

## Starting in DAVE IDE

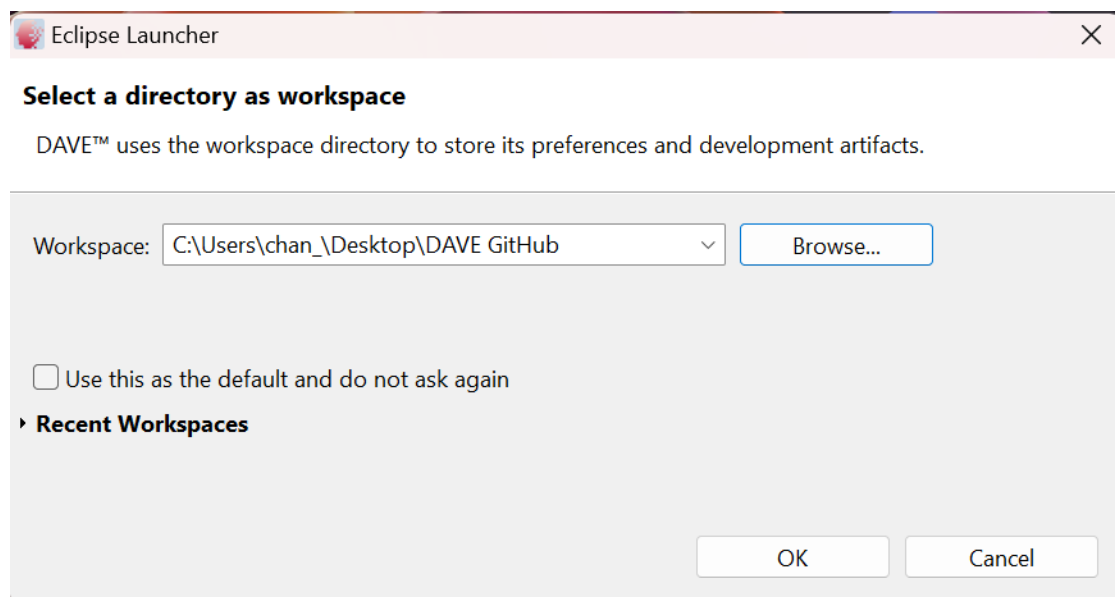
### Layout and Toolbar



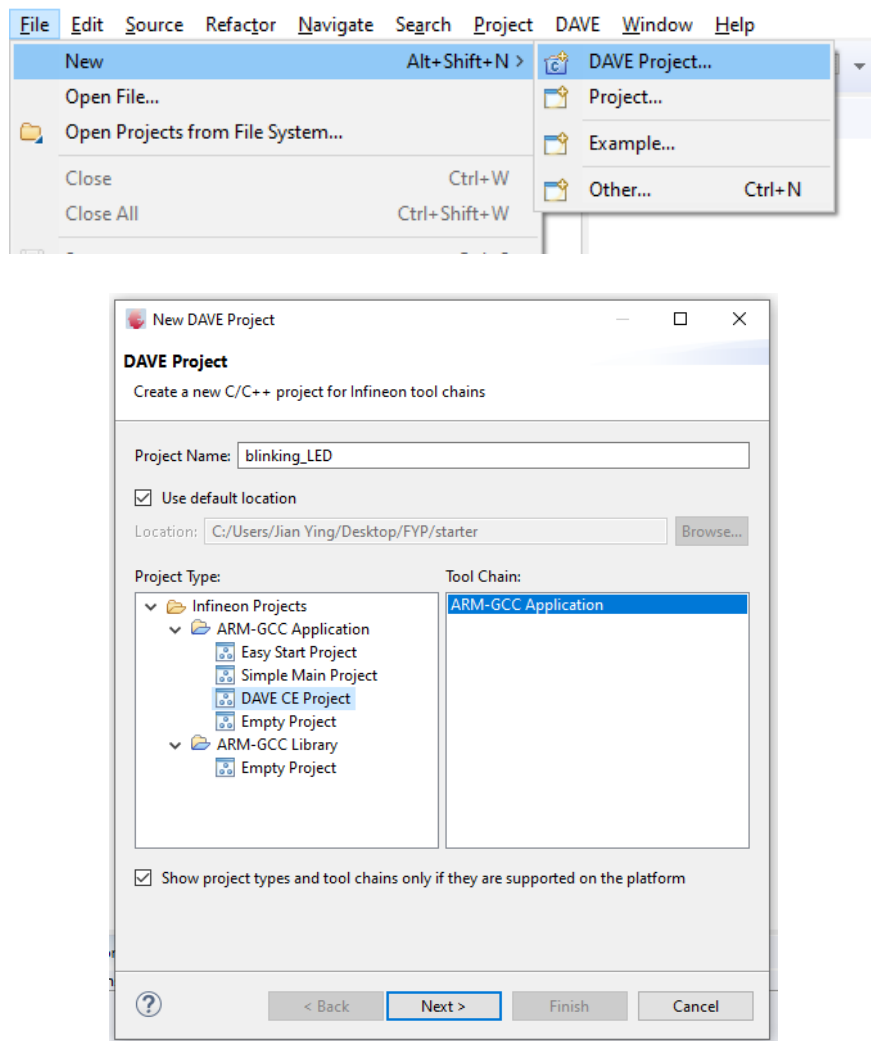
1. Build Active Project – ‘Compile’ the Project
2. Rebuild Active Project – ‘Re-compile’ the Project
3. Add New APP – Opens the DAVE APP Library and select APPs to add into Project
4. Install DAVE APP / Example / Device Library – Install additional APP / Example Project files / Device’s Library from local directory
5. Report – View “Resource Mapping”, “Pin Allocator”, “Signal Assignment” and “APPs” in a table form.
6. Manual Pin Allocator – Manual I/O pin assignment in a table form
7. Manual Resource Assignment – View resource requirements of APPs
8. Global Interrupt – Configure interrupt pre-emption priority and sub-priority
9. Generate Code – Process of generating all necessary code for APPs within Project
10. Pin Mapping Perspective – GUI for the pin assignment of the chipset
11. Debug – Opens up Debug Configuration and begin process of debugging onto microcontroller

### Creating a new project

Launch “DAVE IDE” and select a directory/folder as the workspace.



“File” > “New” > “DAVE Project” to create a new project within the workspace.

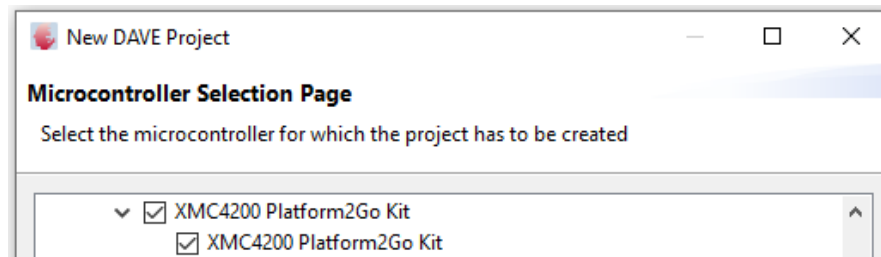


Give it a “Project Name” and select an appropriate project type.

## Project Types

- Simple Main Project – equipped with ‘startup files’, a set of low-level device drivers, CMSIS (Common Microcontroller Software Interface Standard) libraries, linker script and empty main function. For user to expand on the project.
- Easy Start Project – Build on an empty project and provides a toy ADC application.
- DAVE CE (Code Engine) Project – Access to full power of CE projects. Includes various drives and startup files, access to configurable applications library (DAVE APPs – xxx).
- Empty Project – Does not contain a main function. Copy completed application developed outside of DAVE into the project and start using the IDE. DAVE will automatically create makefiles and build environment.

Select the corresponding Microcontroller (XMC4200 Platform2Go) for the Project to ensure the respective libraries are loaded.



What is included in the Project file?

Dave

- Generated – .c and .h files for added DAVE APPs will appear under this folder
- Model – Documentations and other files for added DAVE APPs

Libraries

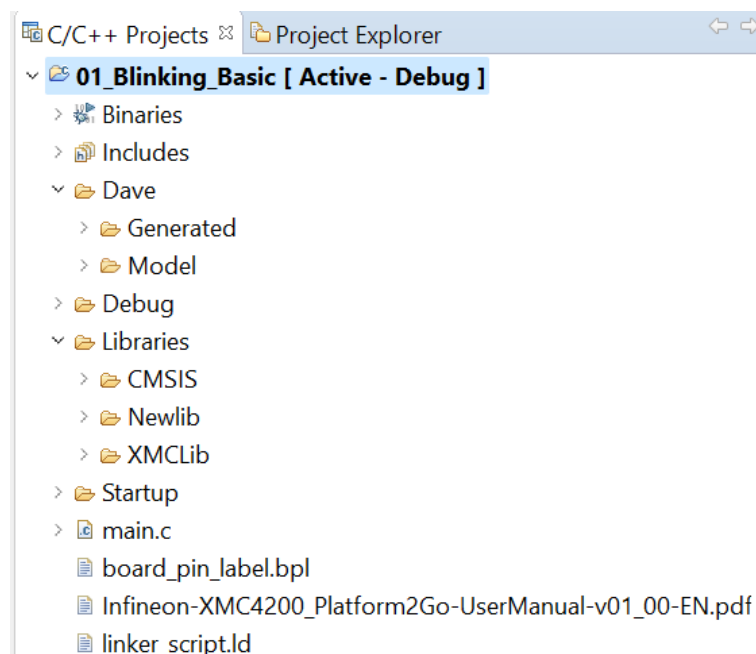
- CMSIS – enables consistent device support and simple software interfaces
- XMCLib – XMC low-level peripheral drivers for chosen microcontroller
- NewLib – stubs (placeholders) for implementation

Startup

- Containing vector table, default interrupt handlers and program loader

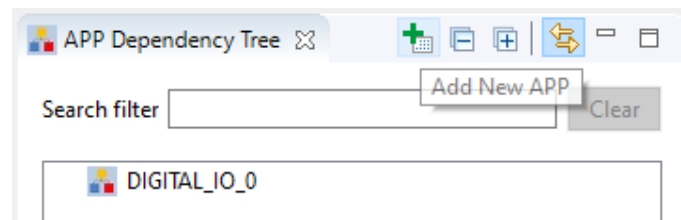
Main.c

- The main application code, implementation of the code (akin to the 'int main(void)')

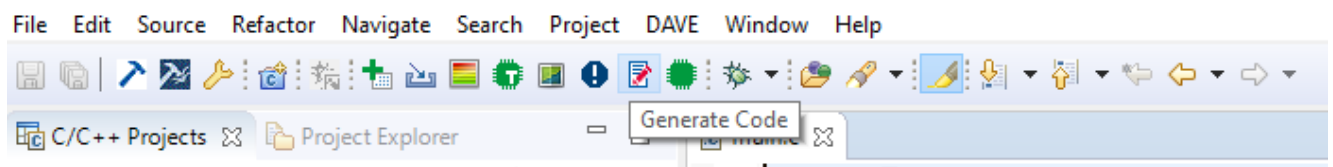


## APP Dependency Tree

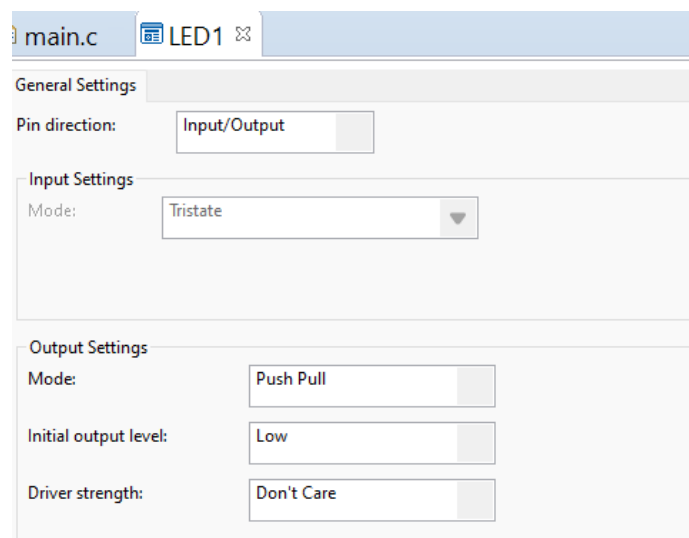
Under the “APP Dependency Tree”, all APPs used in the project will appear under this section. “Add New APP” to select APPs from the library and add into the Project.




Make sure to “Generate Code” after addition of any APPs to generate the necessary files (respective folder will be created under “Dave” > “Generated” containing the config files, .c, .h) required for the APP to work.



Double clicking each of the APP under the Dependency Tree will open the API to “Configure APP Instance”. Below is an example based on DIGITAL\_IO’s APP.

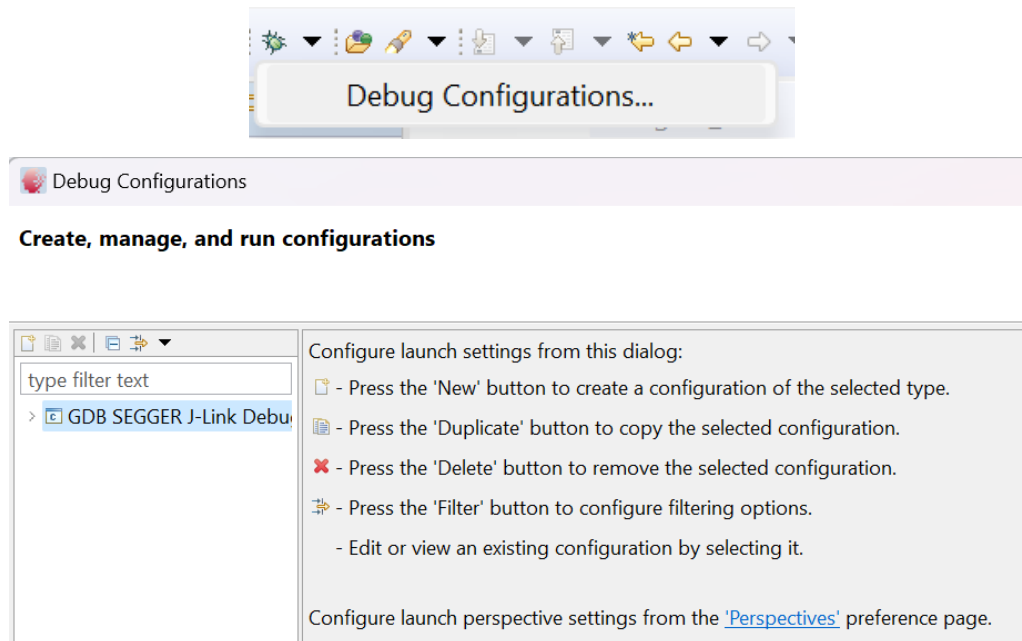


## Coding & Building Project Code

Tutorial for a few basic sample projects can be found [here](#). Begin coding the project within the “main.c” file located within the project. Upon completion of writing the program and ready for debugging, make sure to “Build Active Project”  . The initial build will take a longer time as it is building all libraries. Make sure to “Build Active Project” again after making any changes to the code file before debugging.

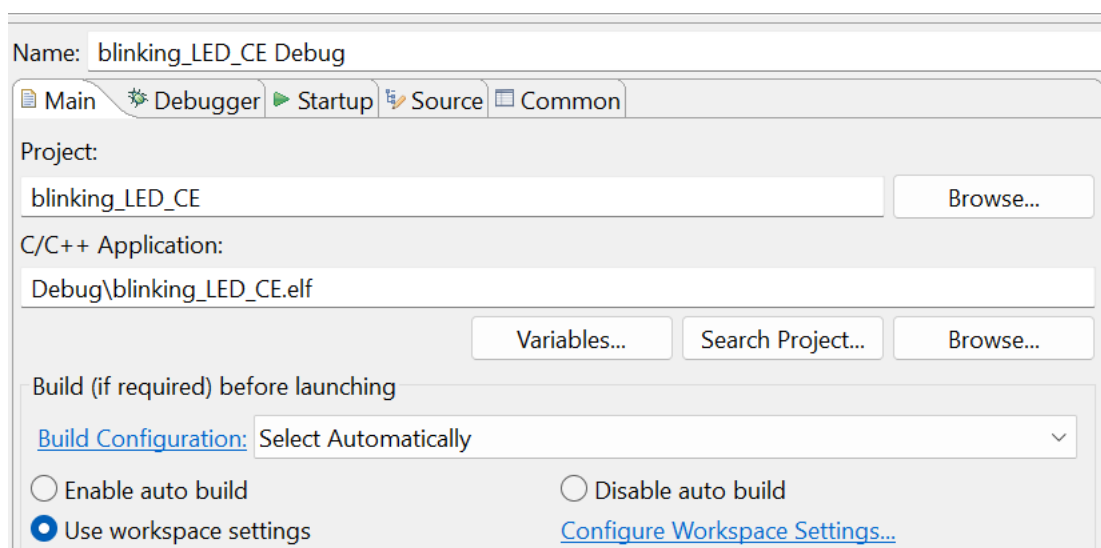
## Debugging

Connect the microcontroller via USB and begin to Debug. You will initially see only “Debug Configurations...”. Click to open the “Debug Configurations” window. Make sure to have J-Link (or any other debugger/flashing tools) and its necessary applications have been installed at this point.



Click the 'New' button or double click “GDB SEGGER J-Link Debugger” to create a new debugging configuration.

Check that it is debugging the correct project and project file. If the C/C++ Application file is empty, either 'Build Active Project' first or check for compilation errors within your code.



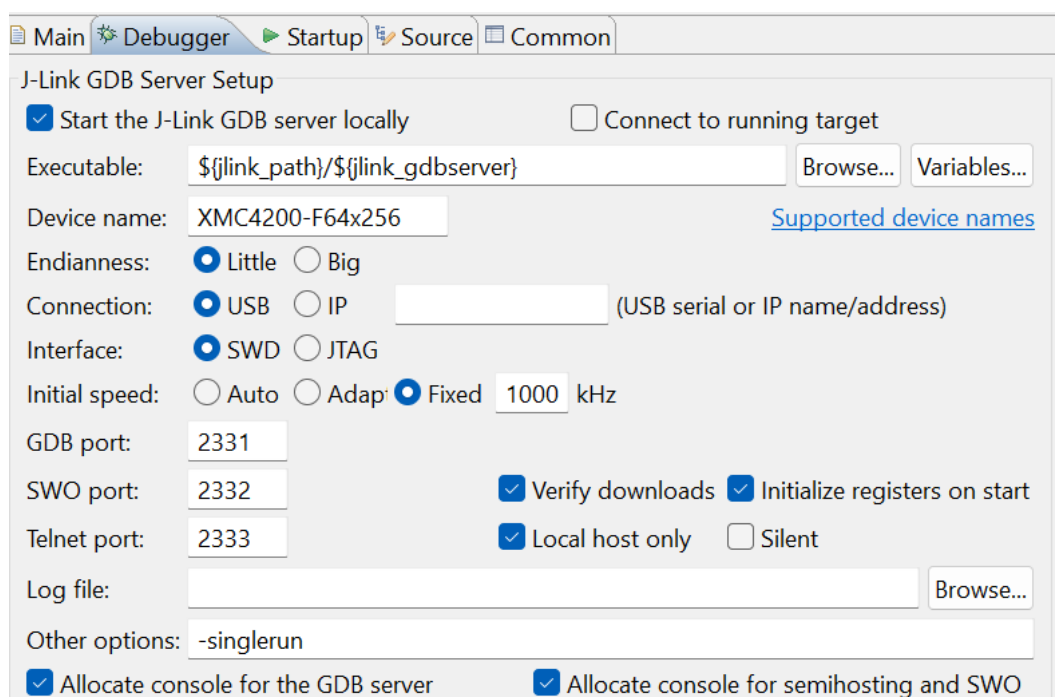
Under the 'Debugger' tab, the "Other options" string is:


```
-singlerun -strict -timeout 0
```

Update the "Other options" string to the following to avoid error messages:

```
-singlerun
```

Also ensure that the 'Device Name' matches the microcontroller model (selected in previous [step](#)), else the connection will fail to set up.

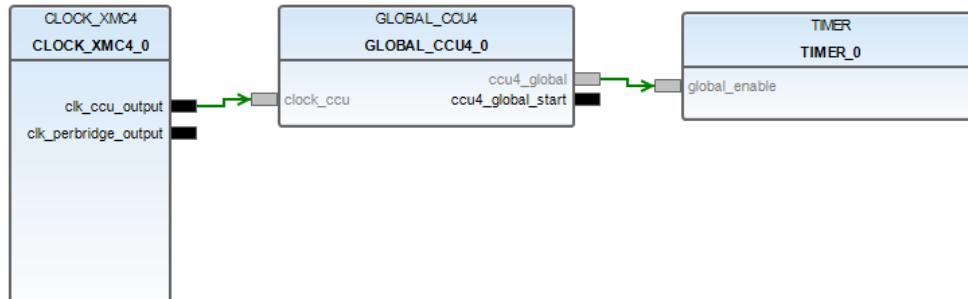


Once debugging process is done, the Debug tab should open. Press the 'Resume'  or F8 key to run the program. Your program should be executed and running on the microcontroller at this moment.

## Other Windows & Tabs

### HW Signal Connectivity

For visualisation of the 'hardware' signal connections of all the APPs. Certain APPs have other dependency such as the TIMER APP requiring a timer slice (CCU4), as shown below:

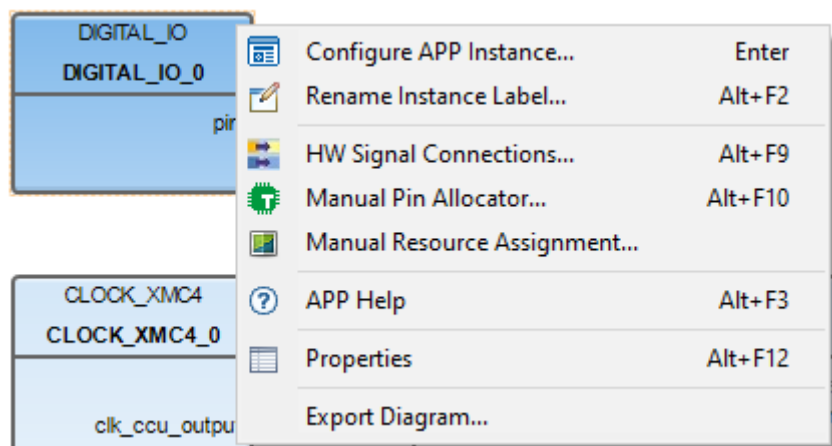


### Configuring individual APP Instances

Each APP in the DAVE Library can be added into a project multiple times and each APP is called an 'instance'. The number of instances for each APP type may vary based on the resources available on the microcontroller (e.g. there are limited timer slices).

Right-click on the individual window to configure the instance of the APP

- Configure APP Instance – opens up the user API to configure the settings.
- Rename Instance Label – Rename the instance to easily call upon in the code, this name is also known as the **handle\_ptr** (handle pointer).
- HW Signal Connections – Hardware connection between Source Signal to a different (target) APP Instance.
- Manual Pin Allocator – Input/Output pin assignment (eg. DIGITAL\_IO as output to user-configurable LED1 on P0.1).



## Sample Projects

All project files can be downloaded from the GitHub repository.

### Blinking LED (Hard-coded / Based on CPU Speed)

#### APP Dependency

##### 1. DIGITAL\_IO

#### Project File

This project will make use of “DIGITAL\_IO” APP for pin allocation of the user-configurable LED on P0.1 (specifically for XMC4200 Platform2Go) toggling at a hard-coded interval with reference to the CPU Clock Speed of 80MHz. Below is a snippet of the code.

```
DIGITAL_IO_ToggleOutput(&LED1);  
for(float count = 0 ; count < 2500000 ; count++); // 0.5 sec ON | 0.5 sec OFF  
DIGITAL_IO_ToggleOutput(&LED1);  
for(float count = 0 ; count < 2500000 ; count++); // 0.5 sec OFF | 0.5 sec ON
```

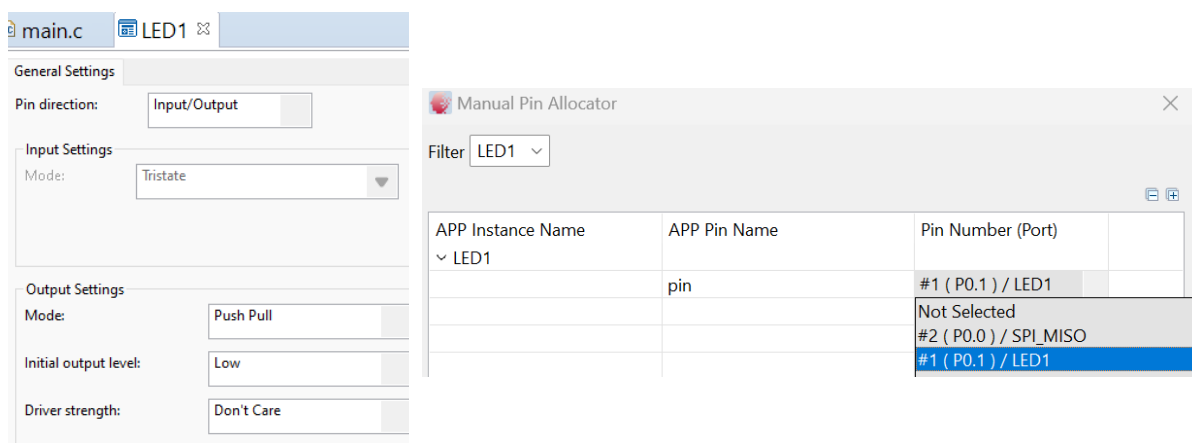
The instance of “DIGITAL\_IO” has also been renamed to “LED1” for easier reference.

The for-loop is a simple method to add a time delay before each toggling of the output pin and is derived by the follow formula, based on the CPU Clock Speed, where t is number of seconds.

$$count = \frac{CPU\ Clock\ Speed}{16 \div t} = \frac{80\ 000\ 000}{16 \div 0.5second} = 2\ 500\ 000$$

#### Additional Configurations

Configure APP Instance and change the “Pin Direction” to Input/Output so that it can be assigned to an output pin. Also, make sure to manually pin allocate the ‘pin’ to the corresponding “Pin Number”, P0.1 in this case.





## Blinking LED with TIMER APP (Common 'Clock Speed')

### APP Dependency

1. DIGITAL\_IO
2. TIMER (Use CCU4 Timer Module)
3. INTERRUPT

### Project File

Working off the previous project with the addition of "TIMER" and "INTERRUPT" APP. The "TIMER" APP uses a unit of timer slice (either CCU4 or CCU8) to generate an accurate delay / set desired time interval in micro-seconds. The "INTERRUPT" APP to create interrupt connectivity triggered by the TIMER's time interval (called an event).

The idea of this project is to create a common 'clock speed' / tick-rate based on a common factor of any required/desired frequency in the program and having a counter until the appropriate number of ticks (or time delay) has passed. For example, setting the TIMER's time interval as 0.1ms (milli-second) / 100µs (micro-second) as the tick-rate. To achieve a toggling frequency of the LED at 20Hz would require 500 ticks while at 50Hz it would require 200 ticks.

```
#define TIMER_d1MS 100*100U
volatile uint32_t timetick_count = 0; // Global variable (unsigned-int 32bit)
for time tick count
void interval_handler(void)
{
    TIMER_ClearEvent(&TIMER);
    TIMER_Stop(&TIMER);
    /* Set the time division for the TIMER */
    TIMER_SetTimeInterval(&TIMER , TIMER_d1MS);

    if(timetick_count == 200) // Condition will be based on the desired frequency. Eg: 20Hz @
interval of 0.1ms = 500 ticks
    {
        DIGITAL_IO_ToggleOutput(&LED1);
        timetick_count = 0;
    }
    else
    {
        timetick_count++;
    }
    TIMER_Start(&TIMER);
}
```

This is a new function within the main.c file, as referenced by the Interrupt Handler (user-configurable) that is called upon at every time interval as defined by the user

(event generation). The desired time interval is also defined at the start of the main.c file or it can be configured in the User API for the TIMER APP. Note that the base time unit is in micro-second and there is a scaling factor of 100 times to be applied, such that:

$$0.1ms = 100\mu s \times 100U$$

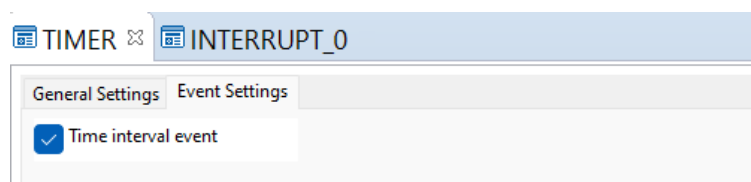
Scaling factor ↙

The variable keeping track on the number of ticks (timetick\_count) is declared as an unsigned integer and allocated 32-bit memory with the additional tag of 'volatile'. This ensures that there is no aggressive optimisation in the implementation phase by the compiler as the variable only updates its value when the function 'interval\_handler' is called upon (read more [here](#)).

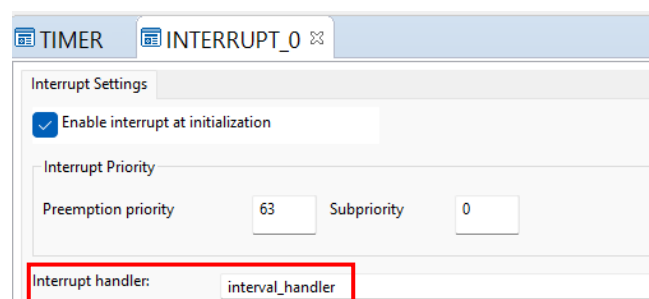
This function is called upon when an interrupt occurs (the 'event'), generated by the TIMER APP at the configured time interval. The function first clears the interrupt status so that subsequent events will be perceived as a new event. The TIMER is then stopped and time interval re-defined as 0.1ms. If the tick-rate has reached the desired time delay, it will toggle the LED output and reset the count. Else, the tick-rate will continue to increase. The TIMER will start again after checking before conditions and call upon this function after the TIMER has counted for 0.1ms, repeating the entire sequence.

### Additional Configuration

For the TIMER APP, enable 'Time interval event' so that an event can be generated at the user defined time interval.

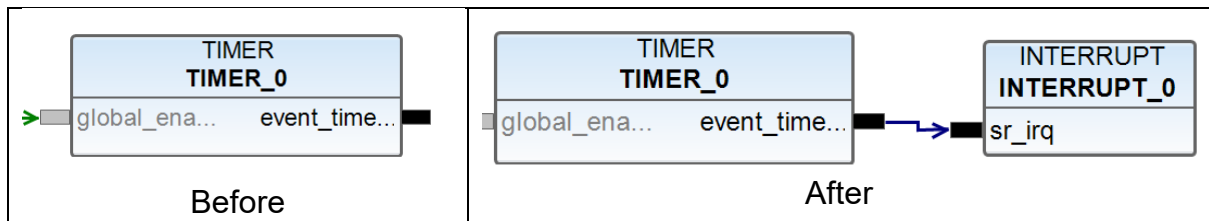


For the INTERRUPT APP, ensure the handler name matches the name of the function.



Setup the following 'HW Signal Connection' between the TIMER and INTERRUPT APP.

HW Signal Connections					
Filter <span>TIMER_0</span>					
	Source APP Instanc...	Source Signal	Connect To	Target APP Instance...	Target Signal
+	TIMER_0				
		event_time_interva	---->	INTERRUPT_0	sr_irq
		Not Selected	---->	Not Selected	Not Selected



## Blinking LED with TIMER APP (Varying Intervals)

### APP Dependency

1. DIGITAL\_IO
2. TIMER (Use CCU4 Timer Module)
3. INTERRUPT

### Project File

Working off the previous project of blinking LED at a desired frequency based on a common clock speed, this project will showcase the TIMER APP can be re-configured during run-time as written within the program while utilising the same APP Dependency. The program has two 5-seconds sequences such that the LED will toggle at 0.1s or 0.5s interval.

```
#define TIMER_500MS 500000*100U // Global variable for 500ms (0.5s) , actual
value scaled by x100
#define TIMER_100MS 100000*100U // Global variable for 100ms (0.1s) , actual
value scaled by x100
volatile uint32_t timetick_count = 0; // Global variable (unsigned-int 32bit)
for time tick count

void interval_handler(void)
{
    /* Clears interrupt status, next event will be considered new */
    TIMER_ClearEvent(&TIMER_0);
    /* Toggles LED */
    DIGITAL_IO_ToggleOutput(&LED1);
    /* A 'count' increment for every tick (varying) of the TIMER */
    timetick_count++;

    /* */
    if(timetick_count == 10) //if TRUE > Stop Timer & Set Interval to 0.1sec
    {
        TIMER_Stop(&TIMER_0); // Stops TIMER if running, no further event.
        /* Sets new time interval (0.1s) */
        TIMER_SetTimeInterval(&TIMER_0 , TIMER_100MS);
    }

    else if (timetick_count == 60 ) //if TRUE > Stop Timer & Set Interval to
0.5sec & Reset count
    {
        TIMER_Stop(&TIMER_0); // Stops TIMER if running, no further event.
        /* Sets new time interval (0.1s) */
        TIMER_SetTimeInterval(&TIMER_0 , TIMER_500MS);
        /* Resets count */
        timetick_count = 0;
    }

    TIMER_Start(&TIMER_0); // Starts TIMER
}
```

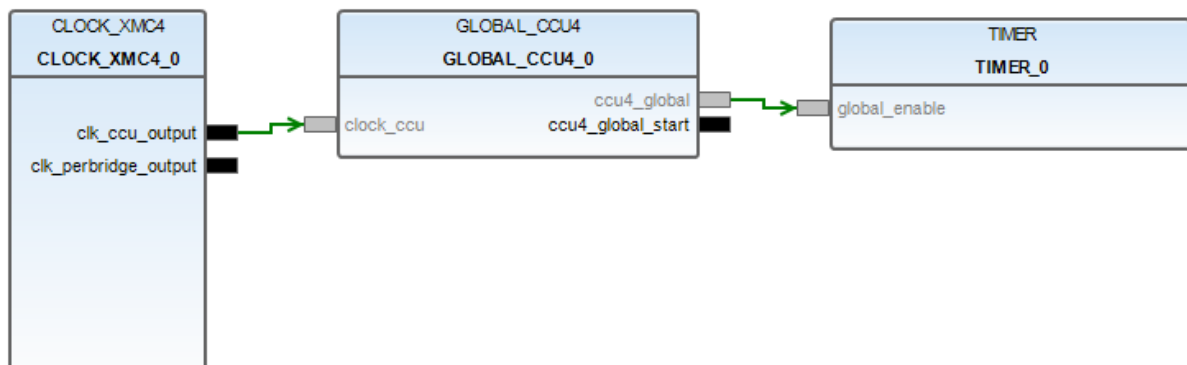
The function is called upon when an interrupt is triggered (at every time interval) and clears the interrupt status, toggles the output of the LED pin, increases the `timetick_count`, and starts the timer. The count immediately reaches '10' as the program initialised with a default `TimerInterval` and enter the if-condition – stops the timer and updates `TimeInterval` to 100ms, exits the if-condition and starts the timer. The functions loops – clears the interrupt event, toggles LED and `timetick_count` increases by 1 every 100ms (or 0.1s), repeats the previous steps until count reaches 60 (increment of 50 count every 0.1s equates to a 5s period). When `timetick_count` reaches 60, it enter the else-if-condition – stops the timer, updates `TimeInterval` to 500ms, and resets `timetick_count`, exits the condition and starts the timer (now running at 500ms intervals). The function loops again repeating the sequence while the count increases every 0.5s and increment of 10 count equates to a 5s period.

#### Additional Configurations

Configurations for the APPs is similar to the previous project, as seen [here](#).

## HW Signal Connectivity

Visualise the 'hardware' signal connections of all the APPs.



Right-click on the individual window to configure the instance of the APP

- Configure APP Instance – opens up the GUI to configure the settings
- Rename Instance Label – Rename the instance to easily call upon in the Project (akin to naming a variable)
- HW Signal Connections – Hardware connection between Source Signal to a different (target) APP Instance
- Manual Pin Allocator – Input/Output pin assignment (eg. DIGITAL\_IO as output to user-configurable LED1 on P0.1)

