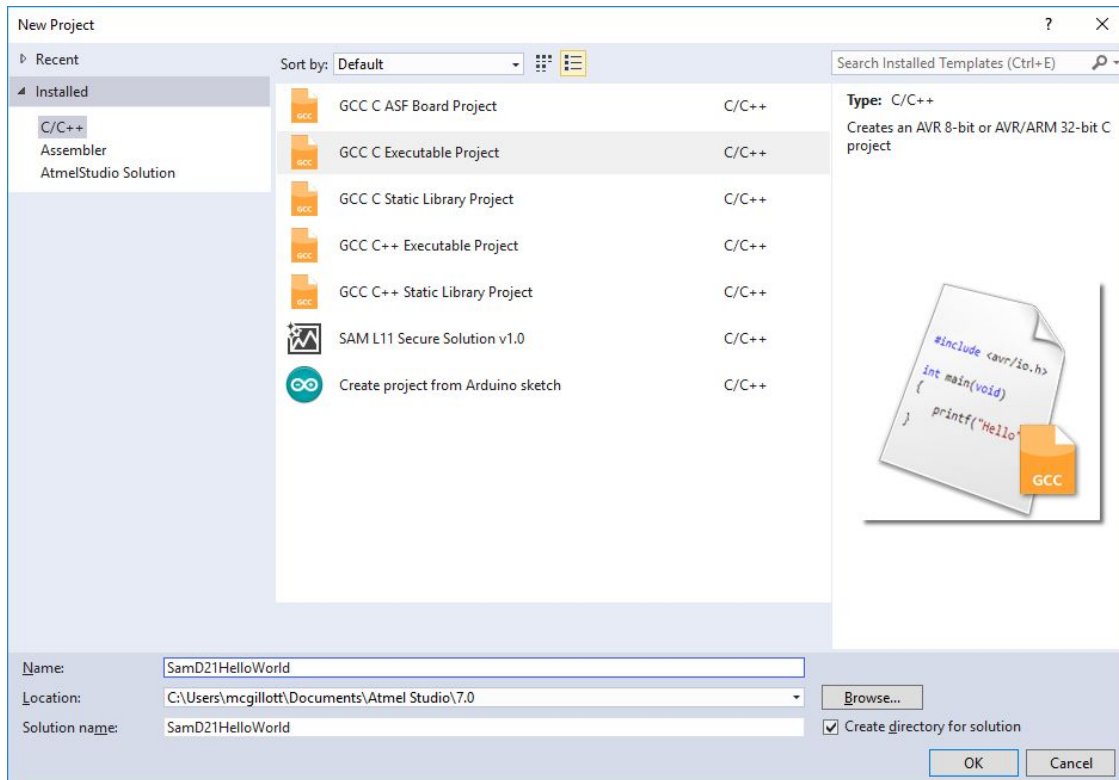


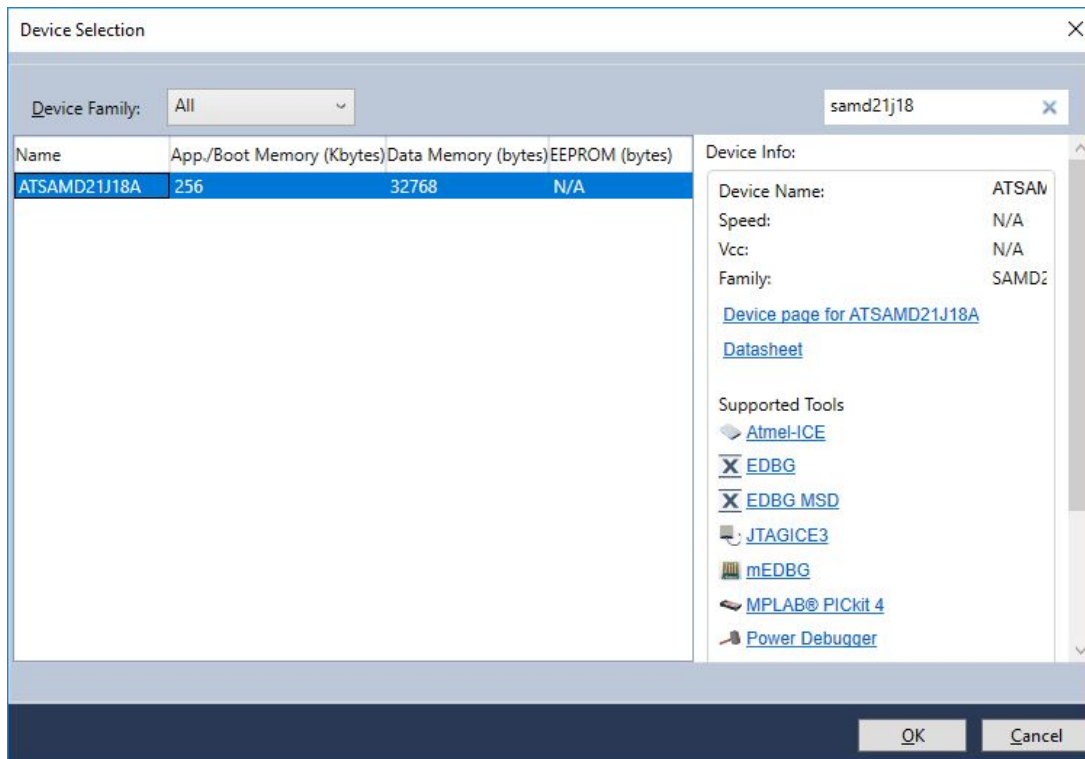
So you have installed Atmel Studio, now what?

To begin, you will need to create a new project. From the file menu, select new->project.

We are going in without an API, so select 'GCC C executable Project' from the list (C++ is also acceptable). Give your project a name and select OK.



Now you need to select your target device. I'm using the SAMD21XPLAINED board which contains the SAMD21J18A device, so I shall select this from the list. Use the search box to quickly locate your device.



Atmel studio will now create a simple project template containing a single 'main.c' file.

Usually the most difficult part of getting started with a ARM based microcontroller is figuring out the clock system. This is the device that distributes the clocks to the core and peripherals, and can sometimes take quite a bit of reading to get to grips with. Its typical when developing on a new platform, especially when not using an API, to be hopping all over the datasheet at first.

Let's take a dive into the SAMD21 datasheet. The first thing we need to know is what the default, or 'reset', state is of the clock system. Under the Clock System section (14), there's a sub-section called 'Clocks after Reset' (14.8) which clearly states the reset values of the clock system.

14.8 Clocks after Reset

On any reset the synchronous clocks start to their initial state:

- OSC8M is enabled and divided by 8
- Generic Generator 0 uses OSC8M as source and generates GCLK_MAIN
- CPU and BUS clocks are undivided

On a Power Reset, the GCLK module starts to its initial state:

- All Generic Clock Generators are disabled except
 - Generator 0 is using OSC8M as source without division and generates GCLK_MAIN
 - Generator 2 uses OSCULP32K as source without division
- All Generic Clocks are disabled except:
 - WDT Generic Clock uses the Generator 2 as source

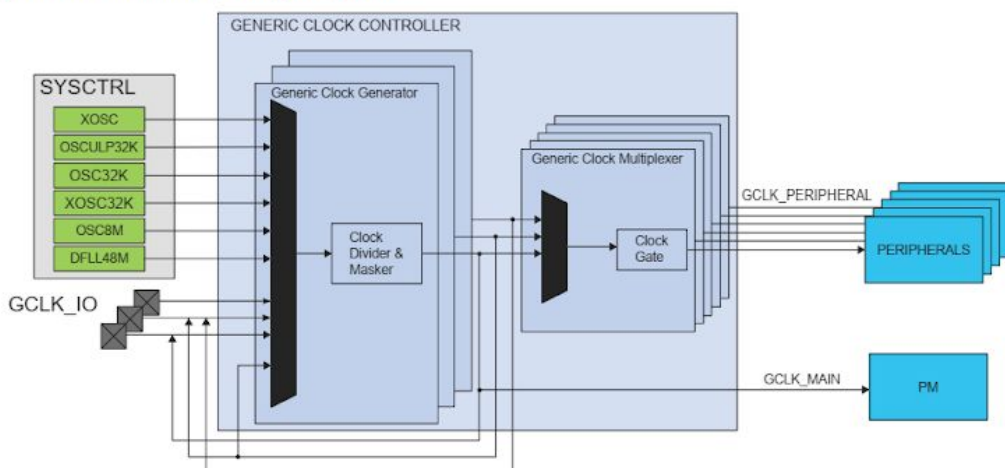
On a User Reset the GCLK module starts to its initial state, except for:

- Generic Clocks that are write-locked , i.e., the according WRTLOCK is set to 1 prior to Reset or WDT Generic Clock if the WDT Always-On at power on bit set in the NVM User Row
- Generic Clock is dedicated to the RTC if the RTC Generic Clock is enabled

On any reset the clock sources are reset to their initial state except the 32KHz clock sources which are reset only by a power reset.

The reset state shows that GCLK_MAIN is running from the internal OSC8M 8MHz oscillator with a DIV8 pre-scaler. But we aren't using ARM micros because they run a 1MHz, any ATmega will do that! Let's get this up to its maximum clock rate of 48MHz. How do we know this, well look in the datasheet at section 37.6 which details the maximum clocks frequencies.

Back to the datasheet. The information above gives a hint that the clock system is something to do with GCLK. So let's take a look at section 15 and get a more detailed look at that GCLK system. After a brief introduction, there's a very handy block diagram explaining the clock distribution.



This gives us a clue that we need to look at SYSCTRL to see what we can use as a suitable clock source. Using this new information now skip to the SYSCTRL section (17) in the datasheet.

Here we are presented with a nice list of the available clock systems on the micro. Conveniently, there's a clock which should provide the 48MHz we want.

- Digital Frequency Locked Loop (DFLL48M)
 - Internal oscillator with no external components
 - 48MHz output frequency
 - Operates standalone as a high-frequency programmable oscillator in Open-Loop mode
 - Operates as an accurate frequency multiplier against a known frequency in Closed-Loop mode

This isn't a simple oscillator though, it's a Frequency Lock Loop, which may require more configuration than other clock modules. More investigation is required, so let's move to the Functional Description part of the SYSCtrl, in section 17.6 of the datasheet. Each clock source is described in detail here with the DFLL48M detailed in 17.6.7.

There's plenty of detail here about the module, including the two modes it can be run in, open or closed. We will use closed loop mode as this will provide the best timing reference for our clock.

There's a nice section in 17.6.7.1 which details the closed loop operation including the procedure how to initialise it. One small problem, the first sentence says that in closed loop mode it is regulated against a reference clock, so at this point we need to put the DFLL on hold briefly whilst we investigate the options for the reference clock.

The table in section 37.6 tells us that the DFLL48M can accept a maximum of 33KHz as its reference clock, and conveniently the development board we are using (SAMD21 Xplained PRO) has a 32.768KHz crystal attached to the micro's XOSC32K unit (see section 7.2.1 for oscillator pinouts). In most microcontrollers, getting a crystal oscillator working is usually a single register configuration, and the SAMD21 is no different. Section 17.8.6 gives up the details of the register used to configure the XOSC32K unit. This section gives us all the information we need to get the oscillator running, so finally it's time to write some code.

Let's start by creating a new function that will eventually initialise all our required clocks, and within this we shall put the code for activating the XOSC32K.

```
void initClocks()
{
    //Get the XOSC32K running
    SYSCtrl->XOSC32K.reg = SYSCtrl_XOSC32K_ENABLE | //Enable bit set
        SYSCtrl_XOSC32K_XTALEN | //Xtal oscillator enabled (connected to XIN32/XOUT32)
        SYSCtrl_XOSC32K_AAMPEN | //Automatic amplitude control
        SYSCtrl_XOSC32K_EN32K | //Override the GPIO to use the pins for xtal
        (2<<SYSCtrl_XOSC32K_STARTUP_Pos); //Give it a bit of startup time;
}
```

If you refer back to the clock system block diagram above, you will see that a clock source is fed into a clock generator, then the output of the clock generator can be multiplexed to the peripherals. From this we can infer that the next thing we need to configure is a clock generator that takes the input from XOSC32K and outputs it to the DFLL reference. The clock generators are described in section 15, where we previously saw the block diagram. The SAMD21 contains 8 clock generators we can use, so we shall use clock generator 1 for our 32KHz input. We don't want to use clock gen 0 as this drives all the chip's system clocks such as the CPU, which we don't want running at 32KHz.

Two registers are used for this operation. The GENCTRL register described in section 15.8.4 is used to configure the generator, and then CLKCTRL (section 15.8.3) is used to configure the multiplexing. The third register we should also configure to be sure is the clock divisor register GENDIV (15.8.5) which can be used to divide the frequency of the clock generators output. Add the following code to the initClocks routine.

```
//Set XOSC32K as the source for CLKGEN1
GCLK->GENCTRL.reg = (1<<GCLK_GENCTRL_ID_Pos) | //Clock Generator 1
    GCLK_GENCTRL_GENEN | //Enable the clock generator
    GCLK_GENCTRL_SRC_XOSC32K ; //XOSC32K as the source
//Feed the output from CLKGEN1 into the DFLL48 reference
GCLK->CLKCTRL.reg = GCLK_CLKCTRL_ID_DFLL48 | //Output to DFLL48
```

```

        GCLK_CLKCTRL_GEN_GCLK1 |      //Source is GCLK1
        GCLK_CLKCTRL_CLKEN;           //Enable
//Set CLKGEN1 divisor to 1
GCLK->GENDIV.reg = (1<<GCLK_GENDIV_ID_Pos) | (1<<GCLK_GENDIV_DIV_Pos);

```

This configures CLKGEN1 to use the XOSC32K as its input, not to divide it, then feed its output to the DFLL48. Using the pre-defined macros makes this much more understandable without needing to consult the datasheet if you need to make a change later.

We finally have our reference signal being fed into the DFLL! Going back to section 17.6.7.1.2 we can follow the instructions to initialise it. Add the following code into the initClocks routine after the XOSC32K initialisation.

```

SYSCTRL->DFLLCTRL.reg = (uint16_t)(SYSCTRL_DFLLCTRL_ENABLE);           //Enable the DFLL48
while(!SYSCTRL->PCLKSR.bit.DFLLRDY);                                   //Wait for it to become ready

//Configure the multiplier and steps for the DFLL
SYSCTRL->DFLLMUL.reg = (31<<SYSCTRL_DFLLMUL_CSTEP_Pos) |               //Use half of max for course step
    (510<<SYSCTRL_DFLLMUL_FSTEP_Pos) |                               //And again for fine step
    (1465<<SYSCTRL_DFLLMUL_MUL_Pos);                                   //Multiplier is output/input (48M/32K)
//We could wait for the DFLL to lock, but to speed up the process we can use the factory calibration
//data to get it close as a starting point
uint32_t factoryCal;
factoryCal = (*(uint32_t*)FUSES_DFLL48M_COARSE_CAL_ADDR) & FUSES_DFLL48M_COARSE_CAL_Msk;
factoryCal >>= FUSES_DFLL48M_COARSE_CAL_Pos;
//Write them to the DFLL48 value register. Once the DFLL gets set to closed loop mode this register
becomes read only
SYSCTRL->DFLLVAL.bit.COARSE = factoryCal;
while(!SYSCTRL->PCLKSR.bit.DFLLRDY);                                   //Wait for it to become ready again (all data written) before
we switch to closed loop mode
//Switch to closed loop mode and don't output a clock until synchronized
SYSCTRL->DFLLCTRL.reg |= (SYSCTRL_DFLLCTRL_MODE | SYSCTRL_DFLLCTRL_WAITLOCK);

```

The first thing that's done here is to enable the DFLL unit, done by writing the appropriate bit in the DFLLCTRL register in the SYSCTRL module (see section 17.8.10). Next we write some values to the multiplier control register. There are three values in here, the course and fine step sizes and the multiplier. The step sizes regulate the maximum step the DFLL can take when adjusting frequency, which will affect both the lock time and overshoot. These don't really matter in our example since we shall wait for a good lock before using the output. The final value is the multiplier. This sets the ratio of output to input frequency. We want a 48MHz clock which will be synchronized with a 32768Hz clock, so the multiplication factor is 1465.

Next we preset the course tuning value of the DFLL. Since we will be using the DFLL in closed loop, this isn't essential, but putting in a value that gives a frequency close to what we want will improve lock time. The microcontroller has a set of built-in calibration values (see section 10.3) which can be used for various features. One of the calibration values is the course tuning value which will get the DFLL close to 48MHz. The code pulls this from the NVM calibration area and stores it to the DFLLVAL register. Once the DFLL is set to closed loop mode, this register becomes read-only so we must do this before switching mode.

Finally, the DFLLCTRL register is written to again, this time to enable the closed loop mode and to set a bit which prevents the clock from outputting a signal until lock has been obtained. This will ensure the clock is running at the right frequency when we want to use it.

The final step of our clock initialisation is to switch the master clock over to using our new 48MHz clock, but there's one last thing we need to check before doing this. Most microcontroller flash memory isn't capable of running at the same speed as the CPU, and so if we set a speed faster than the flash controller can read at, the system will crash or run unpredictably. The table in section 37.12 shows how many wait states we need to enable on the flash memory controller at various frequencies and operation voltages. My microcontroller is powered from 3.3V, which means we need to add 1 wait state to the flash controller. A quick skim through section 22 on the datasheet, which details the

flash memory controller (NVMCTRL), shows us that the wait states are set in the CTRLB register. so we need to do this before switching the clock to 48MHz.

```
//Set the NVM access time to 1 wait state (required for 48MHz operation)
NVMCTRL->CTRLB.bit.RWS = 1;

//Set CLKGEN0 (Master clock) to use the DFLL48 as its source
GCLK->GENCTRL.reg = (0<<GCLK_GENCTRL_ID_Pos) | //Clock Generator 0
                    GCLK_GENCTRL_GENEN | //Enable the clock generator
                    GCLK_GENCTRL_SRC_DFLL48M ; //DFLL48M as the source

//Set CLKGEN0 divisor to 1
GCLK->GENDIV.reg = (0<<GCLK_GENDIV_ID_Pos) | (1<<GCLK_GENDIV_DIV_Pos);
```

At this point we now have a 32KHz crystal feeding the 48MHz DFLL which drives the CPU core and AHB/APB busses, so let's configure something so we can tell we are running at the right frequency.

For this we shall use one of the microcontrollers standard timer/counter units. Section 30 of the datasheet details how each of these units work. For our example we shall use TC4 (TC0-TC2 are actually TCC units). In our code, create another routine called initTC4 and add the following code.

```
void initTC4()
{
    //Drive the clock to TC4 from CLKGEN0
    GCLK->CLKCTRL.reg = GCLK_CLKCTRL_ID_TC4_TC5 | //TC4 and TC5
                      GCLK_CLKCTRL_GEN_GCLK0 | //from GCLK0
                      GCLK_CLKCTRL_CLKEN; //Enabled

    //Enable clock access from the APB bus (allow register access etc)
    //See section 16.6.2.6 for peripherals that are defaultly clocked from the APB
    PM->APBCMASK.reg |= PM_APBCMASK_TC4;

    //Enable TC4 in 16bit counter mode
    TC4->COUNT16.CTRLA.reg = TC_CTRLA_ENABLE | TC_CTRLA_MODE_COUNT16;

    //Enable the interrupt to occur on overflow
    TC4->COUNT16.INTENSET.reg = TC_INTENSET_OVF;

    //Enable TC4 interrupts in the NVIC
    NVIC_EnableIRQ(TC4_IRQn);
}
```

The first part of this configures the GCLK multiplexor to supply the clock to TC4 from CLKGEN0. The table in section 15.8.4 shows that TC4 and TC5 are on the same multiplexer, so must be driven from the same clock, as we aren't using TC5 in this example this isn't a problem.

Next we need to enable to clock to TC4 on the peripheral bus. This enabled the TC4 units registers to be accessed over the data bus.

The next two lines initialize the timer counter unit as a simple 16 bit counter with an interrupt on overflow. The final line enabled the TC4 interrupts in the ARM NVIC.

Interrupts can be quite complicated, but in this case we are only using one. Each peripheral may have several ways that an interrupt can be driven. These are usually controller with 'interrupt enable' flags in the peripherals control register. Any of this interrupts, if enabled, will register that an interrupt is pending from the peripheral in the NVIC, the ARM's interrupt controller. The interrupt can be stopped here if the NVIC isn't configured to allow the peripheral to drive the interrupt so we need to tell the NVIC to allow interrupts from TC4.

Since we are going to use the interrupt from the timer to cause an event, we need to write an interrupt handler for the TC4 unit. Typically, interrupt handlers are already weakly defined in the devices header files and just need body code to function. In most cases, these are in the form of xxxx_Handler, where xxxx is the name of the unit calling the interrupt.

Add the following code to the project to provide an interrupt handler for TC4.

```
void TC4_Handler()
{
    static uint32_t ledToggleCounter = 0;
    TC4->COUNT16.INTFLAG.reg = TC_INTFLAG_OVF;    //Clear the interrupt flag for overflow
    if (++ledToggleCounter == 366)                  //Count to 366 ((48E6/65536)/2)
    {
        ledToggleCounter = 0;                      //Reset the counter
        PORT->Group[1].OUTTGL.reg = 1<<30;         //Toggle the LED pin PB30
    }
}
```

The first thing we need to do in the interrupt is clear any interrupt flags that are set (and not automatically cleared). If these are not cleared, the interrupt will fire again as soon as it exits. The rest of the code in here counts to 366 then toggles the output of one of the GPIO pins (PB30). If all our clocks are correct, the LED should toggle every half a second.

Finally we can go back to the main() function and bring everything together:

```
int main(void)
{
    /* Initialize the SAM system */
    SystemInit();

    initClocks();
    initTC4();

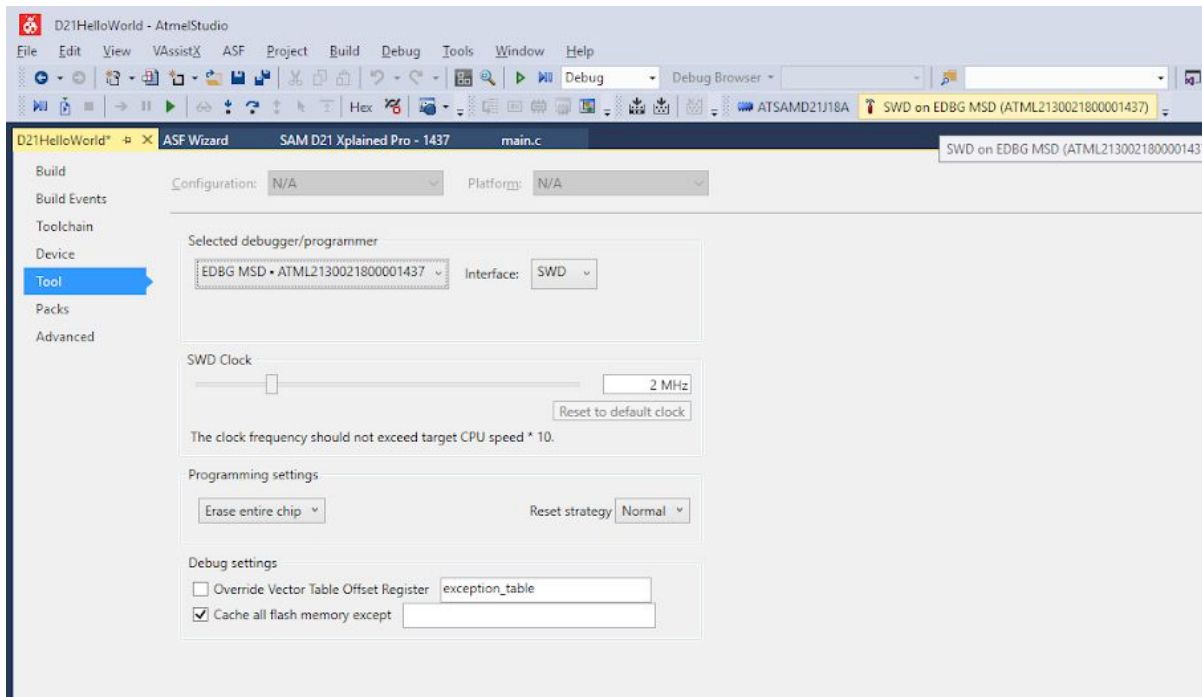
    //Enable Pin PB30 as an output
    PORT->Group[1].DIRSET.reg |= 1<<30;

    //Global interrupt enable
    __enable_irq();

    while (1)
    {
        //Do nothing
    }
}
```

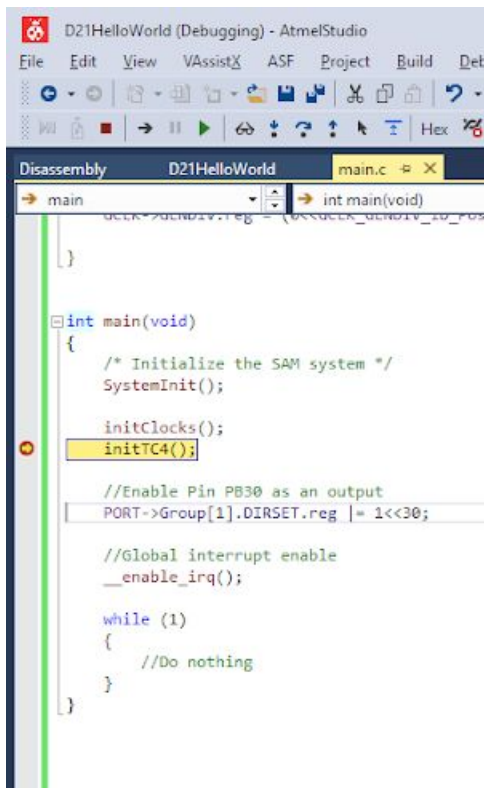
This should get our SAMD21 running and flash the onboard LED attached to PB30 with a frequency of 1Hz.

Now lets deploy this to our micro. We need to select the programmer that Atmel Studio will use. These can be any evaluation board with an on-board debugger, or a seperate J-Link compatible SWD programmer. Click the hammer icon on the toolbar to go through to the tool selection and use the drop down list to select the correct tool.



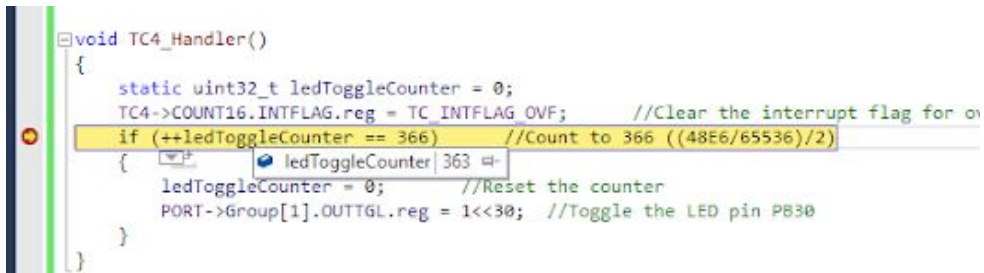
Next, ensure the target is powered, then press the 'play' button on the toolbar. The code will be built and deployed to the target, and the IDE will enter interactive debugging mode.

In debug mode, the microcontroller can be stopped on demand by clicking the 'pause' icon, or can be made to stop at a chosen point by inserting breakpoints. To add a breakpoint, click in the margin on the code editing area, this will add a red dot and the microcontroller will cease execution when this line is reached.



Note: The IDE attempts to insert the breakpoint based on the compiled output. If the compiler settings are using optimisation, this may change the order of execution which can lead to unexpected behaviour when debugging. Its best where possible to turn the optimiser to off (-O0) when doing live debugging.

Once the micro has entered the break mode, you can single step through the code line by line, or reset the micro by using the buttons on the toolbar. You can also inspect the value of variables by either hovering over it with the cursor or adding it to a watch list.



```
void TC4_Handler()
{
    static uint32_t ledToggleCounter = 0;
    TC4->COUNT16.INTFLAG.reg = TC_INTFLAG_OVF; //Clear the interrupt flag for overflow
    if (++ledToggleCounter == 366) //Count to 366 ((48E6/65536)/2)
    {
        ledToggleCounter = 0; //Reset the counter
        PORT->Group[1].OUTTGL.reg = 1<<30; //Toggle the LED pin PB30
    }
}
```

The screenshot shows a code editor with a C function `TC4_Handler()`. A red circle with a white dot, representing a debugger breakpoint, is placed on the line `if (++ledToggleCounter == 366)`. The line is highlighted in yellow. A tooltip is visible over the variable `ledToggleCounter` in the `if` statement, showing its current value as 363. The code includes comments for clearing the interrupt flag, counting to 366, resetting the counter, and toggling the LED pin PB30.