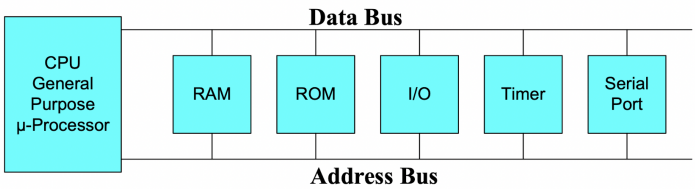
VE373 FINAL RC

L1 - L9

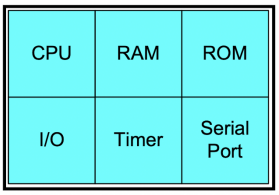
2022.07.31

L1 — Introduction to Embedded Systems

1. **Microprocessor (MPU) & Microcontroller (MCU)**



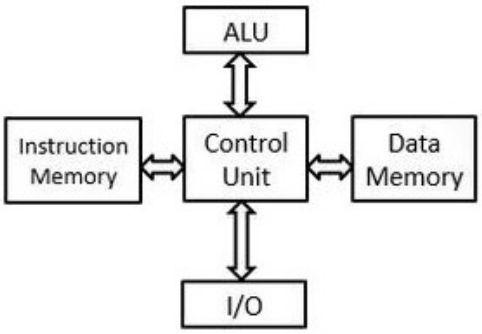
(a) Microprocessor



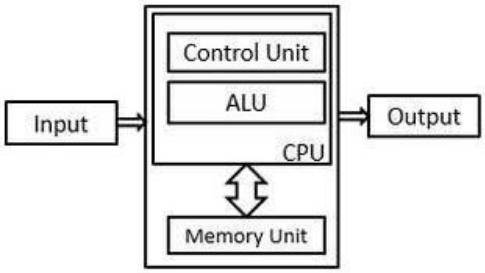
(b) Microcontroller

Microprocessor	Micro Controller
Microprocessor is heart of Computer system.	Micro controller is a hear of embedded system.
It is just a processor. Memory and I/O components have to be connected externally.	**Micro controller has external processor along with internal memory and I/O components.**
The circuit is large.	The circuit is small.
Cannot be used in compact systems and hence inefficient.	Can be used in compact systems and hence it is an efficient technique.
Cost of the entire system increases.	Cost of the entire system is low.
Due to external components, the entire power consumption is high. Hence it is not suitable to used with devices running on stored power like batteries.	Since external components are low, total power consumption is less and can be used with devices running on stored power like batteries.
Most of the microprocessors do not have power saving features.	Most of the micro controllers have power saving modes like idle mode and power saving mode. This helps to reduce power consumption even further.
Since memory and I/O components are all external, each instruction will need external operation, hence it is relatively slower.	Since components are internal, most of the operations are internal instruction, hence speed is fast.
Microprocessor have less number of registers, hence more operations are memory based.	Micro controller have more number of registers, hence the programs are easier to write
Microprocessors are based on Von Neumann model/architecture where program and data are stored in same memory module.	Micro controllers are based on Harvard architecture where program memory and Data memory are separate.
Mainly used in personal computers.	Used mainly in washing machines, MP2 players.

2. **Von Neumann & Havard Architecture**



(a) Harvard Architecture



(b) Von Neumann Architecture

Harvard architecture	Von Neumann architecture
It required two memories for their instruction and data.	**It required only one memory for their instruction and data.**
Harvard architecture is required separate bus for instruction and data.	Von Neumann architecture is required only one bus for instruction and data.
Processor can complete an instruction in one cycle	Processor needs two clock cycles to complete an instruction.
Easier to pipeline, so high performance can be achieve.	Low performance as compared to Harvard architecture.
Comparatively high cost.	It is cheaper.

L2 — PIC MCU Architecture

1. **PIC32**

- MIPS architecture, M4K core, RISC, 5-stage pipelining
- **32-bit address and data bus**
- **200 MHz max frequency**
- 32 32-bit general purpose registers (GPR)
- 64K to 512K on-chip flash memory
- 16K to 128K bytes on-chip SRAM
- **4GB virtual memory space**
- SFRs in CPU and coprocessor0 (CP0)
- Multiple interrupt sources and interrupt vectors
- **A variety of peripherals**
- 7 sets of I/O ports

Figure 3: 32-bit PIC32 MCU Example: Architecture

n-bit data address bus = support up to 2^n byte memory.

PIC32 is a combination of Von Neumann and Hasrvard architecture. Harvard architecture on physical falsh memory, and Von Neumann on virtual level and SRAM. See here for more details if interested.

2. **Special Function Register**

   Used to config, control and monitor various aspects of the microprocessor's function.

   For example, `T1CON`, `TMR1` and `PR1` when using Timer 1.

L3 — Embedded Programming

1. **Compiler**

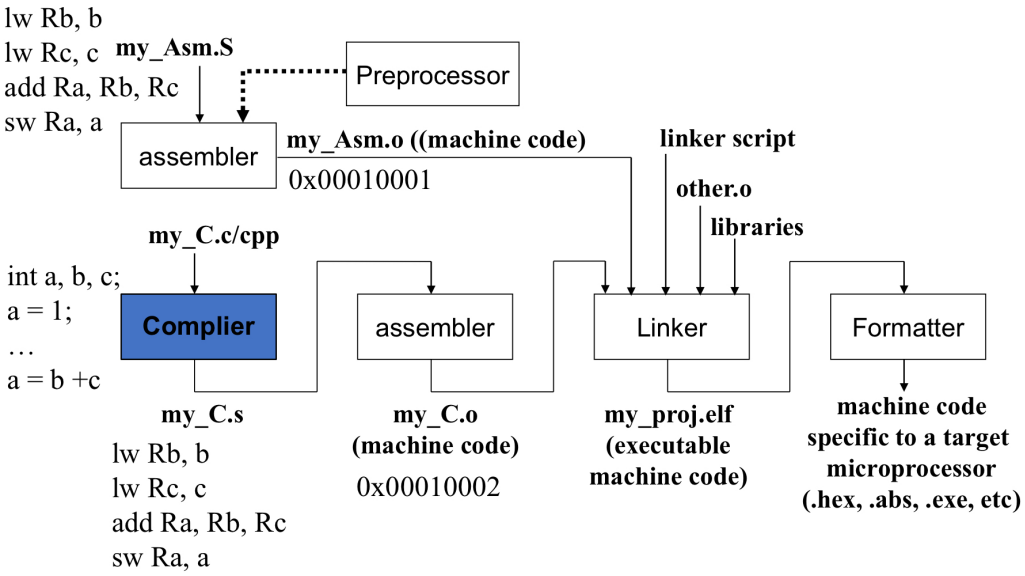


Figure 4: Workflow

2. **Const & Volatile**
   - const: the value not supposed to be written by program (read only).
   - volatile: the value can be changed by something other than program so it should be reexamined frequently.

3. **Directives**

   Directives: compiler dependent commands.

   e.g. `#pragma interrupt func_name ipln`.

L4 — Timers and IO

1. **Oscillator**
   - **Internal oscillator**:
     - Integrated on-chip within the processor
     - Low frequency accuracy and stability.
     - Most of internal oscillator are RC circuits
   - **External oscillator**
     - Located off-chip on the PCB
     - High frequency accuracy and stability
     - Most of external oscillator are crystal oscillator

   Peripherals don't have a high clock frequency:
   - No need
   - Limited by parasitics
   - Power consumption

2. **Timer**

   Two types, five timers on PIC32:

- Type A — Timer 1
  - 16-bit synchronous/asynchronous timer/event counter
  - Internal or external clock source
  - Gated external event timer
- Type B — Timer 2, 3, 4, 5
  - 16/32-bit synchronous timer/event counter
  - Internal or external clock source
  - Gated external event timer

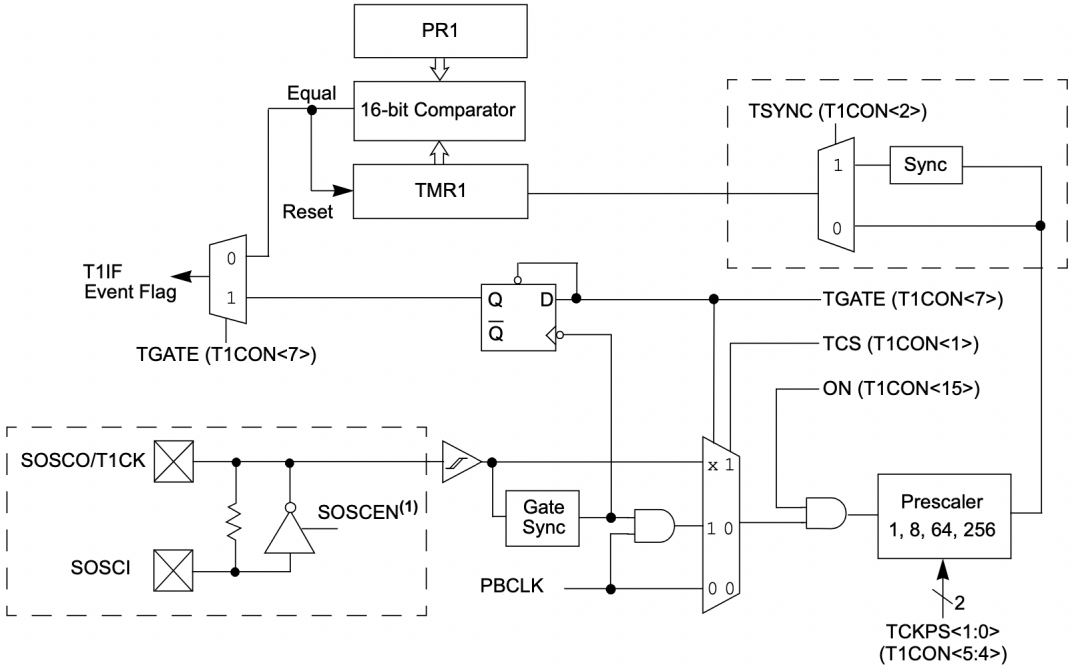


Figure 5: Timer 1 block diagram (Datasheet Page. 197)

3. **16-bit synchronous timer setup steps**

   Use internal clock source — `PBCLK` as example.

   1. Clear control bit `ON` (`TxCON<15>=0`) to disable timer.
   2. Clear control bit `TCS` (`TxCON<1>=0`) to select internal clock source `PBCLK`.
   3. Select desired clock prescale value.
   4. Load/clear timer register `TMRx` to specify initial counting value.
   5. Load period register `PRx` with desired 16-bit match value.
   6. If interrupts used:
      i. Clear interrupt flag bit
      ii. Configure interrupt priority and subpriority levels
      iii. Set interrupt enable bit
   7. Set control bit `ON` (`TxCON<15>=1`) to start timer

L4 — Timers and IO

1. **GPI/O Ports**

   Each port has 4 associated registers for its operation:

   - `TRISx` register: Data Direction register, or Tri-State Control register

  - `1` for input
  - `0` for output
  - all port I/O pins are defined as inputs after a device Reset. Certain I/O pins are shared with analog peripherals and default to analog inputs after a device Reset.
- `LATx` register: output latch
  - used to write date to the port I/O pins. The `LATx` Latch register holds the data written to either the `LATx` or `PORTx` registers.
  - reading the LATx Latch register reads the last value written to the corresponding PORT or Latch register.
- `PORTx` register: reads the levels on the pins of the device
  - used to read the current state of the signal applied to the port I/O pins
  - writing to a `PORTx` register performs a write to the port's latch, `LATx` register, latching the data to the port's I/O pins
- `ODCx` register: Open-drain control

  Pins are configured as digital outputs by setting the corresponding `TRIS` register bits = 0. When configured as digital outputs, these pins are CMOS drivers or can be configured as open-drain outputs by setting the corresponding bits in the Open-Drain Configuration (`ODCx`) register.

  The open-drain feature allows generation of outputs higher than V_{DD} (e.g., 5V) on any desired 5V tolerant pins by using external pull-up resistors. The maximum open-drain voltage allowed is the same as the maximum V_{IH} specification.

2. **`CLR`, `SET` And `INV` Registers**

   Provide fast atomic bit manipulations, perform operation in hardware atomically.

   Reading SET, CLR and INV registers returns undefined values.

   E.g. `T2CONSET = 0x8000` in homework 1.

L5 — Interrupts

1. **Detect events**

   Dealing with asynchronous events:

   - Exceptions caused by program execution
   - Peripheral needs attention or has completed a requested action
   - Application makes a system call

2. **Software polling**

   - We can repeatedly poll the application for peripherals.
   - When an event occurs, detect this via a poll and take action
   - Common in small or low-performance real-time embedded systems
     - Predictable timing
     - Low hardware cost
   - In other systems, wastes CPU time
     - Affect processor's responsiveness and efficiency

```
1 /* Timer control */
2 T2CON = 0x0;       // Stop Timer2 and clear control register
3                    // prescale 1:1, internal clock source
4 TMR2 = 0x0;        // Clear Timer
5 PR2 = 0xFFFF;
6 T2CONSET = 0x8000; //Start Timer2
7
8 /* polling */
9 while (TMR2 != 1000);
10
11 /* move on */
12 ...
```

3. **Interrupt**

   - Let the application for peripheral notify us automatically.
   - Take action (respond to the event) when such a notification occurs (or shortly later).

   When a special event or error occurs

   - I/O controller interrupts CPU – by a call from the I/O hardware
   - CPU stops executing the current program
   - CPU goes to run an **interrupt handler** (a special piece of program) or **interrupt service routine** (ISR)
   - After finish executing the ISR, CPU returns back to the interrupted program and resume
   - Not synchronized to instruction execution, may happen between any two instructions

4. **Typical Types of Interrupts**

   - Internal (exception)
     - By CPU errors
     - By bad instructions
   - External (interrupt)
     - External I/O device
     - Timer
     - Reset
     - System failure

   Interrupt comes as a high-level signal or a rising edge depending on processors

5. **SFRs for Interrupt**

   - `INTCON` – Interrupt Control Register
   - `INTSTAT` – Interrupt Status Register
   - `TPTMR` – Temporal Proximity Timer Register
   - `IFSx` – Interrupt Flag Status Registers
   - `IECx` – Interrupt Enable Control Registers
   - `IPCx` – Interrupt Priority Control Registers

6. **Interrupt Control**

   1. Enable interrupt globally
   2. Config `INTCON` to select multi vector mode.
   3. Enable individual interrupt, set priority
   4. When interrupt event happens, flag is set regardless of the enable bit
   5. If enabled, interrupt service routine (ISR)
   6. In ISR, service interrupt, clear interrupt flag (for most interrupt requests)
   7. Main program resumes

7. **Enable/disable all interrupt sources globally**

   - `asm("ei"); // inline MIPS32 assembly`
   - `asm("di");`
   - Corresponding to enable/disable IE bit (`STATUS<0>`)
   - Must be enabled before individual interrupt can be used

8. **Operation Mode**

   - Controlled by `MVEC` bit in `INTCON`

- **Single vector mode:**
  - All interrupt requests will be serviced at one vector address (default upon reset)
  - One location for all different interrupts
  - have to determine what exception/interrupt has just happened, by checking status registers
  - `if...else...` structure
- **Multi vector mode:** Interrupt requests will be serviced at the calculated vector address
  - One location for one (or a small number of) interrupt
  - Take actions or handle interrupts directly

9. **Interrupt Vector Table**

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source(1)	IRQ Number	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INT0 – External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1 – Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>
IC1 – Input Capture 1	5	5	IFS0<5>	IEC0<5>	IPC1<12:10>	IPC1<9:8>
OC1 – Output Compare 1	6	6	IFS0<6>	IEC0<6>	IPC1<20:18>	IPC1<17:16>
INT1 – External Interrupt 1	7	7	IFS0<7>	IEC0<7>	IPC1<28:26>	IPC1<25:24>
T2 – Timer2	8	8	IFS0<8>	IEC0<8>	IPC2<4:2>	IPC2<1:0>
IC2 – Input Capture 2	9	9	IFS0<9>	IEC0<9>	IPC2<12:10>	IPC2<9:8>
OC2 – Output Compare 2	10	10	IFS0<10>	IEC0<10>	IPC2<20:18>	IPC2<17:16>
INT2 – External Interrupt 2	11	11	IFS0<11>	IEC0<11>	IPC2<28:26>	IPC2<25:24>
T3 – Timer3	12	12	IFS0<12>	IEC0<12>	IPC3<4:2>	IPC3<1:0>
IC3 – Input Capture 3	13	13	IFS0<13>	IEC0<13>	IPC3<12:10>	IPC3<9:8>
OC3 – Output Compare 3	14	14	IFS0<14>	IEC0<14>	IPC3<20:18>	IPC3<17:16>
INT3 – External Interrupt 3	15	15	IFS0<15>	IEC0<15>	IPC3<28:26>	IPC3<25:24>
T4 – Timer4	16	16	IFS0<16>	IEC0<16>	IPC4<4:2>	IPC4<1:0>
IC4 – Input Capture 4	17	17	IFS0<17>	IEC0<17>	IPC4<12:10>	IPC4<9:8>

10. **ISR — Interrupt Service Routine**

    Interrupt flag should be cleared in ISR. Otherwise interrupt will be triggered over and over again.

    In ISR, you need to

    - Clear interrupt flag
    - Perhaps, stop peripheral
    - Perhaps, disable interrupt (if in critical section)
    - Do whatever you need to do, fix problem, respond to the peripheral
    - Perhaps, some configuration
    - Perhaps, restart peripheral for next interrupt
    - Perhaps, re-enable interrupt
    - Perhaps, start another peripheral for another interrupt
    - Return to the interrupted instruction

11. **Critical section/region**

    Should NOT be interrupted

    Should disable interrupts while executing those instructions

12. **Interrupt priority**

    Devices needing more urgent attention get higher priority

    Higher priority interrupt can interrupt execution of a lower priority interrupt

    Three priority levels

    - Group level: 0 - 7 (highest)
    - Subgroup level: 0 - 3 (highest)
    - Natural level: 0 - 63 (lowest)

13. **Preemption**

    Higher priority interrupt request (IRQ) preempts (overrides) any lower priority interrupts (on group level).

    No preemption on subgroup and natural levels.

14. **IPL and RIPL**

    - Requested interrupt priority level (RIPL)
      - In field `RIPL` (`CAUSE<12:10>`)
      - 0...7, encode requested interrupt priority
    - Current CPU priority (IPL)
      - In field `IPL` (`STATUS<12:10>`)
      - Usually default 0 (no interrupt), changes to IRQ priority when servicing IRQ
      - 0...7, encode the MIPS interrupt priority hierarchy
    - Preemption happens only when `RIPL` > `IPL`

15. **How to write ISR**

```
#pragma interrupt foo ipl4 vector @23
void foo(void){
  ...
}
// foo will be located at the address of interrupt vector 23

#pragma interrupt bar ipl5 vector 23
void bar(void){
  ...
}
// bar will be located in general purpose program memory
// A dispatch function targeting bar will be created at exception vector address
    ↪ 23
```

    `iplx`, `x` = 0...7, 0 means interrupt disabled. `x` must match the group interrupt priority level specified in `IPCx`.

    Each interrupt vector has 8 words, usually holding dispatch function that associates the vector address with the real interrupt handler function.

16. **Change Notice (CN)**

    Generate interrupt upon change of state on selected CN pins

    - CN pins must be configured as inputs
    - Two adjacent reads (delayed by `SYSCLK`) of CN PORT are compared, mismatch generated if different
    - Any mismatch provides an interrupt signal

17. **CN configuration**

    1. Disable all interrupts
    2. Set selected CN pin as input
    3. Enable CN module (`CNCON.ON = 1`)

4. Enable individual CN inputs, and pull ups (optional)
5. Read corresponding PORT registers to clear pre-existing mismatch condition
6. Configure the CN interrupt priority
7. Clear CN interrupt flag
8. Enable CN interrupt
9. Enable all interrupts

18. **CN ISR**

    1. Temporarily disable CN interrupt
    2. PORT should be read to clear mismatch condition
    3. Interrupt flag should be cleared by user program
    4. Current PORT value should be compared with the last PORT value of the same pin to determine which CN pin changed
    5. Re-enable CN interrupt

    Note:

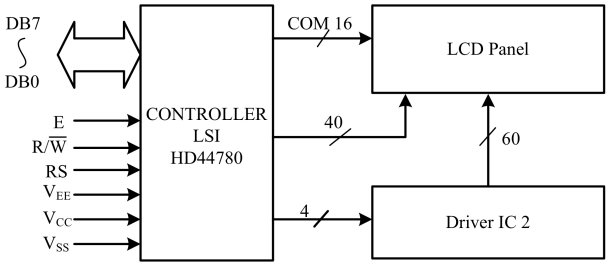
    1. CN interrupt is a persistent interrupt: only clearing IF flag is not enough, must clear the corresponding condition.
    2. CN interrupts of all pins go into the same vector address (vector number 26). Therefore in ISR we should check which IO pin is changed.

L6 — Liquid-Crystal Display (LCD) Driver

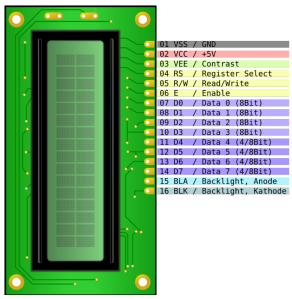
1. **Make good use of datasheet**

   The datasheet of LCD is file `lcd1602a_398762.pdf` in `./Reference_Materials/PIC32 Starter Kit/`.

2. **LCD module**



(a) LCD module



(b) LCD pins definition

3. **Display Data RAM (DDRAM)**

   - Can hold up to 80 bytes (characters)
   - 40 locations mapped to Line1: `0x00 - 0x27`
   - 40 locations mapped to Line2: `0x40 - 0x67`
   - 2 line × 16 display window is aligned with DDRAM from the head

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

2-Line by 16-Character Display

- Display window can be shifted left or right

4. **Code Generate ROM (CGROM)**

   It can be understood as the 8-bit encoding of actual character patterns. Almost the same as ASCII chart except some characters.

5. **Control Inputs**

   - RS, R/W

RS	R/W	Operation
0	0	Write data as a configuration instruction
0	1	Read data as busy flag (DB7) and address counter (DB6 to DB0)
1	0	Write data as a character to be displayed
1	1	Read data from the internal memory

   - E – data transfer enable
     a High to Low transition on E enables the data transfer

6. **Instruction table**

   Defines the inputs to perform specific instruction. Refer to the datasheet.

7. **Timing requirements**

   - Important, must be satisfied.
   - All instruction execution time must be satisfied, except
     - For 4-bit interface, two consecutive writes, one for high nibble, one for low nibble, need no delay in between, but need 40 μsec after
     - read instruction needs no time
   - Each write operation is enabled by a high to low transition on E input
     including the two consecutive nibble write

8. **Initialization**

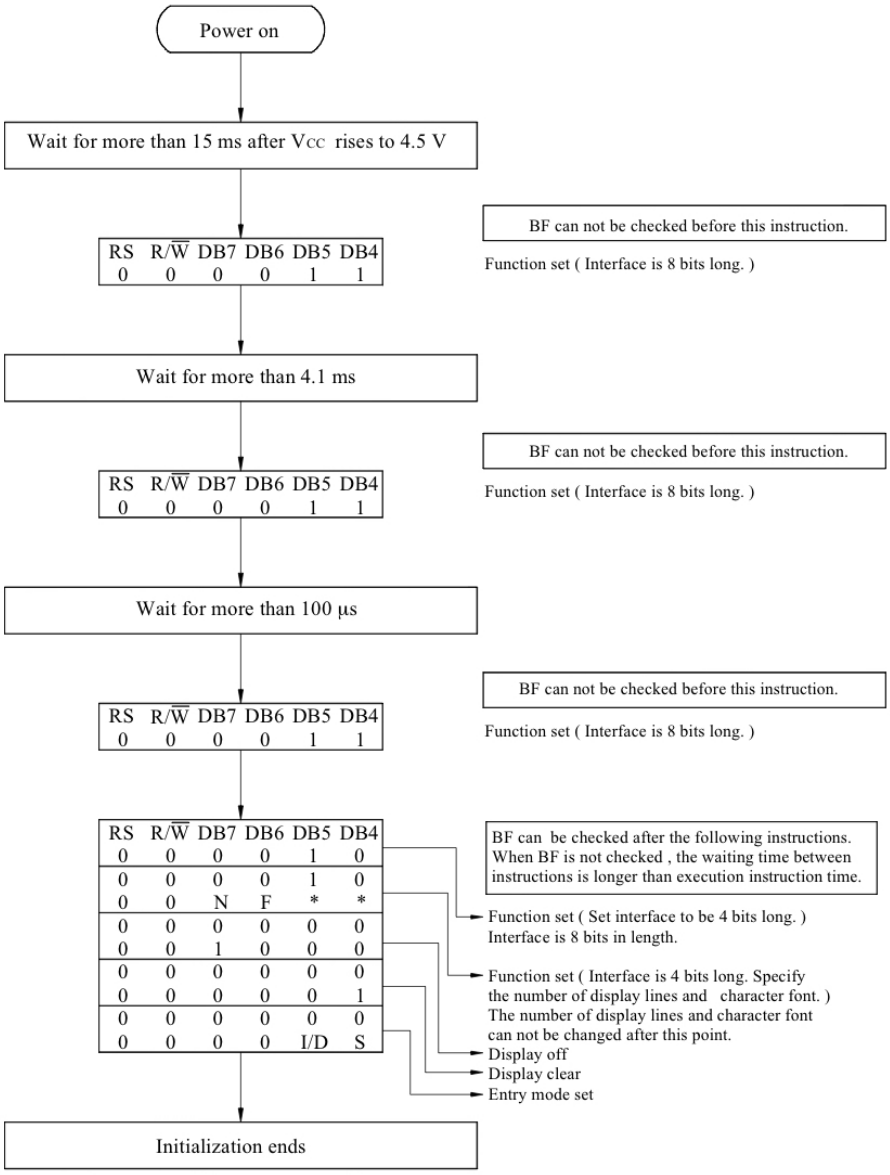


Figure 7: LCD 4-bit interface initialization

L7 — Power-saving Operations

1. **Static/Dynamic power**

   Dynamic power is comprised of switching and short-circuit power.

   Static power is comprised of leakage, or current that flows through the transistor when there is no activity.

$$P_{\text{dynamic}} = CV^2 f$$
$$P_{\text{static}} = VI_{\text{static}}$$

2. **Reduce power consumption**

   - Reduce operation frequency (lower frequency, lower power consumption)
   - Halt CPU or disable peripheral modules

3. **9 power saving modes**

   - CPU running

  - FRC RUN mode: CPU uses `FRC` clock source
  - LPRC RUN mode: CPU uses `LPRC` clock source
  - SOSC RUN mode: CPU uses `SOSC` clock source
  - Peripheral bus scaling mode: `PBCLK` is fraction of `SYSCLK`
- CPU halted
  - SLEEP mode: anything using `SYSCLK` (CPU and peripherals) halted, peripheral using other clock source are operating – lowest power consumption
  - POSC IDLE mode: Primary Oscillator
  - FRC IDLE mode: Fast RC Oscillator (8 MHz)
  - LPRC IDLE mode: Low-Power RC Oscillator (32 KHz)
  - SOSC IDLE mode: Secondary Oscillator (32.768 KHz)

4. **SLEEP mode**

- Characteristics
  CPU and most peripherals are halted
  System clock source is shut down
  Lowest power consumption
  Several peripherals alive to wake up CPU
- Enter SLEEP mode
  `SLPEN` (`OSCCON<4>`) bit set, followed by "wait" assembly instruction
- Exit sleep mode (wake up)
  By these events
  - Interrupt from operating peripheral
  - Watch dog timer
  - Reset

  May need start-up delay
  - Program not start running until clock signal detected stable
  - May report fail if delay is too long
- Example

```
SYSKEY = 0x0;              // Write invalid key to force lock
SYSKEY=0xAA996655;         // write Key1 to SYSKEY
SYSKEY=0x556699AA;         // Write Key2 to SYSKEY
OSCCONSET = 0x10;          // Set power-saving mode as SLEEP
SYSKEY = 0x0;              // Write invalid key to force lock

asm volatile ("wait");     // Put device in selected power-saving mode
                           // code execution will resume here after
                           // wake and the ISR is complete.
                           // "volatile" forces instruction location in
                           // the program

...user code after wake-up...
```

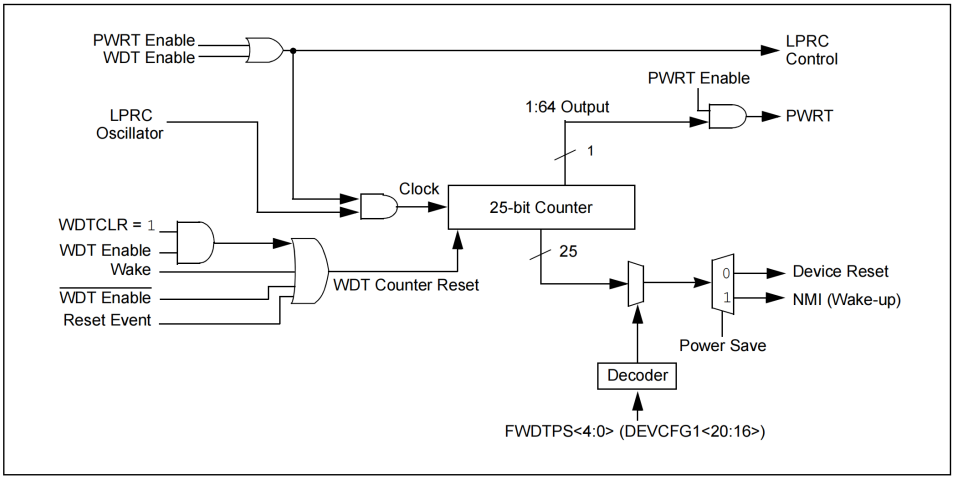
5. **IDLE mode**

- CPU is halted
  - System clock source is still running
  - Peripherals alive, unless individually configured to halt when in IDLE
  - Low start-up latency
  - Operating clock frequency may be slowed down to reduce power consumption
- Enter IDLE mode
  `SLPEN` (`OSCCON<4>`) bit cleared followed by "wait" assembly instruction, e.g.
  `OSCCONCLR = 0x10; asm ("wait");`

- Exit from IDLE mode
  - Interrupt from operating peripheral
  - Watch dog timer
  - Reset

6. **Wake-up CPU**
   - Peripheral interrupt
     - If current CPU priority is greater (IPL > RIPL), CPU remains halted System remains IDLE, or system enters IDLE from SLEEP
     - If requested peripheral IRQ priority is greater (RIPL > IPL) CPU services IRQ, then continues executing instruction following "wait"
   - WDT (watch dog timer) Non-Maskable Interrupt (NMI)

FIGURE 28-1: WATCHDOG TIMER AND POWER-UP TIMER BLOCK DIAGRAM



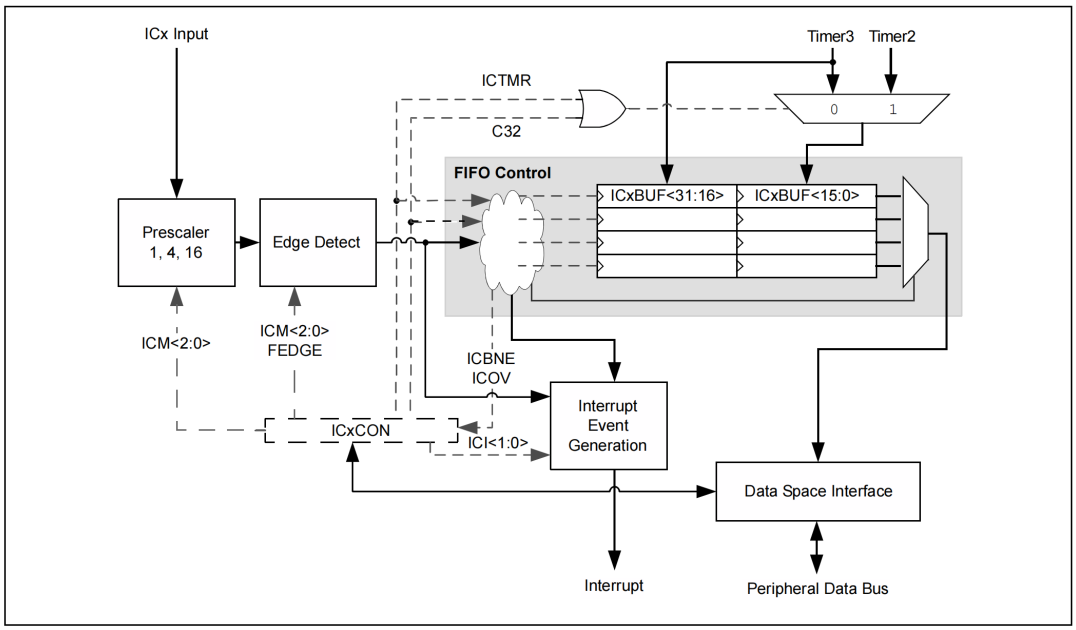
Also used to check software malfunction. Error if software not periodically clear WDT

L8 — Input Capture

The Input Capture module is useful in applications requiring frequency (period) and pulse measurement.

1. **Advantage (comparing to naive solution)**
   1. Accurate
   2. No preemption issue
   3. Don't bother CPU too much
2. **Block diagram**

FIGURE 15-1: INPUT CAPTURE BLOCK DIAGRAM



3. **Operation modes**

   - Simple capture event modes
     - Capture timer value on every falling edge
     - Capture timer value on every rising edge
     - Capture timer value on every edge, with specified starting edge (rising or falling)
   - Prescaled capture event modes
     - Capture timer value on every 4th rising edge
     - Capture timer value on every 16th rising edge
   - Edge detect mode
   - Interrupt-only mode

4. **Persistent interrupt**

   Interrupt is not cleared unless the interrupt condition is cleared.

Table 15-4: Interrupt Persistence Conditions

ICxCON Value	Set Condition	Persistence
ICI<1:0> = 11	Interrupt on every fourth capture event.	Interrupt is active if the number of FIFO entries is equal to 4.
ICI<1:0> = 10	Interrupt on every third capture event.	Interrupt is active if the number of FIFO entries is greater than or equal to 3.
ICI<1:0> = 01	Interrupt on every second capture event.	Interrupt is active if the number of FIFO entries is greater than or equal to 2.
ICI<1:0> = 00 or Edge Detect modes (see the ICM<2:0> bits in the ICxCON register (Register 15-1))	Interrupt on every capture event.	Interrupt is active if the number of FIFO entries is greater than or equal to 1.
ICOV = 1	Interrupt on fifth capture event if FIFO is full.	Interrupt is active until the error condition flag (ICxCON.ICOV) is cleared.

5. **Simple capture event modes**

   - 2–3 T_{PB} delay after the event due to synchronization
   - Capture input is sampled by PBCLK, therefore input signal high and low width $> T_{PB}$.

6. **Edge Detect Mode**
    - Prescaler and interrupt count not used.
    - Buffer overflow never signals.
    - Interrupt request on every capture.
7. **Interrupt-only mode**
    - Rising edge on ICx triggers an interrupt
        - Not functioning during normal operation
        - No timer capture
        - No buffer update
    - Used only as a wake-up mechanism for SLEEP or IDLE modes
8. **Capture buffer flags**
    - `ICBNE` (`ICxCON<3>`): IC buffer not empty
        - Read-only
        - Signals when 1 or more entries
    - `ICOV` (`ICxCON<4>`): IC buffer overflow
        - Signals on the 5th capture
        - All extra capture values are lost, until flag cleared
        - Flag cleared when
            * IC Module disabled
            * Module reset
            * `ICBNE` becomes 0 — IC buffer empty
9. **Capture event and interrupt event**
    - Capture event: capture change of `ICx` and store timer value in the FIFO.
    - Interrupt event: interrupt after certain amount of capture events.
10. **Examples**

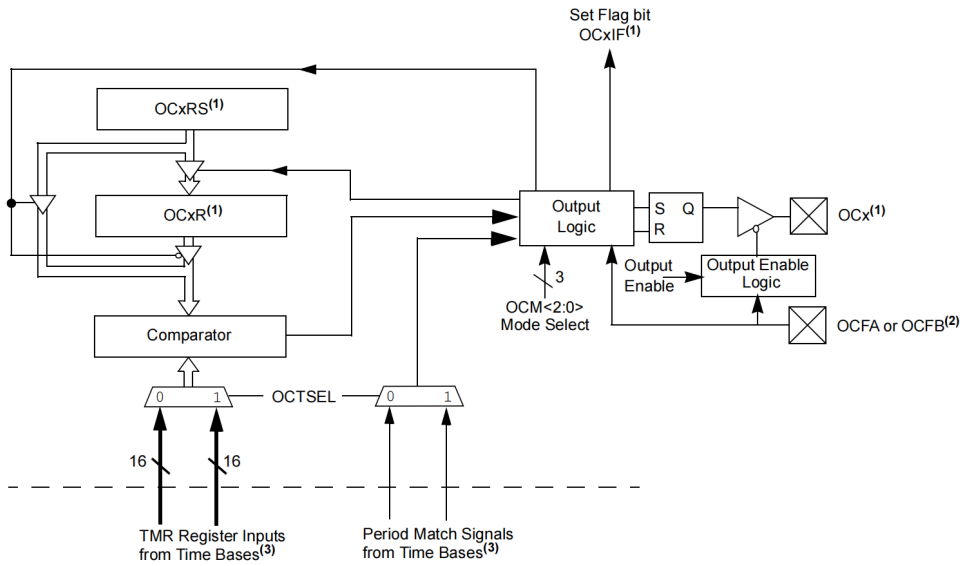
    See reference manual or slides.

L9 — Output Compare and PWM

The Output Compare module is used to generate a single pulse or a series of pulses in response to selected time base events.

1. **Block diagram**

FIGURE 16-1: OUTPUT COMPARE MODULE BLOCK DIAGRAM



Note 1: Where 'x' is shown, reference is made to the registers associated with the respective output compare channels, 1 through 5.
2: The OCFA pin controls the OC1-OC4 channels. The OCFB pin controls the OC5 channel.
3: Each output compare channel can use one of two selectable 16-bit time bases or a single 32-bit timer base.

2. **Operation mode**

   - Single Compare Match mode
     Drive high, drive low, toggle
   - Dual Compare Match mode
     Single output pulse, continuous output pulses
   - Simple Pulse Width Modulation (PWM) mode
     With or without fault protection input

3. **Single Compare Match mode**

   - Compare forces `OCx` pin high; initial state of pin is low. Interrupt is generated on the single compare match event.
   - Compare forces `OCx` pin low; initial state of pin is high. Interrupt is generated on the single compare match event.
   - Compare toggles `OCx` pin. Toggle event is continuous and an interrupt is generated for each toggle event.

4. **Dual Compare Match mode**

   Single or continues pulse.

5. **Single pulse special situations**

   - `PRy >= OCxRS > OCxR = TMRy = 0`
     The initial match of `OCxR` and `TMRy` at 0 doesn't drive high, work normally afterwards
   - `PRy >= OCxR >= OCxRS`
     Match `OCxR` first, match `OCxRS` in next counting round
   - `OCxRS > PRy >= OCxR`
     Only rising edge generated on first match, then signal remains high, no interrupt generated
   - `OCxR > PRy`
     Not working, `OCx` remains low

6. **PWM Signal (Pulse-width modulation)**

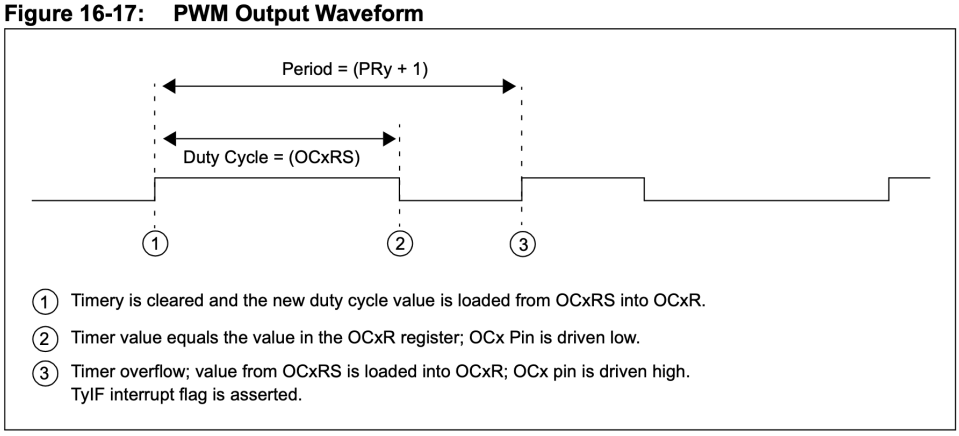
   Information is encoded into the signal width.

   - Advantages of PWM
     - Average value proportional to duty cycle
     - Low power used in transistors used to switch the signal
     - Fast switching possible due to MOSFETS and power transistors at speeds in excess of 100 kHz
     - **Digital signal is resistant to noise less**
     - Heat dissipated versus using resistors for intermediate voltage values
   - Disadvantages of PWM
     - Cost Complexity of circuit
     - Radio Frequency Interference
     - Voltage spikes
     - Electromagnetic noise

7. **PWM Operation Mode**

   In PWM mode, the `OCxR` register is a **read-only** slave duty cycle register and `OCxRS` is a buffer register that is written by the user to update the PWM duty cycle.

Figure 16-17: PWM Output Waveform



8. **Calculations**

$$T_{\text{PWM}} = (PR + 1) \times T_{\text{PB}} \times \text{Timer Prescale Value}$$

$$2^{\text{PWMResolution}} = \frac{T_{\text{PWM}}}{T_{\text{timer}}}$$

$$\text{PWMResolution (bits)} = \log_2 \left(\frac{F_{\text{PB}}}{F_{\text{PWM}} \times \text{Timer Prescale Value}} \right)$$

9. **Configuration**

   1. Set the PWM period by writing to the selected timer period register (`PRy`).
   2. Set the PWM duty cycle by writing to the `OCxRS` register.
   3. **Write the `OCxR` register with the initial duty cycle.**
   4. Enable interrupts, if required, for the timer and Output Compare modules. The output compare interrupt is required for PWM Fault pin utilization.
   5. Configure the Output Compare module for one of two PWM Operation modes by writing to the Output Compare mode bits, `OCM<2:0>` (`OCxCON<2:0>`).
   6. Set the `TMRy` prescale value and enable the time base by setting `TON` (`TxCON<15>`) `= 1`.