Name:……. **Signature**…… **ID-No**:…….

Registers related to TMR0 operation

Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	After POR, BOR	After other Resets
TMR0L	Timer0 Module Low Byte Register (counts bits 7 to 0)								xxxx xxxx	uuuu uuuu
TMR0H	Timer0 Module High Byte Latch (latches counter bits 15 to 8)								0000 0000	0000 0000
INTCON	GIE / GIEH	PEIE GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x	0000 0000u
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	1111 1111	1111 1111
TRISA	–	PortA data direction register							–111 1111	–111 1111

Legend: x=unknown, u=unchanged, – =unimplemented, crossed cells are not used by Timer0

T0CON (control register for TIMER0).

bit-7	TMR0ON	1: Enables Timer0, 0: Stops counting.
bit-6	T08BIT	1: for 8-bit counter, 0: for 16-bit counter.
bit-5	T0CS	Source of the T0CKI pulses. 1: T0CK1 (RA4) , 0:osc-Clock.
bit-4	T0SE	Active edge of the counting, 1: High to Low, 0:Low to High,
bit-3	PSA	Abort Prescaler 1: disable prescaler, 0: enable prescaler,
bit-0..2	T0PS0...T0PS2	Prescaler Factor 000, 001 ... ,111 ==> 2, 4, 8, 16, 32, 64, 128, 256.

Upon RESET, TMR0ON=1, T0CS=0, and T0PS2...0=111.

8-bit counting: Use only TMR0L. Even if TMR0L is an 8-bit register, reading or writing it takes 2cc.
16-bit counting: In reading, read first TMR0L, then TMR0H. In writing, write first TMR0H then TMR0L.

Information for CC8E coding

Precedence of CC8E operators

( )	expression block (Highest prec.)
++ --	increment, decrement
~	bitwise complement
*** / %**	multiplication, division, modulo
+ -	addition, subtraction
<< >>	left-shift, right-shift
< <= > >=	relational operators
== !=	equality, inequality
&	logical and
^	exclusive or
|	inclusive or
&&	logical and with break-on-false
||	logical or with break-on-true
= += -= *= /= etc.	assignments (Lowest preced.)

Integer variable types

```
unsigned a8; // 8 bit unsigned
char a8; // 8 bit unsigned
unsigned long i16;//16 bit unsigned
char varX;
char counter, L_byte, H_byte;
bit ready; // 0 or 1
bit flag, stop, semafor;
int i; // 8 bit signed
signed char sc; // 8 bit signed
long i16; // 16 bit signed
uns8 u8; // 8 bit unsigned
uns16 u16; // 16 bit unsigned
uns24 u24; // 24 bit unsigned
uns32 u32; // 32 bit unsigned
int8 s8; // 8 bit signed
int16 s16; // 16 bit signed
int24 s24; // 24 bit signed
int32 s32; // 32 bit signed
```

For a 16-bit integer constant **N**
Lower-8-bit of **N** is **(N)%256**;
Upper-8-bit of **N** is **(N)/256**;
Lower-8-bit of **-N** is **(65536-N)%256** or **-N%256**;
Upper-8-bit of **-N** is **(65536-N)/256** or **-N/256**;
For an 8-bit number **N**, **-N = 256-N**

Conditional-operations: == != > >= < <=
Conditions:

[**++**|**--**]<var.> <cond-oper> <value>[**&&** <cond1>][**||** <cond2>]

```
if (x = = 7) ..
if (Carry = = 1 && a1 < a2) ..;
if (y > 44 || Carry || x != z) ..
if (--index > 0) ..;
if (bx = = 1 || ++i < max) ..
if (sub_1() != 0) ..
```

Internal functions translated into special instructions or instruction sequences.

```
btsc, btss, clearRAM, clrwdt, decsz, incsz, nop,
nop2, retint, rl,rr, sleep, skip, skipL, skipM,
swap, decsnz, incsnz, addWFC, subWFB, subFWB, rlnc,
rrnc, negate, decadj, multiply, skipIfEQ, skipIfLT,
skipIfGT, skipIfZero, pushStack, popStack,
softReset, tableRead, tableReadInc, tableReadDec,
tableReadPreInc, tableWrite, tableWriteInc,
tableWriteDec, tableWritePreInc
```

Predefined PIC registers (Header files define the rest)

```
TOSU, TOSH, TOSL, STKPTR, PCLATU, PCLATH, PCL,
TBLPTRU, TBLPTRH, TBLPTRL, TBLPTR, TABLAT, PRODH,
PRODL,   INTCON, INTCON2, INTCON3,INDF0, POSTINC0,
POSTDEC0, PREINC0, PLUSW0, FSR0H, FSR0L, FSR0,
W, WREG, BSR, BSRL, INDF1, POSTINC1, POSTDEC1,
PREINC1, PLUSW1, FSR1H, FSR1L, FSR1, INDF2,
POSTINC2, POSTDEC2, PREINC2, PLUSW2, FSR2H, FSR2L,
FSR2, STATUS, Carry, DC, Zero_, Overflow, Negative
```

Extensions to ANSI C:

The individual bits can be separated by the period '.':

```
0b0100 ;   0b.0.000.1.01.00000
bin(0100) ;  bin(0001.0100) ; etc.
```

Partitions of variables:

i8.7 ... i8.0	: bits of the 8-bit variable **i8**.
i16.15 ... i16.0	: bits of the 16-bit variable **i16**.
i16.low8	: least significant byte of **i16**;
i16.high8	:most significant byte of **i16**.

CC8E command line arguments:

```
CC8E.exe file.ext -Ln -p18F452
```

file.ext is c-source file, -Ln produces list file, -p selects processor.

Name:……. **Signature**…… **ID-No**:…….

CMPE423 Embedded System Design

Interrupt Settings Information

CC8E interrupt templates for PIC18F452

Non hierarchical (flat, only low priority)
for flat case interrupt vector is at address 0x008.

```
#pragma origin 0x08
void LPISR(void){
  // Low priority interrupt service routine
  static char WX,STATUSX,BSRX ;
        // to save W, STATUS, BSR
  STATUSX=STATUS;WX=WREG;BSRX=BSR;
  if(xlpIF) {
   ... ...  //minimum service on xlp interrupt
    xlpIF=0;};
  if(ylpIF) {
   ... ...  //minimum service on ylp interrupt
    ylpIF=0;};
  WREG=WX;BSR=BSRX;STATUS=STATUSX;
   // restore registers
  retint();   // return from interrupt
 }

void initIS(void){
   // initialize interrupt servicing
   xlpIE=1 ; // interrupt on xlp enabled
   xlpIP=0 ; // low priority interrupt on xlp
   ylpIE=1 ; // interrupt on ylp enabled
   ylpIP=0 ; // low priority interrupt on ylp
   IPEN=0   ; // all interrupts low priority
   PEIE=1   ; // (=GIEL) Peripheral (Group-b)
              // interrupts enabled
   GIE=1    ; // Low and High-priority all
              // interrupts enabled
   }
... ...
void main(void){
  Initialize();
  initIS();//initializes interrupts xxx and yyy
  do{ //mainloop
     ... ...
    }while(1);
  }
```

Two level Hierarchical (Low Priority and High Priority)
for hierarchical two-level case
high-priority interrupt vector is at 0x008,
low-priority interrupt vector is at address 0x018

```
void HPISR(void); //prototype (*)
#pragma origin 0x08
void HPVector(void){HPISR();}
#pragma origin 0x18
void LPISR(void){ //high priority ISR
  STATUSX=STATUS;WX=WREG;BSRX=BSR;
  if(xlpIF) {
    ... ...  // min. service on xhp interrupt
    xlpIF=0;}
  WREG=WX;BSR=BSRX;STATUS=STATUSX;
  retint();   // return from interrupt
  }
void HPISR(void){
  STATUSX=STATUS;WX=WREG;BSRX=BSR;
  if(xhpIF) {
    ... ... // min. service on xhp interrupt
     xhpIF=0;}
  WREG=WX;BSR=BSRX;STATUS=STATUSX;
  retint();  // return from interrupt
  }
void initIS(void){
   // initialize interrupt servicing
   xlpIE=1 ; // interrupt on xlp enabled
   xlpIP=0 ; // low priority interrupt on xlp
   xhpIE=1 ; // interrupt on xhp enabled
   xhpIP=0 ; // low priority interrupt on xhp
   IPEN=1   ; // two level interrupts
   GIEL=1   ; //(=GIE)Low-prior. Interr.enabled
   GIEH=1   ; //(=PEIE)Low-High-all int, enable
   }
... ...
void main(void){
   ...exactly same to NonHier. case.
  }
```

(*) if service routine fits in 16-byte then no need to hp-prototype and linking it to the vector.

Initialization of interrupts on some commonly used devices:

interrupt:	Timer0 -overflow	Int0 edge	Int1 edge	Int1 edge	RB.4-7-edge	UART-received	UART-transmit	ADC-over	value to set
enable:	**TMR0IE**	**INT0IE**	**INT1IE**	**INT2IE**	**RBIE**	**RCIE**	**TXIE**	**ADIE**	**=1**
flag:	**TMR0IF**	**INT0IF**	**INT1IF**	**INT2IF**	**RBIF**	**RCIF**	**TXIF**	**ADIF**	**=0**
priority:	**TMR0IP**	(always High)	**INT1IP**	**INT2IP**	**RBIP**	**RCIP**	**TXIP**	**ADIP**	low:**0**,high:**1**
edge:		**INTEDG0**	**INTEDG1**	**INTEDG2**					↑:**1**, ↓:**0**

Global interrupt flag settings

For non hierarchical **IPEN=0 ; PEIE=1 ; GIE=1 ;** For hierarchical: **IPEN=1 ; GIEL=1 ; GIEH=1 ;**
Actually, **PEIE** and **GIEL** flags are the same flags, and, **GIE**and **GIEL** are the same. If **IPEN=1**, then **PEIE/GIEL** enables all low priority interrupts, and, **GIE/GIEH** enables high priority interrupts

Register Save and Restore Rules

-If only bits are tested and manipulated (set or cleared) no need to save any register.
-If any register operation is necessary then save **STATUS; WREG; BSR.**
-If any pointer operation is necessary then save **FSR; TABLAT; TBLPTRH; TBLPTRL;**
-If mainloop is empty then no need to save registers in Low-Priority ISR.

Critical Code Rules

-Variables manipulated in the interrupt service routines are critical variables.
-If manipulation of a critical variable is necessary in the mainline then interrupts must be disabled before manipulating the critical variables, and interrupts must be enabled after the manipulation is over.
-If a part of the code is time-critical so that an interrupt may destroy its timing, the interrupts must be disabled before that code, and enabled after the critical part is over.