Application Note 30

Title:

Frequently Used Software Routines for EM 4-bit Microcontrollers

Product Family: **4-bit Microcontroller**

Part Number: EM66xx, EM65xx

Keywords: 4 bits microcontroller, rotate left, carry flag, register, rotate right, index register, increment, decrement, buffer, subroutine levels, ADD, SUB, Shift, data tables, ROM, RAM, lookup algorithm, linear lookup, binary lookup, speed, trigonometric functions, jump tables, indirect jumps, N-way branching, immediate ALU instructions, accumulator, instruction timing, jump instructions, recursive calculation, data tables, 8-bit data, 16 bit binary division, 16 bit binary multiplication,

Date: June, 2005 REV. C

Table of Content:

Application Note 10

1. Instruction Timing

The 4 bits microcontroller operates at an oscillator frequency, equivalent to a clock period with a duty cycle of 50%. An instruction cycle of the 4 bits uC takes two clock periods. Each instruction cycle is split into 4 phases as shown in the following diagram.

```
        ----+     +-----+     +-----+     +---
CLK     P4  | P1  | P2  | P3  | P4  | P1  |
            +-----+     +-----+     +-----+
            +-----+                 +-----+
∅1          |     |                 |     |
        ----+     +-----------------+     +---
                  +-----+                 +---
∅2                |     |                 |
        ----------+     +-----------------+
                        +-----+
∅3                      |     |
        ----------------+     +---------------
        ----+                 +-----+
∅4          |                 |     |
            +-----------------+     +---------

            |<--------->| Clock Period
            |<--------------------->| Instruction Cycle
```

During the 4 phases the microcontroller performs the following functions:

Phase 1: **t**he instruction register is initialised.

Phase 2: **t**he instruction is fetched from ROM (or forced to an *interrupt subroutine call* when an interrupt request has been acknowledged) and is decoded at the end of phase 2

Phase 3: ALU operations are executed, the stack pointer is managed and data are read from RAM.

Phase 4: **d**ata are written to RAM and the program counter is incremented or modified.

<u>Remarks:</u>

As can be seen, the 4 bits microcontroller executes all the instructions of the instruction set (including *Jump* instructions) within in one instruction cycle, which is equal to two clock cycles.

Application Note 24

2. Compilation when using Macro

Look at the three examples below.
The first and second examples are assembled correctly, but not the third one. The ***jpnz START*** is assembled to the code ***7FFF***, which is wrong.
The label (i.e. START) needs to be defined before it is used in a macro call (i.e. brset RegSysCntl2, 0100b, START).

1st example without macro: **THAT WORKS !**

```
; ======================================
;            MAIN PROGRAM
; ======================================
MAIN:              call        INIT

                   sti         …

                   ldi         0100b
                   and         RegSysCntl2
                   jpnz        START

                   call        SUB

START:             sti         …
```

2nd example with macro: **THAT WORKS !**

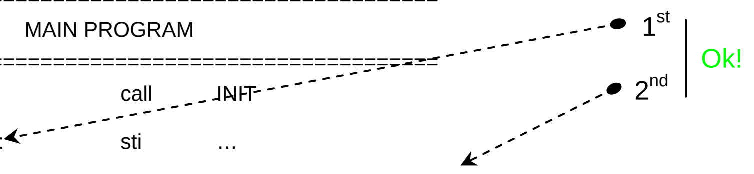
```
; ======================================
;            MAIN PROGRAM
; ======================================
MAIN:              call        INIT

START:             sti         …

                   brset       RegSysCntl2, 0100b, START

                   call        SUB

                   sti         …
```



Macro of the examples 2 - 3:

```
;-----------------------------------------------------------------
;  MACRO - BITCHECK
;-----------------------------------------------------------------

brset                    .macro ADDR, DATA, LABEL

                         ldi        DATA
                         and        ADDR
                         jpnz       LABEL

.macend
```

3rd example with macro: **THAT DOESN'T WORK !**

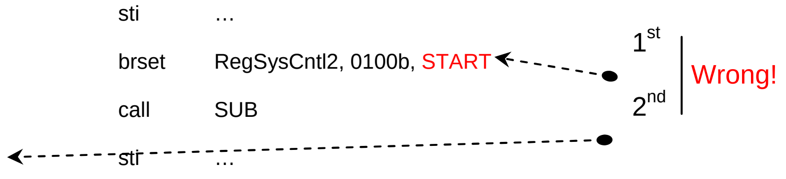
```
; ======================================
;            MAIN PROGRAM
; ======================================
MAIN:              call        INIT

                   sti         …

                   brset       RegSysCntl2, 0100b, START

                   call        SUB

START:             sti         …
```



Application Note 1

3. How to rotate left a register through carry

The 4 bits uC does not provide a *Rotate Left* instruction. Therefore it is necessary to implement the *Rotate Left* function by a small piece of code, which could be written as a subroutine as shown below, but could be also inserted as an inline code block in the main program.

The following figure gives a graphical representation of the *Rotate Left* operation:

```
+-------------------------------+
|   +---+   +---------------+   |
+<--| C |<--| Register or A |<--+
    +---+   +---------------+
            Bit3              Bit0
```

ROLX - ***Rotate Left Indexed***

The following code rotates left a RAM register pointed to by the index register. It is written as a subroutine:

```
ROLX:       JPC ROLX1    ;if Carry is already set, branch to code that
                         ;rotates Carry into Bit0
            SHLX         ;get data and shift left, Bit3 -> C, 0 -> Bit0
            STAX         ;store data
            RET          ;return
ROLX1:      SHLX         ;get data and shift left, Bit3 -> C, 0 -> Bit0
            JPC ROLX2    ;as Carry may be reset by the following
                         ;instructions the condition C = 1 must be
                         ;handled separately.
            STAX         ;store data
            INCX         ;as Carry was set initially, Bit0 must be set
                         ;now (1 -> Bit0). Because Bit0 is 0, INCA sets
                         ;Bit0 without affecting the other bits.
                         ;Carry is reset (0 -> C) !
            STAX         ;store data
            RET          ;return
ROLX2:      STAX         ;store data
            INCX         ;1 -> Bit0, 0 -> C
            STAX         ;store data
            SHRA         ;restore Carry.
                         ;Bit0 -> C here is equivalent to 1 -> C
            RET          ;return
```

ROLR (Reg) - ***Rotate Left Register***

The following code rotates left a RAM register named **Reg** directly. It is written as a code block that can be included in the main program:

```
ROLR:       JPC ROLR1    ;if Carry is already set, branch to code that
                         ;rotates Carry into Bit0
            SHL Reg      ;get data and shift left, Bit3 -> C, 0 -> Bit0
            STA Reg      ;store data
            JMP ROLR3    ;exit
```

```
ROLR1:         SHL Reg       ;get data and shift left, Bit3 -> C, 0 -> Bit0
               JPC ROLR2     ;as Carry may be reset by the following
                             ;instructions the condition C = 1 must be
                             ;handled separately.
               STA Reg       ;store data
               INC Reg       ;as Carry was set initially, Bit0 must be set
                             ;now (1 -> Bit0). Because Bit0 is 0, INCA sets
                             ;Bit0 without affecting the other bits.
                             ;Carry is reset (0 -> C) !
               STA Reg       ;store data
               JMP ROLR3     ;exit
ROLR2:         STA Reg       ;store data
               INC Reg       ;1 -> Bit0, 0 -> C
               STA Reg       ;store data
               SHRA          ;restore Carry.
                             ;Bit0 -> C here is equivalent to 1 -> C
ROLR3:                       ;exit
```

ROLA - ***Rotate Left Accumulator***
The following code rotates left the accumulator. It is written as a subroutine. Note that a temporary register named **`Temp`** is used to save and restore the accumulator.

```
ROLA:          JPC ROLA1     ;if Carry is already set, branch to code that
                             ;rotates Carry into Bit0
               STA Temp      ;store accu
               SHL Temp      ;shift left, Bit3 -> C, 0 -> Bit0
               RET           ;return
ROLA1:         STA Temp      ;store accu
               SHL Temp      ;shift left, Bit3 -> C, 0 -> Bit0
               JPC ROLA2     ;as Carry may be reset by the following
                             ;instructions the condition C = 1 must be
                             ;handled separately.
               STA Temp      ;store accu
               INC Temp      ;as Carry was set initially, Bit0 must be set
                             ;now (1 -> Bit0). Because Bit0 is 0, INCA sets
                             ;Bit0 without affecting the other bits.
                             ;Carry is reset (0 -> C) !
               RET           ;return
ROLA2:         STA Temp      ;store accu
               INC Temp      ;1 -> Bit0, 0 -> C
               STA Temp      ;save A
               SHRA          ;restore Carry.
                             ;Bit0 -> C here is equivalent to 1 -> C
               LDR Temp      ;restore A
               RET           ;return
```

IMPORTANT:
Note that the Carry flag may be reset by some instructions. If you want to preserve the value of the Carry flag you have several options:

1. You may use a register as a temporary flag variable which you set according to the value of the Carry flag and which you can interrogate later.
2. Depending on the value of the Carry flag you may branch to different parts of your program using the **`JPC`** or **`JPNC`** instruction. In each part you may later restore the value of the Carry flag, for example by setting or clearing *Bit0* of the accumulator and then executing a **`SHRA`** instruction. The code examples in this application note demonstrate this method.

Application Note 2

4. How to rotate right a register through carry

The EM66xx does not provide a *Rotate Right* instruction. Therefore it is necessary to implement the *Rotate Right* function by a small piece of code, which could be written as a subroutine as shown below, but
could be also inserted as an inline code block in the main program.
The following figure gives a graphical representation of the *Rotate Right* operation:

```
+------------------------------+
|   +---------------+   +---+   |
+->-| Accumulator   |->-| C |->-+
    +---------------+   +---+
    Bit3              Bit0
```

RORX - ***Rotate Right Indexed***
The following code rotates right a RAM register pointed to by the index register. It is written as a subroutine:

```
RORX:         JPC RORX1     ;if Carry is already set, branch to code that
                            ;rotates Carry into Bit0
              LDRXS         ;get data and shift right, Bit0 -> C, 0 -> Bit3
              STAX          ;store data
              RET           ;return
RORX1:        LDRXS         ;get data and shift right, Bit0 -> C, 0 -> Bit3
              JPC RORX2     ;as Carry may be reset by the following
                            ;instructions the condition C = 1 must be
                            ;handled separately.
              STAX          ;store data
              LDI 08H       ;Select the bit to set.
              ADDX          ;as Carry was set initially, Bit0 must be set
                            ;now (1 -> Bit3). Because Bit3 is 0, ADDX sets
                            ;Bit3 without affecting the other bits.
                            ;Carry is reset (0 -> C) !
              STAX          ;store data
              RET           ;return
RORX2:        STAX          ;store data
              LDI 08H       ;Select the bit to set.
              ADDX          ;1 -> Bit3, 0 -> C
              STAX          ;store data
              SHLX          ;restore Carry.
                            ;Bit3 -> C here is equivalent to 1 -> C
              RET           ;return
```

RORR (Reg) - ***Rotate Right Register***
The following code rotates right a RAM register named **Reg** directly. It is written as a code block that can be included in the main program:

```
RORR:         JPC RORR1     ;if Carry is already set, branch to code that
                            ;rotates Carry into Bit0
              SHRR Reg      ;get data and shift right, Bit0 -> C, 0 -> Bit3
              STA Reg       ;store data
              JMP RORR3     ;exit
```

```
RORR1:        SHRR Reg      ;get data and shift right, Bit0 -> C, 0 -> Bit3
              JPC RORR2     ;as Carry may be reset by the following
                            ;instructions the condition C = 1 must be
                            ;handled separately.
              STA Reg       ;store data
              LDI 08H       ;Select the bit to set.
              ADD Reg       ;as Carry was set initially, Bit3 must be set
                            ;now (1 -> Bit3). Because Bit3 is 0, ADD sets
                            ;Bit3 without affecting the other bits.
                            ;Carry is reset (0 -> C) !
              STA Reg       ;store data
              JMP RORR3     ;exit
RORR2:        STA Reg       ;store data
              LDI 08H       ;Select the bit to set.
              ADD Reg       ;1 -> Bit3, 0 -> C
              STA Reg       ;store data
              SHL Reg       ;restore Carry.
                            ;Bit3 -> C here is equivalent to 1 -> C
RORR3:                      ;exit
```

RORA - ***Rotate Right Accumulator***

The following code rotates right the accumulator. It is written as a subroutine. Note that a temporary register named **`Temp`** is used to save and restore the accumulator.

```
RORA:         JPC RORA1     ;if Carry is already set, branch to code that
                            ;rotates Carry into Bit0
              SHRA          ;shift right, Bit0 -> C, 0 -> Bit3
              RET           ;return
RORA1:        SHRA          ;shift right, Bit0 -> C, 0 -> Bit3
              JPC RORA2     ;as Carry may be reset by the following
                            ;instructions the condition C = 1 must be
                            ;handled separately.
              STA Temp      ;store accu
              LDI 08H       ;Select the bit to set.
              ADD Temp      ;as Carry was set initially, Bit3 must be set
                            ;now (1 -> Bit3). Because Bit3 is 0, ADD sets
                            ;Bit3 without affecting the other bits.
                            ;Carry is reset (0 -> C) !
              RET           ;return
RORA2:        STA Temp      ;store accu
              LDI 08H       ;Select the bit to set.
              ADD Temp      ;1 -> Bit3, 0 -> C
              STA Temp      ;save A
              SHL Temp      ;restore Carry.
                            ;Bit3 -> C here is equivalent to 1 -> C
              LDR Temp      ;restore A
              RET           ;return
```

IMPORTANT:

Note that the Carry flag may be reset by some instructions. If you want to preserve the value of the Carry flag you have several options:

1. You may use a register as a temporary flag variable which you set according to the value of the Carry flag and which you can interrogate later.
2. Depending on the value of the Carry flag you may branch to different parts of your program using the **`JPC`** or **`JPNC`** instruction. In each part you may later restore the value of the Carry flag, for example by setting or clearing *Bit3* of the accumulator and then executing a **`SHL`** instruction. The code examples in this application note demonstrate this method.

Application Note 3

5. How to increment or decrement the index register

The 4 bits uC does not provide instructions to increment or decrement the index register. Therefore it is necessary to implement the *Increment* and *Decrement Index Register* functions by a small piece of code, which could be written as a subroutine as shown below.

Note that only bits 0 through 2 of the high nibble of the index register (**XH**) are valid. Bit 3 is ignored when writing to **XH** and is 0 when reading **XH**. Thus incrementing **XH** never sets the Carry flag.

INX - ***Increment Index Register***

The following code increments the index register by 1. It is written as a subroutine:

```
INX:         INC XL ;increment low nibble
             STA XL
             JPNC INX1    ;no carry
             INC XH ;if C = 1 then increment high nibble
             STA XH
INX1:        RET
```

Note that instead of the **JPNC INX1** instruction also the **JPNZ INX1** instruction could be used. When **XL** is incremented beyond 0FH, it becomes 0 and the Zero flag is set as well as the Carry flag.

DEX - ***Decrement Index Register***

The following code decrements the index register by 1. It is written as a subroutine:

```
DEX:         DEC XL ;decrement low nibble
             STA XL
             JPC DEX1     ;carry
             DEC XH ;if C = 0 then decrement high nibble
             STA XH
DEX1:        RET
```

When **XL** is decrement below 0, it becomes 0FH and the Carry flag is cleared.

Application Note 4

6. How to rotate left a large buffer in RAM

The following figure gives a graphical representation of the *Rotate left a large buffer in RAM through carry* operation:

```
+-------------------------------------------------------+
|   +---+   +-------+          +-------+   +-------+    |
+<--| C |<--| Reg_n |<-- - - <--| Reg_2 |<--| Reg_1 |<--+
    +---+   +-------+          +-------+   +-------+
            Bit3    Bit0       Bit3   Bit0  Bit3    Bit0
```

This operation can be performed by using some building blocks described in other application notes. For a detailed description of the subroutines called from the code sample given below see the following application notes:

ROLX	***- Rotate Left Indexed***	AppNote # 1:	*How to rotate left a register through carry*
INX	***- Increment Index Register***	AppNote # 3:	*How to increment or decrement the index register*

The 4 bits uC does not provide **`ROLX`** and **`INX`** instructions which are needed here. However, each of these instructions can be emulated by a small subroutine. The **`INX`** subroutine call destroys the current value of the Carry flag. The **`ROLX`** subroutine also modifies the Carry flag, but its value has to be saved in a temporary register named **`Carry`** to be used again during the next call of **`ROLX`**.

The following sample is the skeleton of a program using the *Rotate left a large buffer in RAM through carry* function. It is limited to a maximum buffer length of 64 bits (equal to 16 nibbles) because only one 4-bit loop counter register (**`LoopCnt`**) is used.

```
;Sample Source Code
; for Rotate left a large buffer in RAM through carry function
;-------------------------------------------------------------------
;use IO and register definitions in common header file:
             INCLUDE IoDef.asm

;CONSTANTS
 BufLen:     EQU 16 ;buffer length, 16 nibbles = 64 bits

;VARIABLES
 Carry:      EQU 0        ;temporary register to save carry flag
 LoopCnt:    EQU 1        ;loop counter for buffer position
 BufBegL:    EQU 2        ;buffer begin address, low nibble
 BufBegH:    EQU 3        ;buffer begin address, high nibble

;PROGRAM
             ORG 0
RESET: JMP Main           ;jump to main program after reset
INTERRUPT:   JMP Handler  ;jump to interrupt handler
Main:        ; ...        ;other application code

ROLBuf:      ;BEGIN Rotate Left Buffer Program Segment
             STI XL, BufBegL          ;initialise buffer pointer
             STI XH, BufBegH
             STI LoopCnt, BufLen      ;initialise loop counter
             STI Carry, 00H           ; reset carry to 00H

ROLBuf0:                              ;DO
             CALL ROLX                ; rotate left RAM location
                                      ;   (C -> B0, B3 -> C)
             JPNC ROLBuf1             ; test the carry C
             STI Carry, 01H           ; Set Carry to 01H
```

```
ROLBuf1:
             CALL INX                    ; increment index value
             DEC LoopCnt                 ; decrement loop counter
             STA LoopCnt                 ; save loop counter
             JPNC ROLBuf2                ; IF LoopCnt < 0 THEN EXIT DO
             SHRR Carry                  ; restore carry (B0 -> C)
             STI Carry, 00H              ; reset carry to 00H
             JMP ROLBuf0                 ; LOOP
ROLBuf2:     SHRR Carry                  ; restore carry (B0 -> C)
             ;END Rotate Left Buffer Program Segment

             ; ...                       ;other application code
             HALT
;use ROLX module from program library:
             INCLUDE ROLX.asm
;use INX module from program library:
             INCLUDE INX.asm
Handler:     ; ...                       ;space reserved for interrupt handler
             RTI
             END                         ;end of source code
```

IMPORTANT:

Only 2 subroutine call levels :

Note that the 4 bits uC permits only 2 subroutine call levels. In most cases one level will be reserved for the interrupt handler, typically serving the timer interrupt to reset the watchdog timer periodically. The code example given in this application note is based on this model.

If you want to use a second subroutine call level you have to disable the interrupt before issuing the level-2 call. You have to make sure that the watchdog timer does not expire, usually by re-enabling the interrupt after returning from the level-2 subroutine.

Carry flag may be reset by some instructions :

Note that the Carry flag may be reset by some instructions. If you want to preserve the value of the Carry flag you have several options:

1. You may use a register as a temporary flag variable which you set according to the value of the Carry flag and which you can interrogate later. The code example in this application note demonstrates this method.

2. Depending on the value of the Carry flag you may branch to different parts of your program using the **`JPC`** or **`JPNC`** instruction. In each part you may later restore the value of the Carry flag, for example by setting or clearing *Bit0* of the accumulator and then executing a **`SHRA`** instruction.

Application Note 5

7. How ADD, SUB and shift instructions handle the carry flag

The **`ADD`** and the **`SUB`** operations are executed without using any previous contents of the Carry flag. However, the Carry flag is set by these operations.

If the *Rotate Right through carry* instruction modifier is applied to the **`ADD`** and **`SUB`** instructions (by appending **`S`** to the basic mnemonics) the basic instructions are followed by an implied **`SHRA`** operation, which modifies the Carry flag accordingly.

***Add* Operations:**	**ADD Reg**	***- Add Register without carry***
	ADDX	***- Add Indexed without carry***
	S	***- Rotate Right through carry* instruction modifier**

For **`ADD`** the Carry flag is set when the result of the operation is larger than **`0FH`**. Otherwise the Carry flag is cleared. Formally the **`ADD`** operation may be described by the following algorithm:

```
A = Reg + A
IF A > 15 THEN C = 1 ELSE C = 0
A = A AND 15
IF A = 0 THEN Z = 0 ELSE Z = 1
```

***Subtract* Operations:**	**SUB Reg**	***- Subtract Register without carry***
	SUBX	***- Subtract Indexed without carry***
	S	***- Rotate Right through carry* instruction modifier**

For **`SUB`** the Carry flag is cleared when the result of the operation is less than **`0FH`**. Otherwise the Carry flag is set. Formally the **`SUB`** operation may be described by the following algorithm:

```
A = Reg - A
IF A < 0 THEN C = 0 ELSE C = 1
A = A AND 15
IF A = 0 THEN Z = 0 ELSE Z = 1
```

***Shift Right* Operations:**	**SHRA**	***- Shift Right Accumulator through carry***
	SHRR Reg	***- Shift Right Register through carry***
	LDRXS	***- Shift Right Indexed through carry***
	S	***- Shift Right through carry* instruction modifier**

The *Shift Right* operations load data from a register in the accumulator and then shift all bits one bit position to the right. This implies that *Bit0* is moved into the Carry flag (*Bit0 -> C*) and that *Bit3* is set to 0 (*0 -> Bit3*).

The following figure gives a graphical representation of the *Shift Right* operation:

```
+---+   +---------------+   +---+
| 0 |->-| Accumulator   |->-| C |
+---+   +---------------+   +---+
        Bit3              Bit0
```

***Shift Left* Operations:**	**SHL Reg**	***- Shift Left Register***
	SHLX	***- Shift Left Indexed***

The *Shift Left* operations load data from a register in the accumulator and then shift all bits one bit position to the left. This implies that *Bit3* is moved into the Carry flag (*Bit3 -> C*) and that *Bit0* is set to 0 (*0 -> Bit0*).

The following figure gives a graphical representation of the *Shift Left* operation:

```
+---+   +---------------+   +---+
| C |-<-| Accumulator   |-<-| 0 |
+---+   +---------------+   +---+
        Bit3              Bit0
```

Application Note 6

8. Creating Data Tables in ROM

The 4 bits uC has separate memories for instructions (ROM) and data (RAM). Each of these memories has its own address space, which may be accessed by two subsets of the instruction set, one for each address space. The instructions referencing ROM addresses typically change the flow of control in a program, such as JMP, CALL, RET instructions etc. The instructions referencing RAM addresses are typically data transfer and data manipulation instructions. As in any program some constant or initial data values are required which have to be stored in ROM to remain persistent when the power supply is turned off. Therefore some instructions are required to transfer data from ROM to RAM or to the accumulator. For this purpose the 4 bits uC provides the so-called *immediate* instructions which are listed below:

```
STI     Reg, Dat        ;R(Reg) = Dat
STIX    Dat             ;R(Idx) = Dat
LDI     Dat             ;A = Dat
```

Sometimes one would like to transfer not only a single data item, but a table of consecutive data items.
Also, sometimes one wants to lookup a data item from a table by referencing it by its position relative to the begin of the table (the so-called *table offset* or *table index*). Some microprocessors provide special instructions to accomplish this. Usually these instructions require at least two instruction cycles: one cycle to fetch the instruction itself from ROM, and the second cycle to transfer the ROM data to the accumulator. More cycles and more instruction words are required for complex instructions with table index and destination register operands. Such complex instructions are typical for CISC processors.
As a RISC processor, the 4 bits uC follows a different approach by deliberately restricting the instruction set for the benefits of reduced chip area and speed, and emulating complex instructions in software.
The following gives several examples of how to create tables and how to work with them. In this application note we are dealing with two classes of ROM tables:

- small tables that may be transferred into RAM
- tables too large to fit into available RAM

A. Small Tables

The most efficient way to work with a small table is to initialise it in RAM and then work with it using the RAM instructions.

Let us assume that we want to create a lookup table for trigonometric functions, for example the sine function between 0 and 90 degrees in steps of 10 degrees. This requires a table with 10 entries.
Let us further assume that is sufficient for the accuracy of our application to represent the maximum value sin(90) = 1 as 0FH (15 decimal). Then we need to build the following table:

x	sin(x)	15*sin(x) rounded
0	0	0
10	0.1736482	3
20	0.3420202	5
30	0.5	7
40	0.6427876	10
50	0.7660444	11
60	0.8660254	13
70	0.9396926	14
80	0.9848077	15
90	1	15

The following small program section in ROM creates the table in RAM:

```
SineTab           EQU    0

CreateSineTab:    STI    SineTab, 0
                  STI    SineTab + 1, 3
                  STI    SineTab + 2, 5
                  STI    SineTab + 3, 7
                  STI    SineTab + 4, 10
                  STI    SineTab + 5, 11
                  STI    SineTab + 6, 13
                  STI    SineTab + 7, 14
                  STI    SineTab + 8, 15
                  STI    SineTab + 9, 15
```

Of course we also could have used the **`STIX Dat`** and **`INC Reg`** instructions to build the table. But that would take more than twice the number of instructions compared to the example given above.
We now may use this table and look up values from it. The next program example assumes that the table index is passed in the accumulator and that we can use the index register to lookup the corresponding value from the table. It also assumes that the start of the table in RAM is at an arbitrary address defined by the symbol **`SineTab`**.

```
SineTabL          EQU    0
SineTabH          EQU    0

Lookup:           STA    XL                   ;save table index
                  STI    XH, SineTabH         ;set high nibble of index register
                                              ;to high nibble of table base address
                  LDI    SineTabL             ;get low nibble of table base address
                  ADD    XL                   ;add table index
                  STA    XL                   ;save result in index register
                  JPNC   Lookup2
                  INC    XH                   ;if Carry then increment high nibble
                  STA    XH                   ;save result in index register
Lookup2:          LDRX                        ;get value from table
```

This program returns the value looked up from the table in the accumulator.

B. Large Tables

Sometimes there may not be sufficient RAM to use the algorithm discussed before. Fortunately, the 4 bits uC provides enough ROM to use one of the following algorithms, which are not as elegant as the RAM approach, but provide the functionality of table lookup in ROM.

We will again use the table from the previous example. This is not really a large table, but using the same table will give you a better opportunity to compare the various algorithms.

For better understanding we will first describe each ROM table lookup algorithm in terms of a high level pseudo language (HLL). Then we will show its compiled version in 4 bits uC assembler language.

1) Linear Lookup Algorithm

HLL Algorithm

This program looks up a value Y as a function of an index X. It uses a decision structure (SELECT CASE structure) which is common to many HLL languages. This is a simple linear lookup algorithm which is easy to program. Its functionality is equivalent to that of a table lookup.

```
LookupSineTab:
     SELECT CASE X
        CASE 0: Y = 0
        CASE 1: Y = 3
        CASE 2: Y = 5
        CASE 3: Y = 7
        CASE 4: Y = 10
        CASE 5: Y = 11
        CASE 6: Y = 13
        CASE 7: Y = 14
        CASE 8: Y = 15
        CASE 9: Y = 15
        CASE ELSE: 'do nothing
       END SELECT
```

EM66xx Assembler Language

The table index is passed to the program in the accumulator and is stored in a temporary register **Temp**. The program returns with the lookup value in the accumulator.

```
LookupSineTab:
       ;SELECT CASE X
         STA  Temp
        ;CASE 0: Y = 0
          JPNZ Case1          ; test for equality
          LDI 0               ; if equal then load lookup value
          JMP EndSelect       ; exit
        ;CASE 1: Y = 3
Case1:   DEC Temp             ;next index value
          STA Temp
          JPNZ Case2          ; test for equality
          LDI 3               ; if equal then load lookup value
          JMP EndSelect       ; exit
        ;CASE 2: Y = 5
Case2:   DEC Temp             ; ... and so on ...
          STA Temp
          JPNZ Case3
          LDI 5
          JMP EndSelect
        ;CASE 3: Y = 7
Case3:   DEC Temp
          STA Temp
          JPNZ Case4
          LDI 7
          JMP EndSelect
        ;CASE 4: Y = 10
Case4:   DEC Temp
          STA Temp
          JPNZ Case5
          LDI 10
          JMP EndSelect
        ;CASE 5: Y = 11
Case5:   DEC Temp
          STA Temp
          JPNZ Case6
          LDI 11
          JMP EndSelect
        ;CASE 6: Y = 13
Case6:   DEC Temp
```

```
                STA Temp
                JPNZ Case7
                LDI 13
                JMP EndSelect
              ;CASE 7: Y = 14
        Case7:   DEC Temp
                STA Temp
                JPNZ Case8
                LDI 14
                JMP EndSelect
              ;CASE 8: Y = 15
        Case8:   DEC Temp
                STA Temp
                SUB Temp
                JPNZ Case9
                LDI 15
                JMP EndSelect
              ;CASE 9: Y = 15
        Case9:   DEC Temp
                STA Temp
                JPNZ CaseElse
                LDI 15
                JMP EndSelect
              ;CASE ELSE: 'do nothing
        CaseElse:
             ;END SELECT
        EndSelect:
```

Of course, in its compiled version, this program does not look very elegant any more. And it uses quite a number of instructions, which means that it consumes ROM space. But always bear in mind, that this is exactly the idea behind any RISC architecture: To use an inexpensive CPU with low cost memory in an optimised combination that is more software oriented than another based solutions.

It should be noted that the HLL algorithm used in this example still has some room for improvement. For example it has not been optimised with regard to speed. It works in a linear fashion which means, that for a table with *n* entries and an equal probability for each of the *n* possible index values to occur, an average of *n/2* steps is required to lookup a value. However, values for smaller indices will be looked up faster than for larger ones.

2) Binary Lookup Algorithm

If a binary decision algorithm is used instead of the linear algorithm demonstrated.
The number of steps will be the same for each value of the table index.

HLL Algorithm

```
 BinLookupSineTab:
  CONST Y0 = 0
  CONST Y1 = 3
  CONST Y2 = 5
  CONST Y3 = 7
  CONST Y4 = 10
  CONST Y5 = 11
  CONST Y6 = 13
  CONST Y7 = 14
  CONST Y8 = 15
  CONST Y9 = 15
```

```
C = ShiftLeft(X)              'Bit 3
IF C = 0 THEN                 '0 <= X < 8
 C = ShiftLeft(X)             'Bit 2
 IF C = 0 THEN                '0 <= X < 4
  C = ShiftLeft(X)            'Bit 1
  IF C = 0 THEN  '0 <= X < 2
   C = ShiftLeft(X)           'Bit 0
   IF C = 0 THEN Y = Y0 ELSE Y = Y1
  ELSE                        '2 <= X < 4
   C = ShiftLeft(X)           'Bit 0
   IF C = 0 THEN Y = Y2 ELSE Y = Y3
  ENDIF
 ELSE                         '4 <= X < 8
  C = ShiftLeft(X)            'Bit 1
  IF C = 0 THEN  '4 <= X < 6
   C = ShiftLeft(X)           'Bit 0
   IF C = 0 THEN Y = Y6 ELSE Y = Y5
  ELSE                        '6 <= X < 8
   C = ShiftLeft(X)           'Bit 0
   IF C = 0 THEN Y = Y8 ELSE Y = Y7
  ENDIF
 ENDIF
ELSE                          '8 <= X < 16
 C = ShiftLeft(X)             'Bit 2
 IF C = 0 THEN                '8 <= X < 12
  C = ShiftLeft(X)            'Bit 1
  IF C = 0 THEN  '8 <= X < 10
   C = ShiftLeft(X)           'Bit 0
   IF C = 0 THEN Y = Y8 ELSE Y = Y9
    ELSE               '10 <= X < 12       ')
   C = ShiftLeft(X)           'Bit 0       ') not needed here
   IF C = 0 THEN Y = Y10 ELSE Y = Y11      ')
  ENDIF
 ELSE                         '12 <= X < 16')
  C = ShiftLeft(X)            'Bit 1       ')
  IF C = 0 THEN  '12 <= X < 14             ')
   C = ShiftLeft(X)           'Bit 0       ')
   IF C = 0 THEN Y = Y12 ELSE Y = Y13      ') not needed here
  ELSE                        '14 <= X < 16')
   C = ShiftLeft(X)           'Bit 0       ')
   IF C = 0 THEN Y = Y14 ELSE Y = Y15      ')
  ENDIF                                    ')
 ENDIF
ENDIF
```

Note that in this example *n = 10*. This would also include index values larger than 10 up to 15. Therefore portions of the program handling these values may be omitted. In this special example the speed advantage over the linear version is not yet very significant. However, already for slightly larger tables the benefits will be notable.

The major disadvantage of the binary lookup algorithm is, that it is not easy to program, even if high level tools (macro processor or HLL compiler) were available. Also, as the HLL version of this algorithm looks much more complicated than the linear one, it might be expected that the assembler version would also be very large. However, the size of the compiled program does not differ too much from that of the linear algorithm.

EM66xx Assembler Language

The table index is passed to the program in the accumulator and is stored in a temporary register **Temp**. The program returns with the lookup value in the accumulator.

```
BinLookupSineTab:
       Y0            EQU  0
       Y1            EQU  3
       Y2            EQU  5
       Y3            EQU  7
       Y4            EQU  10
       Y5            EQU  11
       Y6            EQU  13
       Y7            EQU  14
       Y8            EQU  15
       Y9            EQU  15

                     STA Temp
                     SHL Temp     ;C = ShiftLeft(X)   'Bit
       STA Temp
                     JPC Else8    ;IF C = 0 THEN             '0 <= X < 8
                     SHL Temp     ; C = ShiftLeft(X)         'Bit 2
                     STA Temp
                     JPC Else4    ; IF C = 0 THEN            '0 <= X < 4
                     SHL Temp     ;  C = ShiftLeft(X)        'Bit 1
                     STA Temp
                     JPC Else2    ;  IF C = 0 THEN           '0 <= X < 2
                     SHL Temp     ;   C = ShiftLeft(X)       'Bit 0
                     STA Temp
                     JPC Else1    ;   IF C = 0 THEN Y = Y0 ELSE Y = Y1
                     LDI Y0
                     JMP Exit
       Else1:        LDI Y1
                     JMP Exit     ;   EXIT
       Else2:                     ;  ELSE                    '2 <= X < 4
                     SHL Temp     ;   C = ShiftLeft(X)       'Bit 0
                     STA Temp
                     JPC Else3    ;   IF C = 0 THEN Y = Y2 ELSE Y = Y3
                     LDI Y2
                     JMP Exit
       Else3: LDI Y3
                     JMP Exit     ;   EXIT
                                  ;  ENDIF
       Else4:                     ; ELSE              '4 <= X < 8
                     SHL Temp     ;  C = ShiftLeft(X)        'Bit 1
                     STA Temp
                     JPC Else6    ;  IF C = 0 THEN           '4 <= X < 6
                     SHL Temp     ;   C = ShiftLeft(X)       'Bit 0
                     STA Temp
                     JPC Else5    ;   IF C = 0 THEN Y = Y4 ELSE Y = Y5
                     LDI Y4
                     JMP Exit
       Else5: LDI Y5
                     JMP Exit     ;   EXIT
       Else6:                     ;  ELSE                    '6 <= X < 8
                     SHL Temp     ;   C = ShiftLeft(X)       'Bit 0
                     STA Temp
```

```
             JPC Else7     ;   IF C = 0 THEN Y = Y6 ELSE Y = Y7
             LDI Y6
             JMP Exit
Else7: LDI Y7
             JMP Exit      ;   EXIT
                           ;  ENDIF
                           ; ENDIF
Else8:                     ;ELSE                     '8 <= X < 16
             SHL Temp      ; C = ShiftLeft(X)        'Bit 2
             STA Temp
             JPC Exit      ; IF C = 0 THEN           '8 <= X < 12
             SHL Temp      ;  C = ShiftLeft(X)       'Bit 1
             STA Temp
             JPC Exit      ;  IF C = 0 THEN          '8 <= X < 10
             SHL Temp      ;   C = ShiftLeft(X)      'Bit 0
             STA Temp
             JPC Else9     ;   IF C = 0 THEN Y = Y8 ELSE Y = Y9
             LDI Y8
             JMP Exit
Else9: LDI Y9
                           ;   EXIT
                           ;  ENDIF
                           ; ENDIF
                           ;ENDIF
Exit:
```

Application Note 7

9. How to implement N-way branching

Some application programs need special control structures which are commonly referred to as *N-way branching* and *indirect branching*.

In high level languages (HLL) *N-way branching* may take the form

```
ON X GOTO Label1, Label2, Label3, Label4 'etc.
```

where *X* is an index variable with values in the range between 1 and 4 for this example. The program jumps to one of the target labels *Label1* through *Label4* depending on the value of *X*. In assembler language *N-way branching* is often implemented by using indexed jump tables in memory, from which the target address is looked up and transferred to the program counter. Special jump instructions providing indexed indirect branching may be available for this purpose.

Indirect branching is a form of branching where the target address of a jump instruction is not passed to the instruction directly as an argument, but is assigned to a RAM or register variable, whose address becomes the argument of the indirect jump instruction. In a HLL representation *indirect branching* would take the form

```
        Vector = Label1
        JumpIndirect (Vector)
        '...
        Vector = Label2
        JumpIndirect (Vector)
        '...
Label1:
        '...
Label2:
        '...
```

Some microprocessors provide special instructions to accomplish this. Usually these instructions require at least two instruction cycles: one cycle to fetch the instruction itself from ROM, and the second cycle to transfer the RAM data to the program counter. More cycles and more instruction words are required for complex instructions with table index operands. Such complex instructions are typical for an another processors.

As a RISC processor, the 4 bits uC follows a different approach by deliberately restricting the instruction set for the benefits of reduced chip area and speed, and emulating complex instructions in software.
The 4 bits uC has separate memories for instructions (ROM) and data (RAM). Each of these memories has its own address space, which may be accessed by two subsets of the instruction set, one for each address space. The instructions referencing ROM addresses typically change the flow of control in a program, such as JMP, CALL, RET instructions etc. The instructions referencing RAM addresses are typically data transfer and data manipulation instructions. This architecture has some similarity with some HLL programming languages where data and program code are separated, i.e. a program cannot modify itself in memory.

Therefore, in order to implement *N-way branching* and *Indirect branching* with the 4 bits uC, one has to use a HLL model.

For better understanding of each of the following examples we will first describe each algorithm in terms of a high-level pseudo language (HLL). Then we will show its compiled version in 4 bits uC assembler language.

A. N-way branching

HLL Algorithm
The following HLL algorithm provides *N-way branching* by using a decision structure (SELECT CASE structure) which is common to many HLL languages. This is a simple linear lookup algorithm which is easy to program. Its functionality is equivalent to that of a table lookup. This program example jumps to one of 4 target locations which is a function of an index *X*.

```
OnGoto:
       SELECT CASE X
        CASE 1: GOTO Label1
        CASE 2: GOTO Label2
        CASE 3: GOTO Label3
        CASE 4: GOTO Label4
       END SELECT
       '...
Label1:
       '...
Label2:
       '...
Label3:
       '...
Label4:
       '...
```

EM66xx Assembler Language
The HLL model may compiled into 4 bits uC assembler language as shown below. The index is passed to the program in the accumulator and is stored in a temporary register **`Temp`**. The program jumps to the label associated with the current index.

```
OnGoto:
       ;SELECT CASE X
         STA  Temp
        ;CASE 1: GOTO Label1
Case1:    DEC Temp          ;first index value
          STA Temp
          JPZ Label1;branch if equal
        ;CASE 2: GOTO Label2
Case2:    DEC Temp          ;next index value
          STA Temp
          JPZ Label2;branch if equal
        ;CASE 3: GOTO Label3
Case3:    DEC Temp          ; ... and so on ...
          STA Temp
          JPZ Label3
        ;CASE 4: GOTO Label4
Case4:    DEC Temp
          STA Temp
          JPZ Label4
       ;END SELECT
EndSelect:
       ;...
Label1:
       ;...
Label2:
       ;...
Label3:
       ;...
Label4:
       ;...
```

Even in its compiled version, this program is still very simple. Of course it uses more instructions than a jump table would need, which means that it consumes ROM space. But always bear in mind, that this is exactly the idea behind any RISC architecture: To use an inexpensive CPU with low cost memory in an optimised combination that is more software oriented than another based solutions.

It should be noted that the HLL algorithm used in this example still has some room for improvement. For example it has not been optimised with regard to speed. It works in a linear fashion which means, that for *n* branches and an equal probability for each of the *n* possible index values to occur, an average of *n/2* decisions is required to branch to a label. However, for smaller indices the branch will be faster than for larger ones.

If a binary decision algorithm is used instead of the linear algorithm demonstrated above, the number of decisions until a branch is taken - the binary logarithm of *n* rounded to the next higher integer - and the number of decisions will be the same for each value of the index. For more details on binary decision algorithms please see **AppNote # 6: *Creating Data Tables in ROM.***

B. Indirect branching

As the EM66xx does not provide any instructions that can modify the PC register directly, it might appear that *indirect branching* would not be not possible. However, with the right software architecture, which borrows again from a HLL model, a form of *indirect branching* may be implemented which is safer than an another assembler model and has considerable advantages with regard to software quality and maintainability.

HLL Algorithm

While the classical model for *indirect addressing* uses an *address vector* variable to which an *actual address* is assigned, the model shown below uses an *address index* variable to which an *address index* (also called a *handle*) is assigned, which only *identifies* the actual address. This *handle* is used by a dispatcher code section, which uses the *N-way branching* algorithm explained before, to jump to the address associated with the current value of the *handle*.

```
      Handle = 1
      GOTO Dispatch
      '...
      Handle = 2
      GOTO Dispatch
      '...

Dispatch:
      SELECT CASE Handle
       CASE 1: GOTO Label1
       CASE 2: GOTO Label2
      END SELECT
      '...
Label1:
      '...
Label2:
      '...
```

This model has the advantage that all jump addresses are located in one relatively small section of the program: the dispatcher. This makes it much easier to debug or modify a program and to ensure that only valid addresses are used.

4 bits uC Assembler Language

As can be seen, the compiled version of this algorithm is also very simple. The *handle* is passed to the dispatcher in the accumulator and is stored in a temporary variable **`Temp`**.

```
       ;Handle = 1
        LDI 1
       ;GOTO Dispatch
        JMP Dispatch
       ;...
       ;Handle = 2
        LDI 2
       ;GOTO Dispatch
        JMP Dispatch
       ;...

Dispatch:
       ;SELECT CASE Handle
         STA  Temp
        ;CASE 1: GOTO Label1
Case1:   DEC Temp         ;first index value
          STA Temp
          JPZ Label1;branch if equal
        ;CASE 2: GOTO Label2
Case2:   DEC Temp         ;next index value
          STA Temp
          JPZ Label2;branch if equal
       ;END SELECT
EndSelect:

;...
Label1:
;...
Label2:
;...
```

Application Note 8

10. Immediate ALU instructions

The 4 bits uC provides the following *Immediate* ALU instructions.

```
STI    Reg, Dat     ;R(Reg) = Dat
STIX   Dat          ;R(Idx) = Dat
LDI    Dat          ;A = Dat: SetZ
```

These instructions are called *Immediate* because they contain an operand **Dat** that represents a constant 4-bit value which is stored in ROM together with the instruction word and will be transferred *immediately* to the destination register specified by the instruction when the instruction is executed.
The comment to the right side of the instruction overview shown above contains a short formal definition of what the instruction actually does in terms of the registers and flags being affected.

Each instruction of the 4 bits uC microcontroller is stored in ROM as a fixed length 16-bit data word containing the operator and the operands defined for that instruction.

An individual explanation of the various *Immediate* instructions is given in the following sections. The instruction format definitions use the bit order defined below (*Bit15* down to *Bit0*):

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|15 |14 |13 |12 |11 |10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

A. Store Data Immediate in Register

Mnemonics *STI Reg, Dat*
Formal description R(Reg) = Dat

Binary Code

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 1 | 0 | 0 |d3 |d2 |d1 |d0 | 0 |r6 |r5 |r4 |r3 |r2 |r1 |r0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

d3..d0	data bits of operand **Dat,** representing a constant value between 0 and 15
r6..r0	data bits of operand **Reg,** representing a register address (RAM) between 0 and 127

Explanation The **STI** instruction stores the value specified by its second operand **Dat** in the register whose address is specified by the first operand **Reg**. The symbols **Reg** and **Dat** are only placeholders in this definition. They may be replaced by any numeric value or any previously defined symbol or expression that the assembler language permits and which evaluates to a value within the valid range specified for each operand.

B. Store Data Immediate in Register pointed to by Index Register

Mnemonics	***`STIX    Dat`***
Formal description	`R(Idx) = Dat`

Binary Code

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 1 | 0 | 0 |d3 |d2 |d1 |d0 | 1 | x | x | x | x | x | x | x |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

`d3..d0`	data bits of operand **`Dat,`** representing a constant value between 0 and 15
`x`	undefined bits, ignored by the microcontroller

Explanation

The **`STIX`** instruction stores the value specified by its operand **`Dat`** in the register whose address is currently stored in the index register of the microcontroller. The index register is a 7-bit register which is the combination of a 4-bit register **`XL`** and a 3-bit register **`XH`** which may be accessed at register addresses **`06EH`** and **`06FH`** respectively. The symbol **`Dat`** is only a placeholder in this definition. It may be replaced by any numeric value or any previously defined symbol or expression that the assembler language permits and which evaluates to a value within the valid range specified for the operand.

C. Store Data Immediate in Accumulator

Mnemonics	***`LDI     Dat`***
Formal description	`A = Dat: SetZ`

Binary Code

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | x | x | x | x |d3 |d2 |d1 |d0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

`d3..d0`	data bits of operand **`Dat,`** representing a constant value between 0 and 15
`x`	undefined bits, ignored by the microcontroller

Explanation

The **`LDI`** instruction stores the value specified by its operand **`Dat`** in the accumulator of the microcontroller. The symbol **`Dat`** is only a placeholder in this definition. It may be replaced by any numeric value or any previously defined symbol or expression that the assembler language permits and which evaluates to a value within the valid range specified for the operand. This instruction sets the Zero flag **`Z`** of the microcontroller if **`Dat = 0`** and clears it if **`Dat <> 0`**.

Application Note 9

11. How to increment the accumulator

Increment **and** ***Decrement*** **Instructions**

The 4 bits microcontroller provides a number of instructions which load a register value into the accumulator and then increment or decrement the accumulator. The following is a complete list of these instructions, including the instruction mnemonics and a formal description of the operations performed by the instructions in terms of the registers and flags being affected. Further details of the formal description are given in the ***4 bits uC Programming Model*** documentation.

```
INC    Reg          ;A = R(Reg) + 1: AddC: SetZ
DEC    Reg          ;A = R(Reg) - 1: SubC: SetZ
INCX                ;A = R(Idx) + 1: AddC: SetZ
DECX                ;A = R(Idx) - 1: SubC: SetZ
INCS   Reg          ;INC  Reg: SHRA
DECS   Reg          ;DEC  Reg: SHRA
INCXS               ;INCX  : SHRA
DECXS               ;DECX  : SHRA
```

As can be seen, there is no instruction that increments the accumulator without loading data from a register into the accumulator. Therefore, if one wants to increment a value that is already in the accumulator, one has to write a small piece of code that does the job. The following two code examples emulate *Increment Accumulator* and *Decrement Accumulator* instructions.

INCA ***- Increment Accumulator***

The **`INCA`** function can be programmed with just two instructions and by using a temporary register **`Temp`**:

```
INCA:        STA Temp      ;R(Temp) = A
             INC Temp      ;A = R(Temp) + 1
```

With the first instruction the value in the accumulator is stored in **`Temp`** , which then is loaded back into the accumulator and incremented with the second instruction.

DECA ***- Decrement Accumulator***

The **`DECA`** function can be programmed with just two instructions and by using a temporary register **`Temp`**:

```
DECA:        STA Temp      ;R(Temp) = A
             DEC Temp      ;A = R(Temp) - 1
```

With the first instruction the value in the accumulator is stored in **`Temp`** , which then is loaded back into the accumulator and decrement with the second instruction.

Application Note 11

12. Recursive Calculation with Lookup Tables

HLL Algorithm

Before we discuss how *Recursive Calculation Using a Lookup Table* may be coded in 4 bits uC assembler language, we will describe the function formally by the following high level language (HLL) algorithm, which gives a better understanding of what is needed:

```
    CONST IterationCount = 4             'definition of parameters
    DIM I, Par, Result, InitialValue     'variables
    CONST TableSize = 10
    DIM Table(1 TO TableSize )           'definition of lookup table
    CreateTable Table()                  'assign values to Table()

    'main program is placed here and uses Recursion subroutine

    END

Recursion:
    Par = InitialValue                   'set initial value of Par
    FOR I = 1 TO IterationCount          'iteration loop
     A = Table(Par)                      'lookup value from table
     Par = Calculation(A)                'perform some calculation
    NEXT
    Result = Par                         'set result to Par
    RETURN
```

EM66xx Implementation for 4-bit integers

The challenge of this algorithm with respect to the 4 bits uC is how to implement the table lookup function. The 4 bits uC cannot read data from ROM directly. However, as shown in Application Note #6 (*Creating Data Tables in ROM)*, by using the *immediate* instructions of the 4 bits uC, small data tables can be initialised in RAM, while lookup from larger data tables can be emulated by a **SELECT CASE** construct. The algorithm shown above translates into the following EM66xx assembler program, which represents the simplified case where all variables hold 4-bit integers:

```
    ;CONST IterationCount = 4         'definition of parameters
IterationCount    EQU 4

    ;DIM I, Par, Result, InitialValue
I                 EQU    1
Par               EQU    2
Result            EQU    3
InitialValue      EQU    4

    ;CONST TableSize = 10
    ;DIM Table(1 TO TableSize )       'definition of lookup table
    ;CreateTable Table()
Table             EQU    16
TableL            EQU    16
TableH            EQU    0

CreateTable:
    STI    Table, 0
    STI    Table + 1, 3
    STI    Table + 2, 5
    STI    Table + 3, 7
    STI    Table + 4, 10
    STI    Table + 5, 11
    STI    Table + 6, 13
    STI    Table + 7, 14
    STI    Table + 8, 15
    STI    Table + 9, 15

    ;main program is placed here and uses Recursion subroutine
    ;NOTE that only two subroutine levels are available and that interrupts
    ;have to be disabled when the second level is used !!!

    ;END
END_:
    JMP END_

Recursion:
    ;Par = InitialValue                 'set initial value of Par
     LDR InitialValue
     STA Par

    ;FOR I = 1 TO IterationCount        'iteration loop
     STI IterationCount, I
FOR:
    ; A = Table(Par)                    'lookup value from table
      LDR Par
      CALL Lookup
    ; Par = Calculation(A)              'perform some calculation
      CALL Calculation
      STA Par
    ;NEXT
     DEC I
     JPNZ FOR

    ;Result = Par                       'set result to Par
     LDR Par
     STA Result

    ;RETURN
     RET

Calculation:
    ;dummy
    RET

Lookup:
    STA    XL                           ;save table index
    STI    XH, TableH                   ;set high nibble of index register
                                        ;to high nibble of table base address
    LDI    TableL                       ;get low nibble of table base address
    ADD    XL                           ;add table index
    STA    XL                           ;save result in index register
    JPNC   Lookup2
    INC    XH                           ;if Carry then increment high nibble
    STA    XH                           ;save result in index register
Lookup2:
    LDRX                                ;get value from table
    RET
```

4 bits uC Implementation for 8-bit integers

The 4 bits uC is a 4-bit processor (as far as its data word length is concerned). Thus any data operations requiring larger data word lengths must use multiple 4-bit words. The following assembler program extends the algorithm given above for 8-bit data. As the lookup table will be much larger than in the previous case, a different lookup algorithm based on a **`SELECT CASE`** construct is used. For details of that algorithm refer to Application Note #6 (*Creating Data Tables in ROM)*.

```
;CONST IterationCount = 4      'definition of parameters
IterationCount    EQU 4

    ;DIM I, Par, Result, InitialValue
I                 EQU    1
ParL       EQU    2
ParH       EQU    3
ResultL           EQU    4
ResultH           EQU    5
InitialValueL     EQU    6
InitialValueH     EQU    7
TempL             EQU    8
TempH             EQU    9

    ;main program is placed here and uses Recursion subroutine
    ;NOTE that only two subroutine levels are available and that interrupts
    ;have to be disabled when the second level is used !!!

    ;END
END_:
    JMP END_

Recursion:
    ;Par = InitialValue              'set initial value of Par
     LDR InitialValueL
     STA ParL
     LDR InitialValueH
     STA ParH

    ;FOR I = 1 TO IterationCount     'iteration loop
     STI IterationCount, I
FOR:
    ; Temp = Table(Par)              'lookup value from table
      CALL Lookup
    ; Par = Calculation(Temp)        'perform some calculation
      CALL Calculation
      STA Par
    ;NEXT
     DEC I
     JPNZ FOR
    ;Result = Par                    'set result to Par
     LDR ParL
     STA ResultL
     LDR ParH
     STA ResultH

    ;RETURN
     RET
```

```
Calculation:
    ;dummy
    LDR TempL
    STA ParL
    LDR TempH
    STA ParH
    RET

Lookup:
    ;the algorithm required here has the following structure:
    ; SELECT CASE ParH
    ;  CASE X1H
    ;   SELECT CASE ParL
    ;    CASE X11L
    ;      TempL = Y11L
    ;      TempH = Y11H
    ;    CASE X12L
    ;      '...
    ;   END SELECT
    ;  CASE X2H
    ;   SELECT CASE ParL
    ;    CASE X21L
    ;      TempL = Y21L
    ;      TempH = Y21H
    ;    CASE X22L
    ;      '...
    ;   END SELECT
    ;   '...
    ; END SELECT

    ;see Application Note #6 for details
    ;The following code is just a test dummy:
    LDR ParH
    STA TempH
    LDR ParL
    STA TempL

    RET
```

Conclusions

As can be seen from the previous example, the emulation of 8-bit table lookup may require a substantial amount of code. It should be noted that table lookup - while being a familiar and convenient solution in certain environments and applications - is not the only solution for a given problem.

Depending on the specific application, it might be possible for example, to use *Polynomic Functions* instead of *Recursive Calculation with Lookup Tables.* Polynoms need far less constants than lookup tables and the code size is directly proportional to the number of coefficients. Polynoms may be used to approximate contiguous functions sufficiently well. If step or pulse functions are needed, the code may be even reduced two very few instructions. If non-contiguous functions are involved, they may be approximated piece wise by contiguous functions.

In any case a careful analysis has to be made to find out which approach is most suited for a given application. There is no general solution and the most efficient results are likely to be obtained when the actual requirements of the application are well understood.

References: AppNote #6, *Creating Data Tables in ROM*

Application Note 13

13. 16 bit binary division with 4 bit controller

The following document describes the implementation of the restoring division algorithm for a 16 bit integer division using a 4 bits processor. The result is a 16 bit integer and a 16 bit remainder. The process has been implemented as a subroutine. The program hasn't tests for check the division by 0.

The structure of the 16 bit operands and result is the following:

```
bit15               bit0
+----+----+----+----+
¦0000¦0000¦0000¦0000¦
+----+----+----+----+
OpX_3               OpX_0
```

The operation performed is : `Operand1 / Operand2 = Result ( Remainder )`

Variables : `Op1_X     / Op2_X     = ResX    ( RstX )`

```
;----------------------------------------------------------------------------------------------------
;       MODULE :        DIVIS16.ASM
;       Date last mod   :       04/12/1998              Ch. Mayer
;       Module for division (If the result is too big of FFFFH, it's fault)
;----------------------------------------------------------------------------------------------------
;
;       Variables:
;
;       OP1_1   :       First number (LSB)
;       OP1_2   :       First number
;       OP1_3   :       First number
;       OP1_4   :       First number (MSB)
;       OP2_1   :       Second number (LSB)
;       OP2_2   :       Second number
;       OP2_3   :       Second number
;       OP2_4   :       Second number (MSB)
;       Res1    :       Resultat (LSB)
;       Res2    :       Resultat
;       Res3    :       Resultat
;       Res4    :       Resultat (MSB)
;       Rst1    :       Rest (LSB)
;       Rst2    :       Rest
;       Rst3    :       Rest
;       Rst4    :       Rest (MSB)
;       ComptL  :       Counter of calcul low
;       ComptH  :       Counter of calcul high
;       Carry   :       Save the carry, first memory
;       Carry2  :       Save the carry, second memory
;       Stack1  :       \
;       Stack2  :         => Save temporaly the calcul's value
;       Stack3  :        /
;       Stack4  :       /
;
;------------------------------------------------------------------------------------------
Divi16:
        STI     ComptL, 00H     ; 16 clocks at calcul
        STI     ComptH, 01H
        LDR     OP1_1           ; Load the first value in the result for the calcul
        STA     Res1
        LDR     OP1_2
        STA     Res2
        LDR     OP1_3
        STA     Res3
        LDR     OP1_4
        STA     Res4
        STI     Rst1, 00H
        STI     Rst2, 00H
        STI     Rst3, 00H
        STI     Rst4, 00H

Div100: LDR     OP2_4           ; Substract the fourth value at the rest (MSB)
        SUB     Rst4
        STA     Stack4
        JPC     Div110          ; If the carry is 0, the result is negative, don't jump
        DEC     Carry2          ; If the result is negative, decrement the previous carry
        JPNC    Div180          ; If is always negative, jump to "Div180"
        STA     Carry2          ; Else memories

Div110: LDR     OP2_3           ; Substract the third value at the rest
        SUB     Rst3
        STA     Stack3
        JPC     Div120          ; If the carry is 0, the result is negative
        DEC     Stack4          ; If the result is negative, decrement the "MSB" bit
        STA     Stack4
        JPC     Div120          ; If is always negative, don't jump
        DEC     Carry2          ; If the result is negative, decrement the previous carry
        JPNC    Div180          ; If is always negative, jump to "Div180"
        STA     Carry2          ; Else memories

Div120: LDR     OP2_2           ; Substract the second value at the rest
        SUB     Rst2
        STA     Stack2
        JPC     Div130          ; If the carry is 0, the result is negative
        DEC     Stack3          ; If the result is negative, decrement the previous bit
        STA     Stack3
        JPC     Div130          ; If is always negative, don't jump
        DEC     Stack4          ; If the result is negative, decrement the previous bit
        STA     Stack4
        JPC     Div130
        DEC     Carry2          ; If the result is negative, decrement the previous carry
        JPNC    Div180          ; If is always negative, jump to "Div180"
        STA     Carry2          ; Else memories

Div130: LDR     OP2_1           ; Substract the first value at the rest (LSB)
        SUB     Rst1
        STA     Stack1
        JPC     Div150          ; If the carry is 0, the result is negative
        DEC     Stack2          ; If the result is negative, decrement the previous bit
        STA     Stack2
        JPC     Div150          ; If is always negative, don't jump
        DEC     Stack3          ; If the result is negative, decrement the previous bit
        STA     Stack3
        JPC     Div150          ; If is always negative, don't jump
        DEC     Stack4          ; If the result is negative, decrement the previous bit
        STA     Stack4
        JPC     Div150          ; If is always negative, don't jump
        DEC     Carry2          ; If the result is negative, decremente the previous carry
```

```
            JPNC    Div180          ; If is always negative, jump to "Div180"
            STA     Carry2          ; Else memories

Div150:     LDR     Stack4          ; Memories the new value after calcul
            STA     Rst4
            LDR     Stack3
            STA     Rst3
            LDR     Stack2
            STA     Rst2
            LDR     Stack1
            STA     Rst1

            INC     Res1            ; Add the value one if the result is positive
            STA     Res1
            JPNC    Div180
            INC     Res2
            STA     Res2
            JPNC    Div180
            INC     Res3
            STA     Res3
            JPNC    Div180
            INC     Res4
            STA     Res4
            JPNC    Div180
            INC     Rst1
            STA     Rst1
            JPNC    Div180
            INC     Rst2
            STA     Rst2
            JPNC    Div180
            INC     Rst3
            STA     Rst3
            JPNC    Div180
            INC     Rst4
            STA     Rst4

Div180:     DEC     ComptL          ; Compt bis to 16 calcul, after it's finish
            STA     ComptL
            JPNC    Div300

Div200:     SHLR    Res1            ; Shift left with the all bits
            STA     Res1
            JPNC    Div205
            STI     Carry, 01H
            JMP     Div210
Div205:     STI     Carry, 00H

Div210:     SHLR    Res2
            STA     Res2
            JPNC    Div215
            STI     Carry2, 01H
            JMP     Div220
Div215:     STI     Carry2, 00H

Div220:     LDR     Carry
            ADD     Res2
            STA     Res2
            SHLR    Res3
            STA     Res3
            JPNC    Div225
            STI     Carry, 01H
            JMP     Div230              ; Shift left with the all bits
Div225:     STI     Carry, 00H

Div230:     LDR     Carry2
            ADD     Res3
            STA     Res3

            SHLR    Res4
            STA     Res4
            JPNC    Div235
            STI     Carry2, 01H
            JMP     Div240
Div235:     STI     Carry2, 00H

Div240:     LDR     Carry
            ADD     Res4
            STA     Res4

            SHLR    Rst1
            STA     Rst1
            JPNC    Div245
            STI     Carry, 01H
            JMP     Div250
Div245:     STI     Carry, 00H

Div250:     LDR     Carry2
            ADD     Rst1
            STA     Rst1

            SHLR    Rst2
            STA     Rst2
            JPNC    Div255
            STI     Carry2, 01H
            JMP     Div260
Div255:     STI     Carry2, 00H

Div260:     LDR     Carry
            ADD     Rst2
            STA     Rst2

            SHLR    Rst3
            STA     Rst3
            JPNC    Div265
            STI     Carry, 01H
            JMP     Div270
Div265:     STI     Carry, 00H
Div270:     LDR     Carry2
            ADD     Rst3
            STA     Rst3

            SHLR    Rst4
            STA     Rst4
            JPNC    Div275              ; Shift left with the all bits
            STI     Carry2, 01H
            JMP     Div280
Div275:     STI     Carry2, 00H

Div280:     LDR     Carry
            ADD     Rst4
            STA     Rst4
            JMP     Div100

Div300:     DEC     ComptH
            STA     ComptH
            JPC     Div200
            RET
```

Application Note 14

14. 16 bit binary multiplication with 4 bit controller

The following figure shows the implementation of an algorithm for a 16 bit multiplication using a 4 bit processor. The result is a 32 bit number with overflow being signalled with the carry flag. The process has been implemented as a subroutine. The basic principal involves successive shifts of the operands, one to the left and one to the right. This is followed by addition of the operand shifted to the left, in this case operand1, to the result if bit 0 of the least significant nibble of operand 2 is 1 (16 series of multiplication by 2 and additions). In this way the maximum number of loop cycles is 16 as opposed to a loop addition process where the number of cycles is variable.

The structure of the 16 bit operands is the following:

```
bit15           bit0
+----+----+----+----+
¦0000¦0000¦0000¦0000¦
+----+----+----+----+
OpX_3           OpX_0
```

```
;------------------------------------------------------------------------------------------------
;       MODULE :        MULTI16.ASM
;       Date last mod   :       30/11/1998              Ch.Mayer
;       Module for multiplication 16 bits, result on 32 bits
;------------------------------------------------------------------------------------------------
;
;         VARIABLES FOR THE CALCUL:
;
;       OP1_1   :       First number (LSB)
;       OP1_2   :       First number
;       OP1_3   :       First number
;       OP1_4   :       First number (MSB)
;       OP2_1   :       Second number (LSB)
;       OP2_2   :       Second number
;       OP2_3   :       Second number
;       OP2_4   :       Second number (MSB)
;       Res1    :       Resultat (LSB)
;       Res2    :       Resultat
;       Res3    :       Resultat
;       Res4    :       Resultat
;       Res5    :       Resultat
;       Res6    :       Resultat
;       Res7    :       Resultat
;       Res8    :       Resultat (MSB)
;       Compt   :       Count the position of the calcul
;       Carry   :       Save the carry, memory n°1
;       Carry2  :       Save the carry, memory n°2
;
;------------------------------------------------------------------------------------------------
Multi16:
        STI     Compt, 0FH      ; 16 times for the calcul
        LDR     OP2_1           ; Load the first value in the result for the calcul
        STA     Res1
        LDR     OP2_2
        STA     Res2
        LDR     OP2_3
        STA     Res3
        LDR     OP2_4
```

```
          STA      Res4
          STI      Res5, 00H
          STI      Res6, 00H
          STI      Res7, 00H
          STI      Res8, 00H


Multi100:
          LDI      0001b           ; If the first bit is at one, add the second value
          AND      Res1
          JPZ      Multi200

          LDR      OP1_1           ; Add the first data
          ADD      Res5
          STA      Res5
          JPNC     Multi110        ; Test the carry for the next 4 bits
          STI      Carry, 01H
          JMP      Multi120
Multi110:
          STI      Carry, 00H
Multi120:
          LDR      OP1_2           ; Add the second data
          ADD      Res6
          STA      Res6
          JPNC     Multi130        ; Test carry for the next 4 bits
          STI      Carry2, 01H
          JMP      Multi140
Multi130:
          STI      Carry2, 00H
Multi140:
          LDR      Carry           ; Add the carry at the data
          ADD      Res6
          STA      Res6
          JPNC     Multi145
          STI      Carry2, 01H

Multi145:
          LDR      OP1_3           ; Add the third data
          ADD      Res7
          STA      Res7
          JPNC     Multi150        ; Test carry for the next 4 bits
          STI      Carry, 01H
          JMP      Multi160
Multi150:
          STI      Carry, 00H
Multi160:
          LDR      Carry2          ; Add the carry at the data
          ADD      Res7
          STA      Res7
          JPNC     Multi165
          STI      Carry, 01H

Multi165:
          LDR      OP1_4           ; Add the fourth data
          ADD      Res8
          STA      Res8
          JPNC     Multi170        ; Test carry for the next 4 bits
          STI      Carry2, 08H
```

```
        JMP     Multi180
Multi170:
        STI     Carry2, 00H
Multi180:
        LDR     Carry           ; Add the carry at the data
        ADD     Res8
        STA     Res8
        JPNC    Multi185
        STI     Carry2, 08H

Multi185:
        JMP     Multi205


Multi200:
        STI     Carry2, 00H

Multi205:
        SHRR    Res8            ; Shift right the result value intermediare
        STA     Res8
        JPNC    Multi210        ; Test the carry for the next 4 bits
        STI     Carry, 08H
        JMP     Multi215
Multi210:
        STI     Carry, 00H
Multi215:
        LDR     Carry2          ; Add the carry at the result value
        ADD     Res8
        STA     Res8

        SHRR    Res7            ; Shift right the result value intermediare
        STA     Res7
        JPNC    Multi220        ; Test the carry for the next 4 bits
        STI     Carry2, 08H
        JMP     Multi225
Multi220:
        STI     Carry2, 00H
Multi225:
        LDR     Carry           ; Add the carry at the result value
        ADD     Res7
        STA     Res7

        SHRR    Res6            ; Shift right the result value intermediare
        STA     Res6
        JPNC    Multi230        ; Test the carry for the next 4 bits
        STI     Carry, 08H
        JMP     Multi235
Multi230:
        STI     Carry, 00H
Multi235:
        LDR     Carry2          ; Add the carry at the result value
        ADD     Res6
        STA     Res6

        SHRR    Res5            ; Shift right the result value intermediare
        STA     Res5
        JPNC    Multi240
        STI     Carry2, 08H
        JMP     Multi245
Multi240:
        STI     Carry2, 00H
Multi245:
        LDR     Carry           ; Add the carry for the next 4 bits
        ADD     Res5
        STA     Res5

        SHRR    Res4            ; Shift right the result value intermediare
        STA     Res4
        JPNC    Multi250        ; Test the carry for the next 4 bits
        STI     Carry, 08H
        JMP     Multi255
Multi250:
        STI     Carry, 00H
Multi255:
        LDR     Carry2          ; Add the carry at the result value
        ADD     Res4
        STA     Res4

        SHRR    Res3            ; Shift right the result value intermediare
        STA     Res3
        JPNC    Multi260   ; Test the carry for the next 4 bits
        STI     Carry2, 08H
        JMP     Multi265
Multi260:
        STI     Carry2, 00H
Multi265:
        LDR     Carry           ; Add the carry at the result value
        ADD     Res3
        STA     Res3

        SHRR    Res2            ; Shift right the result value intermediare
        STA     Res2
        JPNC    Multi270   ; Test the carry for the next 4 bits
        STI     Carry, 08H
        JMP     Multi275
Multi270:
        STI     Carry, 00H
Multi275:
        LDR     Carry2          ; Add the carry at the result value
        ADD     Res2
        STA     Res2

        SHRR    Res1            ; Shift right the result value intermediare
        STA     Res1
        LDR     Carry           ; Add the carry for the next 4 bits
        ADD     Res1
        STA     Res1

Multi280:
        DEC     Compt           ; Decrement the compteur
        STA     Compt
        JPC     Multi100

        RET
```

Application Note 19

15. How is the prescaler after reset

This application note describes the prescaler after a reset, on each version of micro-controller.
The normal behaviour of the prescaler doesn't change on the different versions. The difference is only after a reset on the first IRQ clock.
The table 1 describes the different prescaler interrupt after a reset for each version.

The reset of the prescaler is useful when you want to synchronise the prescaler with an external clock for example. But if you use the prescaler in the program, i.e. for a real time clock, you don't need to reset the prescaler after each interrupt.

For more detailed characteristics on the prescaler, refer to the specifications of each individual micro-controller. You can download the latest version of the specifications and some example's programs on our web site. (http://www.emmicroelectronic.com/)

Version	Interrupt source	First int. after reset
EM6x03	1Hz	1 clock
	8Hz	1 clock
	32Hz	1 clock
EM6x04	2Hz	1 clock
	8Hz	1 clock
	32Hz	1 clock
EM6x05	1Hz *	1 clock
	8Hz *	1 clock
	32Hz *	1 clock
EM6x17	1Hz	1 clock
	8Hz	½ clock
	32Hz	½ clock
EM6x20	1Hz	1 clock
	8Hz	½ clock
	32Hz	½ clock
EM6x21	1Hz	1 clock
	8Hz	½ clock
	32Hz	½ clock
EM6x22	1Hz	1 clock
	8Hz	½ clock
	32Hz	½ clock
EM6x40	1 Hz *	1 clock
	586Hz *	½ clock
	9.4kHz *	½ clock

*: With the base's frequency without divisor.

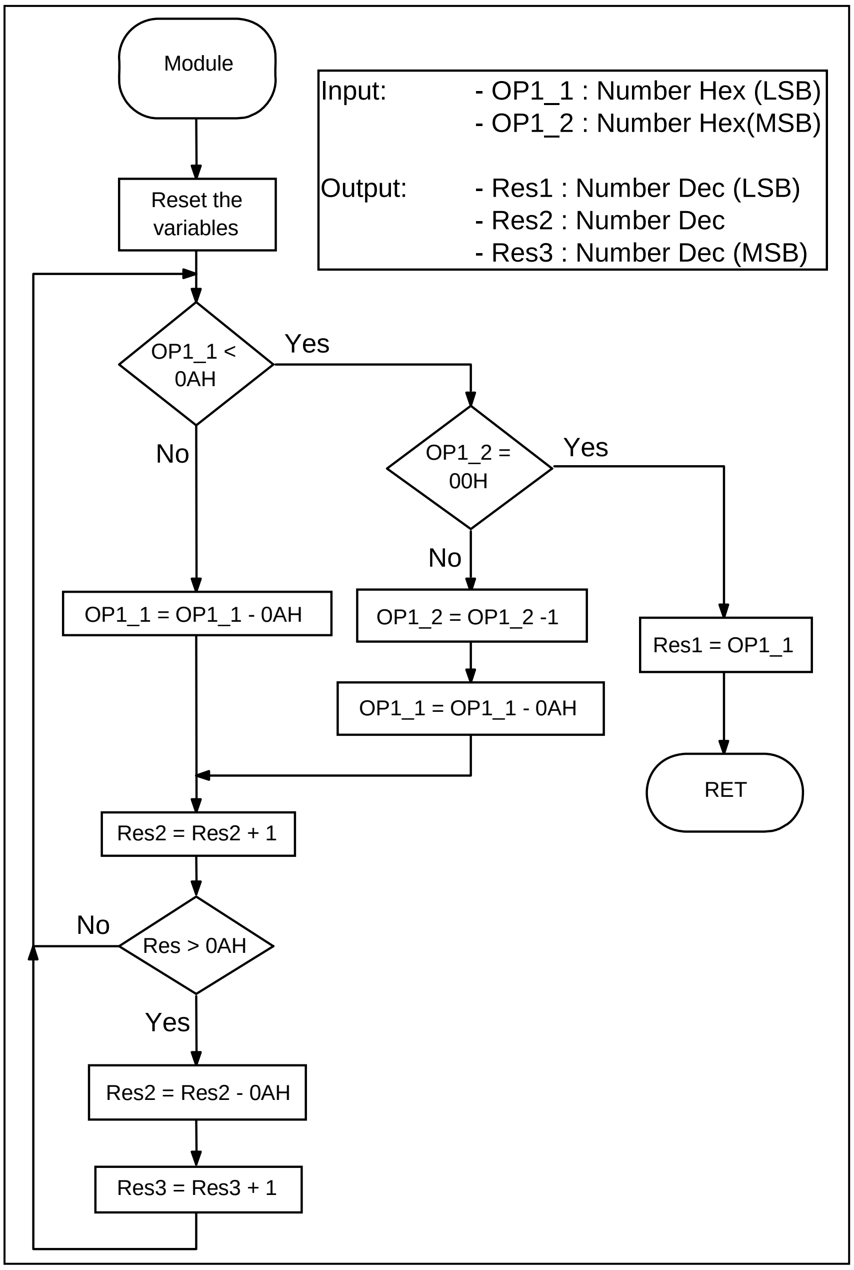
Application Note 20

16. How to convert Hex - Dec

The 4 bits uC does not provide instructions to convert a value hex - dec. Therefore it is necessary to implement the *converter hex - dec* functions by a small piece of code, which could be written as a subroutine as shown below.

This flowchart describes the module to convert Hex-Dec.

If you include this module in your software, you define the five variables in your principal file.



Appendix A: Hex – Dec module

```
;-------------------------------------------------------------------------------------------------------------
;            MODULE :              HEXDEC.ASM
;            Date last mod         :              20/02/1998                    Ch.Mayer
;            Module for convert hex - dec
;-------------------------------------------------------------------------------------------------------------
;            OP1_1     :           First number (LSB)
;            OP1_2     :           First number (MSB)
;            Res1      :           Resultat (LSB)
;            Res2      :           Resultat
;            Res3      :           Resultat (MSB)
;-------------------------------------------------------------------------------------------
HexDec:
             STI       Res1, 00H               ; Reset the variables
             STI       Res2, 00H
             STI       Res3, 00H

HexDec100:
             LDI       0AH                     ; If the LSB number isn't too big of 0AH (OP1_1 < 0AH)...
             SUB       OP1_1
             JPNC      HexDec200
             STA       OP1_1                   ; ... Then save the result
             JMP       HexDec250

HexDec200:                                     ; ... Else decrement the MSB value
             DEC       OP1_2
             JPNC      HexDec300
             STA       OP1_2
             LDI       0AH                     ; \
             SUB       OP1_1                   ;  => OP1_1 = OP1_1 - 0AH
             STA       OP1_1                   ; /

HexDec250:
             INC       Res2                    ; Increment the decimal value
             STA       Res2
             LDI       0AH
             SUB       Res2
             JPNC      HexDec100
             STA       Res2
             INC       Res3
             STA       Res3
             JMP       HexDec100

HexDec300:
             LDR       OP1_1                   ; Add the latest value at the decimal value.
             STA       Res1
             STI       OP1_1, 00H

             RET
```

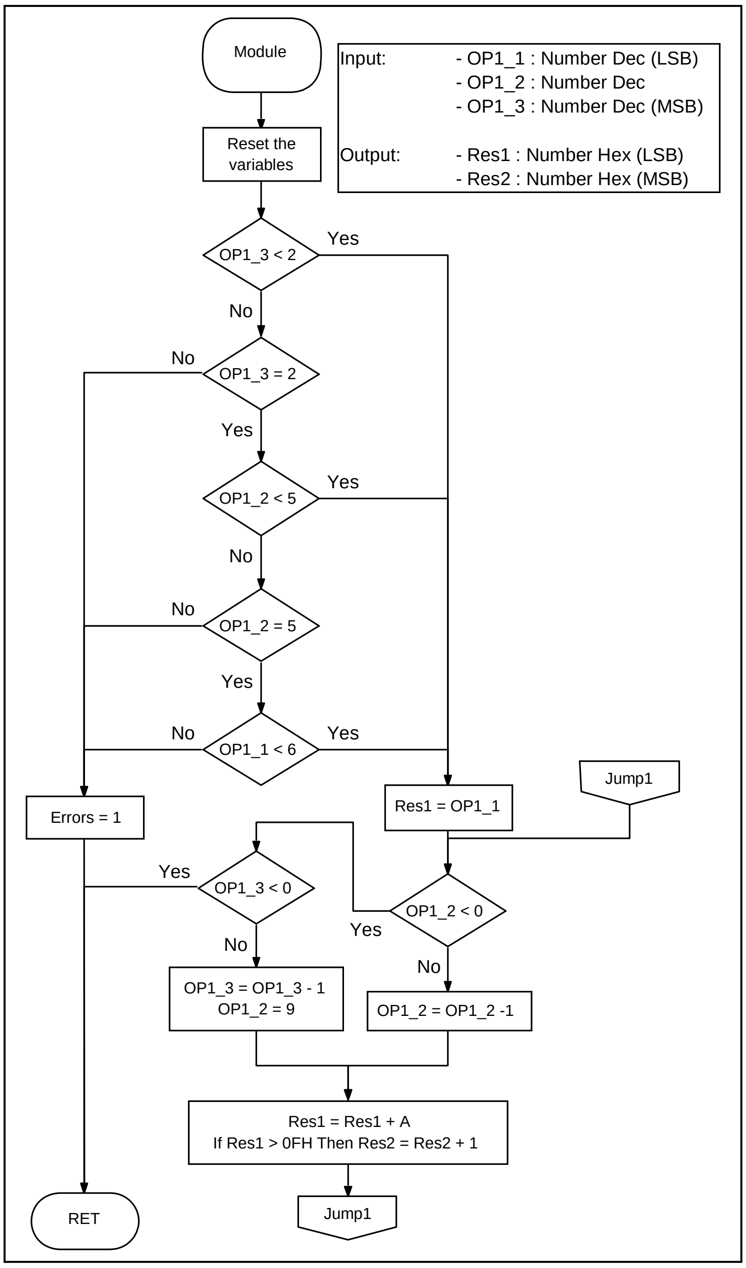
Application Note 21

17. How to convert Dec - Hex

The 4 bits uC does not provide instructions to convert a value dec - hex. Therefore it is necessary to implement the *converter dec - hex* functions by a small piece of code, which could be written as a subroutine as shown below.

This flowchart describes the module to convert Dec - Hex.

If you include this module in your software, you define the five variables in your principal file.



Appendix A: Dec – Hex module

```
;----------------------------------------------------------------------------------------------------
;           MODULE :            DECHEX.ASM
;           Date last mod       :            20/02/1998                   Ch.Mayer
;           Module for convert dec - hex
;----------------------------------------------------------------------------------------------------
;           OP1_1     :         First number (LSB)
;           OP1_2     :         First number
;           OP1_3     :         First number (MSB)
;           Res1      :         Resultat (LSB)
;           Res2      :         Resultat (MSB)
;------------------------------------------------------------------------------------------
DecHex:
            LDI       02H                 ; Test if the decimal value is => 300.
            SUB       OP1_3
            JPNC      DecHex090           ; Value is < 200, then jump to "DecHex090".
            JPZ       DecHex010           ; Value is between 200 and 299, then jump to "DecHex010".
            STI       Errors, 01H         ; Value too big, then error.
            JMP       DecHex400
DecHex010:
            LDI       05H                 ; Test if the decimal value is => 260.
            SUB       OP1_2
            JPNC      DecHex090           ; Value is < 250, then jump to "DecHex090".
            JPZ       DecHex020           ; Value is between 250 and 259, then jump to "DecHex020".
            STI       Errors, 01H         ; Value too big, then error.
            JMP       DecHex400
DecHex020:
            LDI       06H                 ; Test if the decimal value is => 256.
            SUB       OP1_1
            JPNC      DecHex090           ; Value is < 256, then jump to "DecHex090"
            STI       Errors, 01H         ; Value too big, then error.
            JMP       DecHex400
DecHex090:
            LDR       OP1_1               ; Decrement the decimal value for ...
            STA       Res1
            STI       OP1_1, 00H
            STI       Res2, 00H
DecHex100:
            DEC       OP1_2
            JPNC      DecHex200
            STA       OP1_2
            JMP       DecHex300
DecHex200:
            DEC       OP1_3
            JPNC      DecHex400
            STA       OP1_3
            STI       OP1_2, 09H
DecHex300:
            LDI       0AH                 ; ... increment the hexa value.
            ADD       Res1
            STA       Res1
            JPNC      DecHex100
            INC       Res2
            STA       Res2
            JMP       DecHex100
DecHex400:
            RET
```

Application Note 22

18. Protection of internal E^{2}PROM

This AppNote applies only to products EM6640, EM6540, EM6617 and EM6517. When working with a micro-controller incorporating an E^{2}PROM, the memory is normally used to store numbers, codes and other important data.

It is essential to avoid this data being altered inadvertently by extraneous activity, which could lead to a modification of the program and undesired consequences.

A set procedure for writing data to the EEPROM is essential; external disturbances can cause “glitches” and can lead to inadvertent modification of the data stored in the memory during the write process.
The environment of the micro-controller can affect the extent of the risk resulting from external disturbances (such as faulty transformers, magnetic fields etc).
As a consequence, the procedures used to write the memory carry a potential risk of corrupting the E^{2}prom data.

To avoid this problem, and subject to the type of software being written, five proposals are set out below.
Prior consideration should be given to the conditions under which any programming is to take place: for example, will this be done only by the manufacturer or also during the service life of the micro-controller?

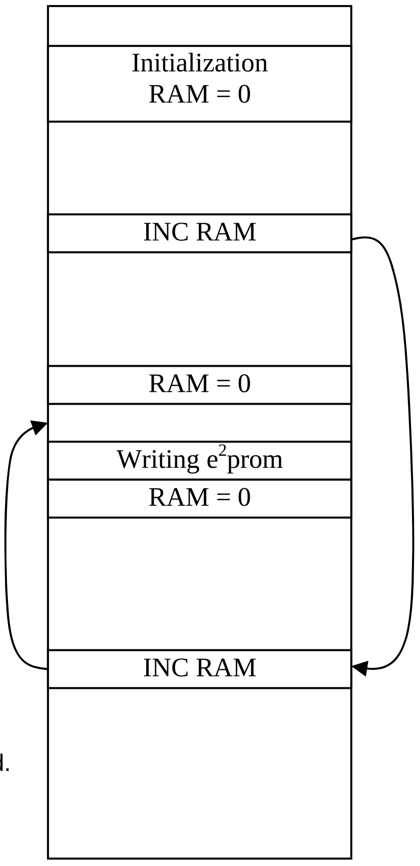


Figure 1.

1) If the E^{2}prom is to be programmed only by the manufacturer, a spare pad can be used. Set this pad to Vdd during the writing phase and to Vss to lock any writing. During the writing phase, the pad must be checked by software to determine whether an instruction given was valid or not.

2) Use the SVLD to ensure that the potential at the terminals of the micro-controller is sufficient to permit writing. This is essential to avoid possible failure of the power supply during writing.

3) Before running the write routine of the E^{2}prom, increment a RAM address at two specific, well-defined locations in the software program. This will ensure that if the software performs normally, this address will contain the value “02H”. During the write phase, this variable should be read and then written to the register. Providing the program is not disturbed, the value written to the register will be correct and the programming can start. When the write process is complete, ensure that the variable at 00H is immediately erased.

4) Data to be written to the E^{2}prom should be temporarily held in the RAM (see fig 2); at this point, a checksum is calculated and also held in the RAM. This checksum is recalculated and compared with the first checksum immediately before writing each 4-bit word to the E^{2}prom. If the two checksums correspond then there has been no corruption of the data by extraneous interference. When all words have been written, the RAM is erased and the checksum is also written to the E^{2}prom. When the recorded data is read from the memory, the checksum is again recalculated and compared to the recorded checksum. If they match it signifies that there were no problems during writing.

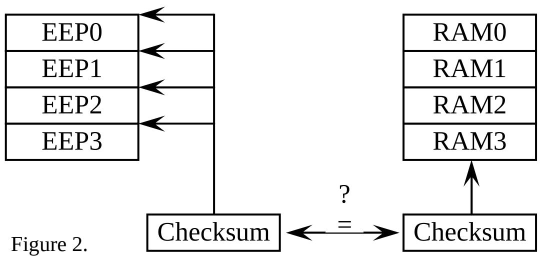


Figure 2.

5) To guarantee a higher level of security, the procedures given in Proposal 4 can be enhanced as follows: once all 4-bits words have been written to the E^{2}prom they can be read back and compared to the data in the RAM. This must be carried out before erasure of the RAM. This additional procedure will verify that the data memorized was not corrupted during the write phase by, for example, a brief power supply dropout.

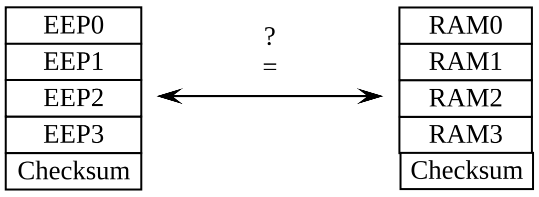


Figure 3.

EM Microelectronic-Marin SA (EM) makes no warranty for the use of its products, other than those expressly contained in the Company's standard warranty which is detailed in EM's General Terms of Sale located on the Company's web site. EM assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of EM are granted in connection with the sale of EM products, expressly or by implications. EM's products are not authorized for use as components in life support devices or systems.

© EM Microelectronic-Marin SA, 06/05, Rev. C