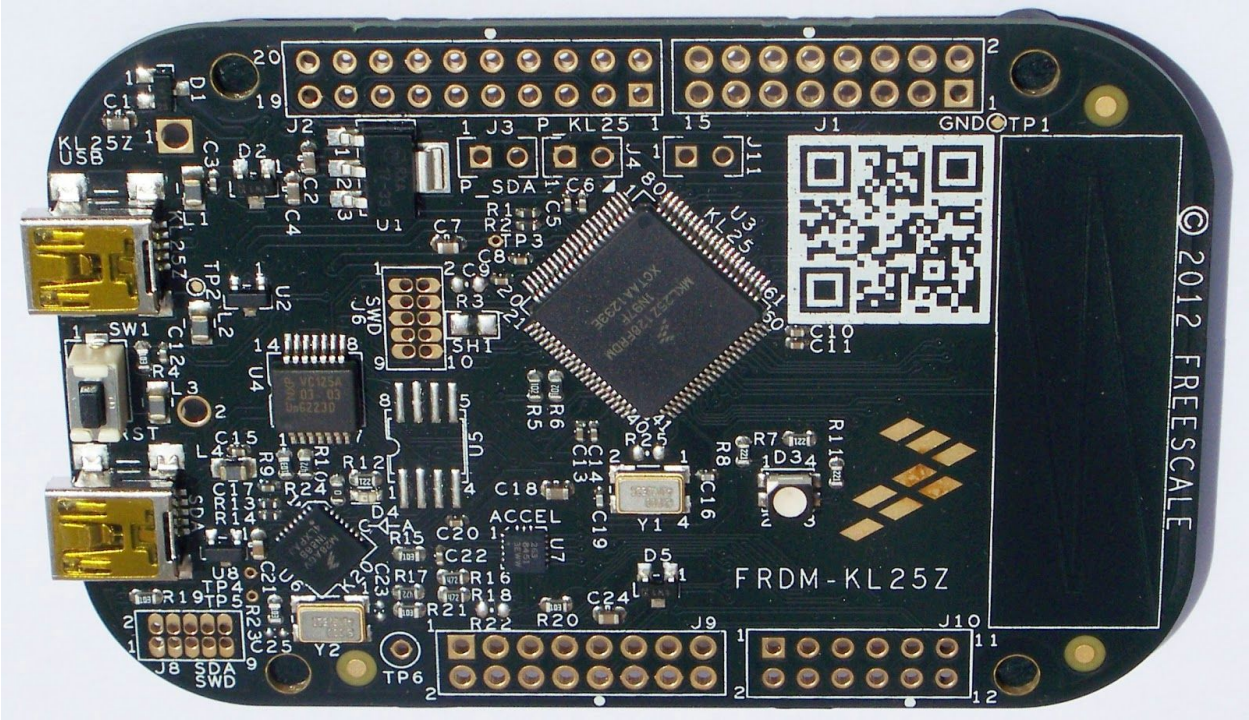
ECEN 5013:
Embedded Software Essentials



Project 2

Microcontroller Timers, DMA and Profiling

Diana Southard

Introduction

The purpose of this lab was to gain familiarity with the programming of the FRDM-KL25Z Freescale Board through an LED Fader program, and then to check the runtime execution of code created in the previous project suing the board's 48 MHs clock. By obtaining the execution times of the memory functions created in project 1, I was able to gain some sense of my code's efficiency. I then used the Freescale Board with modified memory functions to further compare the efficiency of using Direct Memory Access (DMA) to transfer data, comparing the execution time of the board using the original memory functions to the operations once I triggered the DMA to start functioning.

All final code can be found at the Github repository found here:
https://github.com/dSouthard/ECEN5013_Homework

This repository can be found using the clone command:

```
git clone https://github.com/dSouthard/ECEN5013_Homework.git
```

This code in this project was kept to be as portable as possible. The code found at the above repository should be ready to work upon compilation.

Procedure Results

KDS Timer Interrupts and RGB PWM

The LED fader was constructed by setting the LED's red, green, and blue cathodes to PWM-capable pins, and then adjusting their intensity with the use of a calculated sine wave to cycle through the colors of the rainbow, as seen below.

The internal 4Mhz clock was used on the board. This was further divided into the TPM0 and TPM2 in order to generate pulses with a frequency of about 200 Hz, as seen below:

Timer Setup: TPM0/TPM2 Timers Used for LED Fading

```
/* LED Timers: slow enough to be seen by the human eye, fast
 * enough to be a seamless fade --> 200Hz frequency (200 ticks/sec)
 *
 * Accomplished by dividing the fast 4 MHz clock by 128 by setting
 * the prescaler bits to 7 [DIVIDE_BY_128] and further divide it by
 * 155 setting the modulus register to 154 [DIVIDE_TO_200HZ]
 *
 * 4 MHz / 128 / 155 = ~200Hz (201.61Hz)
 */
// TPM0: Used to toggle blue LED
TPM0_BASE_PTR->SC = TPM_SC_CMOD(1) | TPM_SC_PS(DIVIDE_BY_128);
```

```
TPM0_BASE_PTR->MOD = DIVIDE_TO_200HZ;
// TPM2: Used to toggle green/red LEDs
TPM2_BASE_PTR->SC = TPM_SC_CMOD(1) | TPM_SC_PS(DIVIDE_BY_128);
TPM2_BASE_PTR->MOD = DIVIDE_TO_200HZ;
```

In order to accomplish the fading, TPM0 was setup as an interrupt signal, and the LEDs were updated in its interrupt handler, as seen below:

TPM0 IRQ Handler Snippet:

```
// Increment counter used for LED Fading
 portion = 0.5 + sin(phase) / 2;
 TPM2_BASE_PTR->CONTROLS[RED_LED].CnV = TPM2_BASE_PTR->MOD * portion;
 TPM2_BASE_PTR->CONTROLS[GREEN_LED].CnV = TPM2_BASE_PTR->MOD * portion/2;
 TPM0_BASE_PTR->CONTROLS[BLUE_LED].CnV = TPM0_BASE_PTR->MOD * portion/4;
 phase += 0.5;
```

This method allowed the LEDs to gradually transition through a rainbow of color.

Code Profiler using Your Timer

In the previous project, I created four memory functions: `memcpy`, `memmove`, `memzero`, and `reverse`. The purpose of `memmove` was to override the built-in function of the same name. It would transfer a specified length of bytes from a source location to a destination location, checking and accounting for overlap between the two locations. The purpose of `memzero` was to fill a source location with a specified length of zeros. The purpose of `reverse` was to reverse a specified length of memory starting from the source location.

I wanted to use the Freescale Board to gather some statistics on my code's execution length in addition to some other functions that are provided by the standard libraries. To do so, I created a 10 usec counter to use as my execution timer. I recorded the value of the timer counter first before calling the function being profiled and then again when the function was complete. Once completed, I subtracted the two values and report the time in μseconds. This timer was dynamic, using a compile time switch.

The profiling timer was created as shown below:

Timer Setup: TPM1 Timer Used for Profiling

```
/*
* Profiling timer: required to have 10us period -> 100 kHz
       * Accomplished by dividing the fast 4 MHz clock by 8 by setting
       * the prescaler bits to 3 [DIVIDE_BY_8] and further divide it by
       * 5 setting the modulus register to 4 [DIVIDE_TO_10kHZ]
       *
       * 4 MHz / 8 / 5 = 100 kHz
*/
// TPM1: Used to profile functions
TPM1_BASE_PTR->SC = TPM_SC_CMOD(1) | TPM_SC_PS(DIVIDE_BY_8);
```

```
TPM1_BASE_PTR->MOD = DIVIDE_TO_10kHZ;
```

Additionally, the code to read the counter's value before and after a function was called is shown below.

Counter Value Reading Snippet: Testing Memmove

```
int _testMemmove(int byteLength) {
      int startTime = profilerCounter;
      memmove(src,dst,byteLength);
      int endTime = profilerCounter;
      return endTime - startTime;
}
```

This code snippet was repeated for each function being profiled. The only differing test was during the test of strcpy. The strcpy function copies one string to another, going from the beginning of the source string until a null terminator is reached. The strcpy test function can be seen below:

Testing strcpy Snippet: Inserting and Removing String Terminator

```
int _testStrcpy(int byteLength) {
      // Set the last character in src to be null
      // strcpy copies until a null character
      src[byteLength-1] = '\0';
      int startTime = profilerCounter;
      strcpy(dst,src);
      int endTime = profilerCounter;
      // remove null character for future tests
      src[byteLength-1] = 0;
      return endTime - startTime;
}
```

As you can see, a string terminator character was inserted at the desired testing length before the function was tested, then removed afterwards to prevent any issues in future testing.

Each function was profiled using the same length of bytes which were increased in the following increments of bytes: 10, 100, 1000, 5000. To create the necessary memory used in the tests, two test data arrays were created that was filled with numbers counting from 1 to 5000 as arbitrary testing data. One array was used as the source location, the other was used as the destination location. Additionally, the results was stored in a matrix, with each column being dedicated to a particular function being profiled and each row being dedicated to the length that function was tested with.

Each function was tested with the source and destination pointers starting at the beginning of the test data array. This did not allow for overlap between the two functions, simplifying the profiling and creating uniform testing environments between the different profiling case.

Viewing these results, it is clear that my versions of `memmove` and `memzero` work faster than the library versions. This can be attributed to the fact that my versions are much more stripped down than their library counterparts, with less flexibility and portability than what those versions offer.

In order to profile both the standard memory function included in <string.h> and the memory functions that I wrote in project 1, I used the preprocessor to determine which header file should be included. This decision can be changed in main.c, as shown below:'

Main.c Snippet: Determining Which Memory Function to Test

```
// Split work into sections so that the program knows which memory functions to
include and test
// NOTE: ONLY ENABLE ONE OF THE FOLLOWING DEFINITIONS AT A TIME
#define PART2_MYFUNCTIONS // Part 2: profiling memory functions from project 1
//#define PART2_BUILTIN    // Part 2: profiling memory functions from library
```

There is a split between **My Versions** and **Library Versions**. If `PART2_MYFUNCTIONS` is defined, then the results will show the test results corresponding with **My Versions**. If `PART2_BUILTIN` is defined, the the **Library Versions** results are shown. This can be seen in the profiler.c snippet shown below:

Profiler.c Snippet: Determining Which Memory Function to Test

```
// Include project 1 memory functions
#ifdef PART2_MYFUNCTIONS
	#include"memory.h"
#endif
// Include built-in functions
#ifdef PART2_BUILTIN
	#include <string.h>
#endif
```

These definitions also determined the size of the results storage matrix, where the elapsed time per function was stored. In order to view those results, I printed them after having concluded all tests. The function I used can be seen below. It would print out the result matrix as sized by the case in Part 2 that was currently being tested, **My Versions** or **Library Versions**.

Printing Profile Test Results Snippet

```
void printTestResults() {
	int column, row;
	printf ("Printing results: \n");
	for (column = 0; column < MAX_COLUMNS; column++) {
		printf ("Column %d: \t", column);
		for (row = 0; row < MAX_ROWS; row++) {
			printf("%d \t", results[column][row]);
		}
		printf("\n");
```

```
        }
        printf ("End of results. \n");
}
```

After profiling the functions, I profiled the operation of `printf` with various options, number of inputs and print types. The `printf` statements were profiled in the same method as described above, with the results being stored in a single array. Those options and the resulting times can be seen below. This resulted in 13 test cases.

I used the following data for my different variables under test. This ensured that all profiling was as consistent as possible.

Variables Used for `printf` Profiling

```
// Data for testing printf statements
#define PRINT_STRING          "This is my test string"
#define PRINT_VAR1       9191
#define PRINT_VAR2       -5.51
#define PRINT_VAR3       1.010101985
```

My test cases were run as follows:

Test Cases for printf Profiling

```
/* Cases are as follows:
 * 1: Some 20 character string
 * 2: 1 decimal variable, no precision stated
 * 3: 2 decimal variables, no precision stated
 * 4: 3 decimal variables, no precision stated
 * 5: 1 float variable, no precision stated
 * 6: 2 float variables, no precision stated
 * 7: 3 float variables, no precision stated
 * 8: 1 float variable, .2f precision stated
 * 9: 2 float variables, .2f precision stated
 * 10: 2 float variables, .2f precision stated
 * 11: 1 float variables, .8f precision stated
 * 12: 2 float variables, .8f precision stated
 * 13: 3 float variables, .8f precision stated
 */
```

I created several functions to account for these test cases, as seen below:

Profiling printf functions

```
int printNoPrecisionD(int var1, int var2, int var3) {
      int startTime, endTime;
      if (var1 != 0)
            if (var2 != 0)
                  if (var3 != 0) {
                        startTime = profilerCounter;
                        printf("%d, %d, %d", var1, var2, var3);
```

```
                        endTime = profilerCounter;
                    }
                    else {
                        startTime = profilerCounter;
                        printf("%d, %d", var1, var2);
                        endTime = profilerCounter;
                    }
                else {
                    startTime = profilerCounter;
                    printf("%d", var1);
                    endTime = profilerCounter;
                }
            else {
                startTime = profilerCounter;
                printf(PRINT_STRING);
                endTime = profilerCounter;
            }
            return endTime - startTime;
        }
```

From my table above, I can see how adding multiple variables and increasing the precision of the printf statement can dramatically increase the required time needed to complete the function.

DMA Operation and Profiling

My previous profiling included several large data transfers. On the Freescale Board, there is a piece of hardware called the Direct Memory Access (DMA) controller. The DMA controller module enables fast transfers of data, providing an efficient way to move blocks of data with minimal processor interaction. The DMA module has four channels that allow 8-bit, 16-bit, or 32-bit data transfers.

This offloading allows the processor to spend time doing other useful operations called for in the program, greatly improving system performance. The DMA controller has access to the Bus controller and can pass data between peripheral devices and memory or just between memory interfaces.

Any operation involving a DMA channel follows the same three steps:

1. Channel initialization — The transfer control descriptor, contained in the channel registers, is loaded with address pointers, a byte-transfer count, and control information using accesses from the slave peripheral bus.

2. Data transfer — The DMA accepts requests for data transfers. Upon receipt of a request, it provides address and bus control for the transfers via its master connection to the system bus and temporary storage for the read data. The channel performs one or more source read and destination write data transfers.

3. Channel termination — Occurs after the operation is finished successfully or due to an error. The channel indicates the operation status in the channel's DSR, described in the definitions of the DMA Status Registers (`DSRn`) and Byte Count Registers (`BCRn`).

Before a data transfer starts, the channel's transfer control descriptor must be initialized with information describing configuration, request-generation method, and pointers to the data to be moved. The four DMA channels are prioritized based on number, with channel 0 having highest priority and channel 3 having the lowest. For this project, I used DMA Channel 0.

In our previous project, the memory functions were written to transfer bytes or 8-bits at a time. In order to allow the `memove` and `memzero` functions to work with the DMA, I set up DMA0 to transfer 8-bits as seen below:

DMA Setup: 8-bit Transfer Size

```
void dmaSetup() {
	// Using 8-bit transfer on DMA Channel 0
	DMA0->DMA[DMA_CH0].DCR = DMA_DCR_SSIZE(TRANSFER_8BIT) |
DMA_DCR_DSIZE(TRANSFER_8BIT);
}
```

A typical call for the memmove function looks like this:

```
void memmove(uint8_t * src,uint8_t * dst,uint32_t byteLength);
```

I will use these terms as I talk about the general guidelines I followed for programming the DMA. The guidelines are

- `TCD0` is initialized.
  - `SAR0` is loaded with the source (read) address. Because the transfer is from memory to memory, the source address is the starting address of the data block. `DAR0` is initialized with the destination (write) address. Because the transfer is from memory to memory, `DAR0` is loaded with the starting address of the data block to be written. This was accomplished as seen below:

DMA: Loading Source and Destination Address

```
// Load src and dst addresses.
void loadSrc(uint32_t * src) {
	DMA0->DMA[DMA_CH0].SAR = (uint32_t)src;
}
void loadDst(uint32_t * dst) {
	DMA0->DMA[DMA_CH0].DAR = (uint32_t)dst;
}
```

  - `SAR0` and `DAR0` change after each data transfer depending on `DCR0`[`SSIZE`, `DSIZE`, `SINC`, `DINC`, `SMOD`, `DMOD`] and the starting addresses. The increment value was 1 for an 8-bit. If the address register is programmed to remain unchanged, the register is not incremented after the data transfer. This was accomplished as seen below:

DMA: Source and Destination Incrementing Setup

```
/* SAR0 and DAR0 change after each data transfer. However, memzero does not take in *
a dst register. So keep the incrementing in two functions.
*/
void srcIncrement() {
	DMA0->DMA[DMA_CH0].DCR |= DMA_DCR_SINC(INCREMENT_SIZE);
}
void dstIncrement () {
	DMA0->DMA[DMA_CH0].DCR |= DMA_DCR_DINC(INCREMENT_SIZE);
}
```

• `BCR0[BCR]` must be loaded with the total number of bytes to be transferred. It is decremented by 1 at the end of each transfer, due to the transfer size being 8-bit. `DSR0[DONE]` must be cleared for channel startup.This was accomplished as seen below:

DMA: BCR Setup

```
// Load the memory transfer byte size
void loadByteSize(uint32_t size) {
	DMA0->DMA[DMA_CH0].DSR_BCR = DMA_DSR_BCR_BCR_MASK & size;
}
```

• After the channel has been initialized, the transfer is started by setting `DCR0[START]` to use a software-initiated transfer as part of a single 32-bit write to the last 32 bits of the `TCD0`. Because I was using a software-initiated transfer, it is not required to write the `DCR0` with `START` cleared and then perform a second write to explicitly set `START`. Programming the channel for a software-initiated request also causes the channel to request the system bus and start transferring data immediately

DMA: Data Transfer

```
void transferData() {
	// Start the transfer
	DMA0->DMA[DMA_CH0].DCR |= DMA_DCR_START_MASK;
	// Wait while transferring data
	while (!(DMA0->DMA[DMA_CH0].DSR_BCR & DMA_DSR_BCR_DONE_MASK))
	{};
}
```

In order to allow the `memmove` function to work with the DMA Controller, the above functions were called and used only when the `memmove` function determined that there would be no overlap. The DMA controller moves incrementally forward through the source and destination locations. In the case of overlap, memory needs to be copied from the end to the beginning in order to maintain the integrity of the move. For these cases, the `memmove` function behaved the same as it did in Project 1.

DMA: memmove Function Snippet

```
if ((src < dst) && ((src + length) > dst))
      /* < -- src -- >
       * < -- dst -- > Overlap
       * Copy from end of src to end of dst
       * DMA CANNOT handle the transfer, perform it normally
       */
for (index = length; index > 0; index--)
dst[index-1] = src[index-1];
else {
/* All other cases: DMA CAN handle the transfer!
* Transfer left -> right
*/
      // Setup increment of src/dst DMA registers
      srcIncrement();
      dstIncrement();
      // Load registers with value
      loadSrc(src, length);
      loadDst(dst);
      transferData();
}
return 0;
```

In the case of `memzero`, the destination register of `memmove` was set as the source of the transfer. The source incrementing was changed such that no incrementation happened, and the address of the zero register was passed in as the source location. In this way, the DMA controller would continually pass in 0's as it incremented along the destination register, accomplishing our desired effect of filling the source location with the specified number of zeros.

DMA: memzero Function Snippet

```
      // Setup source register to not increment during transfer --> stay in
same spot
      void srcNoIncrement() {
            DMA0->DMA[DMA_REG].DCR &= ~DMA_DCR_SINC_MASK;
      }

      // Implementation of memzero
      int8_t memzero(uint8_t * src, uint32_t length){
            // Check for invalid input
            if (src == NULL || !(length > 0))
            return -1;

            srcNoIncrement();
            dstIncrement();
            loadSrc(&zero);
```

```
        loadDst(src);
        loadByteSize(length);
        transferData();
        return 0;
}
```

The profiling efforts were the same as in the previous part. I used a timer interrupt to create a 10 μsecond counter in order to tell the elapsed time between when a function was called and when it was completed. I used the same test data array setup and profiled `memove` and `memzero` with the same incrementation of byte length as performed in Part 2.

The increase in system efficiency could be seen right away. The DMA controller increased the system efficiency tremendously.

Conclusion

This lab was very useful in learning how to program on the Freescale Board. However, it was very frustrating to get the initial setup working correctly. My personal computer runs Ubuntu and the need to find a Windows 7 machine took up a lot more time than would be expected in order to update the board's firmware to allow to program on it.

The demonstration of how useful DMA operations are to freeing up processor resources was enlightening. It is a much more efficient way to transfer data, resulting in a program that exceeds execution time expectations. The project showed what a benefit using the board's other hardware could have if a project required large data transfer.