Final Exam Solutions

Date: December 19, 2018

Printed Name: _____ _____
Last, First

Your signature is your promise that you have not cheated and will not cheat on this exam, nor will you help others to cheat on this exam. *You will not reveal the contents of this exam to others who are taking the makeup thereby giving them an undue advantage*:

Signature: _____

Instructions:

- ***Write your UT EID on all pages (at the top) and circle your instructor's name at the bottom.***
- Closed book and closed notes. No books, no papers, no data sheets (other than the last four pages of this Exam)
- No devices other than pencil, pen, eraser (no calculators, no electronic devices), please turn cell phones off.
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided. *Anything outside the boxes will be ignored in grading*.
- You have 180 minutes, so allocate your time accordingly.
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.
- Unless otherwise stated, make all I/O accesses friendly.
- *Please read the entire exam before starting.* ***See supplement pages for Device I/O registers.***

Problem 1	10	
Problem 2	5	
Problem 3	10	
Problem 4	5	
Problem 5	15	
Problem 6	10	
Problem 7	5	
Problem 8	10	
Problem 9	10	
Problem 10	20	
Total	100	

(10) Problem 1. Give a one to three word answer for each question.

(1) Part a) What data structure do you use to stream data from an ISR to the main program given the situation where data arrives into the ISR bursts but is processed one byte at a time in the main? ……

FIFO

(1) Part b) With UART transmission we send one start bit, 8 data bits and one stop bit. What term do we use to define these 10 bits? ……

Frame

(1) Part c) What qualifier do we add to an otherwise local variable (scope within a function) so that the variable is defined in permanently in RAM? ……

`static`

(1) Part d) What qualifier do we add to an otherwise global variable so that the scope is restricted to software located within that same file? ……

`static`

(1) Part e) What graphical structure describes the modularity of a system, such that circles and rectangles are modules and arrows represent information passes from one module to another? ……

Data flow graph

(1) Part f) What term is used to describe the smallest difference in input voltage that an ADC can reliably distinguish? ……

Resolution

(1) Part g) What are the units of electrical power? Give as a one-word answer, and not as a combination of other units. ……

watts

(1) Part h) In a real-time system, it is important to respond to critical events. What is the term used to describe the delay between the time a critical event occurs and the time the event is processed? For example, the time between touching a switch and the time the software recognizes the switch is touched. ……

Latency

(1) Part i) A number system where the value 4.125 is represented with the integer 264? ……

Binary fixed point

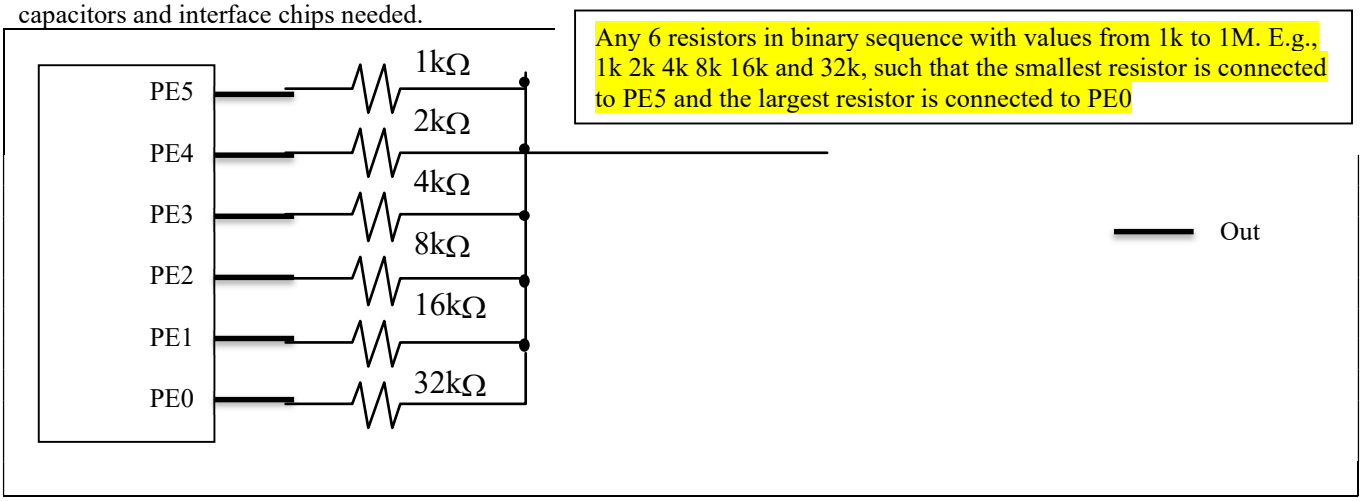
(1) Part j) What is the debugging term used to store important information into arrays? This debugging technique can be used to replace printing (printf) information while the program is running. ……

Dump

(5) Problem 2. Consider the sampling rate chosen for the ADC in Lab 8. Give the relationship for the slowest possible sampling rate (f_s, in Hz), given these parameters: ADC range (V, in volts), number of ADC bits (n, in bits, e.g., 12 bits) and rate at which one moves the slide pot (r, in oscillations per sec).

Nyquist $f_s > 2r$

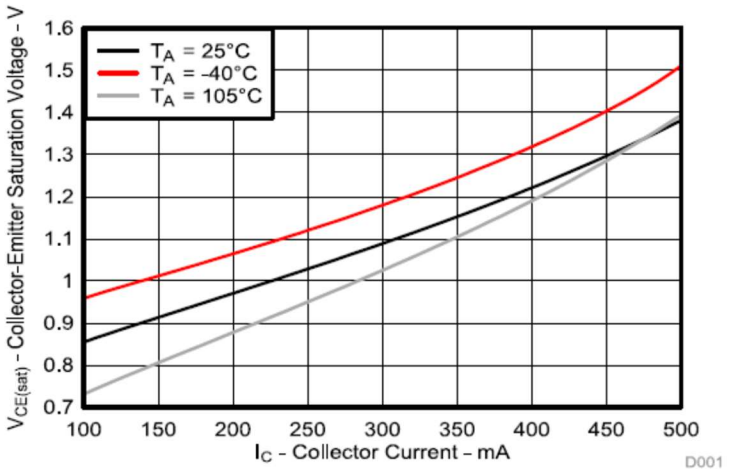
(10) Problem 3: Design a 6-bit DAC connected to Port E using PE5 to PE0. Show the circuit and label all resistors, capacitors and interface chips needed.

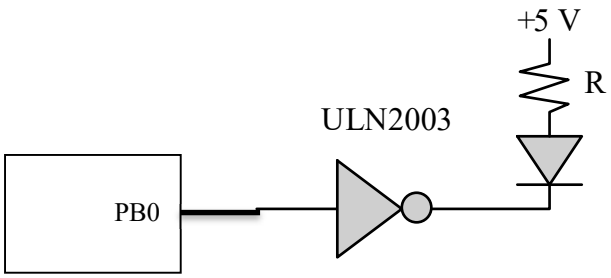


Any 6 resistors in binary sequence with values from 1k to 1M. E.g., 1k 2k 4k 8k 16k and 32k, such that the smallest resistor is connected to PE5 and the largest resistor is connected to PE0

(5) Problem 4. Interface an LED to Port B bit 0 using positive logic. The desired operating point of the LED is 3V and 100 mA. Assume the ULN2003B has an output voltage that depends on the collector current according to this graph at 25 C. Show the circuit and label all resistors, capacitors and interface chips needed.

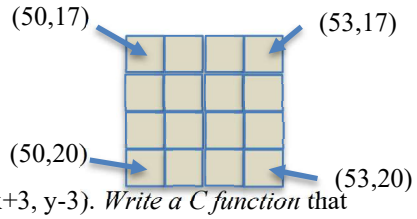
Same as lab 3, except the resistor value.
Must use 5V to power LED
V_{CE} = 0.85V at 100 mA
R = (5-3-0.85V)/100mA
= 1.15V/100mA = 11.5 ohms





(15) Problem 5. Consider a game that has up to 50 boxes. There is an array specifying the current status of each box. Each box is 4 by 4 pixels, and has an (x,y) coordinate, a velocity, a direction, and a life parameter. You may assume the `Box` array has been populated with data. The figure on the right shows one example box at (x,y)=(50,20)

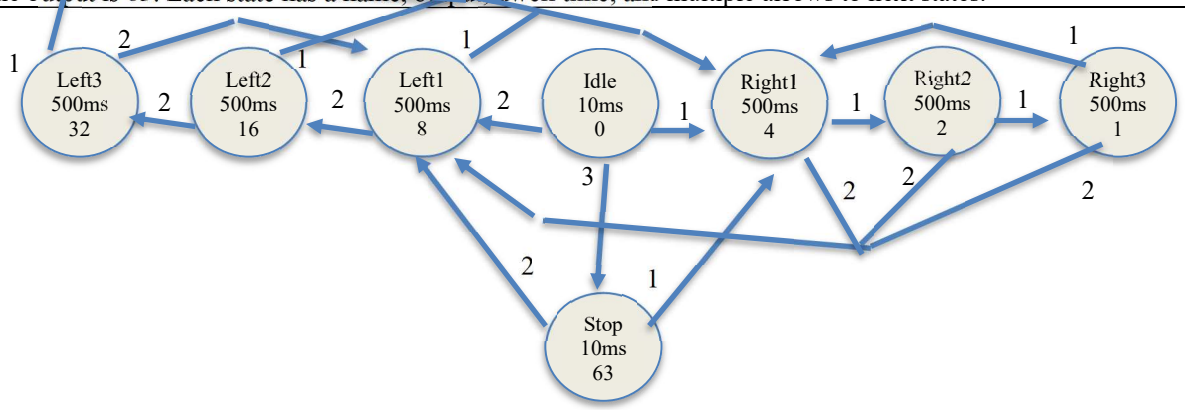
```
typedef enum {dead,alive} status_t;
struct box {
  int16_t x;         // x coordinate, in pixels
  int16_t y;         // y coordinate, in pixels
  int16_t velocity; // velocity, in pixels/frame
  int16_t angle;     // direction, in degrees
  status_t life;};  // dead or alive
typedef box box_t;
```



Each box has 16 pixels in the game world, occupying the square space from (x,y) to (x+3, y-3). *Write a C function* that searches to see if two alive boxes are overlapping (the location of any of the 16 pixels of one box is equal to any of the 16 pixels of another box). If two boxes are occupying the same space, set the life parameter of both boxes to 0. Do not worry about 3 or more boxes occupying the same space.

```
void Search(box_t box[]){ // 50 elements
 int i,j;
  int16_t dx; // x distance between boxes
  int16_t dy; // y distance between boxed
  for(i=0;i<49;i++){
    if((box[i].life == alive){
      for(j=i+1;j<50;j++){
        if(box[j].life == alive){
          dx = box[i].x - box[j].x;
          dy = box[i].y - box[j].y;  // calculate distances
          if((dx>-4)&&(dx<+4)&&(dy>-4)&&(dy<+4)){
            box[i].life = dead;
            box[j].life = dead;
          }
        }
      }
    }
  }
}
```

(10) Problem 6. Draw the state transition graph for a Moore FSM used to control 6 tail lights on a car. There are two inputs and 6 outputs. If the input is 0 the output is 0. If the input is 1 (turn right), the output cycles through the values 4 2 1 every ½ second. If the input is 2 (turn left), the output cycles through the values 8,16,32 every ½ second. If the input is 3 (brake) the output is 63. Each state has a name, output, dwell time, and multiple arrows to next states.



All 0 inputs go to Idle, all 3 inputs go to stop, each state must have 4 arrows

(5) Problem 7. Assume the UART0 has been initialized. Use busy-wait synchronization to implement a C function with the following steps

1) Wait for new serial port input
2) Read the new 8-bit ASCII character data
3) Echo the data by transmitting the same 8-bit data just received
4) Return by value the one character received.

Show what you would place in the .h file

```
//------------UART_InCharEcho------------
// Wait for new serial port input
// Echo received data to transmitter
// Input: none
// Output: ASCII code for character just received
char UART_InCharEcho(void);
```

Show what you would place in the .c file

```
char UART_InCharEcho(void){ char data;
  while((UART0_FR_R&0x00000010) != 0); // UART Receive FIFO Empty
  data = UART0_DR_R;
  while((UART0_FR_R&0x00000020) != 0); // UART Transmit FIFO Full (optional)
  UART0_DR_R = data;
  return data;
}
```

(10) Question 8. The subroutine **mySub** uses a call by value parameter passed on the stack. There are no return parameters. Call by value means the data itself is pushed on the stack. This is not AAPCS compliant. A typical calling sequence is

```
      AREA    |.text|, CODE, READONLY, ALIGN=2
stuff DCD  123          ;32-bit constant
start LDR  R0,=stuff
      LDR  R0,[R0]
      PUSH {R0}         ;the value of the input parameter is pushed
      MOV  R0,#0        ;no cheating, parameter not in R0, on stack
      BL   mySub
      ADD  SP,SP,#4     ;discard parameter
```

The subroutine allocates one 32-bit local variable, **i**, and uses SP stack pointer addressing to access the local variable and the parameter. The binding for these two are

```
in   EQU  20            ;32-bit value that is the input parameter
i    EQU  0             ;32-bit local variable
mySub PUSH {R9,R10,R11,LR}
      SUB SP,SP,#4      ;allocate i
;---------start of body--------------------
    LDR  R11,[SP,#in]   ;Reg R11 is the input parameter data
    STR  R11,[SP,#i]    ;save parameter into local i
;---------end of body----------------------
      ADD SP,SP,#4      ;deallocate i
    POP  {R9,R10,R11,PC}
```

```
SP-> i        0
     R9       4
     R10      8
     R11     12
     LR      16
     123=in  20
```

In the boxes provided, show the binding for **in**, the binding for the local variable **i**, the assembly instruction(s) to allocate **i**, and the assembly instructions to deallocate **i**.

(10) Question 9: Write C code to maintain the elapsed time in minutes. I.e., increment the global variable **Time** once a minute. Include both the initialization and the ISR. Do not worry about priority. Assume bus clock is 16 MHz.

```
uint32_t Time;
void SysTick_Init(void){
  NVIC_ST_CTRL_R = 0;          // disable SysTick during setup (optional)
  NVIC_ST_RELOAD_R = 159999;   // reload value for 10 ms
  NVIC_ST_CURRENT_R = 0;       // any write to current clears it (optional)
  NVIC_ST_CTRL_R = 0x07; // enable SysTick with core clock and interrupts
  Time = 0;  // optional
  EnableInterrupts();

}
void SysTick_Handler(void){ uint32_t static count=0;
  count++;
  if(count == 100*60){ // once a minute
    count=0;
    Time++;
  }
}
```

Many answers here. Consider the line
if(count == N){
The product of N*(RELOAD+1)/16MHz should be 1 minute

(20) Question 10: You are asked to implement a simple postfix calculator as a subroutine in assembly language. The input (call by reference in R0) to your function is a null-terminated character string with the following 12 valid characters 0,1,2,3,4,5,6,7,8,9,+,and*. You may assume all strings are valid and calculate exactly one 32-bit output value. For example

`"5"` returns 5

`"79+"` returns 7+9 = 16

`"58*1+"` returns (5*8)+1 = 41

`"92*7+4*52+*"` returns (((9*2)+7)*4)*(5+2)) = ((18+7)*4)*7) = ((25*4)*7) = (100*7) =700

The basic idea is to fetch a character from the string:

- if it is a + or * operator, pop two numbers from the stack, unsigned 32-bit operate, and push the result
- if it is a digit, convert ASCII to value, push the value (0 to 9) of the digit as a 32-bit value onto the stack
- if it is the null termination, pop one 32-bit value from the stack and return that value in R0

```
;input: R0 points to the string to process
;output: R0 contains the value determined by the calculator
Calc
     LDRB R1,[R0]  ; characters are one byte; could have used signed LDRSB
     ADD  R0,R0,#1 ; next
     CMP  R1,#0
     BEQ  done
     CMP  R1,#'+'
     BNE  nPls
plus POP  {R2,R3}
     ADD  R2,R3 ;add top two elements
     PUSH {R2}
     B    Calc
nPls CMP  R1,#'*'
     BNE  nMul
mult POP  {R2,R3}
     MUL  R2,R3 ;multiply top two elements
     PUSH {R2}
     B    Calc
nMul SUB  R1,#'0'
     PUSH {R1}  ; push digit 0 to 9
     B    Calc
done POP  {R0}
     BX   LR
```

Memory access instructions

```
LDR    Rd, [Rn]        ; load 32-bit number at [Rn] to Rd
LDR    Rd, [Rn,#off]   ; load 32-bit number at [Rn+off] to Rd
LDR    Rd, =value      ; set Rd equal to any 32-bit value (PC rel)
LDRH   Rd, [Rn]        ; load unsigned 16-bit at [Rn] to Rd
LDRH   Rd, [Rn,#off]   ; load unsigned 16-bit at [Rn+off] to Rd
LDRSH  Rd, [Rn]        ; load signed 16-bit at [Rn] to Rd
LDRSH  Rd, [Rn,#off]   ; load signed 16-bit at [Rn+off] to Rd
LDRB   Rd, [Rn]        ; load unsigned 8-bit at [Rn] to Rd
LDRB   Rd, [Rn,#off]   ; load unsigned 8-bit at [Rn+off] to Rd
LDRSB  Rd, [Rn]        ; load signed 8-bit at [Rn] to Rd
LDRSB  Rd, [Rn,#off]   ; load signed 8-bit at [Rn+off] to Rd
STR    Rt, [Rn]        ; store 32-bit Rt to [Rn]
STR    Rt, [Rn,#off]   ; store 32-bit Rt to [Rn+off]
STRH   Rt, [Rn]        ; store least sig. 16-bit Rt to [Rn]
STRH   Rt, [Rn,#off]   ; store least sig. 16-bit Rt to [Rn+off]
STRB   Rt, [Rn]        ; store least sig. 8-bit Rt to [Rn]
STRB   Rt, [Rn,#off]   ; store least sig. 8-bit Rt to [Rn+off]
PUSH   {Rt}            ; push 32-bit Rt onto stack
POP    {Rd}            ; pop 32-bit number from stack into Rd
ADR    Rd, label       ; set Rd equal to the address at label
MOV{S} Rd, <op2>       ; set Rd equal to op2
MOV    Rd, #im16       ; set Rd equal to im16, im16 is 0 to 65535
MVN{S} Rd, <op2>       ; set Rd equal to -op2
```

Branch instructions

```
B    label   ; branch to label    Always
BEQ  label   ; branch if Z == 1   Equal
BNE  label   ; branch if Z == 0   Not equal
BCS  label   ; branch if C == 1   Higher or same, unsigned ≥
BHS  label   ; branch if C == 1   Higher or same, unsigned ≥
BCC  label   ; branch if C == 0   Lower, unsigned <
BLO  label   ; branch if C == 0   Lower, unsigned <
BMI  label   ; branch if N == 1   Negative
BPL  label   ; branch if N == 0   Positive or zero
BVS  label   ; branch if V == 1   Overflow
BVC  label   ; branch if V == 0   No overflow
BHI  label   ; branch if C==1 and Z==0  Higher, unsigned >
BLS  label   ; branch if C==0 or  Z==1  Lower or same, unsigned ≤
BGE  label   ; branch if N == V   Greater than or equal, signed ≥
BLT  label   ; branch if N != V   Less than, signed <
BGT  label   ; branch if Z==0 and N==V  Greater than, signed >
BLE  label   ; branch if Z==1 or N!=V  Less than or equal, signed ≤
BX   Rm      ; branch indirect to location specified by Rm
BL   label   ; branch to subroutine at label
BLX  Rm      ; branch to subroutine indirect specified by Rm
```

Interrupt instructions

```
CPSIE  I               ; enable interrupts  (I=0)
CPSID  I               ; disable interrupts (I=1)
```

Logical instructions

```
AND{S} {Rd,} Rn, <op2> ; Rd=Rn&op2    (op2 is 32 bits)
ORR{S} {Rd,} Rn, <op2> ; Rd=Rn|op2    (op2 is 32 bits)
EOR{S} {Rd,} Rn, <op2> ; Rd=Rn^op2    (op2 is 32 bits)
BIC{S} {Rd,} Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)
ORN{S} {Rd,} Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)
LSR{S} Rd, Rm, Rs      ; logical shift right Rd=Rm>>Rs  (unsigned)
LSR{S} Rd, Rm, #n      ; logical shift right Rd=Rm>>n   (unsigned)
ASR{S} Rd, Rm, Rs      ; arithmetic shift right Rd=Rm>>Rs (signed)
```

```
ASR{S} Rd, Rm, #n      ; arithmetic shift right Rd=Rm>>n  (signed)
LSL{S} Rd, Rm, Rs      ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n      ; shift left Rd=Rm<<n  (signed, unsigned)
```

Arithmetic instructions

```
ADD{S} {Rd,} Rn, <op2> ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12 ; Rd = Rn + im12, im12 is 0 to 4095
SUB{S} {Rd,} Rn, <op2> ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12 ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2> ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12 ; Rd = im12 - Rn
CMP    Rn, <op2>       ; Rn - op2      sets the NZVC bits
CMN    Rn, <op2>       ; Rn - (-op2)   sets the NZVC bits
MUL{S} {Rd,} Rn, Rm    ; Rd = Rn * Rm       signed or unsigned
MLA    Rd, Rn, Rm, Ra  ; Rd = Ra + Rn*Rm    signed or unsigned
MLS    Rd, Rn, Rm, Ra  ; Rd = Ra - Rn*Rm    signed or unsigned
UDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         unsigned
SDIV   {Rd,} Rn, Rm    ; Rd = Rn/Rm         signed
```

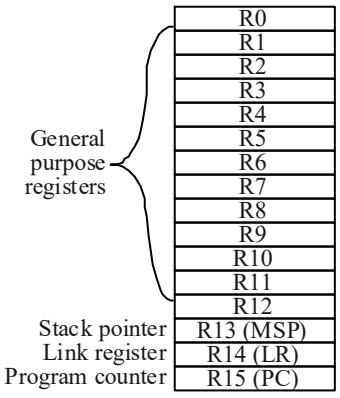
Notes `Ra Rd Rm Rn Rt represent 32-bit registers`

```
value   any 32-bit value: signed, unsigned, or address
{S}     if S is present, instruction will set condition codes
#im12   any value from 0 to 4095
#im16   any value from 0 to 65535
{Rd,}   if Rd is present Rd is destination, otherwise Rn
#n      any value from 0 to 31
#off    any value from -255 to 4095
label   any address within the ROM of the microcontroller
op2     the value generated by <op2>
```

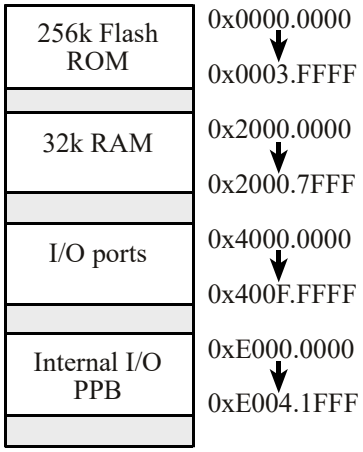
Examples of flexible operand **<op2>** creating the 32-bit number. E.g., **Rd = Rn+op2**

```
ADD Rd, Rn, Rm          ; op2 = Rm
ADD Rd, Rn, Rm, LSL #n  ; op2 = Rm<<n  Rm is signed, unsigned
ADD Rd, Rn, Rm, LSR #n  ; op2 = Rm>>n  Rm is unsigned
ADD Rd, Rn, Rm, ASR #n  ; op2 = Rm>>n  Rm is signed
ADD Rd, Rn, #constant   ; op2 = constant, where X and Y are hexadecimal digits:
```

- produced by shifting an 8-bit unsigned value left by any number of bits
- in the form **0x00XY00XY**
- in the form **0xXY00XY00**
- in the form **0xXYXYXYXY**



Condition code bits
N negative
Z zero
V signed overflow
C carry or unsigned overflow




```
DCB   1,2,3 ; allocates three 8-bit byte(s)
DCW   1,2,3 ; allocates three 16-bit halfwords
DCD   1,2,3 ; allocates three 32-bit words
SPACE 4     ; reserves 4 bytes
```

Address	7	6	5	4	3	2	1	0	Name
$400F.E608			GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA	SYSCTL_RCGCGPIO_R
$4000.53FC	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA	GPIO_PORTB_DATA_R
$4000.5400	DIR	DIR	DIR	DIR	DIR	DIR	DIR	DIR	GPIO_PORTB_DIR_R
$4000.5420	SEL	SEL	SEL	SEL	SEL	SEL	SEL	SEL	GPIO_PORTB_AFSEL_R
$4000.551C	DEN	DEN	DEN	DEN	DEN	DEN	DEN	DEN	GPIO_PORTB_DEN_R

Table 4.5. TM4C123 Port B parallel ports. Each register is 32 bits wide. Bits 31 – 8 are zero.

Address	31	30	29-7	6	5	4	3	2	1	0	Name
0xE000E100		F	…	UART1	UART0	E	D	C	B	A	NVIC_EN0_R

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

Address	31-29	28-24	23-21	20-8	7-5	4-0	Name
$E000ED20	SYSTICK	0	PENDSV	0	DEBUG	0	NVIC_SYS_PRI3_R

Table 9.6. SysTick registers.

Table 9.6 shows the SysTick registers used to create a periodic interrupt. SysTick has a 24-bit counter that decrements at the bus clock frequency. Let f_{BUS} be the frequency of the bus clock, and let n be the value of the **RELOAD** register. The frequency of the periodic interrupt will be $f_{BUS}/(n+1)$. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD** register. Third, we write to the **`NVIC_ST_CURRENT_R`** value to clear the counter. Lastly, we write the desired mode to the control register, **`NVIC_ST_CTRL_R`**. To turn on the SysTick, we set the **ENABLE** bit. We must set **CLK_SRC**=1, because **CLK_SRC**=0 external clock mode is not implemented. We set **INTEN** to arm SysTick interrupts. The standard name for the SysTick ISR is **SysTick_Handler**.

Address	31-2	1	0	Name
$400F.E638		ADC1	ADC0	SYSCTL_RCGCADC_R

	31-14	13-12	11-10	9-8	7-6	5-4	3-2	1-0	
$4003.8020		SS3		SS2		SS1		SS0	ADC0_SSPRI_R

	31-16	15-12	11-8	7-4	3-0	
$4003.8014		EM3	EM2	EM1	EM0	ADC0_EMUX_R

	31-4	3	2	1	0	
$4003.8000		ASEN3	ASEN2	ASEN1	ASEN0	ADC0_ACTSS_R
$4003.80A0		MUX0				ADC0_SSMUX3_R
$4003.80A4		TS0	IE0	END0	D0	ADC0_SSCTL3_R
$4003.8028		SS3	SS2	SS1	SS0	ADC0_PSSI_R
$4003.8004		INR3	INR2	INR1	INR0	ADC0_RIS_R
$4003.8008		MASK3	MASK2	MASK1	MASK0	ADC0_IM_R
$4003.8FC4		Speed				ADC0_PC_R

	31-12	11-0	
$4003.80A8		DATA	ADC0_SSFIFO3_R

Table 10.3. The TM4C ADC registers. Each register is 32 bits wide.

Set Speed to 0001 for slow speed operation. The ADC has four sequencers, but we will use only sequencer 3. We set the **ADC_SSPRI_R** register to 0x3210 to make sequencer 3 the lowest priority. Because we are using just one sequencer, we just need to make sure each sequencer has a unique priority. We set bits 15–12 (**EM3**) in the **ADC_EMUX_R** register to specify how the ADC will be triggered. If we specify software start (**EM3**=0x0), then the software writes an 8 (**SS3**) to the **ADC_PSSI_R** to initiate a conversion on sequencer 3. Bit 3 (**INR3**) in the **ADC_RIS_R** register will be set when the conversion is complete. We can enable and disable the sequencers using the **ADC_ACTSS_R** register. Which channel we sample is configured by writing to the **ADC_SSMUX3_R** register. The **ADC_SSCTL3_R** register specifies the mode of the ADC sample. Clear **TS0**. We set **IE0** so that the **INR3** bit is set on ADC conversion, and clear it when no flags are needed. We will set **IE0** for both interrupt and busy-wait synchronization. When using sequencer 3, there is only one sample, so **END0** will always be set, signifying this sample is the end of the sequence. Clear the **D0** bit. The **ADC_RIS_R**

register has flags that are set when the conversion is complete, assuming the **IE0** bit is set. Do not set bits in the **ADC_IM_R** register because we do not want interrupts. Write one to **ADC_ISC_R** to clear the corresponding bit in the **ADC_RIS_R** register.

UART0 pins are on PA1 (transmit) and PA0 (receive). The **UART0_IBRD_R** and **UART0_FBRD_R** registers specify the baud rate. The baud rate **divider** is a 22-bit binary fixed-point value with a resolution of 2^{-6}. The **Baud16** clock is created from the system bus clock, with a frequency of (Bus clock frequency)/**divider**. The baud rate is

Baud rate = **Baud16**/**16** = (Bus clock frequency)/(16***divider**)

We set bit 4 of the **UART0_LCRH_R** to enable the hardware FIFOs. We set both bits 5 and 6 of the **UART0_LCRH_R** to establish an 8-bit data frame. The **RTRIS** is set on a receiver timeout, which is when the receiver FIFO is not empty and no incoming frames have occurred in a 32-bit time period. The arm bits are in the **UART0_IM_R** register. To acknowledge an interrupt (make the trigger flag become zero), software writes a 1 to the corresponding bit in the **UART0_IC_R** register. We set bit 0 of the **UART0_CTL_R** to enable the UART. Writing to **UART0_DR_R** register will output on the UART. This data is placed in a 16-deep transmit hardware FIFO. Data are transmitted first come first serve. Received data are place in a 16-deep receive hardware FIFO. Reading from **UART0_DR_R** register will get one data from the receive hardware FIFO. The status of the two FIFOs can be seen in the **UART0_FR_R** register (FF is FIFO full, FE is FIFO empty). The standard name for the UART0 ISR is **UART0_Handler**. RXIFLSEL specifies the receive FIFO level that causes an interrupt (010 means interrupt on ≥ ½ full, or 7 to 8 characters). TXIFLSEL specifies the transmit FIFO level that causes an interrupt (010 means interrupt on ≤ ½ full, or 9 to 8 characters).

	31–12	11	10	9	8	7–0	Name
$4000.C000		OE	BE	PE	FE	DATA	UART0_DR_R

	31–3	3	2	1	0	
$4000.C004		OE	BE	PE	FE	UART0_RSR_R

	31–8	7	6	5	4	3	2–0	
$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY		UART0_FR_R

	31–16	15–0	
$4000.C024		DIVINT	UART0_IBRD_R

	31–6	5–0	
$4000.C028		DIVFRAC	UART0_FBRD_R

	31–8	7	6 – 5	4	3	2	1	0	
$4000.C02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART0_LCRH_R

	31–10	9	8	7	6–3	2	1	0	
$4000.C030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN	UART0_CTL_R

	31–6	5-3	2-0	
$4000.C034		RXIFLSEL	TXIFLSEL	UART0_IFLS_R

	31-11	10	9	8	7	6	5	4		
$4000.C038		OEIM	BEIM	PEIM	FEIM	RTIM	TXIM	RXIM		UART0_IM_R
$4000.C03C		OERIS	BERIS	PERIS	FERIS	RTRIS	TXRIS	RXRIS		UART0_RIS_R
$4000.C040		OEMIS	BEMIS	PEMIS	FEMIS	RTMIS	TXMIS	RXMIS		UART0_MIS_R
$4000.C044		OEIC	BEIC	PEIC	FEIC	RTIC	TXIC	RXIC		UART0_IC_R

Table 11.2. UART0 registers. Each register is 32 bits wide. Shaded bits are zero.