Jonathan W. Valvano

First Name: _____ Last Name:_____

April 21, 2017, 10:00 to 10:50am

Open book, open notes, calculator (no laptops, phones, devices with screens larger than a TI-89 calculator, devices with wireless communication). Please don't turn in any extra sheets.

(10) Question 1. You wish to use DMA to move a 256 (0x100) byte buffer from location 0x2000.0000 to location 0x2000.0040 (notice they overlap). What will be the DMA settings?

Start source address =	0x2000.00FF	
Start destination address =	0x2000.013F	
Source data size =	1	1 2 or 4 bytes
Destination data size =	1	1 2 or 4 bytes
Source address increment =	-1	-4, -2, -1, 0, 1, 2, 4
Destination address increment =	-1	-4, -2, -1, 0, 1, 2, 4
Count (number of elements) =	256	

You could have had size 2 (or 4), count 128 (or 64), and decremented by -2 (or -4). The starting address then would be 0x2000.00FE (or 0x2000.00FC), and the destination address 0x2000.013E (or 0x2000.013C).

(25) Question 2. Your OS supports 10 processes. Each process has a number (from 0 to 9). In order to provide protection from one process to another, there will be a separate memory manager for each process. In particular, there will be 10 independent memory managers. You are given a memory manager within your OS with the following prototype:

```
void *malloc(uint32_t size, uint32_t pnum);
```

where **`size`** is the number of bytes to be allocated and **`pnum`** is the process number. The return parameter is a pointer to a memory block of the correct number of bytes. You do not write **`malloc`**, rather you will make the connections so when the user calls **`OS_malloc`**, the appropriate manager for that process is used. In your OS, there is private global containing the process number of currently running process (0 to 9)

```
unsigned long static ProcessNum=0;
```

In OS.h, within the user project there is a prototype for an OS function.

```
void *OS_malloc(uint32_t size);
```

There are other software interrupts (SVC), but you will use #99 for this OS call.

(10) Part a) Give the assembly code for **`OS_malloc`** in the osasm.s file within the user project.

```
OS_malloc
     SVC        #99
     BX         LR
```

(10) Part b) Give the assembly code for the **`SVC_Handler`** within the OS project. You may assume there are other OS calls that use SVC, but you only have to show this one.

```
SVC_Handler
    LDR  R12,[SP,#24]  ; Return address
    LDRH R12,[R12,#-2] ; SVC instruction is 2 bytes
    BIC  R12,#0xFF00   ; Extract trap number in R12
    LDM  SP,{R0-R3}    ; Get any parameters
    PUSH {LR}
    CMP  R12,#99
    BEQ  doMalloc
// other traps
doMalloc  ;R0=size, R1=ProcessNum
    LDR  R1,=ProcessNum
    LDR  R1,[R1]
    BL   malloc        ; Call OS routine #99
    POP  {LR}
    STR  R0,[SP]       ; Store return value
    BX   LR            ; Return from exception
```

(15) Question 3. Consider this user code, written in C, with its corresponding compiler generated assembly using the standard version of Keil (like Labs 1-4). In this code,

`IdleCount` is a global in RAM at address 0x200000CC.
`IdleTask` is in ROM at address 0x00001240.
`WaitForInterrupt` is function, also in ROM but at address 0x00000336
`PB2` is an I/O port at address 0x40005010

Which object code needs patching?

```
void IdleTask(void){
  IdleCount = 0;

  for(;;){

    IdleCount++;




    PB2 ^= 0x04;




    WaitForInterrupt();
  }
}
```

```
    62:   IdleCount = 0;
0x1240 2000      MOVS     r0,#0x00
0x1242 4908      LDR      r1,[pc,#32]  ; @0x00001264
0x1244 6008      STR      r0,[r1,#0x00]
    63:   for(;;){
0x1246 BF00      NOP
    64:     IdleCount++;
0x1248 4806      LDR      r0,[pc,#24]  ; @0x00001264
0x124A 6800      LDR      r0,[r0,#0x00]
0x124C 1C40      ADDS     r0,r0,#1
0x124E 4905      LDR      r1,[pc,#20]  ; @0x00001264
0x1250 6008      STR      r0,[r1,#0x00]
    65:     PB2 ^= 0x04;    // toggle PB2
0x1252 4805      LDR      r0,[pc,#20]  ; @0x00001268
0x1254 6900      LDR      r0,[r0,#0x10]
0x1256 F0800004  EOR      r0,r0,#0x04
0x125A 4903      LDR      r1,[pc,#12]  ; @0x00001268
0x125C 6108      STR      r0,[r1,#0x10]
    66:     WaitForInterrupt();
0x125E F7FFF86A  BL.W     WaitForInterrupt (0x00000336)
0x1262 E7F1      B        0x00001248
0x1264 00CC      DCW      0x00CC <- Patch this
0x1266 2000      DCW      0x2000
0x1268 5000      DCW      0x5000
0x126A 4000      DCW      0x4000
```

(5) Part a) Assume **`IdleTask`** and **`WaitForInterrupt`** are in the same process, and you wish to relocate both functions to another place in memory (without recompiling), but the relative distance between these two functions will remain constant. Look at the machine code for the **`BL.W WaitForInterrupt`** function call. The **`F7`** means BL, but what does the number **0xFFF86A** mean? If you were to relocate these functions, does this object code need patching (changing) in order for the function call to operate properly?

0xFFF86A is the relative distance from 0x00001262 to 0x00000336. In particular, **0xFFF86A** is equal to (0x00000336-0x00001262)/2. Since these are PC relative, no patching is required

(5) Part b) If you were to relocate these functions (without recompiling), you would move all the above machine code as one block. Would you have to make any patching (changing) in order for the access to I/O port PB2 to operate correctly?
No patching needed. In this case the I/O address is fixed and not relative to the PC. However, since the absolute address of PB2 is stored in the DCW 0x5000, DCW 0x4000 (little endian), the I/O port access will operate without needing any patching.

(5) Part c) After relocation (without recompiling), **`IdleCount`** is now at address 0x20004560, how would you patch this machine code?
The absolute address of IdleCount is stored in the DCW 0x00CC, DCW 0x2000. This will need patching to allow access to the new location, change 0x00CC to 0x4560

(50) Question 4. In this question you will implement a very simple file system. You will implement this file system in the 128k internal ROM of your microcontroller. ROM addresses 0 to 0x0001.FFFF will contain programs and other constant data. However, locations 0x0002.0000 to 0x0003.FFFF will contain the disk. The block size of this disk is fixed at 1024 bytes. This means there are 128 blocks: 0x0002.0000, 0x0002.0400, 0x0002.0800,… 0x0003.FC00. You are given two functions to implement the low-level disk operation. The first function given to you will erase a 1024-byte block in ROM. The **addr** parameter must be one of these 0x0002.0000, 0x0002.0400, 0x0002.0800,… 0x0003.FC00 addresses. The return parameter is 0 if successful (you can ignore errors).

```
int Flash_Erase(uint32_t addr);
```

The second function given to you will program a 1024-byte block in ROM. The source parameter is a pointer to a 1024-byte RAM buffer containing the data to be written. The **addr** parameter must be one of these 0x0002.0000, 0x0002.0400, 0x0002.0800,… 0x0003.FC00 addresses. The return parameter is 0 if successful (you can ignore errors).

```
int Flash_Write(uint8_t *source, uint32_t addr);
```

At initialization the entire disk is erased, filled with 0xFF, and you will consider this state as formatted. Initially, of course, there are no files on the disk. Each file has exactly 1024 bytes of data. This file system does not have file names, rather files are identified by a number. Your system should support up to 127 files. The files are numbered from 1 to 127. You will use the **First** block for directory/free space management. File number **n** (1 to 127) will be in the block starting at

```
0x00020000+1024*n
```

You can create C a pointer into the disk. Let **n** be any block 0 to 127. First define a byte pointer,

```
uint8_t *block;
```

Second, set the byte pointer,

```
block = (uint8_t *)(0x00020000+1024*n);
```

Third, you can read the disk using indexed syntax

```
data = block[i];  // read byte i of block n
```

After each file operation all information must be placed onto the disk. However, during execution of your OS commands, you may use this RAM buffer for temporary storage,

```
uint8_t Buffer[1024];
```

(10) Part a) Implement a helper function that reads 1024 bytes of the disk into RAM. Let **n** be the block number (1 to 127) and **buf** be a RAM array into which the data are read.

```
void ReadBlock(uint32_t n, uint8_t buf[1024]){
uint32_t i;
uint8_t *block;
  block = (uint8_t *)0x00020000+1024*n;
  for(i=0;i<1024;i++){
    buf[i] = block[i];
  }
}
```

(25) Part b) Implement the file write function. This function will allocate space for a new file, store the 1024 bytes of data on the disk, update the directory onto the disk, and return the file number of the new file. If the disk is full return -1, otherwise this function returns 1 to 127. Use the first block to hold the directory and free space management.

```
int OS_FileWrite(uint8_t data[1024]){
uint32_t free=1;
// directory/free space is bytes 1 to 127, FF means free
  ReadBlock(0, Buffer);
  while(free < 128){          // find free block
    if(Buffer[free] == 0xff){ // free?
      Flash_Write(data, 0x00020000+1024*free); // data to disk
      Buffer[free] = 1;       // used
  //    Flash_Erase(0x00020000); // not needed
      Flash_Write(Buffer, 0x00020000); // update directory
      return free;
    }
    free++;  // try next block
  }
  return -1; // full
}
```

(15) Part c) Implement to file erase function. This function will erase an existing file, updating the directory on the disk. **n** is the file number to erase. Return 0 (success) if the file used to exist and now it is erased. Return -1 if the file did not exist.

```
int OS_FileErase(uint32_t n){
  if((n<1)||(n>127)) return -1;
  ReadBlock(0, Buffer);           // load directory
  if(Buffer[n] != 1) return -1;   // does not exist
  Flash_Erase(0x00020000);
  Buffer[n] = 0xFF;
  Flash_Write(Buffer, 0x00020000); // update directory
  Flash_Erase(0x00020000+1024*n);
  return 0;
}
```

Extra, unused question

(10) Question xx. Consider a file system that manages a 16 Megabyte (2^{24} bytes) EEPROM storage for a battery-powered embedded system. The block size is fixed at 2^{14} bytes. In other words, the block size is 16,384 bytes. The microcontroller can perform a 2^{14} byte block-write operation in 1 ms. You are not allowed to split one block between two files. 16 bytes of each block are used by the file system to manage pointers, type, size, and free space. These 16 bytes is not considered internal fragmentation. There is one file in this system with 25,000 bytes of data, and another file with 50,000 bytes of data. No other files exist. **What is the total number of bytes of internal fragmentation?** Show your work.

Each block can store up to 16,384-16=16,368 bytes
Little one needs 2 blocks. 25,000=16,368+8,632. 16,368-8,632=7,736 bytes of internal frag
Big one needs 4 blocks. 50,000=3*16,368+896. 16,368-896= 15,472 bytes of internal frag
Total is 7,736 + 15,472 = 23,208 bytes