School of Engineering

COURSEWORK SUBMISSION SHEET

Coursework should be stapled at the top left-hand corner. Please do not use any form of plastic or cardboard binder.

All sections must be completed and the declaration signed for the submission to be accepted. All coursework should be submitted to the coursework boxes located across from N327 Riverside East.
EXTENSIONS – all requests for extensions must be submitted on the appropriate form, prior to the due date.
LATE COURSEWORK – submitted without an Extenuating Circumstances Form will not be processed.

Due Date	Date Submitted	For official use only
By 12 Noon 25th November 2016	**24/11/17**	

STUDENT No. 1705593	
SURNAME Stoddart	
FIRST NAME(S) Christopher Thomas	
COURSE & STAGE	FullTime/PartTime Stage 3 MEng/BEng Electronic & Electrical Engineering BSc (Eng) Electronic & Electrical Engineering
MODULE NUMBER & TITLE	**EN3540 Computer Architecture**
ASSIGNMENT TITLE	**Assessment 1 – software exercises**
LECTURER ISSUING COURSEWORK	**Graeme Dunbar**

I confirm: (a) That the work undertaken for this assignment is entirely my own and that I have not made use of any unauthorised assistance.*
(b) That the sources of all reference material have been properly acknowledged.*
(c) That I accept that the School will dispose of uncollected coursework at the end of term via the paper recycling service.
(d) That I will retain a copy of all coursework until the end of my studies.

*** For information on Academic Misconduct, refer to - http://www.rgu.ac.uk/about/academic-affairs/quality-assurance-and-regulations/academic-regulations/academic-regulations**

Signed ……Christopher Stoddart…..... Date .....24/11/2017.....

Marker's Comments	
Marker **Grant Maxwell**	Grade

EN3540 Computer Architecture

Assessment 1 – software exercises

Coursework Information for Programming Routines

1705593

BENG (HONS) ELECTRONIC AND ELECTRICAL ENGINEERING

November 201

Contents

Exercise 1: Timers and Interrupts

Square Wave Frequency set up

The frequency of the square is set by the DEVAL value, which is calculated by using the required frequency (440 Hz) with a 1:1 spacing ratio. Thus the high and low sections of the wave form are calculated to be: $1.136 \times 10^{-6} us$ long.

$$\left[\left(\frac{1}{440}\right)\Big/_{2} = 1.136 \times 10^{-6}\right]$$

The frequency of the clock (24 MHz) can be divided by the pre-scaler value (32). The period of the input signal can now be calculated by:

$$32/(24 \times 10^{-6}) = 1.333us$$

Now the count can to be calculated and loaded to the counter as:

$$\frac{1.136\times10^{-3}}{1.333\times10^{-6}} = 852.21 \ .$$

The Timer System Control Register 1 (TSCR1), enables the timing register by changing bit 7 to a value of 1.

Creating an Interrupt

Step 1 Declare interrupt service routine

If an interrupt is to be created in the ImageCraft C cross-compiler using C language the following code is necessary to handle the interrupt:

```
#pragma interrupt_handler TimerCh2_ISR

void TimerCh2_ISR( void )
{
   TC2 += DELVAL;
   TFLG1 |= 0x04;
}
```

The code must be declared by using #pragma. This can be declared as a standard function however it is limited in the sense that it cannot handle parameters and therefore not able to return as a result.

A key benefit to this is the processer is only required to handle the timer only when a interrupt occurs.

The Tc2+= is set to DELVAL (its value is declared outside the main loop) is used to calculate the next target count.

TFLG1 is set to 0x04 to allow the OC2 flag to be cleared.

Step 2 Enable interrupts on timer channel 2

The interrupt is enabled on channel 2 by using the following code:

```
   void Timer_Init(void)
{
   TIE = 0x04;
   TSCR2 = 0x05;
   TCTL2 = 0x10;
   TIOS = 0x04;
   TC3 = DELVAL;
   TSCR1 = 0x80;
}
```

The TIE register used to enable or disable the interrupts across all the channels. In this case the code is set to 0x04, to enable channel 2 while disabling all the rest.

TSCR2 controls the number of stages that are required to divide the stages between the bus clock and the main clock timer. PR2, PR1, PR0 are used to define that value. In this case 0x05.

PR2	PR1	PR0	Prescale Factor
0	0	0	1
0	0	1	2
1	0	1	32

The TCTL2 register control bits are used to select the appropriate output when it results in a matching Ocx compare. In this example Oc2 is toggled as a result by setting the control bit at 0x10.

THE TIOS or the Timer Input Capture/Output Compare Select Register bits are used to define what channel is selected to use as the output compare.

Writing a 0 at the appropriate bit makes the corresponding channel acts as an input capture.
Writing a 1 at the appropriate bit makes the corresponding channel acts as an output compare.

0x04 is written to the register as a result.

TC3 is a latching register used to capture the values of the input registers (TC0 – TC3). This is set to the DELVAL value as previously seen in the TC2 Register and is the initial compare value.

TSCR1 simply enables the timer.

Step 3 Enable interrupts on the microcontroller

Enabling the interrupt on the microcontroller requires the following code:

```
#include "DBug12New.h"

void Timer_Init(void);
void TimerCh2_ISR(void);

int main(void)
{
   VectTimerCh2 = (unsigned)TimerCh2_ISR;
   Timer_Init();
   asm("cli");
}
```

The link to the interrupt vectors is declared in the address space. The interrupts location is designated by a pair of bytes that contain the address of the interrupt handling routine. Using the "DBug12New.h" file allows the vector addresses to be stored in specific RAM addresses, the interrupt upon activation, makes the monitor code jump indirectly to the specified RAM addresses located in the file.

VectTimerCh2 is the address in RAM at which the address of the Interrupt Service Routine should be stored. In this example, TimerCh2_ISR is the name of the Interrupt Service Routine.

The asm("cli"); statement enables the maskable interrupt.

Output Compare

To display a message independently while servicing the interrupt, the following code is used;

```
{
While (1)                          /*while loop*/
       {
```

However as the message can only display on the falling edge of bit 7 of the switch board connected to port H. A compare value therefore must be set to prevent the message from displaying at incorrect mode as follows;

```
       if
       ( ((PTH & 0x80) == 0) &&  ((Prevval & 0x80) == 0x80))
       {
       puts("Christopher Stoddart\r");
       }
              Prevval = PTH;                  /*remembers switch value*/
       }
   return 0;
}
```

Now the program will only display the name on the screen when the if loop value and the falling edge bit value are equal.

Exercise 2: Linking C and Assembly Language

Stack Diagram

C function:	`int LocatePosition( unsigned, unsigned, char )`		
	D Register	Data Type	Description
Parameter	unsigned	Unsigned int	1st parameter
Return value	result	int	Calculates memory location

Memory Address	Stack Contents	Offset from SP	Description
			Low Memory ↑
0x3BF6	D: hi	SP+0	(TOS) Save X register
0x3BF7	D: lo	SP+1	
0x3BF8	return addr: hi	SP+2	
0x3BF9	return addr: lo	SP+3	
0x3BFB	Unsigned: hi	SP+4	2nd parameter
0x3BFC	Unsigned: lo	SP+5	
0x3BFA	0x00	SP+6	Set to 0
0x3BFF	char	SP+7	3rd parameter (byte)
0x3C00			Stack Base (empty stack)
			High Memory ↓

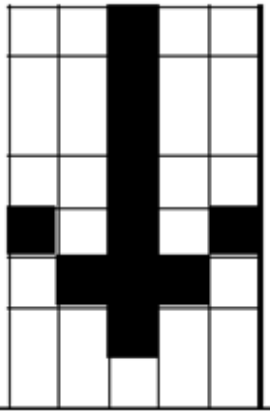
Exercise 3: LCD Character Generator

To create a character that will display on the LCD the following steps are required

1. To generate a character that will be displayed on an LCD, a subroutine that is able to handle this must first be created. The subroutine (lcd_make()) was created for such a purpose. The subroutine works by utilizing two separate parameters. The first is the character number within the ASCII code (0..7). The secondary parameter incorporates a matrix of bytes that are used to create the character.
2. The subroutine should now write the required characters to the CG RAM Addresses from 0x40-0x47.
3. A call to WriteCmd is now utilised to write the eight pixel values to the LCD.

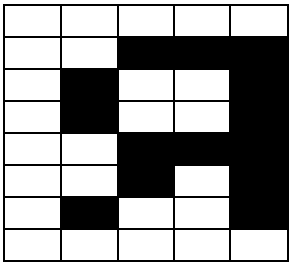
For example the Following generated character values (0x04, 0x04, 0x04, 0x04, 0x15, 0x0e, 0x004, 0x00) would be stored at consecutive incrementing addresses in the CG RAM.

Set CG RAM Address									Data									
Adresse								Hex	Bit								Hex	
									7	6	5	4	3	2	1	0		
0	1	0	0	0	0	0	0	$40				0	0	1	0	0	$04	
					0	0	1	$41				0	0	1	0	0	$04	
					0	1	0	$42				0	0	1	0	0	$04	
					0	1	1	$43				0	0	1	0	0	$04	
					1	0	0	$44	X	X	X	1	0	1	0	1	$15	
					1	0	1	$45				0	1	1	1	0	$0E	
					1	1	0	$46				0	0	1	0	0	$04	
					1	1	1	$47				0	0	0	0	0	$00	



Bit Mapping of the Cyrillic (Russian) character symbol “Я”

Set CG RAM Address							
Address							Hex
0	1	0	0	0	0	0	$40
				0	0	1	$41
				0	1	0	$42
				0	1	1	$43
				1	0	0	$44
				1	0	1	$45
				1	1	0	$46
				1	1	1	$47

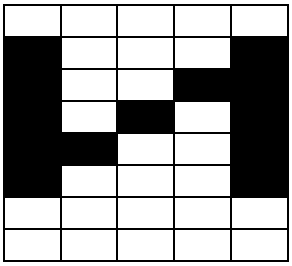


Data					
Bit					Hex
0	0	0	0	0	$00
0	0	1	1	1	$07
0	1	0	0	1	$09
0	1	0	0	1	$09
0	0	1	1	1	$07
0	0	1	0	1	$05
0	1	0	0	1	$09
0	0	0	0	0	$00

The following array was produced { 0x00,0x07,0x09,0x09,0x07,0x05,0x09,0x00 }

Bit Mapping of the Cyrillic (Russian) character symbol “i”

Set CG RAM Address							
Address							Hex
0	1	0	0	0	0	0	$40
				0	0	1	$41
				0	1	0	$42
				0	1	1	$43
				1	0	0	$44
				1	0	1	$45
				1	1	0	$46
				1	1	1	$47



Data					
Bit					Hex
0	0	0	0	0	$00
1	0	0	0	1	$11
1	0	0	1	1	$13
1	0	1	0	1	$15
1	1	0	0	1	$19
1	0	0	0	1	$11
0	0	0	0	0	$00
0	0	0	0	0	$00

The following array was produced {0x00,0x11,0x13,0x15,0x19,0x11,0x00,0x00 }

Exercise 4: ADC

Potentiometer Readings From the Variable Clock Frequency

0x6100	Potentiometer Positions				
Channels	Minimum	Low-Mid	Medium	Med-High	Maximum
0	40	3300	8B80	E000	FEC0
1	40	3000	8580	DCC0	FFCO
2	40	2C00	83C0	D980	FFC0
3	40	2E00	83C0	D800	FFC0

0xE110	Potentiometer Positions				
Channels	Minimum	Low-Mid	Medium	Med-High	Maximum
0	80	2F80	8780	DE80	FF80
1	80	2D80	8780	DC80	FF80
2	80	2D80	8680	E080	FF80
3	80	2D80	8B60	DC80	FF80

0x6150	Potentiometer Positions				
Channels	Minimum	Low-Mid	Medium	Med-High	Maximum
0	8000	B000	AC0	5D80	7FC0
1	8000	AC00	4C0	5D40	7FC0
2	8000	AD40	3C0	5E80	7FC0
3	8000	B080	F80	61C0	7FC0

0xE150	Potentiometer Positions				
Channels	Minimum	Low-Mid	Medium	Med-High	Maximum
0	8080	AF80	780	5E80	7F80
1	8080	AC80	280	5E80	7F80
2	8080	AB80	980	6080	7F80
3	8080	AD80	980	6180	7F80

0x6190	Potentiometer Positions				
Channels	Minimum	Low-Mid	Medium	Med-High	Maximum
0	0	C3	223	377	3FF
1	0	A7	1F9	364	3FF
2	0	A7	213	381	3FF
3	0	B6	230	380	3FF

0xE190	Potentiometer Positions				
Channels	Minimum	Low-Mid	Medium	Med-High	Maximum
0	0	30	8E	E3	FF
1	0	2D	85	E3	FF
2	0	2C	85	DD	FF
3	0	31	8E	E4	FF

Setting the ATD1CTL45 register

The changes necessary to the ATD1CTL45 register to produce the various formats is by changing the values of different functions within the 16 bit word.

The ATDCTL4 Register can be broken down into the following :

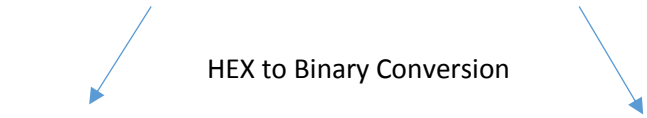
Bit 7 [SRES8] of the register controls the resolution (0 – 10 bit, 1 – 8 bit)
The remaining bits are not used in this exercise but control the sampling time.

The ATDCTL5 Register is made up of the following bit functions:

Bit 7 [DJM] determines if the data is left or right justified (0 – left, 1 - right).
Bit 6 [DSGN] selects if the data is unsigned or signed (0 – unsigned, 1 - signed).
Bit 5 [SCAN] chooses whether a single or continuous scan is used (0 – single, 1 – Continuous).
Bit 4 [Mult] determines how many channels are converted (0 – single, 1 – Multiple).

The following Laboratory example code 0x6100 can be written as;

61	00



ATDCTL4	ATDCLT5
01100001	00000000

Bit 7	0
Bit 6	1
Bit 5	1
Bit 4	0
Bit 3	0
Bit 2	0
Bit 1	0
Bit 0	1

Bit 7	0
Bit 6	0
Bit 5	0
Bit 4	0
Bit 3	0
Bit 2	0
Bit 1	0
Bit 0	0

By changing the 16-bit word values from 0-1 various formats can be created. A simple binary to hex conversion is all that is necessary to get the correct value to be attached to ATD1CTL45 register.

0xE110 8-bit, left justified, unsigned data, 1110 0001 0001 0000
0x6150 10-bit, left justified, signed data, 0110 0001 0101 0000
0xE150 8-bit, left justified, signed data, 1110 0001 0101 0000
0x6190 10-bit, right justified, unsigned data, 0110 0001 1001 0000
0xE190 8-bit, right justified, unsigned data, 1110 0001 1001 0000