Using Interrupts

The difference between "wired" and "weird" is only minimal

A cooking-recipe has many similarities with a program: It tells what to do with which resources in which order. Instructions like "Whip the egg-white until stiff" does not differ much from usual pseudo-code. Now imagine you are stirring a cake dough and suddenly the door-bell rings. You put aside your equipment and open the door. The postman delivers a parcel which you sign for, close the door leaving the parcel on the mantelpiece and continue with your cake. When now you have got a balll-pen in your hand instead of your wooden spoon something with your interrupt has gone wrong.

One major task an embedded designer has to care for are the interrupts, sometimes called "Exceptions". A designer not only cares for interrupts, he will use and handle them because they make project realization easier or even possible. Interrupts -depending on the IDE you use- are often encapsulated in functions or objects helping to maintain their properties.

An interrupt halts the actual program flow and executes an associated piece of code named "interrupt handler". When the handler finishes the original environment of the interrupted program is restored and the execution resumes at the point where the program has been halted. While the handler executes, only interrupts with a higher priority will be served from now on (if any) until the handler finishes.

The interrupt in an embedded micro is normally generated by a programmable pice of hardware named "Interrupt Controller Unit" (ICU) which usually allows for

- Setting an interrupt priority[1]
- Masking interrupts

1 Trough historical reasons are lower priority numbers used to define higher priorities

- Defining the signal edge or a level generating the interrupt
- Firing an interrupt by software command

Some ICUs require to clear the interrupt explicitly, some ICUs do that automatically and some clearing can be done within the encapsulated interrupt-code, get acquainted with your embedded handbook. But always the source of the interrupt has to be cleared before the handler finishes, otherwise when the handler finishes and interrupts are allowed again the non-cleared source would instantly fire the same interrupt again and again.

Since interrupts may fire at any time within the normal program flow without caring for C-statements or lines of code some pitfalls are associated with them, I will later explain how to handle those.

Keep handlers short! That minimizes jitter in the timing and helps to reduce side effects due to a loss of responsiveness while the handler is executed.
Avoid loops in handlers. An absolute no-go is to wait for an event or to use delays in a handler. Check especially system-functions you call from within a handler for delay-loops, as for instance writing to LCDs or peripherals may induce them. A very common practice is just to set a flag within the handler and exit. In the main-loop the flag is tested, acted upon the result and reset back again. This of course only works when the expected frequency of the interrupts is less than the frequency of the main-loop.

Using interrupts can come in very handy. A common use is to catch the incoming data of an interface, store them into a "Circular Buffer" (Pg. 23) and retrieve them when there is time to do so. While all other tasks the embedded performs are running (nearly) uninterrupted the characters get collected in the background. What additionally is needed is an indicator that a complete message has arived.

Sending data over an interface is best done with interrupts, too. A common pitfall when programming this function is to forget to copy the data to be sent into a memory area since the original data may disappear when have been stored in a local variable (Pg. 32) of the calling program.

Reading sensors and averaging the readings in the background will increase the program performance. Since always there is a value ready to work with there is no need for waiting for a conversion ready.