Department of Electrical and Computer Systems Engineering

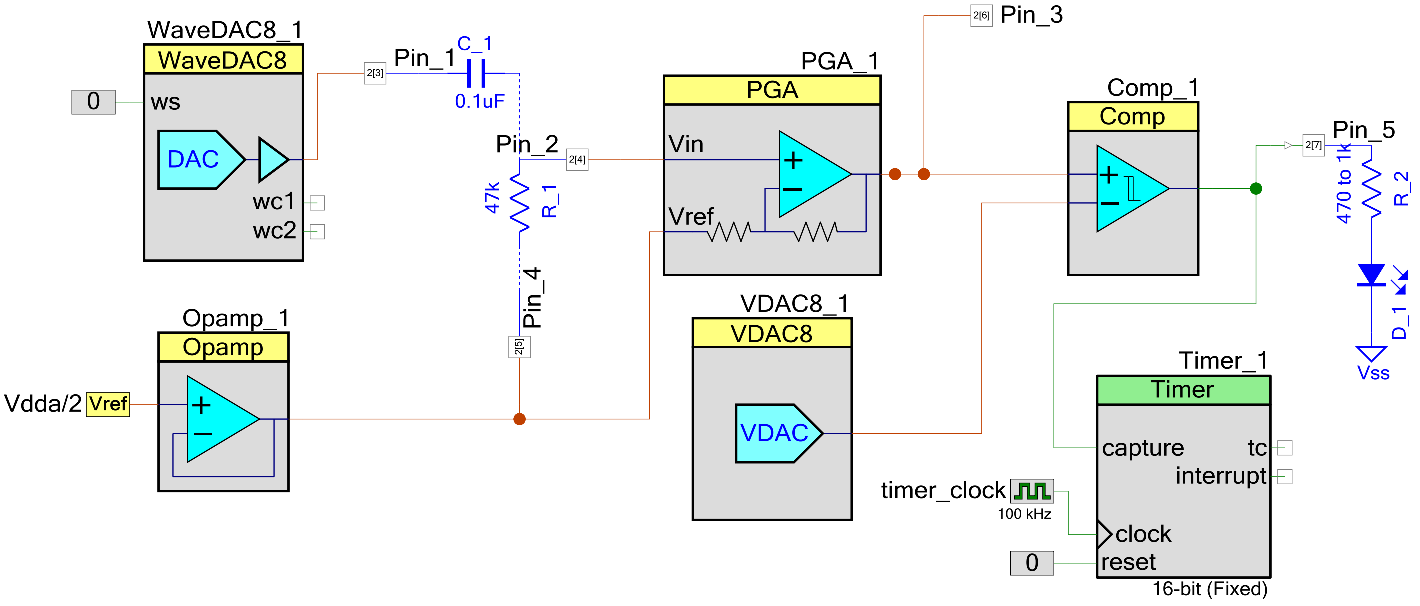
Monash University

TRC3500 Sensors and Artificial Perception

The Ultrasonic Distance Meter: Help

Introduction

The ultrasonic module works at 40kHz which is quite fast. As we may not have an oscilloscope to see the waveforms, we need to think outside the box and come up with a possible solution using what we have. Shown below is a possible test circuit you can create to experiment with. Once you have this working, you can replace the WaveDAC8 and the 0.1uF capacitor with the Rx module.



What we know

The Rx module when receiving sound will be creating a sine wave with a frequency of 40kHz. The amplitude of this signal will vary depending on the strength of the returned sound pressure. If the sound is loud, the wave form will be large. When I say large we are probably talking about 2V AC peak to peak. Unfortunately this unit can be difficult to do testing with because of the high frequency and lack of a digital storage oscilloscope (DSO).

Simulating the Rx module

The first step we might take is to remove the ultrasonic Rx module form the circuit and simulate the signal with some electronic components. In this case I have used a WaveDac8 that creates a sine wave of 10 Hz with a maximum amplitude of 1.02V. This is connected to the input of the amplifier via a 0.1uF capacitor. This is our simulated received signal coming from the ultrasonic Rx module and because the frequency is so low, we can connect a LED to the circuitry to see if we are getting a returned signal. It will flash at a rate of 10Hz which we can see.

Setting the gain

Now as we know the incoming signal from our simulated signal generator ranges from 0 to 1.2V in amplitude, we will set the gain of the amplifier to 4. This will provide a signal of about 0 to 4.8V to work with. If we had our Rx module connected and the returned signal was 50mV we would probably set the gain to the maximum of 50 giving us 2.5V.

Threshold circuitry

The threshold circuitry is made up of the VDAC8 and comparator and is used to stop the timer when a returned signal is detected. Under normal conditions with the ultrasonic module is connected, the output of the Rx module will be producing some signal even when the transmitter is not operating and this is because of ambient noise in the vicinity of the module as well as electrical noise. This signal can cause false tripping and incorrect readings which we do not want. Looking at the comparator now, whenever the positive terminal is greater in voltage than the negative terminal the output will go high and disable the timer. The negative input is set by the VDAC which allows us to control when this triggering occurs. And so if there was some noise of around 100mV arriving on the positive input pin, we would set the output of the VDAC to 150mV thus blocking that signal. As soon as a signal of greater than 150mV arrives on the positive input, the output of the comparator would then go high.

Measuring time of flight

The time of flight circuitry is made up of Timer1. Under normal conditions with all of the electronics working correctly, your distance measuring device may work in this way:

a) Start the hardware module that transmits your sound pulses.(Not shown above)
b) As soon as you have started your transmission, you start your timer (Timer_1)
c) When the pulses arrive back they disable the timer and cause an interrupt.
d) You read timer count which indicates how long the sound took to travel and return.

Things to consider

The combination of the amplifier, the comparator and the VDAC allow you to set the unit to operate under different conditions. Large objects will return a better signal than a small object. Hard objects tend to reflect more of the wave rather than soft objects. The closer you are to the object, the stronger the returned signal will be.

Having the amplifier set too high may cause noise to trip the electronics and stop the counter. Having the amplifier set too low may cause the unit not to trip the counter at all. The end result then is a balancing act between the signal, the gain set on the amplifier and the trigger level threshold. You should practice setting these parameters in accordance with your projects goal which is about a 300mm distance.

Simulation one

So we want to test our circuit above at some basic level. Set the modules as follows:

1. WaveDAC to 10Hz with an output of 1.2V.
2. Set PGA gain to 50. Power level = high
3. Set VDAC output to 3500mV = 3.5V

We will not use the timer or the ISR here. They are a hint which you may wish to implement in your circuit.

Don’t forget to turn on the modules in software.

```
#include "project.h"

int main(void)
{
CyGlobalIntEnable; /* Enable global interrupts. */
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    WaveDAC8_1_Start();
    Opamp_1_Start();
    PGA_1_Start();
    Comp_1_Start();
    VDAC8_1_Start();
    for(;;)
    {
        /* Place your application code here. */
    }
}
```

Compile, download and run the circuit. The LED should be flashing at 10Hz as the amplified sine wave will be larger than the 3.5V trigger level.

Simulation two

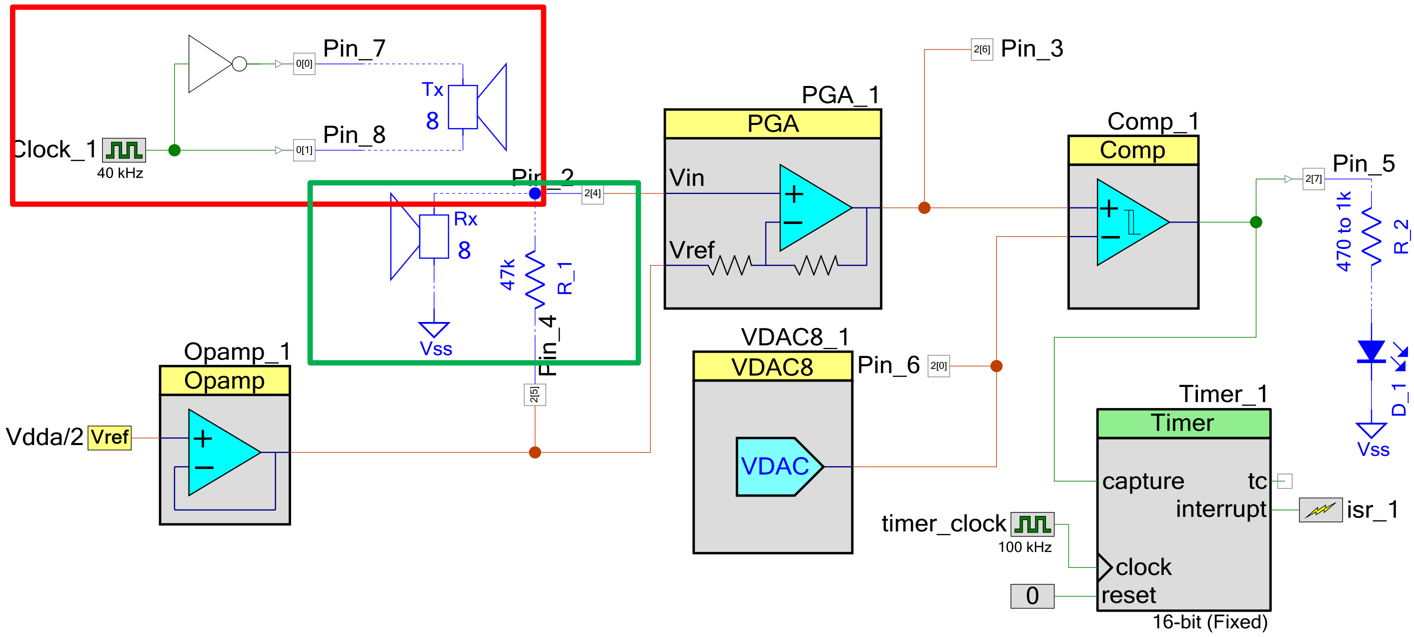
Now stop the program and reduce the gain of the amplifier to 1. Compile and down load it again. The LED should not flash because the output from the amplifier does not go above 3.5V

Simulation three

Stop the program and adjust the gain again to a slightly higher level say 2. Compile and down load it again to see if the LED starts flashing. If not keep repeating the process of increasing the gain until the LED starts to flash. Once you have completed this preliminary test, it’s time to try the ultrasonic module out.

Create the circuit

Remove the WaveDAC and the 0.1uF capacitor from the circuit as they are no longer needed. We will create a transmitter circuit (Red) that runs continuously to help us to debug the system. Include a clock, an inverter and two pins. These will drive the ultrasonic Tx module. Connect the Rx module as shown in green.



Note: I have shown you how to make a voltmeter circuit in another module which could be used in this circuit to help with debugging. This volt meter displayed its readings in the console window. This could help in determining certain pins voltages:

a) Pin_4 = 2.5V
b) Pin_6 = 3.5V (Depends on what you set the VDAC to)
c) more…

So we want to test our circuit above. Set the modules as follows:

a) Set PGA gain to 32. Power level = high
b) Set VDAC output to 3500mV = 3.5V

Copy this code into your circuit.

```
#include "project.h"
int main(void)
{
CyGlobalIntEnable; /* Enable global interrupts. */
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */
    Opamp_1_Start();
    PGA_1_Start();
    Comp_1_Start();
    VDAC8_1_Start();
    for(;;)
    {
        /* Place your application code here. */
    }
}
```

Recompile, down load and run.

Remove any objects in front of the Rx module. When I was testing this circuit, objects up to two meters away were being detected!

If the LED is off, place an object in front of the unit. The LED should come on and will probably be oscillating at 40kHz. You will not see this flash rate because it is too high for us to perceive but the LED will be well lit.

If the LED is on, place your finger over the Rx module. The LED should go off.

Continue experimenting with this circuit until you have a good feel for how it all works. Also note the distances you are able to achieve with the ultrasonics and especially try it with the aluminium angle you were given.

7 segment display help

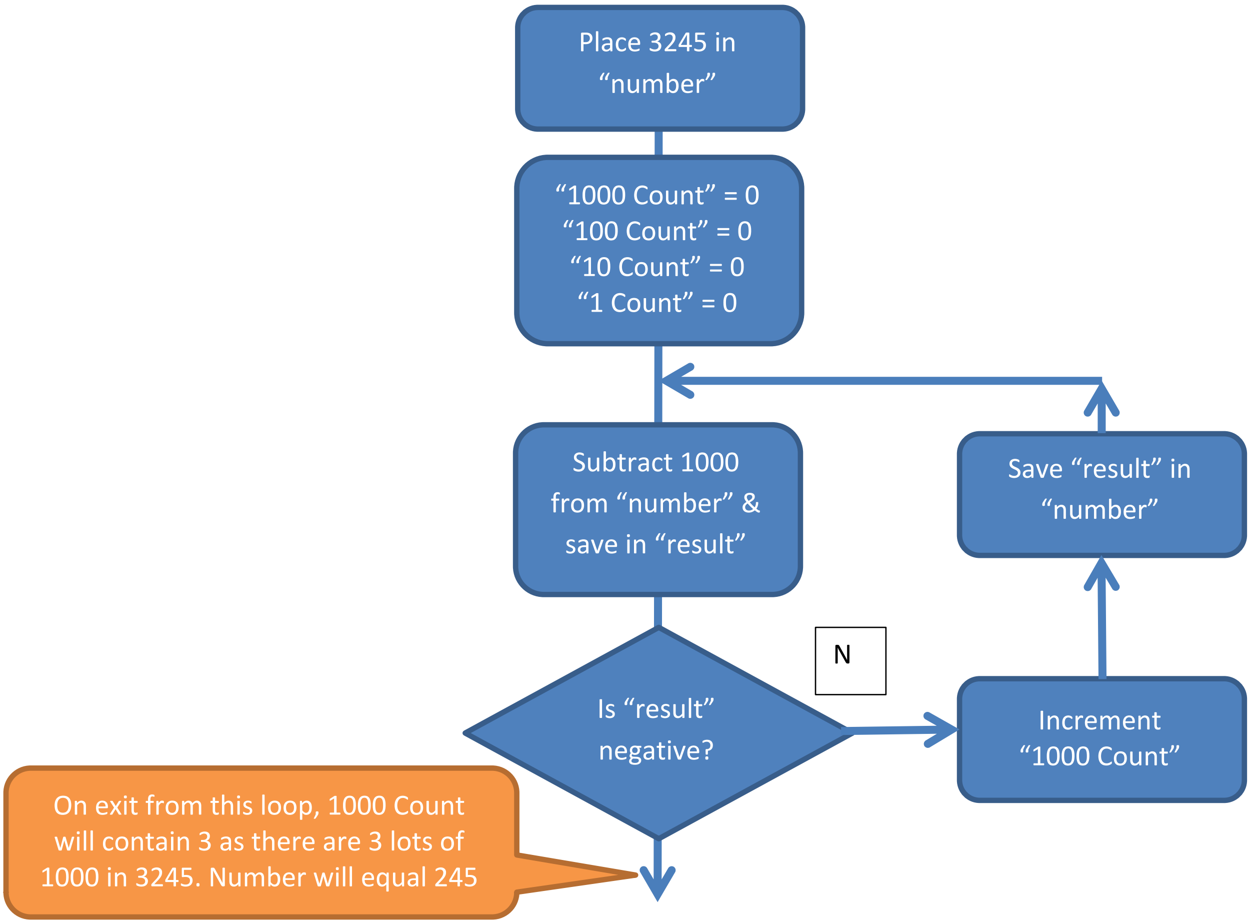
The question often arises on how we turn a number stored in a memory location into information that we can display on our seven segment display. For example, if we take a four digit number such as 3245, we would like the first display to show “3”, the second display to show “2” the third display to show “4” and the fourth display to show “5”. The first thing we can see is that the number is comprised of units, tens, hundreds and thousands.

The number 3245 broken down	
3 *1000	3000
2 * 100	200
4 * 10	40
5 *1	5
	Sum =3245

Above: The number 3245 broken down into its components

The next step is to take our number 3245 and start to dissect it by calculating how many thousands there are, then how many hundreds, then how many tens and finally we are left with the units. So to start the process, we subtract one thousand off and then check to see if we have gone below 0 or negative. If not then we take another 1000 off and so on. Here is a flow chart to make it clearer but first we declare some variables and these are “number”, “1000 Count”, “100 Count”, “10 Count” and “1 Count”.

A flow chart with an algorithm for this is on the following page.


Place 3245 in "number"
"1000 Count" = 0
"100 Count" = 0
"10 Count" = 0
"1 Count" = 0
Subtract 1000 from "number" & save in "result"
Is "result" negative?
N
Increment "1000 Count"
Save "result" in "number"
On exit from this loop, 1000 Count will contain 3 as there are 3 lots of 1000 in 3245. Number will equal 245

Above: Flow chart determining how many 1000's are in the number 3245

At this point "number" will contain the remainder 245 so you now go onto determining how many 100's are in the remainder, then 10's and so on.

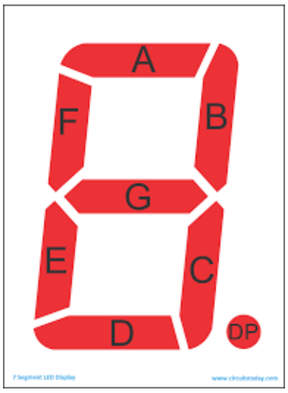
At the end of this your four variables will contain:

Variable and their values	
"1000Count"	3
"100 Count"	2
"10 Count"	4
"1 Count"	5

Above: Shows the results stored in RAM locations

Now we have our digits 3, 2, 4 and 5 we wish to display, however there is another issue. You cannot just use these as they are, they have to be converted. For this we need to know our port pin assignment and this is shown below

Port pin and segment alignment	
Segment “A”	P2.0
Segment “B”	P2.1
Segment “C”	P2.2
Segment “D"	P2.3
Segment “E”	P2.4
Segment “F”	P2.5
Segment “G”	P2.6
Segment “DP”	P2.7



Above: Segment and port pin assignment

Looking at the table below we can see in the left hand column the number “0” we wish to encode (Step 1). The next step (Step 2) is to determine the segments to use and in this case it will be F E D C B and A. Now we create a binary number by putting a “1” for every segment we use (Step 3). Because the segments are active low, that is they turn on when we put a logic low out on the port pin, we need to invert the bits (Step 4). Then we create the hex number which is C0 (Step 5). This is what we can send directly to the port. I have decoded two of them, you will have to do the rest.

Numbers and the conversion process (2 examples shown)				
Step 1	Step 2	Step 3	Step 4	Step 5
Number to display	Segments used	Binary number	Inverted because segments are active low	Hex
0	F, E, D, C, B, A	0011 1111	1100 0000	C0
1				
2	G, E, D, B, A	0101 1011	1010 0100	A4
3				
4				
5				
6				
7				
8				
9				

Above: This table shows the conversion process.

An important point to keep in mind is that the updating of the display will usually happen during an interrupt routine and we want to spend as little time as possible in that. So the best idea is that whenever a new number arrives, we do all of the processing once to create our data and save that into four (4) display variables. So the steps are:

- A new number arrives
- Number is broken up into 1's, 10's, 100's and 1000's
- Hex segment data is located and saved in four (4) ram locations such as

RAM location	Data in RAM location
Disp1Data	C0
Disp2Data	A4
Disp3Data	C0
Disp4Data	A4

Above: The raw hex data to be sent out to the port. This would display "0202" on the four digit display.

Then each time an interrupt occurs, the hex data is read from the RAM and sent straight out with no time spent processing it.

Handling slow external devices

Some peripherals such as a buzzer may need to be operated for long periods of time of around 500mS. When we say long, when compared to the speed of the processor, 500 mS is a life time! We therefore need to be careful in how we interact with these external devices so that the processor does not spend large amounts of time being idle while it waits for external devices to finish. The PSOC is unusual in the sense that we could apply a whole logic block to operating the buzzer and use all most no processor time. However, the vast majority of processors are not like this and it is therefore important we know how to handle this situation in other ways.

Let's look at three different ways in which we can handle this problem without a dedicated hardware block.

Solution one

The first solution involves a very basic concept of turning the buzzer on, waiting 500mS and then turning the buzzer off. This is seldom done for the simple reason that while the processor is in some timing loop, it cannot do anything else. We would term this as "blocking code".

Solution two

The second solution involves some form of event timer. Almost all processors have at least one hardware timer and many processors have three or four. In this solution, we can turn the buzzer on, load a 500mS delay into the countdown timer and then continue on with other work. When the timer reaches 0, an interrupt will occur and the processor can then go and turn the buzzer off. A much better solution but what if there are several things happening at once, all of which need one timer each?

Solution three

The third solution would probably be one of the more common types and is especially useful when you only have one hardware timer in the microcontroller. To help explain this, let’s describe a very typical microcontroller board. Our microcontroller board has a keyboard, a buzzer and a flashing LED. Also it has a requirement to record the length of events and so we will create a variable called “milli” that will be incremented every 1mS. It should be noted that this particular scenario is suitable for a fast processors.

No	Item	“RAM” Name	Timing	Description

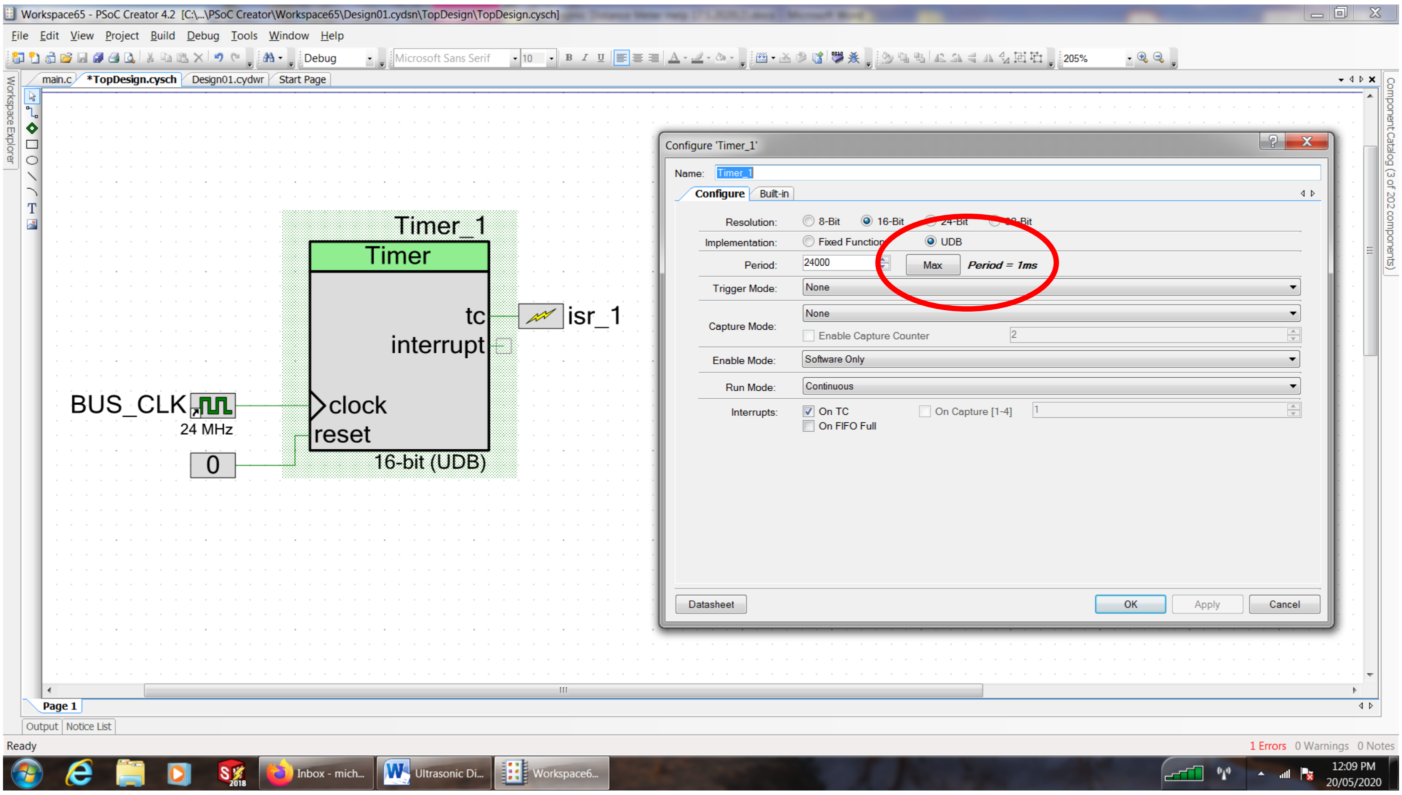
1	LED	“LED 1”	On = 1 sec
			Off = 1 sec
			Runs forever.
				This is a LED that flashes at 0.5Hz. It is sort of the heartbeat of the system and lets the user know its running.

2	Buzzer	“Buzzer”	Min 5mS Max = 3 seconds	The buzzer will operate for multiples of 5mS when required. Allow 150mS
3	Keyboard	“KeyBrd”	Min 5mS Allow 150mS	Keyboard delay of min 5mS. Helps with contact bounce when a key is hit.
4	1mS timer	“milli”	Incremented every 1mS.	32 bit RAM location. Rolls over about every 49.7 days. Can be used to measure periods of time.
There could be many more of these delays but we will stick to 4				

Above: Table describing timing constraints

In this system we will create an interrupt that occurs on a regular basis that will halt the current execution of the processor and divert it to an interrupt service routine. This routine will handle all of our timing needs.

To accomplish this, we will use a hardware timer to continually count down and cause an interrupt when it reaches 0. How often should that be? Well looking at the times in the table above, our shortest time is 1mS so we would like to make a hardware interrupt happen every 1mS.


Workspace65 - PSoC Creator 4.2 [C:\...\PSoC Creator\Workspace65\Design01.cydsn\TopDesign\TopDesign.cysch]
File Edit View Project Build Debug Tools Window Help
main.c
*TopDesign.cysch
Design01.cydwr
Start Page
Timer_1
Timer
tc
interrupt
clock
reset
16-bit (UDB)
isr_1
BUS_CLK
24 MHz
0
Page 1
Output
Notice List
Ready
1 Errors 0 Warnings 0 Notes
Datasheet
OK
Apply
Cancel
12:09 PM
20/05/2020

Configure 'Timer_1'

Name: Timer_1

Configure | Built-in

Resolution: ○ 8-Bit ● 16-Bit ○ 24-Bit ○ 32-Bit

Implementation: ○ Fixed Function ● UDB

Period: 24000 Max *Period = 1ms*

Trigger Mode: None

Capture Mode: None

□ Enable Capture Counter 2

Enable Mode: Software Only

Run Mode: Continuous

Interrupts: ☑ On TC □ On Capture [1-4] 1

□ On FIFO Full

Let's declare a variable to help with our timing needs. The count variables are used to determine when things should happen. We only have one of these in this design and this is the "5mS Count".

At start-up we will assume the variable "5mS Count" is loaded with 5. This only ever happens once.

Looking at the flow chart now, we can see that when the interrupt happens, the first thing it does is increment the 32 bit variable "milli". This variable can be read during normal program execution and can help determine time periods. For instance, if you wish to take a time measurement, the formulae is:

Time in mS = Last reading of "milli" – first reading of "milli". As this timer is 32 bits, it can measure events that run for nearly 50 days with mS accuracy.

Next it decrements "5mS Count" and tests to see if 5mS has elapsed. In this case it hasn't because it's just been decremented to 4 so it does not decrement any other variables and therefore exits the interrupt.

As the interrupts keep on occurring, the "5mS Count" will count down to 0. When this happens, the "5mS Count" is reloaded with 5, "KeyBrd", "Buzzer" and "LED 1" are decremented.

Now some tests are made.

- "KeyBrd" is not checked here and so no action is taken. It will be handled externally.
- If "Buzzer" = 0, we turn off the buzzer.
- If "LED 1" = 0, we toggle the LED and reload this counter with 200 which is 200 x 5mS = 1 second.

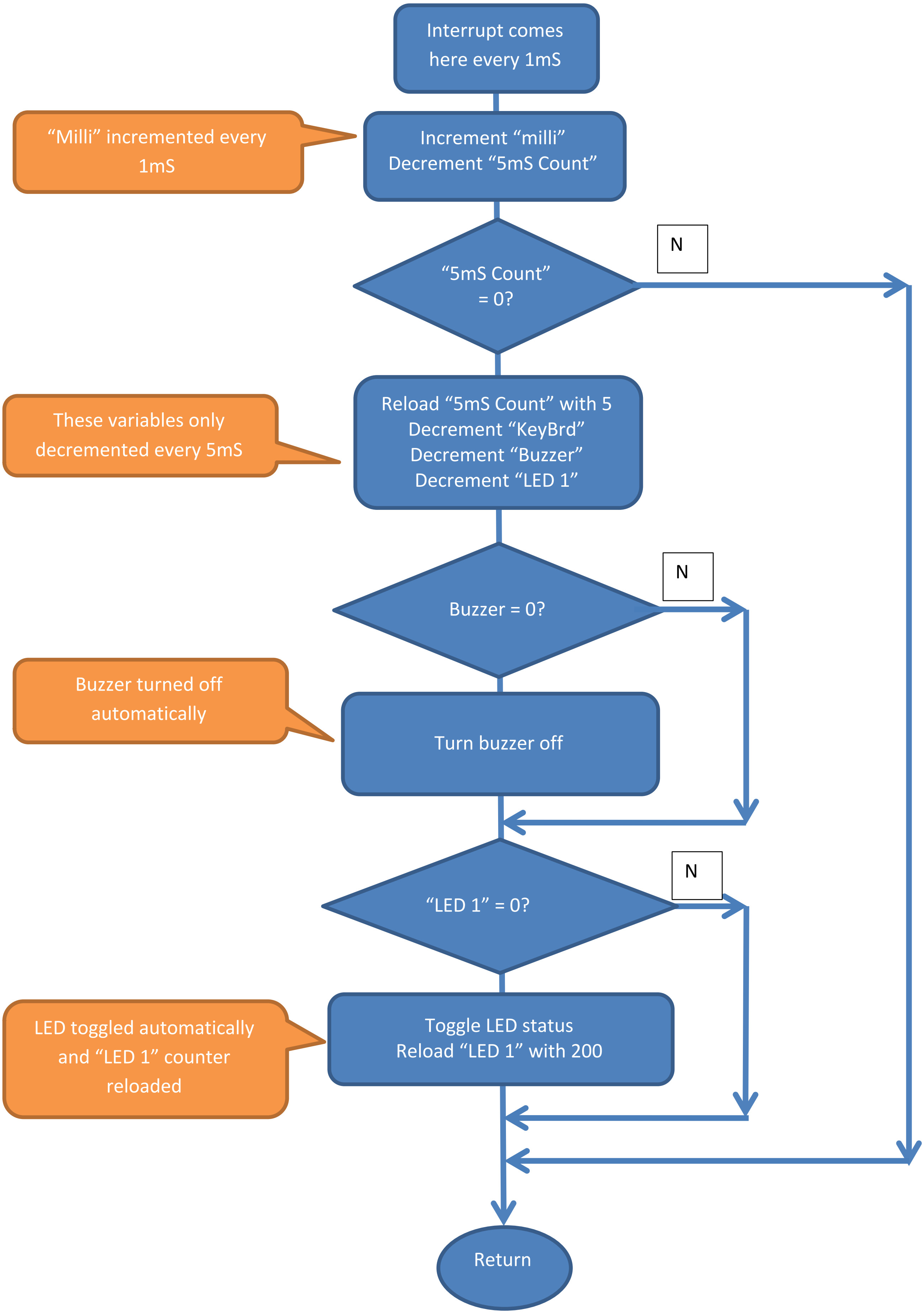
Next it returns from the interrupt.

So let's note a few important points here:

1. Interrupt happens every 1mS
2. "Milli" is incremented every 1mS
3. Interrupt must happen 5 times before any further action is taken

4. After 5 interrupts, “KeyBrd”, “Buzzer” and “LED 1” are decremented.
5. “Buzzer” and “LED 1” tested for 0 and then acted upon.

See also the flow chart on the following page.


Interrupt comes here every 1mS
"Milli" incremented every 1mS
Increment "milli" Decrement "5mS Count"
"5mS Count" = 0?
N
These variables only decremented every 5mS
Reload "5mS Count" with 5 Decrement "KeyBrd" Decrement "Buzzer" Decrement "LED 1"
Buzzer = 0?
N
Buzzer turned off automatically
Turn buzzer off
"LED 1" = 0?
N
LED toggled automatically and "LED 1" counter reloaded
Toggle LED status Reload "LED 1" with 200
Return

An example

To sound the buzzer for 500mS, our code in the main program would look something like this:

```
“Buzzer” = 100;               // Load “Buzzer” variable with 100. 100 x 5mS = 500mS
PortPin_Write(0);      // Turn physical buzzer on
```

And that’s it. We don’t have to worry about it again. The interrupt routine will switch the buzzer off when 500mS has elapsed.

Note: You must load the timer first (“Buzzer” = 100) and then turn on the buzzer. This is because if we turn the buzzer on and an interrupt happens before we can place 100 into “Buzzer”, the buzzer may be turned off straight away.

Writing robust flexible code

You could create a function for buzzer that can accept different lengths of buzz time.

So to turn the buzzer on for 1 second which is (200 * 5mS)

```
SoundBuzzer(200);             //This is our standard function call to operate buzzer

//Buzzer function
//Receives integer
//
SoundBuzzer(int x)
{
Buzzer = x;            // Load “Buzzer” variable with x.
PortPin_Write(0);      // Turn physical buzzer on
}
```

What if you forget to send it a value, buzzer could be on for weeks!

```
//Buzzer function
//Receives integer
//Checks to see buzzer run time is not greater than 3 seconds
//
SoundBuzzer(int x)
{
If(x>600)                     //Check that on time is no greater than 3 seconds (600 * 5mS)
{
x=50;
}
Buzzer = x;            // Load “Buzzer” variable with x.
PortPin_Write(0);      // Turn physical buzzer on
}
```

Some important points:

- It cannot be guaranteed that each interrupt sequence will be the same length.
- This is because sometimes we enter the 5mS section and sometimes we don’t.
- Because the interrupt happens every 1mS, there is some wasted time as with each interrupt call, the processor has to save its registers, enter and then exit the interrupt routine and then restore its registers. Slow processors may have trouble with this interrupt speed.
- This is a small example, a big system may have an extensive number of these counters.

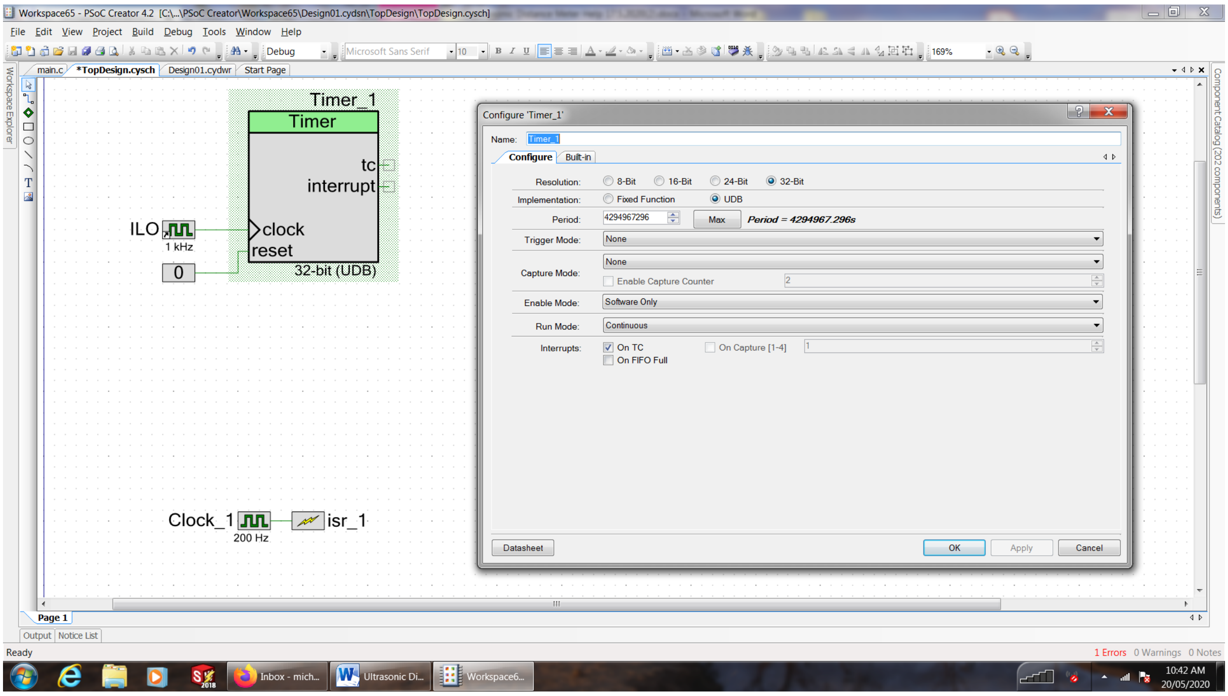
- You can determine the total time spent in side the interrupt routine by setting a port pin high at the start of the interrupt and then setting it low at the end. Using your DSO you can determine in microseconds how long this routine is.
- You may notice that in the program, every so often the buzzer will be turned off even if it was off already. It won't affect the operation. This is the nature of the design.

An improved example

With our previous example, every 1mS the processor would receive an interrupted to attend to some other duties. This is 1000 times per second and results in a lot of wasted time and for the vast majority of interrupts, the only thing that happens is that the variable "milli" is incremented. If we didn't need the "milli" count, we could probably drop the interrupts back to once every 5mS or 200 times per second. This would still satisfy all of the requirements.

Shown below is another possible example but this time we have used an external 32 bit counter that is used in place of the 32 bit "milli" variable. This timer is decremented every mS and can be read by its application programming interfaces (API). Our interrupt will come from Clock_1 which will create an interrupt every 5mS or 200 times per second.

Remember, the engineer when faced with a problem needs to innovate, adapt, adopt or destruct!



Workspace65 - PSoC Creator 4.2 [C:\...\PSoC Creator\Workspace65\Design01.cydsn\TopDesign\TopDesign.cysch]

Configure 'Timer_1'

Name: Timer_1

Configure | Built-in

Resolution: ○ 8-Bit ○ 16-Bit ○ 24-Bit ● 32-Bit

Implementation: ○ Fixed Function ● UDB

Period: 4294967296 Max *Period = 4294967.296s*

Trigger Mode: None

Capture Mode: None

□ Enable Capture Counter 2

Enable Mode: Software Only

Run Mode: Continuous

Interrupts: ☑ On TC □ On Capture [1-4] 1

□ On FIFO Full