EECS 690
Project 1 Report

Program Profiling on a TI Tiva C TI_TM4C1294NCPDT using FreeRTOS

Written By
Ben Sokol
Kaiser Mittenburg

The University of Kansas
Electrical Engineering and Computer Science

September 18, 2018

1 Abstract

Modern programs are very complex and often run many tasks. For the developer, it can be difficult to determine how many resources will be used to execute any given block of code. This report discusses a simple program tracing technique to determine where the program is spending its time. Knowing which bits of memory are being used the most will outline a good place to start optimizing and will lead to more efficient programs. The example is executed using a TI_TM4C1294NCPDT board running FreeRTOS. We found that our program with several small tasks spent its time in just a few areas of memory, which was expected.

2 Revision History

The following table (*Table 2-1*) lists the revision history for this document.

Table 2-1 Revision History

Date	Revision	Description
September 18, 2018	1.0	Initial Release

3 Table of Contents

1 Abstract **1**

2 Revision History **1**

3 Table of Contents **2**

4 List of Figures **2**

5 List of Tables **2**

6 Principles of Operation (POP) **3**

7 Data Structure Descriptions **4**

8 Function Descriptions **4**

9 Parameters **7**

The following table contains a list of parameters used in **7**

10 Testing **9**

11 Lessons Learned **9**

Appendix A – Program Source Code **10**

4 List of Figures

FIGURE 10-1 RESULTS HISTOGRAM 9

5 List of Tables

TABLE 2-1 REVISION HISTORY 1

TABLE 7-1 DATA STRUCTURES 4

TABLE 8-1 FUNCTIONS 4

TABLE 8-2 REFERENCED EXTERNAL FUNCTIONS 6

TABLE 9-1 PARAMETERS 7

6 Principles of Operation (POP)

The project implements a task that repeatedly samples and records hits on the program counter (PC). Our Task_ProgramTrace initializes a timer that will run periodically and call an interrupt service routine (ISR). The mainline of our task will then "spin" (do nothing but still loop) while the ISR obtains the current program counter and saves it. After one minute of sampling, a flag is switched and the ISR will spin, ceasing to collect more data. Now the mainline for Task_ProgramTrace will take the collected data and output the results to a console. Once the results are output, the mainline will flip the flag back so the ISR will collect more data. This will continue until externally terminated.

The key instrument used in synchronizing the task mainline and the ISR is a binary semaphore. Effectively, both will continue to execute forever as the timer that calls the ISR is periodic with no end and the mainline is wrapped in a while (1) block. The mainline while loop will likely execute much faster than the timer takes to terminate a period, so as mentioned, the semaphore will be used for synchronization. This is done by immediately taking the semaphore in the mainline. Once taken, execute the logic. The while loop will be executed again, only this time, it will block on the call to take the semaphore because it has already been taken. Next, the timer will terminate a period and the ISR will be called. The ISR will call a function to obtain the current PC. It is worth noting that the "current" PC we are interested in is the PC that we have context switched from upon entering the interrupt - that PC is the memory address we *were* in, which effectively traces where the program is spending some time. It would be pointless to obtain the real PC, because we already know we are in an interrupt. It is likely possible that the PC can be obtained using standard C mechanisms, however, it is much easier to write a short assembly function to do the job. We know that at least 10 items of 4 bytes each are pushed onto the stack during a context switch and the PC is the 9^{th} item. The stack pointer points to the bottom of the stack, so an offset of 32 bytes will get us to the PC. It is standard assembly convention to load return values into register 0. We then branch back from where we came from using the Link Register.

To keep things simple, we are not going to record every unique memory address that is obtained from the PC - that would be overkill as the discernable memory mappings have a range of memory, not a single location. We notice that the program will likely use less than 32 KiB, so we create 512 bins of 64 bytes each. After obtaining the PC we determine which bin that memory address falls into and increment that bin to denote a hit. Next, we need to check when to stop sampling. Right after the ISR timer was enabled, the current system tick was polled, and the stop system tick mark was calculated. The ISR will now check if that time has been reached, if it has, the control flag will be flipped – this is the mechanism that controls when the ISR collects and the mainline spins, and when the ISR spins and the mainline reports data. The last thing to be done in the ISR is "give" the semaphore back. We did not take the semaphore in the ISR but giving the semaphore back will allow the mainline that has been blocking on a call to take the semaphore, to successfully take it. After taking the semaphore, the mainline will check if the control flag has been flipped – if it has, then we will use a third-party tool, Report Data, to send our results to a console. After all the results have been reported, the bins are emptied, and the control flag is flipped back to allow data collection once again.

7 Data Structure Descriptions

The following table (*Table 7-1*) is a list of all Data Structures used in this project

Table 7-1 Data Structures

Data Structure Name	Type	Data Structure Description
histogram_array	uint32[512]	This array is used to store data values collected from the Program Counter by the ISR. It is initialized to all 0's by the function zero_histogram_array()

8 Function Descriptions

The following table (*Table 8-1*) contains each function defined for this project. *Table 8-2* contains the referenced external functions for each function in *Table 8-1*

Table 8-1 Functions

Function Name (Purpose)	Function Pseudo Code

Task_ProgramTrace
 	
This function's purpose work with Timer_0_A_ISR to collect and report data about current program profiling via UART. This task watches for a flag to be flipped by the ISR signaling that data collection is complete. The flag being flipped causes this task to begin sending data to the queue to be reported. After sending data is complete, the flag is flipped again and the ISR resumes collecting data.
	`// INITIALIZE Timer_0_A_Semaphore and SETUP Timer_A`
	`CALL vSemaphoreCreateBinary WITH Timer_0_A_Semaphore`
	
	`CALL SysCtlPeripheralEnable WITH SYSCTL_PERIPH_TIMER0`
	
	`CALL IntRegister WITH INT_TIMER0A AND Timer_0_A_ISR`
	
	`CALL TimerConfigure WITH TIMER0_BASE AND the bitwise-or combination of TIMER_CFG_SPLIT_PAIR AND TIMER_CFG_A_PERIODIC`
	
	`CALL TimerPrescaleSet WITH TIMER0_BASE, TIMER_A, AND the PRE_SCALE_VALUE`
	
	`CALL TimerLoadSet WITH TIMER0_BASE, TIMER_A, AND LOAD_VALUE`
	
	`CALL TimerIntEnable WITH TIMER0_BASE AND TIMER_TIMA_TIMEOUT`
	
	`CALL IntEnable WITH INT_TIMER0A to enable Timer_0_A interrupt`
	
	`CALL TimerEnable WITH TIMER0_BASE AND TIMER_A to enable start timer`

<table>
<tr><td></td><td><pre>CALL ReportData_SetOutputFormat WITH Excel_CSV to use
                   Excel output format for ReportData

CALL zero_histogram_array to initialize
                                      histogram_array

SET start_Sys_Tick TO xPortSysTickCount
SET stop_Sys_Tick TO THE CALCULATION start_Sys_Tick
       PLUS (REPORT_FREQUENCY_IN_SECONDS MULTIPLIEDBY
                            ONE_SECOND_DELTA_SYS_TICK

// BEGIN taking data
WHILE TRUE
  CALL xSemaphoreTake WITH Timer_0_A_Semaphore AND
                                        portMAX_DELAY
  IF current_ISR_Status EQUALS DONE_COLLECTING
    INCREMENT current_Histogram_Report

    CALL UARTprintf WITH “Done With Report #” message
    CALL report_histogram_data to send data to queue
    CALL zero_histogram_array to zero array

    SET start_Sys_Tick TO xPortSysTickCount
    SET stop_Sys_Tick TO THE CALCULATION
     start_Sys_Tick PLUS (REPORT_FREQUENCY_IN_SECONDS
               MULTIPLIEDBY ONE_SECOND_DELTA_SYS_TICK

    SET current_ISR_Status TO COLLECTING
  ENDIF
ENDWHILE</pre></td></tr>
<tr><td>Get_Value_From_Stack
(Assembly)

This function’s purpose is to obtain the current program counter from the stack</td><td><pre>DECLARE Get_Value_From_Stack as global

IN Get_Value_From_Stack
  LOAD VALUE FROM Stack Pointer, given an offset,
                                        into Register
  Branch back
END Get_Value_From_Stack</pre></td></tr>
<tr><td>Timer_0_A_ISR

Interrupt Service Routine used to collect data from the program counter.</td><td><pre>CALL TimerIntClear WITH TIMER0_BASE and
                                   TIMER_TIMA_TIMEOUT

IF current_ISR_Status EQUALTO COLLECTING
  SET current_PC TO RETURN VALUE FROM CALLING
                  Get_Value_From_Stack WITH PC_OFFSET
  SET current_PC TO THE FLOOR OF current_PC DIVIDEDBY
                                                 64.0

  IF current_PC LESSTHAN SIZE_OF_HISTOGRAM_ARRAY
    INCREMENT the index current_PC in histogram_array
  ELSE
    IF current_PC GREATERTHANOREQUALTO
                              SIZE_OF_HISTOGRAM_ARRAY</pre></td></tr>
</table>

<table>
<tr><td></td><td><pre>
    CALL UARTprinf with error “to large” message
  ELSEIF
    CALL UARTprinf with error “to small” message
  ENDIF

  CALL UARTprinf to output current_PC
ENDIF

IF xPortSysTickCount GREATERTHAN stop_Sys_Tick
  SET current_ISR_Status TO DONE_COLLECTING
ENDIF
ENDIF

CALL xSemaphoreGiveFromISR WITH Timer_0_A_Semaphore
                        AND &xHigherPriorityTaskWoken
</pre></td></tr>
<tr><td>report_histogram_data

This function will add each index of the histogram_array to a ReportData_Item, then add each of those items to the ReportData_Queue</td><td><pre>
FOR each i between 0 and 512
 INIT a ReportData_Item named item;
 SET item TimeStamp as xPortSysTickCount
 SET item ReportName as 42
 SET item ReportValueType_Flg as 0x0
 SET item ReportValue_0 as i
 SET item ReportValue_1 as index i in histogram_array
 SET item ReportValue_2 as 0
 SET item ReportValue_3 as 0

 CALL xQueueSend WITH ReportData_Queue item_ref AND 0
ENDFOR
</pre></td></tr>
<tr><td>zero_histogram_array

This function will set all indices in the histogram_array to 0</td><td><pre>
FOR each index in histogram_array
  SET histogram_array at index to 0
ENDFOR
</pre></td></tr>
</table>

Table 8-2 Referenced External Functions

Function Name	Referenced External Functions
Task_ProgramTrace	IntEnable IntRegister ReportData_SetOutputFormat(Excel_CSV); SysCtlPeripheralEnable TimerConfigure TimerEnable TimerIntEnable TimerLoadSet TimerPrescaleSet

	UARTprintf vSemaphoreCreateBinary xSemaphoreTake
Get_Value_From_Stack	None
Timer_0_A_ISR	TimerIntClear UARTprintf xSemaphoreGiveFromISR
report_histogram_data	xQueueSend()
zero_histogram_array	None

9 Parameters

The following table (*Table 9-1*) contains a list of parameters used in this project

Table 9-1 Parameters

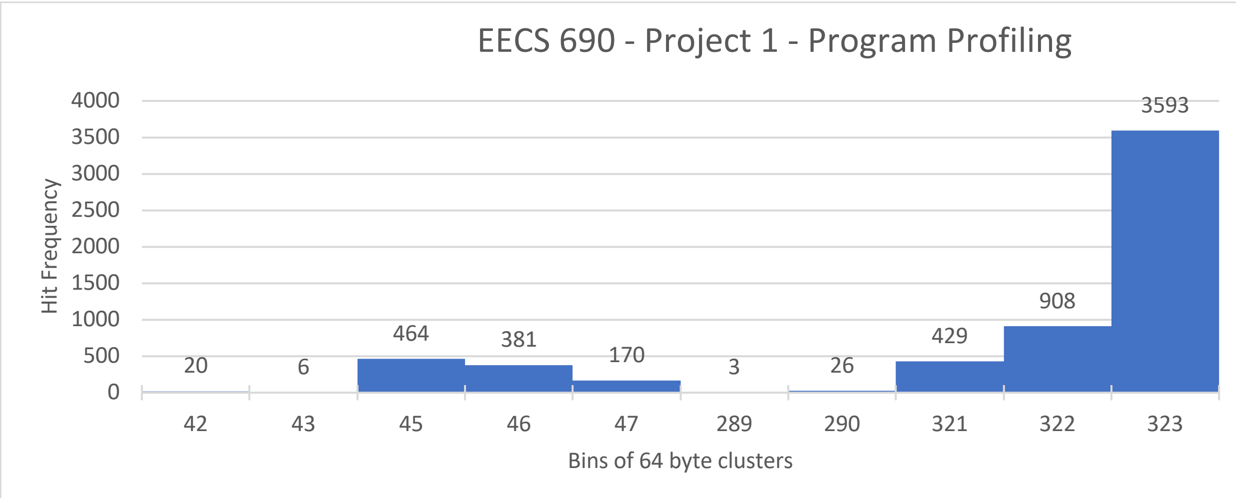
Parameter Name	Parameter Type	Parameter Default Value	Parameter Description
PC_OFFSET	const uint32_t	32	Program Counter Offset, (512 << 6)
LOAD_VALUE	const uint32_t	50000	Load value, chosen arbitrarily
PRE_SCALE_VALUE	const uint32_t	23	Pre-scale value (must be < 256)
SIZE_OF_HISTOGRAM_ARRAY	const uint32_t	512	Contains size of the histogram array. This is the number of possible bins data is collected in
ONE_SECOND_DELTA_SYS_TICK	const uint32_t	10000	1 second in systicks
REPORT_FREQUENCY_IN_SECONDS	const uint32_t	60	How frequently should the program report data.

Timer_0_A_Semaphore;	xSemaphoreHandle	N/A	Semaphor for Timer_0_A
xHigherPriorityTaskWoken	portBASE_TYPE	pdFALSE	Extracted from the ISR to keep stack items to a minimum
Current_PC	uint32_t	0	Contains the current Program Counter. Initialized to 0 because it doesn't matter initial value, as it is only used as a storage container to hold the value from the program counter.
current_ISR_Status	ISR_STATUS_t	COLLECTING	Status of our PC value. Controls when data is collected/reported. COLLECTING is chosen by default to ISR starts collecting immediately.
start_Sys_Tick	uint32_t	0	Variable to hold initial system tick in for each round of gathering data. Initialized to 0 because it will be set in Task_ProgramTrace before it is used. This ensures that as little time is not accounted for by program startup and overhead.
stop_Sys_Tick	uint32_t	0	Variable to hold final system tick in for each round of gathering data. Same as start_Sys_Tick
current_Histogram_Report	uint32_t	0	Tracks how many histogram reports have been output Initialized to 0 because at program start no reports have been made

10 Testing

After one minute of sampling we discovered that very few bins were even hit once. This is not a cause for concern, given that the program only has four tasks being scheduled, and a relatively low number of libraries and technologies being utilized.

Figure 10-1 Results Histogram



As shown in *Figure 10-1*, bin 323 is by far the highest used block of memory. The data was zero indexed, so bin 323 refers to memory range from 20,672 bytes to 20,736 bytes. According to our map file, that memory range belongs to portasm.obj (.text). portasm.obj contains the assembly context switching routines. It makes sense that context switching is being hit a lot, we are context switching every time the ISR is called and every time the CPU executes a different task. Bin 321 and 322 are also in the range of the portasm.obj. Bins 42, 43, 45, 46, and 47 are the tasks.obj file which also makes sense being so large since everything that is running is propagated through a task. Bins 289 and 290 are driverLib.lib file which are not used very much.

11 Lessons Learned

Our results may not be line by line specific, but it provides a good indication of where our time is spent. According to our results, we spend a lot of time context switching - this is not an uncommon resource gobbler. Now that we know this, it may be worth experimenting with parameters that affect context switching. This may also indicate a good reason to use a custom operating system - attributes such as the quantum (CPU execution time) given to each task before switching are changeable. There are many tradeoffs when dealing with context switching, and it may be difficult to pick a solution, however, knowing where to optimize is the first step.

Appendix A – Program Source Code

```c
/**
* @Filename: Task_ProgramTrace.c
* @Author:   Kaiser Mittenburg and Ben Sokol
* @Email:    ben@bensokol.com
* @Email:    kaisermittenburg@gmail.com
* @Created:  August 30th, 2018 [1:35pm]
* @Modified: September 17th, 2018 [6:45pm]
* @Version:  1.0.0
*
* @Description: Periodically traces current program memory location
*
* Copyright (C) 2018 by Kaiser Mittenburg and Ben Sokol. All Rights
Reserved.
*/

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_types.h"
#include "inc/hw_uart.h"

#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <math.h>

#include "Drivers/UARTStdio_Initialization.h"
#include "Drivers/uartstdio.h"

#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/timer.h"

#include "Tasks/Task_ReportData.h"

#include "FreeRTOS.h"
#include "semphr.h"
#include "task.h"


/*************************************************
* External variables
*************************************************/

// Access to current Sys Tick
extern volatile long int xPortSysTickCount;


```

```c
/**********************************************
* External functions declarations
**********************************************/

// Assembly function to get PC from the stack
extern uint32_t Get_Value_From_Stack(uint32_t);

/**********************************************
* Local task constant types
**********************************************/
typedef enum ISR_STATUS_t {
  COLLECTING,      // Should ISR collect data
  DONE_COLLECTING  // ISR is done collecting data, reporting
} ISR_STATUS_t;


/**********************************************
* Local task constant variables
**********************************************/

// Program Constants
// We operate at 120 MHz, which gives a period of 8.33 nS
// The equation 8.33nS * K * M = 10mS
// The LOAD_VALUE (M) must be < 64k, 50,000 chosen arbitrarily
// The PRE_SCALE_VALUE (K) must be < 256. Solving, K = 24
// Since K is zero indexed, K = 23
// We are only interested in memory <= 32KiB which is 2^15
const uint32_t PC_OFFSET                    = 32;
const uint32_t LOAD_VALUE                   = 50000;
const uint32_t PRE_SCALE_VALUE              = 23;
const uint32_t HISTOGRAM_ARRAY_SIZE         = 512;  // (512 << 6) == 32KiB
const uint32_t ONE_SECOND_DELTA_SYS_TICK    = 10000;
const uint32_t REPORT_FREQUENCY_IN_SECONDS = 60;


/**********************************************
* Local task variables
**********************************************/

// The semaphore
xSemaphoreHandle   Timer_0_A_Semaphore;

// Data array
uint32_t           histogram_array[512];

// Extracted from the ISR to keep stack items to a minimum
portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

// The current memory address obtained from the PC
uint32_t           current_PC = 0;

// Status of our PC value. Controls when data is collected/reported
ISR_STATUS_t       current_ISR_Status = COLLECTING;

// Sys_Tick when ISR starts collecting
```

```c
uint32_t            start_Sys_Tick = 0;

// Sys_Tick when ISR needs to stop collecting
uint32_t            stop_Sys_Tick = 0;

// How many reports have been output
uint32_t            current_Histogram_Report = 0;


/************************************************
* Local task function declarations
************************************************/
extern void Timer_0_A_ISR();
extern void Task_ProgramTrace(void* pvParameters);
extern void report_histogram_data();
extern void zero_histogram_array();

/************************************************
* Local task function definitions
************************************************/

/*************************************************************************
* Function Name: Timer_0_A_ISR
* Description:   Interrupt Service Routine used to profile tasks
* Parameters:    N/A
* Return:        void
*************************************************************************/
extern void Timer_0_A_ISR() {
  TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

  if (current_ISR_Status == COLLECTING) {
    // Get the value from the PC
    current_PC = Get_Value_From_Stack(PC_OFFSET);
    current_PC = floor( current_PC / 64.0 );

    // Validate Current_PC value is within size of array and store
    if (current_PC < HISTOGRAM_ARRAY_SIZE) {
      // Increment Bin for Current_PC
      histogram_array[current_PC]++;
    }
    else {
      // Current_PC is out of range.
      // In theory should never enter this else statement
      if (current_PC >= HISTOGRAM_ARRAY_SIZE) {
        UARTprintf("ERROR: (Current_PC / 64) >= %i", HISTOGRAM_ARRAY_SIZE);
      }
      else {
        UARTprintf("ERROR: (Current_PC / 64) < 0");
      }
      UARTprintf(" (Current_PC = %u)\n");
    }

    if (xPortSysTickCount > stop_Sys_Tick) {
      current_ISR_Status = DONE_COLLECTING;
    }
```

```
  }

  // "Give" the Timer_0_A_Semaphore
  xSemaphoreGiveFromISR(Timer_0_A_Semaphore, &xHigherPriorityTaskWoken);
}


/*************************************************************************
* Function Name: Task_ProgramTrace
* Description:   Task used to initialize Timer_0_A_ISR
* Parameters:    void* pvParameters;
* Return:        void
*************************************************************************/
extern void Task_ProgramTrace(void* pvParameters) {
  //Initialize Semaphore and setup Timer
  vSemaphoreCreateBinary(Timer_0_A_Semaphore);

  SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

  IntRegister(INT_TIMER0A, Timer_0_A_ISR);

  TimerConfigure(TIMER0_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_A_PERIODIC);

  TimerPrescaleSet(TIMER0_BASE, TIMER_A, PRE_SCALE_VALUE);

  TimerLoadSet(TIMER0_BASE, TIMER_A, LOAD_VALUE);

  TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

  // Enable Timer_0_A interrupt in NVIC
  IntEnable(INT_TIMER0A);

  // Enable (Start) Timer
  TimerEnable(TIMER0_BASE, TIMER_A);

  // Set data report to Excel format
  ReportData_SetOutputFormat(Excel_CSV);

  // Init the data array
  zero_histogram_array();

  // Set start time based on current systick, stop time = 1 minute later.
  start_Sys_Tick = xPortSysTickCount;
  stop_Sys_Tick = start_Sys_Tick
              + (REPORT_FREQUENCY_IN_SECONDS * ONE_SECOND_DELTA_SYS_TICK);

  // Add values to the histogram when appropriate
  while (1) {
    xSemaphoreTake(Timer_0_A_Semaphore, portMAX_DELAY);
    if (current_ISR_Status == DONE_COLLECTING) {
      current_Histogram_Report++;

      UARTprintf("DONE COLLECTING (%u)\n", current_Histogram_Report);
      report_histogram_data();

```

```
      // Zero array to make sure overflow doesnt happen
      zero_histogram_array();

      // Reset time to start collecting again for 1 minute
      start_Sys_Tick = xPortSysTickCount;
      stop_Sys_Tick = start_Sys_Tick
              + (REPORT_FREQUENCY_IN_SECONDS * ONE_SECOND_DELTA_SYS_TICK);

      // Set flag current_ISR_Status to start collecting again
      current_ISR_Status = COLLECTING;
    }
  }
}


/*************************************************************************
* Function Name: report_histogram_data
* Description:   Function used to send data to ReportData_Queue
* Parameters:    N/A
* Return:        void
*************************************************************************/
extern void report_histogram_data() {
  uint32_t i = 0;
  for (i = 0; i < 512; ++i) {
    ReportData_Item item;
    item.TimeStamp = xPortSysTickCount;
    item.ReportName = 42;
    item.ReportValueType_Flg = 0x0;
    item.ReportValue_0 = i;
    item.ReportValue_1 = histogram_array[i];
    item.ReportValue_2 = 0;
    item.ReportValue_3 = 0;

    // This sends copy of data
    xQueueSend(ReportData_Queue, &item, 0);
  }
}


/*************************************************************************
* Function Name: zero_histogram_array
* Description:   Function used to initialize histogram array to all zeros
* Parameters:    N/A
* Return:        void
*************************************************************************/

extern void zero_histogram_array() {
  uint32_t i = 0;
  for (i = 0; i < 512; ++i) {
    histogram_array[i] = 0;
  }
}
```