EEE 20003: Embedded Microcontrollers (Assignment 2)

Description of the proposed use case with necessary illustration:

Nowadays in many factories, especially the production manufacturing line or the water treatment tank, where there are containers used to collect the water or any liquid substances at the end part of the machine, as shown below:

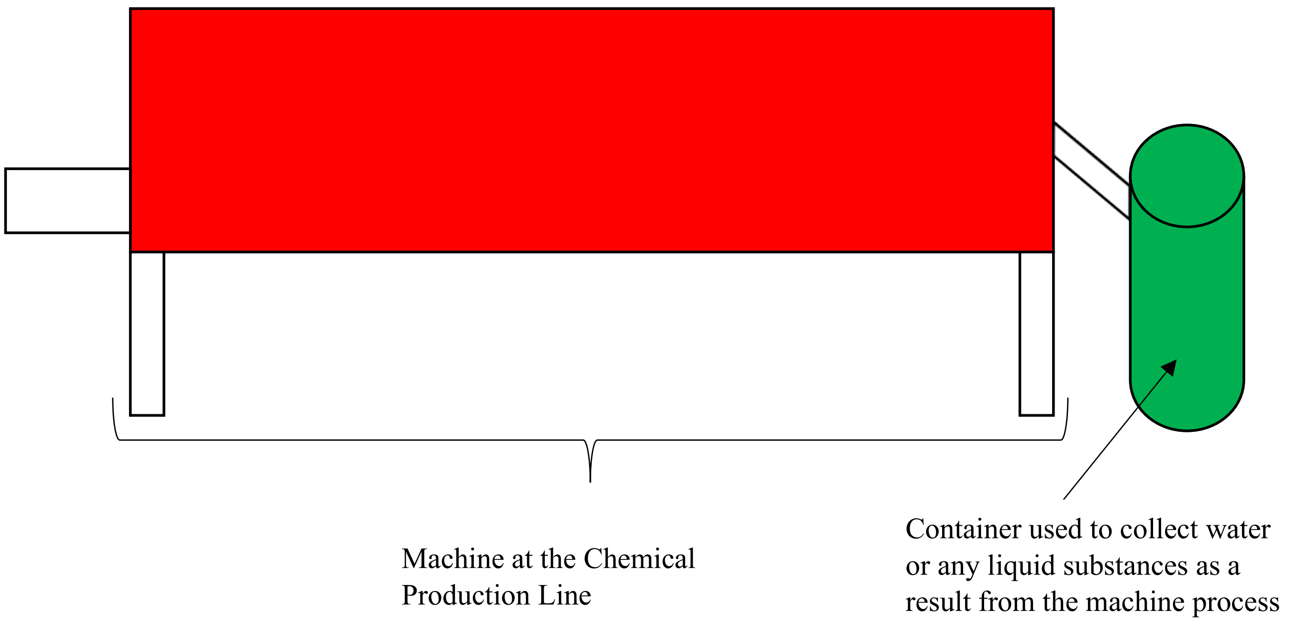


Figure 1: Example of the Machine at the Production Line with the Container at the end of the production line

In some of the chemical factories, where factory operators will take turn standing at the end of the production line, just for the purpose to wait for the container to reach almost full, then they only send the filled containers to other section of the production line. However, this is not an effective approach to monitor the amount of substance inside the container, as this approach utilises unnecessary manpower as the factory operator waste much time waiting in front of the container and doing nothing during that period of time.

From the situation above, a better approach to tackle it is to implement a container level sensing system with the use of LCD screen. This approach will keep track, record and display the real-time container level which collects the water or any liquid substances at the end of the machine line. The system will send some warning signals, such as blinking of lights or producing buzzer siren to aware the operators that the water level inside the container is almost full. In this case, the warning level of the system is kept at 70%, so that it provides sufficient time to let the operators to react and handle the situation.

The diagram below shows the proposed use case of the application, which implements ultrasonic sensor to measure the percentage that the container is filled. The ultrasonic sensor is fixed at a certain height above the container so that the measurement taken is accurate and consistent throughout the process. The reading taken and measured by the sensor is sent to the LCD screen to display the percentage of the "container fullness" and the status of the percentage.

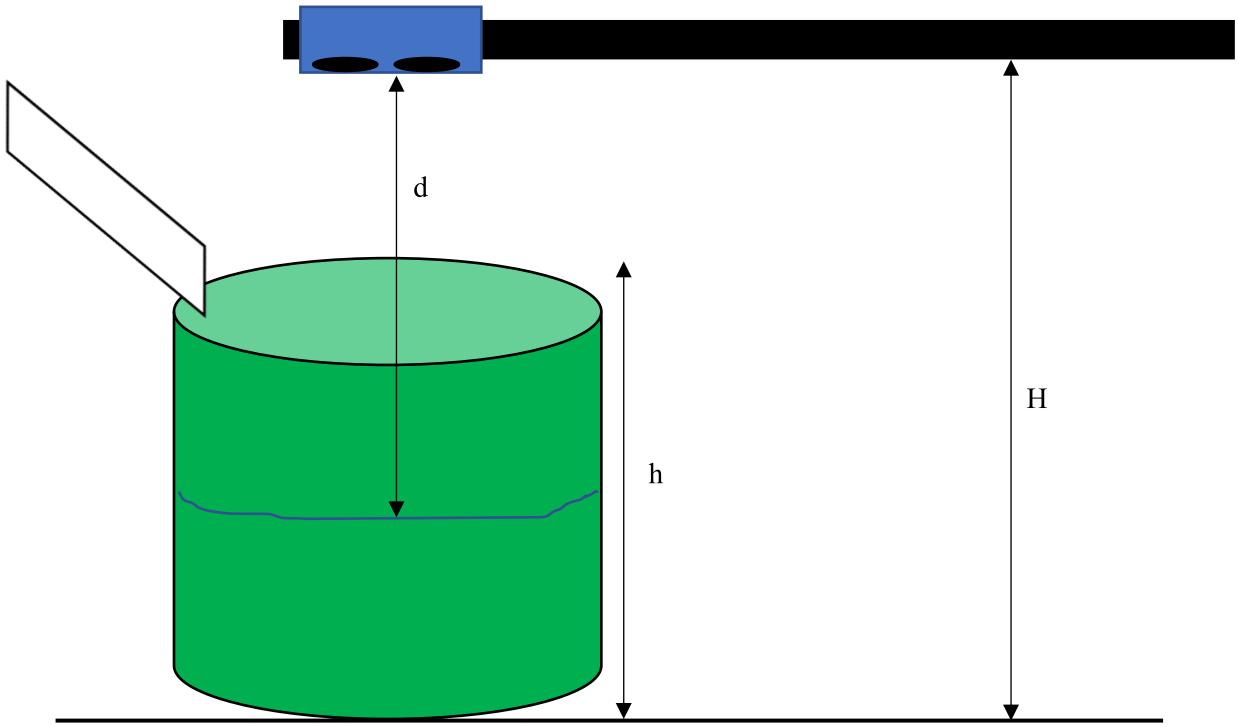


Figure 1(b): Illustration of the Proposed Use Case Application

From the illustration above, the parameters d, h, and H are essential in determining the substance height level inside the container and the percentage that the container is filled with water, where H is the height from the ground to the attached ultrasonic sensor, h is the container height and d refers to the distance from the ultrasonic sensor to the surface of the substances or water.

However, as the ultrasonic sensor only takes the distance from its transmitter to the surface of the substance, which is not the distance concerned. The water/substance inside the container can be calculated as:

$$Water\ Level = H - d$$

For the percentage of the substance level with respect to the container height, it is calculated using the formula:

$$\%\ of\ level = \frac{H - d}{h} \times 100\%$$

Besides measuring the substance or water level, the data is sent to the LCD screen for monitoring purpose. In the LCD, it will display the real-time percentage of the substance or water level inside the container, as well as the level status, as shown below:

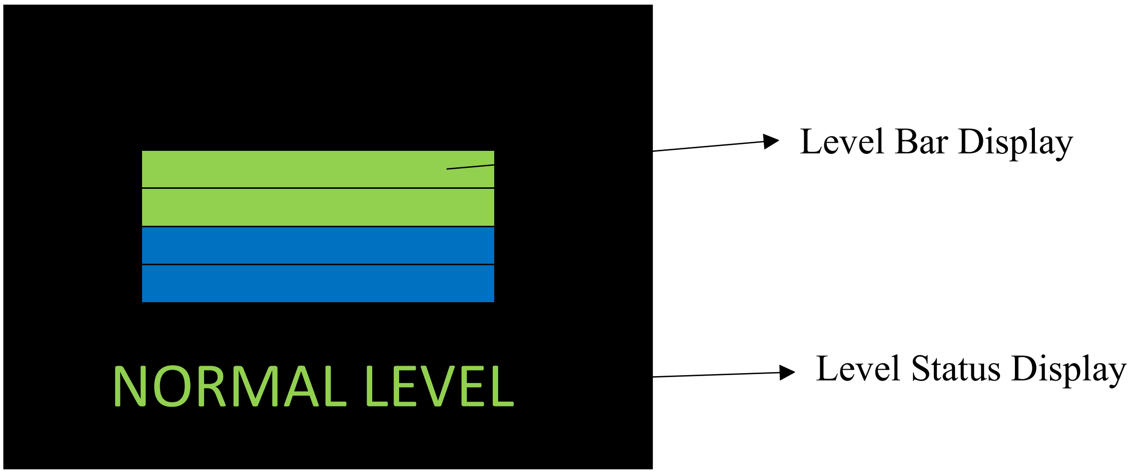


Figure 1(c): Interface in LCD implemented

For the LCD interface, it will display the percentage of the substance level, as well as the level status. The level status consists of 3 level status, which are:

- When substance level is more than or equal to 70 %
  -It will display "TOO HIGH!!!!!" with red text

- When substance level is between 30 % and 70 % with green text
  -It will display "NORMAL LEVEL"

- When substance level is less than or equal to 30 %
  -It will display "TOO LOW!!!!!" with blue text

For the status level, when the percentage reaches more than 70 %, it will send warning messages to create awareness at the surrounding, such as buzzer siren.

From the proposed use case, the objective of this proposed development is to utilise the manpower inside the production line effectively. It also serves as a purpose to keep track and record the substance or water level from time to time, in order to prevent any accident occurring when the substance level has overflown from the container. Or in some situations, in some chemical tanks when the chemical level is too low, causing the machine not be able to process the chemicals effectively.

I/O (Input/Output) table describing the details of every input and output device:

Components:

Components	Quantity	Descriptions
Ultrasonic Sensor	1	Directly connected to the MK20 Freedom Board through the GPIO port with the FTM functions implemented inside the Freedom board.
LCD Screen	1	Connected to the Freedom Board, used to display the percentage of the water/substance level and the water/substance level status.
Buzzer	1	Triggered when the level threshold is exceeded or is way beyond.

Table 2(a): Lists of Components and their functions

Uses of the GPIO Port:

Port Name	Port Type	Descriptions
PTC4	Output	Input pin used as PWM function to send a 10 us pulse from the trigger pin of the ultrasonic sensor
PTA1	Input	Input pin which functions as input capturing to obtain the rising edge of the echo pulse when the ultrasonic sensor initially sends a sound wave.
PTA5	Input	Perform almost the same function as PTA1, but instead of rising edge, it becomes falling edge, and capture the time for the falling edge when the sound wave reflects and returns to the ultrasonic sensor.
PTB3	Output	Used to turn on the buzzer when the substance/water level reaches the upper threshold

Table 2(b): Lists of Port and Pins used in MKD20 and their usages

Circuit diagram of the proposed system:

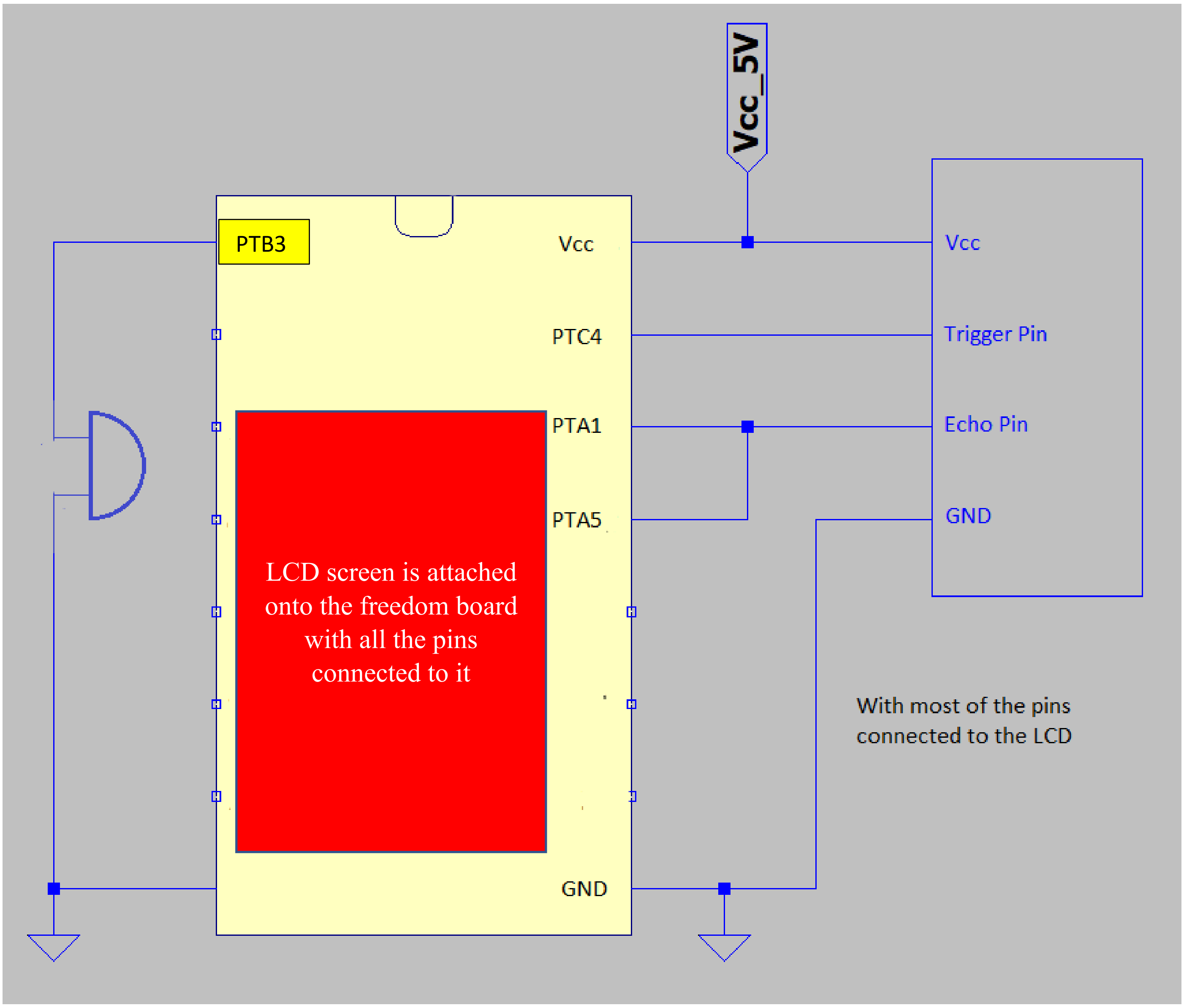


Figure 3(a): Schematic Diagram of the Proposed Use Case, with most of the pins connected to the LCD as well and 4 of the pins from MKD20 are connected to the Ultrasonic Sensor

Block Diagram:

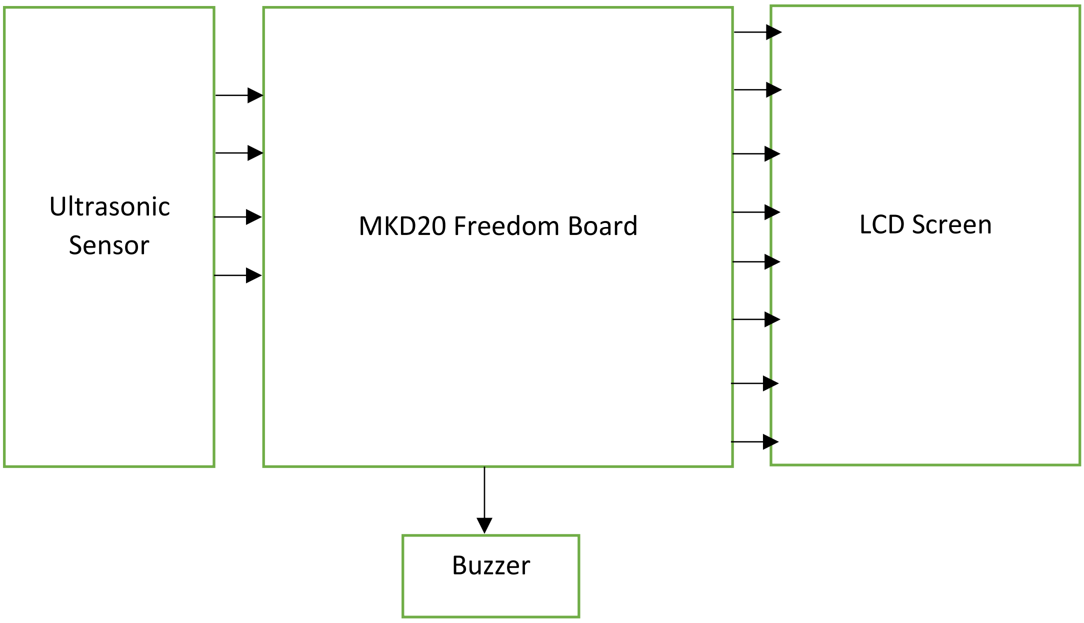


Figure 4(a): Block Diagram of the Proposed Use Case, with the input taken from the ultrasonic sensor, and the output which is connected to LCD screen and the buzzer

From the block diagram above, the ultrasonic sensor measures the height level of the substance/water inside the container using FTM PWM and input capture mode. The captured data is then fed into the MKD20 Freedom Board to display the desired result at the LCD screen. The buzzer will be triggered using PIT(Programmable Interrupt Timer) when the height level is above or beyond a certain range.

From the MKD20, three pins are used to connect the ultrasonic sensor(excluding the Vcc and ground pin), which are:

- PTC4 - Connected to the Trig Pin to send a 10 μs pulse from the transmitter using PWM mode
- PTA1 – Connected to the Echo Pin to capture the rising edge of the pulse when the sensor starts sending the sound wave from the sensor towards the obstacles
- PTA5 – Connected to the Echo Pin as well, but this pin will capture the falling edge of the pulse when the sensor receives the sound wave which is reflected back from the obstacles.

Detects rising edge

Figure 4(b): Illustration of how ultrasonic sensor works

Implementation of the Ultrasonic Sensor with the FTM function inside MKD20 Freedom Board

As mentioned above, the input capture and the PWM(Pulse Width Modulation) are part of the functions of FTM Module. To setup the FTM functions, suitable ports and pins are required as only some of the pins have FTM functions, as shown below:

FRDM-MK20 Pin Muxing (abbreviated)

Label	Pin Name	Default Fn	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	On-board Use	Display
A0	PTC0	ADC0_SE14/TSI0_CH13	ADC0_SE14/TSI0_CH13	PTC0	SPI0_PCS4	PDB0_EXTRG						East-Switch
A1	PTC1	ADC0_SE15/TSI0_CH14	ADC0_SE15/TSI0_CH14	PTC1/LLWU_P6	SPI0_PCS3	UART1_RTS_b	FTM0_CH0		I2S0_TXD0			South-Switch
A2	PTD6	ADC0_SE7b	ADC0_SE7b	PTD6/LLWU_P15	SPI0_PCS3	UART0_RX	FTM0_CH6		FTM0_FLT0			West-Switch
A3	PTD5	ADC0_SE6b	ADC0_SE6b	PTD5	SPI0_PCS2	UART0_CTS_b UART0_COL_b	FTM0_CH5		EWM_OUT_b			Centre-Switch
A4	PTB1	ADC0_SE9/TSI0_CH6	ADC0_SE9/TSI0_CH6	PTB1	I2C0_SDA	FTM1_CH1			FTM1_QD_PHB		Accelerometer (I2C)	North-Switch
A5	PTB0	ADC0_SE8/TSI0_CH0	ADC0_SE8/TSI0_CH0	PTB0/LLWU_P5	I2C0_SCL	FTM1_CH0			TM1_QD_PHA		Accelerometer (I2C)	

Label	Pin Name	Default Fn	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	On-board Use	
	PTB2		ADC0_SE12/TSI0_CH7	PTB2	I2C0_SCL	UART0_RTSb			FTM0_FLT3			
	PTB3		ADC0_SE13/TSI0_CH8	PTB3	I2C0_SDA	UART0_CTSb			FTM0_FLT0			
D13	PTD1	ADC0_SE5b	ADC0_SE5b	PTD1	SPI0_SCK	UART2_CTS_b						SPI-SCK
D12	PTD3			PTD3	SPI0_SIN	UART2_TX						
D11	PTD2			PTD2/LLWU_P13	SPI0_SOUT	UART2_RX						SPI-DIN
D10	PTC2	ADC0_SE4b/CMP1_IN0 TSI0_CH15	ADC0_SE4b CMP1_IN0/TSI0_CH15	PTC2	SPI0_PCS2	UART1_CTS_b	FTM0_CH1		I2S0_TX_FS			Backlight
D9	PTA2	JTAG_TDO/TRACE_SWO	TSI0_CH3	PTA2	UART0_TX	FTM0_CH7					Blue LED	CSn
D8	PTA12			PTA12		FTM1_CH0			I2S0_TXD0	FTM1_QD_PHA		RESETn

Label	Pin Name	Default Fn	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	On-board Use	
D7	PTC4			PTC4/LLWU_P8	SPI0_PCS0	UART1_TX	FTM0_CH3		CMP1_OUT			
D6	PTC3	CMP1_IN1	CMP1_IN1	PTC3/LLWU_P7	SPI0_PCS1	UART1_RX	FTM0_CH2	CLKOUT	I2S0_TX_BCLK		Red LED	
D5	PTA1	JTAG_TDI/EZP_DI	TSI0_CH2	PTA1	UART0_RX	FTM0_CH6						
D4	PTC8	CMP0_IN2	CMP0_IN2	PTC8			I2S0_MCLK					
D3	PTD4			PTD4/LLWU_P14	SPI0_PCS1	UART0_RTS_b	FTM0_CH4		EWM_IN		Green LED	
D2	PTA5			PTA5	USB_CLKIN	FTM0_CH2			I2S0_TX_BCLK			
D1	PTE0			PTE0		UART1_TX				RTC_CLKOUT		
D0	PTE1			PTE1/LLWU_P0		UART1_RX						

Label	Pin Name	Default Fn	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	On-board Use	
	ADC0_DM0	ADC0_DM0	ADC0_DM0								Light Sensor	
	ADC0_DM3	ADC0_DM3	ADC0_DM3								Temp. Sensor	

Label	Pin Name	Default Fn	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	On-board Use	
	PTC9	CMP0_IN3	CMP0_IN3	PTC9			I2S0_RX_BCLK					
	PTD7			PTD7	CMT_IRO	UART0_TX	FTM0_CH7		FTM0_FLT1			
	PTA4	NMI_b	TSI0_CH5	PTA4/LLWU_P3		FTM0_CH1				NMI_b		
	PTC7	CMP0_IN1	CMP0_IN1	PTC7	SPI0_SIN	USB_SOF_OUT	I2S0_RX_FS			I2S0_MCLK		
	PTC6	CMP0_IN0	CMP0_IN0	PTC6/LLWU_P10	SPI0_SOUT	PDB0_EXTRG	I2S0_RX_BCLK			I2S0_MCLK	Accelerometer Int 2	
	PTC5			PTC5/LLWU_P9	SPI0_SCK	LPTMR0_ALT2	I2S0_RXD0		CMP0_OUT	NMI_b		
	PTC11			PTC11/LLWU_P11							Accelerometer Int 1	
	PTD0			PTD0/LLWU_P12	SPI0_PCS0	UART2_RTSb						

Label	Pin Name	Default Fn	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	On-board Use	
	PTA13			PTA13/LLWU_P4		FTM1_CH1			I2S0_TX_FS	FTM1_QD_PHB		
	PTC10			PTC10			I2S0_RX_FS					

Figure 4(c): Pin Muxing details for all ports and pins of MKD20 Freedom Board

From the diagram above, it can be seen that

- Port PTC4 is using FTM0_CH3, which indicates channel 3 of FTM0, Mux(4)
- Port PTA1 is using FTM0_CH6, which indicates channel 6 of FTM0, Mux(3)
- Port PTA5 is using FTM0_CH2, which indicates channel 2 of FTM0, Mux(3)

To use the FTM module inside the MKD20 Freedom Board effectively, first the clock needs to be enabled to the register SIM_SCGC6. Next, since all three ports use the same FTM0, hence, it needs to be enabled to the FTM0 mask by writing `SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;` The register FTM0_SC needs to be disabled so that immediate changes can be made, by disabling the counter clock source, which is `FTM_SC_CLKS(0).`

Next, another important field inside the FTM0_SC register is the prescaler value required. For FTM, the maximum number of bits is 16-bit. For the PWM at the Trigger pin, the minimum period required is 60 ms.

Hence, if the prescaler value is not chosen, the period will be:

$Period = \frac{2^{16}}{48\ MHz} = 1.36\ ms$, which is far from the 60 ms requirement, as the clock frequency is not divided by any pre-scaler value which remains the same as 48 MHz.

In this case, let n be the minimum value of the prescaler value required for this case,

Let say the clock frequency is scaled to a smaller value by 2^n value, which is

$$Scaled\ frequency = \frac{48\ MHz}{2^n}$$

And the period is calculated as: $period = \frac{2^{16}}{\frac{48\ MHz}{2^n}} = \frac{2^{16+n}}{48\ MHz}$

From the situation above, the minimum period is 60 ms,

$$\therefore \frac{2^{16+n}}{48\ MHz} \geq 60 \times 10^{-3}$$

$$2^{16+n} \geq 2{,}880{,}000$$

$$\log 2^{16+n} \geq \log(2{,}880{,}000)$$

$$(16 + n) \log 2 \geq \log(2{,}880{,}000)$$

$$16 + n \geq 21.45$$

$$n \geq 5.45$$

Hence, the minimum prescaler value for this situation is 6, hence the prescale is $2^6 = 64$

For this prescaler value,

$$FTM\ Clock\ Frequency = \frac{48\ MHz}{2^6} = 0.75\ MHz$$

$Period = \frac{2^{16}}{0.75\ MHz} = 87.4\ ms$, which is sufficient enough for the 60 ms minimum period.

$$Number\ of\ ticks\ in\ one\ period = 750{,}000\ Hz \times 60\ ms = 45{,}000$$

$Duty\ Cycle = \frac{10\times10^{-6}}{60\times10^{-3}} \times 45{,}000 = 7.5 \approx 8$ in ticks

From the FTM0_SC register map:

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0								TOF	TOIE	CPWMS	CLKS		PS		
W									0							
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4(d): FTM0_SC register map

From this register, only the CLKS and the PS are concerned. The CPWMS bit is not selected because centre-aligned pulse is only used for specific applications, hence Left-Aligned PWM is used instead.

Besides the FTM0_SC register, there are 4 other registers that need to be configured inside the initialisation of FTM function, which are:

(a) FTM0_CNT register(FTM Count Register)

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0																COUNT															
W																																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4(e): FTM0_CNT register

The COUNT value refers to the value when the count reaches 0. In this case, it would be COUNT = 0.

(b) FTM0_MOD

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved																MOD															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4(f): FTM0_MOD register

The MOD refers to the value(number of ticks) when it is reloaded to a value and then start counting down again. The modulo value will be the subtraction of the ticks per period by 1, which is calculated as :

$$MOD = ticks\ per\ period - 1 = 45000 - 1 = 44999$$

(c) FTM0_CNTIN

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved																INIT															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4(g): FTM0_CNTIN register

The FTM0_CNTIN register contains the initial value of the FTM counter, which in turns found to be 0.

(d) FTM0_CnSC

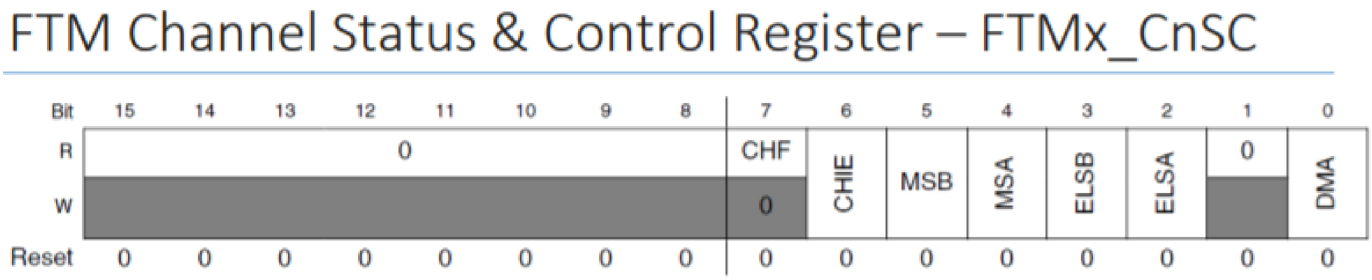


Figure 4(h): FTM0_CnSC register

In this register, since channel interrupts are required to be enabled, hence the CHIE bit needs to be set to 1 to enable it. For the rising and falling edge of the Echo pin, pin PTA1(FTM0_CH6) is responsible for the event at the rising edge, whereas the pin PTA5(FTM0_CH2) is responsible for capturing the event occurs at the falling edge. Hence, the necessary configurations are:

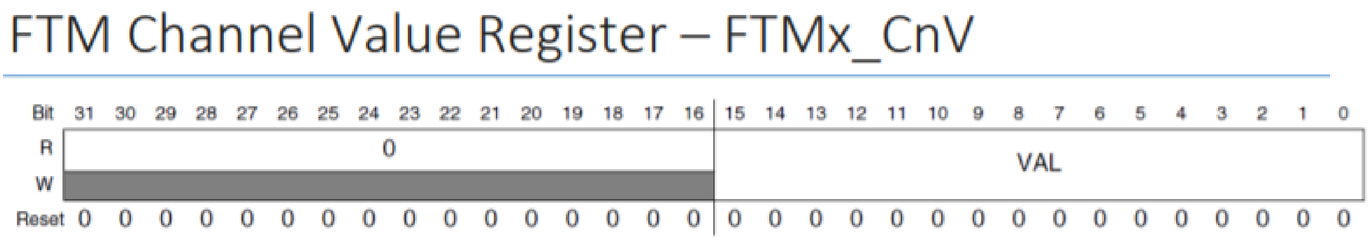
```
FTM0_C6SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSA_MASK;
FTM0_C2SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSB_MASK;
```

where the ELSA and ELSB bit refers to the edge or level selection bit, since two are of them will be used for capturing rising and falling edge, both of these set ups are necessary.

Besides, the duty cycle for the PWM at the trigger input needs to be configured by considering the register FTM0_CnSC again. Since in this case, high true pulses are concerned, hence ELSB bit is chosen.

```
FTM0_CnSC(channel) = FTM_CnSC_MSB_MASK | FTM_CnSC_ELSB_MASK;// Edge-
aligned PWM mode and High true pulses
```

(e) The last register needs to be configured will be the FTM Channel Value Register(FTM0_CnV). The



value inside this register is the channel value used for PWM.

Figure 4(i): FTM0_CnV register

The duty cycle for this situation is:

$$Duty\ Cycle = \frac{10 \times 10^{-6}}{60 \times 10^{-3}} \times 45{,}000 = 7.5 \approx 8$$

Hence, `FTM0_CnV(channel) = 8;`

Code for initialising the FTM0

```
void initialiseFTM0(int period)
{
	// Enable clock to FTM
	SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;

	// Common registers
	FTM0_SC = FTM_SC_CLKS(0); // Disable FTM counter so changes are
immediate
	FTM0_CNTIN = 0;
	FTM0_CNT = 0;//Value when reset to 0
	FTM0_MOD = period-1;//Value of period

	// Left aligned PWM since CPWMS not selected
	FTM0_SC = FTM_SC_CLKS(1)|FTM_SC_PS(FTM0_PRESCALE_VALUE);


	// Enable FTM0 interrupts in NVIC
	NVIC_EnableIrq(INT_FTM0);

	// Channel register
	FTM0_C6SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSA_MASK; // rising edge
is for ELSA MASK
	FTM0_C2SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSB_MASK; // falling edge
is for ELSB MASK
}
```

Consequently, the FTM0_IRQHandler needs to be set up to capture the rising and falling edge resulted from the Echo Pin:

```
void FTM0_IRQHandler(void)
{
	if ((FTM0_C6SC&FTM_CnSC_CHF_MASK) != 0)
	{
		// Clear the interrupt request from FTM0.Ch6
		FTM0_C6SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSA_MASK;

		firstMeasurement = FTM0_C6V;
		firstMeasurementDone = true;
		//FTM0_C2SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSB_MASK;
	}
	if ((FTM0_C2SC&FTM_CnSC_CHF_MASK) != 0)
	{
		// Clear the interrupt request from FTM0.Ch2
		FTM0_C2SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSB_MASK;
		if (firstMeasurementDone)
		{
			// Ignore transitions until 1st has occurred
			secondMeasurement = FTM0_C2V;
			//printf("Second = %d\n",secondMeasurement);
			secondMeasurementDone = true;
		}
	}
}
```

From the code for the FTM0 IRQ Handler above, when the handler detects an interrupt from the FTM0 channel 6, it will clear the interrupt request flag from the FTM0. Channel 6, the variable `firstMeasurement` is then assigned to capture the timing of the rising edges, the measurement for the rising edge is now completed by writing a true to the variable `firstMeasurementDone`.

Next, when when the handler detects an interrupt from the FTM0 channel 2, it will clear the interrupt request flag from the FTM0 channel 2, the variable `secondMeasurement` is then assigned to capture the timing of the falling edges, the measurement for the falling edge is now completed by writing a true to the variable `secondMeasurementDone`.

The function below calculates and returns the height level of the substance/water:

```
getWaterLevelHeight(void)
{
	NVIC_DisableIrq(INT_FTM0);
	int tempFirst = firstMeasurement; //rising edge timing
	int tempSecond = secondMeasurement; //falling edge timing
	printf("First = %d\n",tempFirst);
	printf("Second = %d\n",tempSecond);
	NVIC_EnableIrq(INT_FTM0);
	int distance, height;

	distance = ((tempSecond - tempFirst)/ONE_MICROSECOND)/58;
	height = POSITION_HEIGHT - distance; //height level of the
substance/water

	return height; //in cm
}
```

From the previous FTM0_IRQHandler function, the variables *firstMeasurement* and *secondMeasurement* is used as the timing for the rising and falling edges.

From the datasheet, the formula for calculating the distance from the sensor to the object is:

$$Test\ distance\ =\ high\ level\ time \times velocity\ of\ sound \frac{\frac{340\ metre}{second}}{2}$$

$$= \frac{high\ level\ time}{ticks\ per\ microsecond}\left(\frac{1}{58}\right)$$

The ticks per microsecond is calculated by:

$$\frac{1 \times 10^{-6}}{\left(\frac{1}{750\ kHz}\right)} = 0.75$$

$$Distance = \frac{high\ level\ time}{0.75}\left(\frac{1}{58}\right)$$

Implementation of PIT and SysTick Timer to the Proposed Use Case

(a) Programmable Interrupt Timer

(i) Initialisation of PIT

```
void initialisePIT(int channel, uint32_t interval) {
    // Enable clock to PIT
    SIM_SCGC6   |= SIM_SCGC6_PIT_MASK;
    // Enable PIT module
    PIT_MCR     = PIT_MCR_FRZ_MASK;
    // Set re-load value
    PIT_LDVAL(channel)  = interval-1;
    // Enable this channel with interrupts
    PIT_TCTRL(channel)  = PIT_TCTRL_TEN_MASK|PIT_TCTRL_TIE_MASK;
    // Enable PIT0 interrupts in NVIC
    NVIC_EnableIrq(INT_PIT0+channel);
    // Set arbitrary priority level
    NVIC_SetIrqPriority(INT_PIT0, 8);
}
```

Figure 5(a): Function for the PIT initialization

For the initialisation of the PIT, it is required to enable the clock to the register SIM_SCGC6. At the PIT_MCR register, the FRZ bit is enabled in order for the timer to stop during the debugging mode.

Address: PIT_MCR is 4003_7000h base + 0h offset = 4003_7000h

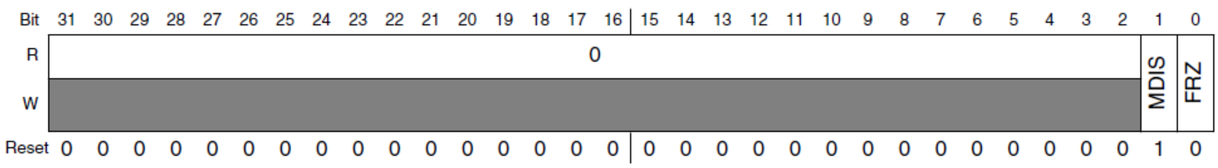


Figure 5(b): Location of FRZ mask at the PIT_MCR register

Since PIT consists of 4 channels, thus they are required to be configured to avoid any conflicts occurred by assigning interval-1 into the PIT_LDVAL register. Here, the variable *interval* refers to the reload value whenever an interrupt is raised.

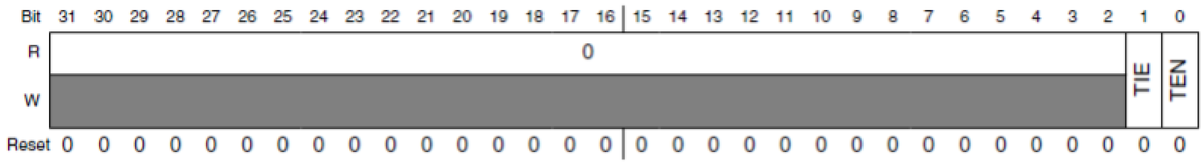


Figure 5(c): PIT_TCTRL register map

For the configuration of the PIT_TCTRL register, TIE and TEN bit need to be enabled so that interrupts and timer can operate by setting both TIE and TEN bit to 1.

(ii) PIT Interrupt Handler

```
void PIT_Ch0_IRQHandler(void) {
    // Toggle the pin PTB2 for the buzzer
    GPIOB_PTOR = BUZZER_MASK;
    // clear the interrupt request from PIT
    PIT_TFLG0 = PIT_TFLG_TIF_MASK;
}
```

Figure 5(d): Code for the PIT Interrupt Handler

The PIT_Ch0_TRQHandler is mainly used for the interrupt handler whenever PIT detects an interrupt. Inside the PIT handler function, the buzzer at pin PTB2 will keep toggling whenever an interrupt is triggered. The TIF mask inside the TFLG0 register will be cleared, in other words the TIF flag is cleared by writing 1.

(iii) Frequency Converter Function

```
int convertPit(int inputfrequency) {
    int ans1 = 0;
    ans1 = ((PIT_CLOCK_FREQUENCY/inputfrequency)/2);
    return ans1;
}
```

Figure 5(e): Function to convert the frequency

The function is used to convert the input frequency from the value declared at the playTone function to the reload value in which will be used at the PIT initialisation. This reload value(in this case ans1) returned by the function will control the duration of raising 1 interrupt.

(iv) playTone function

```
void playTone(int frequency, double duration) {
    // initialise the variable
    int Pits;
    double Ticks;

    /* Convert the input frequency and duration
     * to corresponding reload value for Systick Timer and
     * PIT. */
    Ticks=convertTicks(duration);
    Pits = convertPit(frequency);

    SysTick_Config(Ticks);

    initialisePIT(0, Pits); // Enable the PIT interrupt
    delayMS(1000); // Operate for certain duration
    NVIC_DisableIrq(INT_PIT0+0);// Disable Pit interrupt
    delayMS(1000); // Operate for certain duration
}
```

Figure 5(f): playTone function implementation in the program

The function playTone takes in two parameters, which are frequency and duration of the buzzer when it is turned on and turned off. The frequency and duration are then converted into the reload value by the function convertTicks(duration) and convertPit(frequency). The reload value for the frequency will be used for the PIT and for the duration, it will be used for the SysTick Timer. The function SysTick_Config(Ticks) will handle the ticks which are passed from the convertTicks(duration) function, and the function initialisePIT() will handle the reload value from the frequency. To make sure the tone is played for a certain duration, delayMS(1000) will be used to set the period of on time for the buzzer tone. After that, the PIT interrupt will be disabled after the delay. This function allows the buzzer to play the tone without stopping it unless being terminated, thus creating an infinite loop for the buzzer tone.

(b) SysTick Timer

(i) SysTick_Config

```
uint32_t SysTick_Config(uint32_t ticks) {
    if ((ticks - 1) > SysTick_RVR_RELOAD_MASK)
    {
        /* Reload value impossible */
        return (1);
    }
    /* Set reload register */
    SYST_RVR = SysTick_RVR_RELOAD(ticks-1);
    /* Set Priority for Systick Interrupt */
    NVIC_SetIrqPriority (INT_SysTick, (1<<4) - 1);
    /* Load the SysTick Counter Value */
    SYST_CVR = 0;
    /* Configure Systick */
    SYST_CSR =
            SysTick_CSR_CLKSOURCE_MASK | // Use system core clock
            SysTick_CSR_TICKINT_MASK | // Enable interrupts
            SysTick_CSR_ENABLE_MASK; // Enable timer
/* Function successful */
    return (0);
}
```

Figure 5(g): Code for the configuration of SysTick Timer Interrupt

The following function is required for configuring the SysTick timer to operate properly. It is used to set the interrupt rate and enable the interrupt and timer.

(ii) SysTick_Handler and delayMS() function

```
void SysTick_Handler(void) {
    count++;//count by 1 when there's an interrupt raised
}

void delayMS(unsigned long delay) {
    count = 0;
    while(count<delay);
}
```

Figure 5(h): SysTick Handler function with the delayMS function

SysTick_Handler will be called by the program whenever an interrupt is raised. If this happens, the count will increment by 1.
For the delayMS() function, it is used to create delay period for raising an interrupt to the Timer. The maximum period that can be achieved by the timer is:
$T_{max} = \frac{1}{f_{clock}}(2^n - 1), wher\ f_{clock}$ refers to the clock frequency, which is 48 MHz and n is the number of bits of the timer.
SysTick Timer consists of 24 bits, hence $n = 24$.
$\therefore T_{max} = \frac{1}{48\times10^6}(2^{24} - 1) = 0.3495\ s \approx 0.35\ s$(maximum period achieved by the SysTick timer)

This function also allows the tone generator to produce the buzzer duration exceeding $0.35\ s$.

(iii) Duration Converter function

```
double convertTicks(double duration) {
    double ans = 0;
    ans = ((duration*PIT_CLOCK_FREQUENCY)/1000);
    return ans;
}
```

Figure 5(i): Function to convert duration into reload value

The variable duration will be passed into the function which is to be converted into the reload value for SysTick configuration. The reload value will be used to trigger the interrupt requested with a certain time duration.

PIT Interrupt Calculation

In PIT, the timer loads the value and then starts counting down to zero. In this case, PIT loads the value passed from the frequency at the playTone function.

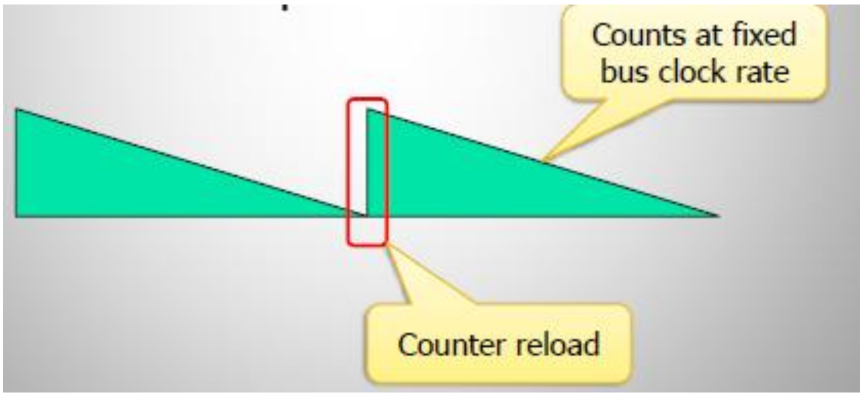


Figure 6(a): Generation of interrupt with waveform

From the waveform representation, the peak of the triangular waveform represents the reload value from the frequency. Each triangular represents a single interrupt, and each period consists of 2 interrupts. The frequency of the waveform is given by:

From the code:

```
#define SYSTEM_CLOCK_FREQUENCY       (48000000UL)//clcok frequency as 48 MHz
#define PIT_CLOCK_FREQUENCY          SYSTEM_CLOCK_FREQUENCY
#define PIT_TICKS_PER_MICROSECOND    (PIT_CLOCK_FREQUENCY/1000000)   // 48 ticks
#define PIT_TICKS_PER_MILLISECOND    (PIT_CLOCK_FREQUENCY/1000)   // 48000 ticks

int convertPit(int inputfrequency) {
     int ans1 = 0;
     ans1 = ((PIT_CLOCK_FREQUENCY/inputfrequency)/2);
     return ans1;
     }
```

The variable ans1 refers to the reload value, and the PIT_CLOCK_FREQUENCY refers to the clock frequency which is 48 MHz.

$$Reload\ Value = \frac{48\ MHz}{2f_{in}}$$

In this case, if an input frequency of 2000 Hz is used in the generation of buzzer tone. Then the reload value is given by:

$$Reload\ Value = \frac{48\ MHz}{2(2000)} = 12{,}000\ ticks$$

Case 2: $f_{in} = 4\ kHz$

$$Reload\ Value = \frac{48\ MHz}{2(4000)} = 6{,}000\ ticks$$

From both cases, it can be seen that to use a buzzer tone frequency of higher value, the reload value must be lower.

SysTick Interrupt Calculation

For SysTick Timer, the waveform used will be the same as from the *Figure 8* above.
From the code below:

```
#define SYSTEM_CLOCK_FREQUENCY        (48000000UL)//clcok frequency as 48
MHz
#define PIT_CLOCK_FREQUENCY           SYSTEM_CLOCK_FREQUENCY
#define PIT_TICKS_PER_MICROSECOND     (PIT_CLOCK_FREQUENCY/1000000)   // 48
ticks
#define PIT_TICKS_PER_MILLISECOND     (PIT_CLOCK_FREQUENCY/1000)   // 48000
ticks

/* Function that used to convert input duration to
* the reload value of Systick Timer */
double convertTicks(double duration) {
	double ans = 0;
	ans = ((duration*PIT_CLOCK_FREQUENCY)/1000);
	return ans;
}
```

From the code below:

$$Reload\ Value = \frac{T_{one\ interrupt} \times f_{clock}}{100}$$

In this case, assume a 1 second of interrupt is to be generated, the reload value required will be:

$$Reload\ Value = \frac{1 \times 48\ MHz}{100} = 480{,}000\ ticks$$

From the equation, the value 1,00 refers to the value from the function delayMS(100). With the implementation of this function, the function allows the delay period to be longer than 1s.

Case 2: If 0.5 s of interrupt is required to be generated:

$$Reload\ Value = \frac{0.5 \times 48\ MHz}{100} = 240{,}000\ ticks$$

From both cases, to create a longer duration of the interrupt, the reload value must be high enough.

LCD Implementation and Interfacing

For the LCD display, there are only two information being displayed on it, with the first one as the water level bar indicator and the water level status.

First of all, the features of the LCD screen are explained as below:

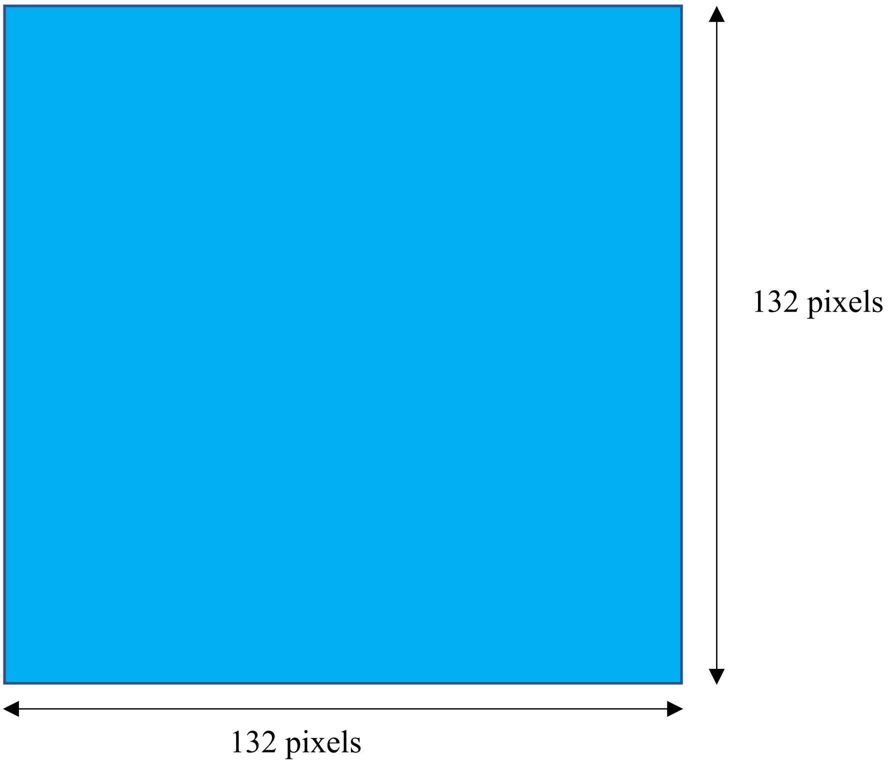


Figure 7(a): Illustration of the LCD screen by number of each pixels in each dimensions

From the table, the LCD screen is equally dimensioned with 132 pixels on each side of the LCD screen. To draw shapes, lines, or display texts on the screen, the coordinates must be considered. For example, the origin starts at $x = 0\ and\ y = 0$ in terms of x and y coordinates. The illustration of how the rectangle bars is explained as below:

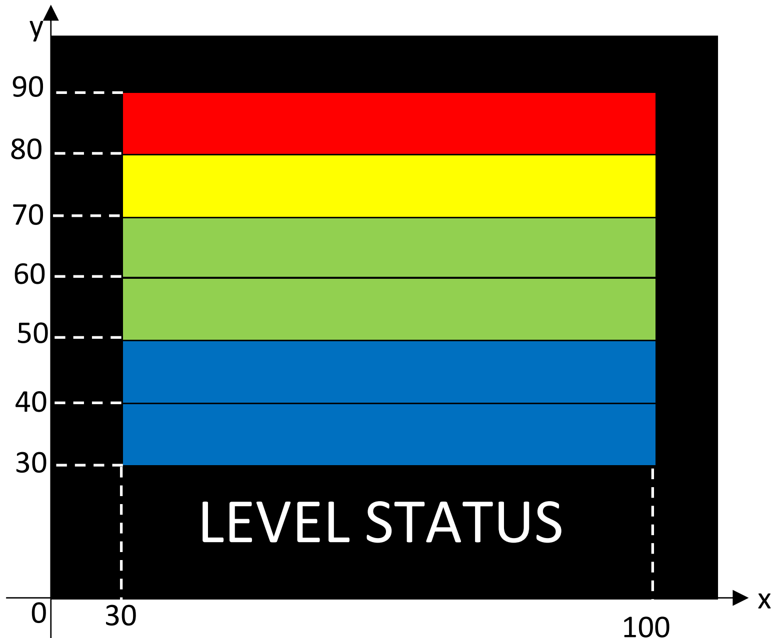


Figure 7(b): Illustration of the Rectangular Water Level Bar Interface with x-y coordinates representation

From the interface above, for example, when a bottommost blue rectangular bar is to be drawn at the LCD, the code is written as:

```
lcd_drawRect(30,30,100,40,BLUE,BLUE);
```

Initial x-coordinate | Initial y-coordinate | Final x-coordinate | Initial y-coordinate | Filled colour of bar | Outlined colour of bar

The code statement above will plot out the rectangular bar with the input x and y coordinates.

Case 1: When water level is below or equal to 30%.

Figure 7(c): LCD Interface when the level is below or equal to 30%

When the water level is below or equal to 30%, the LCD will only draws put two blue rectangular bar, and it will display the text “TOO LOW!!!!!” in blue colour.

Case 2: When water level is between 30 and 70%

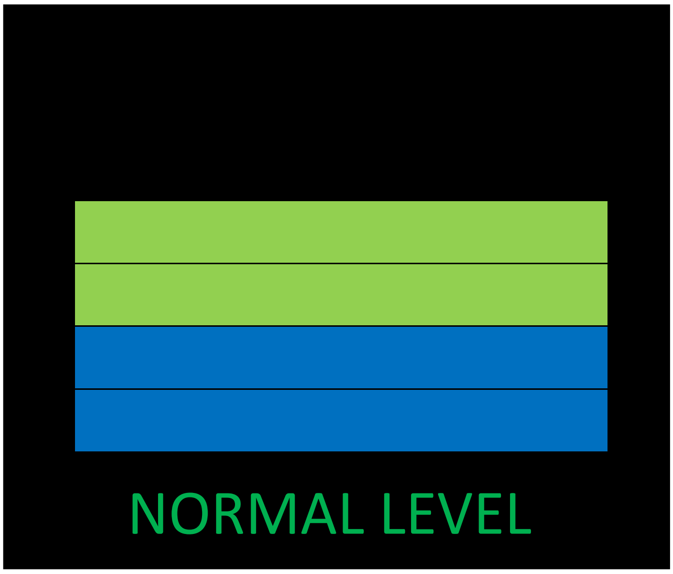


Figure 7(d): LCD Interface when the level is between 30 and 70%

When the water level is between 30 and 70%, the LCD will only draw the first 4 blue rectangular bars, and it will display the text “NORMAL LEVEL” in green colour.(3 bars for 30-50% and 4 bars for 50-70%)

Case 3: When water level is between 70 and 90%

Figure 7(e): LCD Interface when the level is between 70 and 90%

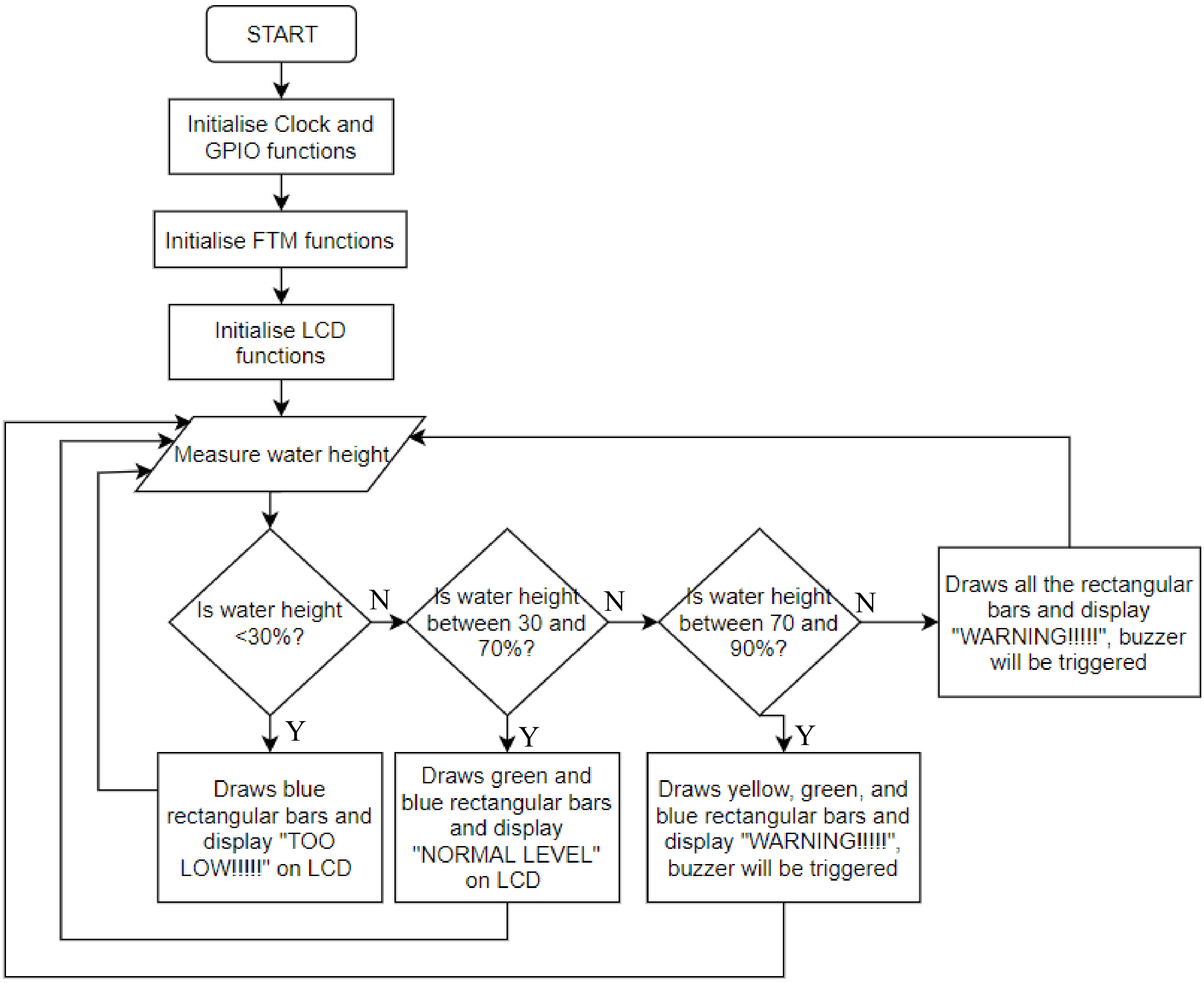
When the water level is between 70 and 90%, the LCD will only draw the first 4 blue rectangular bars, and it will display the text "WARNING!!!!!" in red colour. In addition, the buzzer connected to PTB3 will be triggered.

Case 4: When water level is more than 90%

Figure 7(f): LCD Interface when the level is more than 90%

When the water level is more than 90%, the LCD will only draw all the rectangular bars, and it will display the text "WARNING!!!!!" in red colour. In addition, the buzzer connected to PTB3 will be triggered as well.

Flowchart for the Main Program:



Conclusion

In conclusion, the proposed case being implemented was mostly successful, in which the water level can be detected accurately with the water level display bar on the LCD screen. During the process of measuring the water height level, one of the issue faced is the attachment of the sensor to a certain height to measure the water height effectively. Although there are some rooms for improvements need to be made, especially in terms of the blinking issue when the LCD screen is displaying the data. And other than that, the written C program can be improved with the number of lines reduced, though the method of displaying the water level indicator bar is not the best approach to do so, hence the method can be further improved. In addition, this assignment also further provides a better understanding of the operation of LCD screen and the use of PIT and FTM functions.

Video Demonstration Link:

https://youtu.be/4muat__CLzI

Appendix

Overall Program

```
#include <stdio.h>
#include "clock.h"
#include <stdlib.h>
#include "derivative.h"
#include "nokia_LCD.h"
#include "uart.h"
#include "Freedom.h"
#include "Die.h"
#include "utilities.h"
#include "string.h"

typedef enum {false=0, true=!false} bool;

#define SYSTEM_CLOCK_FREQUENCY      (48000000UL)//Set as 48 MHz
#define PIT_CLOCK_FREQUENCY         SYSTEM_CLOCK_FREQUENCY
#define PIT_TICKS_PER_MICROSECOND   (PIT_CLOCK_FREQUENCY/1000000)
#define PIT_TICKS_PER_MILLISECOND   (PIT_CLOCK_FREQUENCY/1000)
#define BUZZER_MASK (1<<3)                //...0000 1000

#define FTM0_PRESCALE_VALUE             (6)//Prescaler value
#define FTM0_PRESCALE                           64//2^6 = 64
#define FTM0_CLK_FREQUENCY
      (SYSTEM_CLOCK_FREQUENCY/FTM0_PRESCALE)//FTM clock frequency
#define ONE_MICROSECOND (0.75)                  //Ticks in one microsecond
#define PWM_PERIOD (45000)                      //60ms

#define POSITION_HEIGHT                         15 //position from the sensor to the
ground in cm(can be changed if different height is used)
#define CONTAINER_HEIGHT                        13 //Measured height of the container
in cm(can be changed if different container is used)

#define BACKGROUND_COLOUR BLACK
unsigned long count = 0;

static volatile uint16_t firstMeasurement;
static volatile bool     firstMeasurementDone;
static volatile uint16_t secondMeasurement;
static volatile bool     secondMeasurementDone;

char selstr[20]; //String to store selected distance as string

uint32_t SysTick_Config(uint32_t ticks)
{
      if ((ticks - 1) > SysTick_RVR_RELOAD_MASK)
      {
            /* Reload value impossible */
            return (1);
      }
      /* Set reload register */
      SYST_RVR = SysTick_RVR_RELOAD(ticks-1);
      /* Set Priority for Systick Interrupt */
      NVIC_SetIrqPriority (INT_SysTick, (1<<4) - 1);
      /* Load the SysTick Counter Value */
      SYST_CVR = 0;
      /* Configure Systick */
      SYST_CSR =
                  SysTick_CSR_CLKSOURCE_MASK | // Use system core clock
                  SysTick_CSR_TICKINT_MASK | // Enable interrupts
```

```c
                    SysTick_CSR_ENABLE_MASK; // Enable timer
/* Function successful */
      return (0);
}

void SysTick_Handler(void)
{
      count++;
}

void delayMS(unsigned long delay)
{
      count = 0;
      while(count<delay);
}

void initialisePIT(int channel, uint32_t interval)
{
   // Enable clock to PIT
   SIM_SCGC6  |= SIM_SCGC6_PIT_MASK;
   // Enable PIT module
   PIT_MCR     = PIT_MCR_FRZ_MASK;
   // Set re-load value
   PIT_LDVAL(channel)  = interval-1;
   // Enable this channel with interrupts
   PIT_TCTRL(channel)  = PIT_TCTRL_TEN_MASK|PIT_TCTRL_TIE_MASK;
   // Enable PIT0 interrupts in NVIC
   NVIC_EnableIrq(INT_PIT0+channel);
   // Set arbitrary priority level
   NVIC_SetIrqPriority(INT_PIT0, 8);
}

void PIT_Ch0_IRQHandler(void)
{
   // Toggle the pin at PTB3 of GPIO
   GPIOB_PTOR = BUZZER_MASK;
   // clear the interrupt request from PIT
   PIT_TFLG0 = PIT_TFLG_TIF_MASK;
}

double convertTicks(double duration)
{
      double ans = 0;
      ans = ((duration*PIT_CLOCK_FREQUENCY)/1000);
      return ans;
}

int convertPit(int inputfrequency)
{
      int ans1 = 0;
      ans1 = ((PIT_CLOCK_FREQUENCY/inputfrequency)/2);
      return ans1;
}

void playTone(int frequency, double duration)
{
      // initialise the variable
            int Pits;
            double Ticks;

            /* Convert the input frequency and duration
             * to corresponding reload value for Systick Timer and
             * PIT. */
            Ticks=convertTicks(duration);
            Pits = convertPit(frequency);

            SysTick_Config(Ticks);
```

```
            initialisePIT(0, Pits); // Enable the PIT interrupt
            delayMS(100); // Operate for certain duration
            NVIC_DisableIrq(INT_PIT0+0);// Disable Pit interrupt
            delayMS(100); // Operate for certain duration
}

void PortInitialise(void)
{//Set the GPIO required for the measurement
      SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK|SIM_SCGC5_PORTC_MASK;
      PORTC_PCR4 = PORT_PCR_MUX(4)|PORT_PCR_DSE_MASK;//PWM(PTC4 for Trigger Pin)
      PORTA_PCR1 = PORT_PCR_MUX(3)|PORT_PCR_PE_MASK;//Input Capture for Echo Pin(Rising
Edge)-PTA1
      PORTA_PCR5 = PORT_PCR_MUX(3)|PORT_PCR_PE_MASK;//Input Capture for Echo
Pin(Falling Edge)-PTA5
}

void initialiseFTM0(int period)
{
      // Enable clock to FTM
      SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;

      // Common registers
      FTM0_SC = FTM_SC_CLKS(0); // Disable FTM counter so changes are immediate
      FTM0_CNTIN = 0;
      FTM0_CNT = 0;//Value when reset to 0
      FTM0_MOD = period-1;//Value of period

      // Left aligned PWM since CPWMS not selected
      FTM0_SC = FTM_SC_CLKS(1)|FTM_SC_PS(FTM0_PRESCALE_VALUE);


      // Enable FTM0 interrupts in NVIC
      NVIC_EnableIrq(INT_FTM0);

      // Channel register
      FTM0_C6SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSA_MASK; // rising edge is for ELSA
MASK in PWM
      FTM0_C2SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSB_MASK; // falling edge is for ELSB
MASK in PWM
}

void initialiseFTM0_PWM(int period)
{
      // Enable clock to FTM
      SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;

      // Common registers
      FTM0_SC = FTM_SC_CLKS(0); // Disable FTM so changes are immediate

      //Then initialise the following registers
      FTM0_CNTIN = 0;
      FTM0_CNT = 0;
      FTM0_MOD = period-1;

      //Then re-enable the Clock Source
      // Left aligned PWM since CPWMS not selected(Centre-Aligned is used in Specific
Application, Not For This!)
      FTM0_SC = FTM_SC_CLKS(1)|FTM_SC_PS(FTM0_PRESCALE_VALUE);
}

void ConfigureDutyCycle(int channel)
{
      // High-true PWM pulses
      FTM0_CnSC(channel) = FTM_CnSC_MSB_MASK | FTM_CnSC_ELSB_MASK;// Edge-aligned PWM
mode
      // High-true pulses
```

```
	// PWM pulse width
	FTM0_CnV(channel) = 8;
}

void FTM0_IRQHandler(void)
{
	if ((FTM0_C6SC&FTM_CnSC_CHF_MASK) != 0)
	{
		// Clear the interrupt request from FTM0.Ch6
		FTM0_C6SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSA_MASK;

		firstMeasurement = FTM0_C6V;
		firstMeasurementDone = true;
		//FTM0_C2SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSB_MASK;
	}
	if ((FTM0_C2SC&FTM_CnSC_CHF_MASK) != 0)
	{
		// Clear the interrupt request from FTM0.Ch2
		FTM0_C2SC = FTM_CnSC_CHIE_MASK|FTM_CnSC_ELSB_MASK;

		if (firstMeasurementDone)
		{
			// Ignore transitions until 1st has occurred
			secondMeasurement = FTM0_C2V;

			//printf("Second = %d\n",secondMeasurement);
			secondMeasurementDone = true;
		}
	}
}

int getWaterLevelHeight(void)
{
	NVIC_DisableIrq(INT_FTM0);
	printf("First = %d\n",firstMeasurement);
	printf("Second = %d\n",secondMeasurement);
	NVIC_EnableIrq(INT_FTM0);
	int distance, height;

	distance = ((secondMeasurement - firstMeasurement)/ONE_MICROSECOND)/58;
	height = POSITION_HEIGHT - distance;

	return height; //cm
}

int main(void){
	clock_initialise();

	lcd_initialise();
	lcd_clear(BACKGROUND_COLOUR);//Clear LCD screen back to black colour

	SIM_SCGC5|= SIM_SCGC5_PORTB_MASK;//Enable clock function to SIM_SCGC5
	PORTB_GPCLR  = PORT_GPCLR_GPWE(BUZZER_MASK)|PORT_PCR_MUX(1);//Set PTB2 as input
for buzzer
	GPIOB_PDDR  |=  BUZZER_MASK;

	PortInitialise();//Initialise port

	// Configure PWM
	initialiseFTM0(PWM_PERIOD);
	initialiseFTM0_PWM(PWM_PERIOD);

	for(;;) {
		ConfigureDutyCycle(3);//Use Channel 3, PTC4 - FTM0_CH3
```

```
            printf("Water level is at = %d cm\n",getWaterLevelHeight());//Display the
water level height at the console

            //0-10% --> Display first blue rectangular bar
            //10-30% --> Display second blue rectangular bar
            //30-50% --> Display first green rectangular bar
            //50-70% --> Display second green rectangular bar
            //70-90% --> Display yellow rectangular bar
            //90-100% --> Display red rectangular bar

            if((getWaterLevelHeight()) < 0.1*CONTAINER_HEIGHT)//If water level is below
10% level
            {
                  lcd_clear(BACKGROUND_COLOUR);
                  lcd_drawRect(30,30,100,40,BLUE,BLUE);//Only display the lower blue
rectangular bar
                  lcd_drawRect(30,40,100,50,BLACK,BLACK);//BLACK indicates that the
bars are not displayed because the background colour is black
                  lcd_drawRect(30,50,100,60,BLACK,BLACK);
                  lcd_drawRect(30,60,100,70,BLACK,BLACK);
                  lcd_drawRect(30,70,100,80,BLACK,BLACK);
                  lcd_drawRect(30,80,100,90,BLACK,BLACK);
                  lcd_putStr("TOO LOW!!!!!", 10, 14, FontMedium, BLUE, BLACK);
                  printf("Percentage is %d cm\n",waterlevel_percentage);
            }

            //If water is between 10% and 30% level
            if((getWaterLevelHeight()) < 0.3*CONTAINER_HEIGHT &&
(getWaterLevelHeight()) >= 0.1*CONTAINER_HEIGHT)
            {
                  lcd_clear(BACKGROUND_COLOUR);
                  lcd_drawRect(30,30,100,40,BLUE,BLUE);
                  lcd_drawRect(30,40,100,50,BLUE,BLUE);//Display the first two lower
rectangular bars
                  lcd_drawRect(30,50,100,60,BLACK,BLACK);
                  lcd_drawRect(30,60,100,70,BLACK,BLACK);
                  lcd_drawRect(30,70,100,80,BLACK,BLACK);
                  lcd_drawRect(30,80,100,90,BLACK,BLACK);
                  lcd_putStr("TOO LOW!!!!!", 10, 14, FontMedium, BLUE, BLACK);
                  printf("Percentage is %d cm\n",waterlevel_percentage);
            }

            //If water is between 30% and 50% level
            if((getWaterLevelHeight()) < 0.5*CONTAINER_HEIGHT &&
(getWaterLevelHeight()) >= 0.3*CONTAINER_HEIGHT)
                  //If less than 20% of the water level
            {
                  lcd_clear(BACKGROUND_COLOUR);
                  lcd_drawRect(30,30,100,40,BLUE,BLUE);
                  lcd_drawRect(30,40,100,50,BLUE,BLUE);
                  lcd_drawRect(30,50,100,60,GREEN,GREEN);//Display the first three
rectangular bars
                  lcd_drawRect(30,60,100,70,BLACK,BLACK);
                  lcd_drawRect(30,70,100,80,BLACK,BLACK);
                  lcd_drawRect(30,80,100,90,BLACK,BLACK);
                  lcd_putStr("NORMAL LEVEL", 10, 14, FontMedium, GREEN, BLACK);
                  printf("Percentage is %d cm\n",waterlevel_percentage);
            }

            //If water is between 50% and 70% level
            if((getWaterLevelHeight()) < 0.7*CONTAINER_HEIGHT &&
(getWaterLevelHeight()) >= 0.5*CONTAINER_HEIGHT)
            {
                  lcd_clear(BACKGROUND_COLOUR);
                  lcd_drawRect(30,30,100,40,BLUE,BLUE);
                  lcd_drawRect(30,40,100,50,BLUE,BLUE);
                  lcd_drawRect(30,50,100,60,GREEN,GREEN);
```

```
			lcd_drawRect(30,60,100,70,GREEN,GREEN);//Display the first four 
rectangular bars
			lcd_drawRect(30,70,100,80,BLACK,BLACK);
			lcd_drawRect(30,80,100,90,BLACK,BLACK);
			lcd_putStr("NORMAL LEVEL", 10, 14, FontMedium, GREEN, 
BLACK);//Display too low text with blue text
			printf("Percentage is %d cm\n",waterlevel_percentage);
		}

		//If water is between 70% and 90% level
		if((getWaterLevelHeight()) < 0.9*CONTAINER_HEIGHT && 
(getWaterLevelHeight()) >= 0.7*CONTAINER_HEIGHT)
		{
			playTone(2000,0.1);//Trigger buzzer
			lcd_clear(BACKGROUND_COLOUR);
			lcd_drawRect(30,30,100,40,BLUE,BLUE);
			lcd_drawRect(30,40,100,50,BLUE,BLUE);
			lcd_drawRect(30,50,100,60,GREEN,GREEN);
			lcd_drawRect(30,60,100,70,GREEN,GREEN);
			lcd_drawRect(30,70,100,80,YELLOW,YELLOW);//Display the first five 
rectangular bars
			lcd_drawRect(30,80,100,90,BLACK,BLACK);
			lcd_putStr("WARNING!!!!!", 10, 14, FontMedium, RED, BLACK);//Display 
too low text with blue text
			printf("Percentage is %d cm\n",waterlevel_percentage);
		}

		if((getWaterLevelHeight()) <= CONTAINER_HEIGHT && 
(getWaterLevelHeight()) >= 0.9*CONTAINER_HEIGHT)
			//If less than 20% of the water level
		{
			playTone(2000,0.1);//Trigger buzzer
			lcd_drawRect(30,30,100,40,BLUE,BLUE);
			lcd_drawRect(30,40,100,50,BLUE,BLUE);
			lcd_drawRect(30,50,100,60,GREEN,GREEN);
			lcd_drawRect(30,60,100,70,GREEN,GREEN);
			lcd_drawRect(30,70,100,80,YELLOW,YELLOW);
			lcd_drawRect(30,80,100,90,RED,RED);//Display all the rectangular bars
			lcd_putStr("WARNING!!!!!", 10, 14, FontMedium, RED, BLACK);//Display 
too low text with blue text
			printf("Percentage is %d cm\n",waterlevel_percentage);
		}
	}
	return 0;
}
```

Prototype Diagrams:

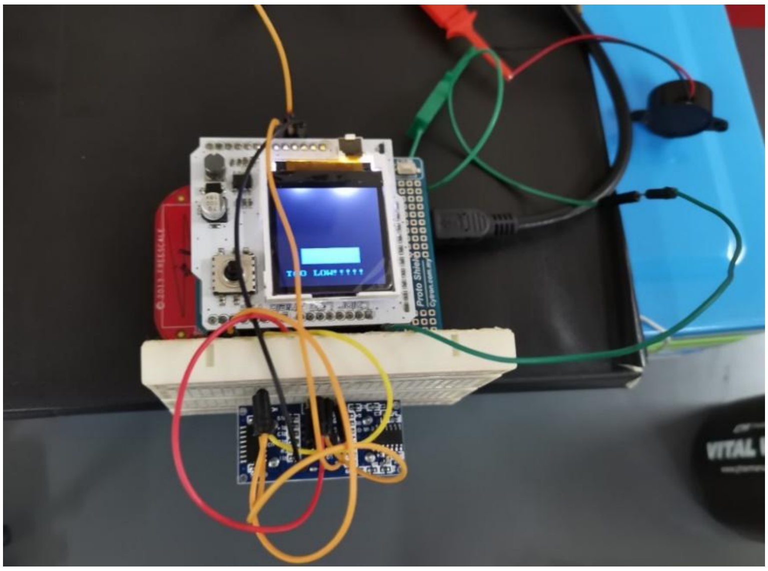


Figure 8(a): LCD interface with the sensor attached

Figure 8(b): Set up of the water level monitoring prototype system