ELEC 5620M – Embedded System Design

Assignment 2 – Stopwatch using ARM A9 Private Timer and Seven Segment Displays

Arul Prakash Samathuvamani

Student Number : 201460156

Abstract

A digital stopwatch was designed using ARM A9 Private Timer and 7-Segment displays and was tested for accurate time keeping and other functionalities. The stopwatch when started, displays time in MM : SS : FF format where MM represents minutes, SS represents seconds and ff represents hundredths of seconds. The code was uploaded to DE1-SoC board and debugged.

Keywords: Stopwatch , ARM A9 , interrupts, private timer

Introduction

A stopwatch is a watch that can be started and stopped in order to identify the exact time of events [1]. The objective of this assignment is to design a digital stopwatch using ARM A9 private timer and 7-segment displays and debug the same using DE1-SoC board. DE1-SoC is hardware design platform with Cyclone V SoC with ARM A9 dual core processor.

The stopwatch has the following specifications.

1. The stopwatch can be started using **Button 1** on the DE1-SoC board.
2. The stopwatch can be stopped using **Button 2** on the DE1-SoC board. When the stopwatch is stopped and the button is pressed again, the stopwatch resets itself.
3. The stopwatch displays time in MM : SS : FF format where MM denotes minutes, SS denotes seconds and FF denotes hundredths of seconds. When the stopwatch has been running for more than an hour, the stopwatch displays time in HH : MM : SS format where HH denotes hour, MM denotes minutes and SS denotes seconds.
4. The stopwatch has split timer functionality. When **Button 3** on the DE1-SoC board is pressed, the time during the button press is recorded but the stopwatch continues to run. When the stopwatch is stopped, the recorded time can be viewed by pressing **Button 3** on the board. The split timer gets reset when the stopwatch is reset.
5. **Button 4** turns on/off the stopwatch.
6. Display animations on the stopwatch when the stopwatch has counted a minute.

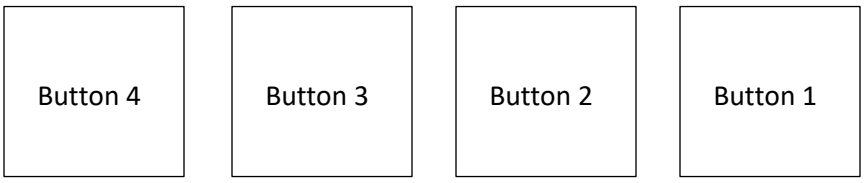

Figure1. Button Layout on DE1- SoC Board.

ARM A9 Private Timer:

Timer allows the user to count the number of clock cycles, which in turn can be used for accurate time keeping. ARM A9 Private Timer is a 32-bit counter that interrupts when it reaches zero. The ARM A9 Private Timer has four registers namely, Private Timer Load Register, Private Timer Counter Register, Private Timer Control Register and Private Timer Interrupt Status Register. The Timer Load Register is loaded with the number of cycles to be counted, the value in the Timer Load Register is copied into Timer Control Register when auto reload mode is enabled. The Timer Control Register decrements for every peripheral clock. The Private Timer Interrupt Status register is flagged high when Private Control Register reaches zero.

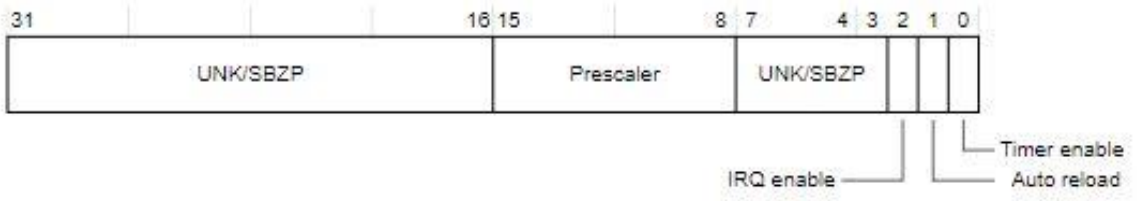

Figure 2. Private Timer Control Bit Assignments

Figure 2. represents Private Timer Control Bit Assignments. When Auto Reload in Private Control Timer is set to high, the value in the Timer Load Register is copied into Timer Control Register when it reaches zero. Prescalar is used to modify the clock period for decrementing event. [2]

The formula for calculating the time it takes to decrement to zero is as below [2],

$$Time\ taken\ in\ seconds = \frac{((Prescalar + 1)\ X\ (\ load\ value + 1))}{Peripheral\ Clock}$$

The Peripheral Clock value is 225 MHz for DE1-SoC, and to count one hundredth of a second, the Timer Load Register should be loaded with 2250000 and to count one second, the Timer Load Register should be loaded with 2250000.[3]

7-Segment Display:

The DE1-SoC Board has six 7-segment displays. From right to left, the first four displays are controlled with base address 0xFF200020 while last two displays are controlled with base address 0xFF200030. Figure 3. represents 7-segment LED bit mapping. Setting 1 would turn on the corresponding LED.

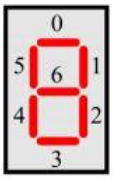

Figure 3. 7-Segment LED mapping

Interrupts:

Interrupt is a mechanism of interrupting the current execution of a process [4]. In case of a interrupt during execution of a program, the ARM A9 executes single instruction at specific memory location called the vector. The vector contains the address to load into program counter in case of an interrupt. The function that handles the interrupt is called Interrupt Service Routine. The system has been designed to flag an IRQ interrupt when Button 4 is pressed. IRQ is a hardware interrupt that executes a specific function of code in response to an event. When IRQ interrupt occurs, processor looks up the address of the function to run from Vector Base Address Register (VBAR). After running the interrupt function, the program returns to the point it was running when the exception occurs.

Configuration of Interrupts is complex. To simplify the task of handling the interrupt, driver HPS_IRQ has been used from Resources.

The IRQ Interrupt has been configured to be flagged high when button 4 is pressed. When Button 4 is pressed, function *signed int HPS_IRQ_registerHandler (HPSIRQSource interruptID. Sleep_function)* calls interrupt function 'sleep_function' which flags the mode to LOW. When the program continues to execute after the interrupt has occurred, the LOW mode triggers the system to sleep using the function __asm("WFI") after turning off the seven

segment display. The interrupt is reset at the end of interrupt function. When interrupt happens again and the system is in mode LOW, the mode is flipped to HIGH. When the program returns to execute after the interrupt function, identifies the mode HIGH and resets to zero waits for the user to start the timer.

IRQ Interrupt using Interrupt Timer to put the system when the stopwatch has been idle was attempted, but does not work as expected and needs to be fixed.

Additional Functionality:

Additionally, the system blinks for two seconds for every time the counter counts a minute. The blinking functionality is added using a function called 'display_blink' which turns off the 7-segment display when called. The function is called two times at the start of every minute.

Flow of the system:

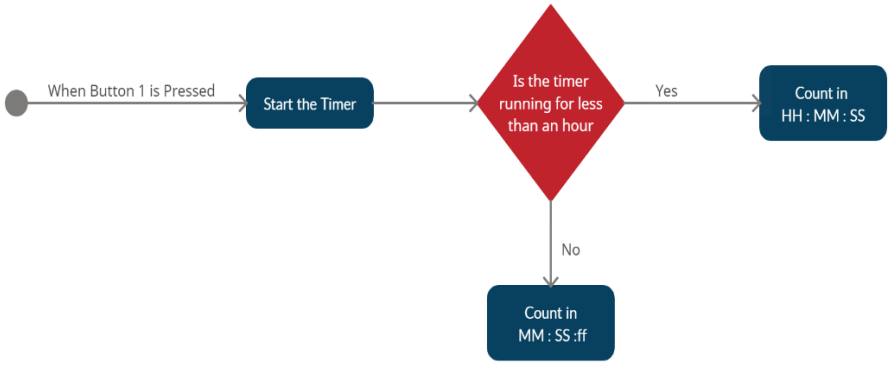

Figure 4. Change of state from MM : SS : ff to HH : MM : SS

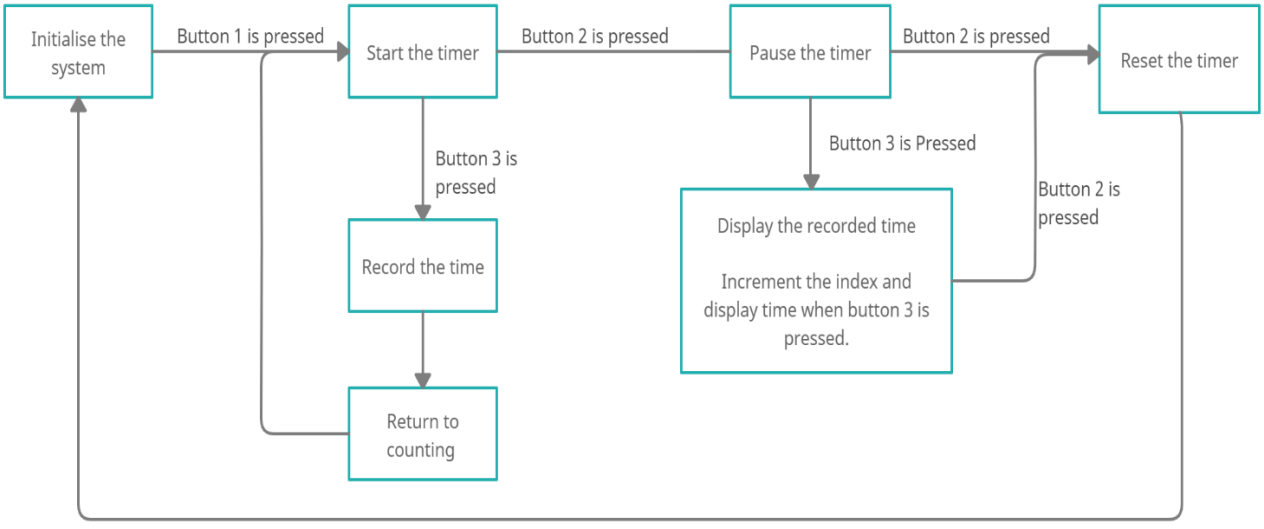

Figure 5. Digital stopwatch program execution flow.

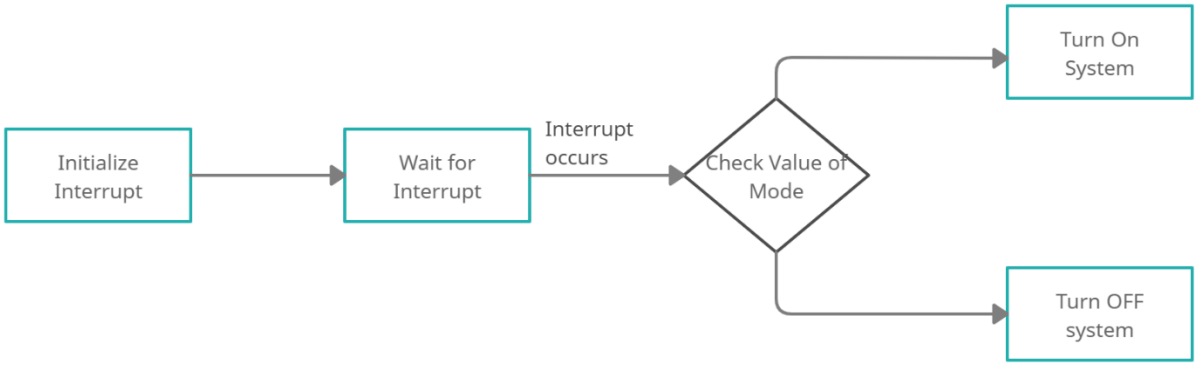

Figure 6. Interrupt execution flow.

Debugging and Error Fixing:

Debugger inside Eclipse is a vital tool to verify the functionality of the program. In order to debug the hardware, DS5 is configured with debug hardware configuration file for DE1-SoC board. Remember to change the scatter file to FPGARomRamIRQ.scat file. The scatter file FPGARomRamIRQ.scat contains the details with regards to vector table.

Breakpoints are used at various places in the code to verify if the program executes in the way the program was intended to. Step over instruction shows the step by step execution of line of code and was vital in execution of the program. Initially, the program didn't stop the timer when button 2 was pressed. The bug was identified with help of break points inside the code.

The code was manually set to start counting from 59th minute. The program flow verified to make the jump from MM : SS : ff counting mode to HH : MM : SS counting mode. Figure 7. Shows the value of Timer Load Register in disassembly view. Using the same tool, It was also verified if the Timer Control Register is loaded with Timer Load Register value when it decrements to zero. Figure 8. Shows the value of interrupt being reset.

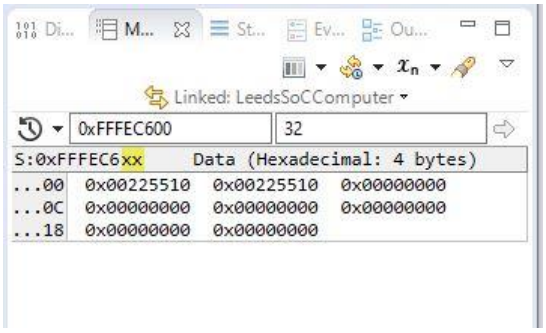

Figure. 7 Disassembly view shows the value of Timer Load Register loading with correct value.

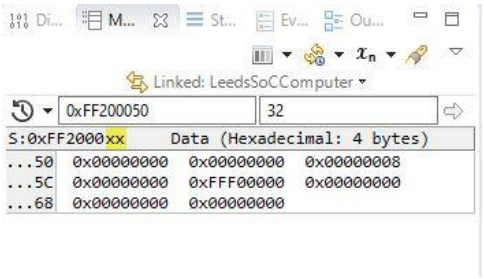

Figure. 8 Disassembly view shows value of the interrupt being reset.

With the help of the debugger tools inside Eclipse, the system has been tested to work for different cases. The system still has bugs in it. The IRQ interrupt when IRQ timer decrements to zero was implemented but has bugs in it and needs to be fixed. When the flow was examined with help of Eclipse Debugger, it was found that the interrupt handler function is not called when IRQ timer decrements to zero. This issue would be fixed and pushed into GitHub repository.

Conclusion:

Digital stopwatch using ARM A9 Private Timer and 7-segment display was designed and its functionality was verified using Eclipse Debugger and DE1-SoC board. This technical assignment gave a deeper insight of the functionality of timers and interrupts in ARM processors. During this technical assignment, one of the motives was to design own drivers for 7-segment displays and push buttons and ARM A9 timers. These rudimentary drivers gave basic understanding of how bit mapping in registers is used to control ARM processor. It also gave an understanding of use of timers for accurate time keeping in ARM A9 processors. Use of watchdog timers in resetting a system when it enters an infinite loop was also understood.

As an additional challenge, functionalities such as split timer, display animations and Interrupt handling was done. It gave a deeper understanding of how ARM handles interrupts. It gave an insight on how interrupts can be used to sleep when an interrupt occurs. Interrupt handler returns to the last executed place when interrupt was actually called. Initially there was lot of bugs when interrupt was used as actual place of return of interrupt is not known. The code was modified in such a way, where ever the program might return, the program works as expected.

Also use of functions, polling, blocking/non-blocking programming methods was understood.

As an improvement to existing system, the bug with IRQ timer would be fixed and additional functionalities such as digital clock, timers would be added in the future versions.

References:

[1]. Oxford English Dictionary. "art, n. 1." OED Online. Oxford University Press, March 2021.

[2]. ARM Cortex A-9 MP Core Technical Reference Manual. ARM Developers.

[3]. DE1-SoC Technical Reference Manual

[4]. Mano, M. Morris. *Computer system architecture*. Prentice-Hall of India, 2003.

Appendix

Main.c

```
/*
 *  Stop Watch Main Function
 *
 *
 *  Created on: Mar 23, 2021
 *      Author: Arul Prakash Samathuvamani | Based on Driver in Unit 1 Examples
 */

#include "SevenSeg.h"
#include "HPS_Watchdog.h"
#include "PrivateTimer.h"
#include "HPS_IRQ.h"
#include "Initialisations.h"

// set private timer interrupt to private timer interrupt base address
volatile unsigned int *private_timer_interrupt_value = (unsigned int *) 0xFFFEC60C;

// set interrupt timer pointer to interrupt timer base address
volatile unsigned int *interrupt_timer_ptr          = (unsigned int *)0xFFC08000;



// Function used to count minutes i.e hour is zero | MM : SS : MS MS

void time_counter_no_hour () {


        // when timer interrupt is high,
        while( ( *private_timer_interrupt_value & 0x1 )  ) {

                reset_interrupt(); // reset the interrupt
```

```
milli_seconds = milli_seconds + 1; // increment milliseconds

if(milli_seconds < 100 ){

        sevenseg_double(0,milli_seconds); // display the updated milliseconds in display

}else if(seconds < 59){ // when milliseconds reaches 100

        if( ((seconds == 60) || (seconds == 2) )&& (minutes > 0)){ // when minute is counted, blink twice
                display_blink(); // turn off the display for 1 second
                milli_seconds = 0; // reset milliseconds
                seconds = seconds + 1; // increment seconds
        }else{
                milli_seconds = 0; // reset milliseconds
                seconds = seconds + 1; // increment seconds
                sevenseg_double(2,seconds); // display the updated seconds
                sevenseg_double(4,minutes); // display the updated minutes
        }

}else if(minutes < 59) { // when the counter is less than an hour, and minute is up


        display_blink(); // blink for one second
        seconds = 0; // reset seconds
        milli_seconds = 0; // reset milliseconds
        minutes = minutes + 1; // increment minute
        //sevenseg_double(4,minutes);

}else{ // else if counter is running for more than an hour

        flag = 0; // set display to HH : MM : SS
        hour = hour + 1; // increment hour for first time
        minutes = 0; // reset minutes
        seconds = 0; // reset seconds
        milli_seconds = 0; // reset milli seconds
        display_blink(); // blink display for a second
        //sevenseg_double(4,hour);
        //sevenseg_double(2,minutes);
```

```
                    intialise_timer ( 225000000 ); // reset time load value to count for one second

                }

            }

}

// function used to count hour | HH : MM : SS

void time_counter_hour() {

// run the loop when the A9 timer interrupt is high
while( ( *private_timer_interrupt_value & 0x1 )  ) {

reset_interrupt(); // reset the interrupt

                                    if( seconds < 59){ // when counter is less than 59 seconds

                                        if( seconds ==2 || seconds == 60){
                                            display_blink(); // turn off the display for 1 second
                                            seconds = seconds + 1; // increment second
                                        }else{
                                            seconds = seconds + 1; // increment second
                                            sevenseg_double(0,seconds); // display updated second
                                            sevenseg_double(2,minutes); // display updated minutes
                                            sevenseg_double(4,hour); // display updated hour
                                        }


                                    }else if(minutes < 59) { // when the counter is less than 59 minutes

                                        display_blink(); // turn off the display for 1 second
                                        seconds = 0; // reset second
                                        minutes = minutes + 1; // increment minute
                                        //sevenseg_double(2,minutes);

                                    }else if (hour < 99 ){ // when hour is less than 99
```

```
                                        display_blink(); // turn off display for one minute
                                        minutes = 0; // reset minute
                                        seconds = 0; // reset seconds
                                        hour = hour + 1; // increment hour
                                        //sevenseg_double(4,hour);
                                        //sevenseg_double(2,minutes);

                                    }else if(hour > 99){ // if hour is more than 99, reset the timer

                                        minutes = 0; //reset minutes
                                        hour    = 0; // reset hour
                                        seconds = 0; //reset seconds
                                        milli_seconds = 0; // reset milliseconds
                                        flag = 1; // set the timer to MM : SS : MS MS mode
                                        intialise_timer ( 2250000 ); // load the timer to count 1 milli second

                                    }

                                }

                HPS_ResetWatchdog(); // reset watchdog

}



unsigned int getPressedKeys () { // To find which key is pressed


    unsigned int key_current_state = *key_ptr; // find what key is pressed

    if( key_current_state != key_last_state){ // if the pressed key is different than previously pressed key

                key_last_state = key_current_state;
                key_pressed = key_current_state; // set the key_pressed to currently pressed key
        }else{
                key_pressed = 0; // if nothing is pressed, set the value to zero.
```

```
            }

    return key_pressed;

}

// Function to capture split timer when button 2 is pressed

void split_timer_capture (){

            SPLITS : if(splits < 20){ // if less than 20 splits is captured

                        split_ms[splits] = milli_seconds; // capture milliseconds
                        split_s[splits]  = seconds;  // capture seconds
                        split_m[splits]  = minutes; //capture minutes
                        split_hr[splits] = hour; //capture hour
                        splits = splits + 1; // increment number of splits captured

            }else{

                        splits = 0; // reset the splits index to zero to count in FIFO way
                        goto SPLITS;
            }

}

// Function used to display the captured split times

void split_timer_display() {


            HPS_ResetWatchdog(); // Reset the watchdog

            if(flag){ // if in MM : SS : MS MS mode

                        sevenseg_double( 0, split_ms[index]); // display the captured milliseconds
                        sevenseg_double(2, split_s[index]); // display the captured seconds
                        sevenseg_double(4,split_m[index]); // display the captured minute
```

```
        }else{ // if in HH : MM : SS mode

                sevenseg_double( 0, split_s[index]); // display captured second
                sevenseg_double(2, split_m[index]); // display captured minute
                sevenseg_double(4,split_hr[index]); // displayu the captured hour

        }

}

// reset the captured split timer data
void flush_split_data(){

        index  = 0; // reset index
        splits = 0; // reset the number of captured splits

        for( loop_variable = 0 ; loop_variable < 20 ; loop_variable++){ // loop to set all the values to zero

                split_s[loop_variable]  = 0;
                split_ms[loop_variable] = 0;
                split_s[loop_variable]  = 0;
                split_hr[loop_variable] = 0;

        }


}

// Function to reset and set all the values to initial state values

void initialise(){


        seven_seg_initialise (); // set the display to display zero

        intialise_timer ( 2250000 ); // set the load value in A9 private timer to count in milliseconds
```

```
        key_pressed = 0; // set the pressed key state to zero

        milli_seconds = 0; // reset milli seconds to zero
        seconds       = 0; // reset the seconds to zero
        hour          = 0; // reset hour to zero
        minutes       = 0; // reset minutes to zero
        flush_split_data(); // call function to reset the split timer values


}

int main(void); // main function declaration


// interrupt function to set the system to sleep or wake up mode

void sleep_function (HPSIRQSource interruptID, bool isInit, void* initParams){

        if(!isInit){

                unsigned int press;

                press = key_ptr[3]; // read push button interrupt register
                key_ptr[3] = press; // set the value again to interrupt register to reset the register

                HPS_ResetWatchdog(); // reset watchdog

                if(mode){ // if interrupt occurs when the system is ON

                        mode = 0; // turn off the system
                        HPS_ResetWatchdog(); // reset watch dog

                }else{ // if interrupt occurs when system is OFF

                        mode = 1; // turn on the system
                        interrupt_timer_ptr[2] = 0;
                }
        }
```

```
}

// Function not working - Interrupt to put the system to sleep when IDLE for some time

void time_out_sleep(HPSIRQSource interruptID, bool isInit, void* initParams){

	if(!isInit){

		unsigned int timer_reset;

		timer_reset = interrupt_timer_ptr[3];

		mode = 1;
	}

	HPS_ResetWatchdog();
}


// Main function declaration

int main(void) {


	// Reset the system at the start
	initialise();

	// Initialise IRQs

	HPS_IRQ_initialise ( NULL );

	// Configure button 4 to call interrupt

	key_ptr[2] = 0x8;

	// configure timer interrupt - does not work
```

```
interrupt_timer_ptr[2] = 0;
interrupt_timer_ptr[0] = 100000000;
interrupt_timer_ptr[2] = 0x03;

// IRQ interrupt handler for timer
HPS_IRQ_registerHandler( IRQ_TIMER_L4SP_0, time_out_sleep);

// Reset watchdog
HPS_ResetWatchdog();

// IRQ interrupt handler for push button press
HPS_IRQ_registerHandler( IRQ_LSC_KEYS, sleep_function);

// Reset watchdog
HPS_ResetWatchdog();


while(1) { // Main Application loop

        interrupt_timer_ptr[2] = 0; // set IRQ timer to OFF

        INITIALISE: if(!mode){ // when the system is put to power down mode

                display_blink(); // turn off the display
                __asm("WFI"); // saving power by putting the processor to sleep

                HPS_ResetWatchdog(); // reset the watchdog

        }else{ // if the system is to be turned on,

                initialise(); // reset the system first
        }

        key_pressed = getPressedKeys (); // check which key is pressed

        HPS_ResetWatchdog(); // reset the watchdog
```

```
if (key_pressed & 0x1){ // start the counter when button 1 is pressed

                while(1){

                key_pressed = getPressedKeys (); // check if other keys are pressed

                if( !(key_pressed & 0x2) && !(key_pressed & 0x4) ){ // if no other key is pressed, continue to count

                        if(!mode){
                                goto INITIALISE; // Used to turn off the system when Interrupt happens
                        }

                        if(flag == 1){ // when counter is running in MM : SS : MS MS mode

                                time_counter_no_hour (); // goto function that counts M M : S S : MS MS
                                HPS_ResetWatchdog(); // reset the watchdog

                        }else{ // when the counter is running in HH : MM : SS mdoe

                                time_counter_hour(); // goto function that counts HH : MM  : SS
                                HPS_ResetWatchdog(); // reset the watchdog

                        }
                        HPS_ResetWatchdog(); // reset the watchdog



                } else if(key_pressed & 0x4){ // if button 3 is pressed then,

                        split_timer_capture (); // goto function that captures the current time

                }else { // else if button 2 is pressed,

                        while (1){

                        HPS_ResetWatchdog(); // Reset the watchdog

                        if(!mode){ // check if the system has been asked to turnoff
```

```
                goto INITIALISE; // turn off the system
            }
            key_pressed = getPressedKeys (); // check which key is pressed

            if( key_pressed & 0x1 ) { // if button 1 is pressed,

                break; // break from the loop and start to count again

            }else if(key_pressed & 0x2){ // if button 2 is pressed,

                initialise(); // Reset the system.
                HPS_ResetWatchdog(); // reset the watchdog

            }else if(key_pressed & 0x4){ // if button 3 is pressed, run in loop to display the captured
splits

                DISPLAY : if(index < splits){ // check if split has been captured,
                    split_timer_display(); // display the split time
                    index++ ; // increase the display index by 1

                }else{ // if all the splits has been displayed, loop again to show the splits

                    index = 0; // set the index to zero
                    goto DISPLAY; // start displaying the splits agani

                }
                    HPS_ResetWatchdog(); // reset the watch dog


            }

        }

        }

    }
}
```

```
        }


}
```

HPS_IRQ_IDs.h – From Unit Reference - Drivers

```
/*
 * HPS IRQ IDs
 * ---------------
 * Description:
 * This file contains an enum type for various peripheral
 * interrupt IDs in the Leeds SoC Computer and Cyclone V HPS.
 *
 */

#ifndef HPS_IRQ_IDS_H_
#define HPS_IRQ_IDS_H_


typedef enum {

/* ARM A9 MPCORE devices (there are many; only a few are defined below) */
   IRQ_MPCORE_GLOBAL_TIMER        = 27,
   IRQ_MPCORE_PRIVATE_TIMER       = 29,
   IRQ_MPCORE_WATCHDOG            = 30,

/* ARM A9 CPU Interrupts (there are 40 in total; you shouldn't need any of them)*/
   IRQ_CPU0_PARITYFAIL            = 32,
   IRQ_CPU0_PARITYFAIL_BTAC       = 33,
   IRQ_CPU0_PARITYFAIL_GHB        = 34,
   IRQ_CPU0_PARITYFAIL_I_TAG      = 35,
   IRQ_CPU0_PARITYFAIL_I_DATA     = 36,
   IRQ_CPU0_PARITYFAIL_TLB        = 37,
   IRQ_CPU0_PARITYFAIL_D_OUTER    = 38,
   IRQ_CPU0_PARITYFAIL_D_TAG      = 39,
   IRQ_CPU0_PARITYFAIL_D_DATA     = 40,
   IRQ_CPU0_DEFLAGS0              = 41,
```

```
    IRQ_CPU0_DEFLAGS1             = 42,
    IRQ_CPU0_DEFLAGS2             = 43,
    IRQ_CPU0_DEFLAGS3             = 44,
    IRQ_CPU0_DEFLAGS4             = 45,
    IRQ_CPU0_DEFLAGS5             = 46,
    IRQ_CPU0_DEFLAGS6             = 47,
    IRQ_CPU1_PARITYFAIL           = 48,
    IRQ_CPU1_PARITYFAIL_BTAC      = 49,
    IRQ_CPU1_PARITYFAIL_GHB       = 50,
    IRQ_CPU1_PARITYFAIL_I_TAG     = 51,
    IRQ_CPU1_PARITYFAIL_I_DATA    = 52,
    IRQ_CPU1_PARITYFAIL_TLB       = 53,
    IRQ_CPU1_PARITYFAIL_D_OUTER   = 54,
    IRQ_CPU1_PARITYFAIL_D_TAG     = 55,
    IRQ_CPU1_PARITYFAIL_D_DATA    = 56,
    IRQ_CPU1_DEFLAGS0             = 57,
    IRQ_CPU1_DEFLAGS1             = 58,
    IRQ_CPU1_DEFLAGS2             = 59,
    IRQ_CPU1_DEFLAGS3             = 60,
    IRQ_CPU1_DEFLAGS4             = 61,
    IRQ_CPU1_DEFLAGS5             = 62,
    IRQ_CPU1_DEFLAGS6             = 63,
    IRQ_SCU_PARITYFAIL0           = 64,
    IRQ_SCU_PARITYFAIL1           = 65,
    IRQ_SCU_EV_ABORT              = 66,
    IRQ_L2_ECC_BYTE_WR            = 67,
    IRQ_L2_ECC_CORRECTED          = 68,
    IRQ_L2_ECC_UNCORRECTED        = 69,
    IRQ_L2_COMBINED               = 70,
    IRQ_DDR_ECC_ERROR             = 71,

/* FPGA interrupts (there are 64 in total; Some have aliases for Leeds SoC Computer) */
    IRQ_INTERVAL_TIMER            = 72, // IRQ_FPGA0  = 72,
    IRQ_LSC_KEYS                  = 73, // IRQ_FPGA1  = 73,
    IRQ_FPGA2                     = 74,
    IRQ_FPGA3                     = 75,
```

```
IRQ_FPGA4                   = 76,
IRQ_FPGA5                   = 77,
IRQ_LSC_AUDIO               = 78, // IRQ_FPGA6  = 78,
IRQ_LSC_PS2_PRIMARY         = 79, // IRQ_FPGA7  = 79,
IRQ_LSC_JTAG                = 80, // IRQ_FPGA8  = 80,
IRQ_LSC_IrDA                = 81, // IRQ_FPGA9  = 81,
IRQ_FPGA10                  = 82,
IRQ_LSC_GPIO_JP1            = 83, // IRQ_FPGA11 = 83,
IRQ_LSC_GPIO_JP2            = 84, // IRQ_FPGA12 = 84,
IRQ_FPGA13                  = 85,
IRQ_FPGA14                  = 86,
IRQ_FPGA15                  = 87,
IRQ_FPGA16                  = 88,
IRQ_LSC_PS2_SECONDARY       = 89, // IRQ_FPGA17 = 89,
IRQ_FPGA18                  = 90,
IRQ_FPGA19                  = 91,
IRQ_FPGA20                  = 92,
IRQ_FPGA21                  = 93,
IRQ_FPGA22                  = 94,
IRQ_FPGA23                  = 95,
IRQ_FPGA24                  = 96,
IRQ_FPGA25                  = 97,
IRQ_FPGA26                  = 98,
IRQ_FPGA27                  = 99,
IRQ_FPGA28                  = 100,
IRQ_FPGA29                  = 101,
IRQ_FPGA30                  = 102,
IRQ_FPGA31                  = 103,
IRQ_FPGA32                  = 104,
IRQ_FPGA33                  = 105,
IRQ_FPGA34                  = 106,
IRQ_FPGA35                  = 107,
IRQ_FPGA36                  = 108,
IRQ_FPGA37                  = 109,
IRQ_FPGA38                  = 110,
IRQ_FPGA39                  = 111,
```

```
    IRQ_FPGA40              = 112,
    IRQ_FPGA41              = 113,
    IRQ_FPGA42              = 114,
    IRQ_FPGA43              = 115,
    IRQ_FPGA44              = 116,
    IRQ_FPGA45              = 117,
    IRQ_FPGA46              = 118,
    IRQ_FPGA47              = 119,
    IRQ_FPGA48              = 120,
    IRQ_FPGA49              = 121,
    IRQ_FPGA50              = 122,
    IRQ_FPGA51              = 123,
    IRQ_FPGA52              = 124,
    IRQ_FPGA53              = 125,
    IRQ_FPGA54              = 126,
    IRQ_FPGA55              = 127,
    IRQ_FPGA56              = 128,
    IRQ_FPGA57              = 129,
    IRQ_FPGA58              = 130,
    IRQ_FPGA59              = 131,
    IRQ_FPGA60              = 132,
    IRQ_FPGA61              = 133,
    IRQ_FPGA62              = 134,
    IRQ_FPGA63              = 135,

/* HPS device interrupts (76 in total) */
    IRQ_DMA0                = 136,
    IRQ_DMA1                = 137,
    IRQ_DMA2                = 138,
    IRQ_DMA3                = 139,
    IRQ_DMA4                = 140,
    IRQ_DMA5                = 141,
    IRQ_DMA6                = 142,
    IRQ_DMA7                = 143,
    IRQ_DMA_ABORT           = 144,
    IRQ_DMA_ECC_CORRECTED   = 145,
```

IRQ_DMA_ECC_UNCORRECTED = 146,

IRQ_EMAC0 = 147,

IRQ_EMAC0_TX_ECC_CORRECTED = 148,

IRQ_EMAC0_TX_ECC_UNCORRECTED = 149,

IRQ_EMAC0_RX_ECC_CORRECTED = 150,

IRQ_EMAC0_RX_ECC_UNCORRECTED = 151,

IRQ_EMAC1 = 152,

IRQ_EMAC1_TX_ECC_CORRECTED = 153,

IRQ_EMAC1_TX_ECC_UNCORRECTED = 154,

IRQ_EMAC1_RX_ECC_CORRECTED = 155,

IRQ_EMAC1_RX_ECC_UNCORRECTED = 156,

IRQ_USB0 = 157,

IRQ_USB0_ECC_CORRECTED = 158,

IRQ_USB0_ECC_UNCORRECTED = 159,

IRQ_USB1 = 160,

IRQ_USB1_ECC_CORRECTED = 161,

IRQ_USB1_ECC_UNCORRECTED = 162,

IRQ_CAN0_STS = 163,

IRQ_CAN0_MO = 164,

IRQ_CAN0_ECC_CORRECTED = 165,

IRQ_CAN0_ECC_UNCORRECTED = 166,

IRQ_CAN1_STS = 167,

IRQ_CAN1_MO = 168,

IRQ_CAN1_ECC_CORRECTED = 169,

IRQ_CAN1_ECC_UNCORRECTED = 170,

IRQ_SDMMC = 171,

IRQ_SDMMC_PORTA_ECC_CORRECTED = 172,

IRQ_SDMMC_PORTA_ECC_UNCORRECTED = 173,

IRQ_SDMMC_PORTB_ECC_CORRECTED = 174,

IRQ_SDMMC_PORTB_ECC_UNCORRECTED = 175,

IRQ_NAND = 176,

IRQ_NANDR_ECC_CORRECTED = 177,

IRQ_NANDR_ECC_UNCORRECTED = 178,

IRQ_NANDW_ECC_CORRECTED = 179,

IRQ_NANDW_ECC_UNCORRECTED = 180,

IRQ_NANDE_ECC_CORRECTED = 181,

```
    IRQ_NANDE_ECC_UNCORRECTED       = 182,
    IRQ_QSPI                        = 183,
    IRQ_QSPI_ECC_CORRECTED          = 184,
    IRQ_QSPI_ECC_UNCORRECTED        = 185,
    IRQ_SPI0                        = 186,
    IRQ_SPI1                        = 187,
    IRQ_SPI2                        = 188,
    IRQ_SPI3                        = 189,
    IRQ_I2C0                        = 190,
    IRQ_I2C1                        = 191,
    IRQ_I2C2                        = 192,
    IRQ_I2C3                        = 193,
    IRQ_UART0                       = 194,
    IRQ_UART1                       = 195,
    IRQ_GPIO0                       = 196,
    IRQ_GPIO1                       = 197,
    IRQ_GPIO2                       = 198,
    IRQ_TIMER_L4SP_0                = 199, //HPS Timer 0
    IRQ_TIMER_L4SP_1                = 200, //HPS Timer 1
    IRQ_TIMER_OSC1_0                = 201,
    IRQ_TIMER_OSC1_1                = 202,
    IRQ_WDOG0                       = 203, //HPS Watchdog 0
    IRQ_WDOG1                       = 204,
    IRQ_CLKMGR                      = 205,
    IRQ_MPUWAKEUP                   = 206,
    IRQ_FPGA_MAN                    = 207,
    IRQ_NCTIIRQ_0                   = 208,
    IRQ_NCTIIRQ_1                   = 209,
    IRQ_RAM_ECC_CORRECTED           = 210,
    IRQ_RAM_ECC_UNCORRECTED         = 211

} HPSIRQSource;



#endif /* HPS_IRQ_IDS_H_ */
```

HPS_IRQ.c – From Unit Reference Drivers

```
/*
 * Cyclone V HPS Interrupt Controller
 * ----------------------------------
 * Description:
 * Driver for enabling and using the General Interrupt
 * Controller (GIC). The driver includes code to create
 * a vector table, and register interrupts.
 *
 * The code makes use of function pointers to register
 * interrupt handlers for specific interrupt IDs.
 *
 * Company: University of Leeds
 * Author: T Carpenter
 *
 * Change Log:
 *
 * Date       | Changes
 * -----------+-----------------------------------
 * 10/09/2018 | Embed creation of Vector Table and setup of VBAR
 *            | into driver to avoid extra assembly files.
 *            | Add ability to use static variables in ISR handlers
 * 12/03/2018 | Creation of driver
 *
 */

#include "HPS_IRQ.h"



/*
 * Declare various useful pointer addresses in the Cyclone V HPS
 */

#define CPSR_M  0
#define CPSR_T  5
#define CPSR_F  6
```

```
#define CPSR_I  7
#define CPSR_A  8
#define CPSR_E  9
#define CPSR_GE 16
#define CPSR_J  24
#define CPSR_Q  27
#define CPSR_V  28
#define CPSR_C  29
#define CPSR_Z  30
#define CPSR_N  31

// Interrupt controller (GIC) CPU interface(s)
#define MPCORE_GIC_CPUIF     0xFFFEC100                 // PERIPH_BASE + 0x100
#define ICCICR               (0x00/sizeof(unsigned int))  // + to CPU interface control
#define ICCPMR               (0x04/sizeof(unsigned int))  // + to interrupt priority mask
#define ICCIAR               (0x0C/sizeof(unsigned int))  // + to interrupt acknowledge
#define ICCEOIR              (0x10/sizeof(unsigned int))  // + to end of interrupt reg

// Interrupt (INT) controller (GIC) distributor interface(s)
#define MPCORE_GIC_DIST      0xFFFED000                 // PERIPH_BASE + 0x1000
#define ICDDCR               (0x000/sizeof(unsigned int)) // + to distributor control reg
#define ICDISER              (0x100/sizeof(unsigned int)) // + to INT set-enable regs
#define ICDICER              (0x180/sizeof(unsigned int)) // + to INT clear-enable regs
#define ICDIPTR              (0x800/sizeof(unsigned int)) // + to INT processor targets regs
#define ICDICFR              (0xC00/sizeof(unsigned int)) // + to INT configuration regs



/*
 * Global variables for the IRQ driver
 */

bool irq_isInitialised = false;

typedef struct {
    HPSIRQSource interruptID;  //The ID of the interrupt source this handler is for
    isr_handler_func handler; //Function pointer to be called to handle this ID
```

```
    bool enabled;
} isr_handler;

unsigned int irq_isr_handler_count;
isr_handler* irq_isr_handlers;
isr_handler_func irq_isr_unhandledIRQCallback;

volatile unsigned int* irq_gic_cpuif_ptr = (unsigned int *)MPCORE_GIC_CPUIF;
volatile unsigned int* irq_gic_dist_ptr  = (unsigned int *)MPCORE_GIC_DIST;

/*
 * Here we create a vector table using an inline assembly function. This function is
 * not compiled using the C compiler, but fed directly to the Linker.
 * In the table we have two types of entry:
 *   LDR PC,=<functionName> ; Sets Page Counter equal to function address.
 *   DCD 0 ; will trigger a jump to "Undefined Instruction" vector
 */
__asm __attribute__((section("vector_table"))) void vector_table(void) {
    //Ensure 8-byte alignment of all used IRQs is preserved.
    PRESERVE8
    //Import names for various functions which are "extern"al to the __asm block.
    extern __main             ; // Program start
    extern __rt_entry         ; // C Standard Library entry point
    extern __irq_isr          ; // IRQ Interrupt Handler
    //Vector Table Entries
    LDR       PC,=__main     ; // 0x00 Reset
    LDR       PC,=__rt_entry ; // 0x04 Undefined Instructions
    DCD       0              ; // 0x08 Software Interrupts (SWI)
    DCD       0              ; // 0x0C Prefetch Abort
    DCD       0              ; // 0x10 Data Abort
    DCD       0              ; // 0x14 RESERVED
    LDR       PC,=__irq_isr  ; // 0x18 IRQ
    DCD       0              ; // 0x1C FIQ
}

/*
```

```
 * Next we need our interrupt service routine for IRQs
 *
 * This will check the interrupt id against all registered handlers
 * and cause an unhandledIRQCallback call if it is an unhandled interrupt
 */

__irq void __irq_isr (void) {
    bool isr_handled = false;
    // Read the ICCIAR value to get interrupt ID
    HPSIRQSource int_ID = (HPSIRQSource)irq_gic_cpuif_ptr[ICCIAR];
    // Check to see if we have a registered handler
    unsigned int handler;
    for (handler = 0; handler < irq_isr_handler_count; handler++) {
        if (int_ID == irq_isr_handlers[handler].interruptID) {
            //If we have found a handler for this ID
            irq_isr_handlers[handler].handler(int_ID, false, NULL); //Call it
            isr_handled = true; //And mark as handled
            break;
        }
    }
    //Check if we have an unhandled interrupt
    if (!isr_handled) {
        irq_isr_unhandledIRQCallback(int_ID, false, NULL); //Call the unhandled IRQ callback.
    }

    //Otherwise write to the End of Interrupt Register (ICCEOIR) to mark as handled
    irq_gic_cpuif_ptr[ICCEOIR] = (unsigned int)int_ID;
    //And done.
    return;
}

//Default handler for unhandled interrupt callback.
void HPS_IRQ_unhandledIRQ(HPSIRQSource interruptID, bool isInit, void* initParams) {
    while(1); //Crash - use the watchdog timer to reset us.
}
```

```
// Stack Pointer can only be written in ASM function, not C function
__asm void HPS_IRQ_WriteStackPointer(unsigned int val) {
    MOV SP, r0
    BX LR
}

//Function to configure the stack in IRQ mode
void HPS_IRQ_ConfigureIRQStack(void) {
    register unsigned int cpsr __asm("cpsr");
    cpsr = (1 << CPSR_I) | (1 << CPSR_F) | (18 << CPSR_M); //Switch context to IRQ mode
    HPS_IRQ_WriteStackPointer(0xFFFFFFF8); //Set the stack pointer
    cpsr = (1 << CPSR_I) | (1 << CPSR_F) | (19 << CPSR_M); //Switch context back to SVC mode
}

//Initialise HPS IRQ Driver
signed int HPS_IRQ_initialise( isr_handler_func userUnhandledIRQCallback ) {
    //Magic assembly lookup command to get the VBAR (vector table base address register)
    register unsigned int cp15_VBAR __asm("cp15:0:c12:c0:0");
    /* Configure Global Interrupt Controller (GIC) */
    //Disable interrupts before configuring
    __disable_irq();

    // Set the location of the vector table using VBAR register
    cp15_VBAR = (unsigned int)&vector_table;

    // Set Interrupt Priority Mask Register (ICCPMR)
    // Enable interrupts of all priorities
    irq_gic_cpuif_ptr[ICCPMR] = 0xFFFF;

    // Set CPU Interface Control Register (ICCICR)
    // Enable signalling of interrupts
    irq_gic_cpuif_ptr[ICCICR] = 0x1;

    // Configure the Distributor Control Register (ICDDCR)
    // Send pending interrupts to CPUs
```

```
    irq_gic_dist_ptr[ICDDCR] = 0x1;

    //Configure the IRQ mode stack
    HPS_IRQ_ConfigureIRQStack();

    //Initially no handlers
    irq_isr_handler_count = 0;
    irq_isr_handlers = 0x0;

    //Set up the unhandled IRQ callback
    if (userUnhandledIRQCallback != NULL) {
        //If the user has supplied one, use theirs
        irq_isr_unhandledIRQCallback = userUnhandledIRQCallback;
    } else {
        //Otherwise use default
        irq_isr_unhandledIRQCallback = HPS_IRQ_unhandledIRQ;
    }

    //Enable interrupts again
    __enable_irq();

    //Mark as initialised
    irq_isInitialised = true;
    //And done
    return HPS_IRQ_SUCCESS;
}

//Check if driver initialised
// - returns true if initialised
bool HPS_IRQ_isInitialised() {
    return irq_isInitialised;
}

signed int HPS_IRQ_registerHandler(HPSIRQSource interruptID, isr_handler_func handlerFunction) {
    isr_handler* new_handler;
    unsigned int handler;
```

```
volatile unsigned char* diptr;
bool was_masked;
if (!HPS_IRQ_isInitialised()) return HPS_IRQ_ERRORNOINIT;
//First check if a handler already exists (we can overwrite it if it does)
for (handler = 0; handler < irq_isr_handler_count; handler++) {
    if (irq_isr_handlers[handler].interruptID == interruptID) {
        //Found an existing one. We'll just overwrite it, instead of making new one.
        break;
    }
}
//Before changing anything we need to mask interrupts temporarily while we change the handlers
was_masked = __disable_irq();
if (handler == irq_isr_handler_count) {
    //If we failed to find a match, we need to reallocated our handler array to gain more space
    new_handler = (isr_handler*)realloc(irq_isr_handlers, (irq_isr_handler_count + 1)*sizeof(isr_handler) );
    //We should be sure to check that reallocation was a success
    if (new_handler == 0x0) {
        //If realloc returned null, then reallocation failed, cannot register new handler.
        return HPS_IRQ_NONENOUGHMEM;
    }
    //If we were successful in making space, update our global pointer
    irq_isr_handlers = new_handler;
    //Update the count of available handlers
    irq_isr_handler_count = irq_isr_handler_count + 1;
}
//Add our new handler to the end
irq_isr_handlers[handler].handler = handlerFunction;
irq_isr_handlers[handler].interruptID = interruptID;
irq_isr_handlers[handler].enabled = true;
//Then we need to enable the interrupt in the distributor
irq_gic_dist_ptr[ICDISER + (interruptID / 32)] = 1 << (interruptID % 32);
//And set the affinity to CPU0
diptr = (unsigned char*)&(irq_gic_dist_ptr[ICDIPTR + interruptID / 4]);
diptr[interruptID % 4] = 0x1;
//Finally we unmask interrupts to resume processing.
if (!was_masked) {
```

```
        __enable_irq();
    }
    //And done.
    return HPS_IRQ_SUCCESS;
}

signed int HPS_IRQ_unregisterHandler(HPSIRQSource interruptID) {
    unsigned int handler;
    bool was_masked;
    if (!HPS_IRQ_isInitialised()) return HPS_IRQ_ERRORNOINIT;
    //See if we can find the requested handler
    for (handler = 0; handler < irq_isr_handler_count; handler++) {
        if (irq_isr_handlers[handler].interruptID == interruptID) {
            //Found it.
            //Before changing anything we need to mask interrupts temporarily while we change the handlers
            was_masked = __disable_irq();
            //Clear the handler pointer, and mark as disabled
            irq_isr_handlers[handler].handler = 0x0;
            irq_isr_handlers[handler].enabled = false;
            //Then we need to disable the interrupt in the distributor
            irq_gic_dist_ptr[ICDICER + (interruptID / 32)] = 1 << (interruptID & 31);
            //Finally we unmask interrupts to resume processing.
            if (!was_masked) {
                __enable_irq();
            }
            //And done
            return HPS_IRQ_SUCCESS;
        }
    }
    //Whoops, handler doesn't exist.
    return HPS_IRQ_NOTFOUND;
}
```

HPS_IRQ.h – From Unit Resource Drivers

```
/*
 * Cyclone V HPS Interrupt Controller
 * -----------------------------------
 * Description:
 * Driver for enabling and using the General Interrupt
 * Controller (GIC). The driver includes code to create
 * a vector table, and register interrupts.
 *
 * The code makes use of function pointers to register
 * interrupt handlers for specific interrupt IDs.
 *
 * Company: University of Leeds
 * Author: T Carpenter
 *
 * Change Log:
 *
 * Date       | Changes
 * -----------+-----------------------------------
 * 12/03/2018 | Creation of driver
 *
 */

#ifndef HPS_IRQ_H_
#define HPS_IRQ_H_

#include <stdbool.h>
#include <stdlib.h>

#define HPS_IRQ_SUCCESS        0
#define HPS_IRQ_ERRORNOINIT   -1
#define HPS_IRQ_NOTFOUND      -2
#define HPS_IRQ_NONENOUGHMEM  -4



//Include a list of IRQ IDs that can be used while registering interrupts
```

```
#include "HPS_IRQ_IDs.h"

//Function Pointer Type for Interrupt Handlers
// - interruptID is the ID of the interrupt that called the handler.
// - When IRQ handler is called by interrupt, 'isInit' is false and 'initParams'
//   can be ignored.
// - If variables need to be shared with the handler, the user can call the irq
//   handler manually with 'isInit' set to true, and 'initParams' a pointer to
//   the variables that need to be shared.
typedef void (*isr_handler_func)(HPSIRQSource interruptID, bool isInit, void* initParams);
// Two examples of IRQ handlers are as follows:
/* ---Start Examples---
//Handler not needing parameters
void exampleIRQHandlerWithNoParams(HPSIRQSource interruptID, bool isInit, void* initParams) {
    if (!isInit) {
       //IRQ handler stuff...

    }
}
//Handler needing parameters
void exampleIRQHandlerUsingParams(HPSIRQSource interruptID, bool isInit, void* initParams) {
    static someVarType* params = NULL;
    if (isInit) {
       //Initialise the params variable.
        params = (someVarType*)initParams;
    } else if (params != NULL) {
       //IRQ handler stuff...

    } //Optionally, else { //Do something if interrupted before initialised  }
}
// ---End Examples--- */



//Initialise HPS IRQ Driver
// - userUnhandledIRQCallback is either a function pointer to an isr_handler to
//   be called in the event of an unhandled IRQ occurring. If this parameter is
```

```
//   passed as NULL (0x0), a default handler which causes crash by watchdog will
//   be used.
// - Returns HPS_IRQ_SUCCESS if successful.
signed int HPS_IRQ_initialise(isr_handler_func userUnhandledIRQCallback );

//Check if driver initialised
// - Returns true if driver previously initialised
bool HPS_IRQ_isInitialised(void);

//Register a new interrupt ID handler
// - interruptID is the number between 0 and 255 of the interrupt being configured
// - handlerFunction is a function pointer to the function that will be called when IRQ with ID occurs
// - if a handler already exists for the specified ID, it will be replaced by the new one.
// - the interrupt ID will be enabled in the GIC
// - returns HPS_IRQ_SUCCESS on success.
// - returns HPS_IRQ_NONENOUGHMEM if failed to reallocated handler array.
signed int HPS_IRQ_registerHandler(HPSIRQSource interruptID, isr_handler_func handlerFunction);

//Unregister interrupt ID handler
// - interruptID is the number between 0 and 255 of the interrupt being configured
// - the interrupt will be disabled also in the GIC
// - returns HPS_IRQ_SUCCESS on success.
// - returns HPS_IRQ_NOTFOUND if handler not found
signed int HPS_IRQ_unregisterHandler(HPSIRQSource interruptID);

#endif /* HPS_IRQ_H_ */
```

HPS_WatchDog.h – As in Unit Reference Drivers

```
/*
 * HPS Watchdog Reset
 * ------------------------------
 * Description:
 * Simple inline functions for resetting watchdog and
 * returning the current watchdog timer value.
 *
 * Company: University of Leeds
 * Author: T Carpenter
 *
 */

#ifndef HPS_WATCHDOG_H_
#define HPS_WATCHDOG_H_

//#define for backwards compatibility
#define ResetWDT() HPS_ResetWatchdog()

// Function to reset the watchdog timer.
__forceinline static void HPS_ResetWatchdog() {
    *((volatile unsigned int *) 0xFFD0200C) = 0x76;
}

// Function to get value of the watchdog timer
__forceinline static unsigned int HPS_WatchdogValue() {
    return *((volatile unsigned int *) 0xFFD02008);
}

#endif /* HPS_WATCHDOG_H_ */
```

Initialisations.c

```
/*
 * Initialisations.c - Stop Watch
 *
 * This Programs declares all the variables and certain functions to be used in the Main Program
 *
 * This separate header is created to make understanding the need for declaration of variables more easier
 *
 *  Created on: Mar 26, 2021
 *      Author: Arul Prakash Samathuvamani | el20a2ps@leeds.ac.uk
 */



// function used to set all the values of 7-segment display to zero.

#include "SevenSeg.h"

void seven_seg_initialise () {

---// sets all the values in seven segment display to 1

sevenseg_single(0,0);
sevenseg_single(1,0);
sevenseg_single(2,0);
sevenseg_single(3,0);
sevenseg_single(4,0);
sevenseg_single(5,0);

}

// function used to turn of the display, and blink when counter counts a minute. Turns of the 7-segment display.

void display_blink () {

// turns off the seven segment display.
```

```
sevenseg_single(0,10);
sevenseg_single(1,10);
sevenseg_single(2,10);
sevenseg_single(3,10);
sevenseg_single(4,10);
sevenseg_single(5,10);


}
```

Initialisations.h

```
/*
 * Initialisations.h - Stop Watch
 *
 * This Programs declares all the variables and certain functions to be used in the Main Program
 *
 * This separate header is created to make understanding the need for declaration of variables more easier
 *
 *  Created on: Mar 26, 2021
 *      Author: Arul Prakash Samathuvamani | el20a2ps@leeds.ac.uk
 */

#ifndef INITIALISATIONS_H_
#define INITIALISATIONS_H_


// Declaration of Key_ptr pointer that points to key_press address. 4-bit address, changes accordingly to key press.

volatile unsigned int *key_ptr = (unsigned int *) 0xFF200050;

// Function variables used to indentify the pressed key.

unsigned int key_last_state = 0; // Last key press
unsigned int key_pressed; // denotes what is the currently pressed key


// Counter Function Variables
```

```
int milli_seconds = 0 ; // denotes milli_seconds
int seconds       = 0 ; // denotes seconds
int minutes       = 0 ; // denotes minutes
int hour = 0 ; // denotes hour


int flag   = 1; // Used to denote if the system is counting in hours or minutes. If counting in minutes, the flag value is 1, if hour flag is 0

int splits = 0; // Variable used for split timer function. Used to count the number of times split timer button is pressed.

// Split timer variables. Used to store the time when split timer button is pressed. Can store only 20 split times.

int split_ms[20]; // to store milli-seconds
int split_s[20]; // to store seconds
int split_m[20]; // to store minutes
int split_hr[20]; // to store hour

// Variables used to print the split timer values

int index; // denotes current array index in split timer array
int loop_variable; // variable used inside for loop


int mode = 1; // Used for interrupts. When mode is 1, system is ON and functions. When mode is 0, then system is TURNED OFF.

void seven_seg_initialise (void); // function used to set all the values of 7-segment display to zero.

void display_blink (void); // function used to turn of the display, and blink when counter counts a minute. Turns of the 7-segment display.


#endif /* INITIALISATIONS_H_ */
```

PrivateTimer.c

```
/*
 * PrivateTimer.c- Stop Watch
```

```
 *
 *  Created on: Mar 24, 2021
 *      Author: Arul Prakash Samathuvamani | el20a2ps@leeds.ac.uk
 */

#include "PrivateTimer.h"

// points the pointers to corresponding base addresses

// set timer pointer to interrupt timer base address
volatile unsigned int *timer_base_ptr          = (unsigned int *)0xFFFEC600;

// set private timer load pointer to timer load address
volatile unsigned int *private_timer_load      = (unsigned int *)0xFFFEC600;

// set private timer pointer to timer value address
volatile unsigned int *private_timer_value     = (unsigned int *)0xFFFEC604;

// set private timer control pointer to timer control base address
volatile unsigned int *private_timer_control   = (unsigned int *)0xFFFEC608;

// set private timer interrupt to private timer interrupt base address
volatile unsigned int *private_timer_interrupt = (unsigned int *)0xFFFEC60C;



// function to initialise the timer
void intialise_timer ( signed int timer_load_value ) {


*private_timer_load = timer_load_value ; // set the timer load value to timer load address

// set PRESCALAR value of timer to zero. E = 1, A =1, I =0. i.e enable the timer
// dis
*private_timer_control = ( 0 << 8 ) | (0 << 2) | (1 << 1) | (1 << 0);
```

```
}

// Check if the timer has reached counting down to zero, if yes interrrupt becomes high.
unsigned int interrupt_status ( ) {

unsigned int interrupt_value = *private_timer_interrupt; // declaration used for testing

// return the value of interrupt.
return *private_timer_interrupt;

}


// reset the timer interrupt. Timer starts to count again.
void reset_interrupt ( ) {

// reset the value of interrupt.
*private_timer_interrupt = 0x1;

}
```

PrivateTimer.h

```
/*
 * PrivateTimer.c- Stop Watch
 *
 *  Created on: Mar 24, 2021
 *      Author: Arul Prakash Samathuvamani | el20a2ps@leeds.ac.uk
 */


#ifndef PRIVATETIMER_H_
#define PRIVATETIMER_H_
```

```
// function to initialise the timer
void intialise_timer( signed int timer_load_value );

// Check if the timer has reached counting down to zero, if yes interrrupt becomes high.
unsigned int interrupt_status ( void ) ;

// reset the timer interrupt. Timer starts to count again.
void reset_interrupt ( void );

#endif /* PRIVATETIMER_H_ */
```

SevenSeg.c

```
/*
 * SevenSeg.c
 *
 *7-Segment Display Driver
 *
 *  Created on: Mar 23, 2021
 *      Author: Arul Prakash Samathuvamani | Based on Driver in Unit 1 Examples
 */

#include "SevenSeg.h"

/* Driver Functionality
 *
 * 7-segment display displays the value denoted by 7-bit address.
 *
 * X X X X X X X -  The corresponding dash turns on and off based on the value of X. Please refer technical report for mapping of bits.
 *
 */



// set the lower base address for 7-segment display. Denotes display 0-4
volatile unsigned char *sevenseg_base_low_addr = (unsigned char *) 0xFF200020;
```

```
// set the higher base address for 7-segment display. Dentoes display 5 and 6
volatile unsigned char *sevenseg_base_high_addr = (unsigned char *) 0xFF200030;

#define number_of_low_display 4 // Define the number of displays addressed by lower base address.

#define number_of_high_display 2 // Define the number of displays addressed by higher base address.


// Write the corresponding value to 7-Segment memory address.
void sevenseg_write(unsigned int display, unsigned char value){

if(display < number_of_low_display ) { // if the display number is in lower base

sevenseg_base_low_addr[display] = value; // write the value to lower base address

}else{

display = display - number_of_low_display; // else find the corresponding display in higher base address
sevenseg_base_high_addr[display] = value; // and write the value to higher base address
                                                                                                                    }

}

// Sets the value for single display
void sevenseg_single(unsigned int display, unsigned int value){


if(value < 10 ){ // for a single display, values can be from 0-9 for stop watch or the display is turned off

if ( value == 0 ) {

sevenseg_write(display, 63); // If the value is zero , write 111110 decimal.
}else if ( value == 1 ) {

sevenseg_write(display, 6); // if the value is 1, write the corresponding decimal.
```

```
}else if ( value == 2 ) {

sevenseg_write(display, 91); // if the value is 2, write the corresponding decimal.
}else if ( value == 3 ){

sevenseg_write(display, 79); // if the value is 3, write the corresponding decimal.
}else if( value == 4) {

sevenseg_write(display, 102); // if the value is 4, write the corresponding decimal.
}else if ( value == 5) {

sevenseg_write(display, 109); // if the value is 5, write the corresponding decimal.
}else if(value == 6){

sevenseg_write(display, 125); // if the value is 6, write the corresponding decimal.
}else if(value == 7){

sevenseg_write(display, 7); // if the value is 7, write the corresponding decimal.
}else if(value == 8){

sevenseg_write(display, 127); // if the value is 8, write the corresponding decimal
}else if(value == 9){

sevenseg_write(display, 103); // if the value is 9, write the corresponding decimal.
}

}else{
sevenseg_write(display, 0); // else turn off the display.
}




}

// used to set value in two displays.
```

```
void sevenseg_double(unsigned int display, unsigned int value){



// if the value is less than 10, then only one display is needed
if(value <10){

sevenseg_single  (display,value); // display the value in first display
sevenseg_single  (display+1,0); // and display zero in the next

}
// if the value is greater than 10, then both the displays are needed
else{

sevenseg_single (display,(value % 10)); // display the lower digit in first display
sevenseg_single (display+1,( value / 10 ) ); // display the higher digit in second display

}


}
```

SevenSeg.h

```
/*
 * SevenSeg.h
 *
 *7-Segment Display Driver
 *
 *  Created on: Mar 23, 2021
 *      Author: Arul Prakash Samathuvamani | Based on Driver design in Unit 1 Examples.
 */

#ifndef SEVENSEG_H_
#define SEVENSEG_H_
```

```
void sevenseg_write (unsigned int display , unsigned char value);

void sevenseg_single(unsigned int display, unsigned int value);

void sevenseg_double(unsigned int display, unsigned int value);

#endif /* SEVENSEG_H_ */
```