 720 cycles (t = 720/MCLK)
```

```
;
; Examples: sqrt (10000000h) = 4000.0000h
;           sqrt     (2710h) = 0000.0064h
;           sqrt        (2h) = 0001.6a09h = 92681 = 1.4142 * 2^16
;
Sqrt     Mov      x_MSB,h_MSB
         Mov      x_LSB,h_LSB
         Clr      x_MSB
         Clr      x_LSB
         Clr      y_MSB
         Clr      y_LSB
         Mov      #32,i
Sqrt10   SetC                      ; x <<= 1; x++;
         Rlc      x_LSB
         Rlc      x_MSB
         Sub      x_LSB,y_LSB      ; y.l –= x.l;
         Subc     x_MSB,y_MSB
         Jhs      Sqrt12           ; if (y.l & Minus)
         Add      x_LSB,y_LSB      ; {
         Addc     x_MSB,y_MSB      ;   y.l += x.l;
         Sub      #2,x_LSB         ;   x.l –= 2; }
Sqrt12   Inc      x_LSB            ; x.l++;
;                                    <y.l, HilfsReg> <<= 2
         Rla      h_LSB            ; <y.l, HilfsReg> <<= 1
         Rlc      h_MSB
         Rlc      y_LSB
         Rlc      y_MSB
         Rla      h_LSB            ; <y.l, HilfsReg> <<= 1
         Rlc      h_MSB
         Rlc      y_LSB
         Rlc      y_MSB
         Dec      i
         Jne      Sqrt10
         Ret
```

5.1.8.2 Square Root for 32-Bit Numbers

The following subroutine uses the Newtonian-approximation method for calculating the square root. The number of iterations depends on the length of the

operand. The subroutine was developed by A. Mühlhofer/TID. The general formula is:

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

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

Where m = 2 (square root)

$$\sqrt{A} = X$$

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

$$X_0 = A/2$$

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

```
Ah       .EQU     R8                      ;High word of A
Al       .EQU     R9                      ;Low word of A
XNh      .EQU     R10                     ;High word of result
XNl      .EQU     R11                     ;Low word of result
; 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    .EQU    $
        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
```

5.1.9 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 (i.e., 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 approach shown 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
        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
```

5.1.10 Random Number Generation

The linear congruential method is used (introduced by D. Lehmer in 1951). The advantages of this method are speed, code simplicity, and ease of use. However, if care is not taken in choosing the multiplier and increment values, the results can quickly 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:

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

Where:

$Rndnum_n$	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)

Many hours of research have 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. The following text 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 returns a random number between 0 and 65,536 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. This is OK if the code is allowed 3 calls to *warm up* before the numbers are considered valid. 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, an initialization routine INIRndnum must be called. Then a subroutine Rndum16 is called to calculate the random numbers as often as needed. The code necessary and the description of the subroutine MPYU can be found in Section 5.1.1, *Unsigned Multiplication 16 x 16-bits*.

```
;
; 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
HW_MPY   .set     0                        ; 1: HW–MPYer on chip
;
; ALLOCATION RANDOM NUMBER IN RAM–ADDRESS 200h
;
         .bss     Rndnum,2,0200h
;
; SUBROUTINE: INITIALIZE RANDOM NUMBER GENERATOR:
; Load the SEED value and produce the 1st random number
;
INIRndnum  .equ   $                        ; Uses Rndnum16
         MOV      #SEED,Rndnum             ; Initialize generator
;
; SUBROUTINE: GENERATES NEXT RANDOM NUMBER
; HW_MPY = 0: 169 cycles
; HW_MPY = 1:  26 cycles
;
Rndnum16  .equ    $
         .if      HW_MPY=0                 ; No MPYer
         MOV      Rndnum,IROP2L            ; Prepare multiplication
         MOV      #MULT,IROP1              ; Prepare multiplication
```

```
        CALL    #MPYU                   ; Call unsigned MPY (5.1.1)
        ADD     #INC,IRACL              ; Add INC to low word of product
;
; Overwrite old random number with low word of new product
;
        MOV     IRACL,Rndnum            ; Result to Rndnum and IRACL
        .else                           ; HW MPYer on chip
        MPYU    Rndnum,#MULT            ; Rndnum x MULT
        MOV     &ResLo,Rndnum           ; Low word of product
        ADD     #INC,Rndnum             ; Add INC to low word
        .endif
        RET                             ; Random number in Rndnum
EXAMPLE: Use of the Random Generator (1st call and succeeding calls).
;
; First call: produce the 1st random number
;
        CALL    #INIRndum               ; Initialize generator
        ....
; Second and all other calls to get the next random number
;
        CALL    #Rndnum16               ; Next random number to register
        ....                            ; IRACL and location Rndnum
```

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

5.1.11 Rules for the Integer Subroutines

Despite the fact that the subroutines shown previously can only handle integer numbers, it is possible to use numbers with fractional parts. It is necessary only 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 good programming practice to indicate in the comment part of the software listing where the decimal point is currently located. The indication can have the following form:

N.M

where

- 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 simple:

- ❏ Addition and subtraction: Positions after the decimal point have to be equal. The position is the same for the result.
- ❏ Multiplication: Positions after the decimal point can be different. The two positions are added to get the result's position after the decimal point.
- ❏ Division: Positions after the decimal point can be different. The two positions are subtracted to get the result's position. (Dividend – divisor)

Table 5–1. Examples for the Virtual Decimal Point

First Operand	Operation	Second Operand	Result
NNN.MMM	+	NNNN.MMM	NNNN.MMM
NNN.M	×	NN.MMM	NNNNN.MMMM
NNN.MM	–	NN.MM	NNN.MM
NNNN.MMMM	:	NN.MMM	NN.M
NNN.M	+	NNNN.M	NNNN.M
NNN.MM	×	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 needs 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 can be used if a smaller number is to be divided by a larger one.

EXAMPLES for the division:

Table 5–2. Rules for the Virtual Decimal Point

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 12.34h loaded      8.8
MOV     R15,IROP1               ; Operand fetched             2.3
CALL    #MPYS                   ; Signed MPY                  10.11
CALL    #SHFTRS3                ; Remove 3 fraction bits      10.8
ADD     #00678h,IRACL           ; Add Constant 6.78h          10.8
ADC     IRACM                   ; Add carry                   10.8
```

5.2 Table Processing

One of the development targets of the MSP430 was the capability of processing tables because software can be written more legible and more functional 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
- Simplifies 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.

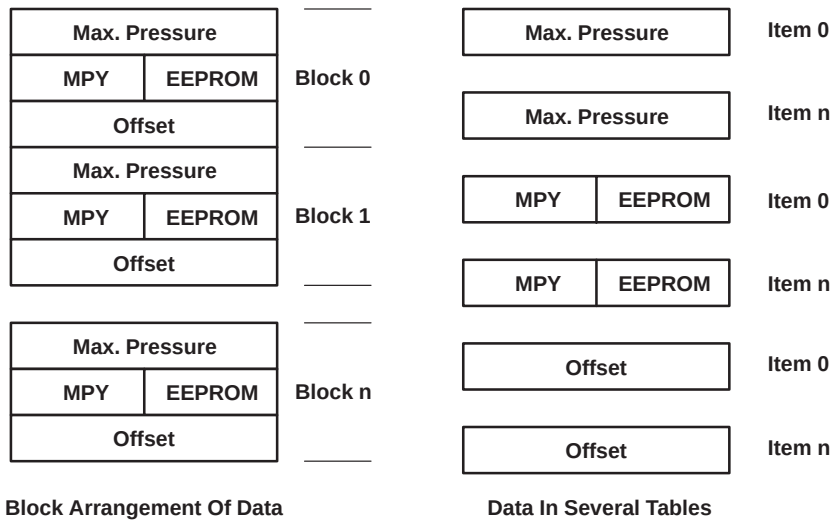

Figure 5–5. Data Arrangement in Tables

EXAMPLE: A table arranged in blocks is shown. Some examples for random access are given. The addressed tables refer to Figure 5–5

```
;Block Arrangement of data
;
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
```

EXAMPLE: A table arranged in several tables is shown. Some examples for random access are given. The addressed tables refer to Figure 5–5

```
; 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: (word offset)
; Examples with identical functions as for the block arrangement
; shown in the example before
;
         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

The output value of a function often 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 5–6 shows a function like the one described. The output value T depends on the input values X and Y.

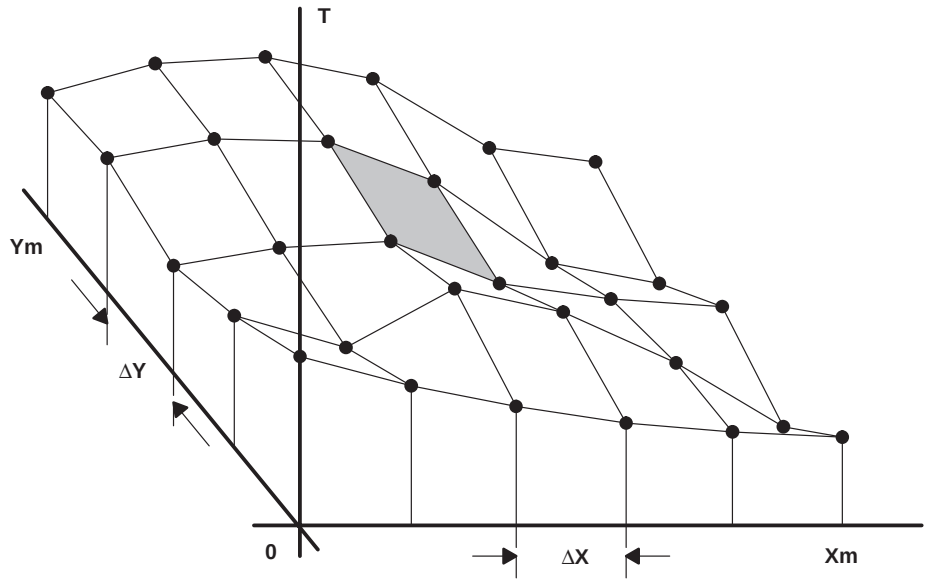

Figure 5–6. Two-Dimensional Function

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.

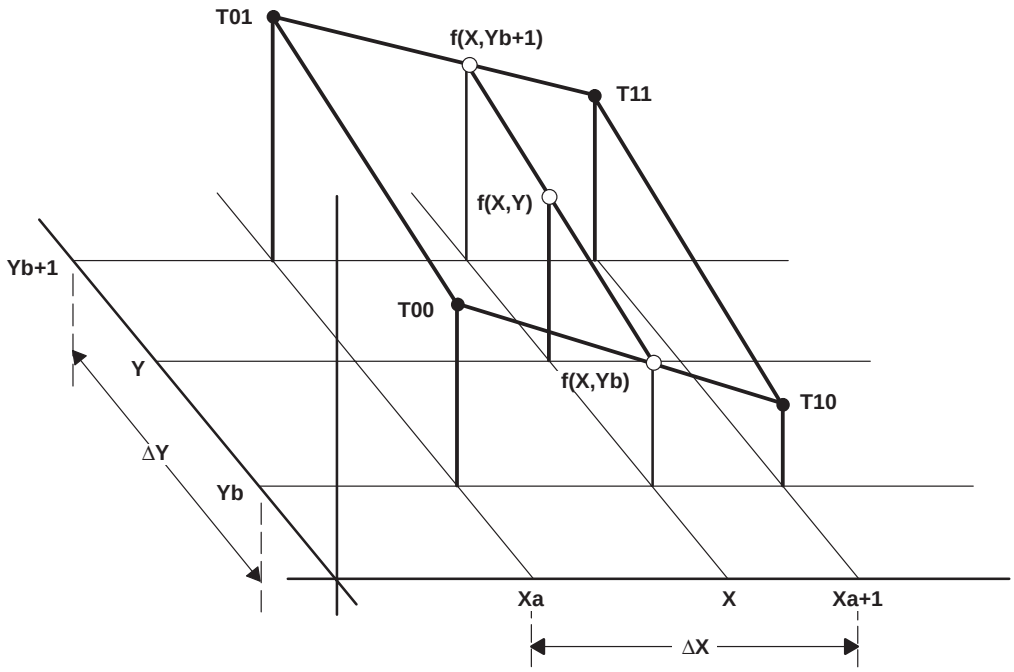

Figure 5–7. Algorithm for Two-Dimensional Tables

The calculation formulas are:

$$f(X, Yb) = \frac{X - Xa}{Xa_{+1} - Xa} \times (T10 - T00) + T00 = \frac{X - Xa}{\Delta X} \times (T10 - T00) + T00$$

$$f(X, Yb+1) = \frac{X - Xa}{\Delta X} \times (T11 - T01) + T01$$

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

These formulas need division. There are two possible ways to avoid 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

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

This adaptation leads to:

$$\frac{f(X,\ Yb)}{\Delta Y} = (X - Xa) \times (T'10 - T'00) + T'00$$

$$\frac{f(X,\ Yb+1)}{\Delta Y} = (X - Xa) \times (T'11 - T'01) + T'01$$

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

The output value f(X,Y) is now calculated 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, the adaptation parts and masks have to be changed.

ITEM	X	Y	COMMENT
Delta	2	4	ΔX and ΔY
Input value format	8.2	7.1	Bits before/after decimal point
Starting value	0	0	X0 resp. Y0
End value	42	56	XM resp. YN
Input value (RAM, reg)	XIN	YIN	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 adaption 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 adaption 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 previous example uses unsigned values for X and Y (the upper left hand table of Figure 5–8 shows this). If X or Y or both are signed values, the structure of the table and its entry point have to be changed. The following examples in Figure 5–8 show how to do that.

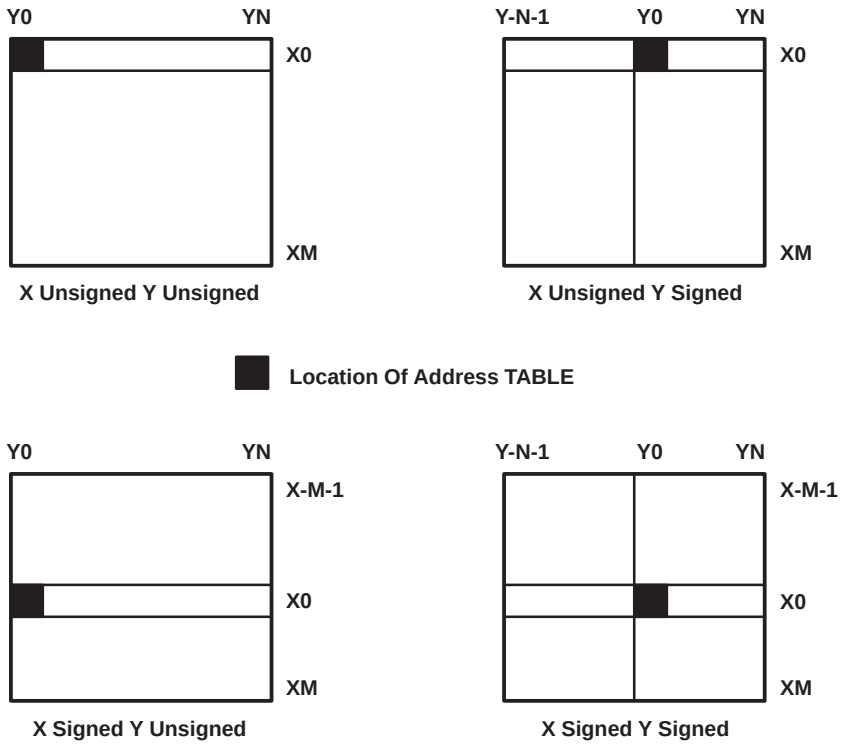

Figure 5–8. Table Configuration for Signed X and Y

The previous 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 T depends on three input variables X, Y and Z, 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 is to compute these figures is to calculate the output values for both two-dimensional tables f(X,Y,Zc) and f(X,Y,Zc+1) with the subroutine TABCAL2. The two results are used for the final calculation:

$$f(X,\ Y,\ Z) \ = \ \frac{Z - Zc}{Zc + 1 - Zc} \times \left(f(X,\ Y,\ Zc + 1) - (f(X,\ Y,\ Zc) \right) + f(X,\ Y,\ Zc)$$

The following figure shows this method. The output values Txxx are calculated for Zc and for Zc+1. Out of these two output values, the final value f(X,Y,Z), is calculated.

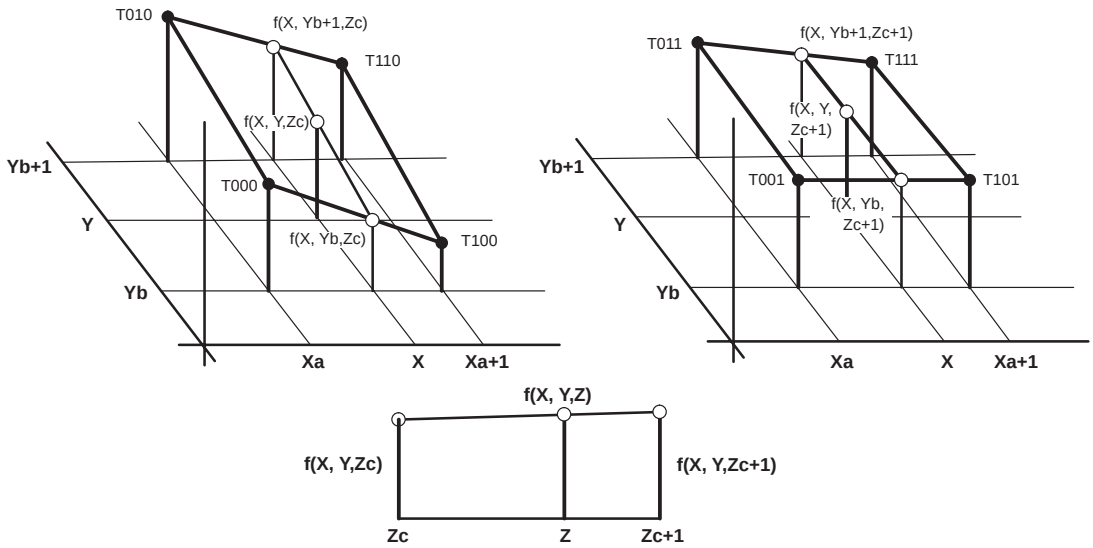

Figure 5–9. Algorithm for a Three-Dimensional Table

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

ITEM	X	Y	Z	COMMENT
Delta	2	4	256	ΔX, ΔY, ΔZ
Input value format	8.2	7.1	0	Bits after decimal point
Starting value	0	0	0	X0, Y0, Z0
End value	42	56	$2^{14}-1$	XM, YN, ZP
Input value (RAM, register)	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–dim. Table
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 adaption 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 unwanted signal components, it may be necessary to average the ADC results. Six 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 divided by two afterwards.
4) Wave Digital Filtering: Complex filter algorithms, which need only small amounts of calculation power, are used for the signal conditioning.
5) Rejection of Extremes: the largest and the smallest samples are rejected from the measured values and the remaining samples are added-up and averaged.
6) Synchronization of the measurements to hum

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

5.3.1 Oversampling

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

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

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
```

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

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 and is deleted at the other end (both items share the same RAM location). To reduce the calculation 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 accumulated sum of N samples. The same rule is valid for 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 behavior is shown in the following figure.

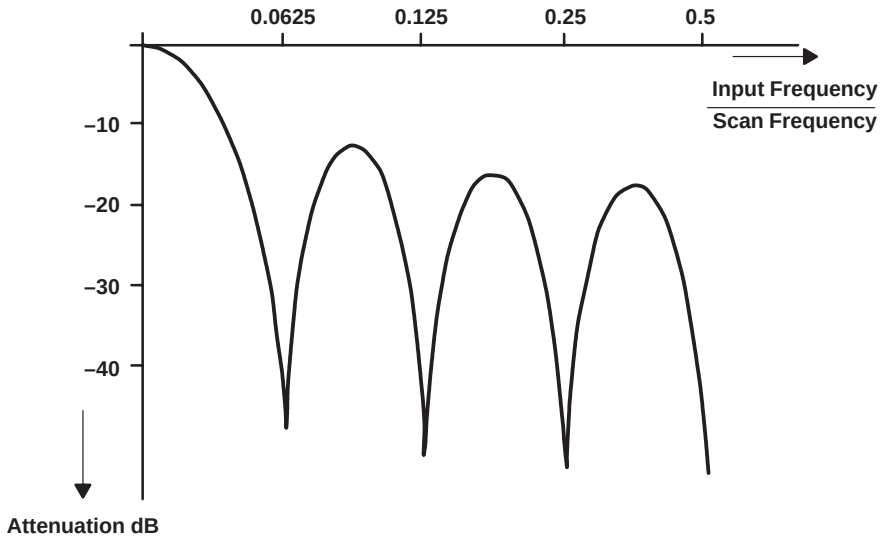

Figure 5–10. Frequency Response of the Continuous Averaging Filter

- Advantages
  - Low current consumption with only one measurement
  - Fast update of buffer
  - Good suppression of certain frequencies (multiples of scan frequency)
  - Low-pass filter characteristic
- Disadvantages
  - RAM allocation: N words are needed for the circular buffer

EXAMPLE: An interrupt driven routine (e.g. from the ADC, which is started by the basic timer) that updates a circular buffer with N items is shown. 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 actual 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 current measurement result are added and then divided by two. This gives every measurement result a certain weight.

Table 5–3. Sample Weight

MEASUREMENT TIME	WEIGHT	COMMENT
t0	0.5	Actual measurement
t0 – Δt	0.25	Last measurement
t0 – 2Δt	0.125	
t0 – 3Δt	0.0625	
t0 – 4Δt	0.03125	
t0 – nΔt	$2^{-(n+1)}$	

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

- Very short code
- Only one RAM word needed

❑ Disadvantages

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

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

```
        .BSS    WSSUM,2                 ; Accumulated weighted sum
;
WSHND   ADD     &ADAT,WSSUM             ; Add actual 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 even under nonlinear operating conditions resulting from overflow and roundoff effects

❑ Low coefficient word length 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 WDF algorithms specialized CAD programs have been designed 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 is given in the "*TEXAS INSTRUMENTS Technical Journal*" November/December 1994.

- Disadvantages
  - Complex algorithm. Support software needed for finding algorithm
- Advantages
  - Low current consumption because only one measurement per time slice needed
  - 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 high due to (N+2) ADC conversions
- Advantages
  - Simple programming
  - Very good suppression of spikes (extremes are rejected)
  - Small amount of RAM needs (4 words)

The following software example adds six ADC samples, subtracts the two extremes, and returns with the sum of the four medium samples. The constant N can 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 divisions needed (right shifts only). Register use is possible 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,1                 ; Counter for N+2
;
SEHND   CLR     SESUM                   ; Initialize buffers
        .IF     N>2
        CLR     SESUM+2
        .ENDIF
        MOV.B   #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
        ADC     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.B   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 ac frequency can help to overcome this problem. Figure 5–11 shows the influence of the ac voltage during the measurement of a single sensor. The necessary number of measurements (here 10 are needed) is split into two equal parts, the second part is measured after exactly one half of the period T_{ac} of the ac frequency. The hum introduced to the two parts is equal but has different signs. Therefore the accumulated influence (the sum) is nearly zero.

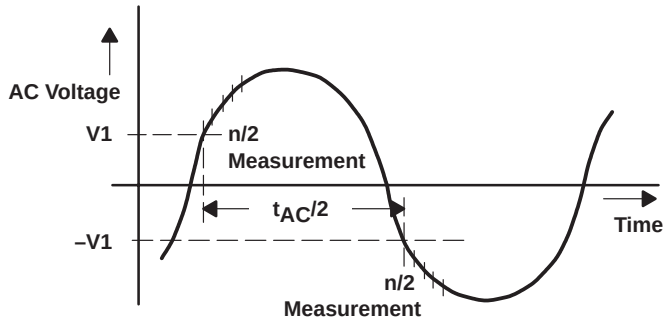

Figure 5–11. Reduction of Hum by Synchronizing to the AC Frequency. Single Measurement

If the basic timer is used for the timing then the following numbers of basic timer interrupts can be used.

Table 5–4. Basic Timer Frequencies for Hum Suppression

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

The formulas to get the above errors are:

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

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

where:

- e_t Maximum time error due to fixed basic timer frequency (in %)
- e_r Maximum remaining influence of the hum (in %) compared to a measurement without hum cancellation
- T_{BT} Period of basic timer frequency ($1/f_{BT}$)
- T_{ac} Period of ac ($1/f_{ac}$)
- k Number of basic timer interrupts to reach $T_{ac}/2$ respective of T_{ac}

If difference measurements are used, the two measurements to be subtracted should be made with a delay of exactly one ac 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.

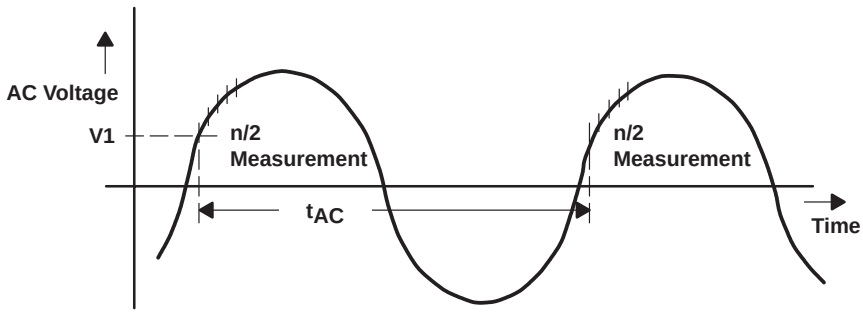

Figure 5–12. Reduction of Hum by Synchronizing to the AC Frequency. Differential Measurement

If the basic timer is used for the timing then the following numbers of basic timer interrupts can be used.

Table 5–5. Basic Timer Frequencies for Hum Suppression

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

The formulas to get the previous results are:

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

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

The software needed for the modification of the Basic Timer frequency without the loss of the exact time base is shown in Section 6.1.1, *Change of the Basic Timer Frequency*.

5.4 Real-Time Applications

Real-time applications for microprocessors are defined often as follows:

- The controlling processor is able, under worst case conditions, to finish the necessary control algorithms before the next sample of the control input arrives.

The architecture of the MSP430 is ideally suited for real time applications due to its system clock generation. The system clock MCLK of the CPU is not generated by a second crystal, which needs a lot of time until it is oscillating with the nominal frequency. But, by the multiplication of the frequency of the 32-kHz crystal that is oscillating continuously.

5.4.1 Active Mode

The active mode shows the fastest response to interrupts because all of the internal clocks are operating at their nominal frequencies. The active mode is recommended when the speed of the MSP430 is the critical factor of an application.

5.4.2 Normal Mode is Low-Power Mode 3 (LPM3)

This mode is used for battery-driven systems where the power consumption plays an overwhelming role. Battery lifetimes over ten years are only possible when the CPU is switched off whenever its processing capability is not needed.

Despite the switched-off CPU, the MSP430 is at the start address of the interrupt handler within eight MCLK cycles; the system clock oscillator is then working at the correct frequency. This means true real-time capability, no delay due to the slow coming-up of the main oscillator crystal (up to 400 ms) is slowing down the system behavior.

See Section 6.5, *The System Clock Generator*, for the details of the programming.

5.4.3 Normal Mode is Low-Power Mode 4 (LPM4)

The low power mode 4 is used if there are relatively long time elapses between two interrupt events. The power consumption goes below 0.1mA if this mode is used All oscillators are switched off and only the RAM and the interrupt hardware are powered.

Despite this inactivity the MSP430 CPU is at the start of the interrupt handler within eight cycles of the programmed DCO tap. See Section 6.5, *The System Clock Generator* for the details of the speed-up of the CPU.

5.4.4 Recommendations for Real-Time Applications

- ❏ Switch on the GIE bit (SR.3) as soon as possible. Within the interrupt handlers, only the tasks that do not allow interruption should be completed first. This allows nested interrupts and avoids the blocking of other interrupts.
- ❏ Interrupt handlers (foreground) should be as short as possible. All calculations should be made in the background part of the program. The communication between these two software parts is made by status bytes. See Section 9.2.5, *Flag Replacement by Status Usage*.
- ❏ Use status bytes and calculated branches.
- ❏ The interrupt capability of the I/O ports makes input polling superfluous. Any change of an input is seen immediately. Use of the ports this way is recommended.
- ❏ Disabling and enabling of the peripheral interrupts during the software run is not recommended. Additional interrupt requests can result from these manipulations. The use of status bytes is recommended instead. They inform the software if an interrupt is valid or not. If not, it is neglected.

5.5 General-Purpose Subroutines

The following, tested software examples can be of help during the software development phase. The examples can not fit into any application, but they can be modified easily 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 a cold start and a warm start. The reason for this 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 done by comparing RAM bytes with known content to their nominal value. These RAM bytes could be identification codes or non-critical test patterns (e.g. A5h, F0h). If the tested RAM locations contain the correct pattern, a spurious signal caused 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 a power loss) 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 Section 6.5, *The System Clock Generator*, and Section 6.6, *The RESET Function*.

```
; 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
;
         MOV      #0300h,SP                 ; Init. Stack Pointer
         CALL     #RAMCLR                   ; Clear complete RAM
         MOV      #0A5F0h,&0200h            ; Insert test word
;
```

```
; System frequency MCLK is set to 2.048MHz
;
        MOV.B   #64–1,&SCFQCTL          ; 64 x 32kHz = 2.048MHz
        MOV.B   #FN_2,&SCFI0            ; DCO current for 2MHz
;
; Waiting loop for the DCO of the FLL to settle: 130ms
;
        CLR     R5                      ; 3 x 65536us = 186ms
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

The RAM is cleared starting at label RAMSTRT up to label RAMEND (inclusive).

```
;
; Definitions for the RAM block (depends on MSP430 type)
;
RAMSTRT  .EQU    0200h                  ; Start of RAM
RAMEND   .EQU    02FEh                  ; Last RAM address (return address)
; Subroutine for the clearing of the RAM block
RAMCLR   CLR     R5                     ; Prepare index register
RCL      CLR.B   RAMSTRT(R5)            ; 1st RAM address
         INC     R5                     ; Next address
         CMP     #RAMEND–RAMSTRT+1,R5   ; 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 previous subroutine can be enlarged to any length of the binary part simply by the adding of registers for the storage of the BCD number (a binary number with n bits needs approximately 1.2 x n bits for BCD format).

If numbers containing fractions have to be converted to BCD, the following algorithm can 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 hexadecimal number 0A8Bh has the binary format MMM.NNN. The decimal value is therefore 337,375. The steps to get the BCD number are:

3) 0A8Bh is to be multiplied by 5^3 or 125_{10} due to its 3 fractional bits. 0A8Bh x 125_{10} =0525DFh

4) 0525DFh has the decimal equivalent 337,375. The correct BCD number with 3 fractional digits.

To convert the previous 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 (999999999)
```

```
;
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 each digit with its decimal value (10^0, 10^1, . . .). To reduce code length, the HORNER scheme is used as follows:

$$R5 = X0 + 10(X1 + 10(X2 + 10X3))$$

```
; The packed BCD number contained in R4 is converted to a 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                   ; XN+10XN+1
         CLR     R6
         DEC     R8                      ; THROUGH ?
         JZ      END                     ; YES
```

```
MPY10   RLA     R5                      ; NO, MULTIPLICATION WITH 10
        MOV     R5,R7                   ; DOUBLED VALUE
        RLA     R5
        RLA     R5
        ADD     R7,R5                   ; VALUE X 8
        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, scanning is necessary. The scanning outputs can be I/O-ports and unused segment outputs, On. The scanning input can be I/O ports and analog inputs, An, switched to the function of digital inputs. If I/O ports are used for inputs, 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 desired signal edges. If one of the desired input signal changes occurs, an interrupt is given and wake-up takes place.

Figure 5–13 shows a keyboard with 16 keys.

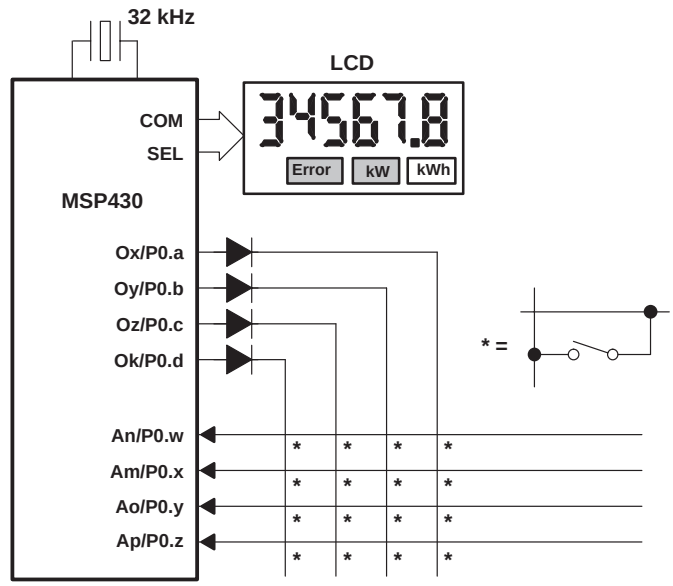

Figure 5–13. Keyboard Connection to MSP430

Figure 5–14 shows some possibilities for connecting external signals to the MSP430:

- ❏ The first row contains keys. The decoupling diode in the row selection line prevents a pressed key from shorten other signals. If more than one key can be activated simultaneously, then all keys need to have a decoupling diode.
- ❏ The second row contains diodes. This is a simple way to identify the version used to a system.
- ❏ The third row selects digital signals coming from peripherals with outputs that can be switched to high-impedance mode.
- ❏ The fourth row uses an analog switch to connect digital signals to the MSP430. The output of a CMOS gate and the output of a comparator are shown.

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 ms to 100 ms, and the information read is 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.

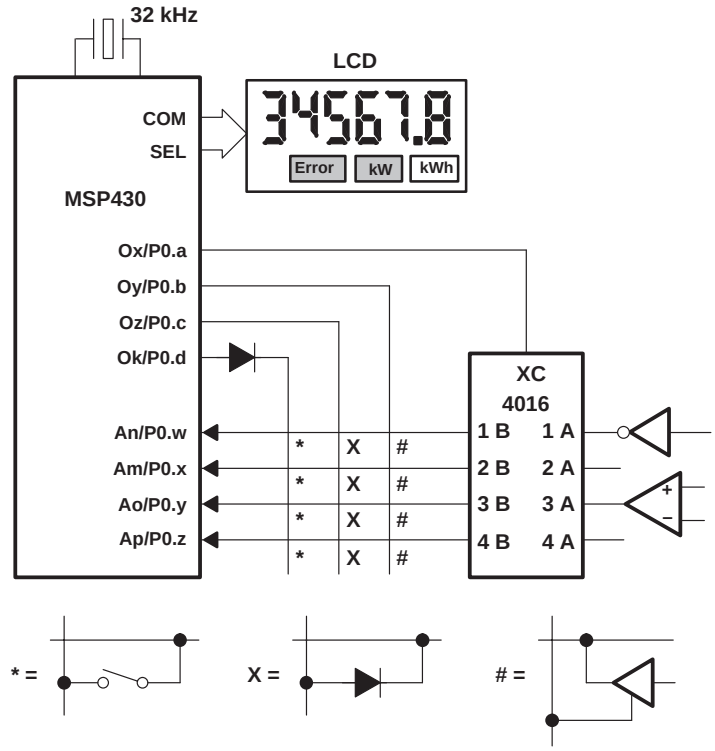

Figure 5–14. Connection of Different Input Signals

5.5.6 Temperature Calculations for Sensors

Several sensors can be connected to the MSP430. Chapter 2, *The Analog-to-Digital Converters*, describes the different possible ways of doing this. Independent of the ADC or sensor type used, a binary number N is finally delivered from the ADC that represents the measured value K:

$$K = f(N)$$

where:

K	Measured value (temperature, pressure etc.)
N	Result of ADC

The function f(N) is normally non-linear for sensors, and, therefore, a calculation is needed to get the measured value K. The linearization of sensors by resistors is described in Section 4.7.1, *Sensor Connection and Linearization*.

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

- ❏ Table processing
- ❏ 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. Figure 5–15 shows such a non-linear sensor characteristic.

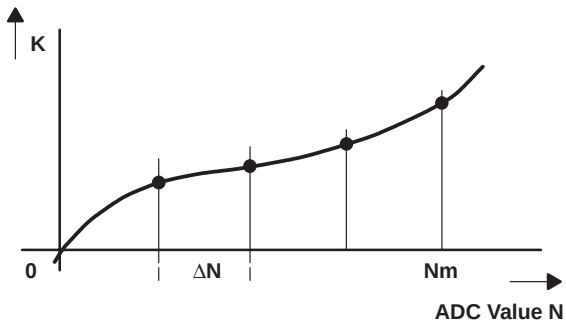

Figure 5–15. Nonlinear Function K = f(N)

Steps for the development of a sensor table:

1) Definition of the external circuitry used at the ADC inputs (see Chapter 2, *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 previous 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 ; K0, K1, ...KM
;
TABCAL1  MOV      &ADAT,IROP1               ; ADC result N 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        ; KA from table
        MOV     TABLE+2(IROP1),R14      ; KA+1 from table
;
; K = ((XIN–KA)/deltaN) x (KA+1 – KA) + KA
;
        SUB     R15,XIN                 ; XIN – KA
        MOV     R14,IROP2L              ; KA+1
        SUB     R15,IROP2L              ; KA+1 – KA)
        MOV     XIN,IROP1               ; XIN – KA
        CALL    #MPYS                   ; (XIN – KA) x (KA+1 – KA)
        CALL    #SHFTRS6                ; /deltaN
        CALL    #SHFTRS1                ; deltaN = 2^7
        ADD     R15,IRACL               ; + KA, result in IRACL
        RET
```

5.5.6.2 Algorithms for Sensor Calculations

If the sensor characteristic can be described by a function, K = f(N), then no table processing is necessary. The value K can be calculated out of the ADC result N. The coefficients a_n and b_n can be found with PC computer software (e.g. MATHCAD), with formulas by hand, or by the MSP430 itself. These categories are for example:

Linear Equation

$$K = a_1 \times N + a_0$$

Quadratic Equation

$$K = a_2 \times N^2 + a_1 \times N + a_0$$

Cubic Equation

$$K = a_3 \times N^3 + a_2 \times N^2 + a_1 \times N + a_0$$

Root Equation

$$K = a_0 \pm \sqrt{b_1 \times N + b_0}$$

Hyperbolic Equation

$$K = \frac{b_1}{N + b_0} + a_0$$

Steps for the development of a sensor algorithm:

1) Definition of the hardware circuitry used at the ADC inputs (See Chapter 2, *The ADC*, for the different possibilities)
2) Definition of the format of the algorithm (floating point: 2- or 3-word package, integer software: bits after decimal point M.N)
3) Definition of a value for K to be measured (temperature, pressure etc.)
4) Calculation of the nominal sensor resistance for the previous chosen value of K
5) Calculation of the voltage at the analog input Ax for this sensor resistance (See Chapter 2, *The ADC*, for the formulas used with the different circuits)
6) Calculation of the ADC result N for this input voltage at analog input Ax
7) Repetition of steps 3 to 6 depending on the algorithm used: twice for linear equations, three times for quadratic, hyperbolic and root equations, four times for cubic equations.
8) Decision of the sensor characteristic: look for best suited equation.
9) Calculation of the coefficients a_n and b_n out of the calculated pairs of values K_n and the ADC result N_n. See Section 5.5.6.3*, Coefficient Calculation for the Equations*.

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

$$K = a_2 \times N^2 + a_1 \times N + a_0$$

with N representing the ADC result. The corrected ADC result (see the previous text) is stored in XIN. The three terms are stored in the ROM locations A2, A1 and A0.

```
;
A2       .WORD    07FE3h                    ; Quadratic term              +–0.14
A1       .WORD    00346h                    ; Linear term                 +–0.14
A0       .WORD    01234h                    ; Constant term               +–15.0
;
QUADR    MOV      XIN,IROP1         ; Corrected ADC result               14.0
         MOV      A2,IROP2L         ; Factor A2                          +–0.14
         CALL     #MPY                      ; XIN x A2                   14.14
         ADD      A1,IRACL          ; (XIN x A2) + A1                    +–0.14
         ADC      IRACM                     ; Carry to HI reg
         CALL     #SHFTL3                   ; To IRACM                   14.1
         MOV      IRACM,IROP2L      ; (XIN x A2) + A1 –> IROP2L          14.1
         CALL     #MPYS                     ; (XIN x A2) + A1) x XIN     28.1
         CALL     #SHFTL2                   ; Result to IRACM            15.0
         ADD      A0,IRACM          ; Add A0                             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 IRACM and IRACL. The terms A2, A1, A0 can 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 Section 5.6, *The Floating Point Package*, for details.

5.5.6.3 Coefficient Calculation for the Equations

With two pairs (linear equation), three pairs (quadratic, hyperbolic and root equations), or four pairs (cubic equations) of K_n and N_n, the coefficients a_n and b_n can be calculated. The formulas are shown in the following.

where:

- K_n Calculation result for the ADC result N_n (e.g. temperature, pressure)
- N_n Input value for the calculation e.g. ADC result
- a_n Coefficient for the N^n value of the polynoms
- b_n Coefficients for the hyperbolic and root equations

Linear equation:

$$K = a_1 \times N + a_0$$

$$a_1 = \frac{K_2 - K_1}{N_2 - N_1} \qquad a_0 = \frac{K_1 \times N_2 - K_2 \times N_1}{N_2 - N_1}$$

Quadratic Equation:

$$K = a_2 \times N^2 + a_1 \times N + a_0$$

$$a_2 = \frac{(K_2 - K_1) - a_1 \times (N_2 - N_1)}{N_2^2 - N_1^2} \qquad a_0 = K_1 - a_2 \times N_1^2 - a_1 \times N_1$$

$$a_1 = \frac{(K_2 - K_1) \times (N_3^2 - N_2^2) - (K_3 - K_2) \times (N_2^2 - N_1^2)}{(N_2 - N_1) \times (N_3^2 - N_2^2) - (N_3 - N_2) \times (N_2^2 - N_1^2)}$$

Cubic Equation:

$$K = a_3 \times N^3 + a_2 \times N^2 + a_1 \times N + a_0$$

The equations for the four coefficients a_n are too complex. Shift the calculation task to the calibration PC and use MATHCAD or something similar to it.

Root Equation:

$$K = a_0 \pm \sqrt{b_1 \times N + b_0}$$

$$a_0 = 0.5 \times \frac{(N_2 - N_1) \times (K_3^2 - K_1^2) - (N_3 - N_1) \times (K_2^2 - K_1^2)}{(N_2 - N_1) \times (K_3 - K_1) - (N_3 - N_1) \times (K_2 - K_1)}$$

$$b_1 = \frac{(K_2^2 - K_1^2) - 2a_0 \times (K_2 - K_1)}{N_2 - N_1} \qquad b_0 = (K_1 - a_0)^2 - b_1 \times N_1$$

Hyperbolic Equation:

$$K = \frac{b_1}{N + b_0} + a_0$$

$$b_0 = \frac{(N_3 \times K_3 - N_1 K_1) \times (N_2 - N_1) - (N_2 \times K_2 - N_1 \times K_1) \times (N_3 - N_1)}{(N_3 - N_1) \times (K_2 - K_1) - (N_2 - N_1) \times (K_3 - K_1)}$$

$$b_1 = (K_1 - a_0) \times (N_1 + b_0)$$

$$a_0 = \frac{(N_3 \times K_3 - N_1 K_1) + b_0 \times (K_3 - K_1)}{N_3 - N_1}$$

EXAMPLE: the sensor used has a quadratic characteristic $R = d_2 \times K^2 + d_1 \times K + d_0$. This means, the value K is described best by the root equation (inverse to the quadratic characteristic of the sensor):

$$K = a_0 \pm \sqrt{b_1 \times N + b_0}$$

where the sensor resistance R is replaced by the ADC result N. During the calibration with the values for K_n 0, 200, and 400, the following ADC Results, N_n, were measured:

Calc.Value K_n (°C, hP, V)		ADC Value N_n	
K_1	0	N_1	4196
K_2	200	N_2	4430
K_3	400	N_3	4652

With the previous numbers the coefficients a_0, b_0 and b_1 can be calculated:

$$a_0 = 0.5 \times \frac{(4430 - 4196) \times (400^2 - 0^2) - (4652 - 4196) \times (200^2 - 0^2)}{(4430 - 4196) \times (400 - 0) - (4652 - 4196) \times (200 - 0)} = 4000$$

$$b_1 = \frac{(200^2 - 0^2) - 2 \times 4000 \times (200 - 0)}{4430 - 4196} = -6666.6667$$

$$b_0 = (0 - 4000)^2 - (-6666.6667) \times 4196 = 43.97371E6$$

With the above calculated coefficients, the negative root value is to be used:

$$K = a_0 - \sqrt{b_1 \times N + b_0}$$

5.5.7 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 also. The next two sections show how to implement the algorithms of this standard and how the encrypted data can be sent by the MSP430.

5.5.7.1 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 64 totally-scrambled 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* from ANSI):

- ❏ Initial Permutation: 64-bits plain text to 64-bit encrypted text via table IP
- ❏ 32-bit to 48-bit permutation via table E
- ❏ 48-bit to 32-bit permutation via tables S1 to S8
- ❏ 32-bit to 32-bit permutation via table P
- ❏ 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:

Where:

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)

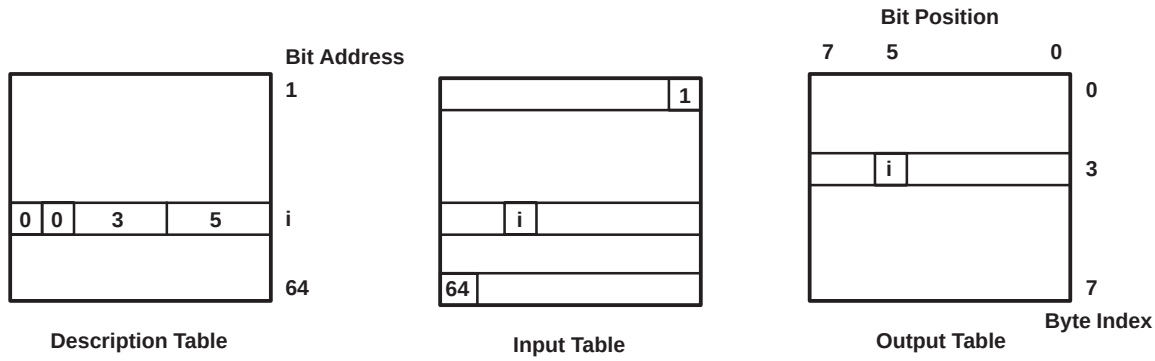

Figure 5–16. DES Encryption Subroutine

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

- The initialization of the subroutine PERMUT decides which permutation takes place. The address of the actual description table is written to pointer register PTPOI.
- Permutations are always made from table PTIN (input table) to table PTOUT (output table).
- Only 1s are processed during the permutation. This saves 50% of processing time. The output buffer is therefore cleared initially by the PERMUT subroutine.

- ❏ 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 ORs (XOR). This means that the number of permutations multiplied with the number of cycles per bit gives an estimation of the processing time needed. 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	16×48
3)	48-bit to 32-bit permutation	16×32
4)	32-bit to 32-bit permutation	16×32
5)	Inverse initial Permutation:	64
6)	Key permutations choice 1	56
7)	Key permutations choice 2	16×48
	Sum of permutations	2744

Number of cycles typically $(2744 \times 43 \times 0.5)$ 58996 cycles 32 ones in block

maximum (2744×43) 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 bytes.

5.5.7.2 Output Sequence for 19.2-kHz Biphase Space Code

The encrypted information is normally output with a Biphase Code. Figure 5–17 shows such a modulation. At the beginning of a bit, a level change occurs. A zero bit has an additional level change in the middle of the bit, a one bit has the same information during the whole bit.

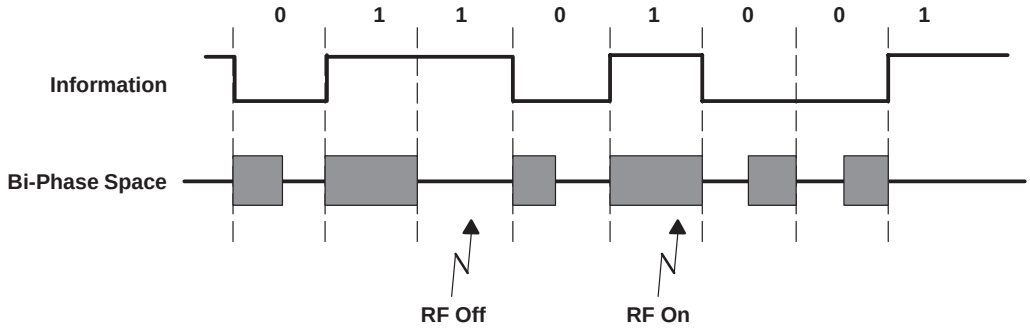

Figure 5–17. Biphase Space Code

The output sequence is written for P0.4 (as shown in Section 4.4, *Heat Allocation Counter*). This means that no constant of the constant generator can be used. If P0.0, P0.1, P0.2, or P0.3 are used, the instructions that address the ports are one cycle shorter and the delay subroutines have to be adapted.

The following output sequence is written with counted instructions per bit due to the normal use of batteries (Vcc = 3V) for these applications. This means the maximum MCLK is 2.2 MHz. If the supply voltage is 5 V, MCLK frequencies up to 3.3 MHz are possible. These high operating frequencies allow the use of interrupt driven output sequences.

The interrupt approach makes strict real time programming necessary. Any interrupt handler must be interruptible (EINT is one of the first instructions of any interrupt handler). Hardware examples are shown in Section 4.8, *RF Readout*.

```
; 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. MCLK = 1MHz
;
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                     ; Register allocation
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 HI              16
;
; 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 Subroutines: 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 RET
;
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
```

5.5.8 Status/Input Matrix

A few subroutines are described that handle the inputs coming from keys, signals etc. They check, if the inputs are valid, for the given status of the program.

5.5.8.1 Matrix with Few Valid Combinations

The following subroutine checks if for a given program status an input (e.g., via the keyboard) is valid or not; and, if valid, which response is necessary. This solution is recommended if only few valid combinations exist out of a large possible number (see Figure 5–18).

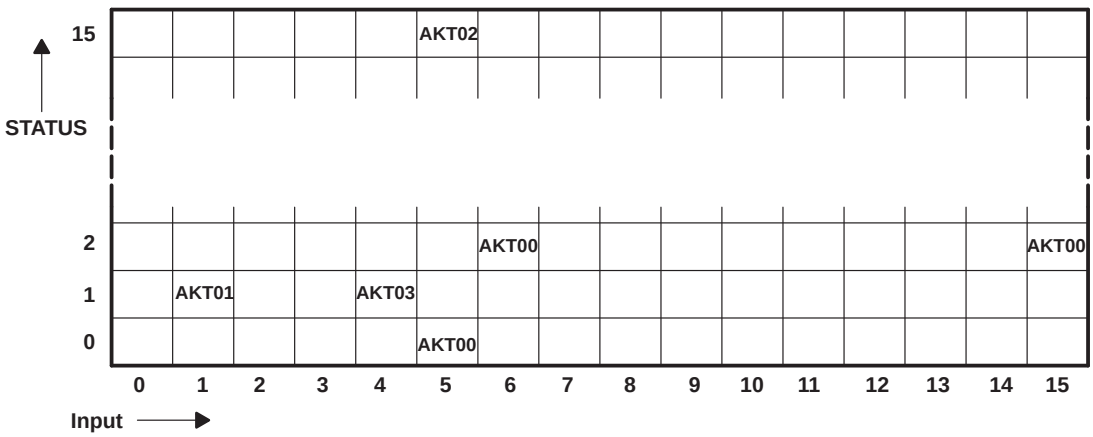

Figure 5–18. Matrix for Few Valid Combinations

- ❏ Call
  - ■ Input number in R5
  - ■ Status in R4
- ❏ Return
  - ■ R4 = 0: Input not valid (not included in the table)
  - ■ R4 # 0: task number in R4

```
CALL     MOV      STATUS,R4                ; Program status to R4
         MOV      INPUT,R5                 ; New input to R5
         CALL     #STIMTRX                 ; Check validity
         ...                               ; R4 contains info
;
STIMTRX  MOV.B    STTAB(R4),R4             ; Start of table for status
         ADD      #STTAB,R4                ; Table address to R4
L$3      CMP.B    @R4+,R5                  ; New input included in table?
         JEQ      L$2                      ; Yes, output it
         INC      R4                       ; No, skip response byte
         TST.B    0(R4)                    ; End of status table? (0)
         JNE      L$3                      ; No, try next input
         CLR      R4                       ; Yes, end of table reached
         RET                               ; Input invalid, return with R4 = 0
L$2      MOV.B    @R4,R4                   ; Input valid, return with task
         RET                               ; number in R4
;
; Table with relative start addresses for the status tables
;
STTAB    .BYTE    ST0–STTAB,ST1–STTAB,ST2–STTAB,...ST15–STTAB
;
; Status tables: valid inputs,response,..,0  (up to 15 inputs)
;
ST0      .BYTE    IN5,AKT00,0              ; Status 0 table
ST1      .BYTE    IN1,AKT01,IN4,AKT03,0    ; Status 1 table
ST2      .BYTE    IN15,AKT00,IN6,AKT06,0   ; Status 2 table
         ....                              ; Status 3 to 14
ST15     .BYTE    IN5,AKT02,0              ; Status 15 table
```

With a small change, the task to do is also executed within the subroutine:

```
CALL     MOV      STATUS,R4                ; Program status to R4
         MOV      INPUT,R5                 ; New input to R5
         CALL     #STIMTRX                 ; Check validity and execute task
         ...                               ; R4 = 0: invalid input
;
STIMTRX  MOV.B    STTAB(R4),R4             ; Start of table for status
```

```
        ADD     #STTAB,R4               ; to R4
L$3     CMP.B   @R4+,R5                 ; New input included?
        JEQ     L$2                     ; Yes, proceed
        INC     R4                      ; No, skip task address
        TST.B   0(R4)                   ; End of status table? (0)
        JNE     L$3                     ; No, try next input
        CLR     R4                      ; End of table reached
        RET                             ; Input invalid, return with R4 = 0
L$2     MOV.B   @R4,R4                  ; Input valid, go to task
        ADD     R4,PC                   ; offset to AKT00 in R4
AKT00   ...                             ; Task 00
        RET
AKT01   ...                             ; Task 01
        RET
AKT06   ...                             ; Task 06
        RET
AKT03   ...                             ; Task 03
        RET
;
; Table with relative start addresses for the status tables
;
STTAB   .BYTE   ST1–STTAB,ST2–STTAB,...ST15–STTAB
;
; Status tables: valid inputs,task–table_start,0
;
ST1     .BYTE   IN5,AKT00–AKT00,0       ; Status 1 table
ST2     .BYTE   IN1,AKT01–AKT00,IN4, AKT03–AKT00,0
ST3     .BYTE   IN15,AKT00–AKT00,IN6,AKT06–AKT00,0
        ....                            ; Status 4 to 14
ST15    .BYTE   IN5,AKT02–AKT00,0       ; Status 15 table
```

5.5.8.2 Matrix With Valid Combinations Only

The following subroutine executes the tasks belonging to the 16 possible STATUS/INPUT combinations. The handler start addresses must be within 254 bytes relative to the label STTAB. The number of combinations can be enlarged to any value.

- Call
  - Input number in RAM byte INPUT (four possibilities 0 to 3)
  - Program status in RAM byte STATUS (four possibilities 0 to 3)
- Return
  - No information returned

```
CALL     CALL     #STIMTRX                  ; Execute task for input
;
STIMTRX  MOV.B    STATUS,R4                 ; Program status 00xx
         MOV.B    INPUT,R5                  ; Input (key, Intrpt,) 0yy
         RLA      R4                        ; STATUS x 4: 00xx –> 0xx0
         RLA      R4                        ; 0xx0 –> 0xx00
         ADD      R5,R4                     ; Build table offset: 0xxyy
         MOV.B    STTAB(R4),R4              ; Offset of Start of table
         ADD      R4,PC                     ; Handler start to PC
STTAB    .BYTE    AKT00–STTAB               ; Action STATUS = 0, INPUT = 0
         .BYTE    AKT01–STTAB               ; Action STATUS = 0, INPUT = 1
         .BYTE    AKT02–STTAB,AKT03–STTAB,AKT04–STTAB,AKT05–STTAB
         ...
         .BYTE    AKT12–STTAB,AKT13–STTAB,AKT14–STTAB,AKT15–STTAB
;
; Action handlers for the 16 STATUS/INPUT xy combinations
;
AKT00    ...                               ; Handler for task 0,0
         RET
AKT01    ...                               ; Handler for task 0,1
         RET
         ...                               ; Tasks 02 to 31
AKT32    ...                               ; Handler for task 3,2
         RET
AKT33    ...                               ; Handler for task 3,3
         RET
```

The next subroutine also executes the tasks belonging to the 16 possible STATUS/INPUT combinations. Here the handler start addresses can be located in the complete 64K-byte address space. The number of STATUS/INPUT combinations can be enlarged to any value.

- Call
  - Input number in RAM byte INPUT (five possibilities 0 to 3)
  - Program status in RAM byte STATUS (four possibilities 0 to 3)
- Return
  - No information returned

```
CALL     CALL     #STIMTRX                ; Execute task for input
;
STIMTRX  MOV      STATUS,R4               ; Program status 00xx
         MOV      INPUT,R5                ; Input (key, Intrpt) 0yy
         RLA      R4                      ; 00xx –> 0xx0
         RLA      R4                      ; 0xx0 –> 0xx00
         ADD      R5,R4                   ; 0xxyy table offset
         RLA      R4                      ; To word addresses
         MOV      STTAB(R4),PC            ; Offset of Start of table
;
STTAB    .WORD    AKT00                   ; Action STATUS = 0, INPUT = 0
         .WORD    AKT01                   ; Action STATUS = 0, INPUT = 1
         ...                              ; Action handlers AKT02 to AKT32
         .WORD    AKT33                   ; Action STATUS = 3, INPUT = 3
```

5.6 The Floating-Point Package

Floating-point arithmetic is necessary if the range of the numbers used 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 many subroutines and applications, like the square root, are included in the software example chapter.

The floating-point package makes use of the RISC architecture of the MSP430 family. During the initialization of the subroutines, the arguments are copied into registers R4 to R15 and the complete calculations take place there. After the completion of the calculation, the result is placed on top of the stack.

5.6.1 General

The floating-point package (FPP) consists of 3 files supporting the .FLOAT format (32 bits) and the .DOUBLE format (48 bits):

- ❏ FPPDEF4.ASM: the definitions used with the other two files
- ❏ FPP04.ASM: the basic arithmetic operations add, subtract, multiply, divide and compare
- ❏ CNV04.ASM: the conversions from and to the binary and the BCD format

Notes:

The file FPP04.ASM can be used without the conversions, but the conversion subroutines CNV04.ASM need the FPP04.ASM file. This is due to the common completion parts contained in FPP04.ASM.

The explanations given for the FPP version 04 are valid also for the FPP version 03. The only difference between the two versions is the hardware multiplier that is included in the version 04. Other differences are mentioned in the ajoining sections. FPP4 is upward compatible to FPP3.

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 assembly time variable HW_MPY defines if the hardware multiplier is used or not during the multiplication subroutine:

HW_MPY = 0:	No use, the multiplication is made by a software loop
HW_MPY = 1:	The 16 × 16 bit hardware multiplier is used

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 conversions
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 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 the stack pointer, 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) points to when the subroutine is called. The address contained in register RPRES is used only for the addressing of Argument 1.

The results of the basic arithmetic operations (add, subtract, multiply and divide) are also contained in the RAM address @SP or 0(SP), and the registers RESULT_MID and RESULT_LSB after the return from these subroutines. Using these registers for data transfers saves program space and execution time.

Between FPP subroutine calls, all registers can be used freely. The result of the last operation is stored on the stack. See previous note.

If, at an intermediate stage of the basic arithmetic operations, a renormalization shift of one or more bit positions to the left is required, then valid bits are available for the shift into the low-order bit positions during renormalization. These bits are named guard bits. With some other FPPs having no guard bits, zeroes are shifted in, which means a loss of accuracy.

The registers that hold the pointers are called:

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

The following choices can be used to address the two operands:

1) $RESULT_{NEW}$ = @(RPRES) <operator> @(RPARG)
2) $RESULT_{NEW}$ = @(RPRES) <operator> $RESULT_{OLD}$
3) $RESULT_{NEW}$ = $RESULT_{OLD}$ <operator> @(RPARG)

- To 1: RPRES and RPARG both point to the arguments for the next operation. This is the default and is independent of the address pointed to either a new argument or a result. The result of the operation is written to the address in the SP.
- 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 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:

Formulas 2 and 3 are not equal, they allow use of the result on the TOS in two ways with division and subtraction. No time is needed and no ROM-consuming moves are necessary if the result is the divisor or the subtrahend for the next operation.

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, if it is used as an operand.

3) The result is located on the top of the stack (TOS), the stack pointer, 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*, *Denormalized Number*, or *Infinity* exist. In this way, the MSP430 FPP has a larger range than other FPPs and allows a higher speed with less memory used. This is because no unnecessary checks for invalid numbers are made.

5) Every floating point operation outputs a valid floating point number that can be used immediately by other operations.

6) If a result is too large (exceeds the number range), the signed maximum number is output. An error indication is given in this case (see Table 5–6, *Error Indication*).

7) The CPU registers used are modified within the FPP subroutines, but do not contain valid data after a return from the subroutine. This means, they can be used freely between the FPP subroutines for other purposes.

5.6.3 The Basic Arithmetic Operations

The FPP is designed for fast and memory saving calculations. So register instructions are ideally suited for this operation. A common save and recall routine for the registers used at the beginning and the end of an arithmetical expression is an additional option. The subroutines FLT_SAV and FLT_REC should be applied as shown in the following examples.

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.

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

- Errors: Normal error handling. See Section 5.6.3.5, *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 points to the same location as it did before the subroutine call.

  The stack pointer, RPRES, and RPARG point to the MSBs of the floating point sum. If an error occurred (N = 1 after return), the result is the number that best represents the correct result: 0 resp. $\pm 3.4 \times 10^{38}$.
- EXAMPLE: The floating point number (.FLOAT format) contained in the ROM starting at address NUMBER is added to the RAM location pointed to by R5. The result is written to the RAM addresses RES and RES+2 (LSBs).

```
DOUBLE   .EQU     0
         MOV      R5,RPRES          ; Address of Argument 1 in R5
         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 register RPARG is subtracted from the floating point number pointed to by register RPRES. With 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 and is subtracted from the result.

RESULT on TOS = @(RPRES) – @(RPARG)

- Errors: Normal error handling. See Section 5.6.3.5, *Error Handling*, for a detailed description.
- Output: The floating point difference of the two arguments is placed on top of the stack. The stack pointer points to the same location as it did before the subroutine call.

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

- EXAMPLE: The floating point number (.DOUBLE format) contained in the ROM locations starting at address NUMBER is subtracted from RAM locations pointed to by R5. The result is written to the RAM addresses pointed to by R5.

```
DOUBLE   .EQU    1
         MOV     R5,RPRES          ; Address of Argument1 in R5
         MOV     #NUMBER,RPARG     ; Address of Argument2
         CALL    #FLT_SUB          ; ((R5)) - (NUMBER) -> TOS
         JN      ERR_HND           ; Error occurred, check reason
         MOV     @RPRES+,0(R5)     ; Store FPP result (MSBs)
         MOV     @RPRES+,2(R5)
         MOV     @RPRES,4(R5)      ; 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 MSB of the mantissa is set, only the LSB–1 is added to the result. The multiplication subroutine returns the same result regardless of whether the hardware multiplier is used (HW_MPY = 1) or not (HW_MPY = 0).

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

- Errors: Normal error handling. See Section 5.6.3.5, *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 points to the same location as it did before the subroutine call.

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

- Special Cases: $0 \times 0 = 0$ $0 \times X = 0$ $X \times 0 = 0$

❑ 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. With 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 and is added to the result.

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

❑ Errors: Normal error handling. See Section 5.6.3.5, *Error Handling,* for a detailed description. Division by zero is indicated also.

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

The stack pointer, RPRES, and RPARG point to the MSBs of the floating point quotient. If an error occurred (N = 1 after return), the result is the number that best represents the correct result. 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 R5. The result is written to the RAM addresses pointed to by R5.

```
DOUBLE   .EQU     1
         MOV      R5,RPARG          ; Address of dividend
         MOV      #NUMBER,RPRES     ; Address of divisor
         CALL     #FLT_DIV          ; (NUMBER) / ((R5)) –> TOS
         JN       ERR_HND           ; Error occurred, check reason
         MOV      @RPRES+,0(R5)     ; Store FPP result (MSBs)
         MOV      @RPRES+,2(R5)
         MOV      @RPRES,4(R5)      ; LSBs
         ...                        ; Continue with program
```

Examples for the Basic Arithmetic Operations

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

1) The registers used 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 address 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 previously allocated.
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 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 to the value after the save register subroutine.
8) The registers used, 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 registers used, 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 previously allocated.

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) and 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 to the value after the save register subroutine.

8) The registers used, 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            ; Store MIDs and LSBs
         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 is reset to zero, no error occurred. If the N-bit is set to one, an error occurred. The kind of error can be seen in Table 5–6. The columns .FLOAT and .DOUBLE show the returned results for each error.

Table 5–6. Error Indication Table

Error	Status	.FLOAT	.DOUBLE
No error	N=0	xxxx,xxxx	xxxx,xxxx,xxxx
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 Dividend positive Dividend negative	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. The allocation in Figure 5–19 is shown for the use of FLT_SAV.

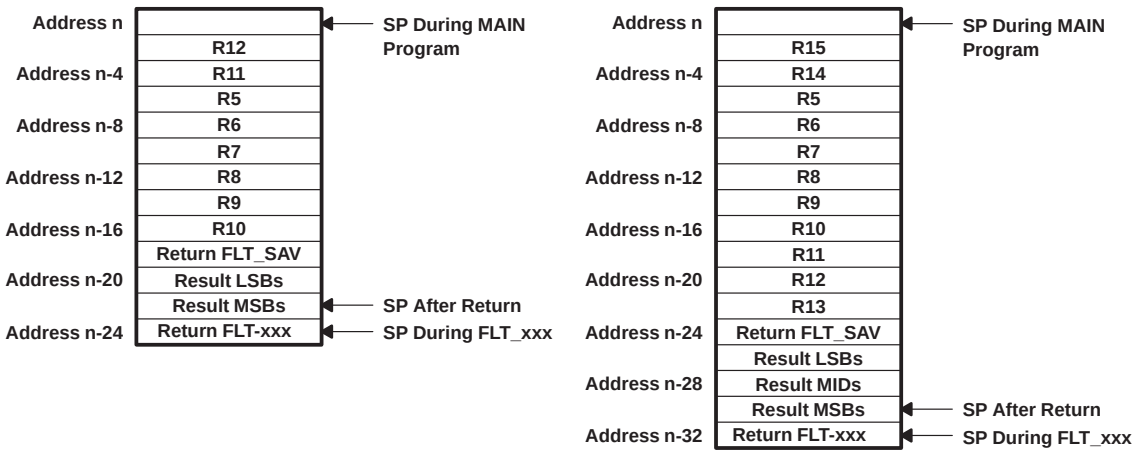

Figure 5–19. Stack Allocation for .FLOAT and .DOUBLE Formats

The FPP-subroutines correctly work only when the previous allocation is provided. This means the SP points to the return address on the stack. If the FPP-subroutines are called inside of a subroutine, a new result area must be allocated because the return address of the calling subroutine is now at the location the SP points to. The return address is overwritten in this case. The following example shows the correct procedure:

```
SUBR    SUB     #(ML/8)+1,SP      ; Allocate new result area
        MOV     @RPARG+,0(SP)     ; Fetch argument 2 to new
        MOV     @RPARG+,2(SP)     ; result area
        .if     DOUBLE=1
        MOV     @RPARG,4(SP)
        .endif
        MOV     SP,RPARG          ; Point again to argument 2
        MOV     #xx,RPRES         ; Point to argument 1
        CALL    #FLT_xxx          ; Use new result area for calc.
        ...                       ; Continue with calculations
        MOV     @SP+,result       ; Free allocated stack
        MOV     @SP+,result+2     ; Store result, correct SP
        .if     DOUBLE=1
        MOV     @SP+,result+4
        .endif
        RET
```

Note that it is strongly recommended that conscientious housekeeping be provided for SP to avoid stack overflow.

5.6.3.7 Number Range and Resolution

E = exponent of the floating point number. See Section 5.6.5, *Internal Data Representation* for more information.

.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	FFFF,FFFF	$-2^{127} \times (2 - 2^{-23})$	$= -3.402823 \times 10^{38}$
Resolution		$2^{-23} \times 2^{E}$	$= 119.2093 \times 10^{-9}\ 2^{E}$

.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	FFFF,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 signs are compared in a first step to find out the relation of the two arguments. When the signs of the two operands are equal, the mantissas are compared. After the comparison, the status bits of the status register (SR) hold the result: The registers RPRES and RPARG point to the same location the SP points to (for the FPP version 3 they were not defined).

Table 5–7. Comparison Results

Relations	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 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
        JHS     ARG1_GT_ARG2      ; ARG1 is greater than ARG2
        ......                    ; ARG1 is less than ARG2
EQUAL   ......                    ; ARG1 and ARG2 are equal
ARG1_GT_ARG2  ..                  ; ARG1 is greater than ARG2
;
; Other possibilities after the return
;
        CALL    #FLT_CMP          ; Comparison: result to SR
```

```
        JHS     ARG1_GE_ARG2      ; ARG1 is greater/equal ARG2
        ......                    ; ARG1 is less than ARG2
;
        CALL    #FLT_CMP          ; Comparison: result to SR
        JNE     ARG1_NE_ARG2      ; @RPRES not equal to @RPARG
        ......                    ; ARG1 is equal to ARG2
;
        CALL    #FLT_CMP          ; Comparison: result to SR
        JLO     ARG1_LT_ARG2      ; ARG1 is less than ARG2
        ......                    ; ARG1 is greater/equal ARG2
```

5.6.5 Internal Data Representation

The following description explains 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
- 24 (or 40 in the case of DOUBLE format) least significant bits hold the sign and the mantissa.

Figure 5–20. Floating Point Formats for the MSP430 FPP

Where:

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 previous equation is the floating zero. It is represented by all zeroes (32 if FLOAT format or 48 if DOUBLE format). No negative zero exists, the corresponding number (0080,0000) is a valid non-zero number and is the smallest negative number.

A frequently asked question is why the MSP430 floating point format does not conform to the widely used IEEE format. There are two main reasons why this is not the case:

1) The MSP430 is often used in a real time environment where calculations need to be completed before the next input data are present.

2) Battery-supplied applications make calculations quickly to produce longer battery lifes (up to 10 years for example).

These two main reasons make a run-time optimized floating point package necessary. The format of the floating-point number plays an important role in reaching this target.

- With the MSP430-format, every floating-point number represents a valid value. No invalid combinations like Not a Number, Denormalized Number, or Infinity exist. This way the MSP430 FPP has a larger range than other FPPs. This allows a higher speed with the smallest memory usage. This eliminates the need for unnecessary checks for invalid numbers.

- The exponent of the IEEE-format is located in two bytes because of the location of the sign in the MSB of the floating point number. With the MSP430-format, the exponent resides completely within the high byte of the most significant word and can, therefore, use the advantages of the byte-oriented architecture of the MSP430. No shifts and no bit handling are necessary to manipulate the exponent.

5.6.5.1 Computation of the Mantissa M

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

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

The result of the previous calculation is always:
$2 > M \geq 1$

Because 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 called hidden bit. The sign bit is located at this place instead:

Sm = 0: positive Mantissa

Sm = 1: negative Mantissa

Note:

The mantissa of a negative floating point number is NOT represented as a 2's-complement number, only the sign bit (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: The exponent is negative

MSB of exponent = 1: The 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 one operand 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
```

Table 5–8 shows the number of cycles needed for the previously shown calculations:

Table 5–8. CPU Cycles needed for Calculations

Operation		.FLOAT	.DOUBLE	Comment
Addition	X + Y	184	207	
Subtraction	X – Y	177	199	
Multiplication	X × Y	395	692	Software Loop
Multiplication	X × Y	153	213	Hardware MPYer
Division	X / Y	405	756	
Comparison	X – Y	37	41	

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. See Section 5.6.7.2, Binary to Floating Point Conversions.
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 when it is the result of the previous operation on the TOS

3) The result is located on the top of the stack (TOS), SP, RPARG, and RPRES point to the most significant word of the result

4) Only integers are converted. See Section 5.6.7.3, *Handling of Noninteger Numbers*, for the handling of non-integer numbers

5) The result is normally calculated using truncation, except when 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 Section 5.6.7.4, Rounding and Truncation, for details.

6) The subroutines can 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 (three) allocated words on the top of the stack. These words contain the result after the completed operation. A simple instruction is used for this allocation. It is the same allocation that is necessary anyway for the basic arithmetic operations. The possible instructions follow:

```
ML       .equ     24               ; For .FLOAT. ML = 40 for .DOUBLE
FPL      .equ     (ML/8)+1         ; Length of one FP number
         SUB      #4,SP            ; .FLOAT format allocation
         SUB      #6,SP            ; .DOUBLE format allocation
   or    SUB      #(ML/8)+1,SP     ; For both formats
   or    SUB      #FPL,SP          ; For both formats
```

8) The FPP04.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.

Binary to Floating Point Conversions

Binary numbers, 16-bit, 32-bit, and 40-bit long, are converted to floating point numbers. The subroutine call used defines if the binary number is treated as a signed or an unsigned number. No errors are possible, the N-bit of SR 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 (see Figure 5–21.

Range: –32768 to + 32767 (08000h to 07FFFh)

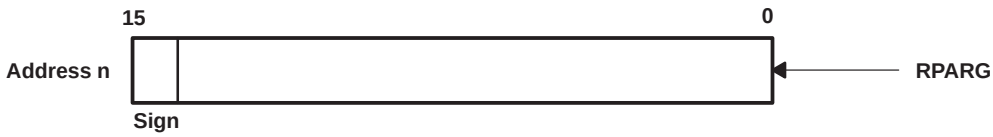

Figure 5–21. Signed Binary Input Buffer Format 16 Bits

CNV_BIN16U The 16-bit number, RPARG points to, is treated as a 16-bit unsigned number (see Figure 5–22).

Range: 0 to + 65535 (00000h to 0FFFFh)

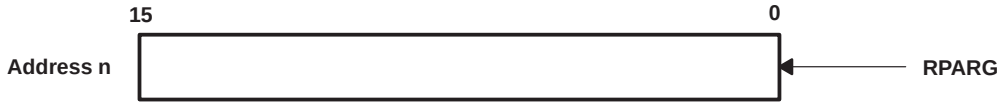

Figure 5–22. Unsigned Binary Input Buffer Format 16 Bits

CNV_BIN32 The 32-bit number, RPARG points to, is treated as a 32-bit signed number (see Figure 5–23).

Range: -2^{31} to $+2^{31} - 1$ (08000,0000h to 07FFF,FFFFh)

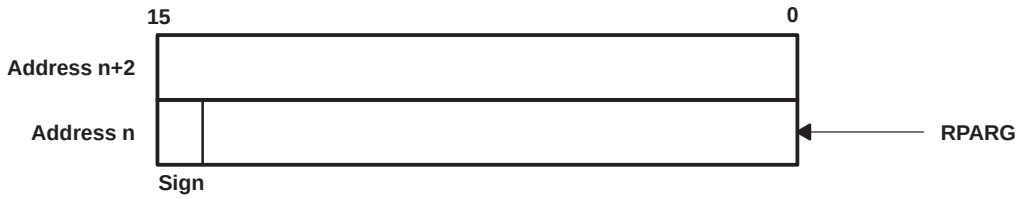

Figure 5–23. Signed Binary Input Buffer Format 32 Bits

CNV_BIN32U The 32-bit number, RPARG points to, is treated as a 32-bit unsigned number (see Figure 5–24).

Range: 0 to $2^{32} - 1$ (00000,0000h to 0FFFF,FFFFh)

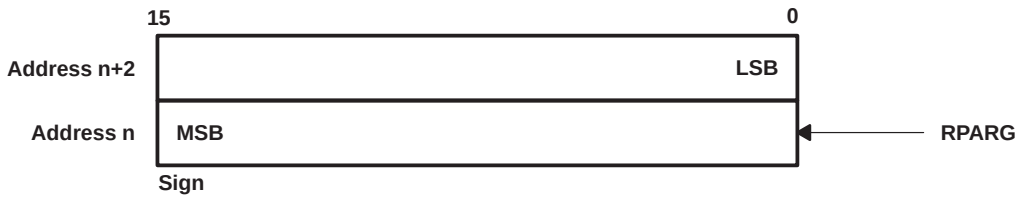

Figure 5–24. Unsigned Binary Input Buffer Format 32 Bits

CNV_BIN40 The 48-bit number, RPARG points to, is treated as a 40-bit signed (unsigned number) (see Figure 5–25).

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)

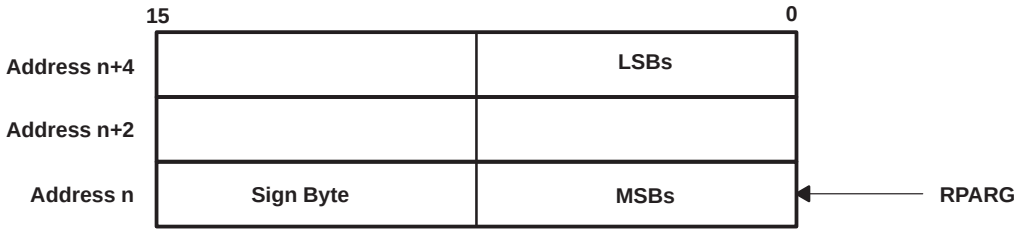

Figure 5–25. Binary Number Format 48 Bit

The previous 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 call (signed or unsigned) used, 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 (see Figure 5–26).

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)

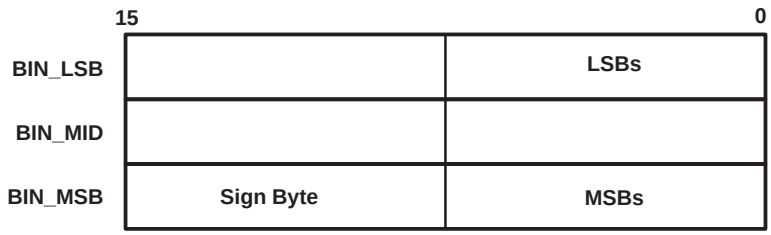

Figure 5–26. Binary Number Format 48 Bit

Note:

Input values outside of the 40-bit range, shown previously, 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 SR is always cleared on return.

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

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

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

```
DOUBLE   .EQU     1                ; Define .DOUBLE format
         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
         ...
```

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

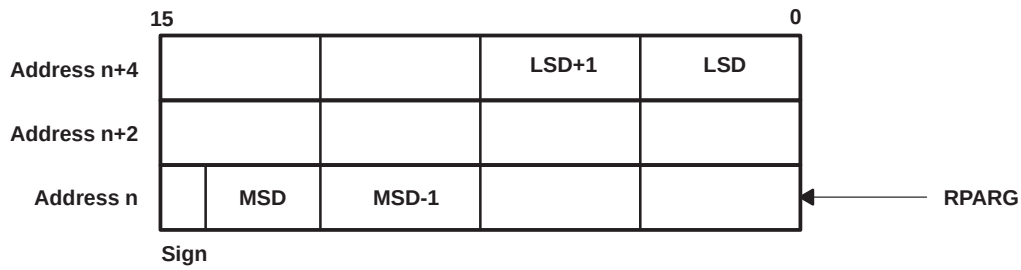

Figure 5–27. BCD Buffer Format

CNV_BCD_FP The 12-digit number (contained in 3 words, see Figure 5–27), 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 MSD 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                ; Define .FLOAT format
         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
```

Floating Point to Binary Conversion

The floating point number pointed to by register RPARG is converted to a 40-bit signed binary number located on the top of the stack after conversion (see Figure 5–28).

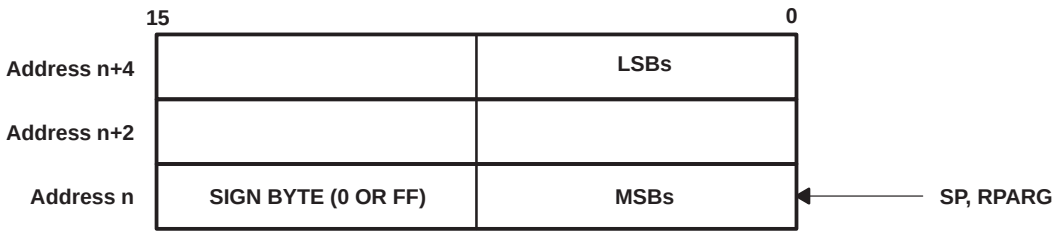

Figure 5–28. Binary Number Format

CNV_FP_BIN The floating point number at the address in RPARG 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, put 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. The sign uses a full byte.

.FLOAT 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 result is read.

.DOUBLE 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, 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
```

Floating Point to Binary-Coded Decimal Conversion

The floating point number at the address in RPARG is converted to a signed 12-digit BCD number located on the top of the stack after conversion (see Figure 5–27). The MSD of the result has a maximum value of 7 because the sign bit uses the MSB position.

CNV_FP_BCD The floating point number at the address in RPARG 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. These errors set the N-bit in the status register:

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

Otherwise, the N bit is cleared.

The result, on the 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 (see Figure 5–27).

.FLOAT 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 SP, RPRES and RPARG point to the MSDs of the three-word BCD result.

EXAMPLE: The floating point number (.FLOAT format) contained in RAM locations starting at label FPHI (MSBs) is converted to a 12-digit BCD number. The result is written to RAM addresses RES, 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
         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 handle only 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 larger

3) The integration of non-integer numbers is easier outside of the conversion subroutines

4) The execution time grows longer due to the necessary successive divisions or multiplies by 10. This cannot be tolerated in real time environments.

Binary to Floating-Point Conversion

If the location of the decimal point in the binary or hexadecimal number is known, the correction of the result is as follows:

The resulting floating point number is divided by the constant 2^n for binary numbers or 16^m for hexadecimal numbers (with m = 0.25 n). This is made simply by subtracting 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
```

Binary-Coded Decimal (BCD) to Floating-Point Conversion

If the location of the decimal point in the BCD number is known, 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 ($-8 \times 10^{11} + 1$ to $+8 \times 10^{11} - 1$) 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 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 contains 123456 and 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 can 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                 ; Correction constant 10
```

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 can occur if the floating-point number is very large. A very large floating-point number cannot be converted to binary format.

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 re-

sult. This means the result needs to be 16-times larger. For example, the floating-point number is 12.125 and 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 can be modified then a simplified code can 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
```

Floating Point to Binary Coded Decimal Conversion

If the BCD result of this conversion contains n digits after the decimal point, the following procedure can be used.

The floating-point number is multiplied by the constant 10^n before the conversion call. Overflow can occur if the floating-point number is very large. A very large floating-point number cannot be converted to BCD format due to the buffer length limit (12 digits maximum).

EXAMPLE: The floating-point number contained in the RAM locations starting at label FPHI (MSBs) is 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 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), the intermediate result is incremented. Otherwise, the result is not affected. If a carry occurs during the incrementing, the exponent is also corrected. Rounding is used if SW_RND = 1 is selected during assembly.

Rounding is applied (when SW_RND = 1) 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 (SW_RND = 1), the ROM code of the conversion subroutines is approximately 26 bytes larger than with truncation selected (SW_RND = 0).

5.6.7.5 Execution Cycles

To illustrate how long data conversion takes, the required cycles for each conversion are given for the converted values 1 and the largest possible value ($8 \times 10^{11} - 1$ for BCD conversions and $2^{40} - 1$ for binary conversions). The cycle count is given for the .FLOAT and for the .DOUBLE format and rounding is used.

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

Table 5–9. Execution Cycles of the Conversion Routines

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 of the Floating Point Package

The memory requirements of an implemented floating-point package depend 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.

Table 5–10. Memory Requirements without Hardware Multiplier

Package	.FLOAT	.DOUBLE
Basic Arithmetic Operations	604	696
Conversion Subroutines	342	338
Complete FPP	946	1034

Table 5–11. Memory Requirements with Hardware Multiplier

Package	.FLOAT	.DOUBLE
Basic Arithmetic Operations	638	786
Conversion Subroutines	342	338
Complete FPP	980	1124

5.6.9 Inclusion of the Floating-Point Package into the Customer Software

This section shows how to insert the floating-point package into the user's software. The symbolic definition of the working registers makes it necessary to include the FPP-definition file (FPPDEF4.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 fails. The two files FPP04.ASM and CNV04.ASM need to be located together as shown in the following examples. This is due to the common parts that are connected with jumps.

The constant DOUBLE decides which FPP version is generated. It is assumed that the FPP files are located in a directory named `c:\fpp`. If this is not the case, then the name of this directory is to be used.

```
;
         .text    08000h            ; ROM/EPROM start address
STACK    .equ     0600h             ; Initial value for SP
;
DOUBLE   .equ     1                 ; Use .DOUBLE format FPP
SW_UFLOW .equ     0                 ; Underflow is no error
SW_RND   .equ     1                 ; Use rounding for conversions
HW_MPY   .equ     1                 ; Use the hardware multiplier
;
         .copy    c:\fpp\fppdef4.asm        ; FPP Definitions
         .copy    c:\fpp\fpp04.asm          ; FPP file
         .copy    c:\fpp\cnv04.asm          ; FPP Conversions
;
; Customer software starts here
;
START    MOV      #STACK,SP                 ; Allocate stack
         .....                              ; User's SW starts here
; Power–up start address:
;
         .sect    "RstVect",0FFFEh
         .word    START                     ; Reset vector
```

A second possibility is shown in the following. The FPP is 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
HW_MPY   .equ     0                 ; No hardware multiplier
;
         .copy    c:\fpp\fppdef4.asm        ; FPP Definitions
;
; Customer software starts here
```

```
;
START    MOV      #STACK,SP                ; Allocate stack
         .....                             ;
         .....                             ; End of user's software
         .copy    c:\fpp\fpp04.asm         ; Copy FPP file
         .copy    c:\fpp\cnv04.asm         ; Copy conversions
;
; Power–up start address:
;
         .sect    "RstVect",0FFFEh
         .word    START                    ; Reset vector
```

5.6.10 Software Examples

The following subroutines for mathematical functions use the same conventions like the basic arithmetic functions described previously.

- RPARG points to the operand X for single operand functions (lnX, e^X)
- RPRES points to the first operand (base) and RPARG to the second one if two operands are used (e.g. for the power function a^b)
- The result of the operation is placed on the top of the stack, RPARG, RPRES and SP point to the result.

5.6.10.1 Square Root Subroutine

The following subroutine shows the use of the floating-point package for the calculation of the square root of a number X. The NEWTONIAN approach is used:

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

The subroutine uses the RPARG register as a pointer to the number X and places the result on the top of the stack.

The algorithm used for the first estimation – exponent/2 and different correction for even and odd exponents – leads to the worst case estimation errors of +8% and –13%. This relatively exact estimations lead to only four iteration loops to get the full accuracy.

The number range of X for the square-root function contains all positive numbers including zero. Negative values for X return the previous result on the top of the stack and the N bit set as an error indication.

The calculation errors for the square-root function are shown in the following table. They indicate relative errors.

Table 5–12. Relative Errors of the Square Root Function

X	.FLOAT	.DOUBLE	Comment
$+3.0\times10^{-39}$	6.8×10^{-8}		Smallest FPP number
0.0	0	0	Zero
1.0	0	0	
6.0	$+5.4\times10^{-9}$	$+1.3\times10^{-12}$	
8.0	$+6.7\times10^{-8}$	$+1.3\times10^{-12}$	
$+3.4\times10^{38}$	$+4.6\times10^{-9}$	$+2.2\times10^{-11}$	Largest FPP number

Calculation times:

.FLOAT with hardware multiplier:	2300 cycles	4 iterations
.FLOAT without hardware multiplier:	2300 cycles	(no multiplication used)
.DOUBLE with hardware multiplier:	4000 cycles	4 iterations
.DOUBLE without hardware multiplier:	4000 cycles	

```
; Square Root Subroutine X^0.5      Result on TOS = (@RPARG)^0.5
;
; Call:  MOV      #addressX,RPARG   ; RPARG points to address of X
;        CALL     #FLT_SQRT         ; Call the square root function
;        ...                        ; RPARG, RPRES and SP point to
;                                   ; result X^0.5. N–bit for error
;
; Range: 0 =< X < 3.4x10^+38
;
; Errors:         X < 0:   N = 1    Result: previous result
;
; Stack: FPL + 2 bytes
;
; Calculates the square root of the number X, RPARG points to.
; SP, RPARG and RPRES point to the result on TOS
;
FLT_SQRT .equ $
         TST.B    0(RPARG)          ; Argument negative?
         JN       SQRT_ERR          ; Yes, return with N = 1
```

```
        MOV     @RPARG+,2(SP)     ; Copy X to result area
        MOV     @RPARG+,4(SP)
        .if     DOUBLE=1
        MOV     @RPARG+,6(SP)
        .endif
        CLR     HELP
        .if     DOUBLE=1
        TST     6(SP)             ; Check for X = 0
        JNE     SQ0
        .endif
        TST     4(SP)
        JNE     SQ0
        TST     2(SP)
        JEQ     SQ3               ; X = 0: result 0, no error
;
SQ0     PUSH    #4                ; Loop count (4 iterations)
        PUSH    FPL+4(SP)         ; Push X on stack for Xn
        PUSH    FPL+4(SP)
        .if DOUBLE=1
        PUSH    FPL+4(SP)
        .endif
;
; 1st estimation for X^0.5: exponent even: 0.5 x fraction + 0.5
;                                  exponent odd:  fraction .or. 0.30h
;                                  exponent/2
;
        RRA.B   1(SP)             ; Exponent/2
        JC      SQ1               ; Exponent even or odd?
        RRA.B   @SP               ; Exponent is even:
        JMP     SQ2               ; 0.5 + 0.5 x fraction
SQ1     BIS.B   030h,0(SP)        ; Exponent is odd: correction
SQ2     XOR.B   #040h,1(SP)       ; Correct exponent
;
SQLOOP  MOV     SP,RPARG          ; Pointer to Xn
        MOV     SP,RPRES
        ADD     #FPL+4,RPRES      ; Pointer to X
```

```
          SUB     #FPL,SP          ; Allocate stack for result
          CALL    #FLT_DIV         ; X/xn
          ADD     #FPL,RPARG       ; Point to xn
          CALL    #FLT_ADD         ; X/xn + xn
          DEC.B   1(RPRES)         ; 0.5 x (X/xn + xn) = xn+1
          MOV     @SP+,FPL–2(SP)   ; xn+1 –> xn
          MOV     @SP+,FPL–2(SP)
          .if     DOUBLE=1
          MOV     @SP+,FPL–2(SP)
          .endif
          DEC     FPL(SP)          ; Decrement loop counter
          JNZ     SQLOOP
          MOV     @SP+,FPL+2(SP)   ; N = 0 (FLT_ADD)
          MOV     @SP+,FPL+2(SP)   ; Root to result space
          .if     DOUBLE=1
          MOV     @SP+,FPL+2(SP)
          .endif
          ADD     #2,SP            ; Skip loop count
SQ3       BR      #FLT_END         ; To completion part
SQRT_ERR  MOV     #FN,HELP         ; Root of negative number: N = 1
          JMP     SQ3              ;
```

5.6.10.2 Cubic-Root Subroutine

The cubic root of a number is calculated the same as the square root, using the Newtonian approach. The formula for the cubic root of X is:

$$x_{n+1} = \frac{1}{3}\left(2x_n + \frac{X}{x_n^2}\right)$$

The subroutine uses the RPARG register as a pointer to the number X and places the result on the top of the stack.

The algorithm used for the first estimation – exponent/3 and a constant fraction value ±1.4 – leads to worst case estimation errors of +40% and –37%. This estimation leads to four (.FLOAT) or five (.DOUBLE) iteration loops to get the full accuracy.

The number range of X for the cubic-root function contains all numbers including zero. No error is possible.

The calculation errors for the cubic-root function are shown in the following table. They indicate relative errors.

Table 5–13. Relative Errors of the Cubic Root Function

X	.FLOAT	.DOUBLE	Comment
-3.4028×10^{38}	1.2×10^{-8}	$+2.2\times10^{-13}$	Most negative number
–1.0	0	0	–1.0
-2.9387×10^{-39}	1.7×10^{-7}	-3.8×10^{-13}	Least negative number
0.0	0	0	Zero
$+2.9387\times10^{-39}$	-1.7×10^{-7}	$+3.8\times10^{-13}$	Least positive number
+1.0	0	0	+1.0
$+3.4028\times10^{38}$	-1.2×10^{-8}	-2.2×10^{-13}	Most positive number

Calculation times:

.FLOAT with hardware multiplier: 5000 cycles 4 iterations

.FLOAT without hardware multiplier: 6100 cycles

.DOUBLE with hardware multiplier: 10200 cycles 5 iterations

.DOUBLE without hardware multiplier: 12600 cycles

```
; Cubic Root Subroutine X^1/3  Result on TOS = (@RPARG)^1/3
;
; Call:  MOV      #addressX,RPARG ; RPARG points to X
;        CALL     #FLT_CBRT       ; Call the cubic root function
;        ...                      ; RPARG, RPRES, SP point to result
;                                 ; Result on the top of the stack
;
; Formula:        xn+1 = 1/3(2xn + X x xn^–2)
;
; Range:          –3.4x10^+38 =< X =< 3.4x10^+38
;
; Errors:                          No errors possible
;
; Stack:          2 x FPL + 2 bytes
;
; Calculates the cubic root of the number X, RPARG points to.
; SP, RPARG and RPRES point to the result on TOS
;
```

```
FLT_CBRT MOV     @RPARG+,2(SP)     ; Copy X to result area
         MOV     @RPARG+,4(SP)
         .if     DOUBLE=1
         MOV     @RPARG+,6(SP)
         .endif
         .if DOUBLE=1
         TST 6(SP)                 ; Check for X = 0
         JNE CB0
         .endif
         TST 4(SP)
         JNE CB0
         TST 2(SP)
         JEQ CB3                   ; X = 0: result 0
;
CB0      .equ    $
         .if     DOUBLE=0          ; Loop count
         PUSH    #4                ; .FLOAT  4 iterations
         .else
         PUSH    #5                ; .DOUBLE 5 iterations
         .endif
         PUSH    FPL+4(SP)         ; Push X on stack for Xn
         PUSH    FPL+4(SP)
         .if DOUBLE=1
         PUSH    FPL+4(SP)
         .endif
;
; 1st estimation for X^1/3:        exponent/3, fraction = +–1.4
;
         MOV.B   1(SP),RPARG       ; Exponent of X 00xx
         AND     #080h,0(SP)       ; Only sign of X remains
         ADD     #08034h,0(SP)     ; +–1.4 for 1st estimation
         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     #FPL,SP         ; Allocate stack for result
        CALL    #FLT_MUL        ; xn^2
        ADD     #2*FPL+4,RPRES  ; Point to A
        CALL    #FLT_DIV        ; X/xn^2
        INC.B   FPL+1(SP)       ; xn x 2
        ADD     #FPL,RPARG      ; Point to 2xn
        CALL    #FLT_ADD        ; X/xn^2 + 2xn
        MOV     #FLT3,RPARG     ; 1/3 x (X/xn^2 + 2xn) = xn+1
        CALL    #FLT_DIV
        MOV     @SP+,FPL–2(SP)  ; xn+1 –> xn
        MOV     @SP+,FPL–2(SP)
        .if     DOUBLE=1
        MOV     @SP+,FPL–2(SP)
        .endif
        DEC     FPL(SP)         ; Decr. loop count
        JNZ     CBLOOP
        MOV     @SP+,FPL+2(SP)  ; Result to result area
        MOV     @SP+,FPL+2(SP)  ; Cubic root to result space
        .if     DOUBLE=1
        MOV     @SP+,FPL+2(SP)
        .endif
        ADD     #2,SP           ; Skip loop count
CB3     CLR     HELP            ; No error
        BR      #FLT_END        ; Normal termination
;
        .if     DOUBLE=1
FLT3    .DOUBLE 3.0             ; Constant for cubic root
        .else
FLT3    .FLOAT  3.0
```

```
.endif
```

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.

```
SUB     #ML/8+1,SP        ; Allocate work area
MOV     #NUMBER,RPARG     ; Address of NUMBER to RPARG
CALL    #FLT_SQRT         ; Square root of NUMBER on TOS
JN      ERROR             ; NUMBER is negative
CALL    #FLT_SQRT         ; Fourth root on TOS
MOV     @SP+,RESULT       ; 4th root MSBs
MOV     @SP+,RESULT+2     ; Correct SP to previous result
.if     DOUBLE=1
MOV     @SP+,RESULT+4     ; LSBs for DOUBLE
.endif
```

5.6.10.4 Other Root Subroutines

Using the same calculations shown previously, higher roots can also be calculated using the Newtonian approach. The generic formula for the mth root out of A is:

$$x_{n+1} = \frac{1}{m}\left((m-1)x_n + \frac{A}{x_n^{m-1}}\right)$$

To get short calculation times – which means only few iterations are necessary – the choice of the first estimation x_0 is very important. For the above formula a good first iteration x_0 is (M = mantissa, E = exponent):

$$x_0 = \left(\frac{(M-1)}{m} + 1\right) \times 2^{E/m}$$

5.6.10.5 Calculations With Intermediate Results

If a calculation cannot be executed simply and has intermediate results, a new result space is used. This is done by subtracting 4 (.FLOAT) or 6 (.DOUBLE) from the stack pointer.

EXAMPLE: The following function for e is to be calculated. The example is valid for both formats:

$$e = a \times b - \frac{c}{d}$$

```
FPL     .equ    (ML/8)+1          ; Length of a FPP number
;
        SUB     #FPL,SP           ; Allocate result space 0 (RS0)
        MOV     #a,RPRES          ; Address argument 1
        MOV     #b,RPARG          ; Address argument 2
        CALL    #FLT_MUL          ; a x b -> RS0
        SUB     #FPL,SP           ; Allocate result space 1 (RS1)
        MOV     #c,RPRES          ; Address c
        MOV     #d,RPARG          ; Address d
        CALL    #FLT_DIV          ; c/d -> RS1
        ADD     #FPL,RPRES        ; Address (a x b) in RS0
        CALL    #FLT_SUB          ; e = (a x b) - c/d -> RS1
        MOV     @SP+,FPL-2(SP)    ; Result e to RS0
        MOV     @SP+,FPL-2(SP)    ; Overwrite (a x b) with e
        .if     DOUBLE=1
        MOV     @SP+,FPL-2(SP)    ; LSBs for DOUBLE
        .endif
;
; Housekeeping is made, SP points to RS0 again, but not
; RPARG and RPRES
```

EXAMPLE: The multiply-and-add (MAC) function for e shown in the following is calculated. The example is written for both formats:

$$e_{n+1} = a \times b + e_n$$

```
        SUB     #ML/8+1,SP        ; Allocate result space
        MOV     #a,RPRES          ; Address argument 1
        MOV     #b,RPARG          ; Address argument 2
        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 or LSBs
        .if     DOUBLE=1
        MOV     @RPARG+,e+4       ; LSBs
        .endif
;
; 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 done by simply resetting the sign bit of the 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 done by simply inverting the sign bit of the 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 in the following example. The pointer RPARG is loaded with the address of the number. The result is then placed on the top of the stack. No error is possible. Numbers below 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
; Call   MOV     #number,RPARG     ; Address to RPARG
;        CALL    #FLT_INTG         ; Call subroutine
;        ...                       ; Result on TOS
;
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
```

```
        ....                    ; Integer on TOS
```

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 in the following example. 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 previously.

Integer values or very large numbers return a zero value due to the given 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 used
; Call   MOV     #number,RPARG     ; Address to RPARG
;        CALL    #FLT_FRCT         ; Call subroutine
;        ...                       ; Result on TOS
;
FLT_FRCT PUSH    RPARG             ; 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     ML/8+1(SP),RPRES  ; Operand address to RPRES
         CALL    #FLT_SUB          ; Operand – Integer part to TOS
         .if     DOUBLE=1          ; Housekeeping:
         MOV     @SP+,ML/8+3(SP)   ; Fractional part back
         .endif
         MOV     @SP+,ML/8+3(SP)   ; To result area
         MOV     @SP+,ML/8+3(SP)
         ADD     #2,SP             ; Skip saved operand address
         CLR     HELP              ; No error
         BR      #FLT_END          ; Use FPP termination
```

```
;
```

EXAMPLE: the fractional part of the floating-point number R5 points to is placed on TOS.

```
MOV     R5,RPARG          ; Load pointer with address
CALL    #FLT_FRCT         ; Calculate fractional part
....                      ; Fractional part on TOS
```

5.6.10.10 Approximation of Integrals

Simpson's Rule states that the area A limited by the function f(x), the x-axis, x_0 and x_N is approximately:

$$A = \int_{x_0}^{x_N} f(x) \approx \frac{1}{3} \times h \times \left[(y_0 + y_N) + 2(y_2 + y_{4 \ldots} y_{N-2}) + 4(y_1 + y_{3 \ldots} y_{N-1}) \right]$$

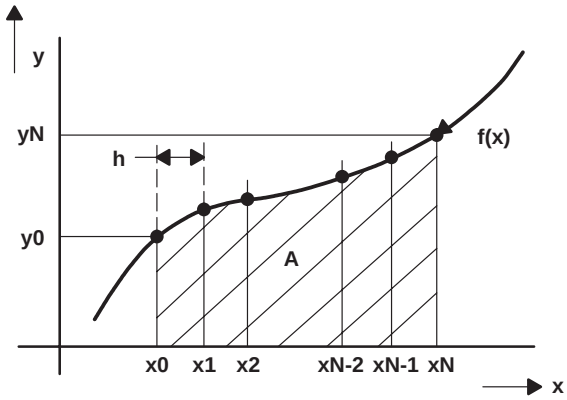

Figure 5–29. Function f(x)

The subroutine SIMPSON, in the following code, processes N+1 inputs pointed to by register RPARG and computes the area A after the measurement of sample N. The result is written back to the RAM location A.

This integration method can be used for the calculation of the apparent power with electronic electricity meters. The absolute values of current and voltage are added up and are multiplied afterwards.

```
;
; Subroutine for the approximation of integrals. Samples
; y0 to yN are processed and stored in location A.
; Nmax = 254 (if larger, a word has to be used for INDEXn)
```

```
;
; Call:  CLR.B    INDEXn          ; Before 1st call: n = 0
; LOOP   MOV      #sample,RPARG   ; Address of yn
;        CALL     #SIMPSON        ; Process sample yn
;        CMP.B    #N+1,INDEXn     ; YN processed?
;        JLO      LOOP            ; No, proceed
;        ...                      ; Yes, integral in A
;
N        .equ     8               ; Max. index (must be even)
         .if      DOUBLE=0
A        .equ     0200h           ; summed up value (integral)
INDXn    .equ     0204H           ; Index n (0 to N)
FLT3     .float   3.0
h        .float   0.32            ; Difference h: yn+1 – yn
         .else
A        .equ     0200h           ; Summed up value (integral)
INDXn    .equ     0206H           ; Index n (0 to N)
FLT3     .double  3.0
h        .double  0.32            ; Difference h: yn+1 – yn
         .endif
SIMPSON  SUB      #(ML/8)+1,SP    ; Allocate new workspace
         MOV      @RPARG+,0(SP)   ; Fetch yn
         MOV      @RPARG+,2(SP)
         .if DOUBLE=1
         MOV      @RPARG,4(SP)
         .endif
         CMP.B    #0,INDXn        ; 1st value y0?
         JEQ      Y0
         CMP.B    #N,INDXn        ; Last value yN?
         JEQ      YN
         BIT      #1,INDXn        ; Odd or even n?
         JZ       YEVEN
         INC.B    1(SP)           ; Odd: value x 4
YEVEN    INC.B    1(SP)           ; Even: value x 2
         MOV      #A,RPARG        ; Fetch summed–up value A
         MOV      SP,RPRES        ; New sample yn on TOS
```

```
        CALL    #FLT_ADD        ; Add it to A
        JMP     Y0              ; Store added result in A
;
YN      MOV     #A,RPARG        ; Last value yN: calculate
        MOV     SP,RPRES        ; New sample yn on TOS
        CALL    #FLT_ADD        ; Add last result to A
        MOV     #FLT3,RPARG     ; To constant 3.0
        CALL    #FLT_DIV        ; Divide summed–up value by 3.0
        MOV     #h,RPARG        ; Multiply with distance h
        CALL    #FLT_MUL
;
Y0      MOV     @SP+,A          ; Store result to A
        MOV     @SP+,A+2        ; and correct stack
        .if     DOUBLE=1
        MOV     @SP+,A+4
        .endif
        INC.B   INDXn           ; Next n
        RET                     ; Return with integral in A
```

EXAMPLE: The function f(x) described by the calculated results on top of the stack is integrated using Simpson's rule..

```
        CLR.B   INDXn           ; Initialization: INDXn = 0
INTLOP  ...                     ; Calculation, result on TOS
        CALL    #SIMPSON        ; Process samples y0 to yN
        CMP.B   #N+1,INDXn      ; Last sample yN processed?
        JLO     INTLOP          ; No, continue
        ...                     ; Yes, result in A
```

5.6.10.11 Statistical Calculations

The mean value, the standard deviation, and the variance of measured samples can be calculated with the following subroutines.

- ❏ STAT_INIT clears the RAM locations used for data gathering.
- ❏ STAT_PREP adds the input sample to the RAM location SUMYi, the squared input sample to SUM2Yi and increments the sample counter N.

- STAT_CALC calculates mean, standard deviation, and variance from these three values and writes them back to the RAM locations used for data recording.

$$MeanValue = \frac{\sum_{i=1}^{i=N} y_i}{N}$$

$$Variance = \frac{\sum_{i=1}^{i=N} y_i^{\,2} - \frac{\left(\sum_{i=1}^{i=N} y_i\right)^2}{N}}{N} = \frac{\sum_{i=1}^{i=N} y_i^{\,2} - MeanValue \times \sum_{i=1}^{i=N} y_i}{N}$$

$$StandardDeviation = \sqrt{\frac{\sum_{i=1}^{i=N} y_i^{\,2} - \frac{\left(\sum_{i=1}^{i=N} y_i\right)^2}{N}}{N-1}} = \sqrt{Variance \times \frac{N}{N-1}}$$

```
;
; RAM locations for the input samples:
;
N         .equ     0200h             ; Number of input samples (binary)
SUMYi     .equ     N+(ML/8)+1        ; Summed-up samples yi
SUM2Yi    .equ     SUMYi+(ML/8)+1    ; Sum of squared samples yi
;
; The same RAM-locations are used for the three results:
;
MEANV     .equ     N                 ; Mean Value after return
STDDEV    .equ     SUMYi             ; Standard Deviation after return
VARIANCE .equ      SUM2Yi            ; Variance after return
;
          .if      DOUBLE=1
FLT1      .DOUBLE  1.0               ; Floating 1.0
          .else
FLT1      .FLOAT   1.0
          .endif
;
```

```
; STAT_INIT initializes the RAM–locations for statistics
;
STAT_INIT CLR     N                 ; Clear sample counter
          CLR     SUM2Yi            ; Clear sum of squared samples
          CLR     SUM2Yi+2
          .if     DOUBLE=1
          CLR     SUM2Yi+4
          .endif
          CLR     SUMYi             ; Clear sum of input samples
          CLR     SUMYi+2
          .if     DOUBLE=1
          CLR     SUMYi+4
          .endif
          RET
; STAT_PREP sums–up the sample pointed to by RPARG in SUMYi
; (summed–up yi) and in SUM2Yi (summed–up squared yi).
; The binary sample counter N is incremented
;
STAT_PREP PUSH    RPARG             ; Save address of input sample
          SUB     #(ML/8)+1,SP      ; Allocate stack space
          MOV     RPARG,RPRES       ; Copy input sample address
          CALL    #FLT_MUL          ; (yi)^2
          MOV     #SUM2Yi,RPRES     ; Add (yi)^2 to SUM2Yi
          CALL    #FLT_ADD          ; (yi)^2 + SUM2Yi
          MOV     @SP,SUM2Yi        ; Sum back to SUM2Yi
          MOV     2(SP),SUM2Yi+2
          .if     DOUBLE=1
          MOV     4(SP),SUM2Yi+4
          .endif
          MOV     (ML/8)+1(SP),RPARG        ; Fetch sample address
          MOV     #SUMYi,RPRES      ; Add yi to SUMYi
          CALL    #FLT_ADD
          MOV     @SP+,SUMYi        ; Summed–up yi
          MOV     @SP+,SUMYi+2      ; House keeping
          .if     DOUBLE=1
          MOV     @SP+,SUM2Yi+4
```

```
        .endif
        ADD     #2,SP           ; Remove sample address
        INC     N               ; Increment N
        RET
;
; STAT_CALC calculates the Mean Value, the Variance and the
; Standard Deviation from the N samples input to the subroutine
; STAT_PREP.
; The three calculated statistical values are stored in:
; Mean Value:             N
; Variance:               SUM2Yi
; Standard Deviation:       SUMYi
;
STAT_CALC       SUB             #(ML/8)+1,SP    ; Allocate stack space
        MOV     #N,RPARG        ; Convert N to FP–format
        CALL    #CNV_BIN16U     ; Binary to FPP on TOS
        SUB     #(ML/8)+1,SP    ; To save N on stack
        MOV     #SUMYi,RPRES    ; Summed–up yi/N
        CALL    #FLT_DIV        ; Mean Value on TOS
        MOV     @SP,MEANV       ; Store Mean Value
        MOV     2(SP),MEANV+2
        .if     DOUBLE=1
        MOV     4(SP),MEANV+4
        .endif
;
; The Mean Value on TOS is used for the calculation
; of the Variance:
; Variance = (Sum(yi^2) – Mean Value x Sum(yi)/N)/N
;
        MOV     #SUMYi,RPARG    ; Mean Value x Sum(yi)
        CALL    #FLT_MUL        ;
        MOV     #SUM2Yi,RPRES   ; To Sum(yi^2)
        CALL    #FLT_SUB        ; Sum(yi^2) – MV x Sum(yi)
        ADD     #(ML/8)+1,RPARG ; Point to N
        CALL    #FLT_DIV        ; Variance on TOS
        MOV     @SP,VARIANCE    ; Store Variance
```

```
        MOV     2(SP),VARIANCE+2
        .if     DOUBLE=1
        MOV     4(SP),VARIANCE+4
        .endif
;
; The Variance on TOS is used for the calculation of the
; Standard Deviation: Std Dev. = SQUROOT(Variance x N/(N–1)
;
        ADD     #(ML/8)+1,RPARG   ; Point to N
        CALL    #FLT_MUL          ; Variance x N
        MOV     @SP+,STDDEV       ; Store value for later use
        MOV     @SP+,STDDEV+2
        .if     DOUBLE=1
        MOV     @SP+,STDDEV+4
        .endif
        MOV     #FLT1,RPARG       ; Build N–1
        MOV     SP,RPRES          ; point to N
        CALL    #FLT_SUB          ; N–1 on TOS
        MOV     #STDDEV,RPRES     ; point to (Variance x N)
        CALL    #FLT_DIV          ; Variance x N/(N–1)
        CALL    #FLT_SQRT         ; StdDev = SQROOT(Var x N/(N–1))
        MOV     @SP+,STDDEV       ; Store Standard Deviation
        MOV     @SP+,STDDEV+2
        .if     DOUBLE=1
        MOV     @SP+,STDDEV+4
        .endif
        RET
;
```

EXAMPLE: The normal calling sequence for the statistical calculations is shown in the following. The input samples are contained in the ADC-result register ADAT.

```
        CALL    #STAT_INIT        ; Initialization: clear used RAM
STATLOP MOV     #ADAT,RPARG       ; Set pointer to ADC–result
        CALL    #CNV_BIN16U       ; Convert ADC–result to FP on TOS
        CALL    #STAT_PREP        ; Process samples y1 to yN
        ....                      ; Continue
```

```
        CMP.B    #xx,N              ; yN processed?
        JLO      STATLOP            ; No, next sample
;
; N samples are pre-processed: Calculate Mean Value, Variance,
; and Standard Deviation out of SUMYi, SUM2Y1 and N
;
        CALL     #STAT_CALC         ; Call calculation subroutine
        ....                        ; Results in sample locations
```

5.6.10.12 Complex Calculations

Complex numbers of the form (a + jb) can be used in calculations also. The four basic arithmetic operations are shown for complex numbers. Pointers RPARG and RPRES are used in the same way as with the normal FPP subroutines. They point to the real parts of the complex numbers used for input and to the result on the TOS after the completion of the subroutine. The real and imaginary part of a complex number need to be allocated in the way shown in Figure 5–30 (shown for .FLOAT format).

Stack Usage: The subroutines need up to 36 bytes (.DOUBLE) or 28 bytes (.FLOAT) of stack space (complex division). Not included in this numbers is the initially allocated result space. No error handling is provided. It is assumed that the numbers used stay within the range of the floating-point package.

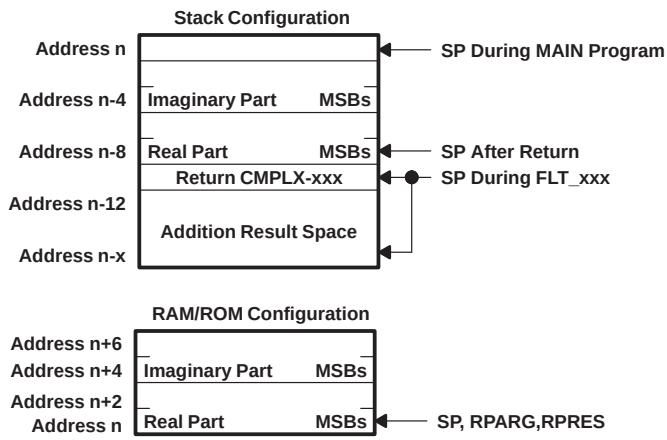

Figure 5–30. Complex Number on TOS and in Memory (.FLOAT Format)

```
FPL      .equ     (ML/8)+1           ; Length of an FP-number (bytes)
;
; Complex calculation is made with the complex numbers RPARG
```

```
; and RPRES point to.
;
; Call:  MOV      #arg1,RPRES       ; Address argument1
;        MOV      #arg2,RPARG       ; Address argument2
;        CALL     #CMPLX_xxx        ; Calculate arg1 op arg2
;        ...                        ; Result on TOS. Pointed to
;        ...                        ; by SP, RPARG and RPRES
;
; Complex Subtraction: (a + jb) – (c + jd). @RPRES – @RPARG.
; Stack Usage: 16 bytes (.DOUBLE)   12 bytes (.FLOAT)
;
CMPLX_SUB MOV     #0FFFFh,HELP      ; Define subtraction
         JMP      CL$1              ; To common part
;
; Complex Addition: (a + jb) + (c + jd). @RPRES + @RPARG
; Stack Usage: 16 bytes (.DOUBLE)   12 bytes (.FLOAT)
;
CMPLX_ADD CLR     HELP              ; Define addition
CL$1     PUSH     RPRES             ; Save argument pointer
         .if      DOUBLE=1
         PUSH     10(RPARG)         ; LSBs imaginary part d
         PUSH     8(RPARG)          ; MIDS imaginary part d
         .endif
         PUSH     6(RPARG)          ; The coming words depend on
         PUSH     4(RPARG)          ; DOUBLE
         PUSH     2(RPARG)          ; LSBs real part c
         PUSH     @RPARG            ; MSBs real part c
         TST      HELP              ; Addition or subtraction?
         JZ       CA
;
; Subtraction: the complex number (c + jd) is negated
;
         XOR      #080h,0(SP)       ; Negate real part c
         XOR      #080h,FPL(SP)     ; Negate imaginary part d
CA       MOV      SP,RPARG          ; Point to c
         CALL     #FLT_ADD          ; Add real parts (a + c)
```

```
        MOV     @SP+,2*FPL+2(SP)  ; To real storage
        MOV     @SP+,2*FPL+2(SP)  ; Housekeeping
        .if     DOUBLE=1
        MOV     @SP+,2*FPL+2(SP)
        .endif
        MOV     SP,RPARG          ; Point to d
        MOV     FPL(SP),RPRES     ; Restore RPRES
        ADD     #FPL,RPRES        ; To imaginary part b
        CALL    #FLT_ADD          ; Add imaginary parts (b + d)
        MOV     @SP+,2*FPL+2(SP)  ; To imaginary storage
        MOV     @SP+,2*FPL+2(SP)  ; Housekeeping
        .if     DOUBLE=1
        MOV     @SP+,2*FPL+2(SP)
        .endif
        ADD     #2,SP             ; Skip saved RPRES
        JMP     CMPLX_RT          ; Result on TOS
;
; Complex Division: (a + jb)/(c + jd). @RPRES/@RPARG
;
; The Complex Division uses the inverted divisor and
; the multiplication afterwards:
; (a + jb)/(c + jd) = (a + jb) x 1/(c + jd)
; with: 1/(c + jd) = (c – jd)/(c^2 + d^2)
;; Stack Usage: 36 bytes (.DOUBLE)   28 bytes (.FLOAT)
;
CMPLX_DIV PUSH    RPRES             ; Save RPRES (dividend a + jb))
        PUSH    RPARG             ; Save RPARG (divisor c + jd)
        SUB     #FPL,SP           ; Allocate result space
        MOV     RPARG,RPRES       ; Fetch real part c
        CALL    #FLT_MUL          ; c^2
        SUB     #FPL,SP           ; Allocate result space
        MOV     2*FPL(SP),RPARG   ; Fetch imaginary part d
        ADD     #FPL,RPARG
        MOV     RPARG,RPRES       ; Copy address of d
        CALL    #FLT_MUL          ; d^2
        ADD     #FPL,RPARG        ; to c^2
```

```
        CALL    #FLT_ADD          ; c^2 + d^2
        PUSH    FPL(SP)           ; Copy c^2 + d^2
        PUSH    FPL(SP)
        .if     DOUBLE=1
        PUSH    FPL(SP)
        .endif
        MOV     SP,RPARG          ; To (c2 +d2)
        MOV     3*FPL(SP),RPRES   ; Pointer to (c + jd)
        ADD     #FPL,RPRES        ; Address d
        CALL    #FLT_DIV          ; d/(c^2 + d^2) imag. part
        XOR     #080h,0(SP)       ; –d/(c^2 + d2)
        MOV     @SP+,2*FPL–2(SP)  ; Store imaginary part
        MOV     @SP+,2*FPL–2(SP)  ; to final location
        .if     DOUBLE=1
        MOV     @SP+,2*FPL–2(SP)
        .endif
        MOV     SP,RPARG          ; To copy of c^2 + d^2
        MOV     2*FPL(SP),RPRES   ; To (c + jd)
        CALL    #FLT_DIV          ; c/(c^2 + d2)
;
; Prepare the interface to the multiplication and call it:
; RPARG points to 1/(c + jd)   yet made by FLT_DIV
; RPRES points to   (a + jb)
;
        MOV     2*FPL+2(SP),RPRES   ; address of (a + jb)
        CALL    #CMPLX_MUL        ; (a +jb) x 1/(c +jd)
        MOV     #FPL,HELP         ; Result to final location
CDIVL   MOV     @SP+,2*FPL+4(SP)
        DEC     HELP
        JNZ     CDIVL
;
        JMP     CMPLX_RET         ; To common housekeeping
;
; Complex Multiplication: (a + jb)x(c + jd). @RPRES x @RPARG
;(a + jb)x(c + jd) = ac + jad + jbc – bd
; Stack Usage: 24 bytes (.DOUBLE)   18 bytes (.FLOAT)
```

```
;
CMPLX_MUL PUSH    RPRES             ; Save pointer to (a + jb)
          PUSH    RPARG             ; Save pointer to (c + jd)
;
; real Part ac – bd
;
          SUB     #FPL,SP           ; Allocate result space for a x c
          CALL    #FLT_MUL          ; a x c
          SUB     #FPL,SP           ; Allocate result space for b x d
          MOV     2*FPL(SP),RPARG   ; To c + jd
          MOV     2*FPL+2(SP),RPRES ; To a + jb
          ADD     #FPL,RPARG        ; To jd
          ADD     #FPL,RPRES        ; To jb
          CALL    #FLT_MUL          ; jb x jd = –bd
          ADD     #FPL,RPRES        ; To a x c
          CALL    #FLT_SUB          ; (a x c) – (b x d)
          MOV     @SP,FPL(SP)       ; Store ac – bd
          MOV     2(SP),FPL+2(SP)
          .if     DOUBLE=1
          MOV     4(SP),FPL+4(SP)
          .endif
;
; Imaginary Part j(ad + bc)
;
          MOV     2*FPL(SP),RPARG   ; To c + jd
          MOV     2*FPL+2(SP),RPRES ; To a + jb
          ADD     #FPL,RPARG        ; To jd
          CALL    #FLT_MUL          ; a x d
          SUB     #FPL,SP           ; Allocate result space for b x c
          MOV     3*FPL(SP),RPARG   ; To c + jd
          MOV     3*FPL+2(SP),RPRES ; To a + jb
          ADD     #FPL,RPRES        ; To b
          CALL    #FLT_MUL          ; b x c
          ADD     #FPL,RPARG        ; To a x d
          CALL    #FLT_ADD          ; ad + bc
          MOV     @SP+,4*FPL+4(SP)  ; To imaginary result
```

```
        MOV     @SP+,4*FPL+4(SP)
        .if     DOUBLE=1
        MOV     @SP+,4*FPL+4(SP)
        .endif
        ADD     #FPL,SP          ; To real result
        MOV     @SP+,FPL+4(SP)   ; To real result
        MOV     @SP+,FPL+4(SP)
        .if     DOUBLE=1
        MOV     @SP+,FPL+4(SP)
        .endif
;
; RPARG, RPRES and SP point to the real part of the result
; on the TOS
;
CMPLX_RET ADD   #4,SP            ; Skip pointers
CMPLX_RT MOV    SP,RPARG
        ADD     #2,RPARG
        MOV     RPARG,RPRES
        RET
```

EXAMPLE: The complex number at address CN1 is divided by a complex number at address CN2. The result (on TOS) is added to a RAM value CST3 and stored there.

```
;
        SUB     #2*FPL,SP        ; Allocate result space
        ....
        MOV     #CN1,RPRES       ; Address of CN1
        MOV     #CN2,RPARG       ; Address of CN2
        CALL    #CMPLX_DIV       ; CN1/CN2 –> TOS
        MOV     #CST3,RPARG      ; Address of CST3
        CALL    #CMPLX_ADD       ; CN1/CN2 + CST3 –> TOS
        MOV     @RPARG+,CST3     ; Store result in CST3
        MOV     @RPARG+,CST3+2   ; Save result space
        MOV     @RPARG+,CST3+4
        MOV     @RPARG+,CST3+6
        .if     DOUBLE=1
        MOV     @RPARG+,CST3+8
```

```
MOV     @RPARG+,CST3+10
.endif
...                         ; Continue with complex calc.
ADD     #2*FPL,SP           ; Terminate complex calc.
...
```

5.6.10.13 Trigonometric and Hyperbolic Functions

Four subroutines are shown for the calculation of the sine, cosine, hyperbolic sine, and hyperbolic cosine. All four subroutines use the same kernel, only the initialization part is different for each of them. Expansion in series is used for the calculation. The formulas are (X is expressed in radians):

Sine function:

$$\sin X = \sum_{1}^{n} \frac{X^{2n-1}}{(2n-1)!} \times (-1)^{n+1}$$

Cosine function:

$$\cos X = \sum_{0}^{n} \frac{X^{2n}}{(2n)!} \times (-1)^{n}$$

Hyperbolic sine function:

$$\sinh X = \sum_{1}^{n} \frac{X^{2n-1}}{(2n-1)!}$$

Hyperbolic cosine function:

$$\cosh X = \sum_{0}^{n} \frac{X^{2n}}{(2n)!}$$

The number range for X is $\pm 2\pi$ for all four functions. Outside of this range, the error increases relatively fast due to the fast growing terms of the sequences (X^{2n} and X^{2n+1}).

If the trigonometric functions have to be calculated for numbers outside of this range, two possibilities exist:

- Sine and cosine: addition or subtraction of 2π until the number X is back in the range $\pm 2\pi$. The subroutine FLT_RNG can be used for this purpose.

- Hyperbolic sine and cosine: increase of the software variable Nmax (normally 30.0, see the following software) that defines the number of iterations. If this variable is changed to 120.0 (60 iterations), the deviations in the range 10^{-12} (.DOUBLE format) or. 10^{-6} (.FLOAT format) are possible for X input values up to 65. The input number that delivers results near the maximum numbers $\pm 10^{38}$.

Note:

The following subroutines are optimized for ROM space and accuracy, but not for run time. They are not intended as part of a floating point package, but as a place to begin if needed.

The calculation errors for the trigonometric functions are shown in the following table. They indicate absolute errors; the difference to the correct values.

Table 5–14. Errors of the Trigonometric Functions

Angle X	.FLOAT		.DOUBLE	
	Sin	Cos	Sin	Cos
0	0	-20×10^{-9}	0	0
$\pm\pi/2$	-21×10^{-9}	-64×10^{-9}	0	0
$\pm\pi$	3.8×10^{-9}	-225×10^{-9}	0	0
$\pm 2\pi$	-1.3×10^{-6}	2×10^{-6}	0	-80×10^{-12}

The errors of the hyperbolic functions are shown in the following table. They indicate relative errors. The differences to the correct values are related to the correct values.

Table 5–15. Errors of the Hyperbolic Functions

Angle X	.FLOAT		.DOUBLE	
	Hyperbolic Sine	Hyperbolic Cosine	Hyperbolic Sine	Hyperbolic Cosine
0	0	0	0	0
$\pm\pi/2$	85×10^{-9}	160×10^{-9}	-126×10^{-12}	255×10^{-12}
$\pm\pi$	55×10^{-9}	100×10^{-9}	-242×10^{-12}	-474×10^{-12}
$\pm 2\pi$	34×10^{-9}	218×10^{-9}	-153×10^{-12}	-309×10^{-12}

Calculation times (Nmax = 30.0: 15 iterations). The number of cycles is the same one for all four functions:

.FLOAT with hardware multiplier: 18000 cycles

.FLOAT without hardware multiplier: 26000 cycles

.DOUBLE with hardware multiplier: 28000 cycles

.DOUBLE without hardware multiplier: 42000 cycles

```
; Sine, Cosine, Hyperbolic Sine, Hyperbolic Cosine of X (radians)
;
; Call:  MOV      #addressX,RPARG ; RPARG points to operand X
;        CALL     #FPP_xxx          ; Call the function
;        ...                                ; RPARG, RPRES, SP point to result
;
; Range: –2xPi < X < +2xPi          for larger numbers FAST loss of
; accuracy
; Stack allocation: (4 x FPL + 4) words are needed (Basic FPP
; Functions are included)
;
; Initialization for the trigonometric and hyperbolic functions
;  +--------------+-------+-------+--------+--------+
;  | INIT         | sin X | cos X | sinh X | cosh X |
;  +--------------+-------+-------+--------+--------+
;  | Sign Mask    | 080h  |  080h | 000h   |  000h  |
;  | n            | 1.0   |  0.0  | 1.0    |  0.0   |
;  | Series Term  |   X   |  1.0  |   X    |  1.0   |
;  | Result Area  |   X   |  1.0  |   X    |  1.0   |
;  +--------------+-------+-------+--------+--------+
;
FPL      .equ     (ML/8)+1          ; Length of FPP numbers (bytes)
;
; Floating Point Sine Function: Result on TOS = SIN(@RPARG)
; Prepare the stack with the initial constants
;
FLT_SIN  PUSH     #80h              ; Sign mask (toggle)
         JMP      SINc
;
; Hyperbolic Sine Function: Result on TOS = SINH(@RPARG)
;
FLT_SINH PUSH     #00h              ; Sign mask (always pos.)
SINc     PUSH     #0                ; n: 1
         .if DOUBLE=1
```

```
        PUSH    #0
        .endif
        PUSH    #08000h         ; .FLOAT 1.0
;
        .if DOUBLE=1
        PUSH    4(RPARG)        ; Series term: X
        .endif
        PUSH    2(RPARG)
        PUSH    @RPARG
        JMP     TRIGCOM         ; To common part
;
; Floating Point Cosine Function: Result on TOS = COS(@RPARG)
; Prepare the stack with the initial constants
;
FLT_COS PUSH    #80h            ; Sign mask (toggle)
        JMP     COSc
;
; Hyperbolic Cosine Function: Result on TOS = COSH(@RPARG)
;
FLT_COSH PUSH   #00h            ; Sign mask (always pos.)
COSc    PUSH    #               ; n: 0
        .if DOUBLE=1
        PUSH    #0
        .endif
        PUSH    #00h            ; .FLOAT 0.0
;
        .if DOUBLE=1
        PUSH    #0              ; Series term: 1.0
        .endif
        PUSH    #0
        PUSH    #08000h         ; .FLOAT 1.0
;
; Common part for sin X, cos X, sinh X and cosh X
; The functions are realized by expansions in series
;
TRIGCOM .equ    $
```

```
        .if DOUBLE=1
        PUSH    4(RPARG)          ; Push X onto stack (gets X^2)
        .endif
        PUSH    2(RPARG)          ; X^2 is calculated once
        PUSH    @RPARG
        MOV     RPARG,RPRES       ; Both pointers to X
        CALL    #FLT_MUL          ; X^2 to actual stack
;
        ADD     #FPL,RPARG        ; Copy series term to result space
        MOV     @RPARG+,3*FPL+4(SP)       ; is X or 1.0
        MOV     @RPARG+,3*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,3*FPL+8(SP)
        .endif
        SUB     #FPL,SP           ; Result space for calculations
        MOV     SP,RPRES
;
; The actual series term is multiplied by X^2/(n+1)x(n+2) to
; get the next series term
;
TRIGLOP MOV     #FLT2,RPARG       ; Address of .FLOAT 2.0
        ADD     #3*FPL,RPRES      ; Address n
        CALL    #FLT_ADD          ; n + 2
        MOV     @RPARG+,3*FPL(SP) ; (n+2) -> n
        MOV     @RPARG+,3*FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,3*FPL+4(SP)
        .endif
;
; Build (n+1)x(n+2) for next term. (n+2)^2 - (n+2) = (n+1)x(n+2)
;
        MOV     RPRES,RPARG       ; Both point to (n+2)
        CALL    #FLT_MUL          ; (n+2)^2
        ADD     #3*FPL,RPARG      ; Point to old n
        CALL    #FLT_SUB          ; (n+2)^2 -(n+2) = (n+1)x(n+2)
;
```

```
; The series term is divided by (n+1)x(n+2)
;
        ADD     #2*FPL,RPRES      ; Point to series term
        CALL    #FLT_DIV          ; Series term/(n+1)x(n+2)
        ADD     #FPL,RPARG        ; Point to x^2
        CALL    #FLT_MUL          ; ST x X^2/(n+1)x(n+2)
        JN      TRIGERR           ; Error, status in SR and HELP
;
; The sign of the new series term is modified dependent on
; the sign mask.  0: always positive   080h: alternating + –
;
        XOR     4*FPL(SP),0(SP)   ; Modify sign with sign mask
        MOV     @RPARG+,2*FPL(SP) ; Save new series term
        MOV     @RPARG+,2*FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,2*FPL+4(SP)
        .endif
        ADD     #3*FPL+4,RPARG    ; Point to result area
        CALL    #FLT_ADD          ; Old sum + new series term
        MOV     @RPARG+,4*FPL+4(SP)       ; Result to result area
        MOV     @RPARG+,4*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,4*FPL+8(SP)
        .endif
;
; Check if enough iterations are made: iterations = Nmax/2
;
        CMP     Nmax,3*FPL(SP)    ; Compare n with Nmax
        JLO     TRIGLOP           ; Only MSBs are used
;
; Expansion in series done. Error indication (if any) in HELP
; The completion part of the FPP is used
;
TRIGERR ADD     #4*FPL+2,SP       ; Housekeeping: free stack
        BR      #FLT_END          ; To completion part of FPP
;
```

```
        .if DOUBLE=1
FLT2    .DOUBLE 2.0              ; Constant 2.0
        .else
FLT2    .FLOAT   2.0
        .endif
Nmax    .FLOAT   30.0            ; Iterations x 2 (MSBs used only)
```

5.6.10.14 Other Trigonometric and Hyperbolic Functions

With the previous calculated four functions (sin, cosin, hyperbolic sin, and hyperbolic cosin), five other important functions can be calculated: tangent, cotangent, hyperbolic tangent, hyperbolic cotangent, and exponential functions.

$$\tan X = \frac{\sin X}{\cos X} \qquad \cot X = \frac{\cos X}{\sin X} \qquad \tanh X = \frac{\sinh X}{\cosh X} \qquad \coth X = \frac{\cosh X}{\sinh X}$$

$$e^X = \sum \frac{X^n}{n!} = \sinh X + \cosh X$$

To calculate one of the five functions, the two functions it consists of are calculated and combined.

The errors of the five functions can be calculated with the errors of the two functions used and are shown in Table 5–14 and Table 5–15:

- tan X, cot X, tanh X and coth X: the resulting error is the difference of the two errors
- exp X: the resulting error is the sum of the two errors

Calculation times (Nmax = 30.0: 15 iterations). The number of cycles is the same one for all five functions:

.FLOAT with hardware multiplier:	36000 cycles
.FLOAT without hardware multiplier:	52000 cycles
.DOUBLE with hardware multiplier:	56000 cycles
.DOUBLE without hardware multiplier:	84000 cycles

The same software kernel is used for all five functions. The number contained in R4 decides which function is executed. The range for all five functions is $\pm 2\pi$. For larger numbers a relatively fast loss of accuracy occurs.

```
; Tangent of X (radians)
;
; Call:  MOV      #addressX,RPARG   ; RPARG points to operand X
;        CALL     #FLT_TAN          ; Call the tangent function
;        ...                        ; RPARG, RPRES, SP point to result
;
FLT_TAN  CLR      R4                ; Offset for tan X
         JMP      TRI_COM1          ; Go to common handler
;
; Cotangent of X (radians)
;
; Call:  MOV      #addressX,RPARG ; RPARG points to operand X
;        CALL     #FLT_COT          ; Call the cotangent function
;        ...                        ; RPARG, RPRES, SP point to result
;
FLT_COT  MOV      #2,R4             ; Offset for cot X
         JMP      TRI_COM1          ; Go to common handler
;
; Hyperbolic Tangent of X (radians)
;
; Call:  MOV      #addressX,RPARG ; RPARG points to operand X
;        CALL  #FLT_TANH            ; Call the hyperbolic tangent
;        ...                        ; RPARG, RPRES, SP point to result
;
FLT_TANH MOV      #4,R4             ; Offset for tanh X
         JMP      TRI_COM1          ; Go to common handler
;
; Hyperbolic Cotangent of X (radians)
;
; Call:  MOV      #addressX,RPARG ; RPARG points to address of X
;        CALL     #FLT_COTH         ; Call the hyperbolic cotangent
;        ...                        ; RPARG, RPRES, SP point to result
;
FLT_COTH MOV      #6,R4             ; Offset for coth X
         JMP      TRI_COM1          ; Go to common handler
;
```

```
; Exponential function of X (ex)
;
; Call:  MOV      #addressX,RPARG ; RPARG points to operand X
;        CALL     #FLT_EXP        ; Call the exponential function
;        ...                      ; RPARG, RPRES, SP point to result
;
FLT_EXP MOV      #8,R4           ; Offset for exp X
;
; Common Handler for tan, cot, tanh, coth and exponent function
; Range: –2xPi < X < +2xPi. For larger numbers FAST loss of
; accuracy
;
TRI_COM1 .equ $
         MOV      @RPARG+,2(SP)     ; Copy X to result space
         MOV      @RPARG+,4(SP)
         .if      DOUBLE=1
         MOV      @RPARG,6(SP)
         .endif
         SUB      #FPL,SP           ; Allocate new result space
         SUB      #4,RPARG          ; Point to X again
         CALL     FT1(R4)           ; Calculate 1st function
         JN       TERR2             ; Error: error code in HELP
         SUB      #FPL,SP           ; Allocate cosine result space
         ADD      #FPL+2,RPARG      ; Point to X
         CALL     FT2(R4)           ; Calculate 2nd function
         ADD      #FPL,RPRES        ; Point to result of 1st function
         CALL     FT3(R4)           ; 1st result .OP. 2nd result
         MOV      @SP+,2*FPL(SP)    ; Final result to result area
         MOV      @SP+,2*FPL(SP)
         .if      DOUBLE=1
         MOV      @SP+,2*FPL(SP)
         .endif
TERR2    ADD      #FPL,SP           ; Skip 1st result
         BR       #FLT_END          ; Error code in HELP
;
FT1      .word    FLT_SIN           ; tan = sin/cos    1st function
```

```
        .word   FLT_COS         ; cot = cos/sin
        .word   FLT_SINH        ; tanh = sinh/cosh
        .word   FLT_COSH        ; coth = cosh/sinh
        .word   FLT_COSH        ; exp  = cosh + sinh
;
FT2     .word   FLT_COS         ; tan = sin/cos    2nd function
        .word   FLT_SIN         ; cot = cos/sin
        .word   FLT_COSH        ; tanh = sinh/cosh
        .word   FLT_SINH        ; coth = cosh/sinh
        .word   FLT_SINH        ; exp  = cosh + sinh
;
FT3     .word   FLT_DIV         ; tan = sin/cos    3rd function
        .word   FLT_DIV         ; cot = cos/sin
        .word   FLT_DIV         ; tanh = sinh/cosh
        .word   FLT_DIV         ; coth = cosh/sinh
        .word   FLT_ADD         ; exp  = cosh + sinh
```

If the argument X for trigonometric functions is outside of the range $\pm 2\pi$ then the subroutine FLT_RNG may be used. The subroutine moves the angle X into the range $\pm\pi$.

```
; Subroutine FLT_RNG moves angle X into the range –Pi < X < +Pi
;
; Call:  MOV     #addressX,RPARG  ; RPARG points to operand X
;        CALL    #FLT_RNG         ; Call the function
;        ...                      ; RPARG, RPRES, SP point to result
;
; Range: –100xPI < X < +100xPI     loss of accuracy increases with X
;
FLT_RNG  PUSH    @RPARG           ; Save sign of X on stack
         AND     #080h,0(SP)      ; Only sign remains
         SUB     #FPL,SP          ; Reserve space for 2^n x Pi
         .if DOUBLE=1
         PUSH    4(RPARG)         ; X on stack
         .endif
         PUSH    2(RPARG)
         PUSH    @RPARG
         BIC     #080h,0(SP)      ; |X| remains
```

```
FR1     MOV     FLT2PI,FPL(SP)    ; 2xPi to stack
        MOV     FLT2PI+2,FPL+2(SP)
        .if DOUBLE=1
        MOV     FLT2PI+4,FPL+4(SP)
        .endif
        CMP     @SP,FLTPI         ; Pi – |X|
        JHS     FR2               ; Pi > |X|: range process done
;
; Successive approximation by subtracting 2^n x2Pi
;
FR3     INC.B   FPL+1(SP)         ; 2Pi x 2
        CMP     @SP,FPL(SP)       ; 2^n x 2Pi – |X|
        JLO     FR3               ; 2^n x 2Pi < |X|
        DEC.B   FPL+1(SP)         ; 2^n x 2Pi > |X| divide by 2
        MOV     SP,RPRES          ; Address |X|
        MOV     SP,RPARG
        ADD     #FPL,RPARG        ; Address 2^n x 2Pi
        CALL    #FLT_SUB          ; |X| – 2^n x 2Pi
        JMP     FR1               ; Check if in range now
;
; Move X (now between –Pi and +Pi) to old result space
;
FR2     XOR     2*FPL(SP),0(SP)   ; Correct sign of X
        MOV     @SP+,2*FPL+2(SP)  ; Result to old RS
        MOV     @SP+,2*FPL+2(SP)
        .if DOUBLE=1
        MOV     @SP+,2*FPL+2(SP)
        .endif
        ADD     #FPL+2,SP         ; To return address of FLT_RNG
        BR      #FLT_END
;
        .if DOUBLE=1
FLTPI   .DOUBLE 3.141592653589793          ; Pi
FLT2PI  .DOUBLE 3.141592653589793*2        ; 2xPi
        .else
FLTPI   .FLOAT  3.141592653589793          ; Pi
```

```
FLT2PI   .FLOAT   3.141592653589793*2        ; 2xPi
         .endif
```

5.6.10.15 Faster Approximations for Trigonometric Functions

If the calculation times of the previous iterations are too long and the high accuracy is not needed (e.g. for the calculation of pulse widths for PWM), tables or cubic equations can be used. The table method is described in the MSP430 Software User's Guide (literature number SLAUE11).

With the following four definition points, a cubic approximation to the sin curve is made. The range is 0 to $\pi/2$. All other angles must be adapted to this range.

X1:= 0.0000000000	SIN X1:= 0.0000000000	(0°)
X2:= 0.3490658504	SIN X2:= 0.3420201433	(20°)
X3:= 1.2217304760	SIN X3:= 0.9396926208	(70°)
X4:= 1.5707963270	SIN X4:= 1.0000000000	(90°)

The resulting multiplication factors are:

SIN X = –0.11316874 X^3–0.063641170 X^2 +1.01581976 X

The following results and errors are obtained with the previous factors:

X = 0 ,	SIN X = 0.000000000	0.00%	(0°)
X = $\pi/12$	SIN X = 0.259548457	+0.28%	(15°)
X = $\pi/6$	SIN X = 0.498189297	–0.36%	(30°)
X:=$\pi/4$	SIN X = 0.703738695	–0.47%	(45°)
X:=$\pi/3$	SIN X = 0.864012827	–0.23%	(60°)
X:= $5\pi/12$	SIN X = 0.966827870	+0.09%	(75°)
X:=$\pi/2$	SIN X = 1.000000000	0.00%	(90°)

The error of the previous approximation is within $\pm$0.5% from 0 to 2π. Calculation times:

.FLOAT with hardware multiplier:	880 cycles
.FLOAT without hardware multiplier:	1600 cycles
.DOUBLE with hardware multiplier:	1150 cycles
.DOUBLE without hardware multiplier:	2550 cycles

```
; Sine Approximation: Sin X = A3xX^3 + A2xX^2 + A1xX + A0
; Input range for X: 0 =< X =< Pi/2
; The terms Ax are stored in a table starting with the cubic term
;
        MOV     #X,RPARG          ; Address of X (radians)
        MOV     #A3,R4            ; Address of cubic term for sine
        CALL    #HORNER           ; Cubic approximation
        ...                       ; Use approximated value Sin X
HORNER  .equ    $                 ; R4 points to cubic term
        .if     DOUBLE=1          ; Store X on stack
        PUSH    4(RPARG)          ; for later use
        .endif
        PUSH    2(RPARG)
        PUSH    @RPARG
        SUB     #FPL,SP           ; Locate new result space
        MOV     R4,RPRES          ; Address cubic term A3
        CALL    #FLT_MUL          ; XxA3
        ADD     #FPL,R4           ; Address quadratic term A2
        MOV     R4,RPRES          ;
        CALL    #FLT_ADD          ; XxA3 + A2
        ADD     #FPL,RPARG        ; to X
        CALL    #FLT_MUL          ; X^2xA3 + XxA2
        ADD     #FPL,R4           ; Address linear term A1
        MOV     R4,RPRES
        CALL    #FLT_ADD          ; X^2xA3 + XxA2 + A1
        ADD     #FPL,RPARG        ; to X
        CALL    #FLT_MUL          ; X^3xA3 + X^2xA2 + XxA1
        ADD     #FPL,R4           ; Address constant term A0
        MOV     R4,RPRES
        CALL    #FLT_ADD          ; X^3xA3 + X^2xA2 + XxA1 + A0
        MOV     @SP+,2*FPL(SP)    ; Copy to result area
        .if     DOUBLE=1
        MOV     @SP+,2*FPL(SP)
        .endif
        MOV     @SP+,2*FPL(SP)
        ADD     #FPL,SP           ; SP to return address
```

```
        BR      #FLT_END          ; Use standard FPP return
;
; Multiplication factors for the Sine generation (0 to Pi/2)
; SIN X = –0.11316874 X3–0.063641170 X^2 +1.01581976 X
        .if     DOUBLE=1
A3      .DOUBLE –0.11316874       ; cubic term
A2      .DOUBLE –0.063641170      ; quadratic term
A1      .DOUBLE 1.01581976        ; linear term
A0      .DOUBLE 0.0               ; constant term
        .else
A3      .FLOAT  –0.11316874       ; cubic term
A2      .FLOAT  –0.063641170      ; quadratic term
A1      .FLOAT  1.01581976        ; linear term
A0      .FLOAT  0.0               ; constant term
        .endif
```

Note:

The HORNER algorithm (used previously) can be used for several other purposes. It is only necessary to load the register R4 with the starting address of the appropriate block containing the factors (address A3 with the previous example).

5.6.10.16 The Natural Logarithm Function

The natural logarithm of a number X is calculated with the following formula:

$$ln\ X\ =\ \sum_{1}^{n} \frac{(X-1)^n}{n} \times (-1)^{n-1}$$

The number range of X for the natural logarithm contains all positive numbers except zero. Values of X less than or equal to zero return the largest negative number (-3.4×10^{38}) and the N bit set as an error indication.

The calculation errors for the natural logarithm function are shown in the following table. They indicate relative errors. The errors of the .DOUBLE routine are estimated: no logarithm values greater than 12 digits were available. Table 5–16 shows the relative large errors – especially for the .FLOAT format – for input values X very near to 1.0. This is due to the (X – 1) operation necessary for the calculation. Algorithms used should avoid the calculation of the logarithm of numbers very close to 1.0.

Table 5–16. Relative Errors of the Natural Logarithm Function

X	.FLOAT	.DOUBLE	Comment
2.938736×10^{-39}	-1.5×10^{-7}	-4.5×10^{-12}	Smallest FPP number
1.00	0	0	
1.0001	-3.5×10^{-5}	-1.4×10^{-8}	Missing resolution
1.00001	$+5.2 \times 10^{-3}$	-8×10^{-8}	at results near zero
1.000001	$+6.7 \times 10^{-2}$	$+2.4 \times 10^{-7}$	See above
1.95	$+1.5 \times 10^{-7}$	$+5 \times 10^{-12}$	
10^{6}	$+3.6 \times 10^{-8}$	$+1.5 \times 10^{-11}$	
10^{12}	$+3.6 \times 10^{-8}$	$+4.5 \times 10^{-12}$	
3.402823×10^{38}	$+1.5 \times 10^{-7}$	$+4.5 \times 10^{-12}$	Largest FPP number

Calculation times:

.FLOAT with hardware multiplier:	13000 cycles	13 iterations
.FLOAT without hardware multiplier:	16000 cycles	
.DOUBLE with hardware multiplier:	34000 cycles	22 iterations
.DOUBLE without hardware multiplier:	43000 cycles	

```
; Natural Logarithm Function:      Result on TOS = LN(@RPARG)
;
; Call:  MOV      #addressX,RPARG ; RPARG points to operand X
;
;        CALL     #FLT_LN         ; Call the function lnX
;        ...                      ; RPARG, RPRES and SP point to lnX
;
; Range: +2.9x10^-38 < X < +3.4x10^38
;
; Errors: X = 0:  N = 1, C = 1, Z = 0       Result: -3.4E38
;         X < 0:  N = 1, C = 1, Z = 0       Result: -3.4E38
;
; Stack usage: 3 x FPL + 6 bytes
;
FLT_LN   PUSH     #0              ; N binary (divisor, power)
         .if DOUBLE=1
         PUSH     4(RPARG)        ; Push X onto stack
```

```
        .endif
        PUSH    2(RPARG)          ;
        PUSH    @RPARG            ;
;
; Check for the legal range of X: 0 < X
;
        MOV     #FLT0,RPRES       ; Check valid range: 0 < X
        CALL    #FLT_CMP
        JHS     LNNEG             ; X is negative
;
; If X is 1.0 then 0.0 is used for the result
;
        MOV     #FLT1,RPRES       ; Check if X= 1
        CALL    #FLT_CMP
        JEQ     LN1P0             ; X is 1: result is 0.0
;
; The exponent of X is multiplied with ln2. Then ln1.5 is added
; to correct the division by 1.5. Result is base for final result
;
        SUB     #FPL,SP           ; Reserve working space
        MOV.B   1(RPARG),HELP     ; Copy exponent of X
        XOR     #80h,HELP         ; Correct sign of exponent
        SXT     HELP                      ;
        MOV     HELP,0(SP)
        MOV     SP,RPARG
        CALL    #CNV_BIN16        ; Exponent to FP format
        MOV     #FLN2,RPARG       ; To ln2
        CALL    #FLT_MUL          ; exp x ln2
        MOV     #FLN1P5,RPARG     ; To ln1.5
        CALL    #FLT_ADD          ; exp x ln2 + ln1.5
        MOV     @RPARG+,2*FPL+4(SP)       ; To result area
        MOV     @RPARG+,2*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,2*FPL+8(SP)
        .endif
;
```

```
; The mantissa of X is converted into the range –0.33 to +0.33
; to get fast convergion
;
        ADD     #FPL,SP          ; Back to X
        MOV     SP,RPRES         ; RPRES points to X
        MOV.B   #80h,1(SP)       ; 1.0 =< X < 2.0
        MOV     #FLT1P5,RPARG    ; To .FLOAT 1.5
        CALL    #FLT_DIV         ; 2/3 =< X < 4/3
        MOV     #FLT1,RPARG      ; To .FLOAT 1.0
        CALL    #FLT_SUB         ; –1/3 =< X < +1/3
;
        .if DOUBLE=1             ;
        PUSH    #0               ; 1.0 to X^N area
        .endif
        PUSH    #0
        PUSH    FLT1
;
        .if DOUBLE=1             ; N (FLT1.0) on stack
        PUSH    #0
        .endif
        PUSH    #0
        PUSH    FLT1
        SUB     #FPL,SP          ; Working area
;
LNLOP   .equ $
        MOV     SP,RPRES
        ADD     #2*FPL,RPRES     ; To X^N
        MOV     SP,RPARG
        ADD     #3*FPL,RPARG     ; To X
        CALL    #FLT_MUL         ; X^(N+1)
        MOV     @RPARG+,2*FPL(SP) ; New X^(N+1) –> X^N
        MOV     @RPARG+,2*FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,2*FPL+4(SP)       ; RPARG points to N
        .endif
        CALL    #FLT_DIV         ; X^N/N
```

```
        INC     4*FPL(SP)         ; Incr. binary N
        BIT     #1,4*FPL(SP)      ; N even?
        JNZ     LN1
        XOR     #80h,0(SP)        ; Yes, change sign of X^N/N
;
LN1     ADD     #4*FPL+4,RPARG            ; Point to result area
        CALL    #FLT_ADD                  ; Old result + new one
        MOV     @RPARG+,4*FPL+4(SP)       ; New result to result area
        MOV     @RPARG+,4*FPL+6(SP)
        .if DOUBLE=1
        MOV     @RPARG+,4*FPL+8(SP)
        .endif
;
; Float N is incremented
;
        MOV     #FLT1,RPARG       ; To .FLOAT 1.0
        ADD     #FPL,RPRES        ; To N
        CALL    #FLT_ADD
        MOV     @RPARG+,FPL(SP)   ; N+1 to N area
        MOV     @RPARG+,FPL+2(SP)
        .if DOUBLE=1
        MOV     @RPARG+,FPL+4(SP)
        .endif
;
; Check if enough iterations are made
;
        CMP     #LNIT,4*FPL(SP)   ; Compare with nec. iterations
        JLO     LNLOP             ; HELP = 0
;
        ADD     #4*FPL+2,SP       ; Housekeeping: free stack
LNE     BR      #FLT_END          ; To completion. Error in HELP
;
LN1P0   ADD     #FPL+2,SP         ; X = 1: result = 0
        BR      #RES0
LNNEG   ADD     #FPL+2,SP         ; X <= 0: –3.4E38 result
        MOV     #0FFFFh,2(SP)     ; MSBs negative
```

```
        BR      #DBL_OVERFLOW
        .if DOUBLE=1
FLT0    .DOUBLE 0.0             ; 0.0
FLT1    .DOUBLE 1.0             ; 1.0
FLT1P5  .DOUBLE 1.5             ; 1.5
FLN1P5  .DOUBLE 0.405465108107  ; ln1.5
FLN2    .DOUBLE 0.6931471805599 ; ln2.0
LNIT    .equ    22              ; Number of iterations
        .else
FLT0    .FLOAT  0.0
FLT1    .FLOAT  1.0
FLT1P5  .FLOAT  1.5
FLN1P5  .FLOAT  0.405465108107
FLN2    .FLOAT  0.6931471805599
LNIT    .equ    13
        .endif
```

To calculate the logarithm of X based to the number 10 the following sequence may be used:

```
        MOV     #addressX,RPARG  ; Address of X
        CALL    #FLT_LN           ; Calculate lnX
        MOV     #FLTMOD,RPARG
        CALL    #FLT_MUL         ; lnX/ln10 = logX
        ...                      ; logX on TOS
        .if     DOUBLE=0
FLTMOD  .FLOAT  0.4342944819033  ; log10/ln10
        .else
FLTMOD  .DOUBLE 0.4342944819033  ; log10/ln10
        .endif
```

5.6.10.17 The Exponential Function

The exponential function e^X is calculated. The number range of X is: $-88.72 \leq X \leq +88.72$. Values of X outside of this range return zero ($X < -88.72$) respective the largest positive number ($+3.4 \times 10^{38}$) and the N bit set as an error indication.

The calculation errors for the exponential function are shown in the following table. They indicate relative errors.

Table 5–17. Errors of the Exponential Function

X	.FLOAT	.DOUBLE	Result
–88.72	-9.4×10^{-7}	-4.5×10^{-11}	2.9470911×10^{-39}
–12.3456	-1.7×10^{-8}	-1.5×10^{-11}	$4.348846154014\times10^{-6}$
0.0	0	0	1.0
2^{-41}	-4.5×10^{-13}	-4.5×10^{-13}	1.0
2^{-25}	-30×10^{-9}		$1.0+29.8\times10^{-9}$
+88.72	-2.8×10^{-6}	-4.5×10^{-11}	Most positive FPP number

Calculation times:

.FLOAT with hardware multiplier: 3200 cycles

.FLOAT without hardware multiplier: 5100 cycles

.DOUBLE with hardware multiplier: 4500 cycles

.DOUBLE without hardware multiplier: 7500 cycles

```
; Exponential Function: e^X.        Result on TOS = e^(@RPARG)
;
; Call:  MOV      #addressX,RPARG   ; RPARG points to operand X
;        CALL     #FLT_EXP          ; Call the exp. function
;        ...                        ; RPARG, RPRES, SP point to result
;
; Range: -88.72 < X < +88.72
;
; Errors:
; X > +88.72: N = 1, C = 1, Z = 1  Result: +3.4E38
; X < -88.72: N = 1, C = 0, Z = 0  Result: 0.0 if SW_UFLOW = 1
;             N = 0, C = x, Z = x  Result: 0.0 if SW_UFLOW = 0
;
; Stack usage:    3 x FPL + 4 bytes
;
FLT_EXP  MOV      @RPARG+,2(SP)     ; Copy X to result area
         MOV      @RPARG+,4(SP)
         .if DOUBLE=1
         MOV      @RPARG,6(SP)
         .endif
```

```
;
; Check if X is inside limits: –88.72 < X < +88.72  (8631h,7218h)
;
         MOV      2(SP),COUNTER     ; MSBs, exp and sign of X
         BIC      #080h,COUNTER     ; |X|
         CMP      #08631h,COUNTER   ; |X| > 88.72? ln3.4x10^38=88.72
         JLO      EXP_L3            ; |X| is in range
         JNE      EXP_RNGOUT        ; X > 88.72 .or. X < –88.72: error
         CMP      #07217h,4(SP)     ; Check LSBs
         JHS      EXP_RNGOUT        ; LSBs show: |X| > 88.72
;
; Prepare exponent of result: N = X/ln2  (rounded)
;
EXP_L3   MOV      SP,RPRES
         SUB      #FPL,SP           ; New working area
         ADD      #2,RPRES          ; To X (result area)
         MOV      #FLTLN2I,RPARG    ; To 2/ln2 (allows MPY)
         CALL     #FLT_MUL          ; 2 x X/ln2
         CALL     #CNV_FP_BIN       ; 2 x X/ln2 –> binary
         .if      DOUBLE=1
         SUB      #2,SP             ; LSBs contain N
         ADD      #FPL–2,RPARG      ; To N
         .else
         ADD      #FPL,RPARG        ; To binary N
         .endif                     ; N is at correct place yet
         RRA      @RPARG            ; /2 for rounding
         JNC      EXPL1             ; No carry, no rounding
         TST      0(RPARG)          ; Sign of N
         JN       EXPL1
         INC      0(RPARG)          ; Round N
EXPL1    CALL     #CNV_BIN16        ; N –> FPP format Xn
;
; Calculation of g: g = X – Xn*(C1 + C2)
;
         MOV      #EXPC,RPARG       ; C1 + C2
         CALL     #FLT_MUL          ; Xn*(C1 + C2)
```

```
        ADD     #FPL+4,RPRES      ; To X
        CALL    #FLT_SUB          ; g = X – Xn*(C1 + C2)
;
; Calculation of mantissa R(g):
; R(g) = 0.5 + g*P(z)/(Q(z) – g*P(z))
;
        SUB     #FPL,SP           ; Area for z = g^2
        CALL    #FLT_MUL          ; z = g^2
;
; Calculation of g*P(z): g*P(z) = g*(p1*z + p0)
;
        SUB     #FPL,SP           ; Area for g*P(z)
        MOV     #EXPP1,RPARG      ; To p1, RPRES points to z
        CALL    #FLT_MUL          ; p1*z
        MOV     #EXPP0,RPARG      ; To p0
        CALL    #FLT_ADD          ; p1*z + p0
        ADD     #2*FPL,RPARG      ; To g
        CALL    #FLT_MUL          ; g*P(z) = g*(p1*z + p0)
        MOV     @SP+,2*FPL–2(SP)  ; Store g*P(z)
        MOV     @SP+,2*FPL–2(SP)
        .if     DOUBLE=1
        MOV     @SP+,2*FPL–2(SP)
        .endif
;
; Calculation of Q(z): Q(z) = (q1*z + q0)         .FLOAT format
;                      Q(z) = (q2*z + q1)*z + q0  .DOUBLE format
;
        SUB     #FPL,SP           ; Area for Q(z)
        .if     DOUBLE=1          ; Quadratic equation
        MOV     #EXPQ2,RPARG      ; To q2
        ADD     #FPL,RPRES        ; To z
        CALL    #FLT_MUL          ; q2*z
        MOV     #EXPQ1,RPARG      ; To q1
        CALL    #FLT_ADD          ; q2*z + q1
        .else                     ; Linear equation
        MOV     #EXPQ1,RPARG      ; To q1
```

```
        .endif
        ADD     #FPL,RPRES        ; To z
        CALL    #FLT_MUL          ; (q2*z + q1)*z  resp. q1*z
        MOV     #EXPQ0,RPARG      ; To q0
        CALL    #FLT_ADD          ; (q2*z + q1)*z + q0 or q1*z + q0
;
; Result mantissa R(g) = 0.5 + g*P(z)/(Q(z) – g*P(z))
;
        ADD     #2*FPL,RPARG      ; To g*P(z), RPRES to Q(z)
        CALL    #FLT_SUB          ; Q(z) – g*P(z)
        ADD     #2*FPL,RPRES      ; To g*P(z)
        CALL    #FLT_DIV          ; g*P(z)/(Q(z) – g*P(z))
        MOV     #FLT0P5,RPARG     ; To 0.5
        CALL    #FLT_ADD          ; R(g)=0.5 + g*P(z)/(Q(z)–g*P(z))
        MOV     @SP+,3*FPL+2(SP)  ; Store R(g) to result area
        MOV     @SP+,3*FPL+2(SP)
        .if     DOUBLE=1
        MOV     @SP+,3*FPL+2(SP)
        .endif
;
; Insert exponent N+1 to result
;
        ADD     #2*FPL,SP         ; To binary N
        SETC                      ; N + 1
        ADDC.B  @SP+,3(SP)        ; Add N + 1 to exponent of result
        BR      #FLT_END          ; To normal return, HELP = 0
;
; X is out of range: test if overflow (+) or underflow (–)
;
EXP_RNGOUT TST.B 2(SP)            ; Overflow? (sign positive)
        JGE     EXP_OVFL          ; Yes, error: handling in FPP04
        BR      #DBL_UNDERFLOW    ; Underflow: depends on SW_UFLOW
EXP_OVFL BR     #DBL_OVERFLOW
;
        .if     DOUBLE=1
FLTLN2I  .double +1.4426950408889634074*2    ; 2/ln2
```

```
EXPC     .double +0.693359375-2.12194440054690582 77E-4   ; c1+c2
EXPP1    .double +0.595042549776E-2          ;p1
EXPP0    .double  0.24999999999992           ;p0
EXPQ2    .double +0.29729363682E-3           ;q2
EXPQ1    .double +0.5356751764522E-1         ;q1
FLT0P5   .equ     $                          ; both are 0.5
EXPQ0    .double +0.50000000000000E+0        ; q0
         .else
FLTLN2I  .float   +1.4426950408889634074*2
EXPC     .float   +0.693359375-2.12194440054690582 77E-4
EXPP1    .float   +0.00416028863
EXPP0    .float    0.24999999950
EXPQ1    .float   +0.04998717878
FLT0P5   .equ     $                          ; both are 0.5
EXPQ0    .float   +0.50000000000
         .endif
```

5.6.10.18 The Power Function

The power function A^B is calculated. The number range for A and B is:

$2.9 \times 10^{-39} \leq A \leq 3.4 \times 10^{38}$
$-88.72 \leq B \times \ln A \leq +88.72$

For the error handling, see the header of the software.

The used formula is:

$$A^B = e^{B \times lnA}$$

The calculation errors for the power function are shown in the following table. They indicate relative errors.

Table 5–18. Relative Errors of the Power Function

X	.FLOAT	.DOUBLE	Result
1^1	0	0	1.0
$(3.4\times10^{38})^0$	0	0	1.0
$(5.5\times10^{-20})^2$	-9×10^{-7}	0	3.025×10^{-39}
1.00007^{88}	$-4.\times10^{-6}$	-9×10^{-10}	1.0061788
$1.00007^{1267513}$	–5.5%	-7×10^{-7}	3.4027×10^{38}

0^5	0	0	0.0
0.1^{-5}	-1.3×10^{-7}	-1.6×10^{-10}	10^5

The previous table shows the large errors for small bases raised by very large exponents. This is due to the natural logarithm function.

Calculation times:

.FLOAT with hardware multiplier: 17000 cycles

.FLOAT without hardware multiplier: 20000 cycles

.DOUBLE with hardware multiplier: 40000 cycles

.DOUBLE without hardware multiplier: 50000 cycles

```
; Power Function: A^B.                        Result on TOS = (@RPRES)^(@RPARG)
;
; Call:  MOV      #addressA,RPRES   ; RPRES points to operand A
;        MOV      #addressB,RPARG   ; RPARG points to operand B
;        CALL     #FLT_POWR         ; Call the power function
;        ...                        ; RPARG, RPRES and SP point to A^B
;
; Range: 2.9x10^-39 < A < 3.4x10^+38
;        -88.72 < BxlnA < +88.72
;
; Errors:         A < 0:              N = 1, C = 1, Z = 0  Result: -3.4E38
;        B x lnA > +88.72: N = 1, C = 1, Z = 1  Result: +3.4E38
;        B x lnA < -88.72:
;          N = 1, C = 0, Z = 0: Result: 0.0 if SW_UFLOW = 1
;          N = 0, C = x, Z = x: Result: 0.0 if SW_UFLOW = 0
;          B x lnA > 3.4E38:    Error handling of multiplication
;
; Stack:  FPL + 4 + (3 x FPL + 8) bytes
;
FLT_POWR .equ     $
         .if      DOUBLE=1
         TST      4(RPRES)          ; Check if A = 0
         JNZ      PWRL1
         .endif
```

```
        TST     2(RPRES)
        JNZ     PWRL1           ; A # 0
        TST     0(RPRES)
        JZ      POWR0           ; A = 0: result = 0
;
PWRL1   PUSH    RPARG           ; Save pointer to exponent B
        SUB     #FPL,SP         ; Working area
        MOV     RPRES,RPARG     ; Pointer to base A
        CALL    #FLT_LN         ; lnA
        JN      PWERR           ; A is negative
        MOV     FPL(SP),RPARG   ; Pointer to exponent
        CALL    #FLT_MUL        ; BxlnA
        JN      PWERR           ; B is too large. HELP # 0
        CALL    #FLT_EXP        ; e^(BxlnA) = A^B
PWERR   MOV     @SP+,FPL+2(SP)  ; To result area
        MOV     @SP+,FPL+2(SP)
        .if     DOUBLE=1
        MOV     @SP+,FPL+2(SP)
        .endif
        ADD     #2,SP           ; Skip exponent pointer
        BR      #FLT_END        ; Error code in HELP
;
POWR0   BR      #RES0           ; A = 0: A^B = 0
```

5.7 Battery Check and Power Fail Detection

The detection of the near loss of the supply voltage is shown for battery driven and for ac–powered MSP430 systems.

Described in the following section are several methods of how to check if the voltage of a battery or an accumulator is above the minimum supply voltage of the MSP430–system. Possibilities are given for the family members having the 14–bit ADC on–chip and also for the members without it.

Three ways, with different hardware, are given to detect power fail situations for ac–driven systems.

For all examples, applications, schematics, diagrams, and proven software code are given for a better understanding.

5.7.1 Battery Check

In microcomputer systems driven by a battery or an accumulator it is necessary to detect when the lowest usable supply voltage is reached. A battery check executed in regular time intervals ensures that the supply voltage is still sufficient. If the lowest acceptable voltage is reached, normally with an added security value, a warning can be given with the LCD. The decision algorithms used can be very different:

- ❏ Simple checks; if the low threshold is reached or not
- ❏ Sophisticated methods using the speed of the voltage reduction ($\Delta V/\Delta t$) dependent on the discharge behavior of the actual battery or accumulator type. For even better estimations the temperature of the battery can also be taken into account.

5.7.1.1 Battery Check With the 14–Bit ADC

Due to the ratiometric measurement principle of the ADC, the measured digital value of a constant, known reference voltage is an indication of the supply voltage of the MSP430C32x. The measured value is inversely proportional to the supply voltage Vcc. Figure 5–31 shows the connecting of the voltage reference for all three explained variants.

Using the auto mode of the ADC, the digital result, N, for an analog input voltage Vin is:

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

With a reference voltage Vref (Vin) of 1.2 V, the supply voltage Vcc (exactly VSVcc) can be measured in steps of approximately 0.3 mV near the voltage Vcc_{min} = 2.5 V.

Note:

If the other analog parts connected to the SVcc–terminal cause a voltage drop that cannot be neglected, it is recommended that the reference diode be connected to an unused TP-output or an O-output. Otherwise, the resulting voltage drop corrupts the result and the calculated value for Vcc is too small.

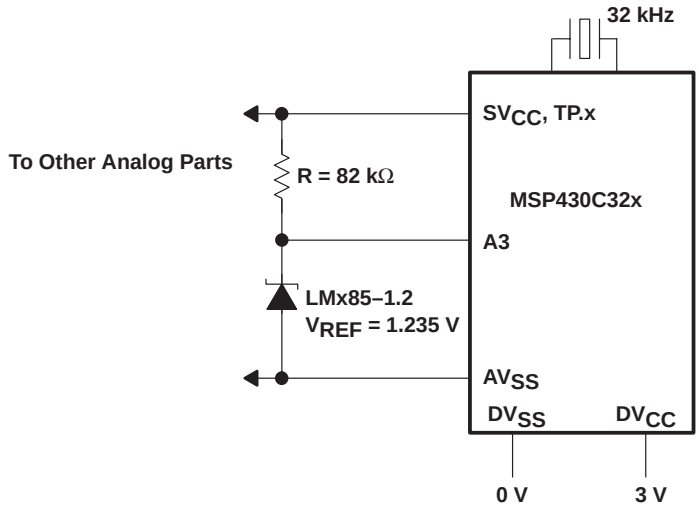

Figure 5–31. Connection of the Voltage Reference

Battery Check With a Reference Measurement

To get the reference for later battery checks, a measurement of the reference voltage (Vref) is made with Vcc = Vcc_{min}. The result is stored in RAM. If the battery should be tested, another measurement is made and the result is compared to the stored value. The result of the comparison determines the status of the battery.

- If the actually measured value exceeds the stored one, then Vcc < Vcc_{min} and a battery low indication is given by software.
- If the actually measured value is lower than the stored one, then Vcc > Vcc_{min}

EXAMPLE: The battery check with a reference measurement is shown for the analog input A3 (see Figure 5–31). During the calibration, a reference measurement is made with the lowest tolerable Vcc (Vcc_{min}). The battery check is then made in regular time intervals (in this example, every hour).

```
; RAM storage for the ADC value measured for Vref with Vccmin
;
ADVref   .EQU     0202h             ; ADC value for Vref at Vccmin
;
; Vccmin (+ security value) is adjusted. A certain code
; at Port0 or a temporary jumper between an input and an
; output leads to this software part
;
         CALL     #MEAS_A3          ; Vref connected to A3
         MOV      &ADAT,ADVref      ; Store reference ADC value
         ...
;
; One hour elapsed: check if Vcc is above Vccmin.
;
         CALL     #MEAS_A3          ; Vref connected to A3
         CMP      ADVref,&ADAT      ; (ADAT) – (ADVref)
         JLO      VCCok             ; Vcc > Vccmin
;
; The actual Vcc is lower than Vccmin. Indicate “Battery
; Low” in the LCD.
;
         CALL     #BATT_LOW         ; Output warning with LCD
VCCok    ...                        ; Continue program
;
; Measurement subroutine for analog input A3. Result in ADAT
;
MEAS_A3  BIC.B    #ADIFG,&IFG2      ; Reset ADC flag ADIFG
         MOV      #ADCLK2+RNGAUTO+CSOFF+A3+VREF+CS,&ACTL
L$101    BIT.B    #ADIFG,&IFG2      ; CONVERSION COMPLETED?
         JZ       L$101             ; IF Z=1: NO
         RET                        ; Yes, return. Result in ADAT
;
```

- Advantages
  - Very precise definition of one voltage
  - Small amount of software code
  - Different reference elements possible without software modifications
- Disadvantages
  - Calibration necessary
  - Relation to only one supply voltage value is known (calibration voltage)

Battery Check With the Calculation of the Voltage

If no reference measurement during a calibration phase is possible, the value of the supply voltage Vcc can be determined by calculation.

The formula is:

$$Vcc = 2^{14} \times \frac{Vref}{N}$$

With:

N	ADC result of the measurement of Vref
Vref	Voltage of the reference diode [V]

EXAMPLE: The actual supply voltage (Vcc) needs to be checked. The previous formula is used for the calculation after the measurement of the reference voltage (Vref). The MSP430 floating point package (32-bit .FLOAT version) is used for all calculations. The hardware is shown in Figure 5–31.

```
FPL      .equ     (ML/8)+1          ; Length of FPP number
         ......                              ; Normal program sequence
;
; One hour elapsed: check if Vcc is above Vccmin or not.
;
         CALL     #FLT_SAV          ; Save FPP registers on stack
         SUB      #FPL,SP           ; Allocate stack for result
         CALL     #MEAS_A3          ; Measure ref. diode at A3 (N)
         MOV      #ADAT,RPARG       ; Address of ADC result
         CALL     #CNV_BIN16U       ; Convert ADC result N to FP
;
; Calculate Vcc = 2^14 x Vref/(ADC–Result)
```

```
;
        MOV     #Vref,RPRES       ; Load address of Vref voltage
        CALL    #FLT_DIV          ; Calculate Vref/N (N on TOS)
        ADD.B   #14,1(RPRES)      ; Vcc = 2^14 x Vref/N (exp+14)
        MOV     #VCCmin,RPARG     ; Compare Vcc to VCCmin
        CALL    #FLT_CMP          ; Vcc – VCCmin
        JHS     BATT_ok           ; Vcc > VCCmin: ok
        CALL    #BATT_LOW         ; Give “Battery Low” Indication
BATT_ok ADD     #FPL,SP           ; Correct SP (result area)
        CALL    #FLT_REC          ; Restore FP registers
        ...                       ; Continue with program
Vref    .FLOAT  1.235             ; Voltage of ref. diode 1.235V
VCCmin  .FLOAT  2.5               ; Vccmin MSP430: 2.5V
```

- Advantages
  - Battery voltage is known (trend calculation possible)
- Disadvantages
  - Error of the reference element is not eliminated
  - Calculation takes time

Battery Check With a Fixed Value for Comparison

This method uses a fixed ROM–based value for the decision; if Vcc is sufficient or not. According to the data sheet of the LMx85–1.2, the typical voltage of this reference diode is 1.235 V with a maximum deviation of ±0.012 V. Therefore, the fixed comparison value (Nref) for the minimum supply voltage (Vcc_{min}) can be calculated:

$$Nref = INT\left|\frac{Vref \times 2^{14}}{Vcc_{min}}\right|$$

With Vcc_{min} = 2.5 V and Vref = 1.235 V ± 0,012 V:

$$Nref = INT\left|\frac{(1.235 \pm 0{,}012\ V) \times 2^{14}}{2.5\ V}\right| = 8093 \pm 78$$

To ensure that the voltage of the battery is above Vcc_{min}, the reference value should be set to:

N_{ref} = 8093 – 78 = 8015

Every measured value below 8015 indicates that the battery voltage is higher than the calculated value, even under worst–case conditions. If the measured value is above 8015, a *Battery Low* warning should be given.

EXAMPLE: The battery check with a fixed value for comparison is executed. The hardware needed is shown in Figure 5–31. The comparison value is stored in ROM at address VCCmin.

```
; One hour elapsed: check if Vcc is above Vccmin or not.
;
        CALL    #MEAS_A3          ; Vref connected to A3
        CMP     VCCmin,&ADAT      ; (ADAT) – (VCCmin)
        JLO     VCCok             ; Vcc > Vccmin
;
; The actual Vcc is lower than Vccmin. Output “Battery
; Low” to the LCD.
;
        CALL    #BATT_LOW         ; Output warning to the LCD
VCCok   ...                       ; Continue program
;
; ROM storage for the calculated ADC value: Vref at Vccmin.
; (worst case value).
;
VCCmin  .WORD   8015              ; ADC value 1.235V at 2.5V
```

- ❏ Advantages
  - ■ Small amount of software code
- ❏ Disadvantages
  - ■ Error of the reference element is not eliminated
  - ■ Fixed reference element
  - ■ Relation to only one supply voltage value is known

5.7.1.2 Battery Check With an External Comparator

With an operational amplifier used as a comparator, a simple battery check can be implemented for MSP430 family members that do not have the 14–bit ADC. Figure 5–32 shows two possibilities:

1) On the left, a simple *Go/No Go* solution. The voltage at P0.7 is high, when Vcc is above Vcc_{min} and is low when Vcc is below this voltage. The threshold voltage Vcc_{min} is:

$$Vcc_{min} = Vref \times \left(\frac{R1}{R2} + 1\right)$$

2) On the right, a circuit that allows the comparison of the battery voltage (Vcc) to three different voltage levels; two of them can be determined, the third results from the calculated resistor values for R1, R2 and R3. This allows to distinguish four ranges of the supply voltage:

- ❑ $Vcc < Vth_{min}$ The supply voltage is below the lowest threshold
- ❑ $Vth_{min} < Vcc < Vth_{mid}$ The supply voltage is between Vth_{min} and Vth_{mid}
- ❑ $Vth_{mid} < Vcc < Vth_{max}$ The supply voltage is between Vth_{mid} and Vth_{max}
- ❑ $Vth_{max} < Vcc$ The supply voltage is above the maximum threshold

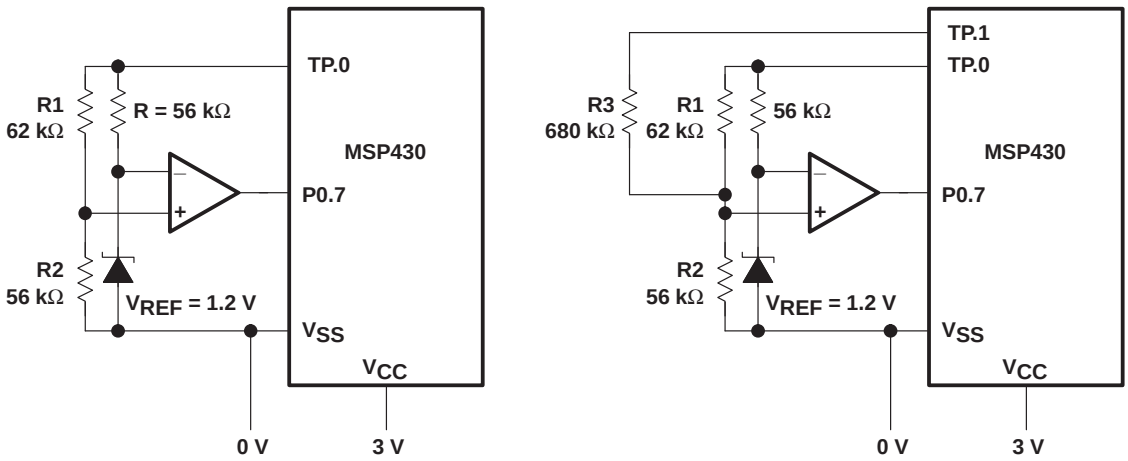

Figure 5–32. Battery Check With an External Comparator

The three different threshold levels are:

- ❑ Resistor R3 is switched off (TP.1 is switched to Hi–Z):

$$Vth_{mid} = Vref \times \left(\frac{R1}{R2} + 1\right)$$

- R3 is switched to Vcc by TP.1:

$$Vth_{min} = Vref \times \left(\frac{R1||R3}{R2} + 1\right)$$

- R3 is switched to Vss by TP.1:

$$Vth_{max} = Vref \times \left(\frac{R1}{R2||R3} + 1\right)$$

If the comparator's output (Vout) is high, Vcc is above the selected threshold voltage, if Vout is low, then Vcc is below this voltage.

The calculation of the resistors R1 to R3 starts with the desired threshold voltage (Vth_{mid}), R1 and R2 are derived from it. Then, the low threshold voltage (Vth_{min}) defines the value of R3. The 3rd threshold (Vth_{max}) results from the other two threshold voltages.

The resistor values shown in Figure 5–32 define the following threshold values:

Vth_{min} = 2.52 V (calculated with second step)

Vth_{mid} = 2.66 V (calculated first)

Vth_{max} = 2.78 V (results from the other two thresholds)

EXAMPLE: With the hardware shown in Figure 5–32 (circuit on right side), the actual battery voltage (Vcc) is compared to three different thresholds. This allows the differentiation of four different ranges for Vcc. For any of the four supply levels, different actions are started at the appropriate labels (not shown). Dependent on the speed of the MSP430 and the comparator used, NOPs may be necessary between the setting of the TP-ports and the bit test instructions BIT.B.

```
; One hour elapsed: check the range Vcc falls in now.
;
        BIS.B   #TP0+TP1,&TPD           ; TP.0 and TP1 active high
        BIS.B   #TP0+TP1,&TPE           ; Comparison with Vthmin
        BIT.B   #P07,P0IN               ; Comparator output
        JZ      BATTlo                  ; Vcc < Vthmin
        BIC.B   #TP1,&TPE               ; TP.1 to HI-Z
        BIT.B   #P07,P0IN               ; Comparator output
```

```
JZ      BATTmid                 ; Vthmin < Vcc < Vthmid
BIC.B   #TP1,&TPD               ; TP.1 active to Vss
BIS.B   #TP1,&TPE               ; Check Vthmax
BIT.B   #P07,P0IN               ; Comparator output
JNZ     BATThi                  ; Vcc > Vthmax
...                             ; Vthmid < Vcc < Vthmax
;
```

- Advantages
  - Four ranges defined (more ranges are possible if desired)
  - Very fast software
  - Different reference elements are possible without software change
- Disadvantages
  - Hardware effort (except if an unused operational amplifier of a quad–pack can be used)

5.7.1.3 Battery Check With the Universal Timer/Port Module

The Universal Timer/Port module allows a relatively accurate measurement of the battery voltage (Vcc). The principle (see Figures 5–33 and 5–34) is as follows: the capacitor (C), also used for the other measurements, is charged–up to the voltage (Vref) of the reference diode. C is then discharged with Rref and the time tref until VC reaches the lower threshold VIT– of the input CIN is measured. Afterwards, C is charged–up fully to the supply voltage (Vcc) and the discharge time (tVcc) is also measured. Vcc is then:

$$Vcc = Vref \times e^{\frac{tVcc - tref}{\tau}}$$

With:

Vcc	Actual supply voltage of the MSP430 [V]
Vref	Voltage of the reference diode [V]
tref	Time to discharge C from Vref to VIT– [s]
tVcc	Time to discharge C from Vcc to VIT– [s]
τ	Time constant for discharge: τ = Rref $\times$ C [s]
VIT–	Lower threshold voltage of input CIN [V]

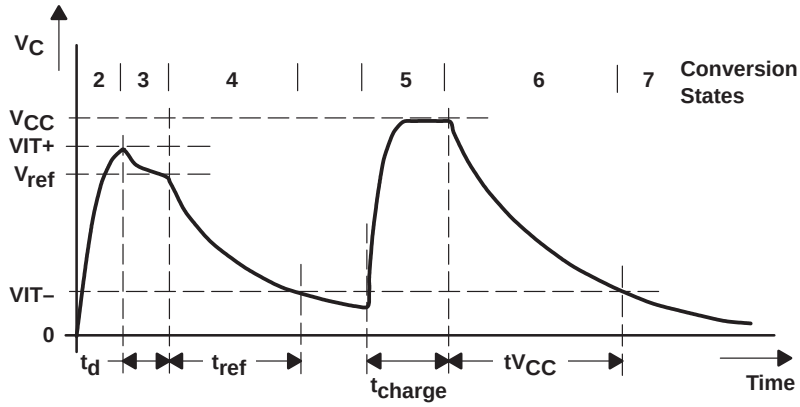

Figure 5–33. Discharge Curves for the Battery Check With the Universal Timer/Port Module

Two hardware possibilities are shown in Figure 5–34:

- The left side uses the existing ADC hardware for the battery check too.
- The right side uses different battery check hardware. This avoids any influence from the battery check and creates precise ADC–measurement hardware.

See the application report, *Voltage Measurement with the Universal Timer/ Port Module*, in Chapter 2 for more information. Here a formula is given that is independent of the time constant (τ).

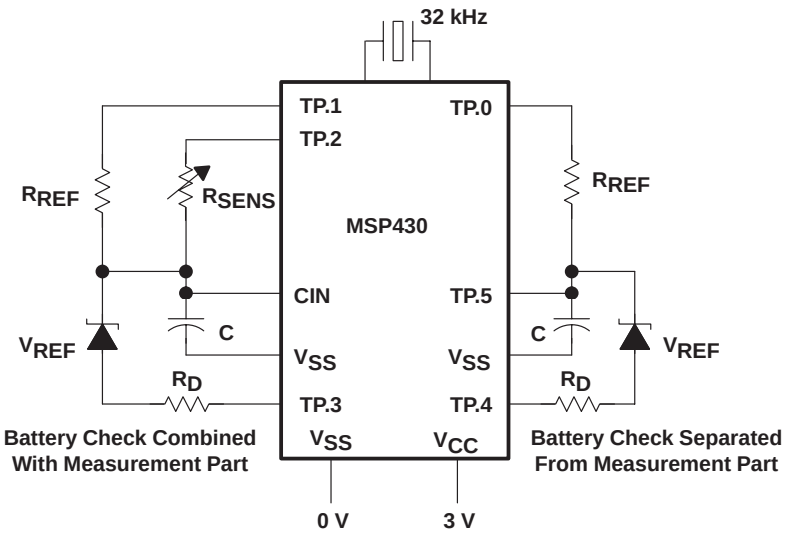

Figure 5–34. Battery Check With the Universal Timer/Port Module

The conditions to be met for the reference voltage (Vref) are:

Vref > VT– Vref must be higher than the lower threshold voltage VT– of the input CIN at Vcc_{max}

Vref < Vcc_{min} Only voltages above Vref can be measured

The previous conditions mean for an MSP430 system supplied with 3 V: Vref = 1.5 V to 2.5 V.

The measurement sequence like shown in Figure 5–33 is described for the left–side circuitry of Figure 5–34 (the following sequence of numbers refer to the *Conversion States* of Figure 5–33):

1) Switch outputs TP.1 to TP.3 to HI–Z

2) Charge capacitor C with resistor (Rref) until input CIN gets high (or up to Vcc), then switch–off Rref (TP.1 is set to HI–Z)

3) Discharge capacitor C with the reference diode and Rd to Vref (TP.3 is set to LO). Discharge time: td > 5 × Rd × C. Set TP.3 to HI–Z.

4) Discharge capacitor C from Vref to VIT– with Rref (TP.1 set to LO). Measure discharge time tref

5) Charge capacitor C with Rref to Vcc (tcharge > 5 × Rref × C)

6) Discharge capacitor C from Vcc to VIT– with Rref (TP.1 set to LO). Measure discharge time (tVcc)

7) Calculate Vcc with the formula shown previously

For the supply voltage range of a 3–V system (Vcc = 2.5V to 3.5V) and a reference voltage of Vref = 2.3 V, the exponential part of the equation can be replaced by a linear function:

$$Vcc = Vref \times \left(1.29 \times \frac{tVcc - tref}{\tau} + 0.97\right)$$

If the Universal Timer/Port Module is used in an ADC application with high accuracy (like a heat volume counter) then the battery–check circuitry should be connected to other I/Os as shown in Figure 5–34 on the right side. This way the measurement of the sensors cannot be influenced by the battery–check circuitry.

The software shown in the application report, *Using the MSP430 Universal Timer/Port Module as an Analog-to-Digital Converter* in Chapter 2, can also be used for the battery check with only a few modifications.

- Advantages
  - Minimum hardware effort if measurement part exists anyway
  - Supply voltage is known after the measurement
- Disadvantages
  - Slow measurement

5.7.2 Power Fail Detection

AC driven systems need a much faster indication of a power–down situation than battery–driven systems. It is a matter of milliseconds, not of hours or days. Therefore, other methods are used. Three of them are described in the following text.

- The non–regulated side of the power supply is observed. If the voltage (VC) of the charge capacitor falls below a certain level (VC_{min}), an interrupt is requested.
- The voltage at the secondary side of the ac transformer is observed. A sufficient level change there resets the watchdog. If the secondary voltage is too low or ceases, an interrupt is requested.
- The non–regulated side of the power supply is observed with a TLC7701. The output of this supply voltage supervisor requests an NMI interrupt or resets the microcomputer.

The interrupt requested by the previous three solutions is used to start the necessary emergency actions:

- ❏ Switching–off all loads to lengthen the available time for the emergency actions
- ❏ Reduction of the system clock MCLK to 1 MHz to be able to use Vcc down to Vcc_{min}
- ❏ Storage of all important values into an external EEPROM
- ❏ Use of LPM3 finally to bridge the power failure eventually

The three hardware proposals can be used with all members of the MSP430 family. The power–fail detection is also called *ac–low detection*. It issues the *ac–low* signal.

5.7.2.1 Power–Fail Detection by Observation of the Charge Capacitor

Here the voltage level of the charge capacitor (Cch) is observed. If the voltage level of this capacitor falls below a certain voltage level (VC_{min}), an interrupt is requested. With the circuit shown in Figure 5–35, VC_{min} is:

$$VC_{min} = Vcc \times \frac{\left(\frac{R3}{R4}+1\right)}{\left(\frac{R1}{R2}+1\right)}$$

R1, R2, R3 and R4 are chosen in a way that delivers the desired threshold voltage (VC_{min}). The regulated supply voltage (Vcc) is used as a reference. The NMI (non–maskable interrupt) can be used to get the fastest possible response.

The remaining time (trem) for actions after a power–fail interrupt is approximately:

$$trem = \left(VC_{min} - Vcc_{min} - Vr\right) \times \frac{Cch}{I_{AM}}$$

Where:

trem	Approximate time from interrupt to the reaching of Vcc_{min}	[s]
Cch	Capacity of the charge capacitor	[F]
I_{AM}	Supply current of the MSP430 system (medium value)	[A]
VC_{min}	Voltage at the charge capacitor that causes ac–low interrupt	[V]
Vcc_{min}	Lowest supply voltage for the MSP430	[V]
Vr	Dropout voltage (voltage difference between output and input) of the voltage regulator for function	[V]

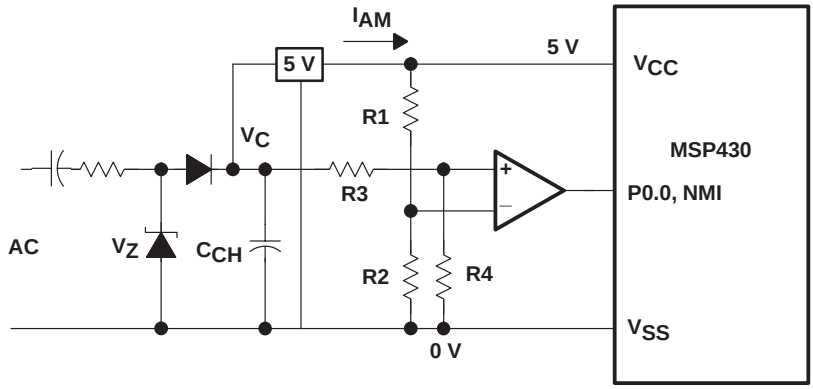

Figure 5–35. Power Fail Detection by Observation of the Charge Capacitor

With the following component values for the hardware shown in Figure 5–35, trem for emergency tasks can be calculated with the following formula.

C_{CH} = 50 µF, V_{CCmin} = 2.5 V, Vr = 1 V, IAM = 2 mA, Vz = 10 V, VC_{min} = 7 V

$$trem = (7V - 2.5V - 1V) \times \frac{50\mu F}{2mA} = 87.5ms$$

This remaining time trem = 87.5 ms allows between 14000 and 87500 instructions (dependent on the addressing modes) for the saving of important values in an EEPROM and other emergency tasks.

Note:

The capacitor power supply shown in Figure 5–35 is used only to demonstrate this hardware possibility. A normal transformer supply as shown with the other hardware examples can also be used.

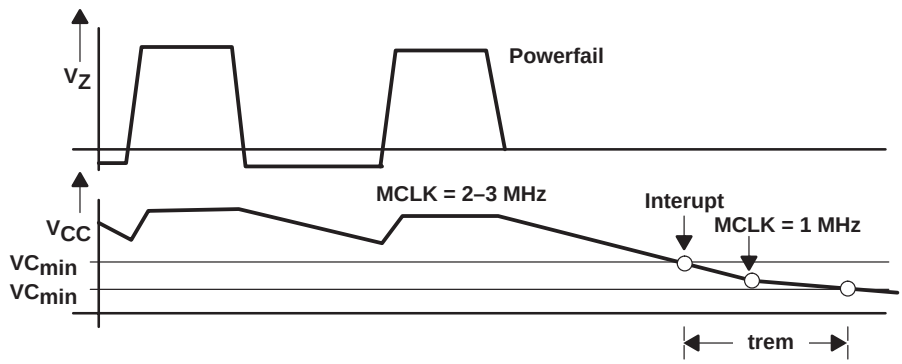

Figure 5–36. Voltages for the Power-Fail Detection by Observation of the Charge Capacitor

The equations shown previously are only valid if the dropout voltage (Vr) of the used voltage regulator (Vr = VC – Vcc) is relatively low. Vr must be:

$$Vr \quad < \quad VC0 - \frac{Vreg \times VC0}{VC\,min}$$

Where:
VC0 Lowest voltage at Cch that outputs low voltage to the MSP430 input [V]
Vreg Nominal output voltage of the voltage regulator [V]

If this condition for Vr is not possible, then another approach is necessary. Figure 5–37 shows a circuitry that is independent of the previously described restriction.

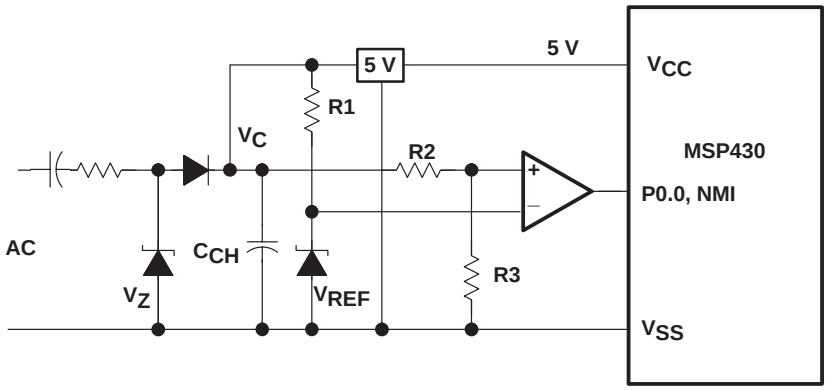

Figure 5–37. Power–Fail Detection by Observation of the Charge Capacitor

The threshold voltage level (VC_{min}) for the interrupt is :

$$VC_{min} = Vref \times \left(\frac{R2}{R3} + 1\right)$$

The time remaining (trem) for emergency tasks can be calculated:

$$trem = \left(VC_{min} - Vcc_{min} - Vr\right) \times \frac{Cch}{I_{AM}}$$

If brown outs are a serious problem, the hardware proposal shown in Figure 5–37 can be used with the RESET/NMI terminal as described in Section 5.7.2.3, *Power-Fail Detection with a Supply Voltage Supervisor*. Instead of the inverted RESET output of the TLC7701, the output of the operational amplifier is used.

EXAMPLE: The interrupt handler and its initialization is shown for the power–fail detection by observation of the charge capacitor with a comparator. After the completion of the emergency tasks, a test is made to check if the supply voltage is still low. If not, the software restarts at label PF_INIT. Otherwise, LPM3 is entered to eventually bridge the power failure. The basic timer checks with its interrupt handler in regular intervals for an indication that the voltage is above VC_{min} again. The hardware shown in Figure 5–35 is used.

```
; SYSTAT contains the current system status: calibration,
; normal run, power fail aso.
;
SYSTAT   .EQU     0200h             ; System status byte
;
; The program starts at label INIT if a power–up occurs
;
INIT     ...                        ; Normal initialization
;
; The program restarts at label PF_INIT if the supply voltage
; returns before Vccmin is reached (short power fail)
;
PF_INIT  MOV      #0300h,SP         ; Restart after power fail
         ...                        ; Special initialization
;
; Initialization: Prepare P0.0 for power fail detection.
;
         BIS.B    #P0IFG0,&IE1      ; Enable P0.0 interrupt
         BIS.B    #P00,&P0IES       ; Intrpt for trailing edge
         BIC.B    #P0IFG0,&IFG1     ; Reset flag (safety)
         ...                        ; Continue with initialization
         EINT                       ; Enable GIE
MAINLOOP MOV.B    #NORMAL,SYSTAT    ; Start normal program
         ...                        ;
;
; P0.0 Interrupt Handler: the voltage VC at Cch fell below a
; minimum voltage VCmin. Switch off all loads and interrupts
; except Basic Timer interrupt.
;
P00_HNDLR BIS     #PD,&ACTL         ; ADC to Power down
```

```
        MOV.B   #32–1,&SCFQCTL    ; MCLK back to 1MHz
        BIC.B   #01Ch,&SCFI0      ; DCO current source to 1MHz
        CLR.B   &TPD              ; Reset all TP–ports
        ...                       ; Store values to EEPROM
;
; All tasks are done, return to PF_INIT if Vcc is above Vccmin
; otherwise go to LPM3 to bridge eventually the power fail time
;
        BIT.B   #P00,&P0IN        ; Vcc above Vcmin again?
        JNZ     PF_INIT           ; Yes, restart program
        MOV.B   #PF,SYSTAT        ; System state is “Power Fail”
        BIS     #CPUoff+GIE+SCG1+SCG0,SR   ; Set LPM3
        JMP     PF_INIT           ; Continue here from BT
;
; Basic Timer Interrupt Handler: a check is made for power
; fail: if actual, only the return of Vcc is checked. If Vcc is
; above VCmin, LPM3 is terminated by modification of stack info
;
BT_HNDLR CMP.B  #PF,SYSTAT        ; System in ”Power Fail” state?
        JNE     BT$1              ; No, normal system states
        BIT.B   #P00,&P0IN        ; Yes: Vcc above VCmin again?
        JZ      BT_RTI            ; No, return to LPM3
        BIC     #CPUoff+SCG1+SCG0,0(SP)    ; Yes, leave LPM3
BT_RTI  RETI
BT$1    ...                       ; Normal Basic Timer handler
;
        .SECT   ”INT_VEC0”,0FFE2h
        .WORD   BT_HNDLR          ; Basic Timer Vector
        .SECT   ”INT_VEC1”,0FFFAh
        .WORD   P00_HNDLR         ; P0.0 Inrtpt Vector
        .WORD   0                 ; NMI not used
        .WORD   INIT              ; Reset Vector
```

- Advantages
  - Precise due to the use of the +5V regulator voltage for reference purposes
  - Fast response to charge losses
- Disadvantages
  - Hardware effort (except an unused operational amplifier of a multiple pack can be used)

5.7.2.2 Power–Fail Detection With the Watchdog

The ac–low detection can also be made with the internal watchdog. The watchdog is reset twice by one half–wave of the ac voltage (Vtr). If this does not occur, due to a power fail, the watchdog initializes the system. The reason for the system reset can be checked during the initialization routine and the necessary emergency actions taken. See the introduction of this section for details of these actions.

The advantage of this method is the unnecessary operational amplifier, the difficulty is to react to *brown–out* conditions. The ac voltage is still active but too low for an error–free run. If a brown out can be excluded or is impossible due to the hardware design, the watchdog solution is a very cheap and reliable possibility for ac–low detection.

If the restricted interval possibilities (only eight discrete time intervals) of the watchdog timer cannot satisfy the system needs, the watchdog timer can be used as a normal timer and the needed interval built by summing–up shorter intervals with software.

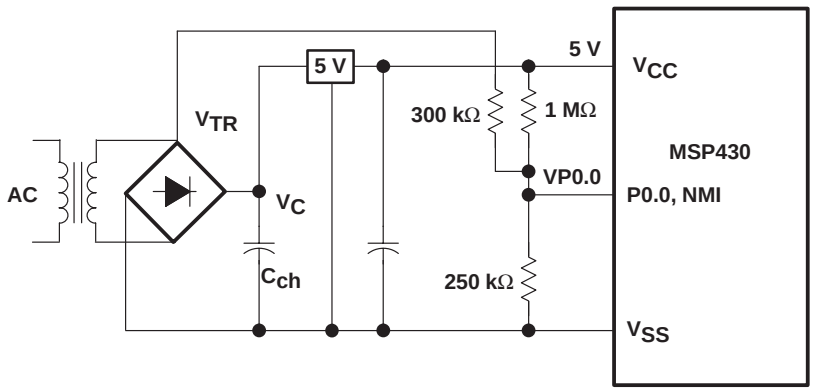

Figure 5–38. Power–Fail Detection With the Watchdog

With the component values shown in Figure 5–38, a square wave out of the ac voltage (Vtr) is reached (the MSP430 inputs have Schmitt–trigger characteristics). The voltages Vtr+ and Vtr– at the transformer output (Vtr) that switch the input voltage at the NMI (or P0.x) input are +7 V and +2 V, respectively. If these two voltage thresholds are carefully adapted to the actual environment, brown–out conditions can also be handled very safely. The equation for trem is:

$$trem \geq \left(Vtr + \; - Vcc_{min} - Vr - Vd\right) \times \frac{Cch}{I_{AM}} - tWD$$

Where:

Vtr+	Transformer voltage that switches the P0.0 input to high [V]
Vd	Voltage drop of one rectifier diode [V]
tWD	Watchdog interval [s]

All other definitions are equal to those explained in Section 5.7.2.1, *Power Fail Detection by Observation of the Charge Capacitor*.

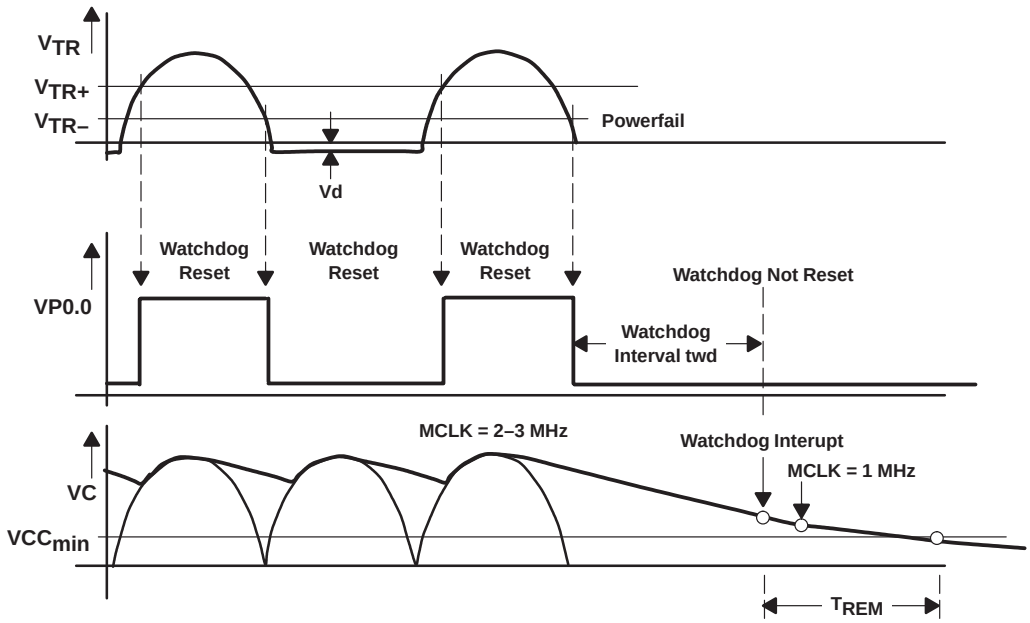

Figure 5–39. Voltages for the Power–Fail Detection With the Watchdog

EXAMPLE: An MSP430 system running with MCLK = 2 MHz uses the watchdog for power–fail detection. The watchdog uses the tap with ($t_{MCLK} \times 2^{15}$) = 16 ms (value after reset). After the completion of the emergency tasks, the soft-

ware checks if the ac voltage is back again with a loop. This is made by checking if P0.0 goes high. If this is the case, the initialization part is entered. The circuit shown in Figure 5–38 is used.

```
; Power–up and watchdog reset start at label INIT. The reason
; for the reset needs to be known (power–up or watchdog)
;
INIT     BIT.B    #WDTIFG,&IFG1     ; Reset by watchdog?
         JNZ      WD_RESET          ; Yes; power fail
;
; Normal reset caused by RESET pin or power–up: Init. system
;
         BIS.B    #4,&SCFI0         ; Switch DCO to 2MHz drive
         MOV.B    #64–1,&SCFQCTL    ; FLL to 2MHz
         MOV      #05A00h+CNTCL,&WDTCTL      ; Reset watchdog
         BIS.B    #P0IE0,&IE1       ; Enable P0.0 intrpt
         ...                        ; Continue initialization
         EINT                       ; Finally set GIE
MAINLOOP ...                        ; Start main program
;
; Reset caused by watchdog: missing main means power fail
; Supply current is minimized to enlarge active time. All
; interrupts except P0.0 interrupt are switched off
;
WD_RESET BIC.B    #03Fh,&TPD        ; Switch off all TP–outputs
         ...                        ; Switch off other loads
         BIS      #PD,&ACTL         ; Power down ADC
         MOV.B    #32–1,&SCFQCTL    ; MCLK back to 1MHz
         BIC.B    #01Ch,&SCFI0      ; DCO drive to 1MHz, FN_x = 0
         ...                        ; Store values to EEPROM
;
; All tasks are done: check if mains is back (P0.0 gets HI).
;
Llow     BIT.B    #P00,&P0IN        ; Actual state of P0.0 pin
         JZ       Llow              ; Still low
         BR       #INIT             ; P0.0 is HI, initialize
;
```

```
; The P00_HNDLR is called twice each period of the mains
; voltage. The watchdog is reset to indicate normal run,
; the edge selection bit of P0.0 is inverted.
;
P00_HNDLR MOV     #05A00h+CNTCL,&WDTCTL      ; Reset watchdog
          XOR.B   #P00,&P0IES        ; Invert edge select for P0.0
          RETI                       ;
;
          .SECT   "INT_VEC1",0FFFAh
          .WORD   P00_HNDLR          ; P0.0 Inrtpt Vector
          .WORD   0                  ; NMI not used
          .WORD   INIT               ; Reset Vector
```

- Advantages
  - Minimum hardware effort
  - Minimum software effort
  - Very fast
  - Brown out conditions can be handled by a precise hardware definition
- Disadvantages
  - Remaining time trem can be calculated only for worst case

5.7.2.3 Power–Fail Detection With a Supply Voltage Supervisor

For extremely safe MSP430 applications, a TLC7701 supply voltage supervisor can be used. The voltage (VC) of the charge capacitor (Cch) is observed. The output signal /RESET indicates if VC is higher or lower than the threshold voltage (Vth). Figure 5–40 shows the schematic for this application. The output signal /RESET of the TLC7701 is used in two different ways, depending on the actual state of the application.

- During power–up, the TLC7701 output is used as a reset signal. The MSP430 is held in the reset state until VC reaches a certain voltage (Vth) (e.g., supply voltage + regulator voltage drop) (see Figure 5–41).
- During run mode the RESET/NMI terminal of the MSP430 is switched to NMI–mode (Non–Maskable Interrupt) by software. If VC falls below Vth, an NMI is requested. The interrupt handler can start all necessary emergency tasks. See the introduction of this section for the description of these tasks.

Note:

This method is quite different from the normal use of the TLC7701. If used the normal way, the device outputs a reset signal in case of a Vcc that is too low. This reset signal stops the CPU of the connected microcomputer and gives no opportunity to save important values to an EEPROM.

With the method described, the output of the voltage regulator can also be observed. This allows the use of a TLC7705. The remaining time (trem) is shorter due to the lower threshold voltage used on the output side. For this application, the TPS7350, which includes the voltage regulator and the supply voltage supervisor, is ideally suited.

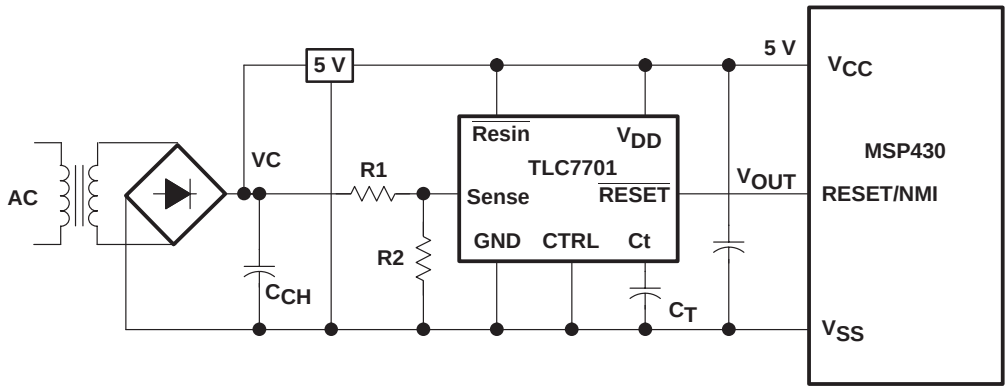

Figure 5–40. Power–Fail Detection with a Supply Voltage Supervisor

Figure 5–41 shows the different system states of the voltage supervisor solution. The voltage (VC) drawing is simplified for a better understanding of the system function. The different *system states* (shown in Figure 5–41) are:

1) The TLC7701 output is low until the voltage (Vth) is reached. The RESET/NMI input of the MSP430 is a reset input after the power–up, so the MSP430–CPU is inactive.

2) After reaching Vth (and the expiration of the delay trc), the MSP430 starts working and switches the RESET/NMI input to NMI–mode (interrupt input).

3) If VC goes below Vth due to a power fail, an interrupt is requested and the necessary tasks (e.g., EEPROM saving.) are started. Finally the RESET/NMI terminal is switched to the RESET function.

4) If (as shown in Figure 5–41) the power fail is short in duration (Vout is high again), the software continues at label INIT (after the elapse of trc).

5) If a real power fail occurs, the emergency tasks are completed and the reset mode for the RESET/NMI terminal is switched on again.

6) This means stop for all MSP430 activities until ac power rises VC above Vth. The MSP430 then restarts with a normal power–up sequence as shown with system state 1.

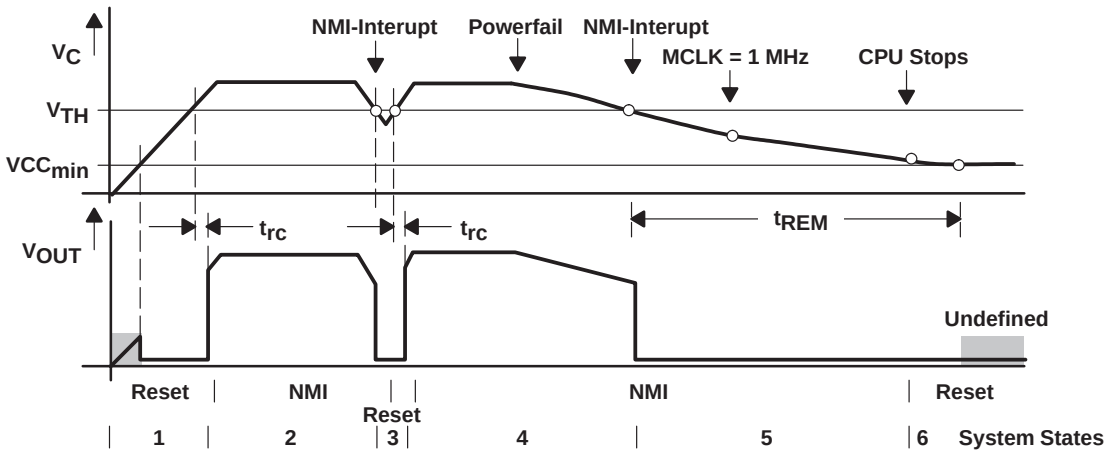

Figure 5–41. Voltages for the Power–Fail Detection With a Supply Supervisor

The formula for the remaining time (trem) is (the time available for emergency tasks):

$$trem = (Vth - Vcc_{min} - Vr) \times \frac{Cch}{I_{AM}}$$

Where:

trem	Approximate time from power-fail interrupt to the reaching of Vcc_{min} [s]
Vth	Threshold voltage for VC. Below this value Vout is low [V]
Vcc_{min}	Lowest supply voltage for the MSP430 [V]
Vr	Dropout voltage of the voltage regulator [V]
Cch	Capacity of the charge capacitor Cch [F]
I_{AM}	Supply current of the MSP430 system (medium value) [A]

The threshold voltage (Vth) of the TLC7701 can be calculated by:

$$Vth = Vref \times \left(\frac{R1}{R2} + 1\right)$$

Where:

Vref Voltage of the internal reference diode of the TLC7701: +1.1 V

R2 Resistor from SENSE input to 0 V. Nominal value 100 kΩ to 200 kΩ

The delay (trc) after the return of VC is defined by the capacitor (Ct) shown in Figure 5–40. If this delay is not desired, Ct is omitted. The formula trc is:

$trc = 21k\Omega \times Ct$

EXAMPLE: The MSP430 system shown in Figure 5–40 with its initialization and run–time software.

```
; Initialization: prepare RESET/NMI as an NMI interrupt input.
;
INIT     MOV      #05A00h+NMI+NMIES+CNTCL,&WDTCTL   ; 1–>0 edge
         ...                         ; Continue with initialization
         EINT                        ; Enable interrupt
MAINLOOP ...                         ; Start normal program here
;
; NMI Interrupt Handler: an oscillator fault or the trailing
; edge of the TLC7701 caused interrupt due to the low input
; voltage VC. Check first the cause of the interrupt.
; The load is reduced to gain time for emergency actions.
;
NMI_HNDLR BIT.B   #OFIFG,IFG1        ; Oscillator fault?
         JNZ      OSCFLT             ; Yes, proceed there
         BIC.B    #03Fh,&TPD         ; Switch off all TP–outputs
         ...                         ; Switch off other loads
         BIS      #PD,&ACTL          ; ADC Power down
         MOV.B    #32–1,&SCFQCTL     ; MCLK back to 1MHz
         BIC.B    #01Ch,&SCFI0       ; DCO drive to 1MHz
         ...                         ; Store values to EEPROM
;
; All tasks are done: switch RESET/NMI to RESET function.
; CPU stops until next power–up sequence. If the TLC7701 output
; is high again (mains back) the program restarts at INIT
;
         MOV      #05A00h+CNTCL,&WDTCTL      ; PC is set to INIT
         BR       #INIT              ; Short power fail: Vcc high
```

```
;
        .SECT    "INT_VEC1",0FFFCh
        .WORD    NMI_HNDLR          ; NMI Vector
        .WORD    INIT               ; Reset Vector
;
```

- ❏ Advantages
  - ■ Extremely safe: can handle any environment with the appropriate software and hardware combination
- ❏ Disadvantages
  - ■ Hardware effort (TLC7701 needed)

5.7.3 Conclusion

The concepts shown for battery check and power–fail detection are only possible due to the MSP430's hardware features:

- ❏ Battery–driven systems can be realized only with microcomputers that need only a very low supply current
- ❏ In ac–driven systems, the available security of MSP430 systems is due to three unique MSP430 features:

1) The low current consumption allows the remaining charge of the (relatively small) charge capacitor to be used for a lot of emergency tasks in case of a power fail
2) The high speed of the CPU allows to finish all these necessary emergency tasks during the remaining time from power–fail detection to the reaching of the lowest usable supply voltage.
3) The wide supply voltage range (+5.5 V down to +2.5 V) increases the time remaining for these tasks.

These three features together allow relatively simple hardware solutions for MSP430 systems, especially the use of small charge capacitors.