CSE211 Introduction to Embedded Systems

Team 3

AIN SHAMS UNIVERSITY
I-Credit Hours Engineering
Programs (i.CHEP)

University of East London

Major Task

CSE211 Introduction to Embedded Systems

Submitted to:

Prof. Dr. Sherif Hammad

Eng/ Hesham Salah

Eng/ Mohamed Tarek

By:

Team 3

ID	Name	Contribution %
19P4401	**Mohamed Fathi Abdelhamid Ali**	**20**
19P5553	Abdel Rahman Ahmed Abdel Aziz Mahmoud Habash	20
19P6218	Mohamed Ahmed Mohamed Abdelhalim Mohanna	20
19P4007	Ahmed Hisham Fathy Hassabou	20
194907	Mahmoud Ahmed Mahmoud Elsheikh	20

Mechatronics and Automation Program

Table of Contents

List of Figures

1.Introduction

This system has 3 modes of operation (Calculator, Stopwatch, Timer) with allowance of Switching modes at any moment. The Stopwatch mode starts automatically and is supported with 3 pushbuttons:

- 1 for pausing the stopwatch
- 1 for resuming the stopwatch
- 1 for resetting the stopwatch

On The other hand, the timer mode once open waits for capturing input from the keypad and then pressing '='. It supports up till 59:59 counting down, once counting is done, the red led in the hardware will light up. Finally, the calculator mode supports 4 mathematical operations (Adding, Subtracting, Dividing, Multiplication). It takes the first number from the user and waits for the operator to be pressed then the second number is taken, and the result is printed and keeps available for more calculations. The following link contains all the needed work related to the project: https://drive.google.com/drive/folders/1Av6pO7-uBwBBLLsxumZ2BBPrWy_NswfZ?usp=sharing

2.Circuit Topology

2.1. Design On Proteus

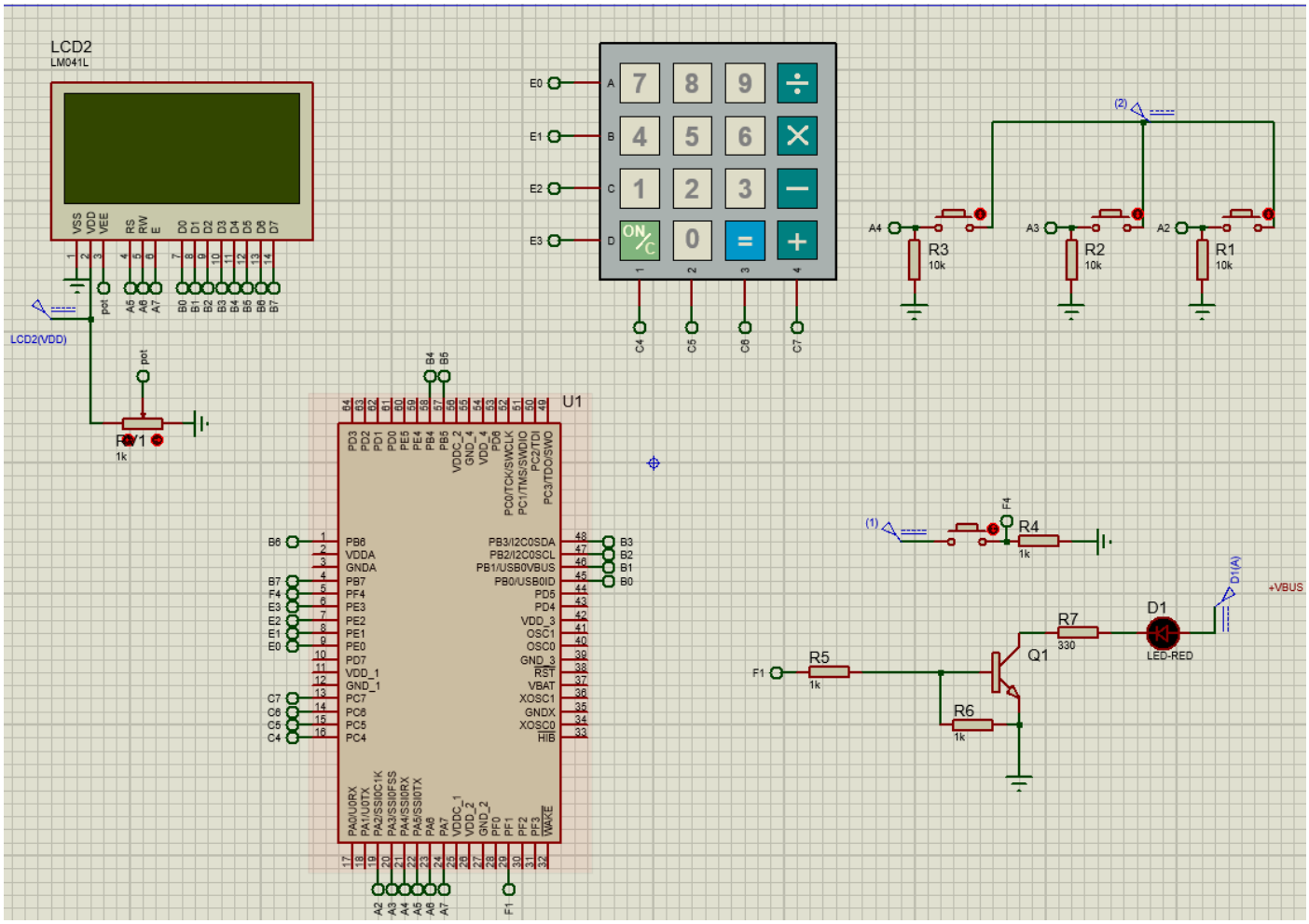


Figure 1 Circuit Design

2.2. Connections

Regarding the **LCD:**

- Pins **D0 to D7** are connected to portB pins 0 to 7
- **RS, Rw & E** are connected to portA pins 5 to 7
- **VSS** to ground
- **VDD** to +5V
- **VEE** to the middle leg of a potentiometer in order to adjust the back lighting

Regarding the **Keypad:**

- **Rows** to PortE pins 0 to 3
- **Columns** to PortC pins 4 t0 7

Stopwatch Buttons:

- **Pause** to A2
- **Resume** to A3
- **Reset** to A4

Modes Button to F4

2.3. Notes

1) All the push buttons are connected to a pull-down resistor then ground from a terminal and the other terminal is connected to the +5V. Where the node between the resistor and pull-down resistor is the -ve pin. When the button is not pressed the input voltage is zero and the digital pin verifies that the status is low (0). One the other hand when the button is pressed input voltage is high and the digital pin verifies that the status is high (1).

2) The LED connection is taken form the microcontroller datasheet

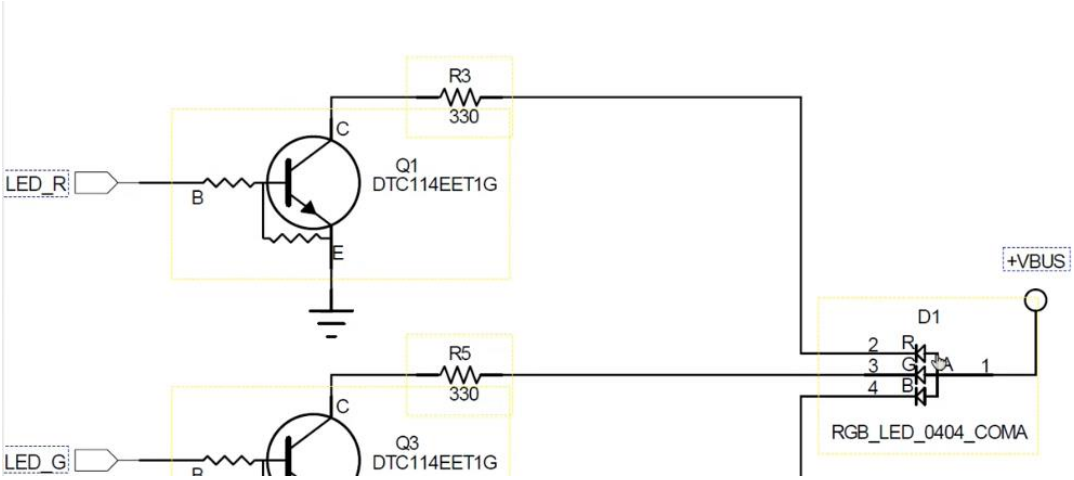


Figure 2 Red_LED connection (data sheet)

3.Flowchart

3.1. Overview

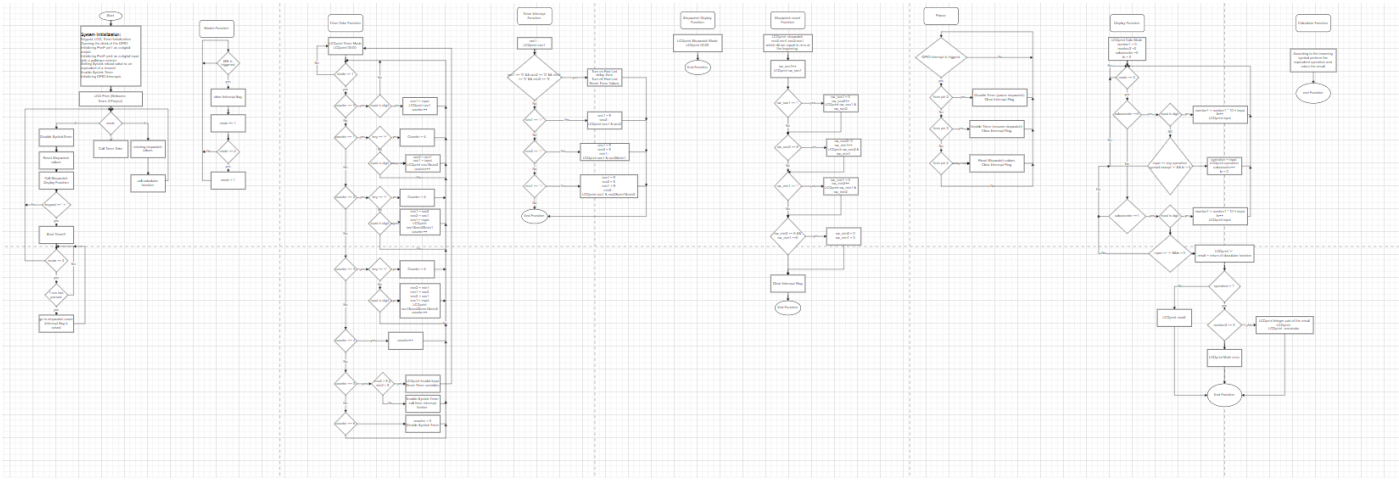


Figure 3 Whole System Flowchart

This is the entire system flow chart. The system consists of several functions all under the main function.

Note: Due to the size of the flowchart some functions will not be completely visible if was place as a whole so it will be displayed in fragments

3.2. Main

This is the main functions that decides which mode will be operated depending on the current mode

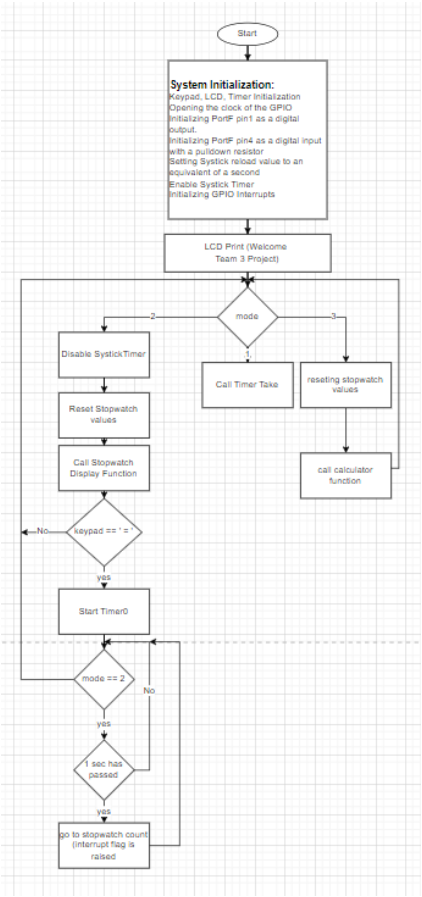


Figure 4 Main Function

3.2.1. System Initialization

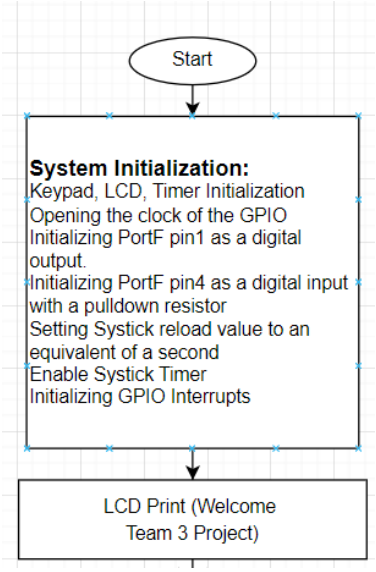


Figure 5 System Initialization

3.2.2. Mode Selection

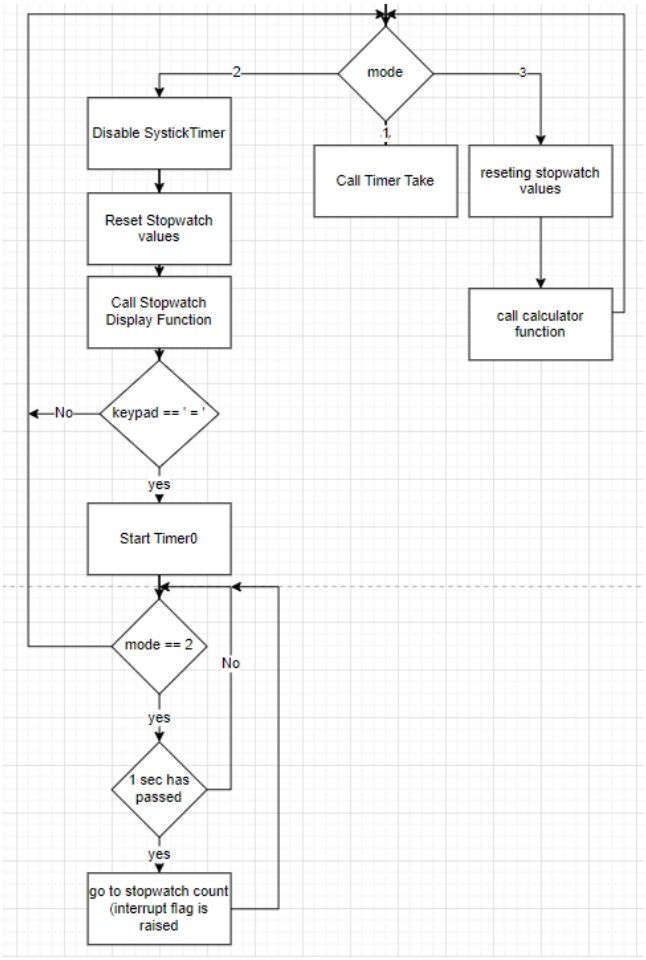


Figure 6 Mode Selection

3.3. Modes Function

Each time the mode push button is pressed an interrupts occur causing the modes function to increment a variable named mode by one. If the mode is equal to 4 it is then returned to be equal to 1

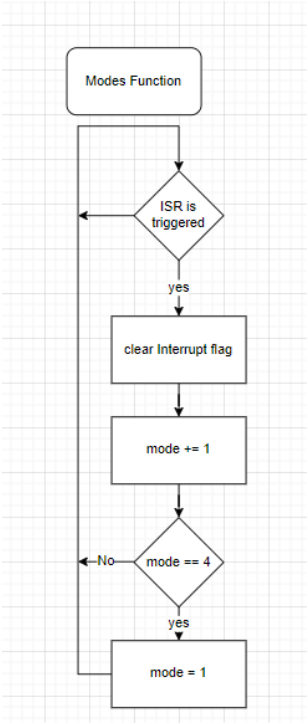


Figure 7 Mode Function

3.4. Timer

The timer mode in our system consists of two function one which is called by the main (Timer Take) and the other is called when an interrupt is caused by Timer Take

3.4.1. Timer Take Function

This function displays the values of the timer during the initialization of the timer and the after every second. Depending on a variable named counter an action will occur

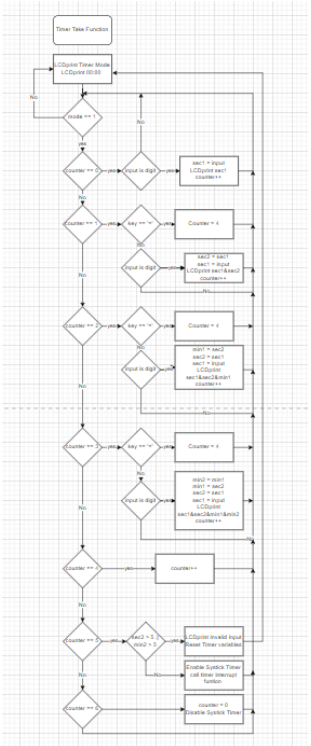


Figure 8 Timer Take Function

Counter 0, 1 & 2 Actions (this for setting the starting values of the timer)

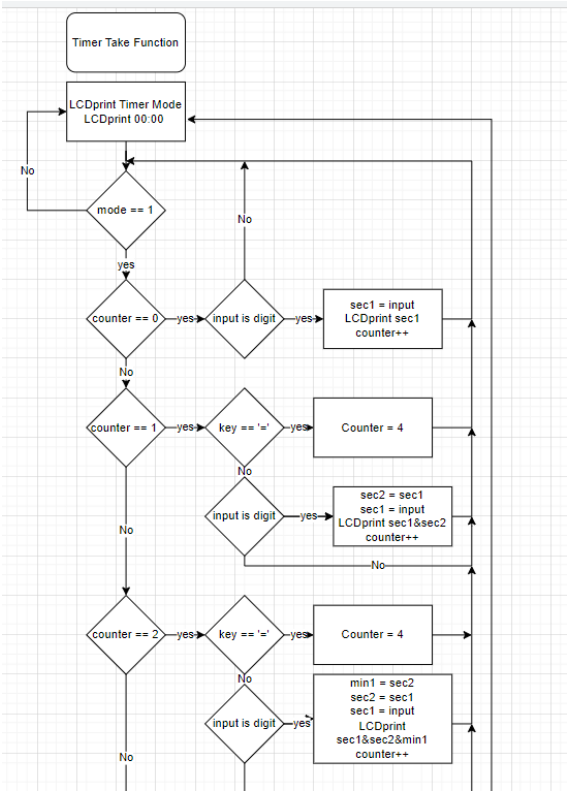


Figure 9 Counter 0, 1 & 2

Counter 3, 4, 5 & 6 Actions (for setting the min2 values and starting or decrementing or resetting of the timer)

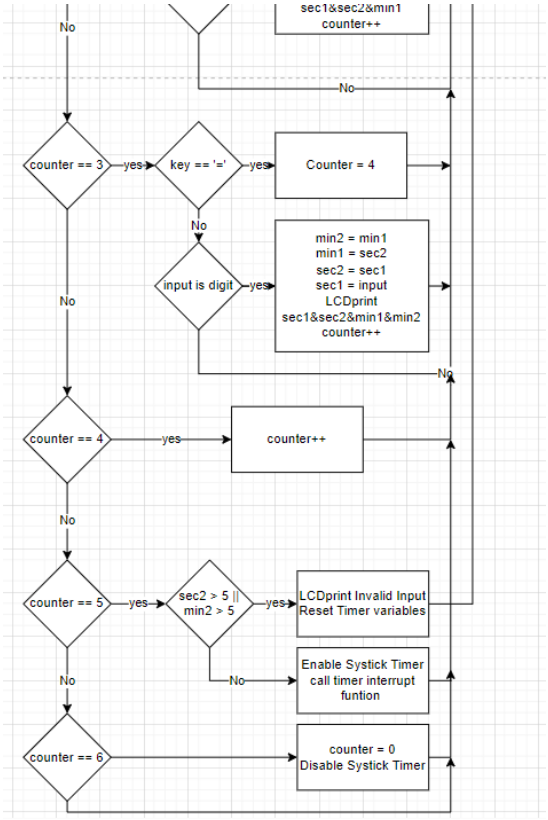


Figure 10 Counter 3, 4, 5 & 6

3.4.2. Timer Interrupt Function

This function occurs if the counter in the timer take function is equal to 5 where it will alter the values of the timer displayed if a certain condition is met.

Note: character '/' is equivalent to number 0 in ASCII.

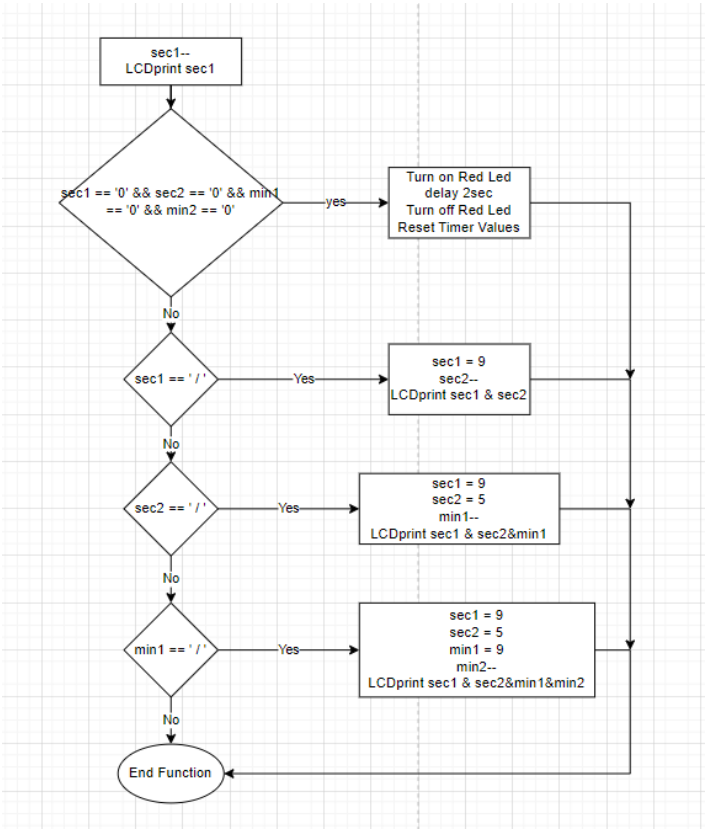


Figure 11 Timer Interrupt Function

3.5. Stopwatch

The stopwatch is mainly controlled by main function where is calls at first for a function called stopwatch display in order to display the stopwatch starting value then after every second it passes it raises an interrupt flag in order to execute a function called stopwatch count. In addition, the stopwatch also is controlled via a pause function.

3.5.1. Stopwatch Display Function

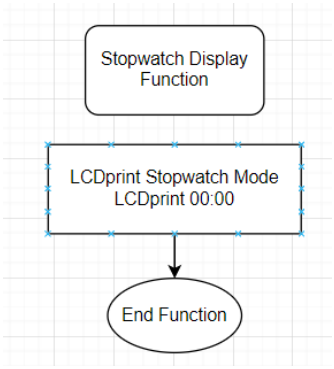


Figure 12 Stopwatch Dispaly Function

3.5.2. Stopwatch Count

This function Increments and alter the values of the stopwatch form a certain condition.

Note: ':' is equivalent to 10 in ASCII

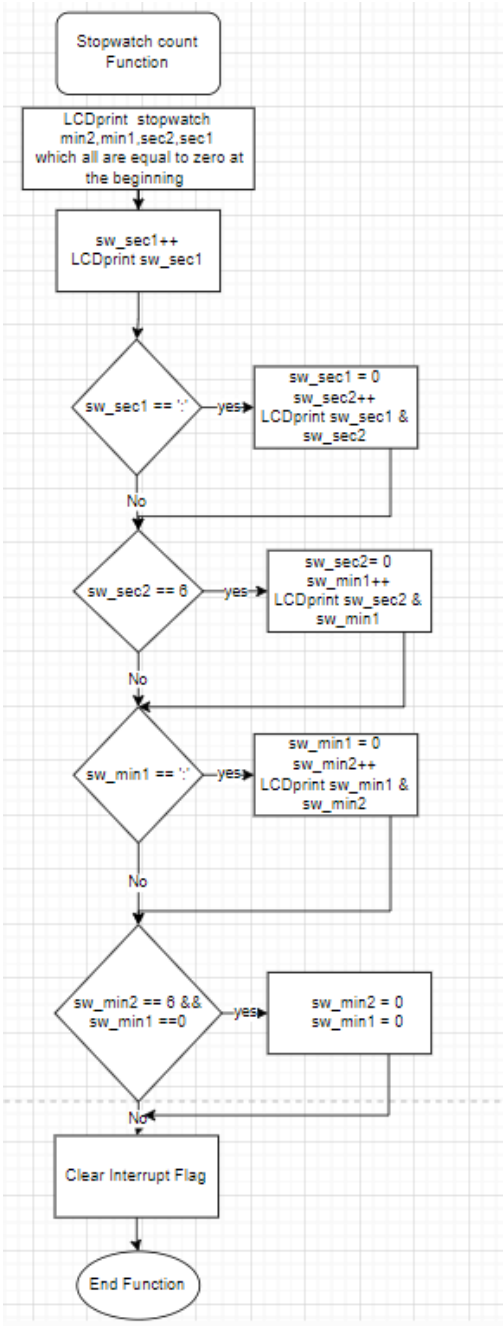


Figure 13 Stopwatch count function

3.5.3. Pause Function

This function occurs when one of three push buttons is pressed as interrupts occurs. Depending on which button is pressed the stopwatch will either pause, resume or reset.

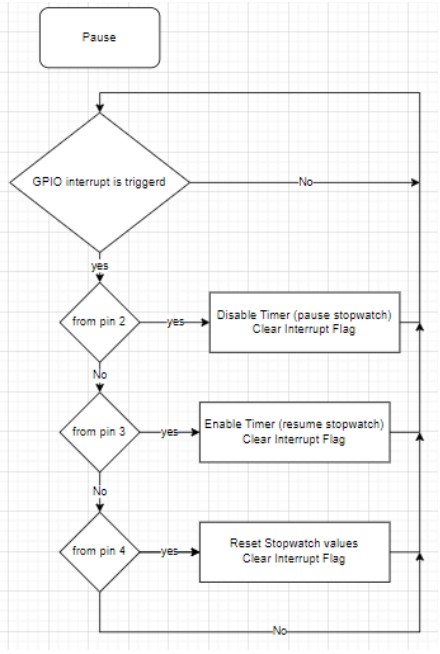


Figure 14 Pause Function

3.6. Calculator

This mode has two functions one to display the calculator parameters and the other to perform the calculator operations

3.6.1. Calculator Display

It displays the input numbers along with the result.

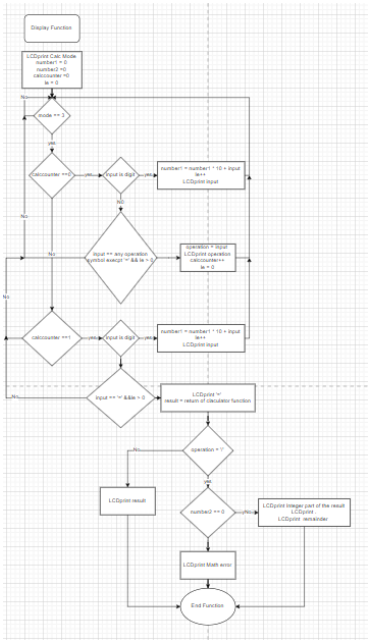


Figure 15 Calculator Display

Displaying the input:

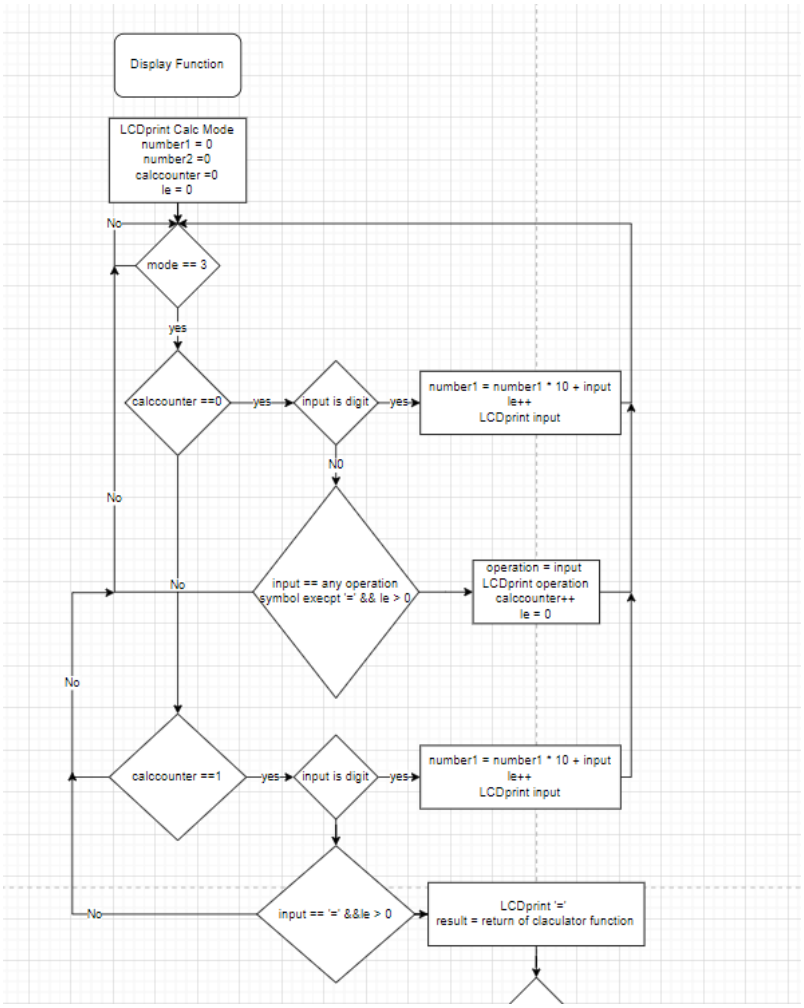


Figure 16 Displaying The input

Displaying the output:

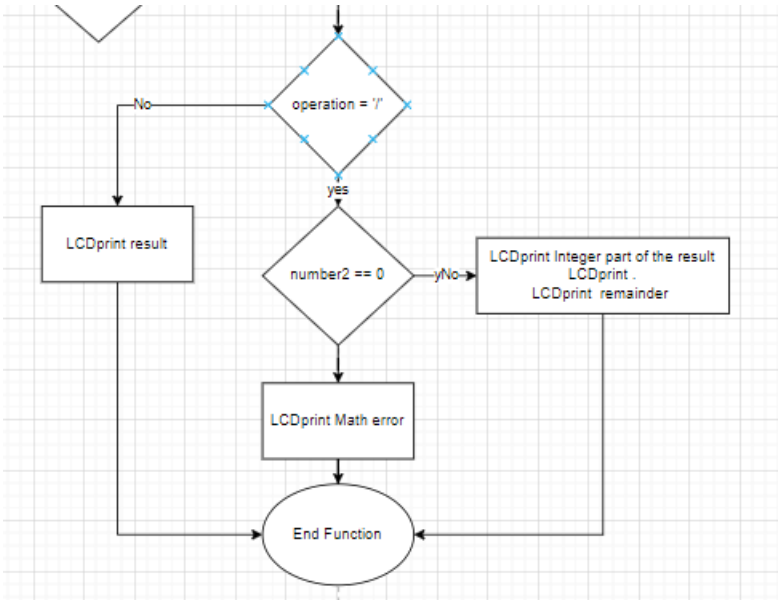


Figure 17 Displaying the output

4.Drivers

4.1. Types.h

```c
/******************************************************************************
 *
 * Module: STANDARD - TYPES
 *
 * File Name: types.h
 *
 * Author: Team 3
 *
 * Description: Variable number of bits data types
 *
 *******************************************************************************/

#ifndef types
#define types
typedef unsigned char          uint8;
typedef signed char            int8;
typedef unsigned short         uint16;
typedef signed short           int16;
typedef unsigned long          uint32;
typedef signed long            int32;
typedef unsigned long long     uint64;
typedef signed long long       int64;
typedef float                  float32;
typedef double                 float64;

typedef unsigned char*          uint8_ptr;
typedef signed char*            int8_ptr;
typedef unsigned short*         uint16_ptr;
typedef signed short*           int16_ptr;
typedef unsigned long*          uint32_ptr;
typedef signed long*            int32_ptr;
typedef unsigned long long*     uint64_ptr;
typedef signed long long*       int64_ptr;
typedef float*                  float32_ptr;
typedef double*                 float64_ptr;

#endif
```

Figure 18 Types.h

4.2. Keypad

4.2.1. Header File

```c
/******************************************************************************
 *
 * Module:  KEYPAD
 *
 * File Name: keypad.h
 *
 * Author: Team 3
 *
 * Description: Driver to use the keypad from the HAL Layer
 *
 *******************************************************************************/

#ifndef KEYPAD_H_
#define KEYPAD_H_

#include "DIO.h"
#include "types.h"
#include "tm4c123gh6pm.h"
#include "LCD.h"
#define KEYPAD_NUM_COLS                 4
#define KEYPAD_NUM_ROWS                 4

void keypad_init(void);
/*
  Initialize the keypad
*/
uint8 keypad_read(void);
/*
  Function that detects the pressed button by setting all rows to HIGH
  then checking if any column matches HIGH with rows
  then the intersection is the button pressed
*/


```

Figure 19 Keypad.h

4.2.2. C file

```c
/*****************************************************************************
 *
 * Module:  KEYPAD
 *
 * File Name: keypad.c
 *
 * Author: Team 3
 *
 * Description: Driver to use the keypad from the HAL layer
 *
 *****************************************************************************/
#include "keypad.h"
extern uint16 mode;
extern uint16 counter;

// mapping of the keypad to match the required behavior from hardware
static const uint8 mapping[4][4] = { {'1', '2', '3', '+'},
                                     {'4', '5', '6', '-'},
                                     {'7', '8', '9', '/'},
                                     {'*', '0', '#', '='} };

void keypad_init(){
  SYSCTL_RCGCGPIO_R |= 0x14;                          //enable clc for port C & E
  while ((SYSCTL_RCGCGPIO_R&0x14)==0);                //wait for clock to be enabled
  GPIO_PORTC_CR_R  |= 0xF0;                           //allow changes to all the bits in port C
  GPIO_PORTE_CR_R  |= 0x0F;                           //allow changes to all the bits in port E
  GPIO_PORTC_DIR_R |= 0xF0;                           //set directions cols are o/ps
  GPIO_PORTE_DIR_R |= 0x00;                           //set directions rows are i/ps
  GPIO_PORTE_PDR_R |= 0x0F;                           //pull down resistor on Rows
  GPIO_PORTC_DEN_R |= 0xF0;                           //digital enable pins in port C
  GPIO_PORTE_DEN_R |= 0x0F;                           //digital enable pins in port E

}
uint8 keypad_read(void)
{
  uint16 m1=mode;
  uint16 m2=counter;
  while(m1==mode && m2==counter)
  {
    uint8 i,j;
    for(i = 0; i < 4; i++)                            //columns traverse
    {
      GPIO_PORTC_DATA_R = (1<<(i+4));
      delayUs(2);
      for(j = 0; j < 4; j++)                          //rows traverse
      {
        if((GPIO_PORTE_DATA_R & 0x0F) & (1<<(j)))
        {
          delayMs(50);
          return mapping[j][i];
        }
      }
    }
  }
  return 't';
}
```

Figure 20 Keypad.c

4.3. LCD

4.3.1.Header File

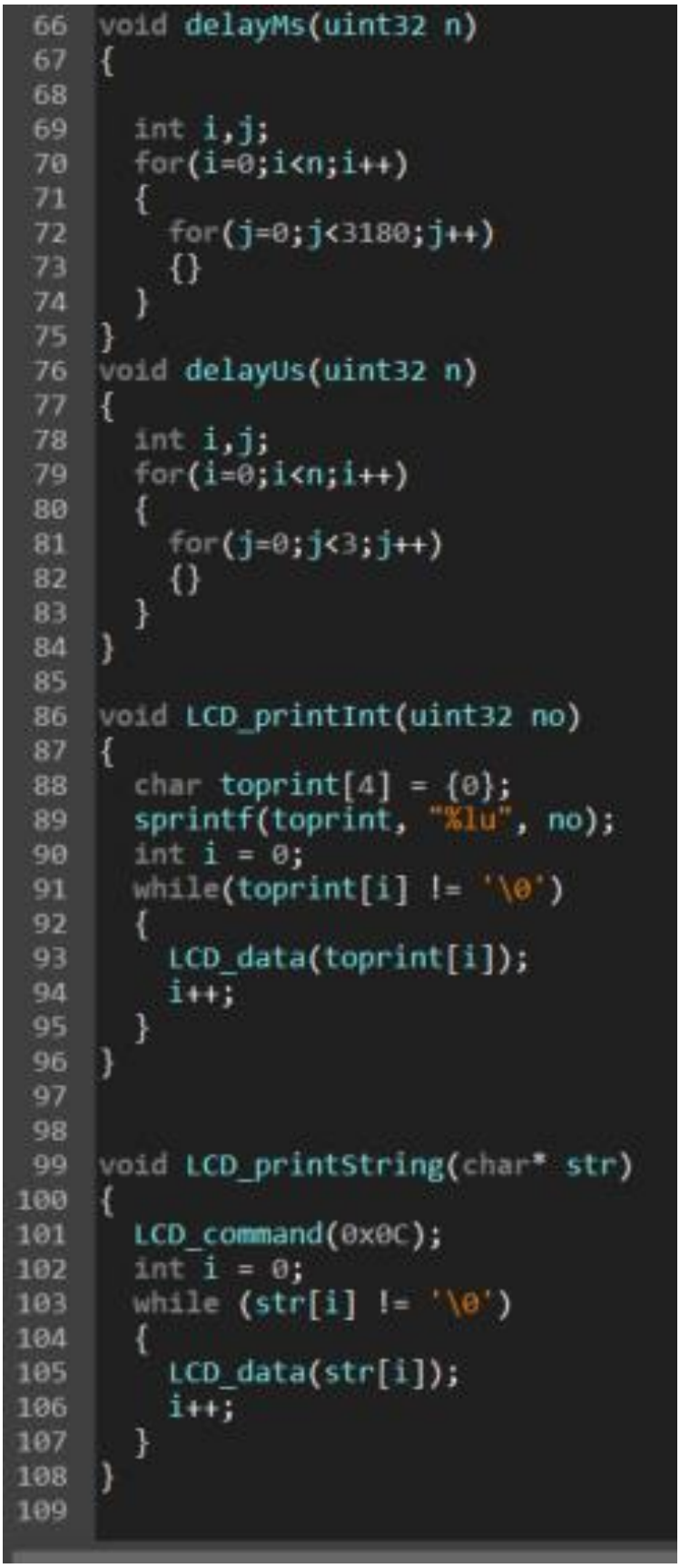
```
/*****************************************************************************
 *
 * Module: LCD
 *
 * File Name: LCD.h
 *
 * Author: Team 3 Referenced from Mazidi Textbook
 *
 * Description: header file of the LCD driver
 *
 *****************************************************************************/

#include "types.h"
#include "tm4c123gh6pm.h"
#include <stdio.h>
#include <stdlib.h>

#define RS 0x20                         // PORTA BIT5 mask (Reset)
#define RW 0x40                         // PORTA BIT6 mask (Read/Write)
#define EN 0x80                         // PORTA BIT7 mask (Enable)

void delayMs(uint32 n);
/*
  Delay in Milliseconds
  Parameter: n milliseconds
*/
void delayUs(uint32 n);
/*
  Delay in Microseconds
  Parameter: n microseconds
*/
void LCD_command (uint8 command) ;
/*
  To Trigger certain operations in the LCD
  commands: 1  -->  Clear screen
          0x80 -->  sets the cursor at first col first row and increment by itself
          0xC0 -->  sets the cursor at first col second row and increment by itself
*/
void LCD_data(uint8 data);
/*
  Displays a character on the LCD
  parameter: The ASCII code of the character to be printed
*/
void LCD_printInt(uint32 no);
/*
  Displays an Ingteger on the LCD
  parameter: The integer to be printed itself
*/
void LCD_printString(char* str);
/*
  Displays a String on the LCD
  parameter: The string to be printed
*/
void LCD_init (void);
/*
  initializes the LCD
*/
```

Figure 21 LCD.h

4.3.2. LCD.c

```c
/**************************************************************************************
 *
 * Module: LCD
 *
 * File Name: LCD.c
 *
 * Author: Team 3 Referenced from Mazidi Textbook
 *
 * Description: Source file of the LCD driver
 *
 **************************************************************************************/

#include "LCD.h"

void LCD_init (void)
{

  SYSCTL_RCGCGPIO_R |= 0x01;

  SYSCTL_RCGCGPIO_R |=0x02;
  GPIO_PORTA_DIR_R |= 0xE0;
  GPIO_PORTA_DEN_R |= 0xE0;
  GPIO_PORTB_DIR_R  = 0xFF;
  GPIO_PORTB_DEN_R  = 0xFF;

  delayMs (100);
  LCD_command (0x30) ;
  delayMs (150);
  LCD_command (0x30) ;
  delayUs (100);
  LCD_command (0x30);
  LCD_command (0x38) ;
  LCD_command (0x06) ;
  LCD_command (0x01) ;
  LCD_command (0x0F) ;

}

void LCD_command (uint8 command)
{
  GPIO_PORTA_DATA_R=0;
  GPIO_PORTB_DATA_R=command;
  GPIO_PORTA_DATA_R=EN;
  delayMs(10);
  GPIO_PORTA_DATA_R=0;
  if (command<4)
  {
    delayMs(5);
  }
  else
  {
    delayMs(5);
  }
}

void LCD_data(uint8 data)
{
  GPIO_PORTA_DATA_R=RS;
  GPIO_PORTB_DATA_R=data;
  GPIO_PORTA_DATA_R=(EN | RS) ;
  delayMs(10);
  GPIO_PORTA_DATA_R=0;
  delayMs(10);
}

```

Figure 22 LCD.c Part 1



```
66  void delayMs(uint32 n)
67  {
68
69    int i,j;
70    for(i=0;i<n;i++)
71    {
72      for(j=0;j<3180;j++)
73      {}
74    }
75  }
76  void delayUs(uint32 n)
77  {
78    int i,j;
79    for(i=0;i<n;i++)
80    {
81      for(j=0;j<3;j++)
82      {}
83    }
84  }
85
86  void LCD_printInt(uint32 no)
87  {
88    char toprint[4] = {0};
89    sprintf(toprint, "%lu", no);
90    int i = 0;
91    while(toprint[i] != '\0')
92    {
93      LCD_data(toprint[i]);
94      i++;
95    }
96  }
97
98
99  void LCD_printString(char* str)
100 {
101   LCD_command(0x0C);
102   int i = 0;
103   while (str[i] != '\0')
104   {
105     LCD_data(str[i]);
106     i++;
107   }
108 }
109
```

Figure 23 LCD.c Part 2

4.4. Timer

4.4.1. Header File

```c
/*****************************************************************************
 *
 * Module: Timer
 *
 * File Name: Timer.h
 *
 * Author: Team 3
 *
 * Description: General Purpose Timer Module
 *
 *****************************************************************************/

#ifndef TIMER_H
#define TIMER_H


#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "types.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "tm4c123gh6pm.h"
#include "bitwise_operation.h"
#include "systick.h"
#include "driverlib/interrupt.h"
#include "driverlib/timer.h"

void timer_init(void);
/*
  Function that initializes the timer module to be ready for starting
*/
void timer_enable(void);
/*
  fires the timer
*/
#endif
```

Figure 24 Timer.h

4.4.2. C file

```c
/*****************************************************************************
 *
 * Module: Timer
 *
 * File Name: Timer.h
 *
 * Author: Team 3
 *
 * Description: General Purpose Timer Module
 *
 *****************************************************************************/

#include "Timer.h"

void timer_init(void)
{

 SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);          // clock enable on timer 0
 while(!(SysCtlPeripheralReady(SYSCTL_PERIPH_TIMER0))); // wait until clock is enable
 TimerDisable (TIMER0_BASE,TIMER_BOTH);                 // disable timer
 TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);       // choose timer working configuration
 TimerLoadSet(TIMER0_BASE, TIMER_A, 16000000-1);        // choose Timer Reg A and set reload value

}

void timer_enable(void)
{
 TimerIntEnable(TIMER0_BASE,TIMER_TIMA_TIMEOUT);        // enable the timer interrupts
 TimerEnable(TIMER0_BASE,TIMER_BOTH);                   // enable the timer itself using both register A & B,knowing that the reload value is set on Reg A
 IntEnable(INT_TIMER0A);
 IntMasterEnable();                                     // enable all interrupts
}
```

Figure 25 Timer.C

5. Tivaware APIs

This section contains all the information about all the tivaware APIs that were used in the whole Project

5.1. GPIO APIs

5.1.1.GPIOIntClear

Datasheet Description:

14.2.3.7 GPIOIntClear

Clears the specified interrupt sources.

Prototype:

```
void
GPIOIntClear(uint32_t ui32Port,
             uint32_t ui32IntFlags)
```

Parameters:

ui32Port is the base address of the GPIO port.

ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

Clears the interrupt for the specified interrupt source(s).

The *ui32IntFlags* parameter is the logical OR of the **GPIO_INT_*** values.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

Figure 26 GPIOIntClear Datasheet

Src:

```
//*****************************************************************************
//
//! Clears the specified interrupt sources.
//!
//! \param ui32Port is the base address of the GPIO port.
//! \param ui32IntFlags is the bit mask of the interrupt sources to disable.
//!
//! Clears the interrupt for the specified interrupt source(s).
//!
//! The \e ui32IntFlags parameter is the logical OR of the \b GPIO_INT_*
//! values.
//!
//! \note Because there is a write buffer in the Cortex-M processor, it may
//! take several clock cycles before the interrupt source is actually cleared.
//! Therefore, it is recommended that the interrupt source be cleared early in
//! the interrupt handler (as opposed to the very last action) to avoid
//! returning from the interrupt handler before the interrupt source is
//! actually cleared.  Failure to do so may result in the interrupt handler
//! being immediately reentered (because the interrupt controller still sees
//! the interrupt source asserted).
//!
//! \return None.
//
//*****************************************************************************
void
GPIOIntClear(uint32_t ui32Port, uint32_t ui32IntFlags)
{
    //
    // Check the arguments.
    //
    ASSERT(_GPIOBaseValid(ui32Port));

    //
    // Clear the interrupts.
    //
    HWREG(ui32Port + GPIO_O_ICR) = ui32IntFlags;
}
```

Figure 27 GPIOIntClear Src

5.1.2. GPIOIntDisbale

Datasheet Description:

14.2.3.8 GPIOIntDisable

Disables the specified GPIO interrupts.

Prototype:

```
void
GPIOIntDisable(uint32_t ui32Port,
               uint32_t ui32IntFlags)
```

Parameters:

ui32Port is the base address of the GPIO port.
ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

This function disables the indicated GPIO interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **GPIO_INT_PIN_0** - interrupt due to activity on Pin 0.

- **GPIO_INT_PIN_1** - interrupt due to activity on Pin 1.
- **GPIO_INT_PIN_2** - interrupt due to activity on Pin 2.
- **GPIO_INT_PIN_3** - interrupt due to activity on Pin 3.
- **GPIO_INT_PIN_4** - interrupt due to activity on Pin 4.
- **GPIO_INT_PIN_5** - interrupt due to activity on Pin 5.
- **GPIO_INT_PIN_6** - interrupt due to activity on Pin 6.
- **GPIO_INT_PIN_7** - interrupt due to activity on Pin 7.
- **GPIO_INT_DMA** - interrupt due to DMA activity on this GPIO module.

Returns:

None.

Figure 28 GPIOIntDisbale Datasheet

Src:

```
//*****************************************************************************
//
//! Disables the specified GPIO interrupts.
//!
//! \param ui32Port is the base address of the GPIO port.
//! \param ui32IntFlags is the bit mask of the interrupt sources to disable.
//!
//! This function disables the indicated GPIO interrupt sources.  Only the
//! sources that are enabled can be reflected to the processor interrupt;
//! disabled sources have no effect on the processor.
//!
//! The \e ui32IntFlags parameter is the logical OR of any of the following:
//!
//! - \b GPIO_INT_PIN_0 - interrupt due to activity on Pin 0.
//! - \b GPIO_INT_PIN_1 - interrupt due to activity on Pin 1.
//! - \b GPIO_INT_PIN_2 - interrupt due to activity on Pin 2.
//! - \b GPIO_INT_PIN_3 - interrupt due to activity on Pin 3.
//! - \b GPIO_INT_PIN_4 - interrupt due to activity on Pin 4.
//! - \b GPIO_INT_PIN_5 - interrupt due to activity on Pin 5.
//! - \b GPIO_INT_PIN_6 - interrupt due to activity on Pin 6.
//! - \b GPIO_INT_PIN_7 - interrupt due to activity on Pin 7.
//! - \b GPIO_INT_DMA - interrupt due to DMA activity on this GPIO module.
//!
//! \return None.
//
//*****************************************************************************
void
GPIOIntDisable(uint32_t ui32Port, uint32_t ui32IntFlags)
{
    //
    // Check the arguments.
    //
    ASSERT(_GPIOBaseValid(ui32Port));

    //
    // Disable the interrupts.
    //
    HWREG(ui32Port + GPIO_O_IM) &= ~(ui32IntFlags);
}
```

Figure 29 GPIOIntDisbale src

5.1.3. GPIOIntEnable

Datasheet Description:

14.2.3.9 GPIOIntEnable

Enables the specified GPIO interrupts.

Prototype:

```
void
GPIOIntEnable(uint32_t ui32Port,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Port is the base address of the GPIO port.
ui32IntFlags is the bit mask of the interrupt sources to enable.

Description:

This function enables the indicated GPIO interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **GPIO_INT_PIN_0** - interrupt due to activity on Pin 0.
- **GPIO_INT_PIN_1** - interrupt due to activity on Pin 1.
- **GPIO_INT_PIN_2** - interrupt due to activity on Pin 2.
- **GPIO_INT_PIN_3** - interrupt due to activity on Pin 3.
- **GPIO_INT_PIN_4** - interrupt due to activity on Pin 4.
- **GPIO_INT_PIN_5** - interrupt due to activity on Pin 5.
- **GPIO_INT_PIN_6** - interrupt due to activity on Pin 6.
- **GPIO_INT_PIN_7** - interrupt due to activity on Pin 7.
- **GPIO_INT_DMA** - interrupt due to DMA activity on this GPIO module.

Note:

If this call is being used to enable summary interrupts on GPIO port P or Q (GPIOIntTypeSet() with GPIO_DISCRETE_INT not enabled), then all individual interrupts for these ports must be enabled in the GPIO module using GPIOIntEnable() and all but the interrupt for pin 0 must be disabled in the NVIC using the IntDisable() function. The summary interrupts for the ports are routed to the INT_GPIOP0 or INT_GPIOQ0 which must be enabled to handle the interrupt. If this is not done then any individual GPIO pin interrupts that are left enabled also trigger the individual interrupts.

Returns:

None.

Figure 30 GPIOIntEnable Datasheet

Src:

```
//*****************************************************************************
//
//! Enables the specified GPIO interrupts.
//!
//! \param ui32Port is the base address of the GPIO port.
//! \param ui32IntFlags is the bit mask of the interrupt sources to enable.
//!
//! This function enables the indicated GPIO interrupt sources.  Only the
//! sources that are enabled can be reflected to the processor interrupt;
//! disabled sources have no effect on the processor.
//!
//! The \e ui32IntFlags parameter is the logical OR of any of the following:
//!
//! - \b GPIO_INT_PIN_0 - interrupt due to activity on Pin 0.
//! - \b GPIO_INT_PIN_1 - interrupt due to activity on Pin 1.
//! - \b GPIO_INT_PIN_2 - interrupt due to activity on Pin 2.
//! - \b GPIO_INT_PIN_3 - interrupt due to activity on Pin 3.
//! - \b GPIO_INT_PIN_4 - interrupt due to activity on Pin 4.
//! - \b GPIO_INT_PIN_5 - interrupt due to activity on Pin 5.
//! - \b GPIO_INT_PIN_6 - interrupt due to activity on Pin 6.
//! - \b GPIO_INT_PIN_7 - interrupt due to activity on Pin 7.
//! - \b GPIO_INT_DMA - interrupt due to DMA activity on this GPIO module.
//!
//! \note If this call is being used to enable summary interrupts on GPIO port
//! P or Q (GPIOIntTypeSet() with GPIO_DISCRETE_INT not enabled), then all
//! individual interrupts for these ports must be enabled in the GPIO module
//! using GPIOIntEnable() and all but the interrupt for pin 0 must be disabled
//! in the NVIC using the IntDisable() function.  The summary interrupts for
//! the ports are routed to the INT_GPIOP0 or INT_GPIOQ0 which must be enabled
//! to handle the interrupt.  If this is not done then any individual GPIO pin
//! interrupts that are left enabled also trigger the individual interrupts.
//!
//! \return None.
//
//*****************************************************************************
void
GPIOIntEnable(uint32_t ui32Port, uint32_t ui32IntFlags)
{
    //
    // Check the arguments.
    //
    ASSERT(_GPIOBaseValid(ui32Port));

    //
    // Enable the interrupts.
    //
    HWREG(ui32Port + GPIO_O_IM) |= ui32IntFlags;
}
```

Figure 31 GPIOIntEnable Datasheet

5.1.4. GPIOIntRegister

Datasheet Description:

14.2.3.10 GPIOIntRegister

Registers an interrupt handler for a GPIO port.

Prototype:

```
void
GPIOIntRegister(uint32_t ui32Port,
                void (*pfnIntHandler)(void))
```

Parameters:

ui32Port is the base address of the GPIO port.

pfnIntHandler is a pointer to the GPIO port interrupt handling function.

Description:

This function ensures that the interrupt handler specified by *pfnIntHandler* is called when an interrupt is detected from the selected GPIO port. This function also enables the corresponding GPIO interrupt in the interrupt controller; individual pin interrupts and interrupt sources must be enabled with GPIOIntEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

Figure 32 GPIOIntRegister Datasheet

Src:

```
//*****************************************************************************
//
//! Registers an interrupt handler for a GPIO port.
//!
//! \param ui32Port is the base address of the GPIO port.
//! \param pfnIntHandler is a pointer to the GPIO port interrupt handling
//! function.
//!
//! This function ensures that the interrupt handler specified by
//! \e pfnIntHandler is called when an interrupt is detected from the selected
//! GPIO port.  This function also enables the corresponding GPIO interrupt
//! in the interrupt controller; individual pin interrupts and interrupt
//! sources must be enabled with GPIOIntEnable().
//!
//! \sa IntRegister() for important information about registering interrupt
//! handlers.
//!
//! \return None.
//
//*****************************************************************************
void
GPIOIntRegister(uint32_t ui32Port, void (*pfnIntHandler)(void))
{
    uint32_t ui32Int;

    //
    // Check the arguments.
    //
    ASSERT(_GPIOBaseValid(ui32Port));

    //
    // Get the interrupt number associated with the specified GPIO.
    //
    ui32Int = _GPIOIntNumberGet(ui32Port);

    ASSERT(ui32Int != 0);

    //
    // Register the interrupt handler.
    //
    IntRegister(ui32Int, pfnIntHandler);

    //
    // Enable the GPIO interrupt.
    //
    IntEnable(ui32Int);
}
```

Figure 33 GPIOIntRegister src

5.1.5. GPIOIntStatus

Datasheet Description:

14.2.3.12 GPIOIntStatus

Gets interrupt status for the specified GPIO port.

Prototype:

```
uint32_t
GPIOIntStatus(uint32_t ui32Port,
              bool bMasked)
```

Parameters:

ui32Port is the base address of the GPIO port.
bMasked specifies whether masked or raw interrupt status is returned.

Description:

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

Returns:

Returns the current interrupt status for the specified GPIO module. The value returned is the logical OR of the **GPIO_INT_*** values that are currently active.

Figure 34 GPIOIntStatus Datasheet

Src:

```
//*****************************************************************************
//
//! Gets interrupt status for the specified GPIO port.
//!
//! \param ui32Port is the base address of the GPIO port.
//! \param bMasked specifies whether masked or raw interrupt status is
//! returned.
//!
//! If \e bMasked is set as \b true, then the masked interrupt status is
//! returned; otherwise, the raw interrupt status is returned.
//!
//! \return Returns the current interrupt status for the specified GPIO module.
//! The value returned is the logical OR of the \b GPIO_INT_* values that are
//! currently active.
//
//*****************************************************************************
uint32_t
GPIOIntStatus(uint32_t ui32Port, bool bMasked)
{
    //
    // Check the arguments.
    //
    ASSERT(_GPIOBaseValid(ui32Port));

    //
    // Return the interrupt status.
    //
    if(bMasked)
    {
        return(HWREG(ui32Port + GPIO_O_MIS));
    }
    else
    {
        return(HWREG(ui32Port + GPIO_O_RIS));
    }
}
```

Figure 35 GPIOIntStatus Datasheet

5.1.5. GPIOIntTypeSet

Datasheet Description:

14.2.3.14 GPIOIntTypeSet

Sets the interrupt type for the specified pin(s).

Prototype:

```
void
GPIOIntTypeSet(uint32_t ui32Port,
               uint8_t ui8Pins,
               uint32_t ui32IntType)
```


Parameters:

ui32Port is the base address of the GPIO port.
ui8Pins is the bit-packed representation of the pin(s).
ui32IntType specifies the type of interrupt trigger mechanism.

Description:

This function sets up the various interrupt trigger mechanisms for the specified pin(s) on the selected GPIO port.

One of the following flags can be used to define the *ui32IntType* parameter:

- **GPIO_FALLING_EDGE** sets detection to edge and trigger to falling
- **GPIO_RISING_EDGE** sets detection to edge and trigger to rising
- **GPIO_BOTH_EDGES** sets detection to both edges
- **GPIO_LOW_LEVEL** sets detection to low level
- **GPIO_HIGH_LEVEL** sets detection to high level

In addition to the above flags, the following flag can be OR'd in to the *ui32IntType* parameter:

- **GPIO_DISCRETE_INT** sets discrete interrupts for each pin on a GPIO port.

The **GPIO_DISCRETE_INT** is not available on all devices or all GPIO ports, consult the data sheet to ensure that the device and the GPIO port supports discrete interrupts.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

In order to avoid any spurious interrupts, the user must ensure that the GPIO inputs remain stable for the duration of this function.

Returns:

None.

Figure 36 GPIOIntTypeSet Datasheet

Src:

```
//*****************************************************************************
//
//! Sets the interrupt type for the specified pin(s).
//!
//! \param ui32Port is the base address of the GPIO port.
//! \param ui8Pins is the bit-packed representation of the pin(s).
//! \param ui32IntType specifies the type of interrupt trigger mechanism.
//!
//! This function sets up the various interrupt trigger mechanisms for the
//! specified pin(s) on the selected GPIO port.
//!
//! One of the following flags can be used to define the \e ui32IntType
//! parameter:
//!
//! - \b GPIO_FALLING_EDGE sets detection to edge and trigger to falling
//! - \b GPIO_RISING_EDGE sets detection to edge and trigger to rising
//! - \b GPIO_BOTH_EDGES sets detection to both edges
//! - \b GPIO_LOW_LEVEL sets detection to low level
//! - \b GPIO_HIGH_LEVEL sets detection to high level
//!
//! In addition to the above flags, the following flag can be OR'd in to the
//! \e ui32IntType parameter:
//!
//! - \b GPIO_DISCRETE_INT sets discrete interrupts for each pin on a GPIO
//! port.
//!
//! The \b GPIO_DISCRETE_INT is not available on all devices or all GPIO ports,
//! consult the data sheet to ensure that the device and the GPIO port supports
//! discrete interrupts.
//!
//! The pin(s) are specified using a bit-packed byte, where each bit that is
//! set identifies the pin to be accessed, and where bit 0 of the byte
//! represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.
//!
//! \note In order to avoid any spurious interrupts, the user must ensure that
//! the GPIO inputs remain stable for the duration of this function.
//!
//! \return None.
//
//*****************************************************************************
```

```
void
GPIOIntTypeSet(uint32_t ui32Port, uint8_t ui8Pins,
               uint32_t ui32IntType)
{
    //
    // Check the arguments.
    //
    ASSERT(_GPIOBaseValid(ui32Port));
    ASSERT(((ui32IntType & 0xF) == GPIO_FALLING_EDGE) ||
           ((ui32IntType & 0xF) == GPIO_RISING_EDGE) ||
           ((ui32IntType & 0xF) == GPIO_BOTH_EDGES) ||
           ((ui32IntType & 0xF) == GPIO_LOW_LEVEL) ||
           ((ui32IntType & 0xF) == GPIO_HIGH_LEVEL));
    ASSERT(((ui32IntType & 0x000F0000) == 0) ||
           (((ui32IntType & 0x000F0000) == GPIO_DISCRETE_INT) &&
            ((ui32Port == GPIO_PORTP_BASE) || (ui32Port == GPIO_PORTQ_BASE))));

    //
    // Set the pin interrupt type.
    //
    HWREG(ui32Port + GPIO_O_IBE) = ((ui32IntType & 1) ?
                                    (HWREG(ui32Port + GPIO_O_IBE) | ui8Pins) :
                                    (HWREG(ui32Port + GPIO_O_IBE) & ~(ui8Pins)));
    HWREG(ui32Port + GPIO_O_IS) = ((ui32IntType & 2) ?
                                   (HWREG(ui32Port + GPIO_O_IS) | ui8Pins) :
                                   (HWREG(ui32Port + GPIO_O_IS) & ~(ui8Pins)));
    HWREG(ui32Port + GPIO_O_IEV) = ((ui32IntType & 4) ?
                                    (HWREG(ui32Port + GPIO_O_IEV) | ui8Pins) :
                                    (HWREG(ui32Port + GPIO_O_IEV) & ~(ui8Pins)));

    //
    // Set or clear the discrete interrupt feature.  This is not available
    // on all parts or ports but is safe to write in all cases.
    //
    HWREG(ui32Port + GPIO_O_SI) = ((ui32IntType & 0x10000) ?
                                   (HWREG(ui32Port + GPIO_O_SI) | 0x01) :
                                   (HWREG(ui32Port + GPIO_O_SI) & ~(0x01)));
}
```

Figure 37 GPIOIntTypeSet src

5.2.Systick

5.2.1. SysTickDisable

Datasheet Description:

28.2.2.1 SysTickDisable

Disables the SysTick counter.

Prototype:

```
void
SysTickDisable(void)
```

Description:

This function stops the SysTick counter. If an interrupt handler has been registered, it is not called until SysTick is restarted.

Returns:

None.

Figure 38 SysTickDisable Datasheet

Src:

```
//*****************************************************************************
//
//! Disables the SysTick counter.
//!
//! This function stops the SysTick counter.  If an interrupt handler has been
//! registered, it is not called until SysTick is restarted.
//!
//! \return None.
//
//*****************************************************************************
void
SysTickDisable(void)
{
    //
    // Disable SysTick.
    //
    HWREG(NVIC_ST_CTRL) &= ~(NVIC_ST_CTRL_ENABLE);
}
```

Figure 39 SysTickDisable src

5.2.2. SysTickEnable

Datasheet Description:

28.2.2.2 SysTickEnable

Enables the SysTick counter.

Prototype:

```
void
SysTickEnable(void)
```

Description:

This function starts the SysTick counter. If an interrupt handler has been registered, it is called when the SysTick counter rolls over.

Note:

Calling this function causes the SysTick counter to (re)commence counting from its current value. The counter is not automatically reloaded with the period as specified in a previous call to SysTickPeriodSet(). If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written to force the reload. Any write to this register clears the SysTick counter to 0 and causes a reload with the supplied period on the next clock.

Returns:

None.

Figure 40 SysTickEnable Datasheet

Src:

```
//*****************************************************************************
//
//! Enables the SysTick counter.
//!
//! This function starts the SysTick counter.  If an interrupt handler has been
//! registered, it is called when the SysTick counter rolls over.
//!
//! \note Calling this function causes the SysTick counter to (re)commence
//! counting from its current value.  The counter is not automatically reloaded
//! with the period as specified in a previous call to SysTickPeriodSet().  If
//! an immediate reload is required, the \b NVIC_ST_CURRENT register must be
//! written to force the reload.  Any write to this register clears the SysTick
//! counter to 0 and causes a reload with the supplied period on the next
//! clock.
//!
//! \return None.
//
//*****************************************************************************
void
SysTickEnable(void)
{
    //
    // Enable SysTick.
    //
    HWREG(NVIC_ST_CTRL) |= NVIC_ST_CTRL_CLK_SRC | NVIC_ST_CTRL_ENABLE;
}
```

Figure 41 SysTickEnable src

5.2.3. SysTickIntDisable

Datasheet Description:

28.2.2.3 SysTickIntDisable

Disables the SysTick interrupt.

Prototype:

```
void
SysTickIntDisable(void)
```

Description:

This function disables the SysTick interrupt, preventing it from being reflected to the processor.

Returns:

None.

Figure 42 SysTickIntDisable Datasheet

Src:

```
//*****************************************************************************
//
//! Disables the SysTick interrupt.
//!
//! This function disables the SysTick interrupt, preventing it from being
//! reflected to the processor.
//!
//! \return None.
//
//*****************************************************************************
void
SysTickIntDisable(void)
{
    //
    // Disable the SysTick interrupt.
    //
    HWREG(NVIC_ST_CTRL) &= ~(NVIC_ST_CTRL_INTEN);
}
```

Figure 43 SysTickIntDisable src

5.2.4. SysTickIntEnable

Datasheet Description:

28.2.2.4 SysTickIntEnable

Enables the SysTick interrupt.

Prototype:

```
void
SysTickIntEnable(void)
```

Description:

This function enables the SysTick interrupt, allowing it to be reflected to the processor.

Note:

The SysTick interrupt handler is not required to clear the SysTick interrupt source because it is cleared automatically by the NVIC when the interrupt handler is called.

Returns:

None.

Figure 44 SysTickIntEnable Datasheet

Src

```
//*****************************************************************************
//
//! Enables the SysTick interrupt.
//!
//! This function enables the SysTick interrupt, allowing it to be
//! reflected to the processor.
//!
//! \note The SysTick interrupt handler is not required to clear the SysTick
//! interrupt source because it is cleared automatically by the NVIC when the
//! interrupt handler is called.
//!
//! \return None.
//
//*****************************************************************************
void
SysTickIntEnable(void)
{
    //
    // Enable the SysTick interrupt.
    //
    HWREG(NVIC_ST_CTRL) |= NVIC_ST_CTRL_INTEN;
}
```

Figure 45 SysTickIntEnable src

5.2.5. SysTickIntRegister

Datasheet Description:

28.2.2.5 SysTickIntRegister

Registers an interrupt handler for the SysTick interrupt.

Prototype:

```
void
SysTickIntRegister(void (*pfnHandler)(void))
```

Parameters:

pfnHandler is a pointer to the function to be called when the SysTick interrupt occurs.

Description:

This function registers the handler to be called when a SysTick interrupt occurs.

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

Figure 46 SysTickIntRegister Datasheet

Src

```
//*****************************************************************************
//
//! Registers an interrupt handler for the SysTick interrupt.
//!
//! \param pfnHandler is a pointer to the function to be called when the
//! SysTick interrupt occurs.
//!
//! This function registers the handler to be called when a SysTick interrupt
//! occurs.
//!
//! \sa IntRegister() for important information about registering interrupt
//! handlers.
//!
//! \return None.
//
//*****************************************************************************
void
SysTickIntRegister(void (*pfnHandler)(void))
{
    //
    // Register the interrupt handler, returning an error if an error occurs.
    //
    IntRegister(FAULT_SYSTICK, pfnHandler);

    //
    // Enable the SysTick interrupt.
    //
    HWREG(NVIC_ST_CTRL) |= NVIC_ST_CTRL_INTEN;
}
```

Figure 47 SysTickIntRegister src

5.2.6. SysTickPeriodSet

Datasheet Description:

28.2.2.8 SysTickPeriodSet

Sets the period of the SysTick counter.

Prototype:

```
void
SysTickPeriodSet(uint32_t ui32Period)
```

Parameters:

ui32Period is the number of clock ticks in each period of the SysTick counter and must be between 1 and 16, 777, 216, inclusive.

Description:

This function sets the rate at which the SysTick counter wraps, which equates to the number of processor clocks between interrupts.

Note:

Calling this function does not cause the SysTick counter to reload immediately. If an immediate reload is required, the **NVIC_ST_CURRENT** register must be written. Any write to this register clears the SysTick counter to 0 and causes a reload with the *ui32Period* supplied here on the next clock after SysTick is enabled.

Returns:

None.

*Figure 48 SysTickPeriodSet **Datasheet***

Src:

```
//*****************************************************************************
//
//! Sets the period of the SysTick counter.
//!
//! \param ui32Period is the number of clock ticks in each period of the
//! SysTick counter and must be between 1 and 16,777,216, inclusive.
//!
//! This function sets the rate at which the SysTick counter wraps, which
//! equates to the number of processor clocks between interrupts.
//!
//! \note Calling this function does not cause the SysTick counter to reload
//! immediately.  If an immediate reload is required, the \b NVIC_ST_CURRENT
//! register must be written.  Any write to this register clears the SysTick
//! counter to 0 and causes a reload with the \e ui32Period supplied here on
//! the next clock after SysTick is enabled.
//!
//! \return None.
//
//*****************************************************************************
void
SysTickPeriodSet(uint32_t ui32Period)
{
    //
    // Check the arguments.
    //
    ASSERT((ui32Period > 0) && (ui32Period <= 16777216));

    //
    // Set the period of the SysTick counter.
    //
    HWREG(NVIC_ST_RELOAD) = ui32Period - 1;
}
```

Figure 49 SysTickPeriodSet src

5.3.Timer

5.3.1. TimerIntClear

Datasheet Description:

29.2.2.15 TimerIntClear

Clears timer interrupt sources.

Prototype:

```
void
TimerIntClear(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the timer module.
ui32IntFlags is a bit mask of the interrupt sources to be cleared.

Description:

The specified timer interrupt sources are cleared, so that they no longer assert. This function must be called in the interrupt handler to keep the interrupt from being triggered again immediately upon exit.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to TimerIntEnable().

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

*Figure 50 TimerIntClear **Datasheet***

Src

```
//*****************************************************************************
//
//! Clears timer interrupt sources.
//!
//! \param ui32Base is the base address of the timer module.
//! \param ui32IntFlags is a bit mask of the interrupt sources to be cleared.
//!
//! The specified timer interrupt sources are cleared, so that they no longer
//! assert.  This function must be called in the interrupt handler to keep the
//! interrupt from being triggered again immediately upon exit.
//!
//! The \e ui32IntFlags parameter has the same definition as the
//! \e ui32IntFlags parameter to TimerIntEnable().
//!
//! \note Because there is a write buffer in the Cortex-M processor, it may
//! take several clock cycles before the interrupt source is actually cleared.
//! Therefore, it is recommended that the interrupt source be cleared early in
//! the interrupt handler (as opposed to the very last action) to avoid
//! returning from the interrupt handler before the interrupt source is
//! actually cleared.  Failure to do so may result in the interrupt handler
//! being immediately reentered (because the interrupt controller still sees
//! the interrupt source asserted).
//!
//! \return None.
//
//*****************************************************************************
void
TimerIntClear(uint32_t ui32Base, uint32_t ui32IntFlags)
{
    //
    // Check the arguments.
    //
    ASSERT(_TimerBaseValid(ui32Base));

    //
    // Clear the requested interrupt sources.
    //
    HWREG(ui32Base + TIMER_O_ICR) = ui32IntFlags;
}
```

Figure 51 TimerIntClear src

5.3.2. TimerIntDisable

Datasheet Description:

29.2.2.16 TimerIntDisable

Disables individual timer interrupt sources.

Prototype:

```
void
TimerIntDisable(uint32_t ui32Base,
                uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the timer module.
ui32IntFlags is the bit mask of the interrupt sources to be disabled.

Description:

This function disables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter has the same definition as the *ui32IntFlags* parameter to TimerIntEnable().

Returns:

None.

*Figure 52 TimerIntDisable **Datasheet***

Src:

```
//*****************************************************************************
//
//! Disables individual timer interrupt sources.
//!
//! \param ui32Base is the base address of the timer module.
//! \param ui32IntFlags is the bit mask of the interrupt sources to be
//! disabled.
//!
//! This function disables the indicated timer interrupt sources.  Only the
//! sources that are enabled can be reflected to the processor interrupt;
//! disabled sources have no effect on the processor.
//!
//! The \e ui32IntFlags parameter has the same definition as the
//! \e ui32IntFlags parameter to TimerIntEnable().
//!
//! \return None.
//
//*****************************************************************************
void
TimerIntDisable(uint32_t ui32Base, uint32_t ui32IntFlags)
{
    //
    // Check the arguments.
    //
    ASSERT(_TimerBaseValid(ui32Base));

    //
    // Disable the specified interrupts.
    //
    HWREG(ui32Base + TIMER_O_IMR) &= ~(ui32IntFlags);
}
```

Figure 53 TimerIntDisable src

5.3.2. TimerIntEnable

Datasheet Description:

29.2.2.17 TimerIntEnable

Enables individual timer interrupt sources.

Prototype:

```
void
TimerIntEnable(uint32_t ui32Base,
               uint32_t ui32IntFlags)
```

Parameters:

ui32Base is the base address of the timer module.
ui32IntFlags is the bit mask of the interrupt sources to be enabled.

Description:

This function enables the indicated timer interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter must be the logical OR of any combination of the following:

- **TIMER_TIMB_DMA** - Timer B uDMA complete
- **TIMER_TIMA_DMA** - Timer A uDMA complete
- **TIMER_CAPB_EVENT** - Capture B event interrupt
- **TIMER_CAPB_MATCH** - Capture B match interrupt
- **TIMER_TIMB_MATCH** - Timer B match interrupt
- **TIMER_TIMB_TIMEOUT** - Timer B timeout interrupt
- **TIMER_RTC_MATCH** - RTC interrupt mask
- **TIMER_CAPA_EVENT** - Capture A event interrupt
- **TIMER_CAPA_MATCH** - Capture A match interrupt
- **TIMER_TIMA_MATCH** - Timer A match interrupt
- **TIMER_TIMA_TIMEOUT** - Timer A timeout interrupt

Returns:

None.

*Figure 54 TimerIntEnable **Datasheet***

```
//*****************************************************************************
//
//! Enables individual timer interrupt sources.
//!
//! \param ui32Base is the base address of the timer module.
//! \param ui32IntFlags is the bit mask of the interrupt sources to be enabled.
//!
//! This function enables the indicated timer interrupt sources.  Only the
//! sources that are enabled can be reflected to the processor interrupt;
//! disabled sources have no effect on the processor.
//!
//! The \e ui32IntFlags parameter must be the logical OR of any combination of
//! the following:
//!
//! - \b TIMER_TIMB_DMA - Timer B uDMA complete
//! - \b TIMER_TIMA_DMA - Timer A uDMA complete
//! - \b TIMER_CAPB_EVENT  - Capture B event interrupt
//! - \b TIMER_CAPB_MATCH  - Capture B match interrupt
//! - \b TIMER_TIMB_MATCH  - Timer B match interrupt
//! - \b TIMER_TIMB_TIMEOUT  - Timer B timeout interrupt
//! - \b TIMER_RTC_MATCH  - RTC interrupt mask
//! - \b TIMER_CAPA_EVENT  - Capture A event interrupt
//! - \b TIMER_CAPA_MATCH  - Capture A match interrupt
//! - \b TIMER_TIMA_MATCH  - Timer A match interrupt
//! - \b TIMER_TIMA_TIMEOUT  - Timer A timeout interrupt
//!
//! \return None.
//
//*****************************************************************************
void
TimerIntEnable(uint32_t ui32Base, uint32_t ui32IntFlags)
{
    //
    // Check the arguments.
    //
    ASSERT(_TimerBaseValid(ui32Base));

    //
    // Enable the specified interrupts.
    //
    HWREG(ui32Base + TIMER_O_IMR) |= ui32IntFlags;
}
```

Figure 55 TimerIntEnable src

5.4. Interrupt Controller in NVIC

5.4.1.IntDisable

Datasheet Description:

17.2.3.1 IntDisable

Disables an interrupt.

Prototype:

```
void
IntDisable(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be disabled.

Description:

The specified interrupt is disabled in the interrupt controller. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Example: Disable the UART 0 interrupt.

```
//
// Disable the UART 0 interrupt in the interrupt controller.
//
IntDisable(INT_UART0);
```

Returns:

None.

Figure 56 .IntDisable Datasheet

Src:

```
//*****************************************************************************
//
//! Disables an interrupt.
//!
//! \param ui32Interrupt specifies the interrupt to be disabled.
//!
//! The specified interrupt is disabled in the interrupt controller.  The
//! \e ui32Interrupt parameter must be one of the valid \b INT_* values listed
//! in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h
//! header file.  Other enables for the interrupt (such as at the peripheral
//! level) are unaffected by this function.
//!
//! \b Example: Disable the UART 0 interrupt.
//!
//! \verbatim
//! //
//! // Disable the UART 0 interrupt in the interrupt controller.
//! //
//! IntDisable(INT_UART0);
//!
//! \endverbatim
//!
//! \return None.
//
//*****************************************************************************
```

Figure 57 IntDisable src part 1

```
void
IntDisable(uint32_t ui32Interrupt)
{
    //
    // Check the arguments.
    //
    ASSERT(ui32Interrupt < NUM_INTERRUPTS);

    //
    // Determine the interrupt to disable.
    //
    if(ui32Interrupt == FAULT_MPU)
    {
        //
        // Disable the MemManage interrupt.
        //
        HWREG(NVIC_SYS_HND_CTRL) &= ~(NVIC_SYS_HND_CTRL_MEM);
    }
    else if(ui32Interrupt == FAULT_BUS)
    {
        //
        // Disable the bus fault interrupt.
        //
        HWREG(NVIC_SYS_HND_CTRL) &= ~(NVIC_SYS_HND_CTRL_BUS);
    }
    else if(ui32Interrupt == FAULT_USAGE)
    {
        //
        // Disable the usage fault interrupt.
        //
        HWREG(NVIC_SYS_HND_CTRL) &= ~(NVIC_SYS_HND_CTRL_USAGE);
    }
    else if(ui32Interrupt == FAULT_SYSTICK)
    {
        //
        // Disable the System Tick interrupt.
        //
        HWREG(NVIC_ST_CTRL) &= ~(NVIC_ST_CTRL_INTEN);
    }
    else if(ui32Interrupt >= 16)
    {
        //
        // Disable the general interrupt.
        //
        HWREG(g_pui32Dii16Regs[(ui32Interrupt - 16) / 32]) =
            1 << ((ui32Interrupt - 16) & 31);
    }
}
```

Figure 58 IntDisable src part 2

5.4.2. IntEnable

Datasheet Description:

17.2.3.2 IntEnable

Enables an interrupt.

Prototype:

```
void
IntEnable(uint32_t ui32Interrupt)
```

Parameters:

ui32Interrupt specifies the interrupt to be enabled.

Description:

The specified interrupt is enabled in the interrupt controller. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Example: Enable the UART 0 interrupt.

```
//
// Enable the UART 0 interrupt in the interrupt controller.
//
IntEnable(INT_UART0);
```

Returns:

None.

Figure 59 .IntEnable Datasheet

Src:

```
//*****************************************************************************
//
//! Enables an interrupt.
//!
//! \param ui32Interrupt specifies the interrupt to be enabled.
//!
//! The specified interrupt is enabled in the interrupt controller.  The
//! \e ui32Interrupt parameter must be one of the valid \b INT_* values listed
//! in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h
//! header file. Other enables for the interrupt (such as at the peripheral
//! level) are unaffected by this function.
//!
//! \b Example: Enable the UART 0 interrupt.
//!
//! \verbatim
//! //
//! // Enable the UART 0 interrupt in the interrupt controller.
//! //
//! IntEnable(INT_UART0);
//!
//! \endverbatim
//!
//! \return None.
//
//*****************************************************************************
```

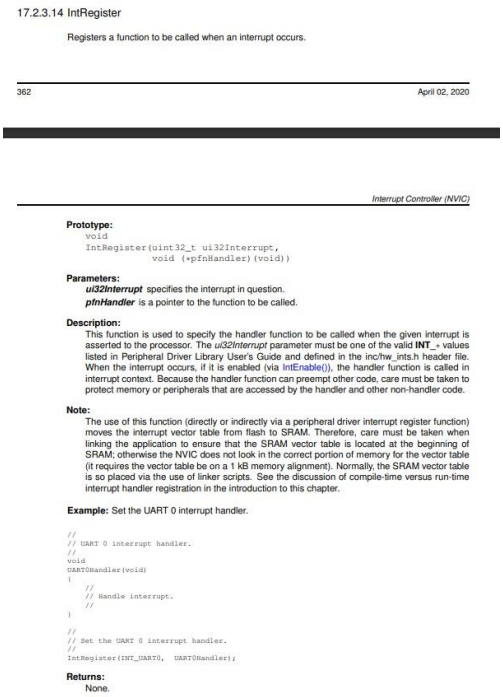
Figure 60 IntEnable src part 1

```
void
IntEnable(uint32_t ui32Interrupt)
{
    //
    // Check the arguments.
    //
    ASSERT(ui32Interrupt < NUM_INTERRUPTS);

    //
    // Determine the interrupt to enable.
    //
    if(ui32Interrupt == FAULT_MPU)
    {
        //
        // Enable the MemManage interrupt.
        //
        HWREG(NVIC_SYS_HND_CTRL) |= NVIC_SYS_HND_CTRL_MEM;
    }
    else if(ui32Interrupt == FAULT_BUS)
    {
        //
        // Enable the bus fault interrupt.
        //
        HWREG(NVIC_SYS_HND_CTRL) |= NVIC_SYS_HND_CTRL_BUS;
    }
    else if(ui32Interrupt == FAULT_USAGE)
    {
        //
        // Enable the usage fault interrupt.
        //
        HWREG(NVIC_SYS_HND_CTRL) |= NVIC_SYS_HND_CTRL_USAGE;
    }
    else if(ui32Interrupt == FAULT_SYSTICK)
    {
        //
        // Enable the System Tick interrupt.
        //
        HWREG(NVIC_ST_CTRL) |= NVIC_ST_CTRL_INTEN;
    }
    else if(ui32Interrupt >= 16)
    {
        //
        // Enable the general interrupt.
        //
        HWREG(g_pui32EnRegs[(ui32Interrupt - 16) / 32]) =
            1 << ((ui32Interrupt - 16) & 31);
    }
}
```

Figure 61 IntEnable src part 2

5.4.3. IntRegister

Datasheet Description:



17.2.3.14 IntRegister

Registers a function to be called when an interrupt occurs.

362 April 02, 2020

Interrupt Controller (NVIC)

Prototype:

```
void
IntRegister(uint32_t ui32Interrupt,
            void (*pfnHandler)(void))
```

Parameters:

ui32Interrupt specifies the interrupt in question.

pfnHandler is a pointer to the function to be called.

Description:

This function is used to specify the handler function to be called when the given interrupt is asserted to the processor. The *ui32Interrupt* parameter must be one of the valid **INT_** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. When the interrupt occurs, if it is enabled (via IntEnable()), the handler function is called in interrupt context. Because the handler function can preempt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other non-handler code.

Note:

The use of this function (directly or indirectly via a peripheral driver interrupt register function) moves the interrupt vector table from flash to SRAM. Therefore, care must be taken when linking the application to ensure that the SRAM vector table is located at the beginning of SRAM; otherwise the NVIC does not look in the correct portion of memory for the vector table (it requires the vector table be on a 1 kB memory alignment). Normally, the SRAM vector table is so placed via the use of linker scripts. See the discussion of compile-time versus run-time interrupt handler registration in the introduction to this chapter.

Example: Set the UART 0 interrupt handler.

```
//
// UART 0 interrupt handler.
//
void
UART0Handler(void)
{
    //
    // Handle interrupt.
    //
}

//
// Set the UART 0 interrupt handler.
//
IntRegister(INT_UART0,  UART0Handler);
```

Returns:

None.

*Figure 62 IntRegister **Datasheet***

Src:

```
//*****************************************************************************
//
//! Registers a function to be called when an interrupt occurs.
//!
//! \param ui32Interrupt specifies the interrupt in question.
//! \param pfnHandler is a pointer to the function to be called.
//!
//! This function is used to specify the handler function to be called when the
//! given interrupt is asserted to the processor.  The \e ui32Interrupt
//! parameter must be one of the valid \b INT_* values listed in Peripheral
//! Driver Library User's Guide and defined in the inc/hw_ints.h header file.
//! When the interrupt occurs, if it is enabled (via IntEnable()), the handler
//! function is called in interrupt context.  Because the handler function can
//! preempt other code, care must be taken to protect memory or peripherals
//! that are accessed by the handler and other non-handler code.
//!
//! \note The use of this function (directly or indirectly via a peripheral
//! driver interrupt register function) moves the interrupt vector table from
//! flash to SRAM.  Therefore, care must be taken when linking the application
//! to ensure that the SRAM vector table is located at the beginning of SRAM;
//! otherwise the NVIC does not look in the correct portion of memory for the
//! vector table (it requires the vector table be on a 1 kB memory alignment).
//! Normally, the SRAM vector table is so placed via the use of linker scripts.
//! See the discussion of compile-time versus run-time interrupt handler
//! registration in the introduction to this chapter.
//!
//! \b Example: Set the UART 0 interrupt handler.
//!
//! \verbatim
//!
//! //
//! // UART 0 interrupt handler.
//! //
//! void
//! UART0Handler(void)
//! {
//!     //
//!     // Handle interrupt.
//!     //
//! }
//!
//! //
//! // Set the UART 0 interrupt handler.
//! //
//! IntRegister(INT_UART0, UART0Handler);
//!
//! \endverbatim
//!
//! \return None.
//
//*****************************************************************************
```

Figure 63 IntRegister src part 1

```
void
IntRegister(uint32_t ui32Interrupt, void (*pfnHandler)(void))
{
    uint32_t ui32Idx, ui32Value;

    //
    // Check the arguments.
    //
    ASSERT(ui32Interrupt < NUM_INTERRUPTS);

    //
    // Make sure that the RAM vector table is correctly aligned.
    //
    ASSERT(((uint32_t)g_pfnRAMVectors & 0x000003ff) == 0);

    //
    // See if the RAM vector table has been initialized.
    //
    if(HWREG(NVIC_VTABLE) != (uint32_t)g_pfnRAMVectors)
    {
        //
        // Copy the vector table from the beginning of FLASH to the RAM vector
        // table.
        //
        ui32Value = HWREG(NVIC_VTABLE);
        for(ui32Idx = 0; ui32Idx < NUM_INTERRUPTS; ui32Idx++)
        {
            g_pfnRAMVectors[ui32Idx] = (void (*)(void))HWREG((ui32Idx * 4) +
                                                             ui32Value);
        }

        //
        // Point the NVIC at the RAM vector table.
        //
        HWREG(NVIC_VTABLE) = (uint32_t)g_pfnRAMVectors;
    }

    //
    // Save the interrupt handler.
    //
    g_pfnRAMVectors[ui32Interrupt] = pfnHandler;
}
```

Figure 64 IntRegister src part 2

6.External Functions

6.1. isdigit

6.1.1. Description

Arguments:

c

The character you want to test. This must be representable as an `unsigned char` or be EOF; the behavior for other values is undefined. Because this argument is interpreted as an `int`, to avoid sign extension on character values greater than 0x7F, you must cast the argument to the `unsigned` data type; otherwise, the function will behave unpredictably.

Library:

libc

Use the `-l c` option to gcc to link against this library. This library is usually included automatically.

Description:

The *isdigit()* function tests for a decimal digit (characters 0 through 9).

Returns:

Nonzero if *c* is a decimal digit; otherwise, zero.

Figure 65 isdigit Description

6.1.1. src

```
#pragma inline=no_body
int isdigit(int _C)
{
  return _C >= '0' && _C <= '9';
}
```

Figure 66 isdigit Code

7. main.c

7.1. Introduction & Inclusion

```
/*****************************************************************************************************
 *
 * Module:  Application
 *
 * File Name: main.c
 *
 * Author: Team 3
 *
 * Description: An application that have 3 modes of operation (Calculator, Stopwatch, Timer)
 *              with allowance of Switching modes at any moment.
 *              ****************************   STOPWATCH   ***************************
 *              the Stopwatch mode starts automatically and is supported with 3 pushbuttons
 *              - 1 for pausing the stopwatch
 *              - 1 for resuming the stopwatch
 *              - 1 for resetting the stopwatch
 *              ****************************     TIMER     ***************************
 *              the timer mode once open waits for capturing input from the keypad and then pressing '='
 *              supports up till 59:59 counting down
 *              once counting is done, the red led in the hardware will light up
 *              ****************************  CALCULATOR   ***************************
 *              calculator mode supports 4 mathematical operations (Adding, Subtracting, Dividing, Multiplication)
 *              takes the first number from the user and waits for the operator to be pressed
 *              then the second number is taken and the result is printed and keeps available for more calculations
 *
 *****************************************************************************************************/

#include "keypad.h"
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h"
#include "driverlib/sysctl.h"
#include "driverlib/systick.h"
#include "tm4c123gh6pm.h"
#include "bitwise_operation.h"
#include "Timer.h"
#include "ctype.h"

```

Figure 67 Introduction & Inclusion

7.2. Variable Declaration

```
/*****************************************************************************************************
 *
 *                                  VARIABLE DECLARATION
 *
 *****************************************************************************************************/

uint8 sec1 = '0';                       // Timer        x  x : x S1
uint8 sec2 = '0';                       // Timer        x  x : S2 x
uint8 min1 = '0';                       // Timer        x  M1: x  x
uint8 min2 = '0';                       // Timer        M2 x : x  x
uint8 sw_sec1 = '0';                    // StopWatch    x  x : x S1
uint8 sw_sec2 = '0';                    // StopWatch    x  x : S2 x
uint8 sw_min1 = '0';                    // StopWatch    x  M1: x  x
uint8 sw_min2 = '0';                    // StopWatch    M2 x : x  x
uint16 counter = 0;
float32 remainderz = 0;
uint16 mode = 0;                        // counter for knowing the current mode

```

Figure 68 Variable Declaration

7.3. Function Prototypes

```
/*******************************************************************************
*
*                                FUNCTIONS PROTOTYPES
*
*******************************************************************************/
void modes(void);
/*
  mode control button connected on pin 4 PORTF
  if mode variable reaches 4 it reset again to 1 to keep on Looping between modes
*/
void stopwatch_count(void);
/*
  function for incrementing the stopwatch
*/
void timertake(void);
/*
  function that allows input of values to initialize with the timer
*/
void timerinterrupt(void);
/*
  function that Decrement the timer
*/
void stopwatch_display(void);
/*
  function for displaying the counting of the stopwatch
*/
void interrupt(void);
/*
  Initializing All GPIO Interrupts and Buttons for Stopwatch
*/
void display(void);
/*
  Calculator Mode
*/

```

Figure 69 Function Prototypes

7.4. main

7.4.1.Setup

```

/*******************************************************************************
*
*                            MAIN APPLICATION
*
*******************************************************************************/
int main(void)
{

  keypad_init ();                                   // initialize the keypad
  LCD_init ();                                      // initialize the LCD
  timer_init ();                                    // initialize the timer
  SYSCTL_RCGCGPIO_R  |= 0x20;                        // enable the clock to PORTF
  GPIO_PORTF_DIR_R   |= (1 << 1);                    // pin 1 set as output for the LED
  GPIO_PORTF_DEN_R   |= (1 << 1);                    // pin 1 Digital enable
  GPIO_PORTF_DEN_R   |= (1 << 4);                    // pin 4 Digital enable
  GPIO_PORTF_PDR_R   |= (1 << 4);                    // pin 4 set as pull down

  LCD_command (1);
  LCD_command (0x80);
  SysTickDisable ();
  SysTickIntDisable ();
  SysTickPeriodSet (16000000  - 1);                 // 1 second period set
  SysTickEnable ();
  LCD_printString ("Welcome" );
  LCD_command (0xC0);
  LCD_printString ("TEAM 3 PROJECT" );
  delayMs (2000);

  GPIOIntRegister(GPIO_PORTF_BASE , modes );          // when an interrupt generates it calls the modes function
  GPIOIntTypeSet (GPIO_PORTF_BASE , 0x10, GPIO_FALLING_EDGE );    // detect falling edges on pin 4
  GPIOIntEnable (GPIO_PORTF_BASE , GPIO_INT_PIN_4 );              // enable interrupts on pin 4

```

Figure 70 Loop

7.4.2. loop

```
128
129  while (1)
130  {
131    switch (mode)
132    {
133
134    case 1:
135      timertake();                                    // Calling timer function
136      break;
137    case 2:
138      SysTickDisable();                               // DISABLE Timer
139      // reset values
140      sec1 = '0';
141      sec2 = '0';
142      min1 = '0';
143      min2 = '0';
144      sw_sec1 = '0';
145      sw_sec2 = '0';
146      sw_min1 = '0';
147      sw_min2 = '0';
148
149      stopwatch_display();                            // Calling Stopwatch display to keep showing stopwatch format
150      if (keypad_read() == '=')
151      {
152        // IF = IS PRESSED START THE STOPWATCH
153        IntRegister(INT_TIMER0A, stopwatch_count);
154        interrupt();                                  // stopwatch operations interrupts setup
155        timer_enable();
156      }
157
158      while (mode == 2) {} // STAY IN THIS CASE IF MODE DOESNT CHANGE
159
160      break;
161    case 3:
162      TimerDisable(TIMER0_BASE, TIMER_BOTH); // FOR STOPWATCH
163      sw_sec1 = '0';
164      sw_sec2 = '0';
165      sw_min1 = '0';
166      sw_min2 = '0';
167
168      display(); // CALLING CALCULATOR
169
170      break;
171    default:
172      break;
173    }
174  }
175 }
176
```

Figure 71 Loop

7.5. Pause Function

```
177
178 /*****************************************************************************
179 *
180 *                              FUNCTIONS DEFINITION
181 *
182 *****************************************************************************/
183
184 // FUNCTION FOR THE STOPWATCH INTERRUPTS
185 void pause()
186 {
187   // IF INTERRUPT CAME FROM PIN 2, DISABLE TIMER (PAUSE STOPWATCH)
188   // CLEAR INTERRUPT FLAG
189   if (GPIOIntStatus(GPIO_PORTA_BASE, 0x04) == 0x04)
190   {
191     TimerDisable(TIMER0_BASE, TIMER_BOTH);
192     GPIOIntClear(GPIO_PORTA_BASE, GPIO_INT_PIN_2);
193   }
194   // IF INTERRUPT CAME FROM PIN 3, ENABLE TIMER (RESUME STOPWATCH)
195   // CLEAR INTERRUPT FLAG
196   else if (GPIOIntStatus(GPIO_PORTA_BASE, 0x08) == 0x08)
197   {
198
199     timer_enable();
200     GPIOIntClear(GPIO_PORTA_BASE, GPIO_INT_PIN_3);
201   }
202    // IF INTERRUPT CAME FROM PIN 4, RESET Values
203    // CLEAR INTERRUPT FLAG
204   else if (GPIOIntStatus(GPIO_PORTA_BASE, 0x10) == 0x10)
205   {
206     sw_sec1 = '0';
207     sw_sec2 = '0';
208     sw_min1 = '0';
209     sw_min2 = '0';
210
211     GPIOIntClear(GPIO_PORTA_BASE, GPIO_INT_PIN_4);
212   }
213 }
214
215 /*****************************************************************************
216 *****************************************************************************/
217
```

Figure 72 Pause Function

7.6. Interrupts Initialization Function

```
214
215 /*****************************************************************************************
216 *****************************************************************************************/
217
218 // INITIALIZING ALL GPIO INTERRUPTS AND BUTTONS FOR STOPWATCH OPERATIONS
219 void interrupt(void)
220 {
221   GPIO_PORTA_DEN_R |= (1 << 2);                    // assigned for the pause button
222   GPIO_PORTA_DEN_R |= (1 << 3);                    // assigned for the resume button
223   GPIO_PORTA_DEN_R |= (1 << 4);                    // assigned for the reset button
224   GPIO_PORTA_PDR_R |= (1 << 2);
225   GPIO_PORTA_PDR_R |= (1 << 3);
226   GPIO_PORTA_PDR_R |= (1 << 4);
227
228   GPIOIntRegister(GPIO_PORTA_BASE, pause);
229   GPIOIntTypeSet(GPIO_PORTA_BASE, 0x04, GPIO_FALLING_EDGE);
230   GPIOIntTypeSet(GPIO_PORTA_BASE, 0x08, GPIO_FALLING_EDGE);
231   GPIOIntTypeSet(GPIO_PORTA_BASE, 0x10, GPIO_FALLING_EDGE);
232
233   GPIOIntEnable(GPIO_PORTA_BASE, GPIO_INT_PIN_2);
234   GPIOIntEnable(GPIO_PORTA_BASE, GPIO_INT_PIN_3);
235   GPIOIntEnable(GPIO_PORTA_BASE, GPIO_INT_PIN_4);
236
237 }
238
```

Figure 73 Interrupts Initialization Function

7.7. Calculation Function

```
239 /*****************************************************************************************
240 *****************************************************************************************/
241
242 // FUNCTION THAT TAKES THE NUMBERS AND OPERATION FROM CALCULATOR
243 // RETURNS THE RESULT
244 int32 calculator(float32 num1, float32 num2, uint8 op)
245 {
246
247   int32 answer;
248
249   switch (op)
250   {
251   case '+':
252     answer = ((uint32)num1 + (uint32)num2);
253     break;
254   case '-':
255     answer = ((int32)num1 - (int32)num2);
256     break;
257   case '*':
258     answer = ((uint32)num1 * (uint32)num2);
259     break;
260   case '/':
261     answer = ((uint32)num1 / (uint32)num2);
262     remainderz = (((float32)(num1 / num2) * 100) - ((uint32)answer * 100));
263     break;
264   default:
265     break;
266   }
267   return answer;
268 }
269
```

Figure 74 Calculation Function

7.8. Calculator Function

```
272
273 // CALCULATOR FUNCTION
274 void display(void)
275 {
276   LCD_command(1);                                    // CLEAR SCREEN
277   LCD_command(0x80);                                 // MOVE CURSOR TO FIRST POSITION ON LCD
278   LCD_printString("Calc Mode");                      // PRINT CALC MODE ON LCD
279   LCD_command(0xC0);                                 // MOVE CURSOR TO SECOND ROW
280   float32 number1 = 0;
281   float32 number2 = 0;
282   uint8 op = 0;
283   float32 result;
284   uint8 x;
285   int calcCounter = 0;
286   int le = 0;
287   while (mode == 3)
288   {
289     uint8 key = keypad_read();
290     switch (calcCounter)
291     {
292     case 0:
293       if (isdigit(key))
294       {
295         x = key;                                     // TAKE FIRST VALUE
296         number1 = (number1 * 10) + (key - 48);       // SAVE NUMBER AND MINUS 48 TO CHANGE FROM ASCII
297         LCD_data(x);                                 // PRINT FIRST VALUE
298         le++;                                        // VARIABLE USED TO TAKE MORE NUMBERS
299         delayMs(200);
300       }
301       else if (((key == '+') || (key == '-') || (key == '*') || (key == '/')) && le > 0)
302       {
303         // IF le IS MORE THAN 0 KEEP TAKING NUMBERS UNTIL YOU GET A SYMBOL
304         op = key;                                    // SAVE OP TO SEND TO CALCULATOR FUNCTION
305         LCD_data(op);                                // PRINT THE OPERATION ON CALCULATOR
306         delayMs(500);
307         calcCounter++;                               // WHEN TAKING OPERATION MOVE TO NEXT CASE
308         le = 0;
309         delayMs(200);
310       }
311       break;
312       // REPEAT FOR SECOND NUMBER
313     case 1:
314       if (isdigit(key))
315       {
316         x = key;
317         number2 = (number2 * 10) + (key - 48);
318         LCD_data(x);
319         le++;
320         delayMs(200);
321       }
322       else if ((key == '=') && le > 0)
323       {
324         // WHEN KEYPAD GIVE = SEND RESULT TO CALCULATOR FUNCTION
325         LCD_data('=');
326         result = calculator(number1, number2, op);
327         if (op == '/')
328         {
329           if (number2 == 0)
330           {
331             // IF NUMBER IS DIVIDED BY 0 PRINT MATH ERROR
332             LCD_command(1);
333             LCD_printString("MATH ERROR!");
334             delayMs(100);
335           }
336           else
337           {
338             // IF OPERATION IS DIVISION PRINT IT AS FLOAT
339             LCD_printInt((uint32)result);
340             delayMs(1);
341             LCD_data('.');
342             delayMs(1);
343             LCD_printInt((uint32)remainderz);
344             delayMs(1000);
345           }
346         }
347         else
348         {
349           // ELSE PRINT THE RESULT
350           LCD_printInt((int32)result);
351
352           delayMs(250);
353         }
354         delayMs(1000); // WAIT FOR A SECOND
355         LCD_command(1); // CLEAR SCREEN
356         return;
357       }
358     }
359   }
360 }
361
```

Figure 75 Calculator Function

7.9. Stopwatch Display Function

```
365 void stopwatch_display(void)
366 {
367   LCD_command(1);
368   LCD_command(0x80);
369   LCD_printString("Stopwatch Mode");
370   LCD_command(0xC0);
371   LCD_printString("00:00");
372 }
```

Figure 76 Stopwatch Display Function

7.10. Stopwatch Count Function

```
373 void stopwatch_count(void)
374 {
375 // Printing 00:00 on LCD
376   LCD_command(0xC0);
377   LCD_data(sw_min2);
378   LCD_command(0xC1);
379   LCD_data(sw_min1);
380   LCD_command(0xC2);
381   LCD_data(':');
382   LCD_command(0xC3);
383   LCD_data(sw_sec2);
384   LCD_command(0xC4);
385   LCD_data(sw_sec1);
386   sw_sec1++;
387   LCD_command(0xC4);
388   LCD_data(sw_sec1);
389   if (sw_sec1 == ':') // : is the ASCII for 10
390   {
391     // Once sec1 reaches 10 turn it to 0 and increment sec2
392     sw_sec1 = '0';
393     LCD_command(0xC4);
394     LCD_data(sw_sec1);
395     sw_sec2++;
396     LCD_command(0xC3);
397     LCD_data(sw_sec2);
398   }
399   if (sw_sec2 == '6')
400   {
401     // Once sec2 reaches 6 turn it to 0 and increment min1
402     sw_sec2 = '0';
403     LCD_command(0xC3);
404     LCD_data(sw_sec2);
405     sw_min1++;
406     LCD_command(0xC1);
407     LCD_data(sw_min1);
408   }
409   if (sw_min1 == ':')
410   {
411     // Once min1 reaches 10 turn it to 0 and increment min2
412     sw_min1 = '0';
413     LCD_command(0xC1);
414     LCD_data(sw_min1);
415     sw_min2++;
416     LCD_command(0xC0);
417     LCD_data(sw_min2);
418   }
419   if (sw_min2 == '6' && sw_min1 == '0')
420   {
421     // When stopwatch finishes reset it
422     sw_min2 = '0';
423     sw_min1 = '0';
424   }
425   // Clearing interrupt flag
426  TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
427 }
428
```

7.11. Modes Function

```
429 /*****************************************************************************
430 *****************************************************************************/
431
432 void modes(void)
433 {
434   GPIOIntClear(GPIO_PORTF_BASE, GPIO_INT_PIN_4);
435   mode++;
436   if (mode == 4)
437     mode = 1;
438 }
439
```

Figure 77 Modes Function

7.12. TimerTake Function

```
443 void timertake(void)
444 {
445   // DISPLAYING TIMER MODE
446   LCD_command(0x80);
447   LCD_command(1);
448   LCD_printString("Timer Mode");
449   LCD_command(0xC0);
450   LCD_printString("00:00");
451   while (mode == 1)
452   {
453     uint8 key = keypad_read();
454     switch (counter)
455     {
456     case 0:
457       // IF A NUMBER IS ENTERED PRESENT IT ON THE SEC1 PLACE ON LCD
458       if (isdigit(key))
459       {
460         sec1 = key;
461         LCD_command(0xC4);
462         LCD_data(sec1);
463         counter++;
464         delayMs(500);
465       }
466       break;
467     case 1:
468
469       if (key == '=')
470       {
471         // IF = IS PRESSED GO TO CASE 5
472         counter = 5;
473         break;
474       }
475       else if (isdigit(key))
476       {
477           // ELSE IF A NUMBER IS PRESSED MOVE SEC1 TO SEC2 AND TAKE NEW SEC1
478         sec2 = sec1;
479         sec1 = key;
480         LCD_command(0xC3);
481         LCD_data(sec2);
482         LCD_command(0xC4);
483         LCD_data(sec1);
484         delayMs(500);
485         counter++;
486       }
487       break;
```

Figure 78 TimerTake function Part1

```
488     case 2:
489       if (key == '=')
490       {
491         // IF = IS PRESSED GO TO CASE 5
492         counter = 5;
493         break;
494       }
495       else if (isdigit(key))
496       {
497         // ELSE IF A NUMBER IS PRESSED SHIFT NUMBERS TO LEFT AND TAKE NEW
498         min1 = sec2;
499         sec2 = sec1;
500         sec1 = key;
501         LCD_command(0xC1);
502         LCD_data(min1);
503         LCD_command(0xC3);
504         LCD_data(sec2);
505         LCD_command(0xC4);
506         LCD_data(sec1);
507       delayMs(500);
508 
509         counter++;
510         break;
511       }
512     case 3:
513 
514       if (key == '=')
515       {
516         // IF = IS PRESSED GO TO CASE 5
517         counter = 5;
518         break;
519       }
520       else if (isdigit(key))
521       {
522         // ELSE IF A NUMBER IS PRESSED SHIFT NUMBERS TO LEFT AND TAKE NEW
523         min2 = min1;
524         min1 = sec2;
525         sec2 = sec1;
526         sec1 = key;
527         LCD_command(0xC0);
528         LCD_data(min2);
529         LCD_command(0xC1);
530         LCD_data(min1);
531         LCD_command(0xC3);
532         LCD_data(sec2);
533         LCD_command(0xC4);
534         LCD_data(sec1);
535       delayMs(500);
536         counter++;
537         break;
```

Figure 79 TimerTake function Part 2

```
539     case 4:
540       counter++;
541       break;
542     case 5:
543       if ((sec2 > '5') || (min2 > '5'))
544       {
545         // IF NUMBER IS MORE THAN 5 IN SEC 2 OR MIN 2
546         // PRINT INVALID INPUT
547         // THEN RETURN FROM FUNCTION
548         LCD_command(1);
549         LCD_printString("INVALID INPUT");
550         min2='0';
551         min1='0';
552         sec2='0';
553         sec1='0';
554         delayMs(1000);
555         counter = 0;
556        return;
557         break;
558       }
559       // IF NOT START THE TIMER
560       SysTickIntRegister(timerinterrupt);
561       SysTickEnable();
562       break;
563     case 6:
564       // CASE USED TO RETURN TIMER TO THE START WHEN TIMER STOPS
565       counter = 0;
566       SysTickDisable();
567
568       return;
569       break;
570     default:
571       break;
572     }
573
574   }
575 }
```

Figure 80 TimerTake function Part 3

7.13. Timer Interrupt Function

```
void timerinterrupt(void)
{
  // START DECREMENTING THE SECONDS
  sec1--;
  LCD_command(0xC4);
  LCD_data(sec1);
  if (sec1 == '0' && sec2 == '0' && min1 == '0' && min2 == '0')
  {
  // IF TIMER REACHES 0 STOP TIMER AND TOGGLE LED
    SysTickDisable();
    GPIO_PORTF_DATA_R ^= 0x02;
    delayMs(2000);
    GPIO_PORTF_DATA_R ^= 0x02;
    min2 = '0';
    min1 = '0';
    sec2 = '0';
    sec1 = '0';
    counter = 6;
    LCD_command(1);
  }
  if (sec1 == '/') // ASCII FOR NUMBER UNDER 0
  {
    // IF SEC1 REACHS 0 PUT IT AS 9 AND DECREMENT SEC2
    sec1 = '9';
    LCD_command(0xC4);
    LCD_data(sec1);
    sec2--;
    LCD_command(0xC3);
    LCD_data(sec2);
  }

  if (sec2 == '/')
  {
    // IF SEC2 REACHS 0 PUT IT AS 5 AND SEC 1 AS 9 AND DECREMENT MIN1
    sec1 = '9';
    LCD_command(0xC4);
    LCD_data(sec1);
    sec2 = '5';
    LCD_command(0xC3);
    LCD_data(sec2);

    min1--;
    LCD_command(0xC1);
    LCD_data(min1);
  }
  if (min1 == '/')
  {
    // IF MIN1 REACHS 0 PUT IT AS 9 AND SEC1 AS 9 AND SEC2 AS 5 AND DECREMENT MIN2
    sec1 = '9';
    LCD_command(0xC4);
    LCD_data(sec1);
    sec2 = '5';
    LCD_command(0xC3);
    LCD_data(sec2);
    min1 = '9';
    LCD_command(0xC1);
    LCD_data(min1);
    min2--;
    LCD_command(0xC0);
    LCD_data(min2);
  }
}
```

Figure 81 Timer Interrupt Function

Code Link: https://drive.google.com/file/d/1_FDW6JymK80H-E0u_BNLpZpqpk910rzX/view?usp=sharing

8.Problem Faced

1. Shifting modes was tricky since the interrupt handler couldn't switch modes until the timer finishes counting.
2. The timer couldn't display '0' in neither character's window due to a misunderstood concept of counting. When was checking that the counter reaches '0' the lcd ended displaying characters at '1' but when changed to check on the character before '0' in the ASCII table, the problem was figured out.

```
if (sec1 == '/') // ASCII FOR NUMBER UNDER 0
{
  // IF SEC1 REACHS 0 PUT IT AS 9 AND DECREMENT SEC2
  sec1 = '9';
  LCD_command(0xC4);
  LCD_data(sec1);
  sec2--;
  LCD_command(0xC3);
  LCD_data(sec2);
}
```

Figure 82 problem fix

3. Pins that are connected together has made the hardware connection challenging
   - PD_0 && PB_6
   - PD_1 && PD_7
4. Timer was based on polling algorithm and that prohibited the exit from the timer mode
5. There were many 'while loops' used and that caused problems of exiting the calculator
6. When was using the while(1) strategy with the keypad we were stuck inside until we changed it to while(case==XX) and that allowed the while loop to break when the condition doesn't fit (mode changed for example).

9.Results

9.1. Circuit

Keypad connection:

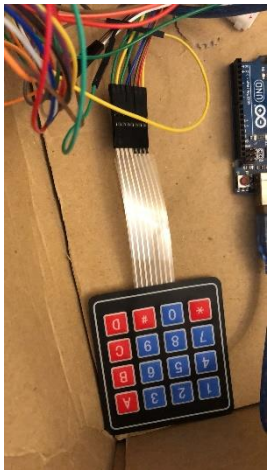


Figure 84 Keypad connection

Pushbutton Connection:

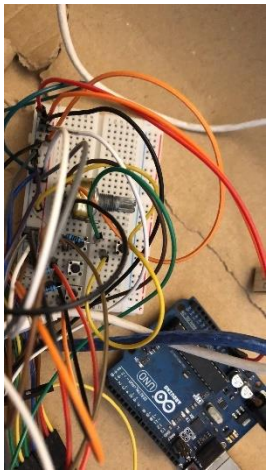


Figure 83 Pushbutton Connection

LCD connection:

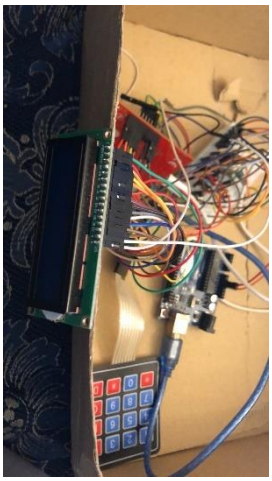


Figure 86 LCD connection

5V from Arduino:

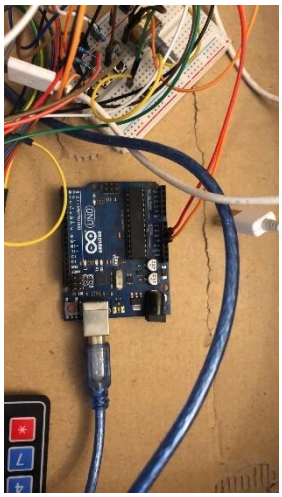


Figure 85 5V from Arduino

9.2. Video

Link: https://drive.google.com/file/d/10WY0s8R8piYng1mKXESCsem7BKFy9hEh/view?usp=sharing

Code & Video Link: https://drive.google.com/drive/folders/1Av6pO7-uBwBBLLsxumZ2BBPrWy_NswfZ?usp=sharing