Maxim > Design Support > Technical Documents > Application Notes > Microcontrollers > APP 958

Keywords: RAM, Internal memory, MCU, Microcontroller, microprocessor, uC, binary arithmetic, binary math, booth's algorithm, multiply, divide, ASCII

APPLICATION NOTE 958

A Collection of Extended Math Subroutines for the MAX7651

Jul 17, 2002

Abstract: This article gives assembly code examples for reserving internal memory, simple ASCII conversion, 32-bit subtraction, 16x16 multiple and 32-bit divide using 8051-compatible microcontrollers such as the MAX7651 and MAX7652.

The MAX7651 flash-programmable 12-bit integrated data acquisition system uses an 8-bit CPU core for all operations. There are cases where 8-bits are not sufficient resolution for data manipulation. An obvious example is when using the internal ADC, which has 12-bit resolution. Collecting several readings and then finding the maximum value requires math subroutines beyond the 8-bits in the CPU registers.

The solution is to use internal RAM registers in a group, and use the MAX7651's CPU to perform the math in 8-bit 'chunks'. Successive operations are performed until the desired result is obtained.

This application note presents several commonly used math subroutines that operate on data larger than 8-bits and is divided into four sections:

- A subroutine for reserving internal RAM to hold variables
- A simple 8-bit ASCII character conversion subroutine which includes leading zero blanking
- Extended ASCII character conversion, which includes subroutines for 32-bit subtraction, 16x16 bit multiplication and 32-bit division
- An example illustrating use of the aforementioned subroutines

Reserving Internal Memory

The following code tells the assembler to reserve internal memory to hold the variables used by the math subroutines. These memory locations can be anywhere in the memory map.

```
;
; Reserve internal RAM for use with the math subroutines
;
; A good starting memory location is at 30H, but the starting location
; can be anywhere in the memory map.
;

DIGIT2:   DS    1    ; 100's digit for ASCII routines
DIGIT1:   DS    1    ; 10's digit
```

```
DIGIT0:  DS   1    ; 1's digit

DATAHI:  DS   1    ; Upper byte of 16-bit register

DATALO:  DS   1    ; Lower byte of 16-bit register

REMAIN:  DS   1    ; Remainder

OP3:     DS   1    ; OP3-OP0 are 4 8-bit registers. For 32-bit math
OP2:     DS   1
OP1:     DS   1
OP0:     DS   1    ; Least-significant byte of 32-bit 'operator'

TEMP3:   DS   1    ; TEMP3-TEMP0 comprise the 32-bit temp register
TEMP2:   DS   1
TEMP1:   DS   1
TEMP0:   DS   1    ; Least-significant byte of temp register
```

Simple ASCII Conversion

In many MAX7651 applications, there is a requirement to use ASCII data for display purposes. The display type may be a LCD, LED, vacuum fluorescent displays or other technology. The most commonly used displays are one or two-line LCD modules. These accept ASCII characters, so the software program must convert binary data into separate ASCII digits. ASCII (an acronym for American Standard Code for Information Interchange) is a seven digit binary code used to represent letters, numbers and symbols.

For example, let's assume you have data in a register that is a positive, 8-bit value from 00H to 0FFH. This corresponds to the binary numerical values 0 to 255. If you want to have the LCD show '127' on the screen, you need to send it three ASCII characters; one for each digit: the '100's digit[1], the '10's digit [2] and the '1's digit [7].

Fortunately, the binary to ASCII conversion is straightforward. An ASCII numerical digit is simply the binary number added to 30H. To generate the three digits, the following subroutine successively divides the original binary data by 100, then subtracts this number from the original number (127/100 = 1 with a remainder of 27). It then takes the remainder and divides by 10 and retains the remainder (27/10 = 2 with a remainder of 7). Each value is then added to 30H to obtain the ASCII values, which are then stored.

In this subroutine, the 8-bit binary number to be converted is passed in the accumulator (register A). Since the MAX7651 uses the accumulator for all of its math functions, the internal register R0 is used to hold intermediate results. If your application needs to retain the value in R0, you simply use another register.

The subroutine uses the MAX7651's multiply instruction (MUL AB) to generate the '100's and '10's digits to be subtracted out, and the ADD instruction to form the final ASCII values. The subroutine also performs 'leading zero blanking', so that if the number is 99 or less, the software will suppress any leading zeros and replace them with a blank space.

```
;
; Subroutine 2_ASCII
;
; Converts the 8-bit ACC into an ASCII digit
;
; ACC and RO are destroyed, previous value in DIGIT2-0 overwritten
;

 2ASCII: MOV    RO,A
         MOV    B,#100        ; Get 100's digit
         MOV    A,R0
         DIV    AB            ; A has quotient, B has remainder
         MOV    DIGIT2,A      ; Save 100's digit
         MOV    B,#100
         MUL    AB            ; Need to subtract out 100's digit
         XCH    A,R0
         CLR    C
         SUBB   A,RO
         MOV    R0,A
         MOV    B,#10         ; Get 10's digit
         DIV    AB
         MOV    DIGIT1,A
         MOV    DIGIT0,B      ; Remainder is the 1's digit
 ;
 ; Now convert to ASCII
 ;
         MOV    A,DIGIT0      ; 1's digit
         ADD    A,#'0'        ; Offset from 30H
         MOV    DIGIT0,A      ; Write it back to memory
         MOV    A,DIGIT1      ; 10's digit
         ADD    A,#'0'        ; Offset from 30H
         MOV    DIGIT1,A      ; Write it back
         MOV    A,DIGIT2      ; 100's digit
         CJNE   A,#0,NOTZ     ; A non-zero value
         MOV    DIGIT2,#' '   ; Blank it
 ;
 ; Blank again?
 ;
         MOV    A,DIGIT1
         CJNE   A,#'0',SKIPBL ; Non-zero abort
```

```
        MOV     DIGIT1,#' '
SKIPBL: RET
NOTZ:   ADD     A,#'0'          ; Offset from 30H
        MOV     DIGIT2,A        ; Write it back
        RET
```

Extended ASCII Conversion

32-Bit Subtraction

The previous subroutine is only useful if the number to be converted is 255 or less. What if the application is measuring temperature in a chemical process, and we want to display temperatures up to 999 degrees? This requires the use of a set of extended math subroutines that divide the data into multiple 8-bit registers.

From the above example, the algorithm is to multiply by the 'digit place' (i.e., 100's, 10's), then subtract out that digit from the original number. Therefore, we need to write an extended subtraction subroutine and an extended multiply subroutine.

The subtraction subroutine is easy to do with the instruction SUBB, which automatically uses the borrow flag. It may seem strange at first glance, because the subroutine does not subtract in 'digits' as we are taught, but in blocks of 255 at a time (the full range of the accumulator). However, it does provide the correct answer.

The subroutine as written subtracts a 32-bit number (TEMP3:TEMP2:TEMP1:TEMP0) from another 32-bit number (OP3:OP2:OP1:OP0) *and places the result back into OP*. The accumulator is used to successively subtract the 8-bit 'chunks' from the original number.

```
;
; Subroutine SUB_32
;
; OP < OP - TEMP
;
; This routine overwrites the ACC and the carry flag (here used as a borrow flag)
; Note that the 2 numbers DO NOT have to be 32-bits
;
;

SUB_32: CLR     C
        MOV     A,OP0
        SUBB    A,TEMP0
        MOV     OP0,A

        MOV     A,OP1
        SUBB    A,TEMP1
        MOV     OP1,A

        MOV     A,OP2
        SUBB    A,TEMP2
```

```
        MOV     OP2,A

        MOV     A,OP3
        SUBB    A,TEMP3
        MOV     OP3,A
        RET
```

16x16 Multiply

The next two subroutines are much more complicated. The first routine is a 16x16 multiply, with a 32-bit result. The routine assumes both numbers are positive (0000H to 0FFFFH). The result is placed into OP3:0.

The subroutine first generates the first 8-bit "digit" using the internal MUL AB instruction. But after that, the routine must perform four separate operations for each "digit": two sets of a multiply/add instruction. This is because we are using binary arithmetic, not decimal arithmetic.

```
;
; Subroutine MUL_16
;
; Multiplies 16-bit number DATAHI:DATALO by 16-bit number OP3:0 and places the result back into
OP3:0
; Uses the 32-bit TEMP3:0 registers as well
;
;

MUL_16:   MOV     TEMP3,#0
          MOV     TEMP2,#0    ; Clear upper 16-bits
;
; Generate lower byte of result
;
          MOV     B,OP0
          MOV     A,DATALO
          MUL     AB
          MOV     TEMP0,A
          MOV     TEMP1,B     ; 1st result
;
; Byte 2 of result
;
          MOV     B,OP1
          MOV     A,DATALO
          MUL     AB
          ADD     A,TEMP1     ; Lower nibble result
          MOV     TEMP1,A
          MOV     A,B
```

```
          ADCC   A,TEMP2
          MOV    TEMP2,A
          JNC    MULOOP1
          INC    TEMP3       ; propogate carry
MULOOP1:  MOV    B,OP0
          MOV    A,DATAHI
          MUL    AB
          ADD    A,TEMP1
          MOV    TEMP1,A
          MOV    A,B
          ADDC   A,TEMP2
          MOV    TEMP2,A
          JNC    MULOOP2
          INC    TEMP3       ; byte 2 is done
;
; Byte 3
;
MULOOP2:  MOV    B,OP2
          MOV    A,DATALO
          MUL    AB
          ADD    A,TEMP2
          MOV    TEMP2,A
          MOV    A,B
          ADDC   A,TEMP3
          MOV    TEMP3,A
;
; Next nibble
;
          MOV    B,OP1
          MOV    A,DATAHI
          MUL    AB
          ADD    A,TEMP2
          MOV    TEMP2,A
          MOV    A,B
          ADDC   A,TEMP3
          MOV    TEMP3,A
;
; Byte 4
;
          MOV    B,OP3
```

```
        MOV     A,DATALO
        MUL     AB
        ADD     A,TEMP3
        MOV     TEMP3,A
        MOV     B,OP2
        MOV     A,DATAHI
        MUL     AB
        ADD     A,TEMP3
;
; Save results
;
        MOV     OP3,A
        MOV     OP2,TEMP2
        MOV     OP1,TEMP1
        MOV     OP0,TEMP0
        RET
```

32-Bit Divide

Now that we can multiply two 16-bit numbers, we can also use this algorithm 'backwards' to divide. However, it requires four intermediate registers (R7, R6, R1, R0) to hold partial quotients. Since we are using binary arithmetic, we can divide by 2 with a simple shift right command. This can be extended by clever "shift and subtraction" to divide by 10's digits. This is called "Booth's Algorithm". The loop is run 32 times (once for each bit-position, which in turn is a power of 2).

```
;
; Subroutine DIV_16
;
; Divides OP3:2:1:0 by DATAHI:DATALO and places results in OP3:0
;
;

DIV_16:     MOV     R7,#0
            MOV     R6,#0           ; Zero partial remainder
            MOV     TEMP0,#0
            MOV     TEMP1,#0
            MOV     TEMP2,#0
            MOV     TEMP3,#0
            MOV     R1,DATAHI       ; Load the divisor
            MOV     R0,DATALO       ; Bit counter
            MOV     R5,#32          ; Shift dividend and msb>carry
DIV_LOOP:   CALL    SHIFT_D
            MOV     A,R6
            RLC     A
```

```
                MOV     R6,A
                MOV     A,R7
                RLC     A
                MOV     R7,A
;
; Now test to see if R7:R6 =>R1:R0
;
                CLR     C
                MOV     A,R7
                SUBB    A,R1            ; see if R7 < R1
                JC      CANT_SUB        ; yes
;
; At this point R7>R1 or R7=R1
;
                JNZ     CAN_SUB         ; R7 is > R1
;
; If R7=R1, test for R6=>R0
;
                CLR     C
                MOV     A,R6
                SUBB    A,R0            ; Carry set if R6 < R0
                JC      CANT_SUB
CAN_SUB:        CLR     C
;
; Subtract divisor from partial remainder
;
                MOV     A,R6
                SUBB    A,R0
                MOV     R6,A
                MOV     A,R7
                SUBB    A,R1            ; A=R7 - R1 - borrow bit
                MOV     R7,A
                SETB    C               ; Shift 1 into quotient
                SJMP    QUOT
CANT_SUB:       CLR     C               ; Shift 0 into quotient
QUOT:           CALL    SHIFT_Q         ; Shift carry into quotient
                DJNZ    R5,DIV_LOOP     ; Did it 32 times?
;
; All done!
;
```

```
            MOV     OP0,TEMP0
            MOV     OP1,TEMP1
            MOV     OP2,TEMP2
            MOV     OP3,TEMP3
DIV_DONE:   RET
;
; Shift the dividend one bit to the left and return msb in carry bit
;
SHIFT_D:    CLR     C
            MOV     A,OP0
            RLC     A
            MOV     OP0,A
            MOV     A,OP1
            RLC     A
            MOV     OP1,A
            MOV     A,OP2
            RLC     A
            MOV     OP2,A
            MOV     A,OP3
            RLC     A
            MOV     OP3,A
            RET
;
; Shift the quotient one bit to the left and shift carry bit into lsb
;
SHIFT_Q:    MOV     A,TEMP0
            RLC     A
            MOV     TEMP0,A
            MOV     A,TEMP1
            RLC     A
            MOV     TEMP1,A
            MOV     A,TEMP2
            RLC     A
            MOV     TEMP2,A
            MOV     A,TEMP3
            RLC     A
            MOV     TEMP3,A
            RET
```

Putting It All Together

Now we have all the subroutines needed for the extended ASCII conversion. The last routine converts a number in the range 0 to 999 (stored in DATAHI:DATALO) into 3 ASCII digits. The algorithm is the same as for the earlier, simple conversion routine, except now we use the three extended math routines to operate on the 16-bit registers.

```
;
; Subroutine CONVERT3
;
; Converts a 16-bit value 000-999 in DATAHI:DATALO to ASCII
; Data stored into DIGIT2 - DIGIT0
;


CONVERT3: MOV   OP0,DATALO
          MOV   OP1,DATAHI
          MOV   OP2,#00
          MOV   OP3,#00
          MOV   TEMP8,DATALO
          MOV   TEMP9,DATAHI  ; Save original for remainder
          MOV   DATALO,#100
          MOV   DATAHI,#00
          CALL  DIV_16        ; Divide number by 100
          MOV   A,OP0         ; Answer is 2-9 + remainder
          ADD   A,#30H        ; Convert to ASCII
          MOV   DIGIT2,A      ; Save it
          MOV   DATALO,#100   ; Convert the remainder
          MOV   DATAHI,#0
          CALL  MUL_16
          MOV   TEMP0,OP0
          MOV   TEMP1,OP1
          MOV   TEMP2,OP2
          MOV   TEMP3,OP3
          MOV   OP0,TEMP8
          MOV   OP1,TEMP9
          CALL  SUB_32        ; Subtract 100's digit
          MOV   A,OP0
          MOV   B,#10         ; 10's digit calculation
          DIV   AB
          ADD   A,#30H
          MOV   DIGIT1,A      ; Get the 10's digit
          MOV   A,B
          ADD   A,#30H
          MOV   DIGIT0,A      ; Get the 1's digit
```

```
;
; Check for zero blanking
;
            MOV     A,DIGIT2
            CJNE    A,#'0',BK_DONE
;
; Blank 100's digit
;
            MOV     DIGIT2,#' '
;
; Now check 10's digit
;
            MOV     A,DIGIT1
            CJNE    A,#'0',BK_DONE
;
; Blank 10's digit
;
            MOV     DIGIT1,#' '
BK_DONE:    RET
```

Conclusion

These routines expand the math capabilities of the MAX7651 to 16-bits. You can modify these subroutines to handle 32-bit data as well. The MAX7651's four-clock cycle CPU greatly speeds up these routines of standard 8051 processors.

Further Reading

"*The Art of Computer Programming*" by Donald Knuth contains detailed explanations of these algorithms (not specific to any processor, but in general terms). This is a 3-volume set that is considered a classic in numerical algorithms.

Related Parts	
MAX7651	Flash Programmable 12-Bit Integrated Data-Acquisition Systems
MAX7651EVKIT	Evaluation Kit for the MAX7651
MAX7652	Flash Programmable 12-Bit Integrated Data-Acquisition Systems

More Information

For Technical Support: http://www.maximintegrated.com/support

For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

---

Application Note 958: http://www.maximintegrated.com/an958
APPLICATION NOTE 958, AN958, AN 958, APP958, Appnote958, Appnote 958