Lab Assignment 3

Jose Monroy Villalobos

San José State University
Department of Computer Engineering
CMPE 146-01, Real-Time Embedded System Co-Design, Fall 2024

Exercise 1.1 Simple Checksum

Function:

```
uint32_t compute_simple_checksum(uint8_t* data, uint32_t length){
   uint32_t i;
   uint32_t checksum = 0;
   for(i = 0; i<length*4; i++)
   {
      if(i<4)
         printf("\ndata[%i]: 0x%08X    Current checksum: 0x%08X \n", i, data[i], checksum);
      if(i>(length*4)-5)
               printf("\ndata[%i]: 0x%08X    Current checksum: 0x%08X \n", i, data[i],
checksum);
      checksum += (uint32_t)data[i] << ((i%4)*8);
   }
   checksum= ~checksum;
   return checksum;
}
```

Debug console Output:

[CORTEX_M4_0] Checksum: 0xF25B8806

One method to speed up the computation would be to process 4 bytes per loop to reduce the number of loops needed and reduce overhead.

Exercise 1.2 Speed Up

Code that does measurement:

```
   uint32_t checksum;
   t1 = MAP_Timer32_getValue(TIMER32_0_BASE);
   checksum = compute_simple_checksum(myData, 2560);
```

```
t0 = MAP_Timer32_getValue(TIMER32_0_BASE);

press_duration_cycles = t1 - t0;
simple_time = (press_duration_cycles / (float)mclk_freq) * 1000000;
printf("simple_time: %.3f us\n", simple_time);

MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
t1 = MAP_Timer32_getValue(TIMER32_0_BASE);
for (ii = 0; ii < 4*2560; ii++)
    MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);

 /* Getting the result from the hardware module */
 hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;
 t0 = MAP_Timer32_getValue(TIMER32_0_BASE);

    press_duration_cycles = t1 - t0;
    hwCalculatedCRC_time = (press_duration_cycles / (float)mclk_freq) * 1000000;
    printf("hwCalculatedCRC_time: %.3f us\n", hwCalculatedCRC_time);
 /* Calculating the CRC32 checksum through software */

    t1 = MAP_Timer32_getValue(TIMER32_0_BASE);
 swCalculatedCRC = calculateCRC32((uint8_t*) myData, 4*2560);
 t0 = MAP_Timer32_getValue(TIMER32_0_BASE);

        press_duration_cycles = t1 - t0;
        swCalculatedCRC_time = (press_duration_cycles / (float)mclk_freq) * 1000000;
        printf("swCalculatedCRC_time: %.3f us\n", swCalculatedCRC_time);
```

```
        float speedup = 0;
        speedup = swCalculatedCRC_time / hwCalculatedCRC_time;
        printf("speedup: %.2f%%\n", speedup*100);
    printf("Checksum: 0x%08X\n", checksum);

    printf("hwCalculatedCRC: 0x%08X\n", hwCalculatedCRC);
    printf("swCalculatedCRC: 0x%08X\n", swCalculatedCRC);
```

Console Output:

```
[CORTEX_M4_0] 3000000


data[0]: 0x0F


data[1]: 0x62


data[2]: 0x29


data[3]: 0x29


data[2556]: 0xC5


data[2557]: 0x15


data[2558]: 0x5C


data[2559]: 0x35
simple_time: 78526.672 us
hwCalculatedCRC_time: 129733.992 us
swCalculatedCRC_time: 832872.313 us
speedup: 641.98%
Checksum: 0xF25B8806
```

hwCalculatedCRC: 0xD9A716C5

swCalculatedCRC: 0xD9A716C5

Hardware was the faster method as it was 642% faster than software when testing it for this lab part. The reason that the hardware method is faster is because the accelerators handle specific tasks and are optimized to do that one task. This accelerator has custom logic in order perform this task. Also the accelerators have no program to run, they just run the custom logic instead of a program.

Exercise 1.3 Simulate Data Corruption:

Important code:

```
 //Original Data
  checksum = compute_simple_checksum(myData, 2560);

  MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
  for (ii = 0; ii < 4*2560; ii++)
       MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);

  hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

  printf("Checksum results with original data\n Simple_Checksum: 0x%08X
hwCalculatedCRC: 0x%08X\n\n",checksum, hwCalculatedCRC);

  //1 bit corrupted
  myData[8000] ^= (1 << 6);
  checksum = compute_simple_checksum(myData, 2560);

  MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
  for (ii = 0; ii < 4*2560; ii++)
       MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);
```

```
   hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

   printf("Checksum results with 1 bit corrupted\n Simple_Checksum: 0x%08X hwCalculatedCRC: 0x%08X\n\n",checksum, hwCalculatedCRC);

   //2 bits corrupted
   myData[9016] ^= (1 << 6);
   checksum = compute_simple_checksum(myData, 2560);

   MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
   for (ii = 0; ii < 4*2560; ii++)
       MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);

   hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

   printf("Checksum results with 2 bits corrupted\n Simple_Checksum: 0x%08X hwCalculatedCRC: 0x%08X\n\n",checksum, hwCalculatedCRC);
```

Console Output:

[CORTEX_M4_0] Checksum results with original data

 Simple_Checksum: 0xF25B8806 hwCalculatedCRC: 0xD9A716C5

Checksum results with 1 bit corrupted

 Simple_Checksum: 0xF25B87C6 hwCalculatedCRC: 0xC35CCC14

Checksum results with 2 bits corrupted

 Simple_Checksum: 0xF25B8786 hwCalculatedCRC: 0xA704C74E

What can be observed is that after 1 bit corruption the simplechecksum changes very little while CRC has dramatic change after 1 bit. After 2 bit corruption simple checksum is still similar to the value before, while CRC does not look anything similar to the original checksum.

For authentication I think CRC is the superior method because the checksum can detect 1 bit change and have a dramatic change. I guess the odds of the corrupted data checksum matching with the actual checksum is greatly reduced with CRC.

Exercise 2.1 Control

1. Compared to the polling method using bit-banding (Exercise 1.3 in Lab 2) to control the LED in the previous lab, does the interrupt method used in this exercise provide a quicker response to the push button actions? Justify your answer.

I think the interrupt method would be the faster of the two, because the button would trigger and interrupt as soon as the button is pressed and would cause the program to go to the ISR. This would be faster because it would alert that it was checked then checking to see if it was pressed continuously. An analogy for this would be checking your phone every second to see if you got a call versus your ears hearing the phone ring alerting you that you are being called. Interrupt method is also a way more power efficient implementation.

https://youtube.com/shorts/hl649TUkwCU?feature=share

Exercise 2.2 Measurement

Debug outputs:

[CORTEX_M4_0] 3000000

Button pressed for 185 ms, read_index = 2

Button pressed for 132 ms, read_index = 4

Button pressed for 172 ms, read_index = 6

Button pressed for 34 ms, read_index = 8

Button pressed for 1489 ms, read_index = 0

Button pressed for 1004 ms, read_index = 2

Button pressed for 1144 ms, read_index = 4

Button pressed for 801 ms, read_index = 6

Button pressed for 1641 ms, read_index = 8

Button pressed for 6202 ms, read_index = 0

Button pressed for 155 ms, read_index = 2

Button pressed for 157 ms, read_index = 4

Button pressed for 111 ms, read_index = 6

Button pressed for 112 ms, read_index = 8

Button pressed for 49 ms, read_index = 0

2. The array queue holds a total of 10 records. Is the queue long enough for this application? Justify your answer.

The queue appears to be long enough for this application because I pressed the button way more than 10 times and the behavior did not change nor performance.

Exercise 3.1 Speedup

Debug output:

[CORTEX_M4_0] Hardware CRC Time: 13008.334 us

Hardware CRC32: 0xEFB5AF2E

DMA CRC Time: 2812.333 us

DMA CRC32: 0xEFB5AF2E

Speedup: 4.63

Exercise 3.2 Different data block sizes

Ran into some bugs at this point I could not resolve. File failed to load and would not show me the errors for the file, which made it quite difficult to debug the source of the issue.

console output:

CORTEX_M4_0: GEL: Encountered a problem loading file: C:\Users\jmv19\workspace_v12\Lab3-Exercise-3.2-Different-Data-Blocks-Sizes\Debug\Lab3-Exercise-3.2-Different-Data-Blocks-Sizes.out Could not open file

Appendix:

Exercise 1.1 Simple Checksum:

uint32_t **compute_simple_checksum**(uint8_t* data, uint32_t length){

```
    uint32_t i;
    uint32_t checksum = 0;
    for(i = 0; i<length*4; i++)
    {
        checksum += (uint32_t)data[i] << ((i%4)*8);
    }
    checksum= ~checksum;
    return checksum;
}


int main(void)
{



    /* Stop Watchdog  */
    MAP_WDT_A_holdTimer();

    static uint8_t myData[10240];
    uint32_t checksum;
    checksum = compute_simple_checksum(myData, 2560);

    printf("Checksum: 0x%08X\n", checksum);

return 0;

}
```

Exercise 1.2 Speed Up:

```c
/* DriverLib Includes */
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>

/* Standard Includes */
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>



#define CRC32_POLY              0xEDB88320
#define CRC32_INIT              0xFFFFFFFF

static uint32_t calculateCRC32(uint8_t* data, uint32_t length);

volatile uint32_t hwCalculatedCRC, swCalculatedCRC;

uint32_t compute_simple_checksum(uint8_t* data, uint32_t length){
    uint32_t i;
    uint32_t checksum = 0;
    for(i = 0; i<length*4; i++)
    {
        checksum += (uint32_t)data[i] << ((i%4)*8);
    }
    checksum= ~checksum;
    return checksum;
}
```

```
int main(void)
{
    static uint8_t myData[10240];
    MAP_Timer32_initModule(TIMER32_0_BASE, TIMER32_PRESCALER_1,
TIMER32_32BIT,
    TIMER32_FREE_RUN_MODE);
    MAP_Timer32_startTimer(TIMER32_0_BASE, 0);
    printf("%u\n", MAP_CS_getMCLK());
    uint32_t mclk_freq = MAP_CS_getMCLK();
    uint32_t ii;
    //Code to display the first value and the last value of the dat file
    for(ii = 0; ii<4; ii++)
    {
        printf("\ndata[%i]: 0x%02X\n", ii, myData[ii]);
    }
    for(ii = 2556; ii<2560; ii++)
      {
          printf("\ndata[%i]: 0x%02X\n", ii, myData[ii]);
      }
    /* Variable to keep track of timing */
    uint32_t t0 = 0, t1 = 0;
    uint32_t press_duration_cycles = 0;
    float simple_time = 0;
    float hwCalculatedCRC_time = 0;
    float swCalculatedCRC_time = 0;

    /* Stop Watchdog  */
    MAP_WDT_A_holdTimer();
```

```
uint32_t checksum;
t1 = MAP_Timer32_getValue(TIMER32_0_BASE);
checksum = compute_simple_checksum(myData, 2560);
t0 = MAP_Timer32_getValue(TIMER32_0_BASE);

//time calculation for simple checksum
press_duration_cycles = t1 - t0;
simple_time = (press_duration_cycles / (float)mclk_freq) * 1000000;
printf("simple_time: %.3f us\n", simple_time);

MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
t1 = MAP_Timer32_getValue(TIMER32_0_BASE);
for (ii = 0; ii < 4*2560; ii++)
    MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);

/* Getting the result from the hardware module */
hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;
t0 = MAP_Timer32_getValue(TIMER32_0_BASE);
//time calculation for hwCalculatedCRC_time
press_duration_cycles = t1 - t0;
hwCalculatedCRC_time = (press_duration_cycles / (float)mclk_freq) * 1000000;
printf("hwCalculatedCRC_time: %.3f us\n", hwCalculatedCRC_time);
/* Calculating the CRC32 checksum through software */

t1 = MAP_Timer32_getValue(TIMER32_0_BASE);
swCalculatedCRC = calculateCRC32((uint8_t*) myData, 4*2560);
t0 = MAP_Timer32_getValue(TIMER32_0_BASE);
```

```
    //time calculation for hwCalculatedCRC_time
    press_duration_cycles = t1 - t0;
    swCalculatedCRC_time = (press_duration_cycles / (float)mclk_freq) * 1000000;
    printf("swCalculatedCRC_time: %.3f us\n", swCalculatedCRC_time);

    //calculates speed up percentage
    float speedup = 0;
    speedup = swCalculatedCRC_time / hwCalculatedCRC_time;
    printf("speedup: %.2f%%\n", speedup*100);
    printf("Checksum: 0x%08X\n", checksum);

    printf("hwCalculatedCRC: 0x%08X\n", hwCalculatedCRC);
    printf("swCalculatedCRC: 0x%08X\n", swCalculatedCRC);
  return 0;

}
//![Simple CRC32 Example]

/* Standard software calculation of CRC32 */
static uint32_t calculateCRC32(uint8_t* data, uint32_t length)
{
    uint32_t ii, jj, byte, crc, mask;;

    crc = 0xFFFFFFFF;

    for(ii=0;ii<length;ii++)
    {
        byte = data[ii];
```

```
        crc = crc ^ byte;

        for (jj = 0; jj < 8; jj++)
        {
            mask = -(crc & 1);
            crc = (crc >> 1) ^ (CRC32_POLY & mask);
        }

    }

    return ~crc;
}
```

Exercise 1.3 Simulate Data Corruption:

```
/* DriverLib Includes */
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>

/* Standard Includes */
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>

#define CRC32_POLY              0xEDB88320
#define CRC32_INIT              0xFFFFFFFF

volatile uint32_t hwCalculatedCRC;

uint32_t compute_simple_checksum(uint8_t* data, uint32_t length){
    uint32_t i;
```

```
    uint32_t checksum = 0;
    for(i = 0; i<length*4; i++)
    {
        checksum += (uint32_t)data[i] << ((i%4)*8);
    }
    checksum= ~checksum;
    return checksum;
}


int main(void)
{
    static uint8_t myData[10240];
    uint32_t ii;

    /* Stop Watchdog  */
    MAP_WDT_A_holdTimer();



    uint32_t checksum;

    //Original Data
    checksum = compute_simple_checksum(myData, 2560);

    MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
    for (ii = 0; ii < 4*2560; ii++)
        MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);
```

```
    hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

    printf("Checksum results with original data\n Simple_Checksum: 0x%08X
hwCalculatedCRC: 0x%08X\n\n",checksum, hwCalculatedCRC);

    //1 bit corrupted
    myData[8000] ^= (1 << 6);
    checksum = compute_simple_checksum(myData, 2560);

    MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
    for (ii = 0; ii < 4*2560; ii++)
        MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);

    hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

    printf("Checksum results with 1 bit corrupted\n Simple_Checksum: 0x%08X
hwCalculatedCRC: 0x%08X\n\n",checksum, hwCalculatedCRC);

    //2 bits corrupted
    myData[9016] ^= (1 << 6);
    checksum = compute_simple_checksum(myData, 2560);

    MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
    for (ii = 0; ii < 4*2560; ii++)
        MAP_CRC32_set8BitData(myData[ii], CRC32_MODE);

    hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;
```

```
    printf("Checksum results with 2 bits corrupted\n Simple_Checksum: 0x%08X
hwCalculatedCRC: 0x%08X\n\n",checksum, hwCalculatedCRC);


return 0;


}
```

Exercise 2.1 Control:

```
/* DriverLib Includes */
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>


/* Standard Includes */
#include <stdint.h>
#include <stdbool.h>


int main(void)
{
    /* Halting the Watchdog */
    MAP_WDT_A_holdTimer();


    //Configuring GreenLed as output and setting it to high as that is default for this exercise
    MAP_GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN1);
    MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);



    //Configuring S1 as an input and enabling interrupts
    MAP_GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);


    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
```

```
    MAP_GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1);
    MAP_GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1,
GPIO_HIGH_TO_LOW_TRANSITION);

    MAP_Interrupt_enableInterrupt(INT_PORT1);

    /* Enabling MASTER interrupts */
    MAP_Interrupt_enableMaster();

    while (1)
    {

    }
}

/* GPIO ISR */
void PORT1_IRQHandler(void)
{
    uint32_t status;

    status = MAP_GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);
    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, status);


    if (status)
    {
        //Check the current state of S1 to detect press or release
        if (MAP_GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1) ==
GPIO_INPUT_PIN_LOW)
```

```
        {
          //Button pressed: turn off the LED and set edge for release
          MAP_GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);
          MAP_GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1,
GPIO_LOW_TO_HIGH_TRANSITION);
        }
        else
        {
          //Button released: turn on the LED and set edge for press
          MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);
          MAP_GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1,
GPIO_HIGH_TO_LOW_TRANSITION);
        }
    }



}
```

Exercise 2.2 Measurement:

```
/* DriverLib Includes */
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>

/* Standard Includes */
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>

#define QUEUE_SIZE 10
struct record {
uint32_t timestamp;
```

```
uint8_t state;
};
struct record queue[QUEUE_SIZE];
volatile int read_index;
volatile int write_index;

int main(void)
{
    /* Halting the Watchdog */
    MAP_WDT_A_holdTimer();



    //timer code
    MAP_Timer32_initModule(TIMER32_0_BASE, TIMER32_PRESCALER_1,
TIMER32_32BIT,
    TIMER32_FREE_RUN_MODE);
    MAP_Timer32_startTimer(TIMER32_0_BASE, 0);
    printf("%u\n", MAP_CS_getMCLK());
    uint32_t mclk_freq = MAP_CS_getMCLK();

    //Configuring GreenLed as output and setting it to high as that is default for this exercise
    MAP_GPIO_setAsOutputPin(GPIO_PORT_P2, GPIO_PIN1);
    MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);



    //Configuring S1 as an input and enabling interrupts
    MAP_GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P1, GPIO_PIN1);

    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, GPIO_PIN1);
```

```
    MAP_GPIO_enableInterrupt(GPIO_PORT_P1, GPIO_PIN1);
    MAP_GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1,
GPIO_HIGH_TO_LOW_TRANSITION);

    MAP_Interrupt_enableInterrupt(INT_PORT1);

    /* Enabling MASTER interrupts */
    MAP_Interrupt_enableMaster();

    uint32_t last_press_time = 0;
    read_index = 0;
    write_index = 0;;
    while (1)
    {
        //Add a dummy delay to simulate background tasks
        __delay_cycles(200 * 3000);

        //Check if there are new records in the queue
        while (read_index != write_index)
        {
            struct record current_record = queue[read_index];
            read_index = (read_index + 1) % QUEUE_SIZE;

            if (current_record.state == 0)
            {
                //Button was released, calculate the duration
                uint32_t duration = last_press_time - current_record.timestamp;
                uint32_t duration_ms = duration / (MAP_CS_getMCLK() / 1000);
                printf("Button pressed for %u ms, read_index = %d\n", duration_ms, read_index);
```

```
            }
            else
            {
                //Button was pressed, store the time
                last_press_time = current_record.timestamp;
            }
        }
    }
}

/* GPIO ISR */
void PORT1_IRQHandler(void)
{
    uint32_t status;

    status = MAP_GPIO_getEnabledInterruptStatus(GPIO_PORT_P1);
    MAP_GPIO_clearInterruptFlag(GPIO_PORT_P1, status);

    if (status)
    {
        //Get the current timestamp
        uint32_t current_time = MAP_Timer32_getValue(TIMER32_0_BASE);

        //Check if the queue is full
        int next_write_index = (write_index + 1) % QUEUE_SIZE;
        if (next_write_index == read_index)
        {
            printf("Queue is full! Dropping record.\n");
```

```
            return;
        }

        //Get button state and add record before incrementing write_index
        uint8_t button_state = (MAP_GPIO_getInputPinValue(GPIO_PORT_P1, GPIO_PIN1) ==
GPIO_INPUT_PIN_LOW);
        queue[write_index].timestamp = current_time;
        queue[write_index].state = button_state;



        if (button_state == 1)
        {
            //Button pressed: turn off the green LED and set edge for release
            MAP_GPIO_setOutputLowOnPin(GPIO_PORT_P2, GPIO_PIN1);
            MAP_GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1,
GPIO_LOW_TO_HIGH_TRANSITION);
        }
        else
        {
            //Button released: turn on the green LED and set edge for press
            MAP_GPIO_setOutputHighOnPin(GPIO_PORT_P2, GPIO_PIN1);
            MAP_GPIO_interruptEdgeSelect(GPIO_PORT_P1, GPIO_PIN1,
GPIO_HIGH_TO_LOW_TRANSITION);
        }

        //Now increment write_index
        write_index = next_write_index;
    }
}
```

Exercise 3.1 Speedup:

```
/* DriverLib Includes */
#include <ti/devices/msp432p4xx/driverlib/driverlib.h>

/* Standard Includes */
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

/* Statics */
static volatile uint32_t crcSignature;

/* DMA Control Table */
#if defined(__TI_COMPILER_VERSION__)
#pragma DATA_ALIGN(controlTable, 1024)
#elif defined(__IAR_SYSTEMS_ICC__)
#pragma data_alignment=1024
#elif defined(__GNUC__)
__attribute__ ((aligned (1024)))
#elif defined(__CC_ARM)
__align(1024)
#endif
uint8_t controlTable[1024];



#define CRC32_SEED          0xFFFFFFFF
```

```c
uint8_t data_array[1024];

#define CRC32_POLY            0xEDB88320
#define CRC32_INIT            0xFFFFFFFF

volatile uint32_t hwCalculatedCRC;
volatile int dma_done = 0;

int main(void)
{
    /* Halting Watchdog */
    MAP_WDT_A_holdTimer();
    uint32_t ii;

    // Timer setup
    MAP_Timer32_initModule(TIMER32_0_BASE, TIMER32_PRESCALER_1,
TIMER32_32BIT, TIMER32_FREE_RUN_MODE);
    MAP_Timer32_startTimer(TIMER32_0_BASE, 0);
    uint32_t mclk_freq = MAP_CS_getMCLK();
    uint32_t t0, t1;
    uint32_t press_duration_cycles;
    float dmaTime, hwCalculatedCRC_time;

    //Hardware method
    t1 = MAP_Timer32_getValue(TIMER32_0_BASE);

    MAP_CRC32_setSeed(CRC32_INIT, CRC32_MODE);
    for (ii = 0; ii < sizeof(data_array); ii++) {
        MAP_CRC32_set8BitData(data_array[ii], CRC32_MODE);
```

```
    }

    hwCalculatedCRC = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

    t0 = MAP_Timer32_getValue(TIMER32_0_BASE);



    press_duration_cycles = t0 - t1;
    //Code to help with wrap around
    if (t0 > t1)
    {
      press_duration_cycles = t1 - t0;
    }
    else
    {
      press_duration_cycles = (UINT32_MAX - t0) + t1;
    }
    hwCalculatedCRC_time = (press_duration_cycles / (float)mclk_freq) * 1000000;
    printf("Hardware CRC Time: %.3f us\n", hwCalculatedCRC_time);
    printf("Hardware CRC32: 0x%08X\n", hwCalculatedCRC);

    //DMA method
    MAP_DMA_enableModule();
    MAP_DMA_setControlBase(controlTable);

    MAP_DMA_setChannelControl(UDMA_PRI_SELECT,
                              UDMA_SIZE_8 | UDMA_SRC_INC_8 | UDMA_DST_INC_NONE |
  UDMA_ARB_1024);
    MAP_DMA_setChannelTransfer(UDMA_PRI_SELECT,
```

```
                        UDMA_MODE_AUTO, data_array, (void*) (&CRC32->DI32),
sizeof(data_array));

    MAP_DMA_assignInterrupt(DMA_INT1, 0);
    MAP_Interrupt_enableInterrupt(INT_DMA_INT1);
    MAP_Interrupt_enableMaster();

    t1 = MAP_Timer32_getValue(TIMER32_0_BASE);

    dma_done = 0;
    MAP_CRC32_setSeed(CRC32_SEED, CRC32_MODE);
    MAP_DMA_enableChannel(0);
    MAP_DMA_requestSoftwareTransfer(0);

    while (dma_done == 0);

    t0 = MAP_Timer32_getValue(TIMER32_0_BASE);

    press_duration_cycles = t0 - t1;
    if (t0 > t1)
    {
        press_duration_cycles = t1 - t0;
    }
    else
    {
        press_duration_cycles = (UINT32_MAX - t0) + t1;
    }
    dmaTime = (press_duration_cycles / (float)mclk_freq) * 1000000;
    printf("DMA CRC Time: %.3f us\n", dmaTime);
```

```
    printf("DMA CRC32: 0x%08X\n", crcSignature);

    //Calculate speedup
    float speedup = hwCalculatedCRC_time / dmaTime;
    printf("Speedup: %.2f\n", speedup);

    while(1)
    {

    }
}

/* Completion interrupt for DMA */
void DMA_INT1_IRQHandler(void)
{
    // Disable the DMA Channel
    MAP_DMA_disableChannel(0);

    //Read the CRC result from the CRC-32 accelerator
    crcSignature = MAP_CRC32_getResultReversed(CRC32_MODE) ^ 0xFFFFFFFF;

    //Set the flag to indicate DMA is done
    dma_done = 1;
}
```

Exercise 3.2 Different data block sizes: