Matrix Knowledge Exchange

Introduction to Interrupts

by Sean King, May 2009

Abstract

One feature of microcontrollers that is often overlooked is interrupt. Interrupts are signals that act directly on the hardware of the CPU within the microcontroller. This allows the CPU to respond to an interrupt immediately. In this article Sean shows us how Flowcode makes the task of enabling and using interrupts; intuitive and straightforward.

Requirements

Software:

- Any licence type of Flowcode v3 or v4 for any variant.

Hardware:

- No Hardware Requirements

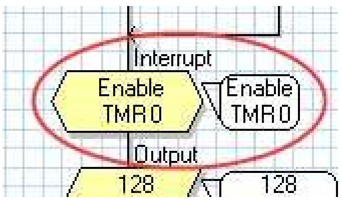
Introduction

A very useful, but often avoided, feature available on almost all microcontrollers is the interrupt.

Interrupts are signals that act directly on the hardware of the CPU within the microcontroller. This allows the CPU to respond to an interrupt immediately. Without interrupts, the main program would need to continually check for a change in the signal(s) - referred to as 'Polling' - which could easily allow these signal changes to be missed.

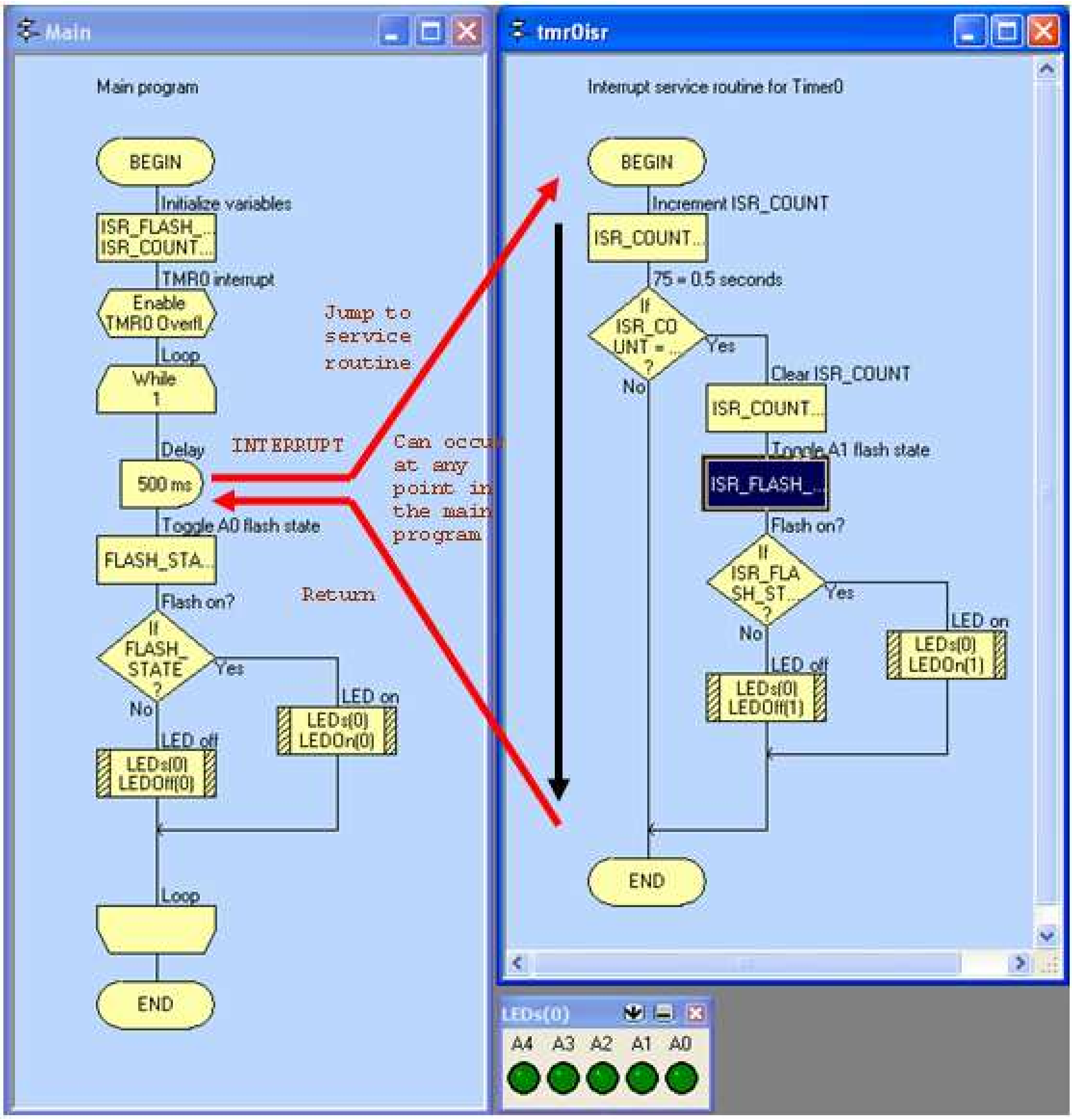
The CPU response to an interrupt is to:

- Stop execution of the main (background) program.
- Store the address of the next instruction of the main program to be executed.
- Store other system information to allow the current state to be restored.
- Jump, via the interrupt handling system of the CPU, to an Interrupt Service Routine (ISR).
- Execute the ISR code.
- Restore the stored system state.
- Jump back to the stored location in the main program.



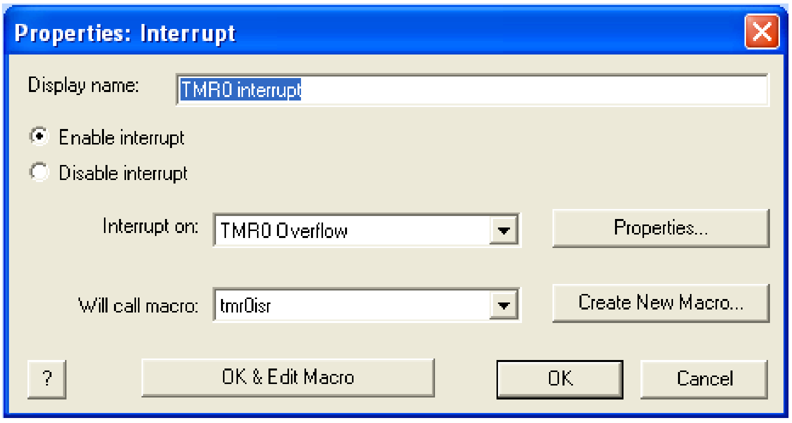
When programming in assembler all these details must be considered very carefully. Higher level languages make the task easier, and Flowcode removes almost all the complexity.

Interrupt responses occur at the end of individual CPU instructions and will probably cause the main program to jump from, and return to, the middle of a Flowcode block.



Interrupts are usually disabled on power-up and must be enabled by the main program

This Flowcode interrupt property panel allows an interrupt source to be selected, the interrupt enabled, and the jump destination (Flowcode macro) to be defined in a single operation.


Properties: Interrupt

Display name: TMR0 interrupt

[✓ Enable interrupt]
Disable interrupt

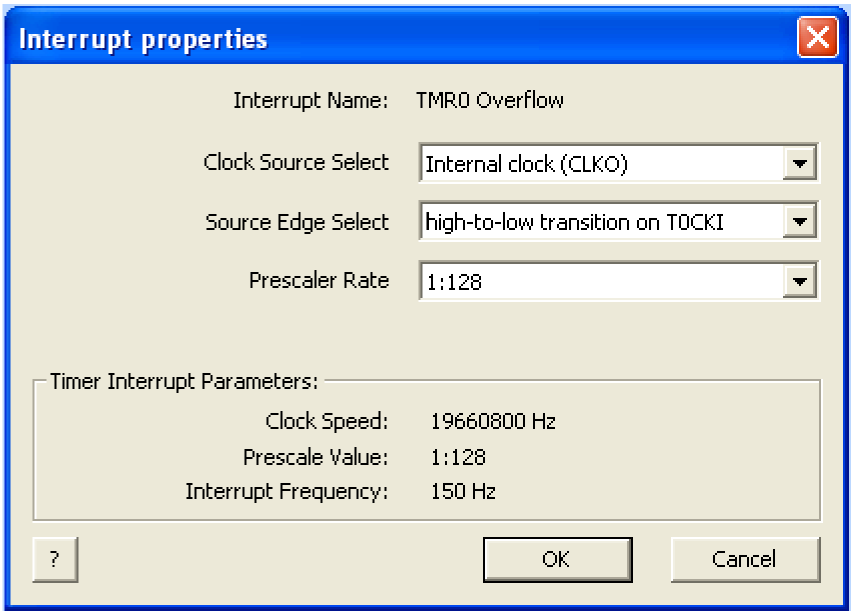
Interrupt on: TMR0 Overflow

Will call macro: tmr0isr

Notice that the interrupt service macro is never called from the main program. It is executed independently in response to the TMR0 interrupt.

The interrupt service routine can use and modify variables that are use by the main program and other macros.

Some of the peripheral devices capable of generating interrupts require additional configuration information to allow the interrupts to be generated under specific conditions. Flowcode also provides easy access to the range of configuration options offered by the target device



Interrupt Name: TMR0 Overflow

Clock Source Select: Internal clock (CLKO)

Source Edge Select: high-to-low transition on T0CKI

Prescaler Rate: 1:128

Timer Interrupt Parameters:

Clock Speed: 19660800 Hz

Prescale Value: 1:128

Interrupt Frequency: 150 Hz

Interrupts common to most microcontrollers are:

- External edge or level triggered (specific individual I/O pins)
- External port change (specific combinations of I/O pins)
- Timer overflow (counter/timer peripheral has exceed its count limit and overflowed back to zero)
- UART receive (data received by UART peripheral - similar for I2C, SPI, etc.)
- UART transmit (UART peripheral transmit buffer empty - similar for I2C, SPI, etc.)
- ADC sample complete (the ADC peripheral has completed a conversion)

Example programs

The two example programs linked to this article illustrate the advantage of using a timer interrupt to flash an LED at a constant and accurate rate, independent of the operation of the main program.

The first program **Interrupt1.fcf** compares the flash rate of two LEDs, one controlled from the main program (A0) in a delay timed loop, the other from a timer interrupt (A1).
The two LEDs flash at approximately the same rate. After 20-30 flashes a small difference can be detected.

The second program **Interrupt2.fcf** is identical to the first, but in this case additional LCD print commands have been introduced into the main program. The effect on the flash rates can be seen almost immediately. The interrupt driven LED is still accurate, but the program controlled LED is completely out of phase within 20 flashes.
Also, the second program is required to detect a button press after an interval of 25 seconds. The easiest way to do this is to suspend the program loop until the button is pressed. However, this also stops the A0 LED from flashing. The program could be rearranged to allow the A0 LED to continue flashing while waiting for the button press, but at the cost of significantly increased complexity.
If the main program contained more loops, branches, delays, and macro calls, the task of maintaining a constant flash rate on the A0 LED would become almost impossible.

Although these example programs demonstrate the trivial process of flashing LEDs, the same principals can be applied to much more significant tasks - like reading and writing streamed data, or generating and detecting precisely timed events.

Interrupt service routine tips:
It is essential that the interrupt service routines are kept as short as possible. Most only require a small amount of data to be transferred, or a simple calculation to be carried out.

Complex or potentially time consuming operations should be carried out in the main program. These include:
- Calling most other component or user macros (LCD, ADC, Communications, etc.), except for basic I/O operations
- While loops - unless the escape condition can be guaranteed within an acceptable period.
- Connection points - unless the flow of the program is understood for all possible conditions.
- Delays (other than very quick microsecond delays)

The fact that interrupt responses can occur at the end of any CPU instruction means that more complex operations can be split by an interrupt. Multi-byte data transfers (int, string, or float) can be interrupted in mid-execution, and, if the interrupt service routine shares these variables it is possible for the values to be corrupted when read or written. There are techniques to avoid these problems, but the simplest approach is to only share byte variables - as these are unaffected.

Further reading

Below are some links to other resources and articles on related subjects, and technical documentation relating to the hardware used for this project...

Flowcode:	http://www.matrixmultimedia.com/flowcode.php
Eblocks:	http://www.matrixmultimedia.com/eblocks.php
Learning Centre:	http://www.matrixmultimedia.com/lc_index.php
User Forums:	http://www.matrixmultimedia.com/mmforums
Product Support:	http://www.matrixmultimedia.com/sup_menu.php