Lab 2: Clocks and Timers

Objective

Demonstrate knowledge of synchronous circuitry and timers.

Background

Please note that this lab uses terminology that was introduced in previous labs.

B1: Synchronous Digital Circuits

The hardware configurations in past labs have been exclusively using asynchronous digital or analog circuits, where the components immediately respond to changes in input. However, there are many situations where components need to update in sync with each other, which leads to synchronous circuit design. For example, when several components produce outputs that are processed by one component, the processing component needs to know when the producing components have finished. Otherwise, it may be processing a new input with the old results from the other inputs, potentially creating an undesirable, unstable result.

The most common way to synchronize components is using a clock. A clock is a digital signal that oscillates between high and low a specified number of times per second, called the period or frequency (see Figure 1). The moment a clock goes high is called a rising edge and the moment a clock goes low is called a falling edge. The clock also has a duty cycle of 50%, which means the signal is high for half of the period. How these clocks are used depends on the implementation, but in most cases (including the PSoC) components that synchronize with each other use the same clock as an input. With the PSoC, components wait until a rising edge before changing state. As long as the clock is no faster than the component that takes the longest to process input and change its state, these components will properly work with each other.

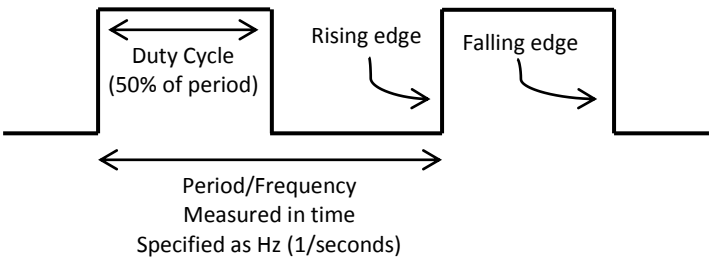


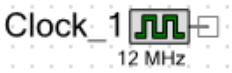
Figure 1: Visual representation of a clock signal

Ensuring that synchronous circuit designs and their clocks work properly is a complex problem, which is beyond the scope of these labs. Fortunately, PSoC Creator is designed so many of the common issues are automatically handled, and it is even designed to warn during application generation when a clock is running too fast for a design to work.

This and future labs introduce components that have many uses and options that may not be immediately used, but may be used in future labs. Information not used in a lab will be covered in the appendix section at the end of the lab.

B2: Clock Component

The clock component is used to gain access to clocks in the system, access a user-defined design-wide clock, or create a new local clock. All design-wide and local clocks take one of the system clocks and divide their frequency by an integral amount.



Clock components can be configured as follows:

- Basic Tab (Figure 2)
  - Clock type: New creates a new, local clock to be only used in this schematic. Existing will use a system clock or a user-defined design-wide clock (the name of the component will change if this is used). See the next section for how to make a design-wide clock.
  - Source: select a clock to use from here. When the clock type is New, <Auto> will choose the best clock to match the desired frequency and tolerance, but a specific clock can be chosen if desired. When the clock type is Source, this selects which clock the component represents.
  - Frequency (New type only): specifies the desired frequency of the local clock.
  - Tolerance (New type only): specifies how far the clock is allowed to be off from the specified frequency. All of the system clocks have a limited accuracy, specified as a percentage range, so that is taken into account with the tolerance parameter. Additionally, since system clock dividers are integers, it may not be possible to get the exact frequency desired. If the tolerance goal cannot be met, PSoC Creator will generate a warning. Uncheck the box if specifying the tolerance is not necessary.

Here are the most important functions in the programming interface. "Clock" is replaced with the name given to the component in the schematic view. For design-wide user clocks, "Clock" is replaced with the name given to the clock, prefixed with "Cy".

void Clock_Start()	Enables clock (clocks are enabled on startup by default)
void Clock_Stop()	Disables clock
void Clock_SetDividerValue (uint16 clkDivider)	Changes the clock's divider, thus changing its frequency on the fly. Specifying 0 will make the divider 65,536

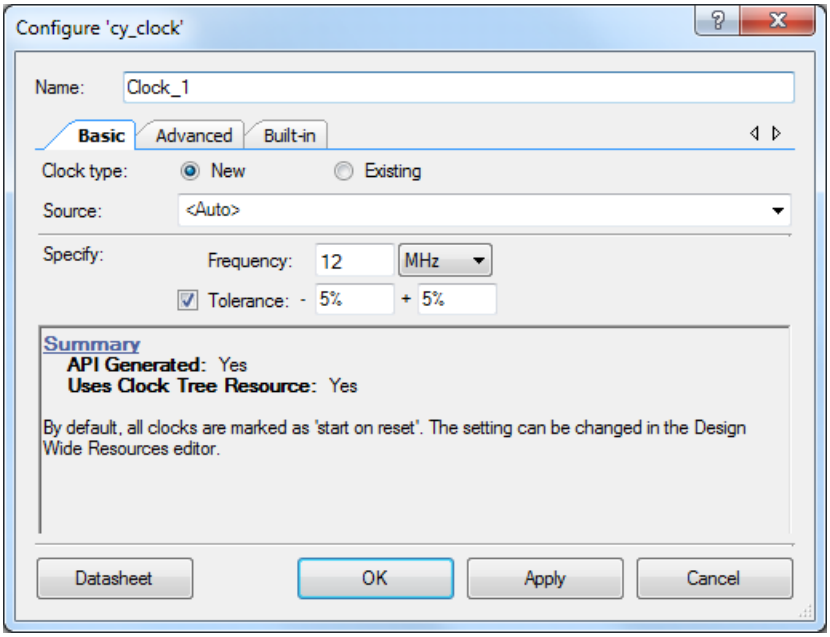


Figure 2: Clock configuration window

B3: System and Design-Wide Clocks

The design wide resources file has a Clocks tab (Figure 3) that can be used to see all of the system, design-wide, and local clocks.

Design-wide clocks are the same as local clocks and are configured the same way, except they can be used in any part of the design. To add one, go to the Clocks tab in the design-wide resources file and click the Add Design-Wide Clock button. The configuration window is the same as local clocks, except there is no clock type selection. These clocks can be deleted and edited by selecting the clock in this list and clicking the appropriate button.

*TopDesign.cysch | main.c | *Lab2.cydwr

Add Design-Wide Clock... | Delete Design-Wide Clock | Edit Clock...

Type	Name	Domain	Desired Frequency	Nominal Frequency	Accuracy (%)	Tolerance (%)	Divider	Start on Reset	Source Clock
System	USB_CLK	DIGITAL	48.000 MHz	? MHz	±0	-	1	☐	IMOx2
System	Digital Signal	DIGITAL	? MHz	? MHz	±0	-	0	☐	
System	XTAL 32kHz	DIGITAL	32.768 kHz	? MHz	±0	-	0	☐	
System	XTAL	DIGITAL	24.000 MHz	? MHz	±0	-	0	☐	
System	ILO	DIGITAL	? MHz	1.000 kHz	-50, +100	-	0	☑	
System	IMO	DIGITAL	3.000 MHz	3.000 MHz	±1	-	0	☑	
System	BUS_CLK (CPU)	DIGITAL	? MHz	24.000 MHz	±1	-	1	☑	MASTER_CLK
System	MASTER_CLK	DIGITAL	? MHz	24.000 MHz	±1	-	1	☑	PLL_OUT
System	PLL_OUT	DIGITAL	24.000 MHz	24.000 MHz	±1	-	0	☑	IMO
Design-wide	designWideClock_1	DIGITAL	50.000 kHz	50.000 kHz	±1	±5	60	☑	Auto: IMO
Local	LocalClock_1	DIGITAL	12.000 MHz	12.000 MHz	±1	±5	2	☑	Auto: MASTER_CLK

Figure 3: Clocks tab in DWR file

System clocks generate the clock signals for the entire system, including the sources for design-wide and local clocks. See the appendix for details on each one.

B4: Fixed Function vs. UDB

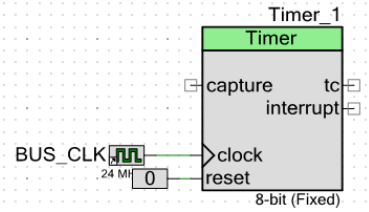
One of the reasons why the hardware of the PSoC is highly configurable is due to the Universal Digital Blocks, or UDBs, that make up a major part of its digital architecture. Each UDB contains a few parts that are often used in hardware components and a programmable logic device (PLD), which allows UDBs to be configured to become one of many components available in the PSoC library, or even a custom component defined by the developer. UDBs can also connect to each other in very flexible ways. Essentially, UDBs are similar to programmable logic blocks found in FPGAs. For more details on how UDBs work, which won't be covered in significant detail in these labs, see chapter 21: Universal Digital Blocks (UDBs) of the PSoC 5LP Architecture Technical Reference Manual (document 001-78426; can be found using Help>Documentation>PSoC Technical Reference Manuals in PSoC Creator).

There are a limited number of UDBs within a PSoC, the number of which depend on the model. The PSoC 5LP used in the development kit has the maximum quantity in the series: 24. To determine how many resources a component uses, look at the Resources section of a component's datasheet. The number of datapath cells indicates how many UDBs are used. Generally, when a component allows a resolution to be specified, each byte (rounding up if there's a fraction of a byte, e.g. 12-bit = 2 bytes) consumes one UDB. PSoC Creator will generate an error when generating the application if the design requires too many resources.

In addition to the UDBs, there are a few pieces of fixed function (FF) hardware in the architecture. This hardware is designed for a specific use, unlike the UDBs. For example, there are a few FF timers in the PSoC 5LP that can be used instead of a UDB timer. Fixed function hardware tends to have more limitations than the equivalent UDB version, but it is generally better to use FF where possible to save UDB resources, especially when making bigger projects or using PSoC models with fewer UDBs. However, if power consumption is a bigger concern, it would be best to look at the datasheets to see which implementation is more power efficient.

B5: Timer Component

The timer component is generally used to generate a periodic event. It is highly configurable and can have other uses, such as dividing a clock or measuring time between hardware events. This is also the first component that contains registers, which are 8-bit values that can be read from or written to that can configure or contain the status of a component.



Note that the appearance of the timer may change, such as having additional inputs and outputs, depending on the configuration. Timer components can be either FF or UDB. Here are the inputs and outputs for the component:

- Inputs
  - Clock: must be connected to a clock. The frequency of the clock determines how often the timer is decremented while enabled (one decrement per rising edge).

    - Reset: resets the period counter to the period value and resets the capture register. Note that this reset is synchronous, so it requires at least one clock rising edge to pass to have effect. This pin is activated with a high logic signal, or 1.
    - Enable (shown with enable mode "hardware" option): allows the hardware to control whether the timer is running. When disabled, outputs are active but the state stays the same. A low input (or 0) disables the timer.
    - Capture (may be disabled): serves as the input for the capture mode. This input serves as the trigger for the capture feature, which copies the current count to the capture register or FIFO. This input is sampled on the rising edge of the clock and is configured by the capture mode option.
- Outputs
    - Trigger (UDB only; shown with trigger mode option): when shown, this input triggers the timer to start. This input is sampled on the rising edge of the clock.
    - TC: when high, indicates that the count value is zero. Synchronized to the clock.
    - Interrupt: driven by interrupt sources specified in the configuration. Triggered interrupts remain asserted until the status register is read (see software interface).
    - Capture_out (UDB only; shown when capture is enabled): indicates when a capture has been triggered.

Interrupts will be covered in Lab 3. Documentation about interrupts related to this component are present for future reference.

Timer components can be configured as follows (Figure 4):

- Resolution: the size of the counter, period, and capture registers. Fixed function can be 8- or 16-bit and UDB can be 8-, 16-, 24-, or 32-bit.
- Implementation: use this to choose between FF or UDB.
- Period: the timer's counter decrements by one for every input clock pulse. The period determines what value the counter should start at, and the time shown on the right is how long it will take for the timer to get to zero with the current input clock and period.
- Trigger mode (UDB only): enabling trigger mode allows the timer to delay starting until a trigger occurs. This can be triggered using rising/falling/both edges in hardware or software.
- Capture mode: if set to anything other than none, an input signal can be used to capture the value of the counter at that moment to the capture register in the FF implementation or to the FIFO in the UDB implementation. The FIFO can hold four values; if the FIFO is full, captures need to be read by the CPU to avoid losing data. In fixed function, a capture always generates an interrupt while UDB has that as an option. UDB supports more capture modes, including software, while fixed function only supports none and rising edge.
    - Enable Capture Counter (UDB only): when enabled, capture does not happen until a specified number of capture signals are received.
- Enable mode: by default, software is used to enable the timer. This option allows hardware to enable the timer as well. When the timer is disabled, all inputs (except enable) are ignored and its state remains the same.

- Run mode
  - Continuous: the timer runs while enabled.
  - One shot: the timer runs for one period and stops. Resetting the timer lets it run for another period. UDB reloads the period on stop while FF stays at where it stopped.
  - One shot (Halt on Interrupt): the timer runs for one period or until it generates an interrupt and stops. Resetting the timer lets it run again from the beginning of the period. UDB reloads the period on stop while FF stays at where it stopped.
  - Note: it may be wise to use a trigger mode when using one shot to ensure the timer does not start prematurely.
- Interrupts
  - On TC: when the period expires, or when the count is 0. TC stands for Terminal Count.
  - On Capture (UDB only; FF always enabled): when a capture happens. Can specify to wait until up to four captures happen, since UDB has four capture registers.
  - On FIFO Full (UDB only): UDB has four capture registers that act as a FIFO.

Here are the most important functions in the programming interface. "Timer" is replaced with the name given to the component in the schematic view.

void Timer_Start()	Initializes and enables the timer
void Timer_Stop()	Disables the timer. If fixed function, it is also powered down.
uint8 Timer_ReadStatusRegister()	Gets the status register and clears the interrupt condition. See the status register section of the timer datasheet for the register's layout.
uint8/16/32 Timer_ReadCounter()	Captures and reads the counter of the timer.
uint8/16/32 Timer_ReadCapture()	In FF, returns the value of the capture register. In UDB, removes the last value in the FIFO and returns it.
uint8/16/32 Timer_ReadPeriod()	Returns the period, or the starting point, of the timer

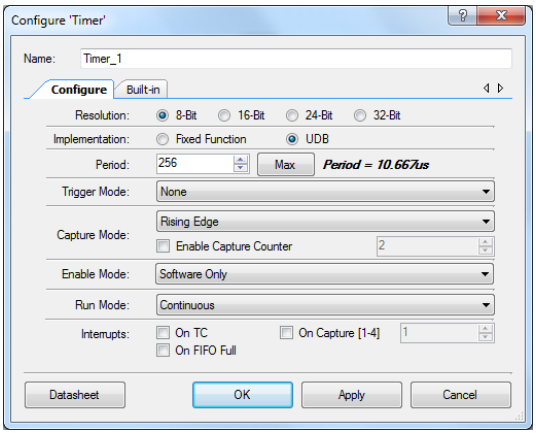


Figure 4: Timer configuration window

You may now start Procedure Part A. Return here for background for part B.

B6: Digital Logic Components

There are many logic gates that can be used to manipulate digital signals in the schematic. Each gate can be configured with terminal width (allows for one gate to represent many gates) and number of input terminals (excluding single terminal input gates like NOT). Here are the logic gates available:

- And
- Nand
- Nor
- Not (inverter)
- Or
- Xnor
- Xor

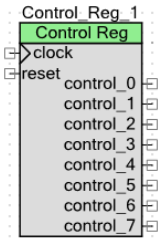
Here are a few more digital logic components: multiplexer, de-multiplexer, logic high, logic low, digital constant, lookup table. None of these components have software interfaces.

B7: Control and Status Registers

Some components do not have software interfaces and some components do not provide software equivalents to all hardware functionality. For example, the timer component does not provide a method to allow the software to reset it. Also, software cannot directly read the output of digital logic components. Registers are used as a data interface between software and components, and they can also be created as separate components for custom interfaces.

Control Register

This is a component that provides up to 8 bits of output from the software. These bits are written to by the software to be routed into the schematic.



The appearance of this component may change based on the configuration. Here are the inputs and outputs:

- Inputs
  - Clock: If at least one of the bits is not configured to be Direct, this input is used to synchronize the updates to the bits.
  - Reset: resets any non-Direct bits to 0. Hidden by default; the External reset parameter displays it.
- Outputs
  - control_0 – control_7: each bit of output. Only the number of bits configured for this register will be displayed.

Control registers can be configured as follows:

- Outputs: the number of outputs, maximum of 8.
- Display as bus: combine all outputs into one terminal.

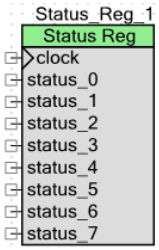
- External reset: provides a method to reset the outputs in the hardware. Can only be enabled if at least one bit is not configured to be Direct.
- Bits: each bit can be configured individually.
    - Mode
        - Direct: values written by the software apply immediately.
        - Sync: values written by the software are synchronized to the specified clock such that the value does not change until a rising edge.
        - Pulse: same as sync, except values of 1 reset back to 0 after one clock period.
    - Initial value: the register can be initially 0 or 1 on system reset.

Here is the programming interface. "ControlReg" is replaced with the name given to the component in the schematic view.

void ControlReg_Write(uint8 control)	Writes a byte to the control register
uint8 ControlReg_Read()	Reads the current value of the control register

Status Register

This is a component that can allow the software to read up to 8 digital signals from the schematic.



The appearance of this component may change based on the configuration. Here are the inputs and outputs:

- Inputs
    - Clock: used to synchronize certain pin modes. It needs to be connected even if no pins need it.
    - status_0 – status_7: each bit of input. Only the number of bits configured for this register will be displayed. If left unconnected, the input will be a constant 0.
- Output
    - intr: if the component is configured to generate interrupts, this pin appears. It should be connected to an interrupt component.

Status registers can be configured as follows:

- Inputs: the number of inputs, maximum of 8.
- Display as bus: combine all inputs into one terminal.
- Generate interrupt: allow changes to the status bits to generate an interrupt. This option can only be used if there are 7 or fewer inputs.

- Bits: each bit can be configured individually
  - Mode
    - Transparent: changes to the input immediately take effect
    - Sticky: the input is sampled on the clock. If the bit is low and the input is high when sampled, the bit goes high and remains high regardless of the input. When the software reads the register, the bit will go low again (is cleared) immediately afterwards.
  - Interrupt mask (only shown if generate interrupt is checked): when set to 1, this bit will trigger the interrupt when it goes high. Note that if multiple bits can generate an interrupt, it is best if they are sticky so the software can determine the cause.

Here are the most important functions in the programming interface. "StatusReg" is replaced with the name given to the component in the schematic view.

uint8 StatusReg_Read()	Reads the current value of the register
void StatusReg_InterruptEnable()	Enables the interrupt output, if configured to have one.

Procedure Part A

Outcome

In this lab, the PSoC will be programmed in two parts. This first part will use a fixed function timer to blink LED4 once every second.

This lab does not describe in detail how to perform tasks that were introduced in Labs 0 and 1. Review the procedures from previous labs if something was forgotten.

Part 1: Designing the System

1. Create a new project in your workspace named Lab2 and open its TopDesign.cysch.
2. Add a Ports and Pins>Digital Output Pin to the schematic area and name it LED4.
3. Add a Digital>Functions>Timer to the schematic area and name it SecondTimer.
4. Note that the timer came with a clock component. Reconfigure the clock to be a new 125 Hz clock named SecondTimer_clock.
5. Reconfigure SecondTimer's period to be 125, which makes the period one second long. Do not change any other options; it should remain fixed function and 8-bit.
6. Connect LED4 to the TC output of SecondTimer.
7. The schematic should look like Figure 5.
8. Open Lab2.cydwr and configure LED4 to use port P6[3].
9. Save all files and generate the application.

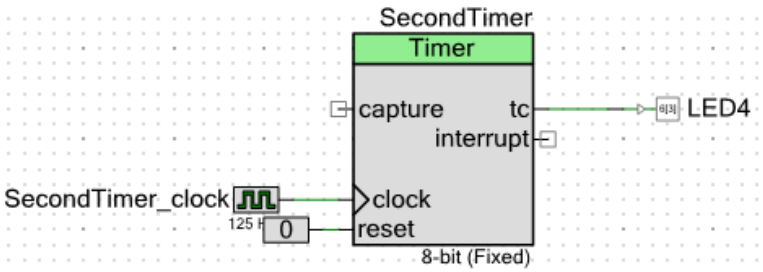


Figure 5: Part A Schematic

Part 2: Programming the Firmware

1. Because all the functionality exists in the hardware, the only thing necessary in the firmware is the hardware initialization.
2. Add the statement SecondTimer_Start(); to the initialization area above the `for` loop in main.c.

Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Test the program to make sure it works:
   - LED4 should blink every second.
   - The LED will only be on for 1/125th of a second (but still visible) because the TC output is high after the timer's counter is 0. While the counter is not zero, it is decremented once every time the 125 Hz clock goes high. When the counter is zero, the next clock tick will reload the period (125) into the counter.

Procedure Part B

Outcome

In this part of the lab, a stopwatch will be created using a UDB timer, two pushbuttons, and software to drive the display and timer reset. SW2 will start the stopwatch and SW3 will stop and reset it. The functionality in Part A will not be changed and will operate concurrently to the stopwatch.

Part 1: Designing the System

1. Reopen TopDesign.cysch.
2. Add a Display>Character LCD to the schematic area and name it Display.
3. Add a Digital>Functions>Timer to the schematic area and name it Stopwatch.
4. The stopwatch will have four decimal places of precision, so change the Stopwatch's clock to be a new 10 kHz clock named Stopwatch_clock. Increasing or decreasing the clock speed will increase or decrease the precision, respectively.
5. Configure the timer to be a UDB, 32-bit timer. Set both trigger and capture modes to rising edge. Set the period to the maximum 32-bit value using the Max button next to the period. Note that will result in the period is approximately 430,000 seconds with the 10 kHz clock, which will be the maximum time the stopwatch can run before overflowing back to 0 seconds.
6. For starting the timer, add a digital input pin named SW2_StopwatchStart configured to be resistive pull-up.

7. For stopping the timer, add a digital input pin named SW3_StopwatchStop configured to be resistive pull-up.
8. Add a Digital>Logic>Not, wire SW2_StopwatchStart to its input, and connect its output to the trigger input of Stopwatch.
9. Add a Digital>Logic>Not, wire SW3_StopwatchStop to its input, and connect its output to the capture input of Stopwatch. Note that the timer does not stop on capture; the software will handle that when it reads that a capture has happened.

Remember that resistive pull-up inputs are high when the button is not pushed and low when the button is pushed. Even though the trigger could be configured to trigger on the falling edge, the first readings on system reset may cause a false trigger for some reason. Therefore, it is better to invert the input and use rising edge triggering instead.

10. The software needs to know if the start button has been pressed. Add a Digital>Registers>Status Register to the schematic and configure it to have only one sticky bit. Name it StopwatchStart.
11. Wire the inverted SW2_StopwatchStart to the status_0 input of StopwatchStart. Add a System>Clock to the schematic, configure it to use the existing BUS_CLK (no need to sample the pushbutton any slower), and connect it to the clock input of StopwatchStart.
12. The software needs to be able to reset the timer, but there is no software interface for that. Instead, it can use a control register in pulse mode to trigger the timer to reset. Delete the constant 0 connected to the reset input of Stopwatch. Add a Digital>Registers>Control Register to the schematic and configure it to have one pulse mode bit with a 0 initial value. Name it StopwatchReset.
13. The reset on the timer component is synchronous, which means the pulse from StopwatchReset needs to be high for at least one clock cycle of the timer's clock. Connect Stopwatch_clock to the clock input of StopwatchReset so both components operate on the same clock. Connect the output control_0 to the reset input of Stopwatch.
14. The schematic should look like Figure 6.
15. Open Lab2.cydwr and map the following pins to the following ports:
    - \Display:LCDPort[6:0]\ – P2[6:0]
    - SW2_StopwatchStart – P6[1]
    - SW3_StopwatchStop – P15[5]
16. Save all files and generate the application.

Part 2: Programming the Firmware

Outcome

The Stopwatch timer in the hardware handles starting the stopwatch, keeping track of time, and capturing the moment the stop button is pressed. The software handles displaying the current state of the stopwatch, started or stopped, and the current time of the stopwatch in seconds. The software also resets the timer after reading the capture, since the timer reset also clears out the timer's captures.

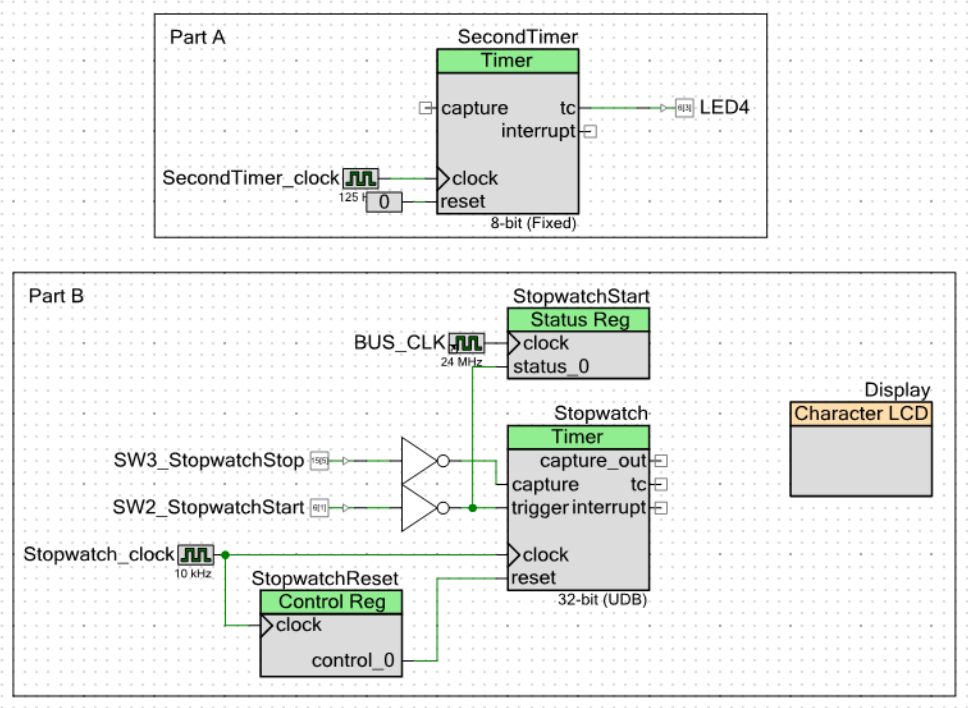


Figure 6: Final schematic

Initialization

Both timers and the character LCD need to be initialized by calling their Start functions. Additionally, the StopwatchStart status register may start high because the initial state of the SW2 input pin (its power-on reset value, which should not be changed when possible) may be set to low, so call Read to clear it.

Display something like “Stopped” on the first line of the display to indicate the stopwatch is ready to go.

Main Loop

Stopwatch stopped

If the stopwatch hasn’t been started yet, read StopwatchStart using StopwatchStart_Read() to see if the start button has been pushed. If the start button has been pushed, clear the display with Display_ClearDisplay() and display something like “Started” on the first line.

Stopwatch started

If the stopwatch has been started, see if a capture has taken place on Stopwatch. To check for this, call its ReadStatusRegister and check to see if the capture bit is set. The code should look like this:

```
if(Stopwatch_ReadStatusRegister() & Stopwatch_STATUS_CAPTURE)
```

Stopwatch_STATUS_CAPTURE is the bitmask for the capture bit in the status register, and performing a bitwise AND on the value of the register will return a non-zero value if a capture happened, or zero otherwise. This capture bit will be cleared after reading the status register, so be sure to only read the status register once per loop.

If a capture has happened, read the capture value (to be printed later) with Stopwatch_ReadCapture(), reset the timer with StopwatchReset_Write(1), read StopwatchStart to clear any presses of the start button that may have happened in the meantime, and display something like "Stopped" on the first line of the display. Resetting the timer means the stopwatch is ready to start again.

If no capture has happened, read the current value of the timer with Stopwatch_ReadCounter().

Since the timer is counting down to 0, subtract the capture or counter value from Stopwatch_ReadPeriod() to get the time passed expressed as number of clock counts.

If the count is not zero, print the number of seconds passed to the second line. There are two methods of doing this:

1. Do a floating-point division of the count by the frequency of Stopwatch_clock: 10,000. Use sprintf to convert the value to a string, as demonstrated below, and print the string to the second line of the display.

```
sprintf(tstr, "%1.4f", seconds); //tstr is a char[16]
```

Starting in PSoC Creator 3.0, the software is compiled with the standard C library implementation called newlib-nano, which is made to be as small as possible for microcontrollers. Because of this, it does not include the code to handle printing or scanning floating point values by default. To re-include support, insert the necessary code below above the main function.

```
asm (".global _printf_float");  //adds ~8000 bytes to the program
asm (".global _scanf_float");   //adds ~9800 bytes to the program
```

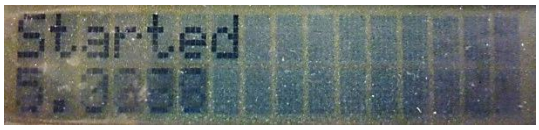
Keep in mind that the PSoC 5LP used in the development kit has only 256kb of flash.

2. Since the frequency of Stopwatch_clock is a power of 10, use integer division to obtain the number of seconds and modulo to obtain the number of ten thousandths of a second. Print the number of seconds, a period, zeroes for padding if necessary, and the number of ten thousandths of a second to the second line of the screen.

Part 3: Running the Project

1. Build and load the project onto the PSoC development kit.
2. Test the program to make sure it works:
   - Starting out, the display should only say Stopped, or equivalent. LED4 should still be blinking once per second.

   - Press SW2. The display should say Started, or equivalent, and should be continually updated with the elapsed time. Ensure that time appears to start at zero.



- Press SW3. The display should say Stopped and show a time close to what was displayed right before pressing SW3.

- Press SW2. The display should say Started and should continually show the elapsed time, starting back over at zero. Ensure there are no excess digits left over from the last run; if there are, the display wasn't cleared before starting.

3. Note that while the stopwatch is running, pressing SW2 has no effect. Pressing SW3 should stop the stopwatch as usual (if it doesn't, then it isn't being read before stopping the stopwatch). However, if SW2 is held while SW3 is pressed, the software will think the stopwatch has stopped and immediately started over again, even though it hasn't since the timer's trigger is set to operate on rising edge. This happens because the StopwatchStart register is high while (and after, until it is read) the input button is pressed, not only when the press initially happens. Lab 3 will introduce new concepts that can fix this problem.

Appendix

A1: System Clocks

These are the system clocks found in the PSoC 5LP:

- MASTER_CLK – This is the master clock, which is actually configured to be one of the other clocks (specifically, Digital Signal, IMO, PLL_OUT, or XTAL). It is used to synchronize many things in the system and therefore must be the fastest clock, excluding the USB clock. Reducing the master clock speed can reduce power consumption, but will also reduce the speed of the system. By default MASTER_CLK is set to PLL_OUT.
- BUS_CLK – This is the speed of the CPU. It may only be configured by dividing MASTER_CLK by an integer, which is 1 by default. Peripherals (hardware in the schematic view) may not operate on clocks faster than BUS_CLK. Additionally, the CPU cannot run faster than 67 MHz.
- IMO – Internal Main Oscillator. This is the high frequency clock in the system. By default it is set to 3 MHz, but it can also run at 6, 12, 24, 48, and 62. This clock has higher accuracy at lower frequencies. IMO also includes a doubler that can be used as an input to USB_CLK. It cannot be disabled, but its oscillator can be replaced with XTAL or Digital Signal if higher accuracy is desired.
- PLL_OUT – This is the output of the PLL (Phased-Locked Loop), which is generally used to take a clock (Digital Signal, IMO, or XTAL) and produce a higher frequency (it can also produce a lower frequency, which is a less common application). Generating the higher frequency introduces no inaccuracies and equivalent outputs consume less power than the IMO, so it is best to run IMO at 3 MHz and use the PLL to generate a higher frequency. The input frequency must be between 1 and 48 MHz and the output must be between 24 and 66

MHz. The PLL may be disabled to save power. By default the input is IMO and the output is 24 MHz.

- ILO – Internal Low-speed Oscillator. A low speed, low power clock used for periodic system operations, such as waking up from sleep to perform checks in a low power system. It can generate three frequencies: 1 kHz (default), 33 kHz, and 100 kHz. This is a very inaccurate clock, so it should not be used for precise timing. It may be disabled if XTAL 32 kHz is enabled.
- XTAL – external crystal clock for a higher accuracy clock than IMO. The input crystal must be 4-25 MHz. This clock is disabled by default. The development kit includes a 24 MHz crystal if the IMO's accuracy is inadequate.
- XTAL 32 kHz – external 32.768 kHz crystal clock. This clock is used for accurate timekeeping using the real-time clock (RTC) built into the PSoC or for generating one second interrupts. It is disabled by default. The development kit includes one of these.
- Digital Signal – allows a digital signal to be used as a clock. Disabled by default.
- USB_CLK – If the project will use the PSoC as a USB device, this must be configured to use another clock running at 48 MHz. It is disabled by default.

All system clocks are configured in a window (Figure 7) that is accessed by selecting one of them in the design-wide resources file and clicking the edit button. More information about system clocks can be found in the Cypress publication **AN60631** and more information about using external oscillators can be found in **AN54439**.

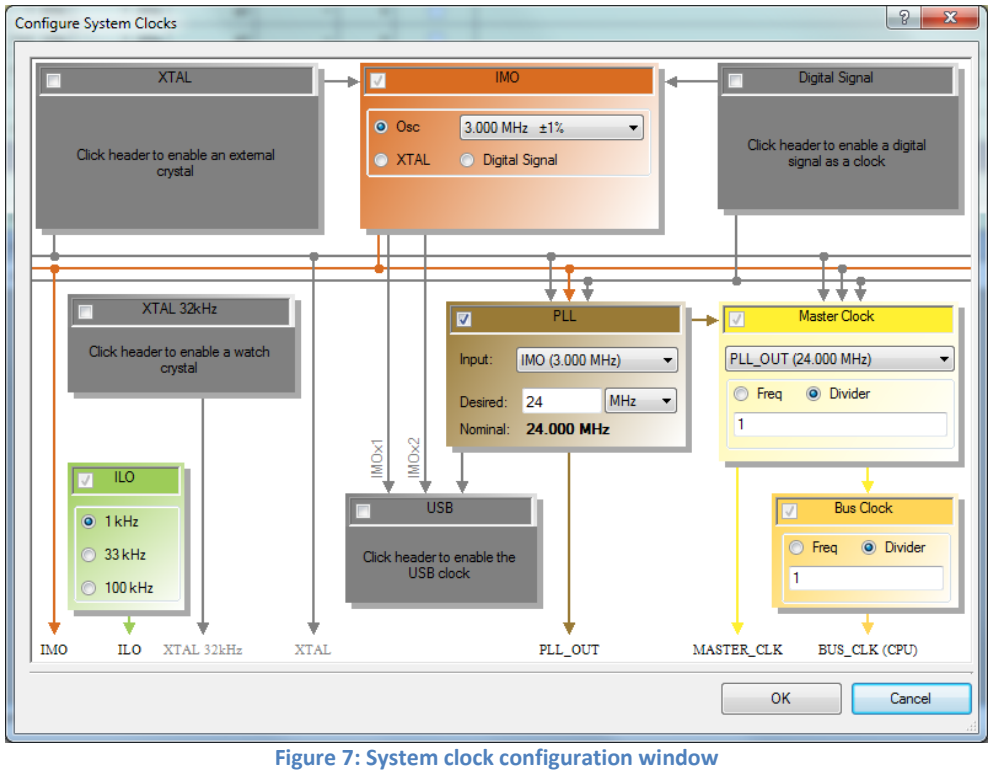


Figure 7: System clock configuration window