

HELLO LED

In this section we will begin by learning how to use the Digital I/O module of the MSP430 microcontroller. We will start by using the API to handle the Digital I/O module functionality. Then we will see in a bit more detail how the Digital I/O module works by studying the user manual of the MSP430. Once we have described the module we will learn how to handle it by using code in the lowest level using assembly, and then do the same in a bit higher level by using C/C++, where we will focus on pointers and how to use them. Finally, we will learn how to create our own API by expanding some functionality of the API we are using.

1. A simple blink program (using the API to toggle an LED)

So let's begin by looking at a simple program to blink an LED, which looks like the following:

Before trying out the code, let's try to

```
1
2  #include "lib/include/Watchdog.hpp"
3  #include "lib/include/Gpio.hpp"
4
5  watchdog::WatchdogTimer dog;
6
7  Gpio::Pin led( Gpio::port::PORT1, Gpio::pins::pin0 );
8
9  int main()
10 {
11     dog.init( watchdog::config::hold::HOLD );
12
13     led.setMode( Gpio::config::mode::gpio );
14     led.setIOMode( Gpio::config::ioMode::output );
15
16     while( 1 )
17     {
18         common::delay_ms( 500 );
19         led.toggle();
20     }
21
22
23 }
24
```

understand what it does and why. To do this let's focus on specific parts of the code one by one.

```
1
2  #include "lib/include/Watchdog.hpp"
3  #include "lib/include/Gpio.hpp"
4
```

From lines 1 to 4, we are including the libraries that our application needs. In this case we are including 2 libraries: the Watchdog and the Gpio library. The first one let's us configure the watchdog timer of our microcontroller, which acts as a timer that once is fire it generates a reset condition. The second library allows to handle the Digital I/O functionality of our microcontroller, which will help us to configure our microcontroller to blink an LED.

```

5  watchdog::WatchdogTimer dog;
6
7  Gpio::Pin led( Gpio::port::PORT1, Gpio::pins::pin0 );
8

```

From lines 5 to 8 we start with the Object Oriented part of our code by creating 2 objects. The first one is called "*dog*" which is from the **WatchdogTimer** class. This object will allow us to handle the watchdog timer functionality. The second one is called "*led*" and is an object of the class **Pin**. This object will allow us to configure the Digital I/O related to a specific port and pin. The port and pin are passed to this object when created, and are **PORT1** and **pin0** which indicates we are making this led related to the pin 0 in port 1 of our microcontroller.

```

11      dog.init( watchdog::config::hold::HOLD );
12
13      led.setMode( Gpio::config::mode::gpio );
14      led.setIOMode( Gpio::config::ioMode::output );

```

After creating these objects, we start using these in lines 11 to 14. In line 11 we use the watchdog timer object by initializing the watchdog; we pass the **HOLD** parameter to make it hold, so that it doesn't start counting, which help us by not resetting the microcontroller every time.

In lines 13 to 14 we use our led object by configuring the related pin's functionality. First we set the mode of the pin to "**gpio**" so that is just a simple digital I/O pin (the other option is "**alternate**", which is used if the pin has to be used by another module, like the ADC converter, UART module, etc.). In line 14 we set the IO mode of the pin, by configuring it as an digital output, which is what we wanted to drive the LED in this pin.

```


16      while( 1 )
17      {
18          common::delay_ms( 500 );
19          led.toggle();
20      }

```

Finally, in lines 16 to 20 we have our main loop. In here we call a delay function in line 18 (from the module "**common**") which will make the microcontroller wait for 500 milliseconds. Then we use our led object again to use a method called toggle, which will make the related pin change its logic state from high to low and from low to high.

So that's all the code we need. In the next section we will learn how to compile, load our code into the microcontroller and debug it using Code Composer Studio.

2. Compiling, loading and debugging our program

So let's check if our code compiles, which means to check if there are any errors in our code. To do so let's use the **build tool** that looks like a little hammer. 

This tool will compile and link our program, and if there are any errors, will inform us in the console panel and the errors panel.

After the building finishes, we should see this on the console, which indicates a successful build.

```


<Linking>
remark #10371-D: (ULP 1.1) Detected no uses of low power mode state changing instructions
Finished building target: msp430cpp.out

```

```

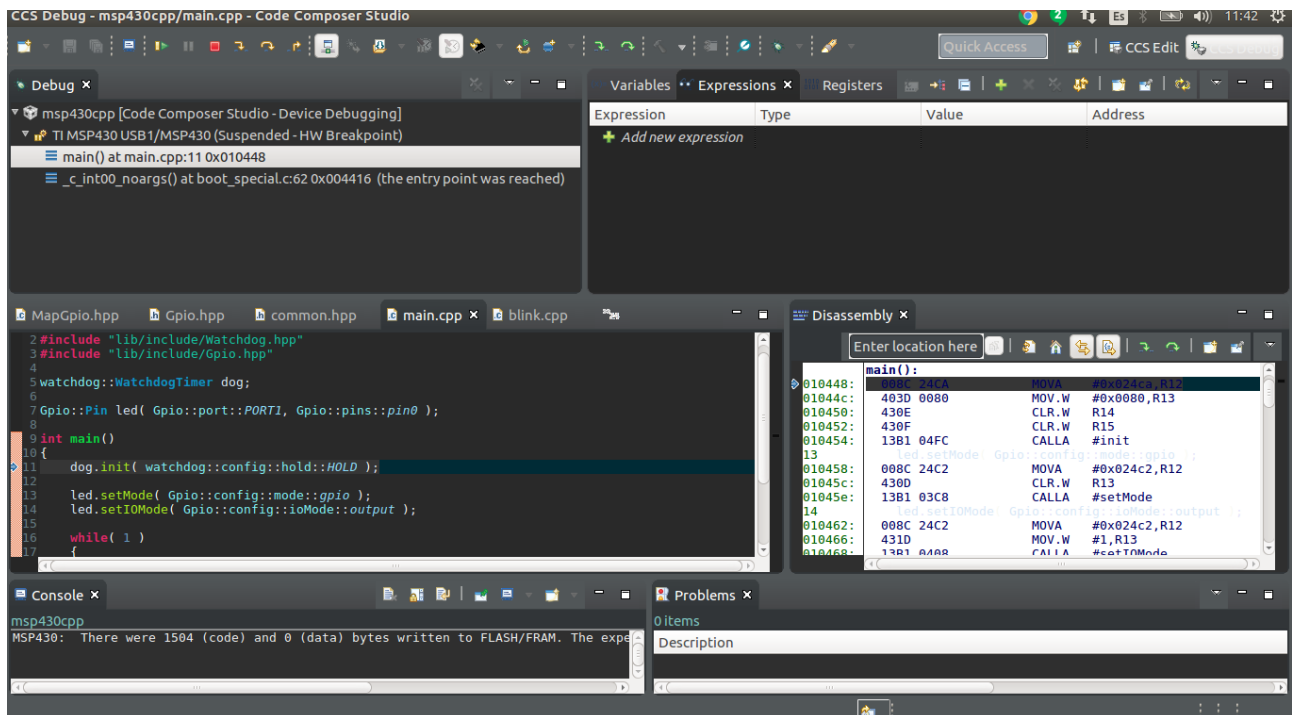
**** Build Finished ****

```

Now it's time to load our program, which is accomplished by using the **debug tool** (a little insect like icon in the top, close to the build icon) 

This action will take us to the debug perspective, which also does a process that loads binary program (msp430cpp.out) into the microcontroller before starting a debug session.

Once the debug session is ready, you should see something like this in Code Composer Studio.



This perspective has some panel which are useful while debugging :

- The code panel, similar to the code panel in the edit perspective. In here we see the code and can place **breakpoints**, which are used to stop the execution of the program.
- The stack panel, where we can see the **stack** and see how deep we are in our function calls.
- The watch panel, to watch registers values, expressions and variables in our program.
- The disassembly panel, to look at our code transformed into the actual assembly code that our microcontroller is running.