****

Proj_1E_Timer_interrupt

A PROGRAM THAT DOES MULTI-TASKING

The program continues to drive the display but the vast majority of time which was spent waiting to increment the display is used to do something useful. In this case doing some arithmetic and sending the results to the PC to be displayed.

IT INTRODUCES

1. The char variable: A char variable is stored in a location in data memory that holds only 8 bits of data rather than 16 used for integer variables. A 'char' variable can hold 2^8 = 256 different values i.e. numbers from 0 to 255. A 'signed char' can hold numbers from -128 to +127 (unsigned is the default option for chars).

2. Project subroutine T1_65ms_clock_tick() This generates an interrupt every 65 mS.

3. ISR(TIMER1_OVF_vect){} This is the Timer1 interrupt service routine. Here, it is the ISR that increments the LED display rather than the main routine.

4. Code other than a timer subroutine that can safely be interrupted. Here the code is executed indefinitely by a while(1) loop.

5. Project subroutines Char_to_PC() and Num_to_PC_U() These send data to the PC and will be considered in more detail later on.

6. Timer T2 This slows down data flow to the PC.

7. long numbers: These occupy 32 bits of data memory and can hold numbers between -2^31 and 2^31 – 1 or between 0 and 2^32 - 1 = 4,294,967, 295.

8. Volatile variables: The theory: Every variable used by the main routine is allocated permanent storage space in data memory. Variables only used by the subroutines or ISRs are allocated shared memory space in data memory. However there may be some variables that are not used by the main routine but that must not be allowed to loose their value or be overwritten by other variables. They also require permanent storage and are defined as volatile. In Practice: A bit of trial and error is often required to determine which variables realy need to be defined as volatile.

THE ISR INTRODUCES

1. Binary numbers: Numbers are saved in data memory as ones and zeros. For example 1 is saved as 1, 2 as 10, 4 as 100, 8 as 1000, 16 as 10000. These will be considered in more detail later.

2. Hex numbers: These will also be considered in more detail later. Note however that the hex number 0x8000 is 1000 0000 0000 0000 in binary.

3. Some logic: i.e. the 'or' and 'and' functions and the shift '<<' or '>>' operators.

4. The '|' (or) function. For example 10001010 | 10101000 = 10101010
The short hand notation 'a |= b' which means that 'a' is set equal to 'a | b'.

5. The 'if-else' statements: Consider the statements
if (m<=5){PORT_1 |= (PORT_1 << 1); m += 1;} else PORT_1 = PORT_1 << 1;
The statements '{PORT_1 |= (PORT_1 << 1); m += 1;}' are only executed if 'm' is less than 6
Otherwise the statement 'PORT_1 = PORT_1 << 1' is executed.

6. The '&' (and) function 11001010 & 10101000 = 10001000 Consider the statement

if (PORT_1 & 0x8000) overflow=1;
It is used here to test the most significant bit (MSB) of PORT_1.

Note: the 'for-loop' is omitted, instead the value of 'PORT_1' is tested every time that the 'while-loop' is executed. After 15 repetitions the least significant bit of Port_1 is reset to 1.

A QUICK LOOK AT SOME OF THE LOGIC

Note: a|b and a&b operate on the bits defined by -a- and -b-
00001010 | 10101000 = 10101010 and 11101010 & 10100011 = 10100010

If PORT_1 starts of as 00000111 then the statement PORT_1 = PORT_1 << 1 changes it to 00001110
and the statement PORT_1 |= (PORT_1 << 1) changes it to 00000111 | 00001110 which equals 00001111

if PORT_1 = 1xxxxxxxxxxxxxxx then "overflow" is set to save the LH 1 which will otherwise be lost at the next shift left. It is then placed in the most RH location. Note x is either 0 or 1; LH is left hand, RH is right hand.