ECEN 5803

Mastering Embedded System Architecture

(Fall-2023)

PROJECT-1 : MODULE 5

DEBUG MONITOR

Date

10/16/2023

Guided By

Prof. Timothy Scherr

Submitted By

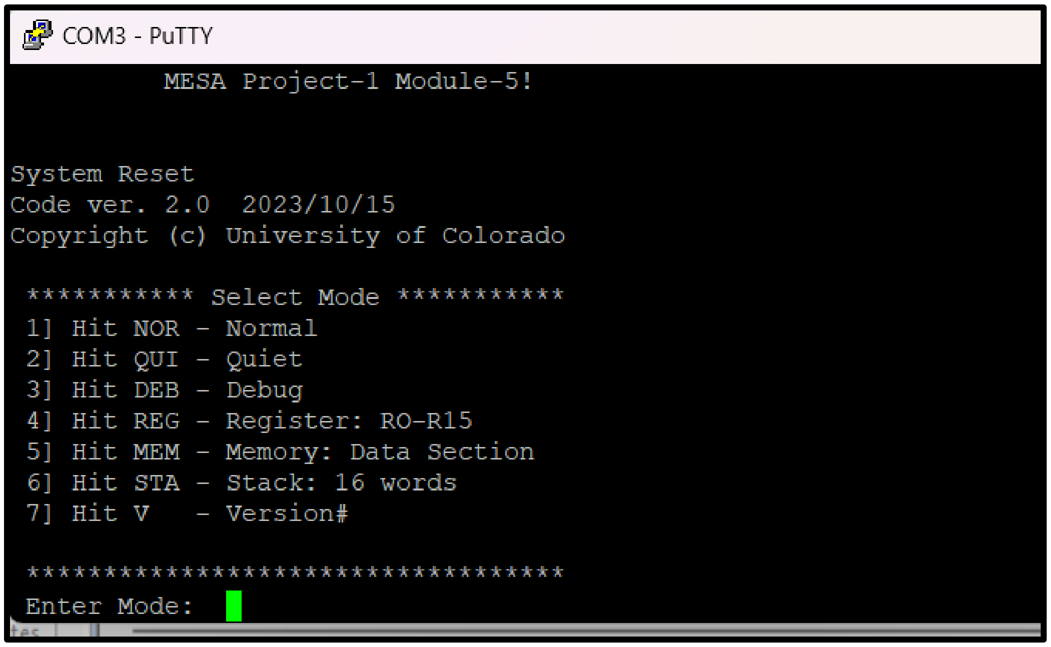
Ajay Kandagal

Kshitija Dhondage

Improvements you should make:

1. Improve the appearance of the debug monitor display and operation. Provide a personal touch and make screenshots of your results.

For a personal touch, we changed the welcome message, updated the date and enumeration of the commands, and added an outline. We also added three commands for access to the Register, Stack, and Memory sections.

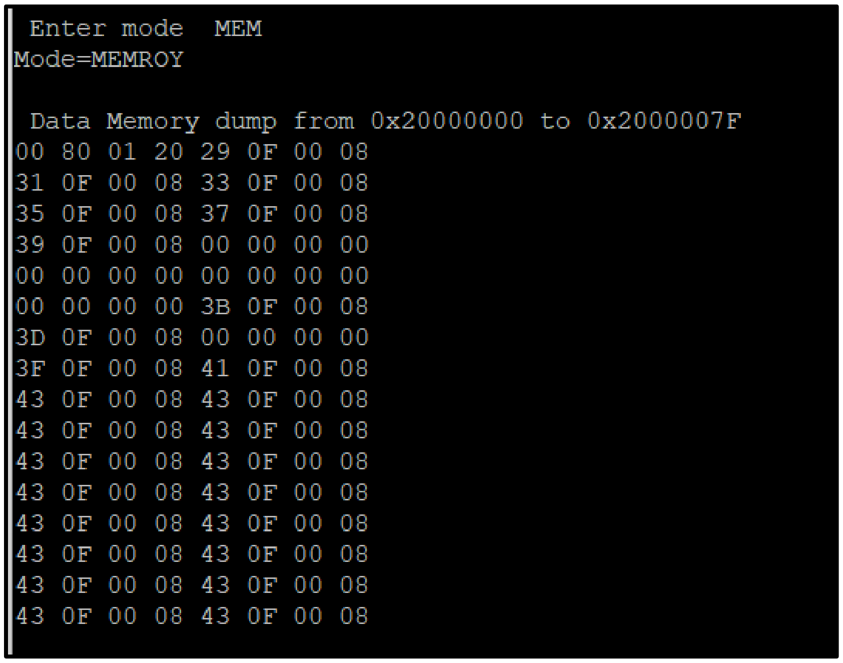


```
COM3 - PuTTY
          MESA Project-1 Module-5!


System Reset
Code ver. 2.0  2023/10/15
Copyright (c) University of Colorado

 *********** Select Mode ***********
 1] Hit NOR - Normal
 2] Hit QUI - Quiet
 3] Hit DEB - Debug
 4] Hit REG - Register: RO-R15
 5] Hit MEM - Memory: Data Section
 6] Hit STA - Stack: 16 words
 7] Hit V   - Version#

 ***********************************
 Enter Mode:
```

Figure 1: Debug monitor initial display and operation.

2. Add commands to the monitor, including display of the registers R0-R15, and display of section of memory.

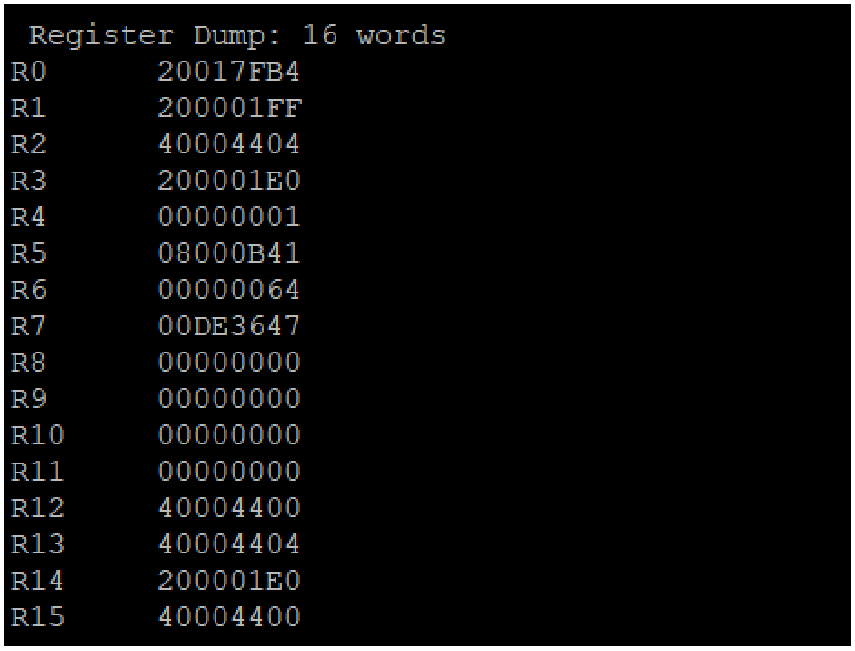
The data memory section can be accessed using the command MEM, which will display the 32-byes data from 0x20000000 to 0x2000007F.



```
 Enter mode  MEM
Mode=MEMROY

 Data Memory dump from 0x20000000 to 0x2000007F
00 80 01 20 29 0F 00 08
31 0F 00 08 33 0F 00 08
35 0F 00 08 37 0F 00 08
39 0F 00 08 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 3B 0F 00 08
3D 0F 00 08 00 00 00 00
3F 0F 00 08 41 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
```

Figure 2: Debug Monitor's section of memory view

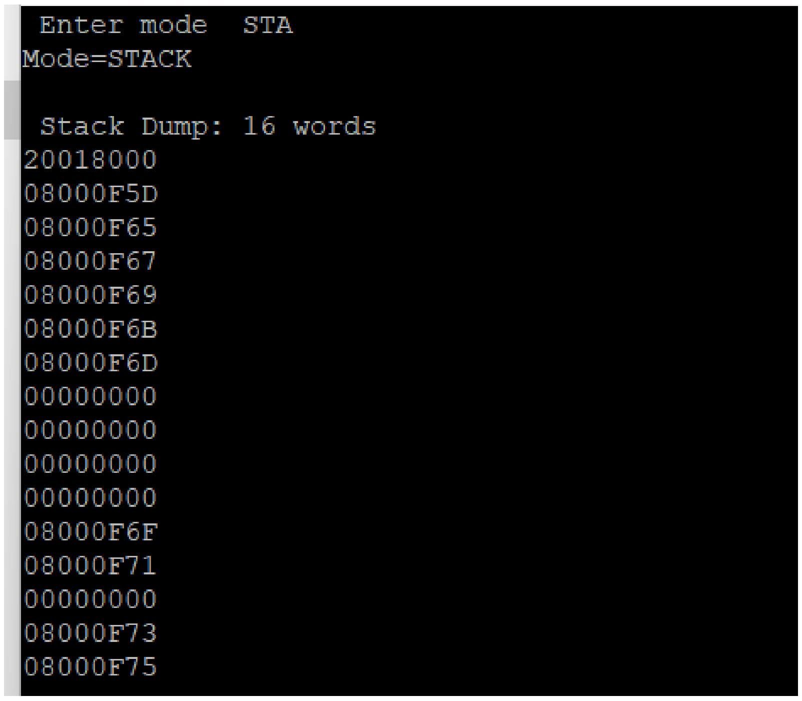
The register values can be accessed using the REG command. The following screenshot shows the register dump from R0 to R15.



```
 Register Dump: 16 words
R0       20017FB4
R1       200001FF
R2       40004404
R3       200001E0
R4       00000001
R5       08000B41
R6       00000064
R7       00DE3647
R8       00000000
R9       00000000
R10      00000000
R11      00000000
R12      40004400
R13      40004404
R14      200001E0
R15      40004400
```

Figure 3: Debug Monitor's register display

The 16 words from stack memory can be accessed using the command STA. Following is the screenshot of running the command.



```
 Enter mode  STA
Mode=STACK

 Stack Dump: 16 words
20018000
08000F5D
08000F65
08000F67
08000F69
08000F6B
08000F6D
00000000
00000000
00000000
00000000
08000F6F
08000F71
00000000
08000F73
08000F75
```

Figure 4: Debug Monitor's stack display

3. Insert code using flags in the timer0 function to blink the Red LED within a 1-second period.

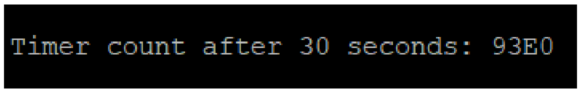
Coding is done.

Questions

Que 1. What is the count shown in timer0 if you let it run for 30 seconds? Explain why it is this.

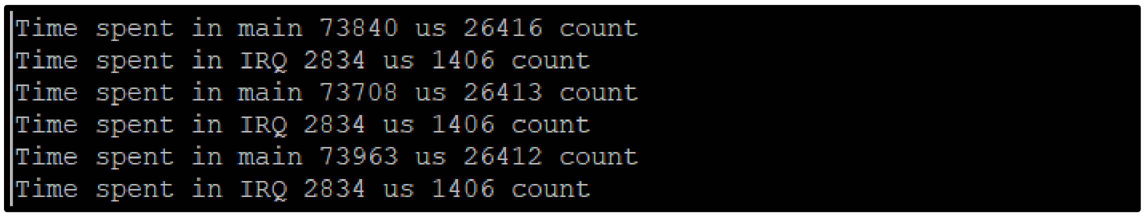
After running the timer0 function for 30 seconds, the count is 93E0 = 37856. Timer0 counter is a 16-bit unsigned value hence the maximum value is 65535. Each counter corresponds to 100 us. Timer0 will overflow four times within 30 seconds hence, on the 5th iteration, the counter will have a value of

(30 seconds in 100us ticks) – (100us ticks in 4 overflows of Timer0) = 300,000 – (65536 * 4) = 37856 = 0x93E0



```
Timer count after 30 seconds: 93E0
```

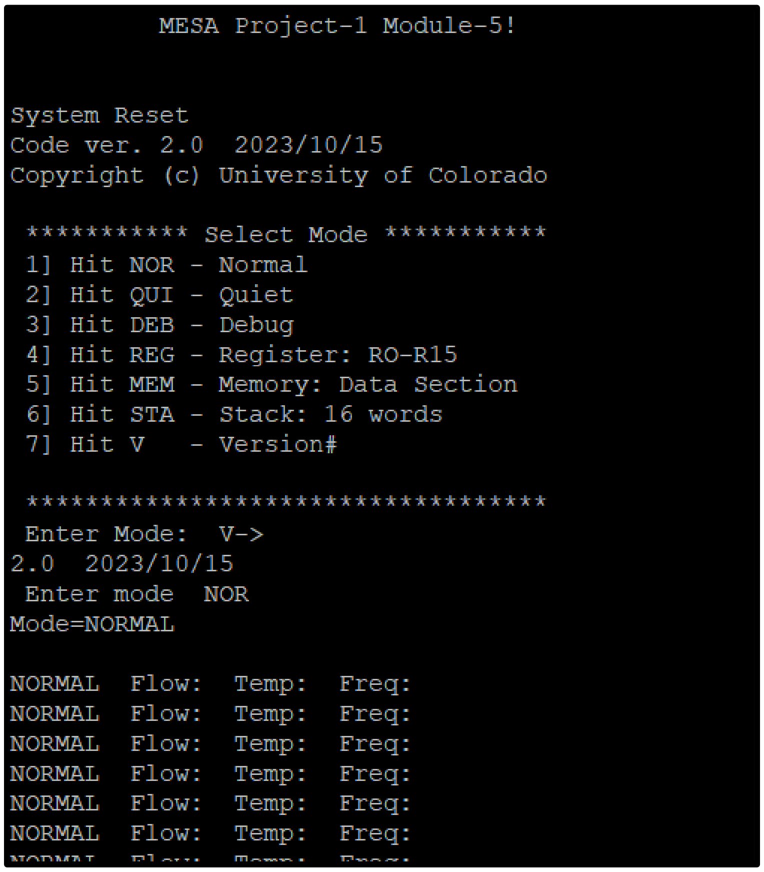
Que 2. How much time does the code spend in the main loop versus Interrupt Service Routines?



```
Time spent in main 73840 us 26416 count
Time spent in IRQ 2834 us 1406 count
Time spent in main 73708 us 26413 count
Time spent in IRQ 2834 us 1406 count
Time spent in main 73963 us 26412 count
Time spent in IRQ 2834 us 1406 count
```

We did a timing analysis for 100ms. In 100 ms, the main loop runs around 26415 times, and the total time observed is 73840 us. The ISR runs around 1406 times, and the total time is 2834 us. If we take the average time for each iteration, the **main runs for around 2.8 us (73840 us / 26415),** and the **IRQ takes 2 us (2834 us / 1406).**

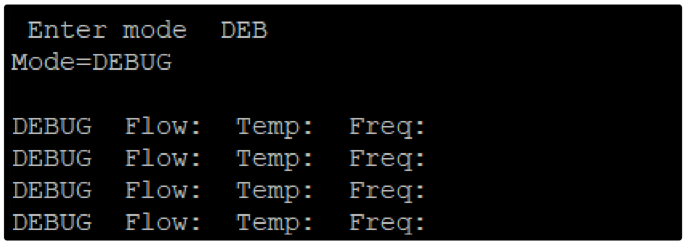
Que 3. Test each of the commands in the Debug Monitor and record the results. Explain anything you see that you did not expect. Are you able to display all the registers?



```
          MESA Project-1 Module-5!


System Reset
Code ver. 2.0  2023/10/15
Copyright (c) University of Colorado

 *********** Select Mode ***********
 1] Hit NOR - Normal
 2] Hit QUI - Quiet
 3] Hit DEB - Debug
 4] Hit REG - Register: RO-R15
 5] Hit MEM - Memory: Data Section
 6] Hit STA - Stack: 16 words
 7] Hit V   - Version#

 ***********************************
 Enter Mode:  V->
2.0  2023/10/15
 Enter mode  NOR
Mode=NORMAL

NORMAL  Flow:  Temp:  Freq:
NORMAL  Flow:  Temp:  Freq:
NORMAL  Flow:  Temp:  Freq:
NORMAL  Flow:  Temp:  Freq:
NORMAL  Flow:  Temp:  Freq:
NORMAL  Flow:  Temp:  Freq:
```

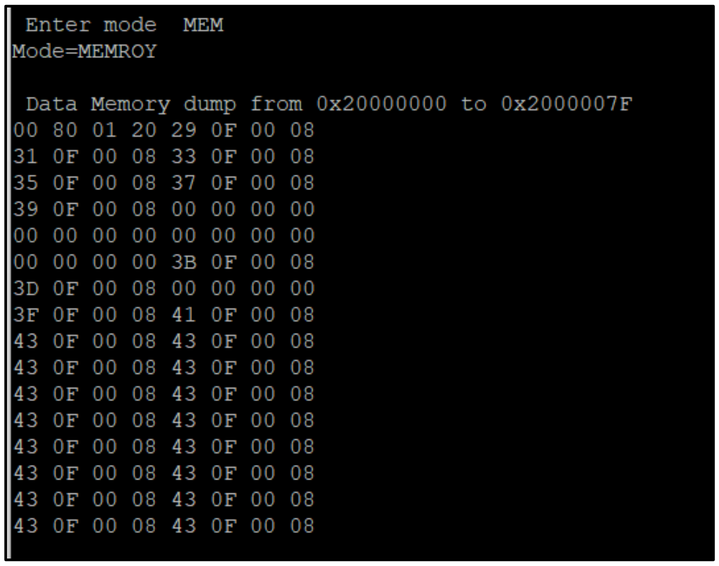
Figure 5: Debug Monitor displaying Version and then entering Normal mode.

When switched from Quiet mode to Normal mode, we observed random values printed on the display. This might be because the tx_buffer keeps filling in the circular buffer as a result, all the characters stored in the buffer previously get printed.



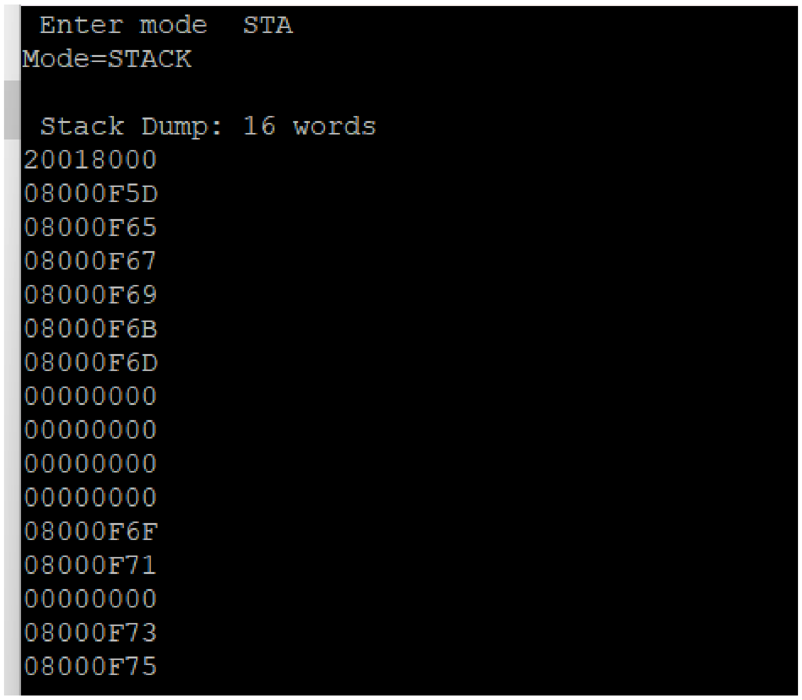
```
 Enter mode  DEB
Mode=DEBUG

DEBUG  Flow:  Temp:  Freq:
DEBUG  Flow:  Temp:  Freq:
DEBUG  Flow:  Temp:  Freq:
DEBUG  Flow:  Temp:  Freq:
```

Figure 6: Debug monitor's view for Debug mode



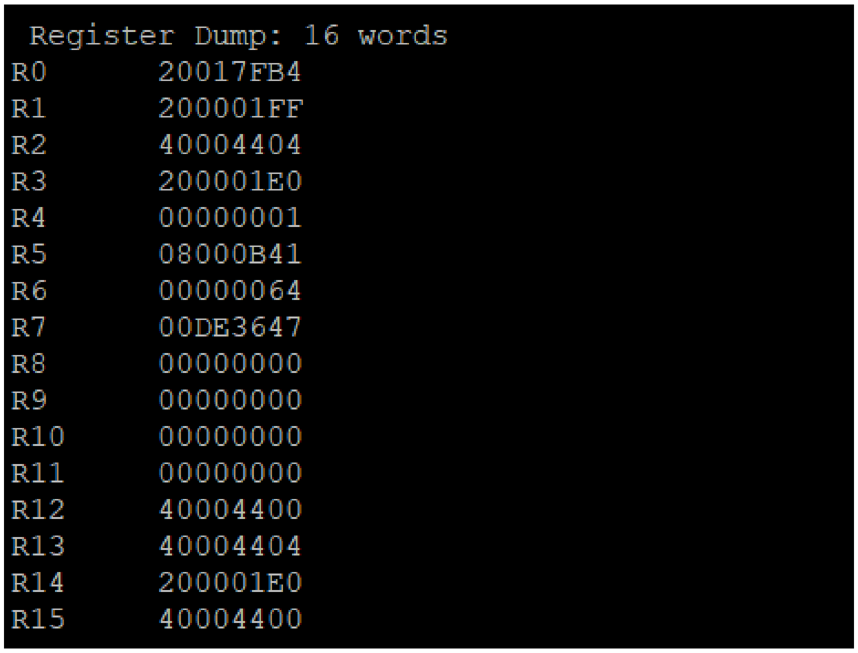
```
 Enter mode  MEM
Mode=MEMROY

 Data Memory dump from 0x20000000 to 0x2000007F
00 80 01 20 29 0F 00 08
31 0F 00 08 33 0F 00 08
35 0F 00 08 37 0F 00 08
39 0F 00 08 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 3B 0F 00 08
3D 0F 00 08 00 00 00 00
3F 0F 00 08 41 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
43 0F 00 08 43 0F 00 08
```

Figure 7: Debug Monitor's view for memory dump



```
 Enter mode  STA
Mode=STACK

 Stack Dump: 16 words
20018000
08000F5D
08000F65
08000F67
08000F69
08000F6B
08000F6D
00000000
00000000
00000000
00000000
08000F6F
08000F71
00000000
08000F73
08000F75
```

Figure 8: Debug monitor's view for Stack dump



```
 Register Dump: 16 words
R0       20017FB4
R1       200001FF
R2       40004404
R3       200001E0
R4       00000001
R5       08000B41
R6       00000064
R7       00DE3647
R8       00000000
R9       00000000
R10      00000000
R11      00000000
R12      40004400
R13      40004404
R14      200001E0
R15      40004400
```

Figure 9: Debug monitor's view for registers print

Que 4. What is the new command you added to the debug menu, and what does it do? Capture a screenshot of the new monitor window.

We implemented the following three new commands:

1. REG: To display Registers R0-R15. This uses assembly instruction store multiple to copy the contents from R0 to R12 to an array.

2. STA: To display the Stack memory. Again, we use assembly instruction __ASM("sp") to access the stack memory and print out the first 16 words.

3. MEM: To display the section of memory. We are displaying 32 words in hex format from 0x20000000 to 0x2000007F.

Que 5. A GPIO pin is driven high at the beginning of the Timer ISR, and low at the end. What purpose could this serve?

Driving a GPIO pin high at the start of a Timer ISR and low at completion is a technique for precise timing analysis and debugging. It ensures that the ISR triggers at the expected time and measures its execution time. To achieve this, we can use a logic analyzer. The analyzer employs markers to capture the timing data by detecting the transitions of the GPIO pin from high to low. This allows for an accurate assessment of the ISR execution time, aiding in performance optimization and troubleshooting, ensuring that the system operates as expected.

Que 6. Estimate the % of CPU cycles used for the main background process, assuming a 100-millisecond operating cycle.

From our previous analysis in Q2, we estimated the execution times of the main and ISR. The main runs for 73840 us. The CPU runs at 84MHz, so each cycle takes around 11.9 ns.

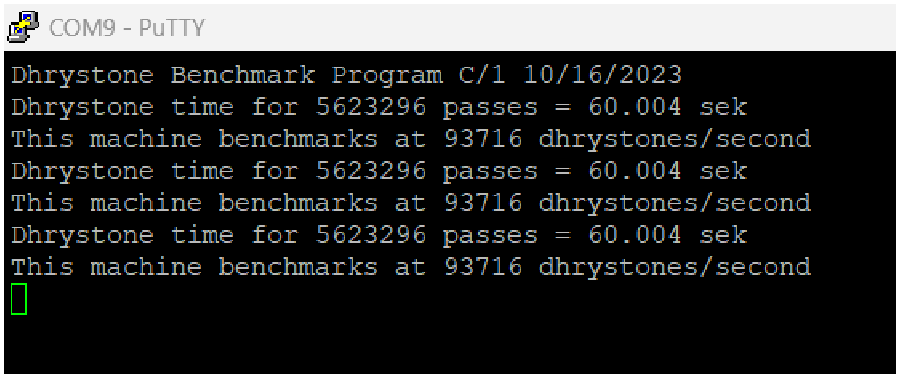
Number of CPU cycles used by Main in 100ms = 73840 us / 11.9 ns = 6.2 million cycles.

% CPU cycles used = 6.2 / 8.4 = 73.8 %

Que 7. What is your DMIPS estimate for the ST STM32F401RE MCU?

After running the mBed Dhrystone Example code, we get the following output. The Nucelo-STM32F401RE scores 93716 Dhrystones per second.

VAX DMIPS = 93716 / 1757 = **53.34**



```
Dhrystone Benchmark Program C/1 10/16/2023
Dhrystone time for 5623296 passes = 60.004 sek
This machine benchmarks at 93716 dhrystones/second
Dhrystone time for 5623296 passes = 60.004 sek
This machine benchmarks at 93716 dhrystones/second
Dhrystone time for 5623296 passes = 60.004 sek
This machine benchmarks at 93716 dhrystones/second
```

Figure 10: Dhrystone benchmarking on STM32F401RE

Appendix

```
/**----------------------------------------------------------------------------

   \file main.cpp

--                                                                           --
--      ECEN 5803 Mastering Embedded System Architecture                     --
--                   Project 1 Module 5                                      --
--                Microcontroller Firmware                                   --
--                      main.cpp                                             --
--                                                                           --
-------------------------------------------------------------------------------
--
--  Designed for:  University of Colorado at Boulder
--
--
--  Designed by:  Tim Scherr
--  Revised by:  Kshitija Dhondage, Ajay Kandagal
--
-- Version: 3.0
-- Date of current revision:  2023-10-16
-- Target Microcontroller: ST STM32F401RE
-- Tools used:  ARM mbed compiler
--              ARM mbed SDK
--              ST Nucleo STM32F401RE Board
--
--
-- Functional Description:  Main code file generated by mbed, and then
--                           modified to implement a super loop bare metal OS.
--
```

```
--      Copyright (c) 2015, 2016, 2022 Tim Scherr  All rights reserved.
--
*/

/* Header Files */
#define MAIN
#include "shared.h"
#undef MAIN

#include "NHD_0216HZ.h"
#include "DS1631.h"
#include "pindef.h"
#include "Timer.h"

/* Define LED I/O pins */
DigitalOut greenLED(LED1); // PA_5
DigitalOut redLED(PA_9);   // PA_9

extern volatile uint16_t SwTimerIsrCounter;
extern volatile uint8_t heartbeat_flag; // flag to create heartbeat with 500ms ON and OFF on Red Led
Ticker tick;                            //  Creates a timer interrupt using mbed methods
Serial pc(USBTX, USBRX);                // UART pins

#if EN_TIMING_BM
Timer timer;
uint32_t main_tstart, main_tdur, irq_tstart, irq_tdur, irq_count;
#endif
```

```
/**
 * @brief  This flip function , changes the on board Green LED state (ON/OFF)
 *
 * @param[in] Void
 *
 * @return    bool
 */
bool flip()
{
   greenLED = !greenLED;
   return greenLED;
}

/**
 * @brief Main function Entry point function
 *
 * @param[in] Void
 *
 * @return    int
 */
int main()
{
   pc.printf("\r\n          MESA Project-1 Module-5!    \n\n\r");
   uint32_t count = 0; // to count the loop

   /* Call the timer function timer0() every 100us */
   tick.attach_us(&timer0, 100);

   /* initialize serial buffer pointers */
   rx_in_ptr = rx_buf;  /* pointer to the receive in data */

   rx_out_ptr = rx_buf; /* pointer to the receive out data*/

   tx_in_ptr = tx_buf;  /* pointer to the transmit in data*/

   tx_out_ptr = tx_buf; /* pointer to the transmit out */


   /* send a starting message to the terminal  */

   UART_direct_msg_put("\r\nSystem Reset\r\nCode ver. ");

   UART_direct_msg_put(CODE_VERSION);

   UART_direct_msg_put("\r\n");

   UART_direct_msg_put(COPYRIGHT);

   UART_direct_msg_put("\r\n");


   set_display_mode();


#if EN_TIMING_BM

   timer.start();

#endif


   while (1) // Cyclical Executive Loop

   {

#if EN_TIMING_BM

      main_tstart = timer.read_us();

#endif

      count++; // counts the number of times through the loop

               // __enable_interrupts();

               //  __clear_watchdog_timer();


      serial();       // Polls the serial port

      chk_UART_msg(); // checks for a serial port message received
      monitor();      // Sends serial port output messages depending
                      //     on commands received and display mode


      /* SwTimerIsrCounter is initially set to 0 and it increments after each
100us.
         Check if the lower 13 bits of SwTimerIsrCounter have exceeded 4095
(0x0FFF),
         When this condition is met, toggle the Green LED to indicate the passage
of
         that time interval. For the first time Green LED will toggle when
approximately
         409.5 milliseconds have passed since the timer started at 0.
         After the initial toggle, the Green LED will continue to blink every
409.5
         milliseconds until the timer overflows and starts counting from 0 again.
This
         creates a periodic pattern of LED blinking based on the timer's count
and
         overflow.
      */
      if ((SwTimerIsrCounter & 0x1FFF) > 0x0FFF)
      {
         flip(); // Toggle Green LED
      }


      /* Blink the externel RED LED with approximately 1 sec period   */
      if (heartbeat_flag)
      {
         redLED = !redLED;
         heartbeat_flag = 0;
      }
```

```
#if EN_TIMING_BM
      main_tdur += timer.read_us() - main_tstart;

      if (SwTimerIsrCounter == 1000)
      {
         pc.printf("\n\rTime spent in main %d us %d count", main_tdur, count);
         pc.printf("\n\rTime spent in IRQ %d us %d count", irq_tdur, irq_count);

         main_tdur = 0;
         irq_tdur = 0;
         count = 0;
         irq_count = 0;
         SwTimerIsrCounter = 0;
      }
#endif
   } /// End while(1) loop
}

/**----------------------------------------------------------------------------
             \file Monitor.cpp
--                                                                           --
--              ECEN 5003 Mastering Embedded System Architecture             --
--                  Project 1 Module 4                                       --
--                Microcontroller Firmware                                   --
--                    Monitor.cpp                                            --
--                                                                           --
-------------------------------------------------------------------------------
--
--  Designed for:  University of Colorado at Boulder
```

```
--

--

--  Designed by:  Tim Scherr

--  Revised by:  Kshitija Dhondage, Ajay Kandagal

--

-- Version: 2.0

-- Date of current revision:  2023-10-16

-- Target Microcontroller: ST STM32F401RE

-- Tools used:  ARM mbed compiler

--              ARM mbed SDK

--              ST Nucleo STM32F401RE Board

--

--

   Functional Description: See below

--

--      Copyright (c) 2015, 2022 Tim Scherr All rights reserved.

--

*/

#include <stdio.h>

#include "shared.h"


/*******************************************************************************

 * Set Display Mode Function

 * Function determines the correct display mode.  The 3 display modes operate as

 *   follows:

 *

 *  NORMAL MODE       Outputs only mode and state information changes

 *                     and calculated outputs

 *
 *  QUIET MODE        No Outputs
 *
 *  DEBUG MODE        Outputs mode and state information, error counts,
 *                    register displays, sensor states, and calculated output
 *                  (currently not all features are operation, could be enhanced)
 *
 * There is deliberate delay in switching between modes to allow the RS-232 cable
 * to be plugged into the header without causing problems.
 ********************************************************************************/

void set_display_mode(void)
{
   UART_direct_msg_put("\r\n *********** Select Mode ***********");
   UART_direct_msg_put("\r\n 1] Hit NOR - Normal");
   UART_direct_msg_put("\r\n 2] Hit QUI - Quiet");
   UART_direct_msg_put("\r\n 3] Hit DEB - Debug");
   UART_direct_msg_put("\r\n 4] Hit REG - Register: RO-R15");
   UART_direct_msg_put("\r\n 5] Hit MEM - Memory: Data Section");
   UART_direct_msg_put("\r\n 6] Hit STA - Stack: 16 words");
   UART_direct_msg_put("\r\n 7] Hit V   - Version#\r\n");
   UART_direct_msg_put("\r\n ***********************************");
   UART_direct_msg_put("\r\n Enter Mode:  ");
}

__asm uint32_t address_ret(int x)
{
   MOVS R3, R0 // Copy the value of R0 that is address X to R3
   LDR R0, [R3]  // Load the value at the address pointed to by R3 into R0
   BX LR // Return to the calling function
```

```c
}

//*****************************************************************************/
/// \fn void chk_UART_msg(void)
///
///  \brief - fills a message buffer until return is encountered, then calls
///           message processing
//*****************************************************************************/
/****************      ECEN 5803 add code as indicated   **********************/
// Improve behavior of this function
void chk_UART_msg(void)
{
   uchar8_t j;
   while (UART_input()) // becomes true only when a byte has been received
   {                    // skip if no characters pending
      j = UART_get();   // get next character

      if (j == '\r') // on a enter (return) key press
      {              // complete message (all messages end in carriage return)
         UART_msg_put("->");
         UART_msg_process();
      }
      else
      {
         if ((j != 0x02)) // if not ^B
         {                // if not command, then
            UART_put(j);  // echo the character
         }
         else
```

```
{
   ;
}
if (j == '\b')
{ // backspace editor
   if (msg_buf_idx > 0)
   {                       // if not 1st character then destructive
      UART_msg_put(" \b"); // backspace
      msg_buf_idx--;
   }
}
else if (msg_buf_idx >= MSG_BUF_SIZE)
{ // check message length too large
   UART_msg_put("\r\nToo Long!");
   msg_buf_idx = 0;
}
else if ((display_mode == QUIET) && (msg_buf[0] != 0x02) &&
         (msg_buf[0] != 'D') && (msg_buf[0] != 'N') &&
         (msg_buf[0] != 'V') && (msg_buf[0] != 'M') &&
             (msg_buf[0] != 'R') && (msg_buf[0] != 'S') &&
         (msg_buf_idx != 0))
{                  // if first character is bad in Quiet mode
   msg_buf_idx = 0; // then start over
}
else
{ // not complete message, store character

   msg_buf[msg_buf_idx] = j;
   msg_buf_idx++;
```

```
            if (msg_buf_idx > 2)
            {
               UART_msg_process();
            }
         }
      }
   }
}

//*****************************************************************************/
///  \fn void UART_msg_process(void)
/// UART Input Message Processing
//*****************************************************************************/
void UART_msg_process(void)
{
   uchar8_t chr, err = 0;
   //   unsigned char  data;

   if ((chr = msg_buf[0]) <= 0x60)
   { // Upper Case
      switch (chr)
      {
      case 'D':
         if ((msg_buf[1] == 'E') && (msg_buf[2] == 'B') && (msg_buf_idx == 3))
         {
            display_mode = DEBUG;
            UART_msg_put("\r\nMode=DEBUG\n");
            display_timer = 0;
         }
```

```
        else
            err = 1;
        break;

    case 'N':
        if ((msg_buf[1] == 'O') && (msg_buf[2] == 'R') && (msg_buf_idx == 3))
        {
            display_mode = NORMAL;
            UART_msg_put("\r\nMode=NORMAL\n");
            // display_timer = 0;
        }
        else
            err = 1;
        break;

    case 'Q':
        if ((msg_buf[1] == 'U') && (msg_buf[2] == 'I') && (msg_buf_idx == 3))
        {
            display_mode = QUIET;
            UART_msg_put("\r\nMode=QUIET\n");
            display_timer = 0;
        }
        else
            err = 1;
        break;

    case 'M':
        if ((msg_buf[1] == 'E') && (msg_buf[2] == 'M') && (msg_buf_idx == 3))
        {
            display_mode = MEMROY;
            UART_msg_put("\r\nMode=MEMROY\n");
            // display_timer = 0;
        }
        else
            err = 1;
        break;

    case 'S':
        if ((msg_buf[1] == 'T') && (msg_buf[2] == 'A') && (msg_buf_idx == 3))
        {
            display_mode = STACK;
            UART_msg_put("\r\nMode=STACK\n");
            // display_timer = 0;
        }
        else
            err = 1;
        break;

    case 'R':
        if ((msg_buf[1] == 'E') && (msg_buf[2] == 'G') && (msg_buf_idx == 3))
        {
            display_mode = REGISTER;
            UART_msg_put("\r\nMode=REGISTER\n");
            // display_timer = 0;
        }
        else
            err = 1;
        break;
```

```
   case 'V':
          if (msg_buf_idx == 1)
      {
              display_mode = VERSION;
             UART_msg_put("\r\n");
         UART_msg_put(CODE_VERSION);
         UART_msg_put("\r\n Enter mode  ");
         display_timer = 0;
          }
          else
             err = 1;
          break;

   default:
      err = 1;
   }
}

else
{ // Lower Case
   switch (chr)
   {
   default:
      err = 1;
   }
}

if (err == 1)
```

```c
   {
      UART_msg_put("\n\rEntry Error!");
   }
   else if (err == 2)
   {
      UART_msg_put("\n\rNot in DEBUG Mode!");
   }
   else
   {
      msg_buf_idx = 0; // put index to start of buffer for next message
      ;
   }
   msg_buf_idx = 0; // put index to start of buffer for next message
}

//****************************************************************************
///   \fn   is_hex
/// Function takes
///  @param a single ASCII character and returns
///  @return 1 if hex digit, 0 otherwise.
///
//****************************************************************************
uchar8_t is_hex(uchar8_t c)
{
   if ((((c |= 0x20) >= '0') && (c <= '9')) || ((c >= 'a') && (c <= 'f')))
      return 1;
   return 0;
}
```

```
__asm void mov_regs(uint32_t *all_reg)
{
   STM r0, {r0-r12}
   BX lr
}

/*******************************************************************************
 *   \fn  DEBUG and DIAGNOSTIC Mode UART Operation
 *******************************************************************************/
void monitor(void)
{

   /**********************************/
   /*     Spew outputs               */
   /**********************************/

   switch (display_mode)
   {
   case (QUIET):
   {
      // UART_msg_put("\r\n ");
      display_flag = 0;
   }
   break;
   case (VERSION):
   {
      display_flag = 0;
   }
   break;
```

```
case (NORMAL):
{
   if (display_flag == 1)
   {
      UART_msg_put("\r\nNORMAL ");
      UART_msg_put(" Flow: ");
      // ECEN 5803 add code as indicated
      //  add flow data output here, use UART_hex_put or similar for
      // numbers
      UART_msg_put(" Temp: ");
      //  add flow data output here, use UART_hex_put or similar for
      // numbers
      UART_msg_put(" Freq: ");
      //  add flow data output here, use UART_hex_put or similar for
      // numbers
      display_flag = 0;
   }
}
break;
case (DEBUG):
{
   if (display_flag == 1)
   {
      UART_msg_put("\r\nDEBUG ");
      UART_msg_put(" Flow: ");
      // ECEN 5803 add code as indicated
      //  add flow data output here, use UART_hex_put or similar for
      // numbers
      UART_msg_put(" Temp: ");
```

```
        //  add flow data output here, use UART_hex_put or similar for

        // numbers

        UART_msg_put(" Freq: ");


        // clear flag to ISR

        display_flag = 0;

     }

  }

  break;


  case (MEMROY):

  {

     if (display_flag == 1)

     {

        uint32_t mem_buff[32];

        int start_addr = 536870912;

        UART_direct_msg_put("\n\r Data Memory dump from 0x20000000 to
0x2000007F\r\n");


        for (uint8_t i = 0; i < 32; i++)

        {

           mem_buff[i] = address_ret(start_addr);


           printf("%02X %02X %02X %02X ", (mem_buff[i] & 0xFF),

                  (mem_buff[i] >> 8) & 0xFF,

                  (mem_buff[i] >> 16) & 0xFF,

                  (mem_buff[i] >> 24) & 0xFF);


           start_addr += 4; // Move to the next 4-byte memory address.
```

```
         if (i % 2 != 0)
         {
            UART_direct_msg_put("\n\r");
         }
      }

      display_flag = 0;
   }
}
break;

case (STACK):
{
   if (display_flag == 1)
   {
      UART_direct_msg_put("\n\r Stack Dump: 16 words \r\n");

          register uint32_t stack_val __asm ("sp");
      uint32_t *stack_addr = (uint32_t *)stack_val;

      for (int i = 0; i < 16; i++)
      {
         UART_direct_hex_put((*stack_addr & 0xFF000000) >> 24);
         UART_direct_hex_put((*stack_addr & 0x00FF0000) >> 16);
         UART_direct_hex_put((*stack_addr & 0x0000ff00) >> 8);
         UART_direct_hex_put(*stack_addr & 0x000000FF);
         UART_direct_msg_put("\n\r");
         stack_addr++;
      }

      UART_direct_msg_put("\n\r");
      display_flag = 0;
   }
}
break;

case (REGISTER):
    if (display_flag == 1)
    {
       uint32_t all_regs[16];
       mov_regs(all_regs);

       UART_direct_msg_put("\n\r Register Dump: 16 words \r\n");

       for (int i = 0; i < 16; i++)
       {
            UART_direct_put('R');
           if ( i < 10) {
               UART_direct_put(i + '0');
            }
            else {
               UART_direct_put('1');
               UART_direct_put((i % 10) + '0');
            }
            UART_direct_msg_put("\t");
            UART_direct_hex_put((all_regs[i] & 0xFF000000) >> 24);
```

```
               UART_direct_hex_put((all_regs[i] & 0x00FF0000) >> 16);

               UART_direct_hex_put((all_regs[i] & 0x0000ff00) >> 8);

               UART_direct_hex_put(all_regs[i] & 0x000000FF);

               UART_direct_msg_put("\n\r");

          }


          display_flag = 0;

       }

   break;


   default:

   {

      UART_msg_put("Mode Error");

   }

   }

}
```