MSP430 MasterLibrary

# Introduction

I don’t have a background in electronics or microcontroller, but, as a programmer, I feel I can give my two cents contribution to the community with this general library.

I’m pretty sure that programming engineers, especially with a strong electronics background, can go crazy writing (and reading) this line of code:

WDTCTL = WDTPW | WDTHOLD;

Bare to the metal, uh? ☺

The problem is: I normally write code in C# AND C++, using STL library, interfacing with SQL Server, and on and on…

If, and I really mean ‘if’, I would write a line of code like that, what could I understand, tomorrow, reading it back? Maybe that I was in trance, the day before!

Perhaps this form:

InitWatchDogTimer(*STOPPED*);

is a little more readable.

What we cannot forget is that optimizations are important, and even if I’m surely not an expert in this field, I tried to write code that results in a compact and efficient form: I wait for critics and suggestion… ☺

So, armed with patience, and the will to create a library that makes my (programmers’) life a little bit better, follow me in the steps I walked toward my result.

# Code size

Since many variants of the MSP430 (especially value line) have very little flash memory, I tried to keep constantly an eye on the resulting executable size. The ‘void’ program, i.e. this:

**void** **main**(**void**)

{

**while** (1)

;

}

gives us a (minimum) executable of 146+2 bytes. I don’t know either the internal details of the processor architecture, or of the CCS compiler, so I take this as a base, without further thoughts.

# MACROS

The most obvious way to hide complexities and strange names, but at the same time not losing in performace, is using macros.

The first macro I wrote was:

**#define** SETLOW(REG, PIN) REG &= ~(PIN)

I don’t want to lose my time watching code closely (I already wear glasses), so I want to eliminate all the strange symbols that can create (unseen) problems. So, following this way, I wrote the corresponding function:

**#define** SETHIGH(REG, PIN) REG |= PIN

The last (fundamental) function is the one to invert a bit:

**#define** TOGGLE(REG, PIN) REG ^= PIN

From this point on, I don’t want to use anymore any symbol for operations on bit (apart from one only particular case, that we’ll se later).

Not always we know in advance which value we want a register pin to get, so we need another macro, where the value is a parameter, like this:

**typedef** **enum** PINVALUE\_VALUES {*PINLOW* = 0, *PINHIGH* = 1} PINVALUE;

**#define** SETVAL(REG, PIN, VALUE) \

**if** (VALUE == PINLOW) \

SETLOW(REG, PIN); \

**else** \

SETHIGH(REG, PIN);

As you can see, I don’t comment the single macros: I really hope they are totally auto explicative, as each good programmer know… and we know it, don’t we? ☺

As a former C++ programmer, I have to admit that this kind of source is pretty error-prone: the advantage is that code is really compact, so in this tradeoff we are not giving priority to code-correctness. I hope I can come back here and try to write a better version of this code, but for the time being…

With this last (for now!) definition:

**#define** WATCHDOG\_STOPPED() WDTCTL = WDTPW | WDTHOLD;

We are ready to write the new version of our first program:

**void** **main**(**void**)

{

WATCHDOG\_STOPPED();

**while** (1)

;

}

Not that much, but the executable is correctly 152+2 bytes: stopping the watchdog (1 instruction with 2 operands) in a 16 bit environment is just 6 bytes.

# Ports

Before writing the usual blinking led program, we have to prepare things. We could