First:_____ Last:_____

This is a closed book exam. You must put your answers in the space provided. You have 3 hours, so allocate your time accordingly. ***Please read the entire exam before starting.***

(1) Q1 a)	**(5) Q3**
(1) Q1 b)	
(1) Q1 c)	**(5) Q6**
(1) Q1 d)	
(1) Q1 e)	**(5) Q7**
(1) Q1 f)	
(1) Q1 g)	**(2) Q9**
(1) Q1 h)	
(1) Q1 i)	**(10) Q10**
(1) Q1 j)	
(3) Q2 a)	**(3) Q11 a)**
(2) Q2 b)	**(2) Q11 b)**
(10) Q4	

(5) Q5 a)	**(5) Q5 b)**

(6) Q8 a)

```
const struct stuff{




};
typedef const struct stuff StuffType;
```

(4) Q8 b)

(4) Q8 c)

(4) Q8 d)

(4) Q12 a)

(8) Q12 b)

(8) Q12 c)

Please read and affirm our honor code:
"The core values of The University of Texas at Austin are learning, discovery, freedom, leadership, individual opportunity, and responsibility. Each member of the university is expected to uphold these values through integrity, honesty, trust, fairness, and respect toward peers and community."

(10) Question 1. State the term that is described by each definition.

Part a) The process of converting an unsigned 8-bit integer into an unsigned 16-bit integer.

Part b) You are given a 4-bit DAC to test. The DAC input is stepped from 0 to 15. For each input change, the change in DAC output is measured. The results are processed by averaging all the changes in output.

Part c) The part of the processor that performs: addition, multiplication, and, or, shift.

Part d) A system where the response time from when new input is ready until when the new input is processed is less than 25 μsec.

Part e) Error that can occur as a result of a left shift.

Part f) Error that can occur as a result of a right shift.

Part g) A variable that can only be accessed by one function.

Part h) A function parameter that is a pointer to the data.

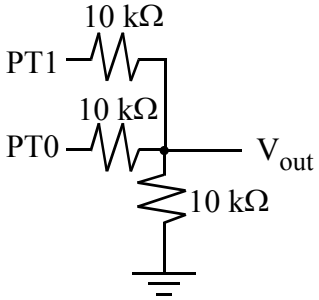
Part i) A characteristic of a debugger when the presence of the collection of information itself makes a large and important effect on the parameters being measured.

Part j) A debugging process that fixes all the inputs to a system, so the systems can be run over an over yielding the same outputs.

(5) Question 2. The system uses an 8-bit ADC and a serial port running at 100 bits/sec. Assume every ADC sample you take must be transmitted over the serial channel.

Part a) How many bytes of information per second are being transferred?

Part b) At this rate, what would the be ADC sampling rate in Hz?



(5) Question 3. What is the output voltage V_{out} when PT1 is high and PT0 is low? Assume V_{OH} is 5V and V_{OL} = 0V.

(10) Question 4. Assume Register X contains the integer portion of an unsigned binary fixed point number with resolution 2^{-4}, and Register Y contains the integer portion of an unsigned binary fixed point number with resolution 2^{-2}. For example, if the first number is 1.5 then Register X equals 24. If the second number is 2.25, then Register Y is 9. Write assembly code that adds the two numbers such that the sum is in Register D with a resolution of 2^{-2}. Since 1.5+2.25 is 2.75, Register D should be returned with 11. No global variables are allowed, but you may use the stack. Handle potential overflow errors by implementing ceiling. Some dropout may occur.

(10) Question 5. There are arrays of 16-bit numbers. The first element of the array is the length and remaining elements are 16-bit signed numbers. For example, here are three such possible arrays.

```
short buf1[5]={4,1000,-1000,0,33};
short buf2[7]={6,-4,100,200,2,0,44};
short buf3[1]={0};
```

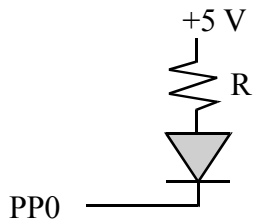
Part a) Write a C function that takes a pointer to an array and returns the difference between the maximum and minimum values. For example

```
Result1 = MaxDiff(buf1);  // should return 2000 = 1000 – (-1000)
Result2 = MaxDiff(buf2);  // should return 204 = 200 – (-4)
Result3 = MaxDiff(buf3);  // should return 0 because array is empty
```

You are not allowed to add global variables. Don't worry about overflow calculating the difference.

Part b) Write an assembly subroutine that performs the same operation. The pointer to the array is passed in Register D, and the result is returned in Register D. You are not allowed to add any global variables. You must use binding to implement local variables.

(5) Question 6. The following interface can be used for low current LEDs. Assume the LED voltage drop is 2 V. The resistor is 1000 Ω. When the software outputs a high, the voltage on PP0 becomes 4.9 V. When the software outputs a low, the voltage on PP0 becomes 0.5 V. What is the LED current when the LED is on?



(5) Question 7. Assume Register B equals $55, Register A equals $F0 and Register X equals $5678. What is the value in Register X after executing these instructions? Give the answer as ??? if the value cannot be determined.

```
        stx  2,-sp
        std  2,sp-
        pulx
        puld
```

(18) Question 8. This question tests your ability to create and use structures.
Part a) Complete the C code that defines a structure containing an array of three 8-bit unsigned numbers, and one 16-bit unsigned number. Call the array **`Position`**, and call the number **`Time`**.

Part b) Use the **`StuffType`** structure to define a ROM-based constant with a **`Position`** of {100,60,50} and a **`Time`** of 1000. Call this constant **`Command`**.

Part c) Write a C code (no function, just code) that accesses the above constant and sets a variable **`max`** to the largest position number of the three. In this case, **`max`** will become 100.

Part d) Write a C function that takes a pointer to a constant and returns the largest position number of the three. One possible way to call your function is

```
max = MaxPosition(&Command);
```

In this case, **`max`** will become 100.

(2) Question 9. You are given two 8-bit numbers, where each number is known to exist between 0 and 100. An 8-bit addition is operated on two numbers. Is it possible for the overflow (V) bit to be set?

(5) Question 10. Assume the E clock is operating at 8 MHz, and TSCR2 = 4. The output compare ISR executes these instructions. What value goes in ????? to make the interrupt frequency 100 Hz?

```
OC6ISR movb #$40,TFLG1
       ldd  TC6
```

```
addd #?????
std  TC6
rti
```

(5) Question 11. Assume the PC contains $4007, and the SP equals $3FF4.

```
$4007 0750    bsr Function
```

Part a) What number is pushed on the stack during the execution of **`bsr`**?

Part b) What is the value in the PC after **`bsr`** is executed?

(20) Question 12. In this problem, your software should output the alphabet 'A' 'B' 'C' … 'Z' over and over using SCI0 serial port. You must use SCI0 interrupts (not output compare). The baud rate is 10000 bits/sec. You may assume the E clock is 8 MHz.

Part a) Show the C code that specifies any global variables you need.

Part b) Write the initialization function in C that sets up the SCI0 interrupts. The main will call this initialization once at the beginning, and then perform unrelated tasks. This function should arm and enable interrupts. No loops are allowed.

Part c) Write the ISR in C that outputs the alphabet using SCI0. No loops are allowed.

`aba` 8-bit add RegA=RegA+RegB
`abx` unsigned add RegX=RegX+RegB
`aby` unsigned add RegY=RegY+RegB
`adca` 8-bit add with carry to RegA
`adcb` 8-bit add with carry to RegB
`adda` 8-bit add to RegA
`addb` 8-bit add to RegB
`addd` 16-bit add to RegD
`anda` 8-bit logical and to RegA
`andb` 8-bit logical and to RegB
`andcc` 8-bit logical and to RegCC
`asl/lsl` 8-bit left shift Memory
`asla/lsla` 8-bit left shift RegA
`aslb/lslb` 8-bit arithmetic left shift RegB
`asld/lsld` 16-bit left shift RegD
`asr` 8-bit arithmetic right shift Memory
`asra` 8-bit arithmetic right shift to RegA
`asrb` 8-bit arithmetic right shift to RegB
`bcc` branch if carry clear
`bclr` bit clear in memory `bclr PTT,#$01`
`bcs` branch if carry set
`beq` branch if result is zero (Z=1)
`bge` branch if signed ≥
`bgnd` enter background debug mode
`bgt` branch if signed >
`bhi` branch if unsigned >
`bhs` branch if unsigned ≥
`bita` 8-bit and with RegA, sets CCR
`bitb` 8-bit and with RegB, sets CCR
`ble` branch if signed ≤
`blo` branch if unsigned <
`bls` branch if unsigned ≤
`blt` branch if signed <
`bmi` branch if result is negative (N=1)
`bne` branch if result is nonzero (Z=0)
`bpl` branch if result is positive (N=0)
`bra` branch always
`brclr` branch if bits are clear `brclr PTT,#$01,loop`
`brn` branch never
`brset` branch if bits are set `brset PTT,#$01,loop`
`bset` bit set in memory `bset PTT,#$04`
`bsr` branch to subroutine
`bvc` branch if overflow clear
`bvs` branch if overflow set
`call` subroutine in expanded memory
`cba` 8-bit compare RegA with RegB, RegA-RegB
`clc` clear carry bit, C=0
`cli` clear I=0, enable interrupts
`clr` 8-bit memory clear
`clra` RegA clear
`clrb` RegB clear
`clv` clear overflow bit, V=0
`cmpa` 8-bit compare RegA with memory
`cmpb` 8-bit compare RegB with memory
`com` 8-bit logical complement to memory
`coma` 8-bit logical complement to RegA
`comb` 8-bit logical complement to RegB
`cpd` 16-bit compare RegD with memory
`cpx` 16-bit compare RegX with memory
`cpy` 16-bit compare RegY with memory
`daa` 8-bit decimal adjust accumulator
`dbeq` decrement and branch if result=0 `dbeq Y,loop`
`dbne` decrement and branch if result≠0 `dbne A,loop`
`dec` 8-bit decrement memory
`deca` 8-bit decrement RegA
`decb` 8-bit decrement RegB
`des` 16-bit decrement RegSP
`dex` 16-bit decrement RegX
`dey` 16-bit decrement RegY
`ediv` RegY=(Y:D)/RegX, 32-bit by 16-bit unsigned divide
`edivs` RegY=(Y:D)/RegX, 32-bit by 16-bit signed divide
`emacs` 16 by 16 signed multiply, 32-bit add
`emaxd` 16-bit unsigned maximum in RegD
`emaxm` 16-bit unsigned maximum in memory
`emind` 16-bit unsigned minimum in RegD
`eminm` 16-bit unsigned minimum in memory
`emul` RegY:D=RegY*RegD, 16 by 16 to 32-bit unsigned multiply
`emuls` RegY:D=RegY*RegD, 16 by 16 to 32-bit signed multiply
`eora` 8-bit logical exclusive or to RegA
`eorb` 8-bit logical exclusive or to RegB
`etbl` 16-bit look up and interpolation
`exg` exchange register contents `exg X,Y`
`fdiv` unsigned fract div, X=(65536*D)/X
`ibeq` increment and branch if result=0 `ibeq Y,loop`
`ibne` increment and branch if result≠0 `ibne A,loop`
`idiv` 16-bit by 16-bit unsigned div, X=D/X, D=remainder
`idivs` 16-bit by 16-bit signed divide, X=D/X, D= remainder
`inc` 8-bit increment memory
`inca` 8-bit increment RegA
`incb` 8-bit increment RegB
`ins` 16-bit increment RegSP
`inx` 16-bit increment RegX
`iny` 16-bit increment RegY
`jmp` jump always
`jsr` jump to subroutine
`lbcc` long branch if carry clear
`lbcs` long branch if carry set
`lbeq` long branch if result is zero
`lbge` long branch if signed ≥
`lbgt` long branch if signed >
`lbhi` long branch if unsigned >
`lbhs` long branch if unsigned ≥
`lble` long branch if signed ≤
`lblo` long branch if unsigned <
`lbls` long branch if unsigned ≤
`lblt` long branch if signed <
`lbmi` long branch if result is negative
`lbne` long branch if result is nonzero
`lbpl` long branch if result is positive
`lbra` long branch always
`lbrn` long branch never
`lbvc` long branch if overflow clear
`lbvs` long branch if overflow set
`ldaa` 8-bit load memory into RegA
`ldab` 8-bit load memory into RegB
`ldd` 16-bit load memory into RegD
`lds` 16-bit load memory into RegSP
`ldx` 16-bit load memory into RegX
`ldy` 16-bit load memory into RegY
`leas` 16-bit load effective addr to SP `leas 2,sp`
`leax` 16-bit load effective addr to X `leax 2,x`
`leay` 16-bit load effective addr to Y `leay 2,y`
`lsr` 8-bit logical right shift memory
`lsra` 8-bit logical right shift RegA
`lsrb` 8-bit logical right shift RegB
`lsrd` 16-bit logical right shift RegD
`maxa` 8-bit unsigned maximum in RegA
`maxm` 8-bit unsigned maximum in memory
`mem` determine the Fuzzy logic membership grade
`mina` 8-bit unsigned minimum in RegA
`minm` 8-bit unsigned minimum in memory
`movb` 8-bit move memory to memory `movb #100,PTT`

`movw`	16-bit move memory to memory	`movw #13,SCIBD`
`mul`	8 by 8 to 16-bit unsigned RegD=RegA*RegB	
`neg`	8-bit 2's complement negate memory	
`nega`	8-bit 2's complement negate RegA	
`negb`	8-bit 2's complement negate RegB	
`oraa`	8-bit logical or to RegA	
`orab`	8-bit logical or to RegB	
`orcc`	8-bit logical or to RegCC	
`psha`	push 8-bit RegA onto stack	
`pshb`	push 8-bit RegB onto stack	
`pshc`	push 8-bit RegCC onto stack	
`pshd`	push 16-bit RegD onto stack	
`pshx`	push 16-bit RegX onto stack	
`pshy`	push 16-bit RegY onto stack	
`pula`	pop 8 bits off stack into RegA	
`pulb`	pop 8 bits off stack into RegB	
`pulc`	pop 8 bits off stack into RegCC	
`puld`	pop 16 bits off stack into RegD	
`pulx`	pop 16 bits off stack into RegX	
`puly`	pop 16 bits off stack into RegY	
`rev`	Fuzzy logic rule evaluation	
`revw`	weighted Fuzzy rule evaluation	
`rol`	8-bit roll shift left Memory	
`rola`	8-bit roll shift left RegA	
`rolb`	8-bit roll shift left RegB	
`ror`	8-bit roll shift right Memory	
`rora`	8-bit roll shift right RegA	
`rorb`	8-bit roll shift right RegB	
`rtc`	return sub in expanded memory	
`rti`	return from interrupt	
`rts`	return from subroutine	
`sba`	8-bit subtract RegA=RegA-RegB	
`sbca`	8-bit sub with carry from RegA	
`sbcb`	8-bit sub with carry from RegB	
`sec`	set carry bit, C=1	
`sei`	set I=1, disable interrupts	
`sev`	set overflow bit, V=1	
`sex`	sign extend 8-bit to 16-bit reg	`sex B,D`
`staa`	8-bit store memory from RegA	
`stab`	8-bit store memory from RegB	
`std`	16-bit store memory from RegD	
`sts`	16-bit store memory from SP	
`stx`	16-bit store memory from RegX	
`sty`	16-bit store memory from RegY	
`suba`	8-bit sub from RegA	
`subb`	8-bit sub from RegB	
`subd`	16-bit sub from RegD	
`swi`	software interrupt, trap	
`tab`	transfer A to B	
`tap`	transfer A to CC	
`tba`	transfer B to A	
`tbeq`	test and branch if result=0	`tbeq Y,loop`
`tbl`	8-bit look up and interpolation	
`tbne`	test and branch if result≠0	`tbne A,loop`
`tfr`	transfer register to register	`tfr X,Y`
`tpa`	transfer CC to A	
`trap`	illegal instruction interrupt	
`trap`	illegal op code, or software trap	
`tst`	8-bit compare memory with zero	
`tsta`	8-bit compare RegA with zero	
`tstb`	8-bit compare RegB with zero	
`tsx`	transfer S to X	
`tsy`	transfer S to Y	
`txs`	transfer X to S	
`tys`	transfer Y to S	
`wai`	wait for interrupt	
`wav`	weighted Fuzzy logic average	
`xgdx`	exchange RegD with RegX	
`xgdy`	exchange RegD with RegY	

Example	Mode	Effective Address
`ldaa #u`	immediate	No EA
`ldaa u`	direct	EA is 8-bit address
`ldaa U`	extended	EA is a 16-bit address
`ldaa m,r`	5-bit index	EA=r+m (-16 to 15)
`ldaa v,+r`	pre-incr	r=r+v, EA=r (1 to 8)
`ldaa v,-r`	pre-dec	r=r-v, EA=r (1 to 8)
`ldaa v,r+`	post-inc	EA=r, r=r+v (1 to 8)
`ldaa v,r-`	post-dec	EA=r, r=r-v (1 to 8)
`ldaa A,r`	Reg A offset	EA=r+A, zero padded
`ldaa B,r`	Reg B offset	EA=r+B, zero padded
`ldaa D,r`	Reg D offset	EA=r+D
`ldaa q,r`	9-bit index	EA=r+q
`ldaa W,r`	16-bit index	EA=r+W
`ldaa [D,r]`	D indirect	EA={r+D}
`ldaa [W,r]`	indirect	EA={r+W}

Freescale 6812 addressing modes **`r`** is **`X`**, **`Y`**, **`SP`**, or **`PC`**

Pseudo op	Meaning
`org`	Where to put subsequent code
`= equ set`	Define a constant symbol
`dc.b db fcb .byte`	Allocate byte(s) with values
`fcc`	Create an ASCII string
`dc.w dw fdb .word`	Allocate word(s) with values
`dc.l dl .long`	Allocate 32-bit with values
`ds ds.b rmb .blkb`	Allocate bytes without init
`ds.w .blkw`	Allocate word(s) without init

n is Metrowerks number

Vector	**n**	Interrupt Source	Arm
$FFFE		**Reset**	None
$FFF8	3	**Trap**	None
$FFF6	4	**SWI**	None
$FFF0	7	**Real time interrupt**	CRGINT.RTIE
$FFEE	8	**Timer channel 0**	TIE.C0I
$FFEC	9	**Timer channel 1**	TIE.C1I
$FFEA	10	**Timer channel 2**	TIE.C2I
$FFE8	11	**Timer channel 3**	TIE.C3I
$FFE6	12	**Timer channel 4**	TIE.C4I
$FFE4	13	**Timer channel 5**	TIE.C5I
$FFE2	14	**Timer channel 6**	TIE.C6I
$FFE0	15	**Timer channel 7**	TIE.C7I
$FFDE	16	**Timer overflow**	TSCR2.TOI
$FFD6	20	**SCI0 TDRE, RDRF**	SCI0CR2.TIE,RIE
$FFD4	21	**SCI1 TDRE, RDRF**	SCI1CR2.TIE,RIE
$FFCE	24	**Key Wakeup J**	PIEJ.[7,6,1,0]
$FFCC	25	**Key Wakeup H**	PIEH.[7:0]
$FF8E	56	**Key Wakeup P**	PIEP.[7:0]

Interrupt Vectors and interrupt number.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0040	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0	TIOS
$0044-5	Bit 15	14	13	12	11	10		Bit 0	TCNT
$0046	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0	TSCR1
$004C	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I	TIE
$004D	TOI	0	PUPT	RDPT	TCRE	PR2	PR1	PR0	TSCR2
$004E	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F	TFLG1
$004F	TOF	0	0	0	0	0	0	0	TFLG2
$0050-1	Bit 15	14	13	12	11	10		Bit 0	TC0
$0052-3	Bit 15	14	13	12	11	10		Bit 0	TC1
$0054-5	Bit 15	14	13	12	11	10		Bit 0	TC2
$0056-7	Bit 15	14	13	12	11	10		Bit 0	TC3
$0058-9	Bit 15	14	13	12	11	10		Bit 0	TC4
$005A-B	Bit 15	14	13	12	11	10		Bit 0	TC5
$005C-D	Bit 15	14	13	12	11	10		Bit 0	TC6
$005E-F	Bit 15	14	13	12	11	10		Bit 0	TC7
$0082	ADPU	AFFC	ASWAI	ETRIGLE	ETRIGP	ETRIG	ASCIE	ASCIF	ATD0CTL2
$0083	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0	ATD0CTL3
$0084	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0	ATD0CTL4
$0085	DJM	DSGN	SCAN	MULT	0	CC	CB	CA	ATD0CTL5
$0086	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0	ATD0STAT0
$008B	CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0	ATD0STAT1
$008D	Bit 7	6	5	4	3	2	1	Bit 0	ATD0DIEN
$008F	PAD07	PAD06	PAD05	PAD04	PAD03	PAD02	PAD01	PAD00	PORTAD0
$0090-1	Bit 15	14	13	12	11	10		Bit 0	ATD0DR0
$0092-3	Bit 15	14	13	12	11	10		Bit 0	ATD0DR1
$0094-5	Bit 15	14	13	12	11	10		Bit 0	ATD0DR2
$0096-7	Bit 15	14	13	12	11	10		Bit 0	ATD0DR3
$0098-9	Bit 15	14	13	12	11	10		Bit 0	ATD0DR4
$009A-B	Bit 15	14	13	12	11	10		Bit 0	ATD0DR5
$009C-D	Bit 15	14	13	12	11	10		Bit 0	ATD0DR6
$009E-F	Bit 15	14	13	12	11	10		Bit 0	ATD0DR7
$00C9	0	0	0	SBR12	SBR11	SBR10		SBR0	SCI0BD
$00CA	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	SCI0CR1
$00CB	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCI0CR2
$00CC	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCI0SR1
$00CD	0	0	0	0	0	BRK13	TXDIR	RAF	SCI0SR2
$00CF	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0	SCI0DRL
$00D0-1	0	0	0	SBR12	SBR11	SBR10		SBR0	SCI1BD
$00D2	LOOPS	SCISWAI	RSRC	M	WAKE	ILT	PE	PT	SCI1CR1
$00D3	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCI1CR2
$00D4	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	SCI1SR1
$00D5	0	0	0	0	0	BRK13	TXDIR	RAF	SCI1SR2
$00D7	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0	SCI1DRL
$0240	PT7	PT6	PT5	PT4	PT3	PT2	PT1	PT0	PTT
$0242	DDRT7	DDRT6	DDRT5	DDRT4	DDRT3	DDRT2	DDRT1	DDRT0	DDRT
$0248	PS7	PS6	PS5	PS4	PS3	PS2	PS1	PS0	PTS
$024A	DDRS7	DDRS6	DDRS5	DDRS4	DDRS3	DDRS2	DDRS1	DDRS0	DDRS
$0250	PM7	PM6	PM5	PM4	PM3	PM2	PM1	PM0	PTM
$0252	DDRM7	DDRM6	DDRM5	DDRM4	DDRM3	DDRM2	DDRM1	DDRM0	DDRM
$0258	PP7	PP6	PP5	PP4	PP3	PP2	PP1	PP0	PTP
$025A	DDRP7	DDRP6	DDRP5	DDRP4	DDRP3	DDRP2	DDRP1	DDRP0	DDRP
$0260	PH7	PH6	PH5	PH4	PH3	PH2	PH1	PH0	PTH
$0262	DDRH7	DDRH6	DDRH5	DDRH4	DDRH3	DDRH2	DDRH1	DDRH0	DDRH
$0268	PJ7	PJ6	0	0	0	0	PJ1	PJ0	PTJ
$026A	DDRJ7	DDRJ6	0	0	0	0	DDRJ1	DDRJ0	DDRJ

TSCR1 is the first 8-bit timer control register

bit 7 **TEN**, 1 allows the timer to function normally, 0 means disable timer including **TCNT**

TIOS is the 8-bit output compare select register, one bit for each channel (1 = output compare, 0 = input capture)

TIE is the 8-bit output compare arm register, one bit for each channel (1 = armed, 0 = disarmed)

TSCR2 is the second 8-bit timer control register

bits 2,1,0 are **PR2**, **PR1**, **PR0**, which select the rate, let **n** be the 3-bit number formed by **PR2**, **PR1**, **PR0**
without PLL **TCNT** is 8MHz/2^{n}, with PLL **TCNT** is 24MHz/2^{n}, **n** ranges from 0 to 7

				E = 8 MHz		E = 24 MHz	
PR2	PR1	PR0	Divide by	TCNT period	TCNT frequency	TCNT period	TCNT frequency
0	0	0	1	125 ns	8 MHz	41.7 ns	24 MHz
0	0	1	2	250 ns	4 MHz	83.3 ns	12 MHz
0	1	0	4	500 ns	2 MHz	167 ns	6 MHz
0	1	1	8	1 µs	1 MHz	333 ns	3 MHz
1	0	0	16	2 µs	500 kHz	667 ns	1.5 MHz
1	0	1	32	4 µs	250 kHz	1.33 µs	667 kHz
1	1	0	64	8 µs	125 kHz	2.67 µs	333 kHz
1	1	1	128	16 µs	62.5 kHz	5.33 µs	167 kHz

SCI0DRL 8-bit SCI0 data register

SCI0BD is 16-bit SCI0 baud rate register, let **n** be the 13-bit number Baud rate is EClk/**n**/16

SCI0CR1 is 8-bit SCI0 control register

bit 4 M, Mode, 0 = One start, eight data, one stop bit, 1 = One start, eight data, ninth data, one stop bit

SCI0CR2 is 8-bit SCI0 control register

bit 7 TIE, Transmit Interrupt Enable, 0 = TDRE interrupts disabled, 1 = interrupt whenever TDRE set
bit 5 RIE, Receiver Interrupt Enable, 0 = RDRF interrupts disabled, 1 = interrupt whenever RDRF set
bit 3 TE, Transmitter Enable, 0 = Transmitter disabled, 1 = SCI transmit logic is enabled
bit 2 RE, Receiver Enable, 0 = Receiver disabled, 1 = Enables the SCI receive circuitry.

SCI0SR1 is 8-bit SCI0 status register

bit 7 TDRE, Transmit Data Register Empty Flag
Set if transmit data can be written to **SCI0DRL**
Cleared by **SCI0SR1** read with TDRE set followed by **SCI0DRL** write
bit 5 RDRF, Receive Data Register Full
set if a received character is ready to be read from **SCI0DRL**
Clear the RDRF flag by reading **SCI0SR1** with RDRF set and then reading **SCI0DRL**

ATD0CTL5 is used to start an ADC conversion

bit 7 DJM is set to 1 for right justified and to 0 for left justified
bits 2-0 specify the ADC channel to sample

ATD0STAT0 is used to tell when the ADC conversion is done

bit 7 SCF cleared on a write to **ATD0CTL5** and is set when the conversion sequence is done

BSR

Branch to Subroutine

Operation: (SP) – $0002 ⇒ SP
$RTN_H : RTN_L ⇒ M_{(SP)} : M_{(SP+1)}$
(PC) + Rel ⇒ PC

Source Form	Address Mode	Object Code	HCS12
BSR *rel8*	REL	07 rr	SPPP

EMAXM

Place Larger of Two Unsigned 16-Bit Values in Memory

Operation: MAX ((D), (M : M + 1)) ⇒ M : M + 1

EMINM

Place Smaller of Two Unsigned 16-Bit Values in Memory

Operation: MIN ((D), (M : M + 1)) ⇒ M : M + 1