

## LỜI NÓI ĐẦU

Vi điều khiển ngày một được ứng dụng rộng rãi trong thực tế, nghiên cứu. Với chức năng vượt trội, tính ổn định cao, giá thành hợp lý Vi điều khiển Pic đang dần thay thế các vi điều khiển 8bits truyền thống.

Hiện nay tài liệu nói về dòng vi điều khiển Pic này có khá nhiều, nhưng chủ yếu là tiếng Anh còn Tiếng Việt thì hầu hết cũng chỉ dùng lại ở một phần của vi điều khiển làm người học khó lòng nắm bắt hết được các chức năng của Vi điều khiển Pic nhất là đối với những người mới học, mới tìm hiểu.

Nhằm giúp người học nắm bắt nhanh chóng, trọn vẹn được hết chức năng cơ bản của Vi điều khiển Pic, qua tài liệu trên mạng cùng với kinh nghiệm bản thân chúng tôi biên soạn lại cuốn “Giáo trình Vi điều khiển pic 16f877a” với hy vọng sẽ giúp đỡ các bạn học viên mới dễ dàng tiếp cận, nghiên cứu và nhanh chóng đi vào ứng dụng dòng vi điều khiển Pic 16f877A.

Giáo trình này chủ yếu biên soạn dựa vào datasheet con 16f877a kết hợp với những bài tập cơ bản, gắn liền lý thuyết đã được chạy thử nghiệm thực tế sẽ giúp học viên dễ dàng hiểu và tiếp cận.

Chắc chắn cuốn sách vẫn còn rất nhiều hạn chế, sai sót. Nhóm tác giả rất mong nhận được ý kiến từ các đồng nghiệp, bạn đọc để chúng tôi tiếp tục nâng cấp, phát triển thành giáo trình hoàn chỉnh hơn.

Mọi góp ý xin gửi về Khoa Cơ Khí, Bộ môn Cơ điện tử, Trường Đại học Nguyễn Tất Thành.

Xin cảm ơn!

NHÓM TÁC GIẢ

Tiêu đề	Trang
<b>LỜI NÓI ĐẦU.....</b>	<b>1</b>
<b>CHƯƠNG 1: TỔNG QUAN VỀ VI ĐIỀU KHIỂN PIC VÀ HỆ SỐ .....</b>	<b>4</b>
1.1. Giới thiệu về Vi điều khiển .....	4
1.2.Các họ Vi điều khiển thông dụng .....	5
1.3.Vi điều khiển Pic của Microchip .....	5
1.4.Ngôn ngữ lập trình cho Pic .....	8
1.5.Mạch nạp .....	9
1.6.Sơ đồ phần cứng cơ bản của Pic .....	10
1.7.Yêu cầu phần cứng thực hành .....	10
1.8.Các hệ thống số (binary number system .....	11
1.9.Mạch Logic cơ bản.....	12
<b>CHƯƠNG 2: NGÔN NGỮ LẬP TRÌNH C .....</b>	<b>15</b>
2.1. Cơ bản về ngôn ngữ lập trình C .....	15
2.2. Cấu trúc của một chương trình C .....	16
2.3.Kiểu dữ liệu .....	17
2.4.Khai báo biến và kiểu biến.....	17
2.5.Toán hạng .....	19
2.6.Statements .....	22
2.7.Lưu đồ giải thuật .....	26
<b>CHƯƠNG 3: GIỚI THIỆU VỀ PHẦN MỀM CCS FOR PIC .....</b>	<b>29</b>
3.1. Tổng quan về CCS .....	29
3.2.Tạo Project trong CCS.....	29
<b>CHƯƠNG 4: VI ĐIỀU KHIỂN 16F877A .....</b>	<b>42</b>
4.1. Cấu trúc vi điều khiển PIC16F877A.....	42
4.2. Bộ nhớ chương trình .....	47
4.3. Các cổng xuất nhập của PIC16F877A .....	53
4.4. Timer.....	66
4.5. ADC .....	77
4.6. Comparator .....	86
4.7. CCP .....	91
4.8. USART .....	100
4.9. MSSP .....	117
4.10. Tổng quan về một số đặc tính của CPU .....	148
<b>CHƯƠNG 5: MẠCH ĐIỆN ỨNG DỤNG .....</b>	<b>157</b>
5.1. Sơ đồ vi điều khiển, mạch nạp .....	157
5.2. Sơ đồ khối nguồn sử dụng LM2578 (5V-3A) .....	157
5.3. Sơ đồ kết nối led 7 đoạn, ADC, Còi .....	158
5.4. Khối Led đơn, Keypad 3x3, LCD1602 .....	158
5.5. Mạch dò line dùng trong robot dùng opamp .....	158
5.6. Mạch dò line trong robot dùng vi điều khiển: .....	159
5.7. Mạch điều khiển động cơ DC cầu H dùng IRF9540-540 .....	159

<b>5.8. Mạch điều khiển động cơ dùng Fet-Relay .....</b>	<b>160</b>
<b>5.9. Mạch điều khiển xilanh, led vẫy .....</b>	<b>160</b>
<b>5.10. Mạch điều khiển động cơ dùng ir2184 .....</b>	<b>161</b>
<b>5.10. Mạch cầu H Dung L298 .....</b>	<b>161</b>
<b>CHƯƠNG 6: BÀI TẬP ÚNG DỤNG .....</b>	<b>162</b>
<b>PHỤ LỤC:</b> .....	<b>226</b>
<b>def_877a.h .....</b>	<b>226</b>
<b>LCD_NVN.h .....</b>	<b>228</b>
<b>Hardware pic 16f877a .....</b>	<b>229</b>
<b>TÀI LIỆU THAM KHẢO: .....</b>	<b>230</b>

## CHƯƠNG I: TỔNG QUAN VỀ VI ĐIỀU KHIỂN PIC VÀ HỆ SỐ

### 1.1 Giới thiệu về Vi Điều Khiển

Vi điều khiển là một máy tính được tích hợp trên một chíp, nó thường được sử dụng để điều khiển các thiết bị điện tử.

Vi điều khiển, thực chất, là một hệ thống bao gồm một vi xử lý có hiệu suất đủ dùng và giá thành thấp (khác với các bộ vi xử lý đa năng dùng trong máy tính) kết hợp với các khối ngoại vi như bộ nhớ, các mô đun vào/ra, các mô đun biến đổi số sang tương tự và tương tự sang số,... Ở máy tính thì các mô đun thường được xây dựng bởi các chíp và mạch ngoài.

Vi điều khiển thường được dùng để xây dựng các hệ thống nhúng. Nó xuất hiện khá nhiều trong các dụng cụ điện tử, thiết bị điện, máy giặt, lò vi sóng, điện thoại, đầu đọc DVD, thiết bị đa phuong tiện, dây chuyền tự động, v.v. Bộ nhớ dữ liệu lưu trữ các dữ liệu tạm thời được sử dụng trong một chương trình và thường dễ bay hơi (ví dụ, dữ liệu bị mất sau khi có điện được tắt).

Hầu hết các vi điều khiển ngày nay được xây dựng dựa trên kiến trúc Harvard, kiến trúc này định nghĩa bốn thành phần cần thiết của một hệ thống nhúng. Những thành phần này là lõi CPU, bộ nhớ chương trình (thông thường là ROM hoặc bộ nhớ Flash), bộ nhớ dữ liệu (RAM), một hoặc vài bộ định thời và các cổng vào/ra để giao tiếp với các thiết bị ngoại vi và các môi trường bên ngoài - tất cả các khối này được thiết kế trong một vi mạch tích hợp. Vi điều khiển khác với các bộ vi xử lý đa năng ở chỗ là nó có thể hoạt động chỉ với vài vi mạch hỗ trợ bên ngoài.

#### **RAM:**

Bộ nhớ RAM, bộ nhớ truy cập ngẫu nhiên, là một mục đích chung thường lưu trữ dữ liệu người dùng trong một chương trình. Bộ nhớ RAM là dễ bay hơi trong ý nghĩa là nó không có thể giữ lại dữ liệu trong sự vắng mặt của nguồn (ví dụ, dữ liệu bị mất sau khi có điện được tắt). Hầu hết các vi điều khiển có một số lượng bộ nhớ RAM.

#### **ROM:**

Bộ nhớ ROM (read only memory), thường tổ chức chương trình hoặc các dữ liệu người dùng cố định. ROM được không bay hơi. Nếu nguồn được rút ra và sau đó cắm lại, các dữ liệu ban đầu sẽ vẫn ở đó. Bộ nhớ ROM được lập trình trong quá trình sản xuất, và người dùng không thể thay đổi nội dung của nó. Bộ nhớ ROM là chỉ hữu ích nếu bạn có phát triển một chương trình và muốn tạo ra một vài nghìn bản sao của nó.

### 1.2 Các vi điều khiển thông dụng

**Họ vi điều khiển AMCC** (do tập đoàn "Applied Micro Circuits Corporation" sản xuất). Từ tháng 5 năm 2004, họ vi điều khiển này được phát triển và tung ra thị trường bởi IBM: 403 PowerPC CPU, PPC 403GCX, 405 PowerPC CPU, PPC 405EP, PPC 405GP/CR, PPC 405GPr, PPC NP405H/L, 440 PowerPC Book-E CPU, PPC 440GP, PPC 440GX, PPC 440EP/EPx/GRx.

#### **Họ vi điều khiển Atmel**

- Dòng 8051 (8031, 8051, 8751, 8951, 8032, 8052, 8752, 8952)
- Dòng Atmel AT91 (Kiến trúc ARM THUMB)
- Dòng AT90, Tiny & Mega – AVR (Atmel Norway design)
- Dòng Atmel AT89 (Kiến trúc Intel 8051/MCS51)
- Dòng MARC4

#### **Họ vi điều khiển Cypress MicroSystems**

CY8C2xxxx (PSOC)

**Họ vi điều khiển Freescale Semiconductor.** Từ năm 2004, những vi điều khiển này được phát triển và tung ra thị trường bởi Motorola.

- Dòng 8-bit: 68HC05 (CPU05), 68HC08 (CPU08), 68HC11 (CPU11)
- Dòng 16-bit: 68HC12 (CPU12), 68HC16 (CPU16), Freescale DSP56800 (DSPcontroller)
- Dòng 32-bit: Freescale 683XX (CPU32), MPC500

MPC 860 (PowerQUICC), MPC 8240/8250 (PowerQUICC II), MPC 8540/8555/8560(PowerQUICC III)

#### **Họ vi điều khiển Fujitsu**

- F MC Family (8/16 bit)
- FR Family (32 bit)
- FR-V Family (32 bit RISC)

#### Họ vi điều khiển Intel

- Dòng 8-bit: 8XC42, MCS48, MCS51, 8061, 8xC251
- Dòng 16-bit: 80186/88, MCS96, MXS296
- Dòng 32-bit: 386EX, i960

#### Họ vi điều khiển Microchip

- PIC 8-bit (xử lý dữ liệu 8-bit, 8-bit data bus)
- Từ lệnh dài 12-bit (Base-line): PIC10F, PIC12F và một vài PIC16F
- Từ lệnh dài 14-bit (Mid-Range và Enhance Mid-Range): PIC16Fxxx, PIC16F1xxx
- Từ lệnh dài 16-bit (High Performance): PIC18F
- PIC 16-bit (xử lý dữ liệu 16-bit)
- PIC điều khiển động cơ: dsPIC30F
- PIC có DSC: dsPIC33F
- Phổ thông: PIC24F, PIC24E, PIC24H
- PIC 32-bit (xử lý dữ liệu 32-bit): PIC32MX

**Họ vi điều khiển National Semiconductor:** COP8, CR16

**Họ vi điều khiển STMicroelectronics:** ST 62, ST7, STM8, STM32 (Cortex-Mx)

**Họ vi điều khiển Philips Semiconductors:** LPC2000, LPC900, LPC700

### 1.3 Vi điều khiển PIC của Microchip

#### 1.3.1. Đặc tính chung:

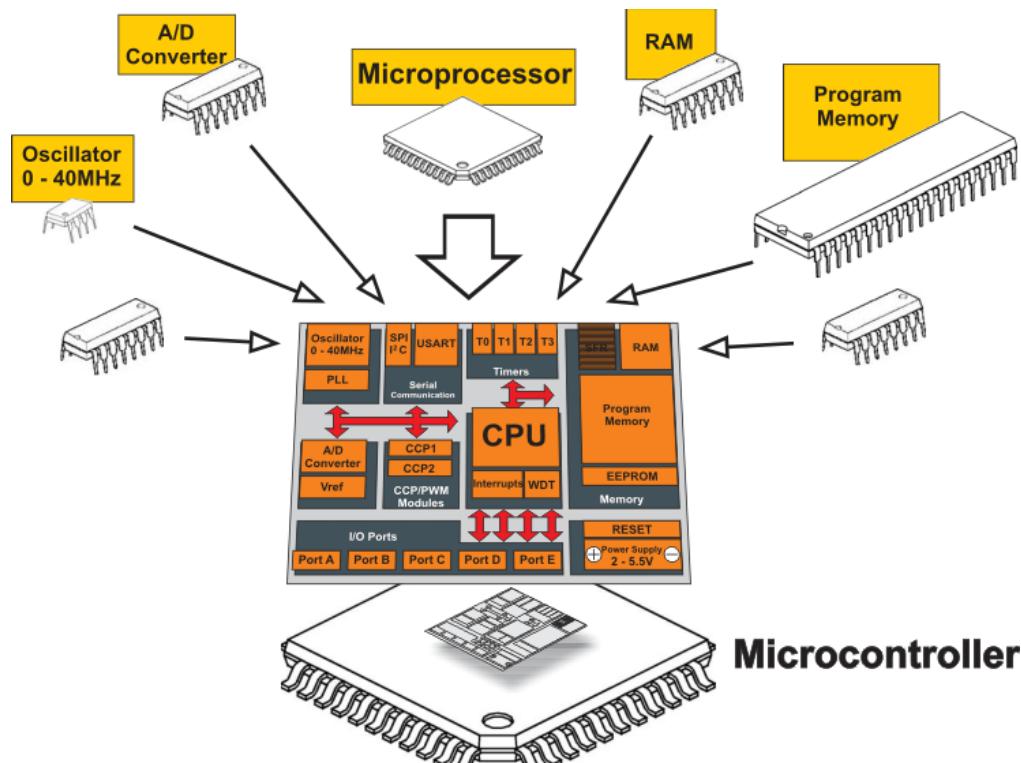
Pic là một họ Vi Điều Khiển RISC được sản xuất bởi công ty Microchip Technology. Dòng Pic đầu tiên là PIC 1650 được phát triển bởi Microelectronics thuộc General Instrument. Dòng PIC 8 bits được sản xuất vào năm 1975.

Pic là viết tắt của “Programable Intelligent Computer”, có thể dịch là “máy tính thông minh khả trinh” do hãng Gerenal Instrument đặt tên cho vi điều khiển đầu tiên của họ: PIC 1650 được thiết kế để dùng làm các thiết bị ngoại vi cho vi điều khiển CP1600. Vi điều khiển này sau đó được nghiên cứu phát triển thêm và từ đó hình thành nên dòng vi điều khiển PIC sau này.

Hiện nay có khá nhiều dòng vi điều khiển PIC khác nhau nhưng chúng có cùng chung 1 số đặc điểm sau:

- ✓ Sử dụng công nghệ tích hợp cao RISC CPU.
- ✓ Người sử dụng có thể lập trình với 35 câu lệnh đơn giản.
- ✓ Tất cả các câu lệnh thực hiện trong một chu kỳ lệnh ngoại trừ 1 số câu lệnh rẽ nhánh thực hiện 2 chu kỳ lệnh.
- ✓ Tốc độ hoạt động: Xung đồng bộ vào/4.
- ✓ Bộ nhớ chương trình Flash 8K?\*14Words.
- ✓ Bộ nhớ Ram 368\*8bytes.
- ✓ Bộ nhớ EEPROM 256\*8 bytes.
- Khả năng của bộ vi xử lý:
  - ✓ Khả năng ngắt lên tới 14 nguồn ngắt trong và ngoài.
  - ✓ Ngăn nhớ Stack được phân chia 8 mức.
  - ✓ Truy cập bộ nhớ bằng địa chỉ trực tiếp hoặc gián tiếp.
  - ✓ Nguồn khởi động lại (POR).
  - ✓ Bộ tạo xung thời gian (PWRT) và bộ tạo dao động (OST).
  - ✓ Bộ đếm xung thời gian (WDT) với nguồn dao động trên chíp (nguồn dao động RC) hoạt động đáng tin cậy.
  - ✓ Có mã chương trình bảo vệ.
  - ✓ Phương thức tiết kiệm điện SLEEP.
  - ✓ Có bảng lựa chọn dao động.
  - ✓ Công nghệ CMOS FLASH/EEPROM nguồn mức thấp, tốc độ cao.
  - ✓ Thiết kế hoàn toàn chỉnh.
  - ✓ Dải điện thế hoạt động 2.0 v đến 5.5 v.

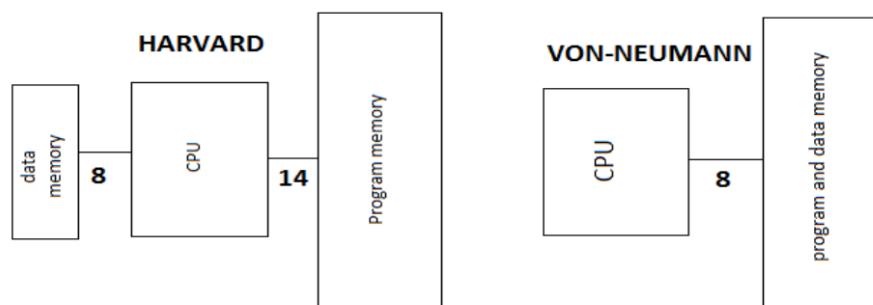
- ✓ Nguồn sử dụng hiện tại 25 mA.
- ✓ Dãy nhiệt động hoạt động công nghiệp.
- ✓ Công suất tiêu thụ thấp.
  - < 0.6 mA Với 5V - 4MHz
  - < 20uA với nguồn 32KHz
  - < 1uA với nguồn dự phòng
- Các đặc tính nổi bật của các thiết bị ngoại vi trên chip:
  - ✓ Timer 0: 8 bit của bộ định thời, bộ đếm với hệ số tỷ lệ trước.
  - ✓ Timer 1: 16 bit của bộ định thời, bộ đếm với hệ số tỉ lệ, có khả năng tăng trong khi ở chế độ sleep qua xung đồng hồ cung cấp cho bên ngoài.
  - ✓ Timer 2: 8 bit của bộ định thời, bộ đếm 8 bit của hệ số tỉ lệ trước và hệ số tỉ lệ sau.
  - ✓ Có 2 chế độ bắt giữa, so sánh, điều chế độ rộng xung (PWM).
  - ✓ Chế độ bắt giữa với 16 bit với tốc độ 12,5ns, chế độ so sánh với 16 bit, tốc độ giải quyết cực đại 200ns, chế độ rộng xung 10 bit.
  - ✓ Bộ chuyển đổi tín hiệu tương tự sang số 10 bit.
  - ✓ Cổng truyền thông nối tiếp SSP với SPI phương thức I<sup>2</sup>C (Chủ/Tớ).
  - ✓ Bộ truyền thông tin đồng bộ, di động (USART/SCL) có khả năng phát hiện 9 bit địa chỉ.
  - ✓ Cổng phụ song song (PSP) với 8 bit mở rộng, với RD, WR và CS điều khiển.



Hình 1.1. Tổng quan vi điều khiển PIC

### 1.3.2. Kiến trúc của PIC

Cấu trúc của một vi điều khiển được thiết kế theo hai dạng kiến trúc: Von Neuman và Havard.



Hình 1.2. Kiến trúc Havard và Von Neuman

Tổ chức phần cứng của PIC được thiết kế theo kiến trúc Havard. Điểm khác biệt giữa kiến trúc Havard và kiến trúc Von-Neuman là cấu trúc bộ nhớ dữ liệu và bộ nhớ chương trình. Đối với kiến trúc Von-Neuman, bộ nhớ dữ liệu và bộ nhớ chương trình nằm chung trong một bộ nhớ, do đó ta có thể tổ chức, cân đối một cách linh hoạt bộ nhớ chương trình và bộ nhớ dữ liệu. Tuy nhiên điều này chỉ có ý nghĩa khi tốc độ xử lý của CPU phải rất cao, vì với cấu trúc đó, trong cùng một thời điểm CPU chỉ có thể tương tác với bộ nhớ dữ liệu hoặc bộ nhớ chương trình. Như vậy có thể nói kiến trúc Von-Neuman không thích hợp với cấu trúc của một vi điều khiển. Đối với kiến trúc Havard, bộ nhớ dữ liệu và bộ nhớ chương trình tách ra thành hai bộ nhớ riêng biệt. Do đó trong cùng một thời điểm CPU có thể tương tác với cả hai bộ nhớ, như vậy tốc độ xử lý của vi điều khiển được cải thiện đáng kể. Một điểm cần chú ý nữa là tập lệnh trong kiến trúc Havard có thể được tối ưu tùy theo yêu cầu kiến trúc của vi điều khiển mà không phụ thuộc vào cấu trúc dữ liệu. Ví dụ, đối với vi điều khiển dòng 16F, độ dài lệnh luôn là 14 bit (trong khi dữ liệu được tổ chức thành từng byte), còn đối với kiến trúc Von-Neuman, độ dài lệnh luôn là bội số của 1 byte (do dữ liệu được tổ chức thành từng byte). Đặc điểm này được minh họa cụ thể trong hình 4.1.

### **1.3.3. RISC và CISC**

Như đã trình bày ở trên, kiến trúc Havard là khái niệm mới hơn so với kiến trúc Von-Neuman. Khái niệm này được hình thành nhằm cải tiến tốc độ thực thi của một vi điều khiển. Qua việc tách rời bộ nhớ chương trình và bộ nhớ dữ liệu, bus chương trình và bus dữ liệu, CPU có thể cùng một lúc truy xuất cả bộ nhớ chương trình và bộ nhớ dữ liệu, giúp tăng tốc độ xử lý của vi điều khiển lên gấp đôi. Đồng thời cấu trúc lệnh không còn phụ thuộc vào cấu trúc dữ liệu nữa mà có thể linh động điều chỉnh tùy theo khả năng và tốc độ của từng vi điều khiển. Và để tiếp tục cải tiến tốc độ thực thi lệnh, tập lệnh của họ vi điều khiển PIC được thiết kế sao cho chiều dài mã lệnh luôn cố định (ví dụ đối với họ 16Fxxxx chiều dài mã lệnh luôn là 14 bit) và cho phép thực thi lệnh trong một chu kỳ của xung clock (ngoại trừ một số trường hợp đặc biệt như lệnh nhảy, lệnh gọi chương trình con ... cần hai chu kỳ xung đồng hồ). Điều này có nghĩa tập lệnh của vi điều khiển thuộc cấu trúc Havard sẽ ít lệnh hơn, ngắn hơn, đơn giản hơn để đáp ứng yêu cầu mã hóa lệnh bằng một số lượng bit nhất định. Vi điều khiển được tổ chức theo kiến trúc Havard còn được gọi là vi điều khiển RISC (Reduced Instruction Set Computer) hay vi điều khiển có tập lệnh rút gọn. Vi điều khiển được thiết kế theo kiến trúc Von-Neuman còn được gọi là vi điều khiển CISC (Complex Instruction Set Computer) hay vi điều khiển có tập lệnh phức tạp vì mã lệnh của nó không phải là một số cố định mà luôn là bội số của 8 bit (1 byte).

### **1.3.4. Các dòng PIC và cách chọn Vi Điều Khiển PIC.**

Các kí hiệu của vi điều khiển PIC:

PIC12xxxx: độ dài lệnh 12 bit

PIC16xxxx: độ dài lệnh 14 bit

PIC18xxxx: độ dài lệnh 16 bit

DSPIC30, dsPIC33, PIC32 ...

C: PIC có bộ nhớ EPROM (chỉ có 16C84 là EEPROM)

F: PIC có bộ nhớ flash

LF: PIC có bộ nhớ flash hoạt động ở điện áp thấp

LV: tương tự như LF, đây là kí hiệu cũ

Bên cạnh đó một số vi điều khiển có kí hiệu xxFxxx là EEPROM, nếu có thêm chữ A ở cuối là flash (ví dụ PIC16F877 là EEPROM, còn PIC16F877A là flash). Ngoài ra còn có thêm một dòng vi điều khiển PIC mới là dsPIC.

Ở Việt Nam phổ biến nhất là các họ vi điều khiển PIC do hãng Microchip sản xuất. Cách lựa chọn một vi điều khiển PIC phù hợp:

Trước hết cần chú ý đến số chân của vi điều khiển cần thiết cho ứng dụng. Có nhiều vi điều khiển PIC với số lượng chân khác nhau, thậm chí có vi điều khiển chỉ có 8 chân, ngoài ra còn có các vi điều khiển 28, 40, 44, ... chân. Cần chọn vi điều khiển PIC có bộ nhớ flash để có thể nạp xóa chương trình được nhiều lần hơn. Tiếp theo cần chú ý đến các khái niệm được tích hợp sẵn trong vi điều khiển, các chuẩn giao tiếp bên trong. Sau cùng cần chú ý đến bộ nhớ chương trình mà vi điều khiển cho phép. Ngoài ra mọi thông tin về cách lựa chọn vi điều khiển

PIC có thể được tìm thấy trong cuốn sách “Select PIC guide” do nhà sản xuất Microchip cung cấp.

<b>Family</b>	<b>ROM [Kbytes]</b>	<b>RAM [bytes]</b>	<b>Pins</b>	<b>Clock Freq. [MHz]</b>	<b>A/D Inputs</b>	<b>Resolution of A/D Converter</b>	<b>Comparators</b>	<b>8/16-bit Timers</b>	<b>Serial Comm.</b>	<b>PWM Outputs</b>	<b>Others</b>
<b>Base-Line 8-bit architecture, 12-bit Instruction Word Length</b>											
PIC10FXXX	0.375 - 0.75	16 - 24	6 - 8	4 - 8	0 - 2	8	0 - 1	1 x 8	-	-	-
PIC12FXXX	0.75 - 1.5	25 - 38	8	4 - 8	0 - 3	8	0 - 1	1 x 8	-	-	EEPROM
PIC16FXXX	0.75 - 3	25 - 134	14 - 44	20	0 - 3	8	0 - 2	1 x 8	-	-	EEPROM
PIC16HVXXX	1.5	25	18 - 20	20	-	-	-	1 x 8	-	-	Vdd = 15V
<b>Mid-Range 8-bit architecture, 14-bit Instruction Word Length</b>											
PIC12FXXX	1.75 - 3.5	64 - 128	8	20	0 - 4	10	1	1 - 2 x 8 1 x 16	-	0 - 1	EEPROM
PIC12HVXXX	1.75	64	8	20	0 - 4	10	1	1 - 2 x 8 1 x 16	-	0 - 1	-
PIC16FXXX	1.75 - 14	64 - 368	14 - 64	20	0 - 13	8 or 10	0 - 2	1 - 2 x 8 1 x 16	USART I2C SPI	0 - 3	-
PIC16HVXXX	1.75 - 3.5	64 - 128	14 - 20	20	0 - 12	10	2	2 x 8 1 x 16	USART I2C SPI	-	-
<b>High-End 8-bit architecture, 16-bit Instruction Word Length</b>											
PIC18FXXX	4 - 128	256 - 3936	18 - 80	32 - 48	4 - 16	10 or 12	0 - 3	0 - 2 x 8 2 - 3 x 16	USB2.0 CAN2.0 USART I2C SPI	0 - 5	-
PIC18FXXJXX	8 - 128	1024 - 3936	28 - 100	40 - 48	10 - 16	10	2	0 - 2 x 8 2 - 3 x 16	USB2.0 USART Ethernet I2C SPI	2 - 5	-
PIC18FXXKXX	8 - 64	768 - 3936	28 - 44	64	10 - 13	10	2	1 x 8 3 x 16	USART I2C SPI	2	-

Hình I.3. Một số thông số chip PIC

#### 1.4. Ngôn ngữ lập trình cho PIC

Ngôn ngữ lập trình cho PIC rất đa dạng. Ngôn ngữ lập trình cấp thấp có MPLAB (được cung cấp miễn phí bởi nhà sản xuất Microchip), các ngôn ngữ lập trình cấp cao hơn bao gồm C, Basic, Pascal, ... Ngoài ra còn có một số ngôn ngữ lập trình được phát triển dành riêng cho PIC như PICBasic, MikroBasic, CCS...

Trong giáo trình này chúng ta sẽ lập trình bằng ngôn ngữ C và dựa trên phần mềm chuyên dụng cho PIC là CCS.

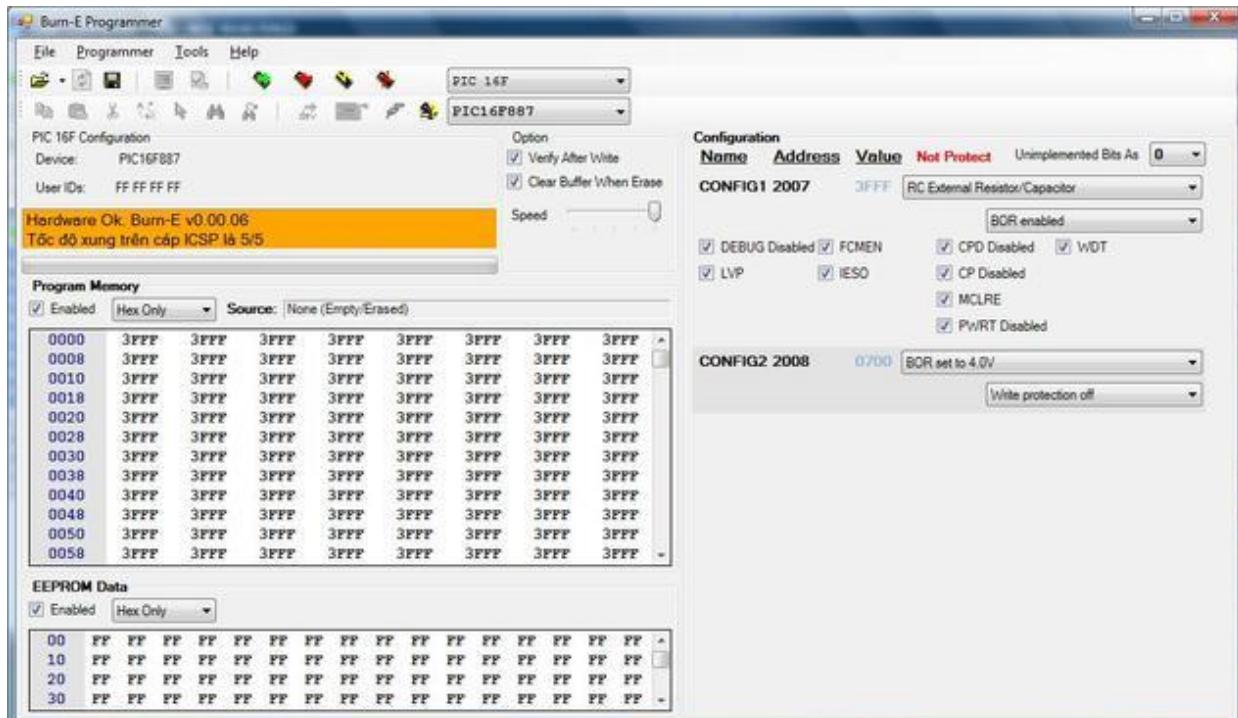
#### 1.5. Mạch nạp:

Đây cũng là một dòng sản phẩm rất đa dạng dành cho vi điều khiển PIC. Có thể sử dụng các mạch nạp được cung cấp bởi nhà sản xuất là hãng Microchip như: PICSTART plus, MPLAB ICD 2, MPLAB PM 3, PRO MATE II. Có thể dùng các sản phẩm này để nạp cho vi điều khiển khác thông qua chương trình MPLAB. Dòng sản phẩm chính thống này có ưu thế là nạp được cho tất cả các vi điều khiển PIC, tuy nhiên giá thành rất cao và thường gấp rất nhiều khó khăn trong quá trình mua sản phẩm. Ngoài ra do tính năng cho phép nhiều chế độ nạp khác nhau, còn có rất nhiều mạch nạp được thiết kế dành cho vi điều khiển PIC. Có thể sơ lược một số mạch nạp cho PIC như sau:

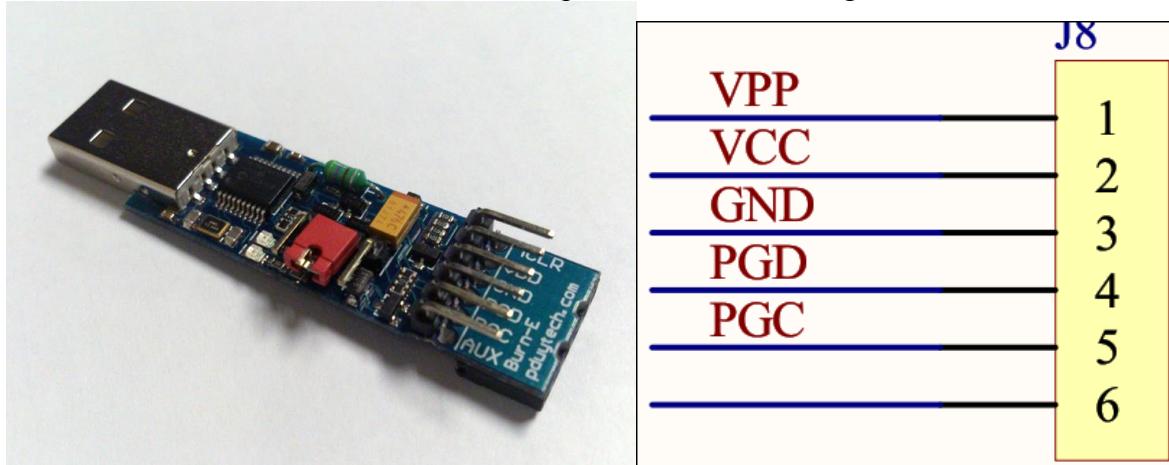
- ✓ JDM programmer: mạch nạp này dùng chương trình nạp Icprog cho phép nạp các vi điều khiển PIC có hỗ trợ tính năng nạp chương trình điện áp thấp ICSP (In Circuit Serial Programming). Hầu hết các mạch nạp đều hỗ trợ tính năng nạp chương trình này.
- ✓ WARP-13A và MCP-USB: hai mạch nạp này giống với mạch nạp PICSTART PLUS do nhà sản xuất Microchip cung cấp, tương thích với trình biên dịch MPLAB, nghĩa là ta có thể trực tiếp dùng chương trình MPLAB để nạp cho vi điều khiển PIC mà không cần sử dụng một chương trình nạp khác, chẳng hạn như ICprog.

- ✓ P16PRO40: mạch nạp này do Nigel thiết kế và cũng khá nổi tiếng. Ông còn thiết kế cả chương trình nạp, tuy nhiên ta cũng có thể sử dụng chương trình nạp Icprog.
- ✓ Burn-E Programmer: Là mạch nạp do Pduytech nghiên cứu và phát triển từ phần cứng đến phần mềm. Qua quá trình sử dụng cho thấy độ ổn định, tốc độ nạp nhanh, hỗ trợ nhiều vi điều khiển, phần mềm dễ dàng sử dụng và luôn được cập nhật, giá cả lại rất hợp túi tiền nên tôi khuyên sử dụng mạch nạp này.

Xem thêm thông tin tại <http://www.pduytech.com/ProductsBurnerBurn-E.html>



Hình 1.4. Giao diện phần mềm Burn-E Programmer

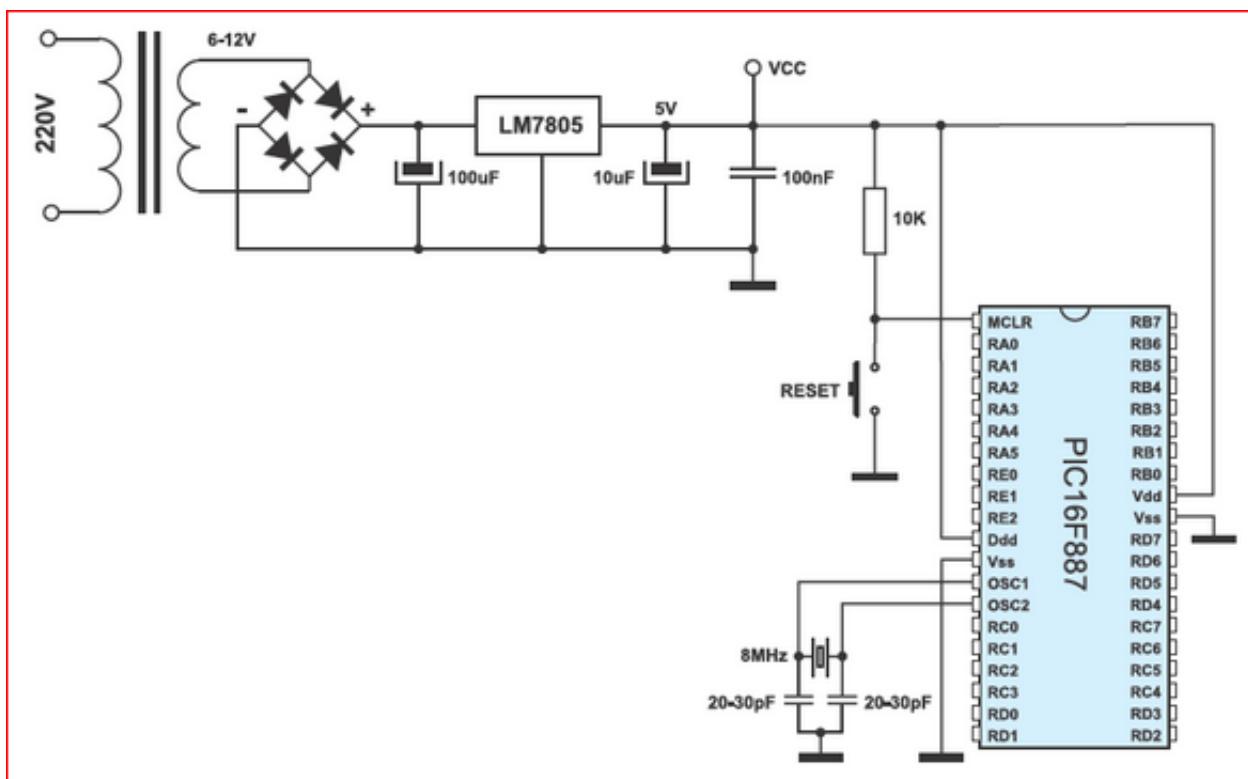


Hình 1.5. Mạch nạp Burn-E Programmer và sơ đồ mạch kết nối với VDK qua chuẩn ICSP Connection



Hình 1.6. Quá trình viết chương trình cho PIC.

### 1.6. Sơ đồ phần cứng cơ bản của Pic:

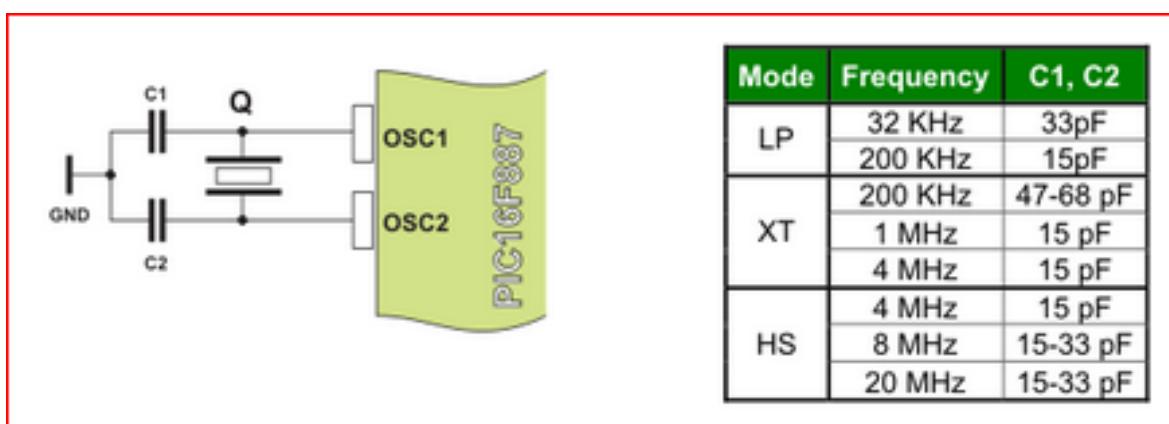


Hình 1.7. Sơ đồ cơ bản của pic

- ✓ Lựa chọn tụ điện thanh anh

Việc lựa chọn Tụ Điện cho Thạch Anh sẽ rất quan trọng, nó giúp cho Vi điều khiển hoạt động ổn định hơn, tránh những sai sót, mất thời gian trong quá trình thử nghiệm.

- ✓ Tân số xung nhịp lấy theo xung thạch anh. Nếu tần số thạch anh=20MHz thì xung nhịp (xung clock) của vi điều khiển = $20\text{MHz}/4=5\text{MHz}$  tức là xung nhịp này có chu kì là 0.2  $\mu\text{s}$ .



Hình 1.8. Thông số thạch anh và tụ.

### 1.7. Yêu cầu phần cứng thực hành:

Để nhanh chóng nắm bắt được cách lập trình C cho Pic thông qua trình dịch CCS ngoài vấn đề mô phỏng trên phần mềm Proteus chúng ta cần trang bị cho mình một Module để tự thực hành. Các bài tập xoay quanh các vấn đề cơ bản: Nháy Led, đọc dữ liệu bàn phím, quét led 7 đoạn, LCD, đọc Encoder, Timer... và giao tiếp với các thiết bị ngoại vi như Real Time IC, EEPROM....

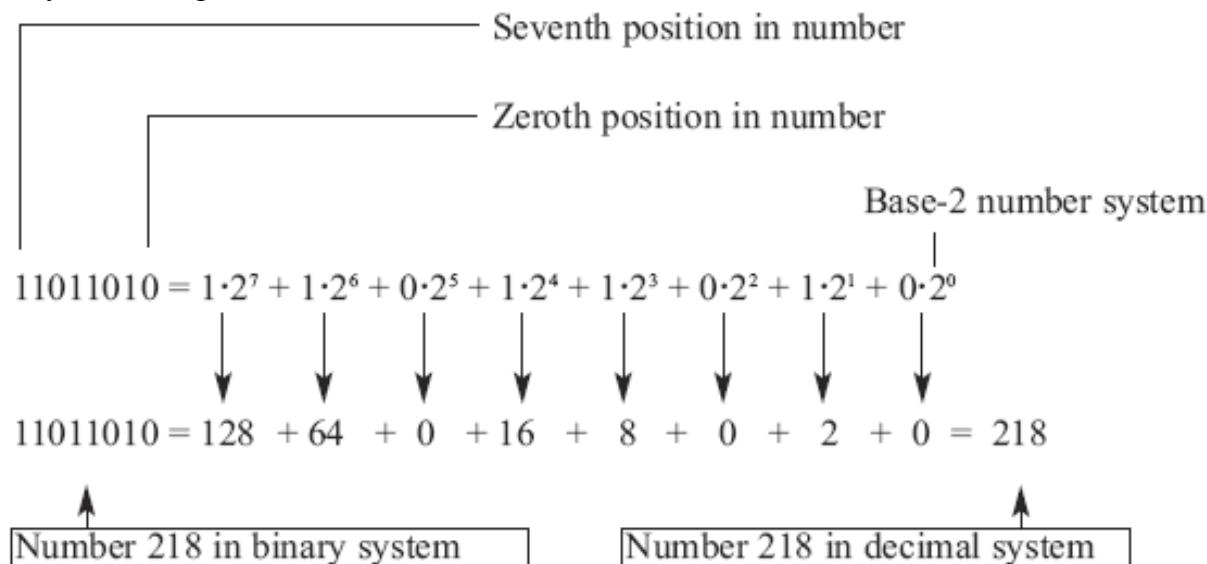
- **Yêu cầu về phần cứng tối thiểu cần có để thực hành:**
- PIC16F877A.
- 1 Board cắm linh kiện.

- Thạch anh 20MHz, tụ 22pF, 10uF, trở 10K, 4K7, 330Ω, nút bấm.
- 10 LED đơn xanh hay đỏ, 4 LED 7 thanh (loại 4 LED liền một đế ).
- MAX232 để giao tiếp máy tính () .
- **Phần cứng mở rộng**
- LCD 1602A loại 2 dòng 16 ký tự (Nếu có LCD 2002 càng tốt).
- Real Time IC DS1307 hay DS1337.
- EEPROM AT24Cxx
- Sensor nhiệt LM335 hay LM35.
- Động cơ bước, động cơ một chiều.

### **1.8. Các hệ thống số (binary number system):**

#### **1.8.1. Hệ thống số nhị phân:**

Hệ nhị phân (hay hệ đếm cơ số hai) là một hệ đếm dùng hai ký tự để biểu đạt một giá trị số, bằng tổng số các lũy thừa của 2. Hai ký tự đó thường là 0 và 1; chúng thường được dùng để biểu đạt hai giá trị hiệu điện thế tương ứng (có hiệu điện thế, hoặc hiệu điện thế cao là 1 và không có, hoặc thấp là 0). Do có ưu điểm tính toán đơn giản, dễ dàng thực hiện về mặt vật lý, chẳng hạn như trên các mạch điện tử, hệ nhị phân trở thành một phần kiến tạo căn bản trong các máy tính đương thời.



Hình I.9. Biểu diễn số 128 dưới dạng nhị phân và thập phân.

#### **1.8.2. Hệ thống số Thập lục phân (Hexadecimal Number System):**

Trong toán học và trong khoa học điện toán, hệ thập lục phân (hay hệ đếm cơ số 16, tiếng Anh: hexadecimal), hoặc chỉ đơn thuần gọi là thập lục, là một hệ đếm có 16 ký tự, từ 0 đến 9 và A đến F (chữ hoa và chữ thường như nhau). Hệ thống thập lục phân hiện dùng, được công ty IBM giới thiệu với thế giới điện toán vào năm 1963. Một phiên bản cũ của hệ thống này, dùng các con số từ 0 đến 9, và các con chữ A đến Z, đã được sử dụng trong máy tính Bendix G-15, ra mắt năm 1956.

Ví dụ, số thập phân 79, với biểu thị nhị phân là 01001111, có thể được viết thành 4F trong hệ thập lục phân (4 = 0100, F = 1111).

#### **1.8.3. Chuyển đổi từ số nhị phân sang số thập phân:**

a) Chuyển đổi số thập lục phân sang số thập phân (Hexadecimal to Decimal Number Conversion):

Để chuyển đổi một số Hexa sang Decimal ta sẽ lấy tổng của từng số nhân 16.

Ví dụ:

A37E (number in hexadecimal system)

$$\begin{array}{rcl}
 & 14 \cdot 16^0 = 14 \cdot 1 & = 14 \\
 & 7 \cdot 16^1 = 7 \cdot 16 & = 112 \\
 & 3 \cdot 16^2 = 3 \cdot 256 & = 768 \\
 & 10 \cdot 16^3 = 10 \cdot 4096 & = 40960
 \end{array}$$

41854 (same number in decimal system)

- b) Chuyển đổi một số thập lục phân sang số nhị phân (Hexadecimal to Binary Number Conversion):

Để chuyển đổi một số thập lục phân sang số nhị phân ta chuyển đổi từng số sang số nhị phân.

Ví dụ:

$$\begin{array}{c}
 E4 = 11100100 \\
 \hline
 | \quad | \\
 E \quad 4
 \end{array}$$

#### 1.8.4. Bit:

Bit là đơn vị thông tin. Bit có thể nhận 2 giá trị: 0 hoặc 1. Nó có thể được biểu diễn theo nhiều cách khác nhau. Có thể là trạng thái đóng hay mở của mạch điện, một vết khắc bằng tia laser trên bề mặt đĩa CD v.v... Các bit có thể dùng để thể hiện số tự nhiên trong hệ nhị phân.

#### 1.8.5. Byte:

Một dãy số liền nhau của một số bit cố định. Trong đại đa số các máy tính hiện đại, byte có 8 bit (octet). Tuy nhiên, không phải máy nào đều dùng byte có 8 bit. Một số máy tính đời cũ đã dùng 6, 7, hay 9 bit trong một byte - một ví dụ là trong cấu trúc 36 bit của bộ máy PDP-10. Một ví dụ khác là đơn vị slab của bộ máy NCR-315. Một byte luôn luôn không chia rời được, nó là đơn vị nhỏ nhất có thể truy nhập được. Một byte 8 bit có thể lưu trữ được 256 giá trị khác nhau ( $2^8 = 256$ ) -- đủ để lưu trữ một số nguyên không dấu từ 0 đến 255, hay một số có dấu từ -128 đến 127, hay một ký tự dùng mã 7 bit (như ASCII) hay 8 bit.

Một byte chia làm 2 nữa “trái và phải” người ta gọi là “high” and “low”



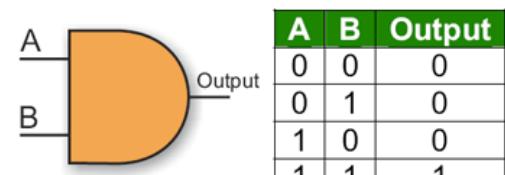
#### 1.9. Mạch Logic cơ bản:

##### 1.9.1. And gate:

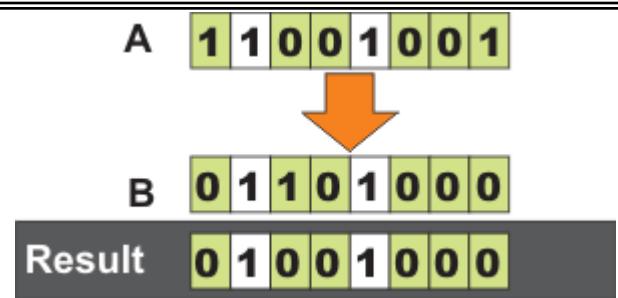
Cổng AND là một cổng logic dùng để thực hiện hàm AND hai hay nhiều biến. Cổng AND có một ngõ vào tùy thuộc số biến và một ngõ ra. Ngõ ra của cổng là hàm AND của các biến ngõ vào. Bên phải là bảng chân trị mô tả hoạt động của cổng AND 2 ngõ vào A và B.

Ngõ ra cổng AND chỉ ở mức cao (1) khi tất cả các ngõ vào ở mức cao (1).

Khi có một ngõ vào ở mức thấp (0) thì ngõ ra luôn ở mức thấp (0) bất chấp các ngõ vào còn lại.



Khi dùng trong lập trình nó được thực hiện theo chương trình. Khi đó nó sẽ đủ để nhớ logic AND trong một chương trình đến đúng với bits của 2 thanh ghi.

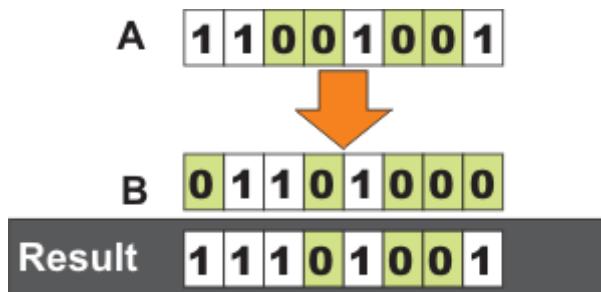
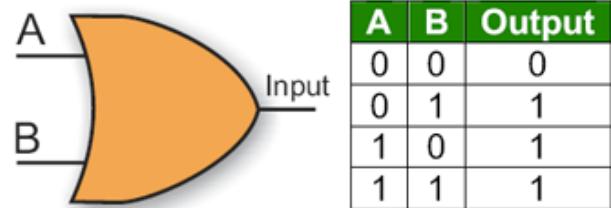


#### 1.9.2. OR Gate:

Cổng OR là một cổng logic dùng để thực hiện hàm OR hai hay nhiều biến. Cổng OR có một ngõ vào tùy thuộc số biến và một ngõ ra. Ngõ ra của cổng là hàm OR của các biến ngõ vào. Bên phải là bảng chân trị mô tả hoạt động của cổng OR 2 ngõ vào A và B.

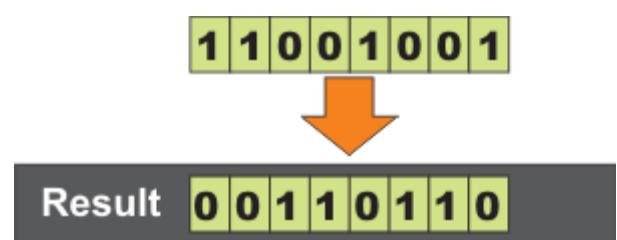
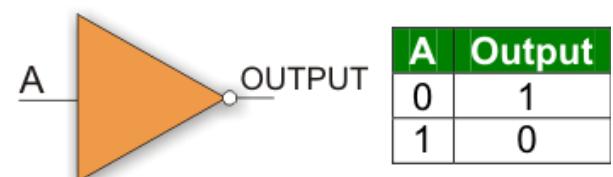
Ngõ ra cổng OR chỉ ở mức thấp (0) khi tất cả các ngõ vào ở mức thấp (0).

Khi có một ngõ vào ở mức cao (1) thì ngõ ra luôn ở mức cao (1) bất chấp các ngõ vào còn lại.



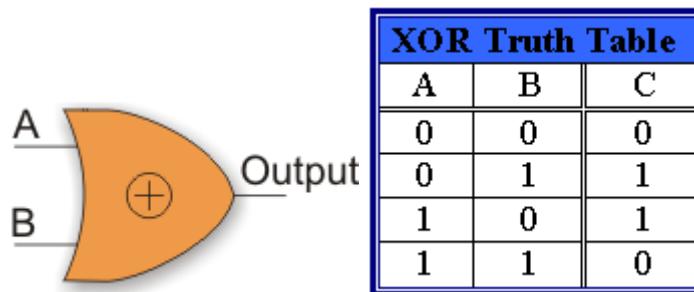
#### 1.9.3. Not Gate:

Cổng logic này chỉ có một ngõ vào và 1 ngõ ra. Khi logic vào mức thấp (0) thì cổng logic ra mức cao (1) và ngược lại. Cổng này dùng để đảo tín hiệu của chính nó và thường được gọi là bộ biến đổi.



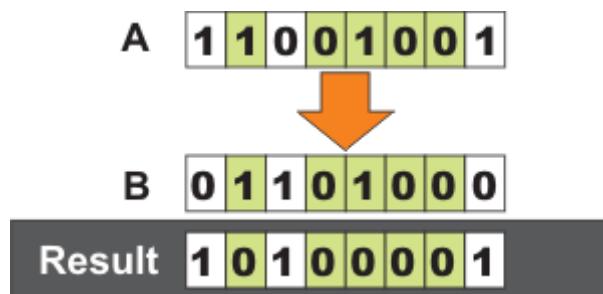
#### 1.9.4. XOR Gate:

Cổng XOR là một cổng logic dùng để thực hiện hàm XOR hai hay nhiều biến. Cổng XOR có một ngõ vào tùy thuộc số biến và một ngõ ra. Ngõ ra của cổng là hàm X OR của các biến ngõ vào. Bên phải là bảng chân trị mô tả hoạt động của cổng OR 2 ngõ vào A và B.



Ngõ ra công XOR chỉ ở mức thấp (0) khi tất cả các ngõ vào ở mức thấp (0) hoặc cùng mức cao (1).

Khi 2 ngõ vào khác mức ở thì ngõ ra luôn sẽ ở mức cao (1).



## CHƯƠNG II: NGÔN NGỮ LẬP TRÌNH C

### 2.1. CƠ BẢN VỀ NGÔN NGỮ C:

- Ngôn ngữ lập trình là một ngôn ngữ dùng để viết chương trình cho máy tính. Ta có thể chia ngôn ngữ lập trình thành các loại sau: ngôn ngữ máy, hợp ngữ và ngôn ngữ cấp cao.
- ✓ *Ngôn ngữ máy (machine language)*: Là các chỉ thị dưới dạng nhị phân, can thiệp trực tiếp vào trong các mạch điện tử. Chương trình được viết bằng ngôn ngữ máy thì có thể được thực hiện ngay không cần qua bước trung gian nào. Tuy nhiên chương trình viết bằng ngôn ngữ máy dễ sai sót, cồng kềnh và khó đọc, khó hiểu vì toàn những con số 0 và 1.
  - ✓ *Hợp ngữ (assembly language)*: Bao gồm tên các câu lệnh và quy tắc viết các câu lệnh đó. Tên các câu lệnh bao gồm hai phần: phần mã lệnh (viết tựa tiếng Anh) chỉ phép toán cần thực hiện và địa chỉ chứa toán hạng của phép toán đó.

**Ví dụ:**

INPUT a; Nhập giá trị cho a từ bàn phím LOAD

ADD b; Cộng giá trị của thanh ghi tổng A với giá trị b

Trong các lệnh trên thì INPUT, LOAD, PRINT, ADD là các mã lệnh còn a, b là địa chỉ. Để máy thực hiện được một chương trình viết bằng hợp ngữ thì chương trình đó phải được dịch sang ngôn ngữ máy. Công cụ thực hiện việc dịch đó được gọi là Assembler.

- ✓ *Ngôn ngữ cấp cao (High level language)*: Ra đời và phát triển nhằm phản ánh cách thức người lập trình nghĩ và làm. Rất gần với ngôn ngữ con người (Anh ngữ) nhưng chính xác như ngôn ngữ toán học. Cùng với sự phát triển của các thế hệ máy tính, ngôn ngữ lập trình cấp cao cũng được phát triển rất đa dạng và phong phú, việc lập trình cho máy tính vì thế mà cũng có nhiều khuynh hướng khác nhau: lập trình cấu trúc, lập trình hướng đối tượng, lập trình logic, lập trình hàm... Một chương trình viết bằng ngôn ngữ cấp cao được gọi là chương trình nguồn (source programs). Để máy tính "hiểu" và thực hiện được các lệnh trong chương trình nguồn thì phải có một chương trình dịch để dịch chương trình nguồn (viết bằng ngôn ngữ cấp cao) thành dạng chương trình có khả năng thực thi. Có thể kể đến các ngôn ngữ như C, Basic, Pascal ...

#### ❖ Ưu nhược điểm của ngôn ngữ bậc thấp:

##### ➢ Ưu điểm

- ✓ Mã máy sinh ra rất ngắn gọn, thời gian xử lý được giảm thiểu.
- ✓ Trình biên dịch hợp ngữ của các họ vi điều khiển đều miễn phí

##### ➢ Nhược điểm:

- ✓ Khó khăn trong việc tiếp cận tập lệnh.
- ✓ Không có sẵn các cấu trúc giải thuật (if .. else, for, switch case v.v.) gây
- ✓ Khó khăn trong việc lập trình.
- ✓ Việc kế thừa và phát triển hầu như là không thể.

#### ❖ Ưu nhược điểm của ngôn ngữ bậc cao:

##### ➢ Ưu điểm

- ✓ Gần với ngôn ngữ con người.
- ✓ Các cấu trúc giải thuật có sẵn tạo sự thuận tiện, dễ dàng diễn đạt thuật toán.
- ✓ Việc kế thừa và phát triển dễ dàng, ít tốn thời gian.
- ✓ C là ngôn ngữ bậc cao, phổ biến và dễ hiểu.
- ✓ Có nhiều trình biên dịch C cho tất cả các chip VĐK
- ✓ Có nguồn tài nguyên phong phú.
- ✓ Sử dụng C giúp rút ngắn thời gian nghiên cứu, thiết kế sản phẩm, nâng cao khả năng kế thừa, phát triển.

##### ➢ Nhược điểm:

- ✓ Mã máy sinh ra thường dài hơn so với hợp ngữ.
- ✓ Các trình biên dịch thường được bán với giá khá cao.

### Chương trình dịch

Như trên đã trình bày, muốn chuyển từ chương trình nguồn sang chương trình đích phải có chương trình dịch (chúng ta sẽ làm quen với trình biên dịch CCS for Pic). Thông thường mỗi một ngôn ngữ cấp cao đều có một chương trình dịch riêng nhưng chung quy lại thì có hai cách dịch: thông dịch và biên dịch. Thông dịch (interpreter): Là cách dịch từng lệnh một, dịch tới đâu thực hiện tới đó. Chẳng hạn ngôn ngữ LISP sử dụng trình thông dịch.

Biên dịch (compiler): Dịch toàn bộ chương trình nguồn thành chương trình đích rồi sau đó mới thực hiện. Các ngôn ngữ sử dụng trình biên dịch như Pascal, C...

Giữa thông dịch và biên dịch có khác nhau ở chỗ: Do thông dịch là vừa dịch vừa thực thi chương trình còn biên dịch là dịch xong toàn bộ chương trình rồi mới thực thi nên chương trình viết bằng ngôn ngữ biên dịch thực hiện nhanh hơn chương trình viết bằng ngôn ngữ thông dịch.

Một số ngôn ngữ sử dụng kết hợp giữa thông dịch và biên dịch chẳng hạn như Java. Chương trình nguồn của Java được biên dịch tạo thành một chương trình đối tượng (một dạng mã trung gian) và khi thực hiện thì từng lệnh trong chương trình đối tượng được thông dịch thành mã máy.

### 2.2. Cấu trúc của một chương trình C:

// Header #include <File.h> #include "File.c"	<ul style="list-style-type: none"> <li>✓ Sẽ bao gồm tất cả các file mà ta dung trong chương trình.</li> <li>✓ Nếu dùng &lt;file.x&gt; thì chương trình compile sẽ tìm trong thư mục bạn đang viết.</li> <li>✓ Nếu bạn dùng "file.x" thì chương trình sẽ tìm thêm trong ổ đĩa c/keilc.</li> </ul>
#Define swich RD1	<ul style="list-style-type: none"> <li>✓ Ở đây ta đã định nghĩa chân RD1 là swich, sau này ta chỉ cần viết swich là chương trình sẽ hiểu là bit RD1.</li> </ul>
Unsigned char x,y; Long n;	<ul style="list-style-type: none"> <li>✓ Sẽ gồm các biến mà bạn sẽ dùng trong chương trình.</li> </ul>
Void program_1(void) { int temp=0;//biến cục bộ // write code here } Void program_2();	<ul style="list-style-type: none"> <li>✓ Gồm các chương trình con bạn viết để gọi trong chương trình main.</li> </ul>
Void main(void) { // write code here }	<ul style="list-style-type: none"> <li>// Hàm chương tình chính</li> <li>✓ Void main(void) {}</li> <li>✓ Chương trình sẽ chạy ở đây cho nên bạn bắt buộc phải viết Hàm này.</li> </ul>
Void program_2(void) { // write code here }	

- ✓ Các câu lệnh trong hàm chính có thể gọi các hàm con đã khai báo hoặc không.
- ✓ Hàm chính và hàm con chỉ có thể gọi các hàm được khai báo phía trên nó.
- ✓ Các câu lệnh trong C kết thúc bằng dấu ;
- ✓ Khi có lời gọi hàm con nào thì chương trình sẽ nhảy đến thực hiện hàm con đó.
- ✓ Sau khi thực hiện xong sẽ nhảy về thực hiện tiếp các hàm hoặc câu lệnh trong chương trình chính.
- ✓ Đặt các lời giải thích bằng dấu // hoặc /\* ... \*/
- ✓ Một ưu điểm nổi bật của C là các bạn có thể tạo ra các bộ thư viện .

**Ví dụ:** sau là tạo thư viện thuvien.h (đuôi .h bạn có thể tạo bằng cách save as .. \*.h ở C ).

```
#ifndef _thuvien_H
#define _thuvien_H
....//mã chương trình
#endif
```

(Đây chính là thư viện mà ta dùng để #include<file.x> ở phần đầu của chương trình.  
Cách viết này giúp ta thực hiện các công việc nhóm và chia sẻ dễ dàng.)

### 2.3. Kiểu dữ liệu (Basic Types)

Type-Specifier (Định nghĩa kiểu)	Size	Range (Vùng giới hạn)		
		Unsigned	Signed	Digits(Kí số)
<b>int1</b>	1 bit number	0 to 1	N/A	1/2
<b>int8</b>	8 bit number	0 to 255	-128 to 127	2-3
<b>int16</b>	16 bit number	0 to 65535	-32768 to 32767	4-5
<b>int32</b>	32 bit number	0 to 4294967295	-2147483648 to 2147483647	9-10
<b>float32</b>	32 bit float	$-1.5 \times 10^{45}$ to $3.4 \times 10^{38}$		7-8

C Standard Type	Default Type	Chú ý:
short	int1	
char	unsigned int8	
int	int8	
long	int16	
long long	int32	
float	float32	<ul style="list-style-type: none"> <li>✓ Tất cả các kiểu ngoại trừ float, được mặc định là không dấu. Tuy nhiên có thể không dấu hoặc có dấu bằng cách khai báo Unsigned hoặc Signed.</li> <li>✓ SHORT là một kiểu đặc biệt sử dụng rất hiệu quả khi lập trình cho bit thao tác vào ra, mảng của các bit (INT1) thì RAM được hỗ trợ ngay lập tức.</li> </ul>

### 2.4. Khai báo biến và kiểu biến:

- ❖ Có 2 loại biến là biến toàn cục(global variable) và biến cục bộ (local variable).
- **Biến toàn cục:**
  - ✓ Biến toàn cục được khai báo bên ngoài tất cả các hàm kể cả hàm main.
  - ✓ Nó thể được khai báo bất kì nơi nào. Thời gian tồn tại nó tồn tại từ lúc chương trình chạy đến khi dừng. Các biến toàn cục không được khai báo trùng nhau.
- **Biến cục bộ:**
  - ✓ Biến cục bộ là những biến được khai báo trong hàm và chỉ có tác dụng trong hàm này, kể cả các biến trong hàm main chỉ có tác dụng trong hàm main. Điều này tương ứng với các

biến có thể có tên chung nhau trong các hàm. Các biến này tự động sinh ra khi hàm hoạt động và tự mất khi ra khỏi hàm.

➤ **Ví dụ:**

```
Int x; //Biến toàn cục
Void main(){
    int y; //Biến cục bộ
    y = 10;
} //ra khỏi main y sẽ bị hủy.
```

❖ **Khai báo biến:**

➤ **Kiểu biến Vùng chứa Tên biến**

✓ **Ví dụ:**

unsigned char tmp; // biến kiểu char không dấu, chứa trong vùng RAM nội truy cập trực tiếp

➤ Có thể gán giá trị ban đầu cho biến ngay khi khai báo.

✓ **Ví dụ:**

Thay vì: unsigned int xdata x;

x=0; // biến kiểu int có dấu, chứa trong RAM ngoài

Ta chỉ cần unsigned int xdata x=0;

- bit flagRun = 1; // biến kiểu bit

➤ **Có thể khai báo cùng lúc nhiều biến có cùng kiểu.**

✓ **Ví dụ:**

signed char x, y, z;

➤ **Chú ý:**

- ✓ Biến phải được khai báo trước khi sử dụng.

- ✓ Để biểu diễn một dãy số hay một bảng dữ liệu, ta có dữ liệu kiểu mảng.

- ✓ Mảng là một tập hợp nhiều phần tử có cùng kiểu giá trị, cùng một tên. Mỗi phần tử được truy cập bằng chỉ số của phần tử đó. Chỉ số mảng bắt đầu tính từ 0.

- ✓ Khai báo mảng như sau:

Loại mảng Vùng chứa Tên mảng [Kích thước]...[Kích thước]

**Ví dụ:**

- int arrSin[10]; // mảng kiểu int, một chiều, có 10 phần tử, chứa trong RAM nội.

- char xdata arrLed [8][16]; //mảng kiểu char không dấu, 2 chiều, kích thước 8 x 16, chứa trong RAM ngoài.

- arrSin[5] = 3; // gán giá trị 3 cho phần tử thứ 6 của mảng arrSin

- char strName[] = "NVN LAB"; // khai báo chuỗi

- Mych = „a“; // khai báo ký tự

❖ **Kiểu biến và từ khóa:**

Kiểu biến	Ý nghĩa	Ví dụ
<b>Static</b>	<ul style="list-style-type: none"> <li>✓ Biến khai báo từ khóa Static là biến nằm trong hàm, được coi là biến cục bộ nhưng vẫn lưu giá trị và có thể sử dụng khi quay trở lại hàm.</li> <li>✓ Phạm vi sử dụng trong hàm khai báo nó. Về mặt thời gian là toàn chương trình. Hàm này được khởi tạo bằng 0 và chỉ được khởi động trong lần một lần khi khởi động.</li> </ul>	<pre>char sumIt(void{     static char sum = 0;     sum = sum + 1;     return sum; } void main(void){     char i;     char result;     for(i=0;i&lt;10;i++)         result = sumIt(); }</pre>
<b>Const</b>	<ul style="list-style-type: none"> <li>✓ Biến hằng có thể là biến biến cục bộ hoặc biến toàn cục tùy vào vị trí</li> </ul>	<pre>const int a = 10; a = b; // sai</pre>

	<ul style="list-style-type: none"> <li>✓ chúng được khai báo.</li> <li>✓ Về phạm vi sử dụng và thời gian tồn tại đều giống như biến toàn cục nhưng giá trị của nó không thể bị thay đổi được bởi lệnh gán.</li> </ul>	
Enum và macro	<ul style="list-style-type: none"> <li>✓ Enum và macro giúp người lập trình có thể thay thế những số (number) bằng tên gọi, cái mà ta có thể nhớ và liên tưởng tới được công dụng của chúng.</li> </ul>	<pre>Enum{start=9, next1, next2, end_val}; =&gt;Next1=10, next2=11, end_val =12. typedef enum {     Bit_RESET = 0,     Bit_SET } BitAction; =&gt; Bit_RESET=0, Bit_SET=1.</pre>
extern	<ul style="list-style-type: none"> <li>✓ Dùng để định nghĩa nó trong 1 module khác.</li> <li>✓ Bao gồm cả biến và hàm.</li> </ul>	<ul style="list-style-type: none"> <li>✓ Trong module LCD.h, khai báo biến: char LCD_value;</li> <li>✓ Muốn gọi biến này trong module main.h ta dùng: <code>extern char LCD_value;</code></li> </ul>
typedef	<ul style="list-style-type: none"> <li>✓ Dùng định nghĩa lại loại biến</li> </ul>	<ul style="list-style-type: none"> <li>✓ <code>typedef signed char int8_t;</code></li> <li>✓ <code>typedef signed short int int16_t;</code></li> </ul>
Struct	<ul style="list-style-type: none"> <li>✓ là kiểu được tạo bằng cách gom một hoặc nhiều biến một cách hợp lý của các kiểu khác nhau, được nhóm với nhau như là một đơn vị độc lập.</li> <li>✓ <code>struct[*] [id] {     type-qualifier [*] id     [:bits]; } [id]</code></li> </ul>	<pre>#include &lt;stdio.h&gt; #include &lt;string.h&gt; typedef struct Books {     char title[50];     int book_id; } Book;  int main( ) {     Book book;     strcpy(book.title,"CProgramming");     book.book_id      =      6495407;     printf("Book title : %s\n",           book.title);     printf( "Book book_id : %d\n",            book.book_id);     return 0; }</pre>

## 2.5. Operators (Toán hạng)

### 2.5.1. Toán tử gán bằng (=):

**b = 5;**  
**a = 2 + b;**  
**a = 2 + (b = 5);**  
**a = b = c = 5;**

### 2.5.2. Toán tử số học (+,-,\*/,%):

Phép toán	Ví dụ
+ : Phép cộng (Addition)	<code>a=2;b=6;c=a+b;=&gt;&gt;c=8.</code>
- : Phép Trừ (Subtraction or Negation)	<code>C=b-a;=&gt;&gt;c=4.</code>
* : Phép Nhân (Multiply)	<code>C=a*b;=&gt;&gt;c=12</code>
/ : Phép Chia (Divide)	<code>C=b/a;=&gt;&gt;c=3</code>
%: Lấy Phần Dư (Trong phép chia) (Modulo)	<code>C=3;d=c%a;=&gt;&gt;d=1.</code>

### **2.5.3. Toán tử gán phục hợp**

<b>Phép toán</b>	<b>Ý nghĩa</b>	<b>Ví dụ</b>
$+=$	Phép toán cộng chỉ định	$A+=B$ is the same as: $A=A+B;$
$\&=$	Phép toán và chỉ định	$A\&=B \Leftrightarrow A=A\&B;$
$^=$	Phép toán XOR chỉ định	$A^=B \Leftrightarrow A=A^B;$
$ =$	Phép toán OR chỉ định	$A =B \Leftrightarrow A=A B;$
$/=$	Phép toán chia chỉ định	$A/=B \Leftrightarrow A=A/B;$
$<<=$	Phép toán dịch trái chỉ định	$A<<=B \Leftrightarrow A=A<<B;$
$\%=$	Phép toán chia lấy phần dư chỉ định	$A\%=B \Leftrightarrow A=A\%B;$
$*=$	Phép toán nhân chỉ định	$A*=B \Leftrightarrow A=A*B;$
$>>=$	Phép toán dịch phải chỉ định	$A>>=B \Leftrightarrow A=A>>B;$
$-=$	Phép toán trừ chỉ định	$A-=B \Leftrightarrow A=A-B;$
$++$	Phép toán cộng chỉ định	$A++ \Leftrightarrow A=A+1;$
$--$	Phép toán trừ chỉ định	$A-- \Leftrightarrow A=A-A;$

➤ **Chú ý: Tính chất tiền tố, hậu tố (Increase and decrement operator)  $++a, a++:$**

$B=3;$

$A=++B;$  // A is 4, B is 4

$B=3;$

$A=B++;$  // A is 3, B is 4

### **2.5.4. Toán tử quan hệ (Relational operator):**

<b>Phép toán</b>	<b>Ý nghĩa</b>	<b>Ví dụ</b>
$=$	Bằng (Is Equal to)	$A=2;b=6;a==b$ sẽ trả giá trị false
$!=$	Khác (Is not Equal to)	$A!=b;$ sẽ trả giá trị True
$<$	Nhỏ hơn (Less Than)	$A<b;$ Sẽ trả giá trị True
$<=$	Nhỏ hơn hoặc bằng (Less Than or Equal to)	$A<=b;$ Sẽ trả giá trị True
$>$	Lớn hơn (Greater Than)	$A>b;$ sẽ trả giá trị False
$>=$	Lớn hơn hoặc bằng (Greater Than or Equal to)	$A*3>=b;$ Sẽ trả giá trị True

Chú ý: Nếu dùng  $=$  (một dấu bằng) hoàn toàn khác với  $==$  (hai dấu bằng).

$(==)$  nhằm so sánh còn  $(=)$  gán giá trị của biểu thức bên phải cho biến ở bên trái .

### **2.5.5. Toán tử Logic (Logical operator):**

<b>Phép toán</b>	<b>Ý nghĩa</b>	<b>Ví dụ</b>
!	Not	$!(5==5)$ trả về false vì biểu thức bên phải ( $5==5$ ) có giá trị true. $!true$ trả về false. $!false$ trả về true.
$\&\&$	Và (and)	$((5==5) \&\& (3>6))$ trả về false $(true \&\& false).$
$\ $	Hoặc (Or)	$((5==5) \  (3>6))$ trả về true $(true \  false).$

### **2.5.6. Toán tử thao tác bit:**

<b>Phép toán</b>	<b>Ý nghĩa</b>	<b>Ví dụ</b>
$\&$	Logical And	$1000.1011 \& 1010.1010 = 1000.1010$

	<b>Logical OR</b>	<b>1000.1011   1010.1010 = 1010.1011</b>
<b>^</b>	<b>XOR Logical exclusive OR</b>	
<b>~</b>	<b>Not Đảo ngược bit</b>	<b>~ 1010.1010 = 0101.0101</b>
<b>&lt;&lt;</b>	<b>SHL Dịch bit sang trái (Left shift operator)</b>	
<b>&gt;&gt;</b>	<b>SHR Dịch bit sang phải (Right shift operator)</b>	

### 2.5.7. Toán tử điều kiện:

Phép toán	Ý nghĩa	Ví dụ
Điều kiện ?	Nếu Điều kiện là true thì giá trị trả về sẽ là	5==3? 4:3 // trả về 3 vì 5≠3.
Kết quả 1 :	Kết quả 1, nếu không giá trị trả về là Kết quả	a>b? a:b // trả về giá trị lớn hơn, a hoặc b.
Kết quả 2	2.	

### 2.5.8. Toán tử khác

Phép toán	Ý nghĩa	Ví dụ
[]	<b>Dùng để biểu diễn phần tử mảng</b>	a[i][j]
.	<b>Dùng để biểu diễn thành phần cấu trúc</b>	Typedef struct { Int a:4; Int b:4; } nvn; Nvn vd;  Vd.a=0; Vd.b =0x2;
*	<b>Dùng để khai báo con trỏ</b>	fun (int*x,int*y){ if(*x!=5) *y=*x+3; }
( type)	<b>là phép chuyển đổi kiểu</b>	(float)(x+y)
,	<b>thường dùng để viết một dãy biểu thức trong toán tử for, thường dùng để viết một dãy biểu thức trong toán tử for</b>	
sizeof	Xác định kích số byte của toán hạng	
->	Phép toán kiểu cấu trúc hay còn gọi là phép toán con trỏ	struct Y { int b; int a; };  int i = 5; i->b = 42; /* Write 42 into `int` at address 7 */ 100->a = 0; /* Write 0 into `int` at address 100 */  (*i).b = 42;
<b>Lưu ý:</b> Trong C coi mọi giá trị khác không là đúng("TRUE") và mọi giá trị bằng không là sai("FALSE")		

### 2.5.9. Thứ tự ưu tiên các phép toán:

<b>STT</b>	<b>Phép toán</b>	<b>Mô tả</b>	<b>Trình tự kết hợp</b>
1	<code>() [ ] -&gt; . sizeof</code>		Trái qua phải
2	<code>! ~ &amp; * - ++ -- (type) sizeof</code>		Phải qua trái
3	<code>* / %</code>	Toán tử số học	Trái qua phải
4	<code>+ -</code>	Toán tử số học	Trái qua phải
5	<code>&lt;&lt;&gt;&gt;</code>	Dịch bit	Trái qua phải
6	<code>&lt;&lt;=&gt;&gt;=</code>	Toán tử quan hệ	Trái qua phải
7	<code>== !=</code>	Toán tử quan hệ	Trái qua phải
8	<code>&amp;</code>	Toán tử thao tác bit	Trái qua phải
9	<code>^</code>	Toán tử thao tác bit	Trái qua phải
10	<code> </code>	Toán tử thao tác bit	Trái qua phải
11	<code>&amp;&amp;</code>	Toán tử logic	Trái qua phải
12	<code>  </code>	Toán tử logic	Trái qua phải
13	<code>?:</code>	Toán tử điều kiện	Phải qua trái
14	<code>= += -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ^= =</code>	Toán tử gán	Phải qua trái
15	<code>,</code>	Dấu phẩy	Trái qua phải

## 2.6. Statements

### 2.6.1. Cấu trúc điều kiện If:

<b>Cấu trúc</b>	<b>Ý nghĩa</b>
<code>if(biểu thức điều kiện) {     Lệnh hoặc khối lệnh; }</code>	Nếu biểu thức điều kiện cho kết quả đúng thì thực hiện Lệnh hoặc khối lệnh.
<b>VD:</b> <code>if (x == 100) y=50; // nếu x==100 thì gán y=50 if (switch == 1) // nếu switch tác động {     led_on(); // bật led     motor_on(); // gọi hàm motor_on. }</code>	

### 2.6.2. Cấu Trúc Điều Kiện If và else:

<b>Cấu trúc</b>	<b>Ý nghĩa</b>
<code>if (biểu thức điều kiện) {     &lt; Lệnh hoặc khối lệnh1&gt;; } Else {     &lt; Lệnh hoặc khối lệnh2&gt;; }</code>	Nếu biểu thức điều kiện cho kết quả đúng thì thực hiện Lệnh hoặc khối lệnh1, ngược lại sẽ thực hiện lệnh hoặc khối lệnh2.
<b>Ví dụ:</b> <b>VD:</b> <code>if (x == 100) y=50; else y=100; //nếu x==100 thì gán y=50, nếu x!=100 gán y=100;</code>	

### 2.6.3. Cấu trúc While:

<b>Cấu trúc</b>	<b>Ý nghĩa</b>
<code>while (biau_thuc) {     &lt;lệnh hoặc khối lệnh&gt;;</code>	Bước 1: Kiểm tra giá trị bieu_thuc Bước 2: Nếu bieu_thuc cho giá trị đúng:

<pre>}</pre>	<p>thực hiện &lt;lệnh hoặc khối lệnh&gt; và quay về bước 1. ☐ Nếu bieu_thuc sai: thoát khỏi vòng lặp</p>
<p>Ví dụ: Int i = 0; while (i &lt;= 5) //thực hiện 5 lần việc tạo xung bitP0_0. {     RD0 = ~ RD0;     delay(1000);     i++; }</p> <p><b>Chú ý:</b> while(1) {}; Tạo vòng lặp mãi mãi , rất hay dùng trong lập trình VXL. Chương trình chính sẽ được viết trong dấu ngoặc {}. Chương trình tương đương với: Main: . . . Ljmp main</p>	

#### 2.6.4. Câu Trúc Do...While:

Câu trúc	Ý nghĩa
<pre>Do {     &lt;Lệnh hoặc khối lệnh&gt;; } while (bieu_thuc)</pre>	<p>Bước 1: thực hiện &lt;Lệnh hoặc khối lệnh&gt; Bước 2: Kiểm tra giá trị bieu_thuc Nếu bieu_thuc cho giá trị đúng: quay về bước 1. Nếu bieu_thuc sai: thoát khỏi vòng lặp</p>

Ví dụ:  
Int i = 0;  
Do  
{  
 RD0 = ~RD0;  
 delay(1000);  
 i++;  
} while (i <= 5) //thực hiện 5 lần việc tạo xung tại pin RD0.

**Chú ý:**

Chức năng của nó là hoàn toàn giống vòng lặp while chỉ trừ có một điều là điều kiện điều khiển vòng lặp được tính toán sau khi <Lệnh hoặc khối lệnh> được thực hiện, vì vậy <Lệnh hoặc khối lệnh> sẽ được thực hiện ít nhất một lần ngay cả khi <bieu\_thuc> không bao giờ được thỏa mãn .Như ví dụ trên kể cả lúc đầu ta gán i >5 thì nó vẫn tăng giá trị 1 lần trước khi thoát.

#### 2.6.5. Câu Trúc Vòng Lặp For:

Câu trúc	Ý nghĩa
<pre>for (&lt;bt1&gt;; &lt;bt2&gt;; &lt;bt3&gt;) {     &lt;Lệnh hoặc khối lệnh&gt;; }</pre>	<p>Chức năng chính của nó là lặp lại &lt;Lệnh hoặc khối lệnh&gt; chừng nào &lt;bt2&gt; còn mang giá trị đúng, như trong vòng lặp while. Nhưng thêm vào đó, for cung cấp chỗ dành cho lệnh khởi tạo và lệnh tăng. Vì vậy vòng lặp này được thiết kế đặc biệt lặp lại một hành động với một số lần xác định.</p>

	<p>Bước 1: Thực hiện bt1          Bước 2: Xác định giá trị bt2          Bước 3:          Nếu bt2 sai: kết thúc vòng lặp for          Nếu bt2 đúng: thực hiện &lt;Lệnh hoặc khối lệnh&gt;. Sau đó sẽ thực hiện bước 4.          Bước 4: Thực hiện bt3, sau đó quay lại bước 2 để bắt đầu vòng lặp mới</p>
--	--

Ví dụ: Hàm tạo delay điển hình nhất trong VDK.

void delayms(int n)	for (i = 0; i < 9; i++) //thực hiện 10 lần việc tạo xung RD0
{ int i,j; for (i=0;i<n;i++) for (j=0;j<1500;j++) { }; }	{ RD0 = ~RD0; delay(1000); }

#### Chú ý:

- ✓ Phần khởi tạo và lệnh tăng không bắt buộc phải có. Chúng có thể được bỏ qua nhưng vẫn phải có dấu chấm phẩy ngăn cách giữa các phần. Vì vậy, chúng ta có thể viết for (;n<10;) hoặc for (;n<10;n++). Bằng cách sử dụng dấu phẩy, chúng ta có thể dùng nhiều lệnh trong bất kì trường nào trong vòng for, như là trong phần khởi tạo. Ví dụ chúng ta có thể khởi tạo một lúc nhiều biến trong vòng lặp:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )  
{  
    // Lệnh hoặc khối lệnh...  
}
```

- ✓ While và for điều là những lệnh lặp nhưng while dùng khi không biết trước số lần lặp, for dung khi biết trước số lần lặp.

#### 2.6.6. Câu Trúc Lựa Chọn Swich...Case:

Câu trúc	Ý nghĩa
switch (biểu thức) { case constant1: block of instructions1; break; case constant1: block of instructions2; break; ..... [default: default block of instructions] }	Nó hoạt động theo cách sau: switch tính biểu thức và kiểm tra xem nó có bằng constant1 hay không, nếu đúng thì nó thực hiện block of instructions 1 cho đến khi tìm thấy từ khoá break, sau đó nhảy đến phần cuối của câu trúc lựa chọn switch. Còn nếu không, switch sẽ kiểm tra xem biểu thức có bằng constant2 hay không. Nếu đúng nó sẽ thực hiện block of instructions 2 cho đến khi tìm thấy từ khoá break. Cuối cùng, nếu giá trị biểu thức không bằng bất kì hằng nào được chỉ định ở trên (bạn có thể chỉ định bao nhiêu câu lệnh case tùy thích), chương trình sẽ thực hiện các lệnh trong phần default: nếu nó tồn tại vì phần này không bắt buộc phải có.

Ví dụ:

```
switch (i) //i== bao nhiêu??  
{  
    case 1: //nếu i==1 bật Led1  
        led1 on();
```

```

delay(1000);
break ;
case 2: //nếu i==2 bật Led2
    led2_on();
    delay(1000);
break ;
default: //nếu i>1&&i<2 tắt 2 Led
    led1_off();
    led2_off();
}

```

**Chú ý:**

Lệnh Switch...Case và if...else về mặt ý nghĩa giống nhau. Nhưng khi sử dụng Switch...Case bộ nhớ sẽ được tạo và lưu sẵn trong ram nên quá trình xử lý và truy xuất sẽ nhanh hơn rất nhiều. Mặt khác nó sẽ làm tốn ram của vi điều khiển rất nhiều.

### 2.6.7. Lệnh Break:

Cấu trúc	Ý nghĩa
break;	Sử dụng break chúng ta có thể thoát khỏi vòng lặp ngay cả khi điều kiện để nó kết thúc chưa được thỏa mãn. Lệnh này có thể được dùng để kết thúc một vòng lặp không xác định hay buộc nó phải kết thúc giữa chừng thay vì kết thúc một cách bình thường.
<b>Ví dụ:</b> while (1) { i++; if (button == 0) break; //nếu button được nhấn=> thoát khỏi vòng lặp }	

### 2.6.8. Lệnh Continue:

Cấu trúc	Ý nghĩa
continue;	✓ Làm cho chương trình thực hiện tiếp vòng lặp mới, bỏ qua các câu lệnh khác nằm sau lệnh continue này. Chỉ được dùng trong các vòng lặp.
<b>Ví dụ:</b> unsigned char strSource= “01203405678”; unsigned char i; for (i = 0; i < 9; i ++) { if (strSource[i] == ‘5’) continue; // nếu ký tự là ‘5’, bỏ qua blink_led(); // nếu ký tự ≠ ‘5’, chớp led }	

**Chú ý:**

- ✓ Đối với do-while, while: continue sẽ chuyển điều khiển về kiểm tra điều kiện của vòng lặp.
- ✓ Đối với for: đưa chương trình trở lên thực hiện bước 3, sau đó mới kiểm tra điều kiện của for (đưa điều khiển về bước 4).

### 2.6.9. Lệnh Go:

Cấu trúc	Ý nghĩa
go;	✓ Lệnh này cho phép nhảy vô điều kiện tới bất kì điểm nào trong chương trình. Nói chung bạn nên tránh dùng nó trong chương trình C++. Tuy nhiên chúng ta vẫn có một ví dụ dùng lệnh goto để đếm ngược:

**2.6.10. Lệnh Return:**

Cấu trúc	Ý nghĩa
return;	<ul style="list-style-type: none"> <li>✓ Lệnh return cho phép thoát ra khỏi một hàm để trở về hàm đã gọi nó.</li> <li>✓ Khi gặp lệnh return, chương trình sẽ bỏ qua các lệnh sau nó để thoát ra khỏi hàm.</li> <li>✓ Các dạng của lệnh return:           <ul style="list-style-type: none"> <li>return;</li> <li>return (bieu_thuc);</li> <li>return bieu_thuc;</li> </ul> </li> <li>✓ Trong thân hàm có thể sử dụng một, hoặc một vài lệnh return. Hoặc cũng có thể không sử dụng lệnh này.</li> <li>✓ Nếu kiểu giá trị trả về của hàm không phải void (Hàm có trả về giá trị) thì trong thân hàm phải sử dụng lệnh return để trả giá trị về cho chương trình.</li> </ul>

**Ví dụ:**

```
void blink_led (void){
    while (1){
        RD0 = 0;
        delay(1000);
        RD0 = 1;
        delay(1000);
        if (button == 0) return; // nếu có nút nhấn => thoát khỏi vòng lặp
    }
}
```

**Ví dụ:**

```
char max (char a, char b){
    if (a > b) return a;
    return b;
}

Int so_sanh(int a,int b)
{
    If(a>b) return 2;           //nếu số a>b thì hàm trả về giá trị 2.
    Else if(a==b) return 1;     //nếu số a=b thì hàm trả về giá trị 1.
    Else return 0;              //nếu số a<b thì hàm trả về giá trị 0.
}
```

Cách khác:

```
Int so_sanh(int a, int b)
{
    Return(a>b) ? 2 : (a==b) ? 1:0;
}
```

**2.7. Lưu đồ giải thuật****2.7.1. Khái niệm giải thuật**

Giải thuật là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy các thao tác trên những dữ liệu vào sao cho sau một số hữu hạn bước thực hiện các thao tác đó ta thu được kết quả của bài toán.

*Ví dụ 1:* Giả sử có hai bình A và B đựng hai loại chất lỏng khác nhau, chẳng hạn bình A đựng rượu, bình B đựng nước mắm. Giải thuật để hoán đổi (swap) chất lỏng đựng trong hai bình đó là:

- ③ Yêu cầu phải có thêm một bình thứ ba gọi là bình C.
- ③ Bước 1: Đổ rượu từ bình A sang bình C.
- ③ Bước 2: Đổ nước mắm từ bình B sang bình A. □ Bước 3: Đổ rượu từ bình C sang bình B.

*Ví dụ 2:* Một trong những giải thuật tìm ước chung lớn nhất của hai số a và b là:

- ③ Bước 1: Nhập vào hai số a và b.
- ③ Bước 2: So sánh 2 số a,b chọn số nhỏ nhất gán cho UCLN.
- ③ Bước 3: Nếu một trong hai số a hoặc b không chia hết cho UCLN thì thực hiện bước 4, ngược lại (cả a và b đều chia hết cho UCLN) thì thực hiện bước 5.
- ③ Bước 4: Giảm UCLN một đơn vị và quay lại bước 3 □ Bước 5: In UCLN - Kết thúc.

### 2.7.2. Các đặc trưng của giải thuật

- ✓ Tính kết thúc: Giải thuật phải dừng sau một số hữu hạn bước.
- ✓ Tính xác định: Các thao tác máy tính phải thực hiện được và các máy tính khác nhau thực hiện cùng một bước của cùng một giải thuật phải cho cùng một kết quả.
- ✓ Tính phổ dụng: Giải thuật phải "vết" hết các trường hợp và áp dụng cho một loạt bài toán cùng loại.
- ✓ Tính hiệu quả: Một giải thuật được đánh giá là tốt nếu nó đạt hai tiêu chuẩn sau:

- Thực hiện nhanh, tốn ít thời gian.
- Tiêu phí ít tài nguyên của máy, chẳng hạn tốn ít bộ nhớ.

Giải thuật tìm UCLN nếu trên đạt tính kết thúc bởi vì qua mỗi lần thực hiện bước 4 thì UCLN sẽ giảm đi một đơn vị cho nên trong trường hợp xấu nhất thì UCLN=1, giải thuật phải dừng. Các thao tác trình bày trong các bước, máy tính đều có thể thực hiện được nên nó có tính xác định. Giải thuật này cũng đạt tính phổ dụng vì nó được dùng để tìm UCLN cho hai số nguyên dương a và b bất kỳ. Tuy nhiên tính hiệu quả của giải thuật có thể chưa cao; cụ thể là thời gian chạy máy có thể còn tốn nhiều hơn một số giải thuật khác mà chúng ta sẽ có dịp trở lại trong phần lập trình C.

### 2.7.3. Ngôn ngữ biểu diễn giải thuật

Để biểu diễn giải thuật, cần phải có một tập hợp các ký hiệu dùng để biểu diễn, mỗi ký hiệu biểu diễn cho một hành động nào đó. Tập hợp các ký hiệu đó lại tạo thành ngôn ngữ biểu diễn giải thuật.

#### 2.7.3.1. Ngôn ngữ tự nhiên

Ngôn ngữ tự nhiên là ngôn ngữ của chúng ta đang sử dụng, chúng ta có thể sử dụng ngôn ngữ tự nhiên để mô tả giải thuật giống như các ví dụ ở trên.

*Ví dụ:* Ta có giải thuật giải phương trình bậc nhất dạng  $ax + b = 0$  như sau:

- ③ Bước 1: Nhận giá trị của các tham số a, b
- ③ Bước 2: Xét giá trị của a xem có bằng 0 hay không? Nếu a=0 thì làm bước 3, nếu a khác không thì làm bước 4.
- ③ Bước 3: (a bằng 0) Nếu b bằng 0 thì ta kết luận phương trình vô số nghiệm, nếu b khác 0 thì ta kết luận phương trình vô nghiệm.

③ Bước 4: ( a khác 0) Ta kết luận phương trình có nghiệm  $x=-b/a$

### 2.7.3.2. Ngôn ngữ sơ đồ (Lưu đồ)

Ngôn ngữ sơ đồ (lưu đồ) là một ngôn ngữ đặc biệt dùng để mô tả giải thuật bằng các sơ đồ hình khối. Mỗi khối qui định một hành động.

Khối	Tác dụng (Ý nghĩa của hành động)	Khối	Tác dụng (Ý nghĩa của hành động)
	Bắt đầu/ Kết thúc		Đường đi
	Nhập / Xuất		Chương trình con
	Thi hành		Khối nội
	Lựa chọn		Lời chú thích

Chẳng hạn ta dùng lưu đồ để biểu diễn giải thuật tìm UCLN nêu trên như sau:

### 2.7.3.3. Một số giải thuật cơ bản

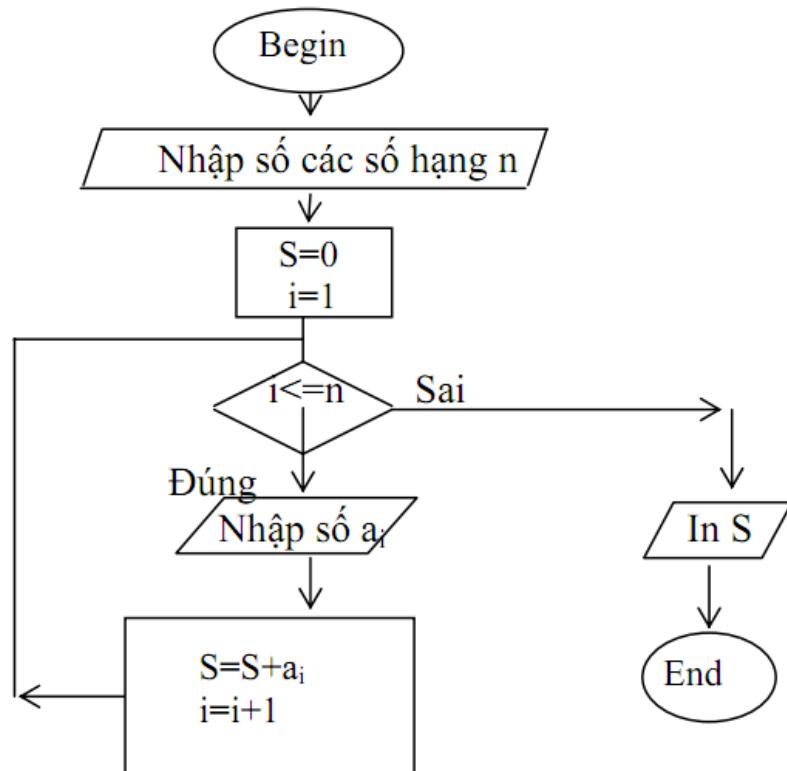
Ví dụ 1: Cần viết chương trình cho máy tính sao cho khi thực hiện chương trình đó, máy tính yêu cầu người sử dụng chương trình nhập vào các số hạng của tổng ( $n$ ); nhập vào dãy các số hạng  $a_i$  của tổng. Sau đó, máy tính sẽ thực hiện việc tính tổng các số  $a_i$  này và in kết quả của tổng tính được.

Yêu cầu: Tính tổng  $n$  số  $S=a_1+a_2+a_3+\dots+a_n$ .

Để tính tổng trên, chúng ta sử dụng phương pháp “cộng tích lũy” nghĩa là khởi đầu cho  $S=0$ . Sau mỗi lần nhận được một số hạng  $a_i$  từ bàn phím, ta cộng tích lũy  $a_i$  vào  $S$  (lấy giá trị được lưu trữ trong  $S$ , cộng thêm  $a_i$  và lưu trả lại vào  $S$ ). Tiếp tục quá trình này đến khi ta tích lũy được  $a_n$  vào  $S$  thì ta có  $S$  là tổng các  $a_i$ . Chi tiết giải thuật được mô tả bằng ngôn ngữ tự nhiên như sau:

- Bước 1: Nhập số các số hạng  $n$ .
- Bước 2: Cho  $S=0$  (lưu trữ số 0 trong  $S$ )
- Bước 3: Cho  $i=1$  (lưu trữ số 1 trong  $i$ )
- Bước 4: Kiểm tra nếu  $i \leq n$  thì thực hiện bước 5, ngược lại thực hiện bước 8.
- Bước 5: Nhập  $a_i$
- Bước 6: Cho  $S=S+a_i$  (lưu trữ giá trị  $S + a_i$  trong  $S$ ) - Bước 7: Tăng  $i$  lên 1 đơn vị và quay lại bước 4.
- Bước 8: In  $S$  và kết thúc chương trình.
- 

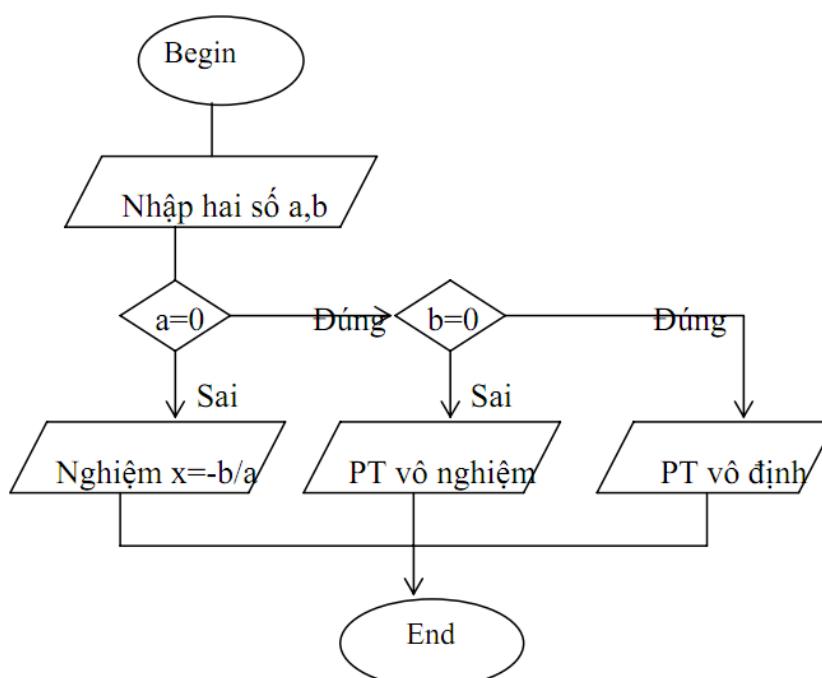
Chi tiết giải thuật bằng lưu đồ:



**Ví dụ 2:** Viết chương trình cho phép nhập vào 2 giá trị a, b mang ý nghĩa là các hệ số a, b của phương trình bậc nhất. Dựa vào các giá trị a, b đó cho biết nghiệm của phương trình bậc nhất  $ax + b = 0$ .

Mô tả giải thuật bằng ngôn ngữ tự nhiên:

- Bước 1: Nhập 2 số a và b
- Bước 2: Nếu  $a = 0$  thì thực hiện bước 3, ngược lại thực hiện bước 4
- Bước 3: Nếu  $b=0$  thì thông báo phương trình vô số nghiệm và kết thúc chương trình, ngược lại thông báo phương trình vô nghiệm và kết thúc chương trình.
- Bước 4: Thông báo nghiệm của phương trình là  $-b/a$  và kết thúc.



## CHƯƠNG III: GIỚI THIỆU VỀ PHẦN MỀM CCS FOR PIC

### 3.1. Tổng quan về CCS

#### 3.1.1. Vì sao ta sử dụng CCS?

Sự ra đời của một loại vi điều khiển đi kèm với việc phát triển phần mềm ứng dụng cho việc lập trình cho con vi điều khiển đó. Vi điều khiển chỉ hiểu và làm việc với hai con số 0 và 1. Ban đầu để việc lập trình cho VĐK là làm việc với dãy các con số 0 và 1. Sau này khi kiến trúc của Vi điều khiển ngày càng phức tạp, số lượng thanh ghi lệnh nhiều lên, việc lập trình với dãy các số 0 và 1 không còn phù hợp nữa, đòi hỏi ra đời một ngôn ngữ mới thay thế. Và ngôn ngữ lập trình Assembly. Ở đây ta không nói nhiều đến Assmebly. Sau này khi ngôn ngữ C ra đời, nhu cầu dùng ngôn ngữ C để thay cho ASM trong việc mô tả các lệnh lập trình cho Vi điều khiển một cách ngắn gọn và dễ hiểu hơn đã dẫn đến sự ra đời của nhiều chương trình soạn thảo và biên dịch C cho Vi điều khiển : Keil C, HT-PIC, MikroC, CCS...

Tôi chọn CCS cho bài giới thiệu này vì CCS là một công cụ lập trình C mạnh cho Vi điều khiển PIC. Những ưu và nhược điểm của CCS sẽ được đề cập đến trong các phần dưới đây.

#### 3.1.2. Giới thiệu về CCS ?

CCS là trình biên dịch lập trình ngôn ngữ C cho Vi điều khiển PIC của hãng Microchip. Chương trình là sự tích hợp của 3 trình biên dịch riêng biệt cho 3 dòng PIC khác nhau đó là:

- PCB cho dòng PIC 12-bit opcodes
- PCM cho dòng PIC 14-bit opcodes
- PCH cho dòng PIC 16 và 18-bit

Tất cả 3 trình biên dịch này được tích hợp lại vào trong một chương trình bao gồm cả trình soạn thảo và biên dịch là CCS, phiên bản mới nhất là PCWH Compiler Ver 5.00xx

Giống như nhiều trình biên dịch C khác cho PIC, CCS giúp cho người sử dụng nắm bắt nhanh được vi điều khiển PIC và sử dụng PIC trong các dự án. Các chương trình điều khiển sẽ được thực hiện nhanh chóng và đạt hiệu quả cao thông qua việc sử dụng ngôn ngữ lập trình cấp cao – Ngôn ngữ C

Tài liệu hướng dẫn sử dụng có rất nhiều, nhưng chi tiết nhất chính là bản Help đi kèm theo phần mềm (tài liệu Tiếng Anh). Trong bản trợ giúp nhà sản xuất đã mô tả rất nhiều về hằng, biến, chỉ thị tiền xử lý, cấu trúc các câu lệnh trong chương trình, các hàm tạo sẵn cho người sử dụng...

### 3.2. Tạo Project trong CCS

#### 3.2.1.Tạo Project đầu tiên trong CCS

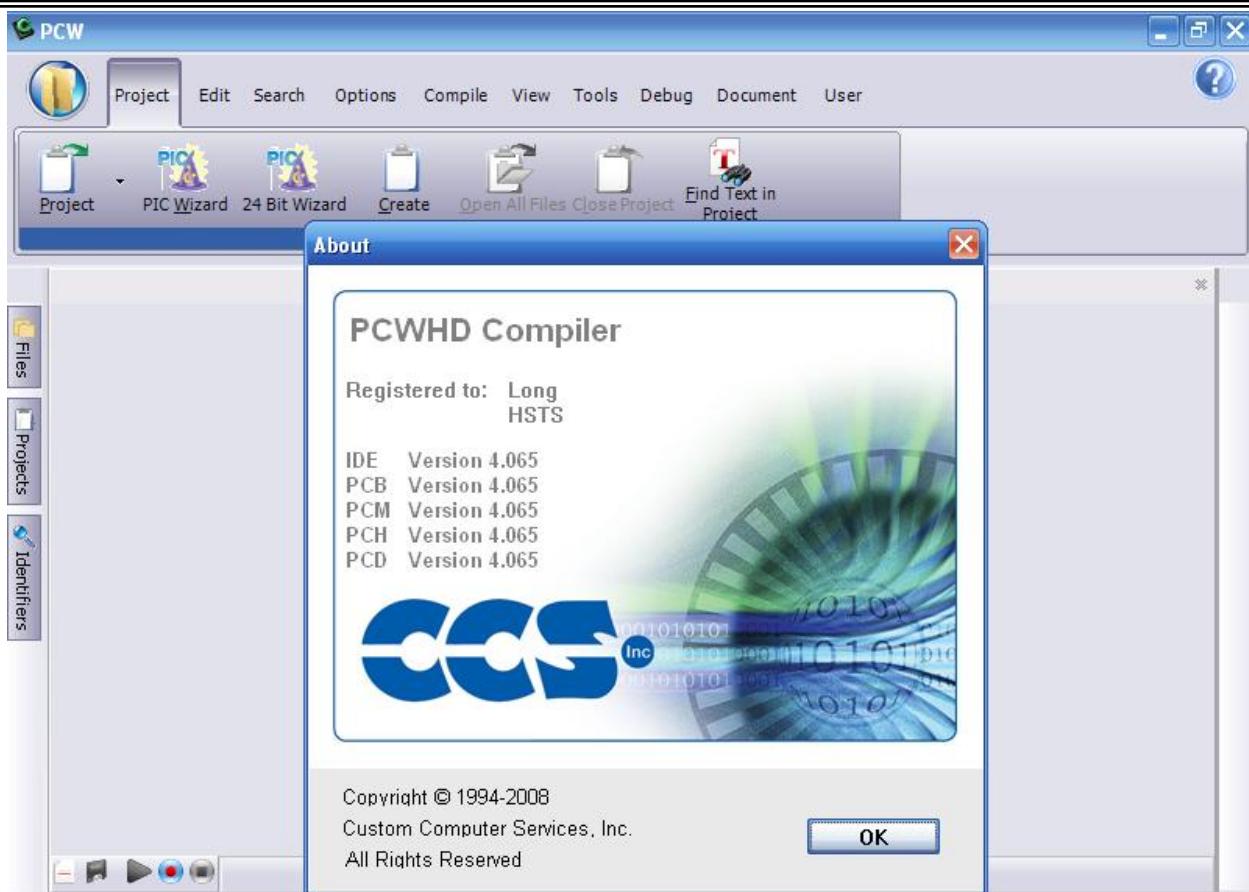
Để tạo một Project trong CCS có nhiều cách, có thể dùng Project Wizard, Manual Creat, hay đơn giản là tạo một Files mới và thêm vào đó các khai báo ban đầu cần thiết và “bắt buộc”.

Dưới đây sẽ trình bày cách tạo một project hợp lệ theo cả 3 phương pháp. Một điều ta cần chú ý khi tạo một Project đó là: khi tạo bắt cứ một Project nào mới thì ta nên tạo một thư mục mới với tên liên quan đến Project ta định làm, rồi lưu các files vào đó. Khi lập trình và biên dịch, CCS sẽ tạo ra rất nhiều files khác nhau, do đó nếu để chung các Project trong một thư mục sẽ rất mất thời gian trong việc tìm kiếm sau này. Đây cũng là quy tắc chung khi ta làm việc với bất kỳ phần mềm nào, thiết kế mạch hay lập trình.

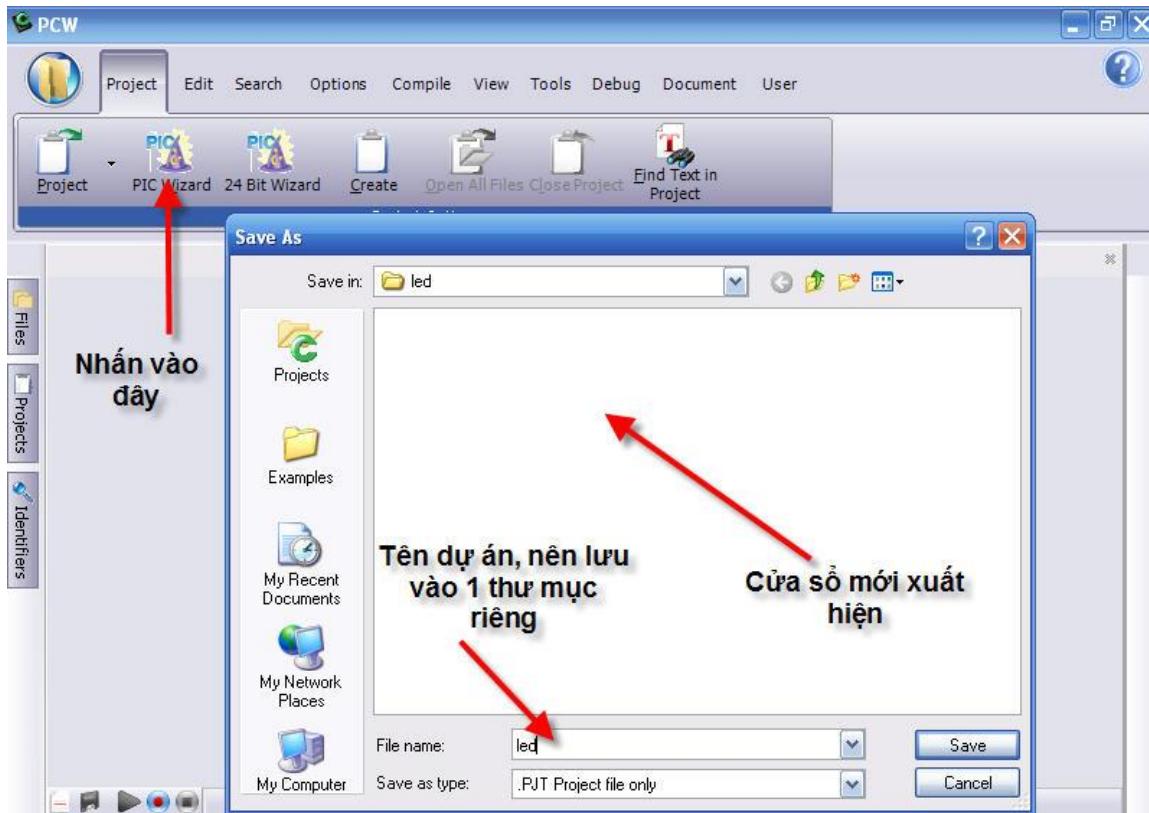
Việc đầu tiên bạn cần làm là khởi động máy tính và bật chương trình PIC C Compiler.

#### 3.2.2. Tạo một PROJECT sử dụng PIC Wizard

Trước hết bạn khởi động chương trình làm việc PIC C Compiler. Từ giao diện chương trình bạn di chuột chọn Project -> New -> PIC Wizard nhấn nút trái chuột chọn.



Sau khi nhấn chuột, một cửa sổ hiện ra yêu cầu ban nhập tên Files cần tạo. Bạn tạo một thư mục mới, vào thư mục đó và lưu tên files cần tạo tại đây.

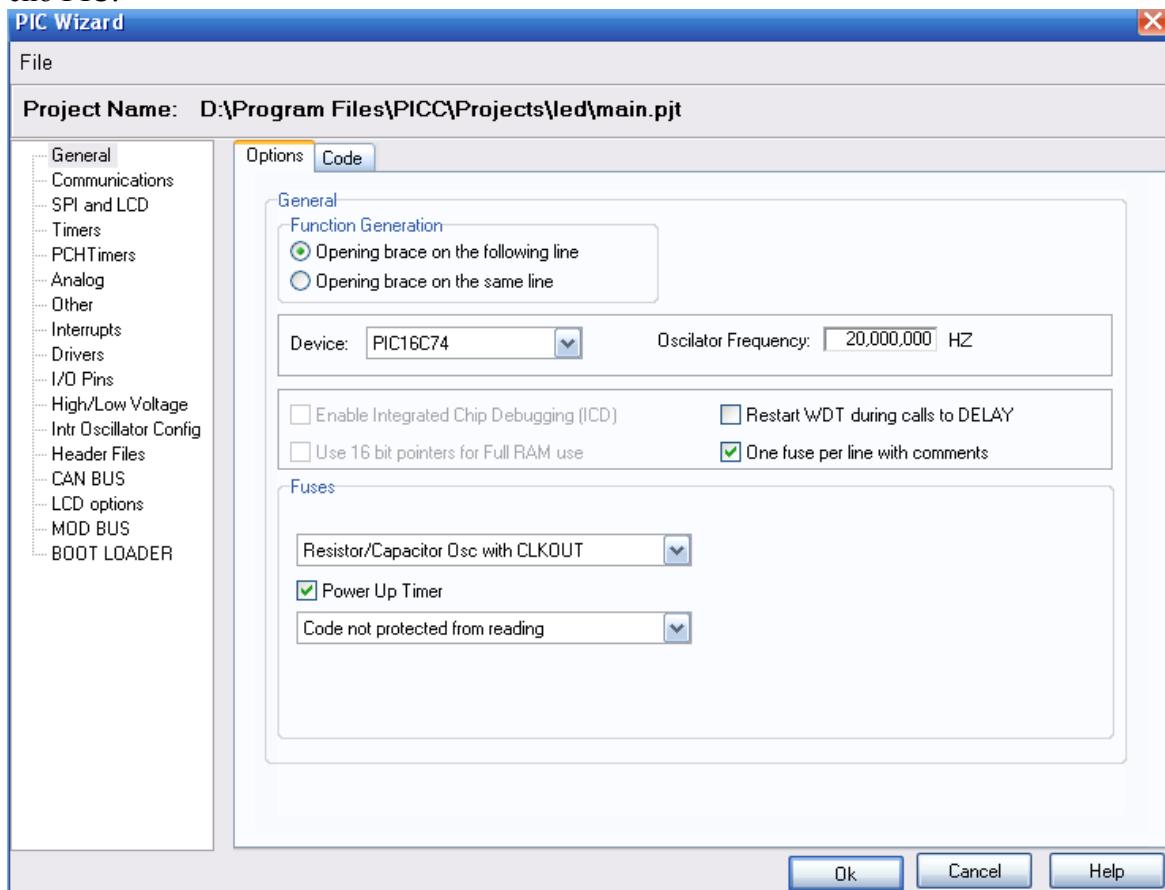


Như vậy là xong bước đầu tiên. Sau khi nhấn nút Save, một cửa sổ New Project hiện ra. Trong cửa sổ này bao gồm rất nhiều Tab, mỗi Tab mô tả về một vài tính năng của con PIC. Ta sẽ chọn tính năng sử dụng tại các Tab tương ứng.

Dưới đây sẽ trình bày ý nghĩa từng mục chọn trong mỗi Tab. Các mục chọn này chính là để cập đến các tính năng của một con PIC, tùy theo từng loại mà sẽ có các Tab tương ứng. Đối với từng dự án khác nhau, khi ta cần sử dụng tính năng nào của con PIC thì ta sẽ chọn mục đó. Tổng cộng có 13 Tab để ta lựa chọn. Tôi giới thiệu những Tab chính thường hay được sử dụng.

### a) Tab General:

Tab General cho phép ta lựa chọn loại PIC mà ta sử dụng và một số lựa chọn khác như chọn tần số thạch anh dao động, thiết lập các bit CONFIG nhằm thiết lập chế độ hoạt động cho PIC.



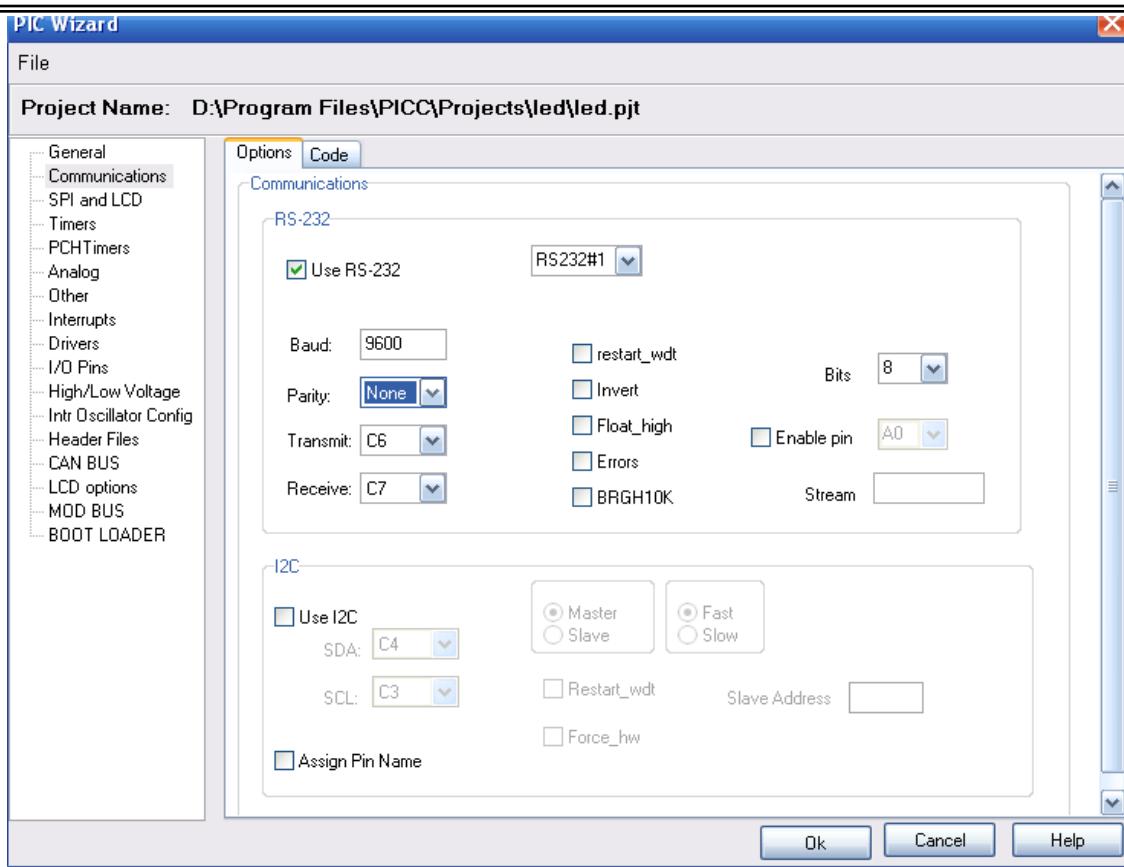
Hình 3.1: Tab General

- Device: Liệt kê danh sách các loại PIC 12F, 16F, 18F... Ta sẽ chọn tên Vi điều khiển PIC mà ta sử dụng trong dự án. Lấy ví dụ chọn PIC16F877A
- Oscilator Frequency: Tần số thạch anh ta sử dụng, chọn 20 MHz (tùy từng loại)
- Fuses: Thiết lập các bit Config như: Chế độ dao động (HS, RC, Internal), chế độ bảo vệ Code, Brownout detected...
- Chọn kiểu con trỏ RAM là 16-bit hay 8-bit.

### b) Tab Communications:

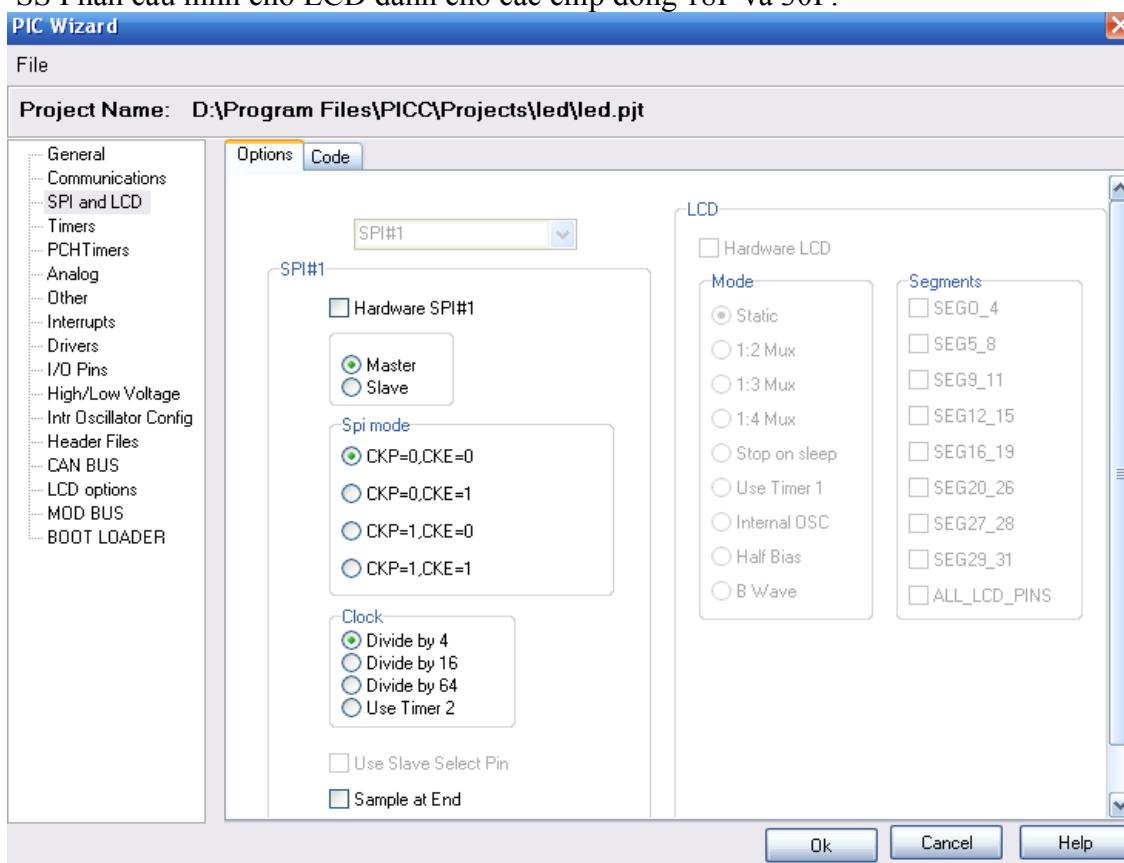
Tab Communications liệt kê các giao tiếp nối tiếp mà một con PIC hỗ trợ, thường là RS232 và I2C, cùng với các lựa chọn để thiết lập chế độ hoạt động cho từng loại giao tiếp. Giao tiếp RS232

Mỗi một Vi điều khiển PIC hỗ trợ một cổng truyền thông RS232 chuẩn. Tab này cho phép ta lựa chọn chân Rx, Tx, tốc độ Baud, Data bit, Bit Parity... Giao tiếp I2C Để sử dụng I2C ta tích vào nút chọn Use I2C, khi đó ta có các lựa chọn: Chân SDA, SCL, Tốc độ truyền (Fast - Slow), chế độ Master hay Slave, địa chỉ cho Slave.



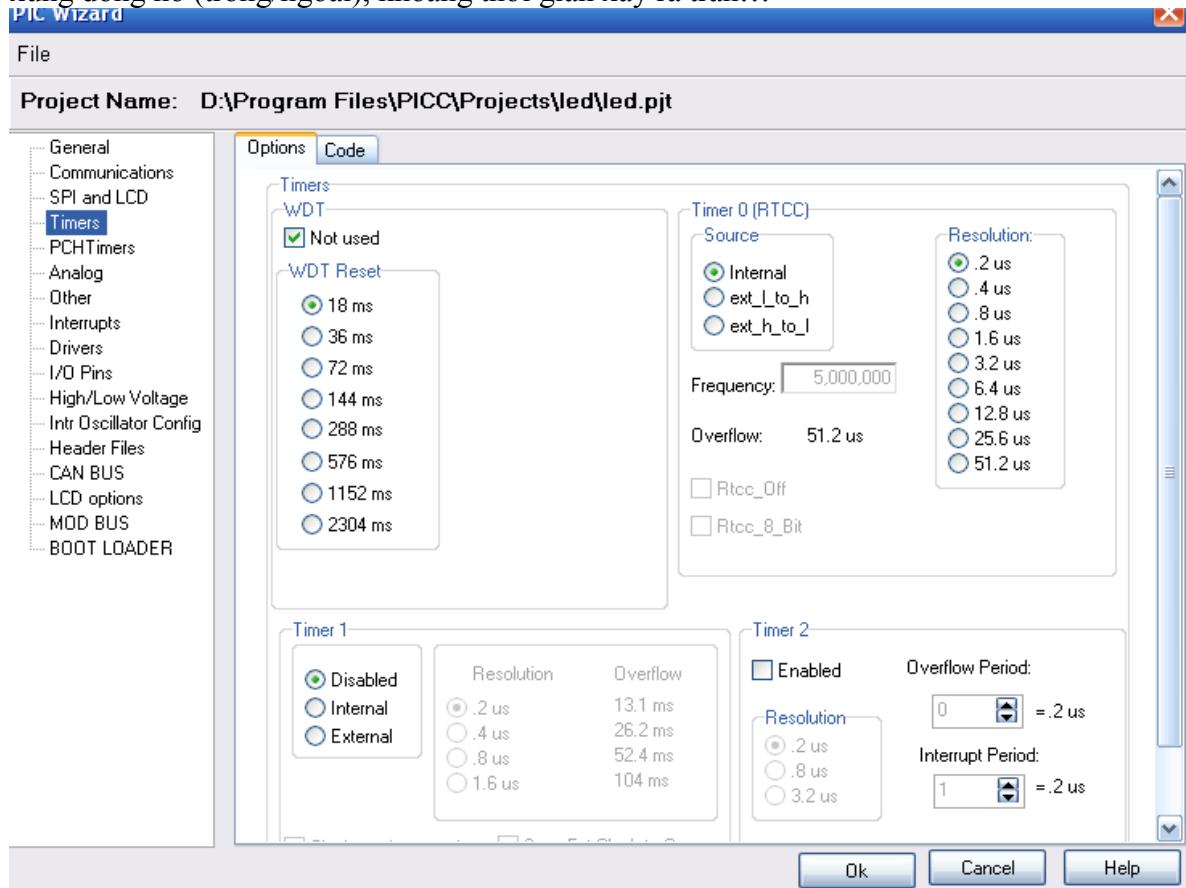
c) Tab SPI and LCD:

Tab này liệt kê cho người dùng các lựa chọn đối với giao tiếp nối tiếp SPI, chuẩn giao tiếp p tốc độ cao mà PIC hỗ trợ về phần cứng. Chú ý khi ta dùng I2C thì không thể dùng SPI và ngược lại. Để có thể sử dụng cả hai giao tiếp này cùng một lúc thì buộc một trong 2 giao tiếp phải lập trình bằng phần mềm (giống như khi dùng I2C cho các chip AT8051, không có hỗ trợ phần cứng SS). Phần cấu hình cho LCD dành cho các chip dòng 18F và 30F.



#### d) Tab Timer

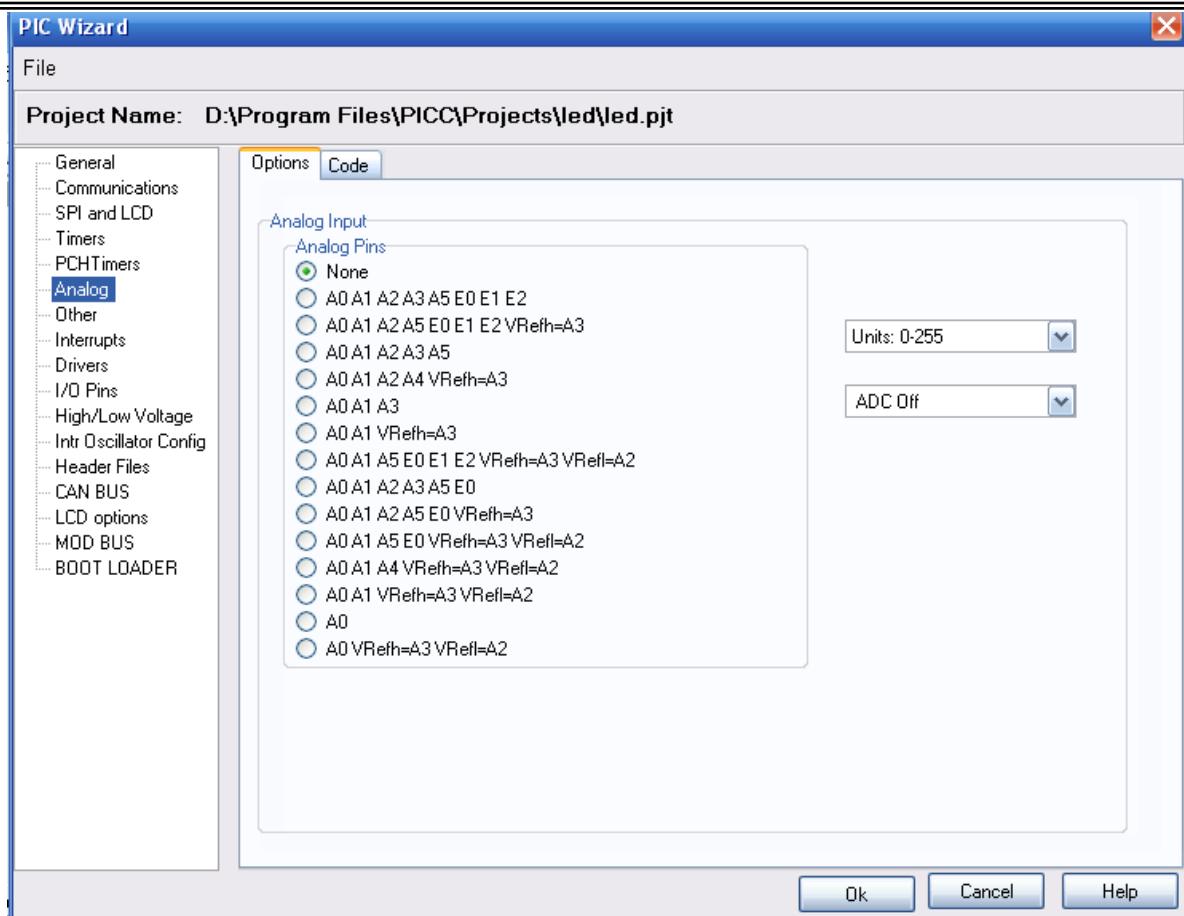
Liệt kê các bộ đếm/định thời mà các con PIC dòng Mid-range có: Timer0, timer1, timer2, WDT... Trong các lựa chọn cấu hình cho các bộ đếm /định thời có: chọn nguồn xung đồng hồ (trong/ngoài), khoảng thời gian xảy ra tràn...



#### e) Tab Analog

Liệt kê các lựa chọn cho bộ chuyển đổi tương tự/số (ADC) của PIC. Tùy vào từng IC cụ thể mà có các lựa chọn khác nhau, bao gồm:

- Lựa chọn cổng vào tương tự
- Chọn chân điện áp lấy mẫu (Vref)
- Chọn độ phân giải: 8-bit = 0 ~ 255 hay 10-bit = 0~1023
- Nguồn xung đồng hồ cho bộ ADC (trong hay ngoài), từ đó mà ta có được tốc độ lấy mẫu, thường ta chọn là internal 2-6 us.
- Khi không sử dụng bộ ADC ta chọn none



#### f) Tab Other:

Tab này cho phép ta thiết lập các thông số cho các bộ Capture/Comparator/PWM.  
**Capture - Bắt giữ**

- Chọn bắt giữ xung theo sườn dương (rising edge) hay sườn âm (falling edge) của xung vào.
- Chọn bắt giữ sau 1, 4 hay 16 xung (copy giá trị của TimerX vào thanh ghi lưu trữ CCCPx sau 1, 4 hay 16 xung).

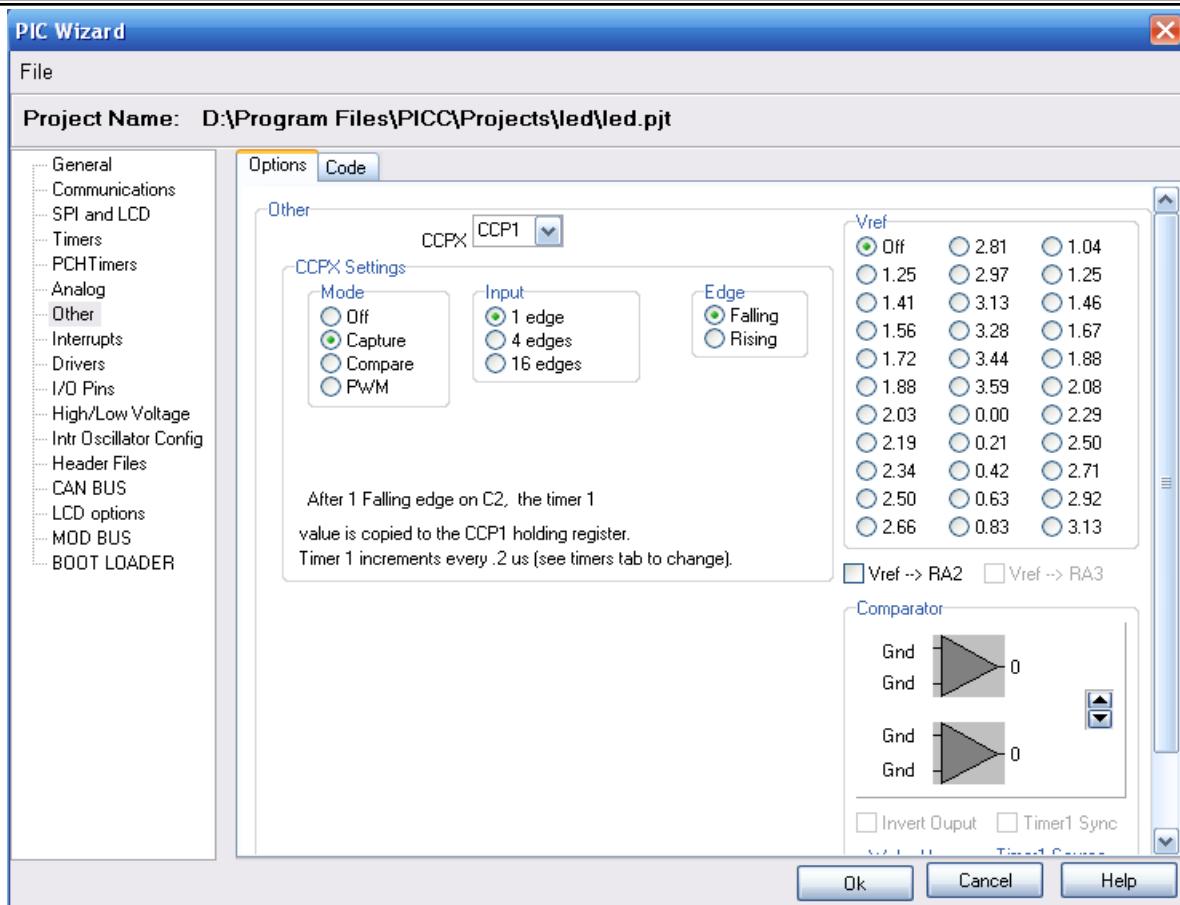
#### Compare - So sánh

- Ta có các lựa chọn thực hiện lệnh khi xảy ra bằng nhau giữa 2 đối tượng so sánh là giá trị của Timer1 với giá trị lưu trong thanh ghi để so sánh. Bao gồm:
  - o Thực hiện ngắt và thiết lập mức 0
  - o Thực hiện ngắt và thiết lập mức 1
  - o Thực hiện ngắt nhưng không thay đổi trạng thái của chân PIC.
  - o Đưa Timer1 về 0 nhưng không thay đổi trạng thái chân.

#### PWM - Điều chế độ rộng xung

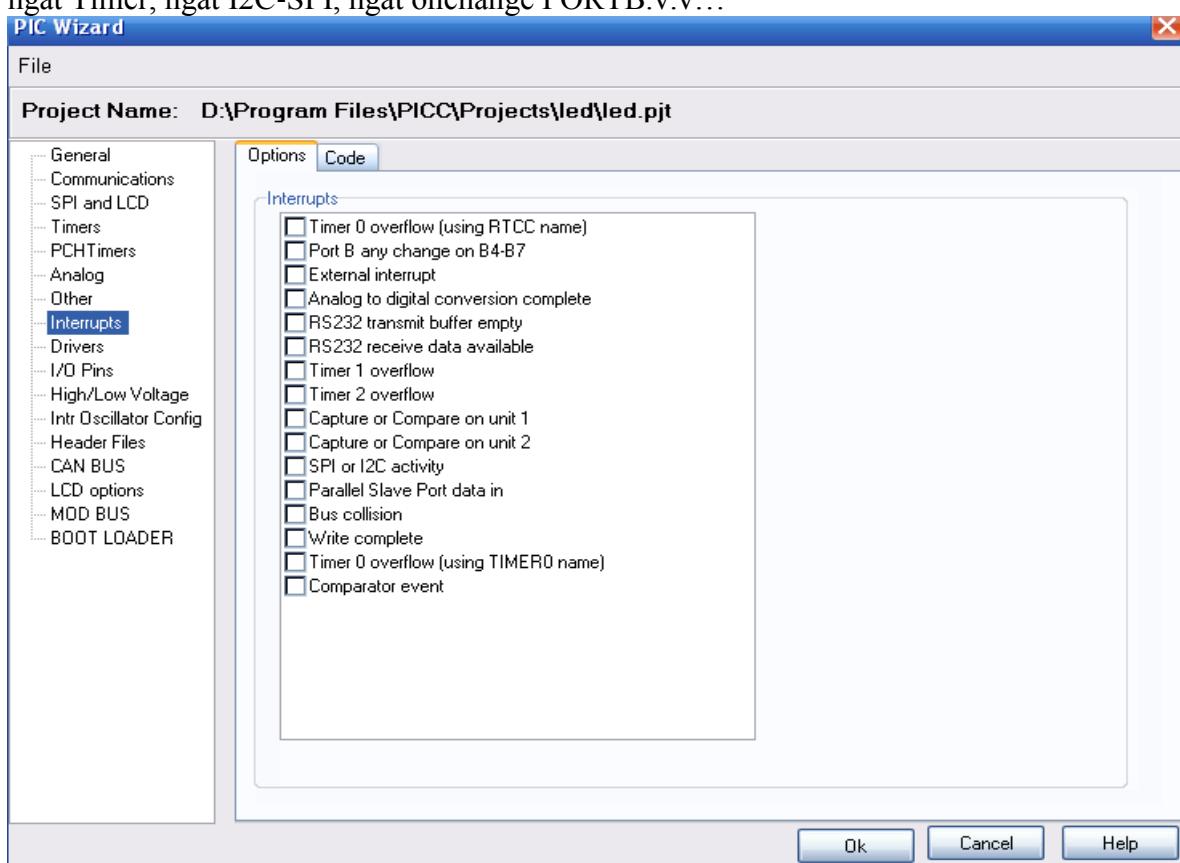
- Lựa chọn về tần số xung ra và duty cycle. Ta có thể lựa chọn sẵn hay tự chọn tần số, tất nhiên tần số ra phải nằm trong một khoảng nhất định.

**Comparator - So sánh điện áp** - Lựa chọn mức điện áp so sánh Vref. Có rất nhiều mức điện áp để ta lựa chọn. Ngoài ra ta còn có thể lựa chọn cho đầu vào của các bộ so sánh.



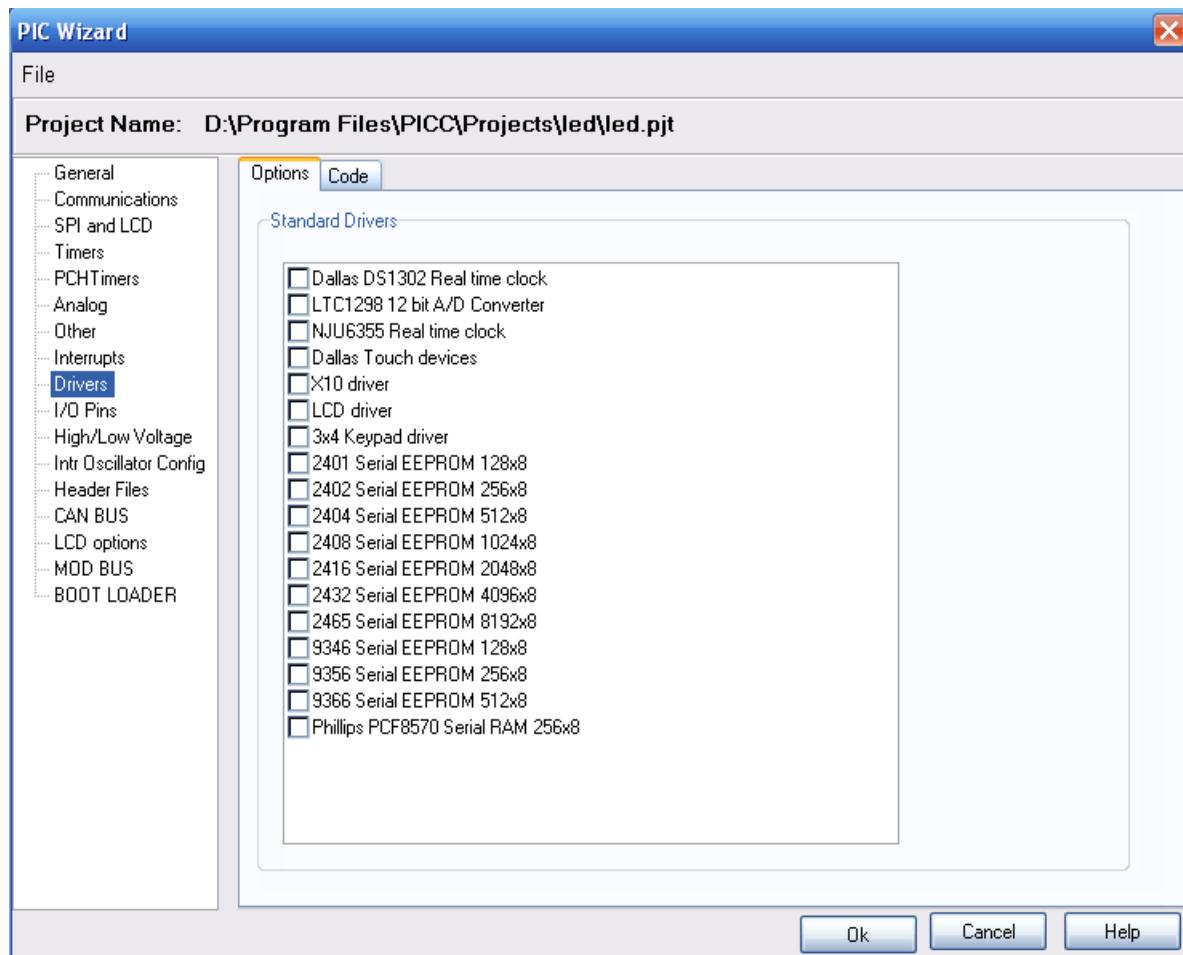
### g) Tab Interrupts:

Tab Interrupts cho phép ta lựa chọn nguồn ngắt mà ta muốn sử dụng. Tùy vào từng loại PIC mà số lượng nguồn ngắt khác nhau, bao gồm: ngắt ngoài 0(INT0), ngắt RS232, ngắt Timer, ngắt I2C-SPI, ngắt onchange PORTB.v.v...

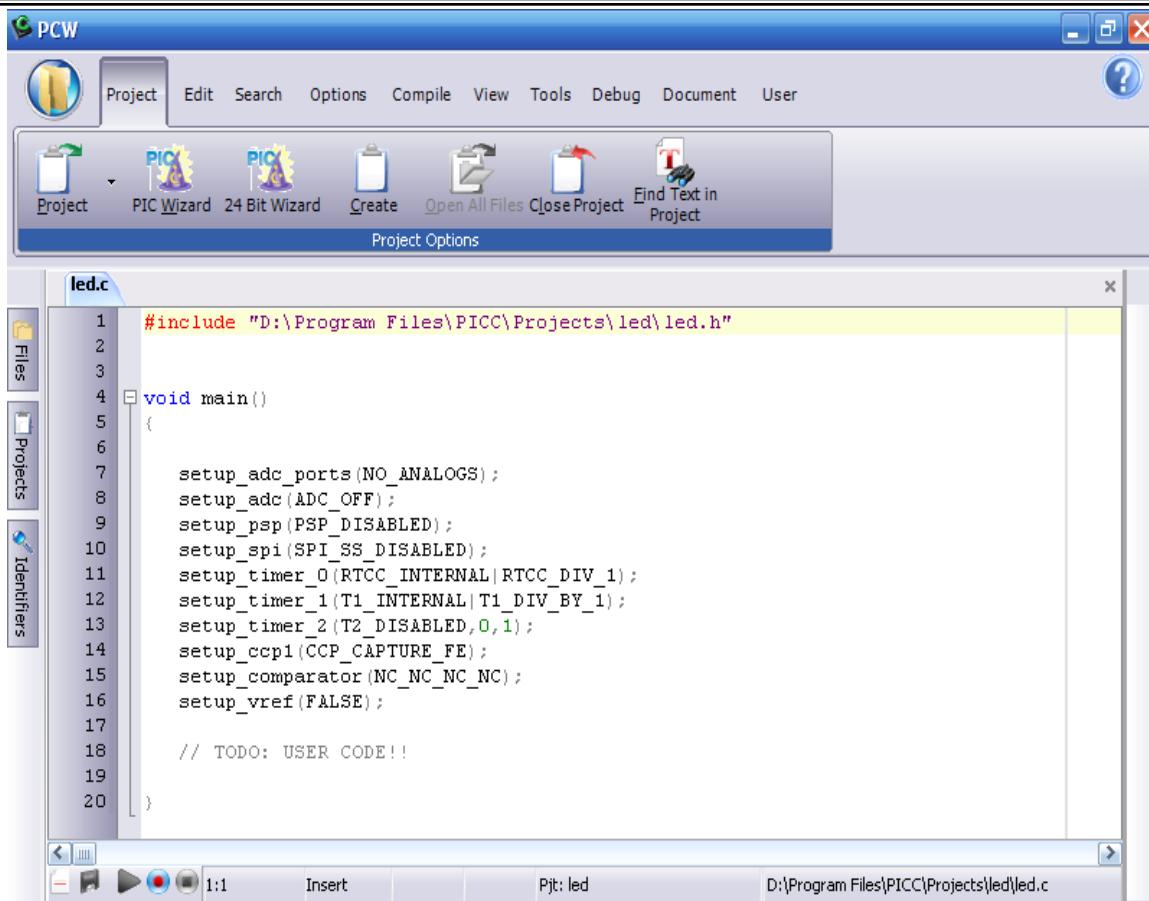


**h) Tab Driver**

Tab Drivers được dùng để lựa chọn những ngoại vi mà trình dịch đã hỗ trợ các hàm giao tiếp. Đây là những ngoại vi mà ta sẽ kết nối với PIC, trong các IC mà CCS hỗ trợ, đáng chú ý là các loại EEPROM như 2404, 2416, 2432, 9346, 9356... Ngoài ra còn có IC RAM PCF8570, IC thời gian thực DS1302, Keypad 3x4, LCD, ADC... Chi tiết ta có thể xem trong thư mục Driver của chương trình: \..\PICC\Drivers



=> Sau các bước chọn trên, ta nhấn OK để kết thúc quá trình tạo một Project trong CCS, một Files ten\_project.c được tạo ra, chứa những khai báo cần thiết cho PIC trong một Files ten\_project.h. Dưới đây là nội dung một files chương trình mẫu.



Phần trên ta đã tìm hiểu cách tạo một Project trong CCS, tuy nhiên theo cách đó mất khá nhiều thời gian, mặt khác mỗi người lập trình sẽ tạo ra những form tài liệu theo cách riêng khác nhau, không đồng nhất. Tài liệu không được chuẩn hóa sẽ gây một số khó khăn cho người đọc, người đọc có thể không hiểu hết những gì mà người lập trình muốn diễn đạt.

Với mục đích đưa ra một form tài liệu chuẩn cho việc lập trình bằng CCS, qua tham khảo bản mẫu của các diễn đàn lớn tôi đưa ra đây một form tài liệu cho việc viết lập trình bằng CCS. Đi kèm văn bản này còn có các files nguồn cho văn bản mẫu, bao gồm files cho PIC16F877A, 16F876A, 16F88. Về sau khi lập trình bạn chỉ việc copy tài liệu này vào thư mục chứa Project của bạn, sửa đổi tên files. Khi cần thay đổi nội dung cấu hình cho PIC bạn chỉ việc tham khảo qua PIC Wizard, xem code và copy đưa vào Project.

Mô tả nội dung chương trình.

- #include 16f877a.h: Đi kèm chương trình dịch, chứa khai báo về các thanh ghi trong mỗi con PIC, dùng cho việc cấu hình cho PIC.
- #include def\_877a.h: Files do người lập trình tạo ra, chứa khai báo về các thanh ghi trong PIC giúp cho việc lập trình được dễ dàng hơn ví dụ ta có thẻ gán PORTB = 0xAA (chi tiết files này sẽ trình bày trong phần dưới đây)
- #device \*=16 ADC = 10: Khai báo dùng con trỏ 8 hay 16 bit, bộ ADC là 8 hay 10 bit
- #FUSES NOWDT, HS: Khai báo về cấu hình cho PIC
- #use delay(clock=20000000): Tần số thạch anh sử dụng
- #use rs232 (baud=9600,...): Khai báo cho giao tiếp nối tiếp RS232
- #use i2c(master, SDA=PIN\_C4,...): Khai báo dùng I2C, chế độ hoạt động
- #include <tên\_file.c>: Khai báo các files thư viện được sử dụng ví dụ LCD\_lib\_4bit.c
- #INT\_xxx : Khai báo địa chỉ chương trình phục vụ ngắt
- Void tên\_chương\_trình (tên\_biến) {}: Chương trình chính hay chương trình con

### i) Mẫu chương trình chuẩn cho lập trình CCS

Chương trình mẫu cho PIC16F877A

```
*****
```

```
/*
* PIC Training Course
* Nguyen Tat Thanh University
*/
******/
```

```
*****/*
* Module      : main.c
* Description  : examples
* Tool        : ccs v5.00xx
* Chip        : 16F877A
* History     : 7/2011
* Version     : v1.0
* Author      : Nguyen Huu Luan
* Mail        : nvn.nhl@gmail.com
* Website     : ntt.edu.vn
* Notes       :
*/
*****/
```

```
///////////
// Mo ta diem khac nhau cac phien ban
// Cac chu thich khac nhu cac che do, khai bao cong xuat nhap.
// Muc dich thuc hien, ung dung....
// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh
// 
/////////
```

```
/*
* Keywords Proteus
* Mo ta lay cac linh kien trong phan men proteus.
*/
******/
```

```
/*
#include <16f877a.h>
#include <def_877a.h>
#device *=16 ADC=8
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP
#use delay(clock=20000000)
#use rs232(baud=9600,parity=N,xmit=PIN_B5,recv=PIN_B2,bits=9)
#use i2c(Master,Fast,sda=PIN_B1,scl=PIN_B4)
#int_xxx // Khai bao chuong trinh ngat
xxx_isr() {
// Code here
}
void Ten_chuong_trinh_con(Ten_Bien) {
// Code here
}
void main() {
// Write code here!
}
```

**Ví dụ:**

```
******/
```

\*

\* PIC Training Course  
\* Nguyen Tat Thanh University  
\*

\*\*\*\*\*\*/

\*\*\*\*\*

\*

\* Module : main.c  
\* Description : Hien thi led 7 doan 0=>>99  
\* Tool : ccs v5.00xx  
\* Chip : 16F877A  
\* History : 7/2011  
\* Version : v1.0  
\* Author : Nguyen Huu Luan  
\* Mail : nvn.nhl@gmail.com  
\* Website : ntt.edu.vn  
\* Notes :

\*\*\*\*\*\*/

//////////

// chuong trinh thiet lap chan cua port B la chan ngo ra quet led //

// chuong trinh thiet lap chan cua port D la chan ngo ra du lieu quet led //

// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh //

// //

//////////

/\*

\* Keywords Proteus  
\* Lay vdk: PIC16F877A  
\* Lay led 7 doan Anot chung: 7SEG  
\* Lay dien tro: Resistor  
\* Lay cong not (Thay transisto) : Not  
\*/

\*\*\*\*\*\*/

#include <16F877A.h>

#include <def\_877a.h> //file header do nguoi dung dinh nghia

#fuses HS,NOWDT,NOPROTECT,NOLVP

#use delay(clock=20000000)

//dinh nghia chan quet led

#define led0 rb1

#define led1 rb0

//dinh nghia portd dua data

#define data\_led portd

char dig[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};// mang chua gia tri led 7 doan

void display0\_99(int8 x){

}

void main()

{

// TODO: USER CODE!!

```

while(true)
{
}
}

//end of main program

```

### j) Các hàm xử lý số, xử lý bit, delay của CCS-C

#### ➤ Các hàm xử lý số:

Sin() cos() tan() Asin() acos() atan()  
 Abs() : lấy trị tuyệt đối  
 Ceil( ) : làm tròn theo hướng tăng  
 Floor ( ): làm tròn theo hướng giảm  
 Exp () : tính  $e^x$   
 Log () :  
 Log10 () :  
 Pow ( ) : tính luỹ thừa  
 Sqrt () : căn thức

=> Các hàm này chạy rất chậm trên các VDK không có bộ nhân phần cứng (PIC 14 ,12 ) vì chủ yếu tính toán với số thực và trả về cũng số thực (32 bit ) và bằng phần mềm .

**Ví dụ:** hàm sin mất 3.5 ms (thạch anh = 20Mhz) để cho KQ. Do đó nếu không đòi hỏi tốc độ thì dùng các hàm này cho đơn giản , như là dùng hàm sin thì khỏi phải lập bảng tra.

Xem chi tiết trên HELP CCS , cũng dễ đọc thôi mà. Hơn nữa chúng ít dùng .

#### ➤ Các hàm xử lý bit và các phép toán:

- ✓ Shift\_right ( address , byte , value )
- ✓ Shift\_left ( address , byte , value )

=> Dịch phải (trái ) 1 bit vào 1 mảng hay 1 cấu trúc . Địa chỉ có thể là địa chỉ mảng hay địa chỉ trỏ tới cấu trúc ( kiểu như &data). Bit 0 byte thấp nhất là LSB .

- ✓ Rotate\_right () , rotate\_left ()

=> Nói chung 4 hàm này ít sử dụng .

- ✓ Bit\_clear ( var , bit ) : dùng xóa (set = 0) bit được chỉ định bởi vị trí bit trong biến var .
- ✓ Bit\_set ( var , bit ) : dùng set=1 bit được chỉ định bởi vị trí bit trong biến var .

var : biến 8 , 16 , 32 bit bất kỳ .

bit : vị trí clear ( set): từ 0-7 (biến 8 bit) , 0-15 (biến 16 bit) , 0-31 (biến 32 bit) .

Hàm không trả về trị .

Ví dụ: Int x;

X=11 ; //x=1011

Bit\_clear (x ,1) ; // x= 1001b = 9

- ✓ Bit\_test ( var , bit ): Dùng kiểm tra vị trí bit trong biến var .

- Hàm trả về 0 hay 1 là giá trị bit đó trong var .
- Var : biến 8, 16 ,32 bit .
- bit : vị trí bit trong var .
- Giả sử bạn có biến x 32 bit đếm từ 0 lên và muốn kiểm tra xem nó có lớn hơn 4096 không ( $4096 = 2^{12} = 1000000000000000b$ ) : If ( x  $\geq 4096$  ) . . . // phép kiểm tra này mất ~5 us Trong 1 vòng lặp , việc kiểm tra thường xuyên như vậy sẽ làm mất 1 thời gian đáng kể. Để tối ưu , chỉ cần dùng: if (bit\_test ( x, 12)=> chỉ mất ~ 0.4 us. (20 Mhz thạch anh) .
- Kiểm tra đếm lên tới những giá trị đặc biệt (  $2^i$  ) thì dùng hàm này rất tiện lợi.

- ✓ Swap ( var ) :

- var : biến 1 byte
- Hàm này tráo vị trí 4 bit trên với 4 bit dưới của var , tương đương var =( var>>4 ) | ( var << 4 )
- Hàm không trả về trị .
- Ví dụ: X= 5 ; //x=00000101b

Swap ( x ) ; //x = 01010000b = 80

- ✓ make8 (var, offset) : Hàm này trích 1 byte từ biến var .
  - var: biến 8,16,32 bit . offset là vị trí của byte cần trích ( 0,1,2,3 ) .
  - Hàm trả về giá trị byte cần trích .
  - Ví dụ: Int16 x = 1453 ; // x=0x5AD  
Y = Make(x, 1) ; //Y= 5 = 0x05
- ✓ make16 (varhigh, varlow): Trả về giá trị 16 bit kết hợp từ 2 biến 8 bit varhigh và varlow . Byte cao là varhigh, thấp là varlow
- ✓ make32 (var1, var2, var3, var4): Trả về giá trị 32 bit kết hợp từ các giá trị 8 bit hay 16 bit từ var1 tới var4. Trong đó var2 đến var4 có thể có hoặc không. Giá trị var1 sẽ là MSB, kế tiếp là var2, ... Nếu tổng số bit kết hợp ít hơn 32 bit thì 0 được thêm vào MSB cho đủ 32 bit .

**Ví dụ:**

```
Int a=0x01, b=0x02, c=0x03, d=0x04; // caùc giaù trò hex
Int32 e;
e = make32 (a, b, c, d); // e = 0x01020304
e = make32 (a, b, c, 5) ; // e = 0x01020305
e = make32 (a, b, 8); // e = 0x00010208
e = make32 (a, 0x1237) ; // e = 0x00011237
```

### ➤ Các hàm delay

Để sử dụng các hàm delay , cần có khai báo tiền xử lý ở đầu file , VD : sử dụng OSC 20Mhz , bạn cần khai báo : #use delay (clock = 20000000)

Hàm delay không sử dụng bất kỳ timer nào . Chúng thực ra là 1 nhóm lệnh ASM để khi thực thi từ đầu tới cuối thì xong khoảng thời gian mà bạn quy định . Tuỳ thời gian delay yêu cầu dài ngắn mà CCS sinh mã phù hợp . có khi là vài lệnh NOP cho thời gian rất nhỏ . Hay 1 vòng lặp NOP . Hoặc gọi tới 1 hàm phức tạp trong trường hợp delay dài . Các lệnh nói chung là vớ vẩn sao cho đủ thời gian quy định là được . Nếu trong trong thời gian delay lại xảy ra ngắt thì thời gian thực thi ngắt không tính vào thời gian delay , xong ngắt nó quay về chạy tiếp các dòng mã cho tới khi xong hàm delay . Do đó thời gian delay sẽ không đúng .

Có 3 hàm phục vụ:

- ✓ delay\_cycles (count )
  - Count: hằng số từ 0 – 255, là số chu kỳ lệnh 1 chu kỳ lệnh bằng 4 chu kỳ máy.
  - Hàm không trả về trị . Hàm dùng delay 1 số chu kỳ lệnh cho trước .
  - Ví dụ: delay\_cycles ( 25 ); // với OSC = 20 Mhz, hàm này delay 5 us
- ✓ delay\_us (time)
  - Time: là biến số thì = 0 – 255, time là 1 hằng số thì = 0 -65535.
  - Hàm không trả về trị.
  - Hàm này cho phép delay khoảng thời gian dài hơn theo đơn vị us.
  - Quan sát trong C / asm list bạn sẽ thấy với time dài ngắn khác nhau , CSS sinh mã khác nhau.
- ✓ delay\_ms (time)
  - Time = 0-255 nếu là biến số hay = 0-65535 nếu là hằng số.
  - Hàm không trả về trị.
  - Hàm này cho phép delay dài hơn nữa.
  - Ví dụ:
 

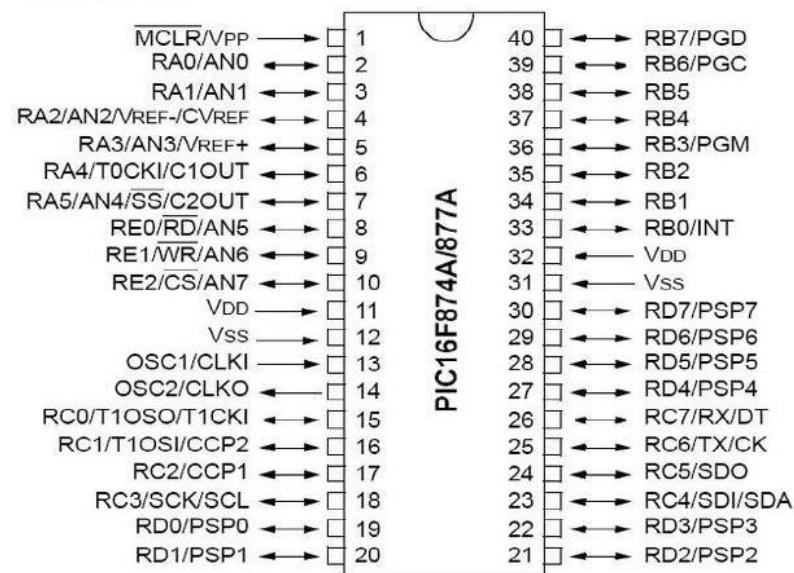
```
Int a = 215;
Delay_us ( a ) ; // delay 215 us
Delay_us ( 4356 ) ; // delay 4356 us
Delay_ms ( 2500 ) ; // delay 2 . 5 s
```

## CHƯƠNG IV. VI ĐIỀU KHIỂN 16F877A

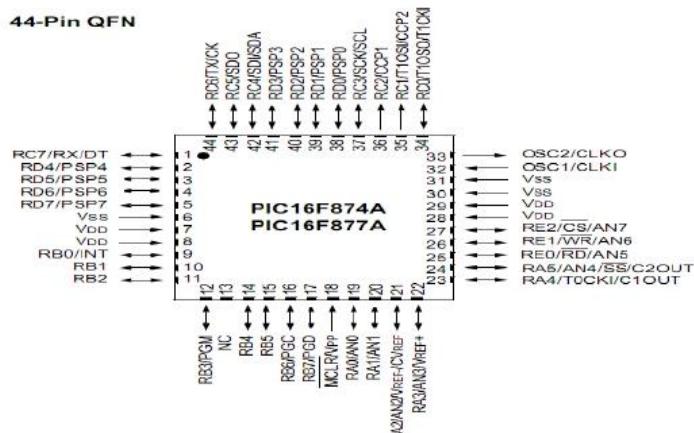
### 4.1. Cấu trúc vi điều khiển PIC16F877A.

#### 4.1.1. Sơ đồ vi điều khiển PIC 16F877A.

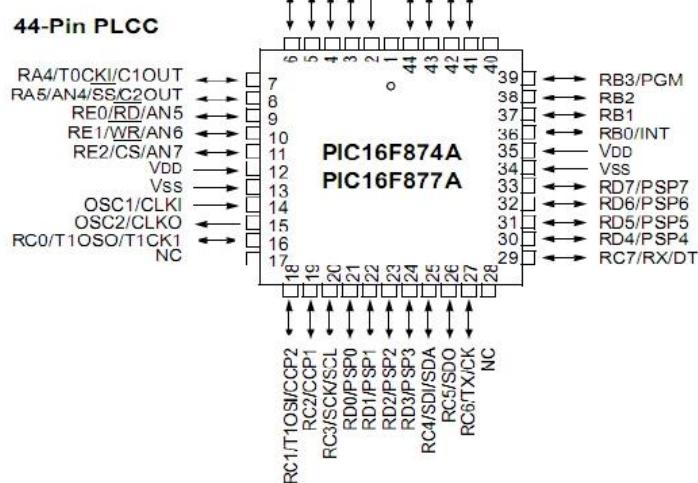
40-Pin PDIP



44-Pin QFN



44-Pin PLCC

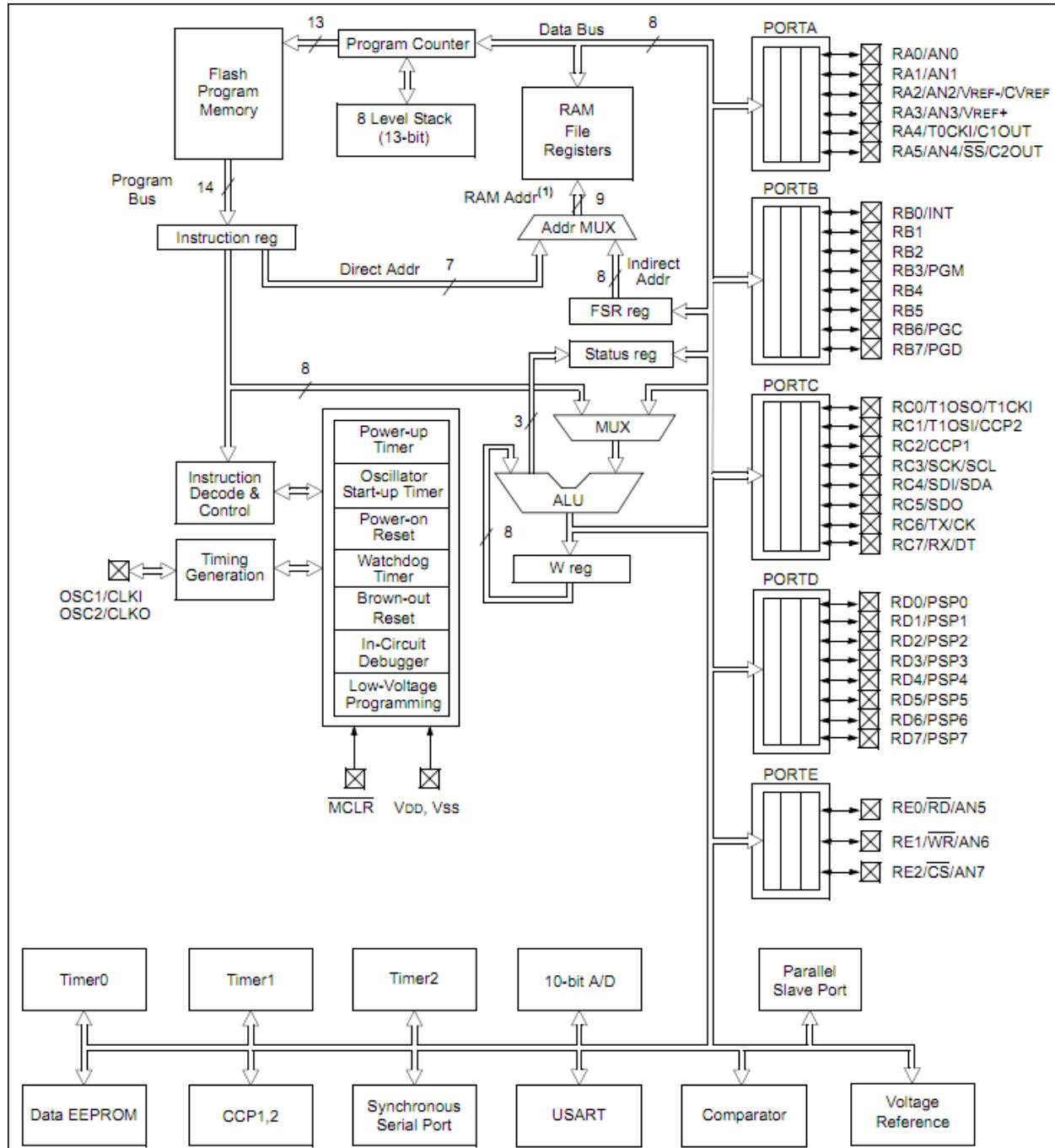


Name	Number (DIP 40)	Function	Description
RE3/MCLR/Vpp	1	RE3	General purpose input Port E
		MCLR	Reset pin. Low logic level on this pin resets microcontroller.
		Vpp	Programming voltage
RA0/AN0/ULPWU/C12IN0-	2	RA0	General purpose I/O port A
		AN0	A/D Channel 0 input
		ULPWU	Stand-by mode deactivation input
		C12IN0-	Comparator C1 or C2 negative input
RA1/AN1/C12IN1-	3	RA1	General purpose I/O port A
		AN1	A/D Channel 1
		C12IN1-	Comparator C1 or C2 negative input
RA2/AN2/Vref-/CVref/C2IN+	4	RA2	General purpose I/O port A
		AN2	A/D Channel 2
		Vref-	A/D Negative Voltage Reference input
		CVref	Comparator Voltage Reference Output
		C2IN+	Comparator C2 Positive Input
RA3/AN3/Vref+/C1IN+	5	RA3	General purpose I/O port A
		AN3	A/D Channel 3
		Vref+	A/D Positive Voltage Reference Input
		C1IN+	Comparator C1 Positive Input
RA4/T0CKI/C1OUT	6	RA4	General purpose I/O port A
		T0CKI	Timer T0 Clock Input
		C1OUT	Comparator C1 Output
RA5/AN4/SS/C2OUT	7	RA5	General purpose I/O port A
		AN4	A/D Channel 4
		SS	SPI module Input (Slave Select)
RE0/AN5	8	C2OUT	Comparator C2 Output
		RE0	General purpose I/O port E
		AN5	A/D Channel 5
RE1/AN6	9	RE1	General purpose I/O port E
		AN6	A/D Channel 6
		AN7	A/D Channel 7
Vdd	11	+	Positive supply
Vss	12	-	Ground (GND)

Name	Number (DIP 40)	Function	Description
RA7/OSC1/CLKIN	13	RA7	General purpose I/O port A
		OSC1	Crystal Oscillator Input
		CLKIN	External Clock Input
RA6/OSC2/CLKOUT	14	OSC2	Crystal Oscillator Output
		CLKO	Fosc/4 Output
		RA6	General purpose I/O port A
RC0/T1OSO/T1CKI	15	RC0	General purpose I/O port C
		T1OSO	Timer T1 Oscillator Output
		T1CKI	Timer T1 Clock Input
RC1/T1OSO/T1CKI	16	RC1	General purpose I/O port C
		T1OSI	Timer T1 Oscillator Input
		CCP2	CCP1 and PWM1 module I/O
RC2/P1A/CCP1	17	RC2	General purpose I/O port C
		P1A	PWM Module Output
		CCP1	CCP1 and PWM1 module I/O
RC3/SCK/SCL	18	RC3	General purpose I/O port C
		SCK	MSSP module Clock I/O in SPI mode
		SCL	MSSP module Clock I/O in I <sup>2</sup> C mode
RD0	19	RD0	General purpose I/O port D
RD1	20	RD1	General purpose I/O port D
RD2	21	RD2	General purpose I/O port D
RD3	22	RD3	General purpose I/O port D
RC4/SDI/SDA	23	RC4	General purpose I/O port A
		SDI	MSSP module Data input in SPI mode
		SDA	MSSP module Data I/O in I <sup>2</sup> C mode
RC5/SDO	24	RC5	General purpose I/O port C
		SDO	MSSP module Data output in SPI mode
RC6/TX/CK	25	RC6	General purpose I/O port C
		TX	USART Asynchronous Output
		CK	USART Synchronous Clock
RC7/RX/DT	26	RC7	General purpose I/O port C
		RX	USART Asynchronous Input
		DT	USART Synchronous Data

Name	Number (DIP 40)	Function	Description
RD4	27	RD4	General purpose I/O port D
RD5/P1B	28	RD5	General purpose I/O port D
		P1B	PWM Output
RD6/P1C	29	RD6	General purpose I/O port D
		P1C	PWM Output
RD7/P1D	30	RD7	General purpose I/O port D
		P1D	PWM Output
Vss	31	-	Ground (GND)
Vdd	32	+	Positive Supply
RB0/AN12/INT	33	RB0	General purpose I/O port B
		AN12	A/D Channel 12
		INT	External Interrupt
RB1/AN10/C12INT3-	34	RB1	General purpose I/O port B
		AN10	A/D Channel 10
		C12INT3-	Comparator C1 or C2 Negative Input
RB2/AN8	35	RB2	General purpose I/O port B
		AN8	A/D Channel 8
RB3/AN9/PGM/C12IN2-	36	RB3	General purpose I/O port B
		AN9	A/D Channel 9
		PGM	Programming enable pin
		C12IN2-	Comparator C1 or C2 Negative Input
RB4/AN11	37	RB4	General purpose I/O port B
		AN11	A/D Channel 11
RB5/AN13/T1G	38	RB5	General purpose I/O port B
		AN13	A/D Channel 13
		T1G	Timer T1 External Input
RB6/ICSPCLK	39	RB6	General purpose I/O port B
		ICSPCLK	Serial programming Clock
RB7/ICSPDAT	40	RB7	General purpose I/O port B
		ICSPDAT	Programming enable pin

#### 4.1.2. Sơ đồ khối Pic 16f877a



Hình 4.1.Sơ đồ khối PIC16F877A

Đây là vi điều khiển thuộc họ PIC16Fxxx với tập lệnh gồm 35 lệnh có độ dài 14bit. Mỗi lệnh đều được thực thi trong một chu kì xung clock. Tốc độ hoạt động tối đa cho phép là 20 MHz với một chu kì lệnh là 200ns. Bộ nhớ chương trình 8Kx14 bit, bộ nhớ dữ liệu 368x8 byte RAM và bộ nhớ dữ liệu EEPROM với dung lượng 256x8 byte.

Số PORT I/O là 5 với 33 pin I/O.

Các đặc tính ngoại vi bao gồm các khối chức năng sau:

- ✓ Timer0: bộ đếm 8 bit với bộ chia tần số 8 bit.
- ✓ Timer1: bộ đếm 16 bit với bộ chia tần số, có thể thực hiện chức năng đếm dựa vào xung clock ngoại vi ngay khi vi điều khiển hoạt động ở chế độ sleep.
- ✓ Timer2: bộ đếm 8 bit với bộ chia tần số, bộ postcaler.
- ✓ Hai bộ Capture/so sánh/điều chế độ rộng xung.
- ✓ Các chuẩn giao tiếp nối tiếp SSP (Synchronous Serial Port), SPI và I2C.
- ✓ Chuẩn giao tiếp nối tiếp USART với 9 bit địa chỉ.

- ✓ Cổng giao tiếp song song PSP (Parallel Slave Port) với các chân điều khiển RD, WR, CS ở bên ngoài.
- ✓ Các đặc tính Analog: 8 kênh chuyên đổi ADC 10 bit. Hai bộ so sánh. Bên cạnh đó là một vài đặc tính khác của vi điều khiển như:
  - Bộ nhớ flash với khả năng ghi xóa được 100.000 lần.
  - Bộ nhớ EEPROM với khả năng ghi xóa được 1.000.000 lần.
  - Dữ liệu bộ nhớ EEPROM có thể lưu trữ trên 40 năm.
  - Khả năng tự nạp chương trình với sự điều khiển của phần mềm.
- ✓ Nạp được chương trình ngay trên mạch điện ICSP (In Circuit Serial Programming) thông qua 2 chân.
- ✓ Watchdog Timer với bộ dao động trong.
- ✓ Chức năng bảo mật mã chương trình.
- ✓ Chế độ Sleep.
- ✓ Có thể hoạt động với nhiều dạng Oscillator khác nhau.

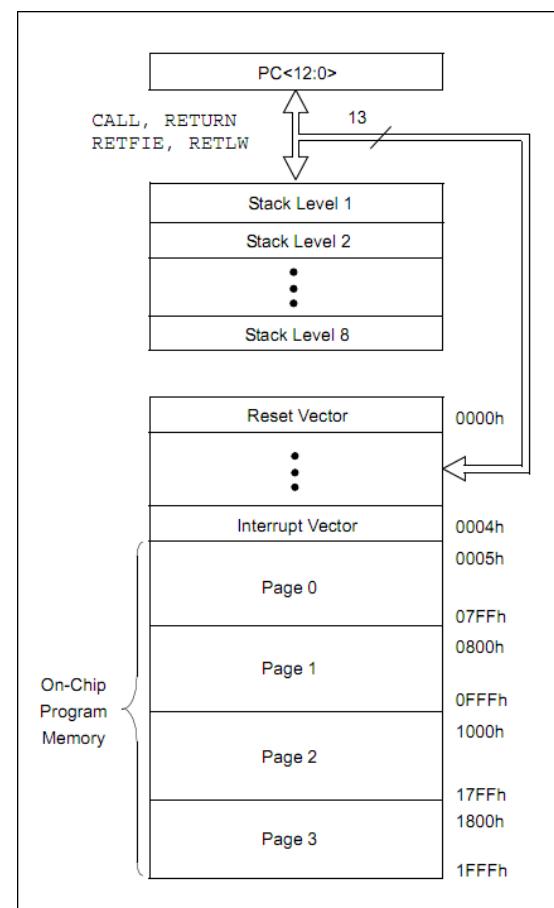
#### 4.1.3. Tổ chức bộ nhớ.

Cấu trúc bộ nhớ của vi điều khiển PIC16F877A bao gồm bộ nhớ chương trình (Program memory) và bộ nhớ dữ liệu (Data Memory).

#### 4.2. Bộ nhớ chương trình

Bộ nhớ chương trình của vi điều khiển PIC16F877A là bộ nhớ flash, dung lượng bộ nhớ 8K word (1 word = 14 bit) và được phân thành nhiều trang (từ page 0 đến page3). Như vậy bộ nhớ chương trình có khả năng chứa được  $8 \times 1024 = 8192$  lệnh (vì một lệnh sau khi mã hóa sẽ có dung lượng 1 word (14bit)). Để mã hóa được địa chỉ của 8K word bộ nhớ chương trình, bộ đếm chương trình có dung lượng 13 bit ( $PC<12:0>$ ). Khi vi điều khiển được reset, bộ đếm chương trình sẽ chỉ đến địa chỉ 0000h (Reset vector). Khi có ngắt xảy ra, bộ đếm chương trình sẽ chỉ đến địa chỉ 0004h (Interrupt vector).

Bộ nhớ chương trình không bao gồm bộ nhớ stack và không được địa chỉ hóa bởi bộ đếm chương trình. Bộ nhớ stack sẽ được đề cập cụ thể trong phần sau.



Hình 4.2.Bộ nhớ chương trình PIC16F877A

#### 4.2.1. Bộ nhớ dữ liệu

Bộ nhớ dữ liệu của PIC là bộ nhớ EEPROM được chia ra làm nhiều bank. Đối với PIC16F877A bộ nhớ dữ liệu được chia ra làm 4 bank. Mỗi bank có dung lượng 128 byte, bao

gồm các thanh ghi có chức năng đặc biệt SFG (Special Function Register) nằm ở các vùng địa chỉ thấp và các thanh ghi mục đích chung GPR (General Purpose Register) nằm ở vùng địa chỉ còn lại trong bank. Các thanh ghi SFR thường xuyên được sử dụng (ví dụ như thanh ghi STATUS) sẽ được đặt ở tất cả các bank của bộ nhớ dữ liệu giúp thuận tiện trong quá trình truy xuất và làm giảm bớt lệnh của chương trình. Sơ đồ cụ thể của bộ nhớ dữ liệu PIC16F877A như sau:

File Address	File Address	File Address	File Address
Indirect addr. (*)	00h	Indirect addr. (*)	80h
TMR0	01h	OPTION_REG	81h
PCL	02h	PCL	82h
STATUS	03h	STATUS	83h
FSR	04h	FSR	84h
PORTA	05h	TRISA	85h
PORTB	06h	TRISB	86h
PORTC	07h	TRISC	87h
PORTD <sup>(1)</sup>	08h	TRISD <sup>(1)</sup>	88h
PORTE <sup>(1)</sup>	09h	TRISE <sup>(1)</sup>	89h
PCLATH	0Ah	PCLATH	8Ah
INTCON	0Bh	INTCON	8Bh
PIR1	0Ch	PIE1	8Ch
PIR2	0Dh	PIE2	8Dh
TMR1L	0Eh	PCON	8Eh
TMR1H	0Fh		8Fh
T1CON	10h		90h
TMR2	11h	SSPCON2	91h
T2CON	12h	PR2	92h
SSPBUF	13h	SSPADD	93h
SSPCON	14h	SSPSTAT	94h
CCPR1L	15h		95h
CCPR1H	16h		96h
CCP1CON	17h		97h
RCSTA	18h	TXSTA	98h
TXREG	19h	SPBRG	99h
RCREG	1Ah		9Ah
CCPR2L	1Bh		9Bh
CCPR2H	1Ch	CMCON	9Ch
CCP2CON	1Dh	CVRCON	9Dh
ADRESH	1Eh	ADRESL	9Eh
ADCON0	1Fh	ADCON1	9Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h
		accesses 70h-7Fh	EFh
			F0h
			FFh
Bank 0	Bank 1	Bank 2	Bank 3

Hình 4.3. Bộ nhớ dữ liệu Pic 16f877a

#### **4.2.2. Thanh ghi chúa năng đặc biệt của SFR**

Đây là các thanh ghi được sử dụng bởi CPU hoặc được dùng để thiết lập và điều khiển các khối chức năng được tích hợp bên trong vi điều khiển. Có thể phân thành ghi SFR làm hai loại: thanh ghi SFR liên quan đến các chức năng bên trong (CPU) và thanh ghi SRF dùng để thiết lập và điều khiển các khối chức năng bên ngoài (ví dụ như ADC, PWM, ...). Phần này sẽ đề cập đến

các thanh ghi liên quan đến các chức năng bên trong. Các thanh ghi dùng để thiết lập và điều khiển các khôi chức năng sẽ được nhắc đến khi ta đề cập đến các khôi chức năng đó.

**Thanh ghi STATUS (03h, 83h, 103h, 183h):** thanh ghi chứa kết quả thực hiện phép toán của khôi ALU, trạng thái reset và các bit chọn bank cần truy xuất trong bộ nhớ dữ liệu.

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	TO	PD	Z	DC	C
bit 7							bit 0

Bit 7: IRP bit chọn bank bộ nhớ dữ liệu cần truy xuất (dùng cho địa chỉ gián tiếp).

IRP = 0: bank 2,3 (từ 100h đến 1FFh)

IRP = 1: bank 0,1 (từ 00h đến FFh)

Bit 6,5: RP1:RP0 hai bit chọn bank bộ nhớ dữ liệu cần truy xuất (dùng cho địa chỉ trực tiếp)

RP1:RP2	BANCK
00	0
01	1
10	2
11	3

Bit 4: TO bit chỉ thị trạng thái của WDT(Watch Dog Timer)

PD = 1 khi vi điều khiển vừa được cấp nguồn, hoặc sau khi lệnh CLRWDT hay

PD SLEEP được thực thi.

PD = 0 khi WDT bị tràn

Bit 3: TO bit chỉ thị trạng thái nguồn

= 1 khi vi điều khiển được cấp nguồn hoặc sau lệnh CLRWDT

= 0 sau khi lệnh SLEEP được thực thi

Bit 2: Z bit Zero

Z=1 khi kết quả của phép toán hay logic bằng 0

Z = 0 khi kết quả của phép toán hay logic khác 0

Bit 1: DC Digit carry/Borrow

DC = 1 khi kết quả phép toán tác động lên 4 bit thấp có nhô.

DC = 0 khi kết quả phép toán tác động lên 4 bit thấp không có nhô.

Bit 0: C Carry/borrow

C=1 khi kết quả phép toán tác động lên bit MSB có nhô.

C=0 khi kết quả phép toán tác động lên bit MSB không có nhô.

**Thanh ghi OPTION\_REG (81h, 181h):** thanh ghi này cho phép đọc và ghi, cho phép điều khiển chức năng pull-up của các chân trong PORTB, xác lập các tham số về xung tác động, cạnh tác động của ngắt ngoại vi và bộ đếm Timer0.

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

Bit7:	<b>RBU</b>	PORTE pull-up enable bit <b>RBU</b> = 1 không cho phép chức năng pull-up của PORTE <b>RBU</b> = 0 cho phép chức năng pull-up của PORTE																											
Bit 6:	INTEDG	Interrupt Edge Select bit INTEDG = 1 ngắt xảy ra khi cạnh dương chân RB0/INT xuất hiện. INTEDG = 0 ngắt xảy ra khi cạnh âm chân RB0/INT xuất hiện.																											
Bit 5	TOCS	Timer0 Clock Source select bit TOSC = 1 clock lấy từ chân RA4/TOCK1. TOSC = 0 dùng xung clock bên trong (xung clock này bằng với xung clock dùng để thực thi lệnh).																											
Bit 4	TOSE	Timer0 Source Edge Select bit TOSE = 1 tác động cạnh lên. TOSE = 0 tác động cạnh xuống.																											
Bit 3	PSA	Prescaler Assignment Select bit PSA = 1 bộ chia tần số (prescaler) được dùng cho WDT PSA = 0 bộ chia tần số được dùng cho Timer0																											
Bit 2:0	PS2:PS0	Prescaler Rate Select bit Các bit này cho phép thiết lập tỉ số chia tần số của Prescaler																											
		<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">Bit value</th> <th style="text-align: center;">TMR0 Rate</th> <th style="text-align: center;">WDT Rate</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">000</td><td style="text-align: center;">1:2</td><td style="text-align: center;">1:1</td></tr> <tr><td style="text-align: center;">001</td><td style="text-align: center;">1:4</td><td style="text-align: center;">1:2</td></tr> <tr><td style="text-align: center;">010</td><td style="text-align: center;">1:8</td><td style="text-align: center;">1:3</td></tr> <tr><td style="text-align: center;">011</td><td style="text-align: center;">1:16</td><td style="text-align: center;">1:8</td></tr> <tr><td style="text-align: center;">100</td><td style="text-align: center;">1:32</td><td style="text-align: center;">1:16</td></tr> <tr><td style="text-align: center;">101</td><td style="text-align: center;">1:64</td><td style="text-align: center;">1:32</td></tr> <tr><td style="text-align: center;">110</td><td style="text-align: center;">1:128</td><td style="text-align: center;">1:64</td></tr> <tr><td style="text-align: center;">111</td><td style="text-align: center;">1:256</td><td style="text-align: center;">1:128</td></tr> </tbody> </table>	Bit value	TMR0 Rate	WDT Rate	000	1:2	1:1	001	1:4	1:2	010	1:8	1:3	011	1:16	1:8	100	1:32	1:16	101	1:64	1:32	110	1:128	1:64	111	1:256	1:128
Bit value	TMR0 Rate	WDT Rate																											
000	1:2	1:1																											
001	1:4	1:2																											
010	1:8	1:3																											
011	1:16	1:8																											
100	1:32	1:16																											
101	1:64	1:32																											
110	1:128	1:64																											
111	1:256	1:128																											

**Thanh ghi INTCON (0Bh, 8Bh, 10Bh, 18Bh):** thanh ghi cho phép đọc và ghi, chứa các bit điều khiển và các bit cờ hiệu khi timer0 bị tràn, ngắt ngoại vi RB0/INT và ngắt interrupt-on-change tại các chân của PORTB.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	
bit 7								bit 0

Bit 7	GIE Global Interrupt Enable bit GIE = 1 cho phép tất cả các ngắt. GIE = 0 không cho phép tất cả các ngắt.
Bit 6	PEIE Peripheral Interrupt Enable bit PEIE = 1 cho phép tất cả các ngắt ngoại vi PEIE = 0 không cho phép tất cả các ngắt ngoại vi
Bit 5	TMR0IE Timer0 Overflow Interrupt Enable bit TMR0IE = 1 cho phép ngắt Timer0 TMR0IE = 0 không cho phép ngắt Timer0

Bit 4 RBIE RB0/INT External Interrupt Enable bit

RBIE = 1 cho phép ngắt ngoại vi RB0/INT

RBIE = 0 không cho phép ngắt ngoại vi RB0/INT

Bit 3 RBIE RB Port change Interrupt Enable bit

RBIE = 1 cho phép ngắt RB Port change

RBIE = 0 không cho phép ngắt RB Port change

Bit 2 TMR0IF Timer0 Interrupt Flag bit

TMR0IF = 1 thanh ghi TMR0 bị tràn (phải xóa bằng chương trình).

TMR0IF = 0 thanh ghi TMR0 chưa bị tràn.

Bit 1 INTF BR0/INT External Interrupt Flag bit

INTF = 1 ngắt RB0/INT xảy ra (phải xóa cờ hiệu bằng chương trình).

INTF = 0 ngắt RB0/INT chưa xảy ra.

Bit 0 RBIF RB Port Change Interrupt Flag bit

RBIF = 1 ít nhất có một chân RB7:RB4 có sự thay đổi trạng thái. Bit này phải được xóa bằng chương trình sau khi đã kiểm tra lại các giá trị của các chân tại PORTB.

RBIF = 0 không có sự thay đổi trạng thái các chân RB7:RB4.

**Thanh ghi PIE1 (8Ch):** chứa các bit điều khiển chi tiết các ngắt của các khối chức năng ngoại vi.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE <sup>(1)</sup>	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE

Bit 7 PSPIE Parallel Slave Port Read/Write Interrupt Enable bit

PSPIE = 1 cho phép ngắt PSP read/write.

PSPIE = 0 không cho phép ngắt PSP read/write.

Bit 6 ADIE ADC (A/D converter) Interrupt Enable bit

ADIE = 1 cho phép ngắt ADC.

ADIE = 0 không cho phép ngắt ADC.

Bit 5 RCIE USART Receive Interrupt Enable bit

RCIE = 1 cho phép ngắt nhận USART

RCIE = 0 không cho phép nhận USART

Bit 4 TXIE USART Transmit Interrupt Enable bit

TXIE = 1 cho phép ngắt truyền USART

TXIE = 0 không cho phép ngắt truyền USART

Bit 3 SSPIE Synchronous Serial Port Interrupt Enable bit

SSPIE = 1 cho phép ngắt SSP

SSPIE = 0 không cho phép ngắt SSP

Bit 2 CCP1IE CCP1 Interrupt Enable bit

CCP1IE = 1 cho phép ngắt CCP1

CCP1IE = 0 không cho phép ngắt CCP1

Bit 1 TMR2IE TMR2 to PR2 Match Interrupt Enable bit

TMR2IE = 1 cho phép ngắt.

TMR2IE = 0 không cho phép ngắt.

Bit 0 TMR1IE TMR1 Overflow Interrupt Enable bit

TMR1IE = 1 cho phép ngắt.

TMR1IE = 0 không cho phép ngắt.

**Thanh ghi PIR1 (0Ch):** chứa cờ ngắt của các khối chức năng ngoại vi, các ngắt

này được cho phép bởi các bit điều khiển chứa trong thanh ghi PIE1.

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF <sup>(1)</sup>	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7	bit 0						

Bit 7 PSPIF Parallel Slave Port Read/Write Interrupt Flag bit

PSPIF = 1 vừa hoàn tất thao tác đọc hoặc ghi PSP (phải xóa bằng chương trình).

PSPIF = 0 không có thao tác đọc ghi PSP nào diễn ra.

Bit 6 ADIF ADC Interrupt Flag bit

ADIF = 1 hoàn tất chuyển đổi ADC.

ADIF = 0 chưa hoàn tất chuyển đổi ADC.

Bit 5 RCIF USART Receive Interrupt Flag bit

RCIF = 1 buffer nhận qua chuẩn giao tiếp USART đã đầy.

RCIF = 0 buffer nhận qua chuẩn giao tiếp USART rỗng.

Bit 4 TXIF USART Transmit Interrupt Flag bit

TXIF = 1 buffer truyền qua chuẩn giao tiếp USART rỗng.

TXIF = 0 buffer truyền qua chuẩn giao tiếp USART đầy.

Bit 3 SSPIF Synchronous Serial Port (SSP) Interrupt Flag bit

SSPIF = 1 ngắt truyền nhận SSP xảy ra.

SSPIF = 0 ngắt truyền nhận SSP chưa xảy ra.

Bit 2 CCP1IF CCP1 Interrupt Flag bit

Khi CCP1 ở chế độ Capture

CCP1IF=1 đã cập nhật giá trị trong thanh ghi TMR1.

CCP1IF=0 chưa cập nhật giá trị trong thanh ghi TMR1.

Khi CCP1 ở chế độ Compare

CCP1IF=1 giá trị cần so sánh bằng với giá trị chứa trong TMR1

CCP1IF=0 giá trị cần so sánh không bằng với giá trị trong TMR1

Bit 0 TMR1IF TMR1 Overflow Interrupt Flag bit

TMR1IF = 1 thanh ghi TMR1 bị tràn (phải xóa bằng chương trình).

TMR1IF = 0 thanh ghi TMR1 chưa bị tràn.

**Thanh ghi PIE2 (8Dh):** chứa các bit điều khiển các ngắt của các khối chức năng CCP2, SSP bus, ngắt của bộ so sánh và ngắt ghi vào bộ nhớ EEPROM

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE
bit 7	bit 0						

Bit 7, 5, 2, 1 Không cần quan tâm và mặc định mang giá trị 0.

Bit 6 CMIE Comparator Interrupt Enable bit

CMIE = 1 Cho phép ngắt của bộ so sánh.

CMIE = 0 Không cho phép ngắt.

Bit 4 EEIE EEPROM Write Operation Interrupt Enable bit

EEIE = 1 Cho phép ngắt khi ghi dữ liệu lên bộ nhớ EEPROM.

EEIE = 0 Không cho phép ngắt khi ghi dữ liệu lên bộ nhớ EEPROM.

Bit 3 BCLIE Bus Collision Interrupt Enable bit

BCLIE = 1 Cho phép ngắt.

BCLIE = 0 Không cho phép ngắt.

Bit 0 CCP2IE CCP2 Interrupt Enable bit

CCP2IE = 1 Cho phép ngắt.

CCP2IE = 0 Không cho phép ngắt.

**Thanh ghi PIR2 (0Dh):** chứa các cờ ngắt của các khối chức năng ngoại vi, các ngắt này được cho phép bởi các bit điều khiển chứa trong thanh ghi PIE2.

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF

bit 7

bit 0

Bit 7, 5, 2, 1: không quan tâm và mặc định mang giá trị 0.

Bit 6 CMIF Comparator Interrupt Flag bit

CMIF = 1 tín hiệu ngõ vào bộ so sánh thay đổi.

CMIF = 0 tín hiệu ngõ vào bộ so sánh không thay đổi.

Bit 4 EEIF EEPROM Write Operation Interrupt Flag bit

EEIF = 1 quá trình ghi dữ liệu lên EEPROM hoàn tất.

EEIF = 0 quá trình ghi dữ liệu lên EEPROM chưa hoàn tất hoặc chưa bắt đầu.

Bit 3 BCLIF Bus Collision Interrupt Flag bit

BCLIF = 1 Bus truyền nhận đang bận khi (đang có dữ liệu truyền đi trong bus) khi SSP hạt động ở chế độ I2C Master mode.

BCLIF = 0 Bus truyền nhận chưa bị tràn (không có dữ liệu truyền đi trong bus).

Bit 0 CCP2IF CCP2 Interrupt Flag bit

Ở chế độ Capture

CCP2IF = 1 đã cập nhật giá trị trong thanh ghi TMR1.

CCP2IF = 0 chưa cập nhật giá trị trong thanh ghi TMR1.

Ở chế độ Compare

CCP2IF = 1 giá trị cần so sánh bằng với giá trị chứa trong TMR1.

CCP2IF = 0 giá trị cần so sánh chưa bằng với giá trị chứa trong TMR1.

**Thanh ghi PCON (8Eh):** chứa các cờ hiệu cho biết trạng thái các chế độ reset của vi điều khiển.

U-0	U-0	U-0	U-0	U-0	U-0	R/W-0	R/W-1
—	—	—	—	—	—	POR	BOR

bit 7

bit 0

Bit 7, 6, 5, 4, 3, 2 Không cần quan tâm và mặc định mang giá trị 0.

Bit 1 POR Power-on Reset Status bit

POR = 1 không có sự tác động của Power-on Reset.

POR = 0 có sự tác động của Power-on reset.

Bit 0 POR Brown-out Reset Status bit

POR = 1 không có sự tác động của Brown-out reset.

POR = 0 có sự tác động của Brown-out reset.

#### 4.2.3. Thanh ghi mục đích GPR

Các thanh ghi này có thể được truy xuất trực tiếp hoặc gián tiếp thông qua thanh ghi FSG (File Select Register). Đây là các thanh ghi dữ liệu thông thường, người sử dụng có thể tùy theo mục đích chương trình mà có thể dùng các thanh ghi này để chia các biến số, hằng số, kết quả hoặc các tham số phục vụ cho chương trình.

#### 4.2.4. STACK

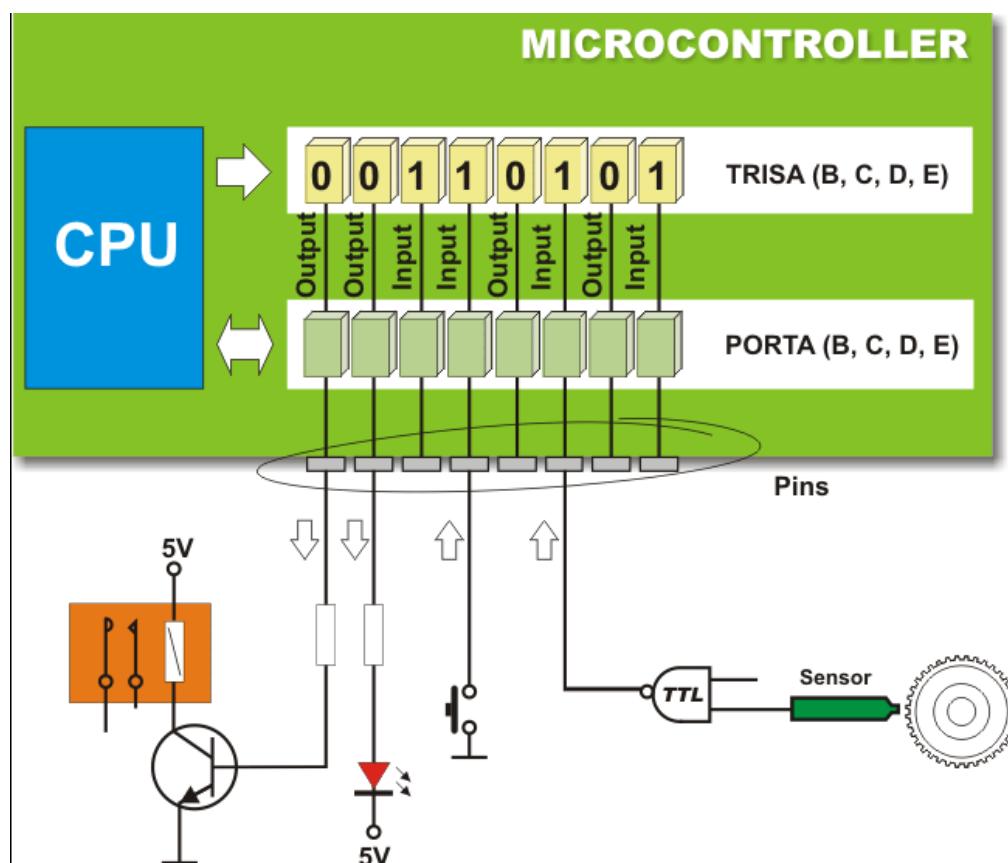
Stack không nằm trong bộ nhớ chương trình hay bộ nhớ dữ liệu mà là một vùng nhớ đặc biệt không cho phép đọc hay ghi. Khi lệnh CALL được thực hiện hay khi một ngắt xảy ra làm chương trình bị rẽ nhánh, giá trị của bộ đếm chương trình PC tự động được vi điều khiển cất vào trong stack. Khi một trong các lệnh RETURN, RETLW hay RETFIE được thực thi, giá trị PC sẽ tự động được lấy ra từ trong stack, vi điều khiển sẽ thực hiện tiếp chương trình theo đúng qui trình định trước. Bộ nhớ Stack trong vi điều khiển PIC họ 16F87xA có khả năng chứa được 8 địa chỉ và hoạt động theo cơ chế xoay vòng. Nghĩa là giá trị cất vào bộ nhớ Stack lần thứ 9 sẽ ghi đè lên giá trị cất vào Stack lần đầu tiên và giá trị cất vào bộ nhớ Stack lần thứ 10 sẽ ghi đè lên giá trị cất vào Stack lần thứ 2.

Cần chú ý là không có cờ hiệu nào cho biết trạng thái stack, do đó ta không biết được khi

nào stack tràn. Bên cạnh đó tập lệnh của vi điều khiển dòng PIC cũng không có lệnh POP hay PUSH, các thao tác với bộ nhớ stack sẽ hoàn toàn được điều khiển bởi CPU.

#### 4.3. Các cổng xuất nhập của PIC16F877A.

Cổng xuất nhập (I/O port) chính là phương tiện mà vi điều khiển dùng để tương tác với thế giới bên ngoài. Sự tương tác này rất đa dạng và thông qua quá trình tương tác đó, chức năng của vi điều khiển được thể hiện một cách rõ ràng. Một cổng xuất nhập của vi điều khiển bao gồm nhiều chân (I/O pin), tùy theo cách bố trí và chức năng của vi điều khiển mà số lượng cổng xuất nhập và số lượng chân trong mỗi cổng có thể khác nhau. Bên cạnh đó, do vi điều khiển được tích hợp sẵn bên trong các đặc tính giao tiếp ngoại vi nên bên cạnh chức năng là cổng xuất nhập thông thường, một số chân xuất nhập còn có thêm các chức năng khác để thể hiện sự tác động của các đặc tính ngoại vi nêu trên đối với thế giới bên ngoài. Chức năng của từng chân xuất nhập trong mỗi cổng hoàn toàn có thể được xác lập và điều khiển được thông qua các thanh ghi SFR liên quan đến chân xuất nhập đó. Vi điều khiển PIC16F877A có 5 cổng xuất nhập, bao gồm PORTA, PORTB, PORTC, PORTD và PORTE. Cấu trúc và chức năng của từng cổng xuất nhập sẽ được đề cập cụ thể trong phần sau.



Hình 4.3. Xuất nhập vi điều khiển

##### 4.3.1. PORTA (Địa chỉ 05h):

PORTA (RPA) bao gồm 6 I/O pin. Đây là các chân “hai chiều” (bidirectional pin), nghĩa là có thể xuất và nhập được. Chức năng I/O này được điều khiển bởi thanh ghi TRISA (địa chỉ 85h). Muốn xác lập chức năng của một chân trong PORTA là input, ta “set” bit điều khiển tương ứng với chân đó trong thanh ghi TRISA và ngược lại, muốn xác lập chức năng của một chân trong PORTA là output, ta “clear” bit điều khiển tương ứng với chân đó trong thanh ghi TRISA. Thao tác này hoàn toàn tương tự đối với các PORT và các thanh ghi điều khiển tương ứng TRIS (đối với PORTA là TRISA, đối với PORTB là TRISB, đối với PORTC là TRISC, đối

với PORTD là TRISD và đối với PORTE là TRISE). Bên cạnh đó PORTA còn là ngõ ra của bộ ADC, bộ so sánh, ngõ vào analog ngõ vào xung clock của Timer0 và ngõ vào của bộ giao tiếp MSSP (Master Synchronous Serial Port). Đặc tính này sẽ được trình bày cụ thể trong phần sau.

### Các thanh ghi SFR liên quan đến PORTA bao gồm:

PORTA (địa chỉ 05h) : chứa giá trị các pin trong PORTA

TRISA (địa chỉ 85h) : điều khiển xuất nhập

CMCON (địa chỉ 9Ch) : thanh ghi điều khiển bộ so sánh.

CVRCON (địa chỉ 9Dh) : thanh ghi điều khiển bộ so sánh điện áp

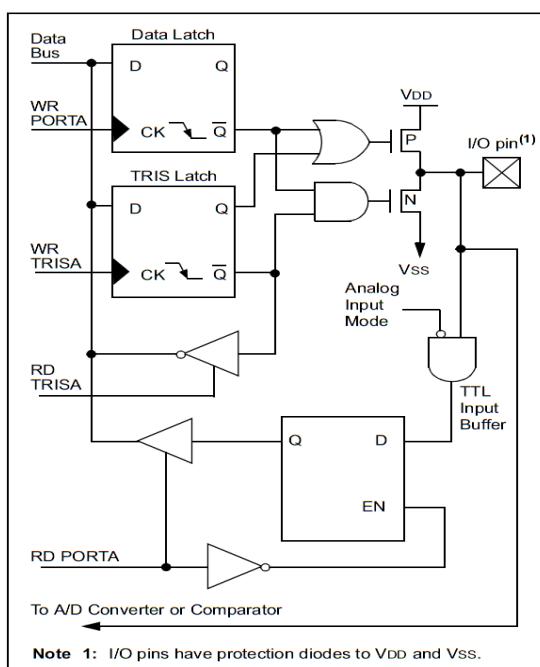
ADCON1 (địa chỉ 9Fh) : thanh ghi điều khiển bộ ADC.

Name	Bit#	Buffer	Function
RA0/AN0	bit 0	TTL	Input/output or analog input.
RA1/AN1	bit 1	TTL	Input/output or analog input.
RA2/AN2/VREF-/CVREF	bit 2	TTL	Input/output or analog input or VREF- or CVREF.
RA3/AN3/VREF+	bit 3	TTL	Input/output or analog input or VREF+.
RA4/T0CKI/C1OUT	bit 4	ST	Input/output or external clock input for Timer0 or comparator output. Output is open-drain type.
RA5/AN4/SS/C2OUT	bit 5	TTL	Input/output or analog input or slave select input for synchronous serial port or comparator output.

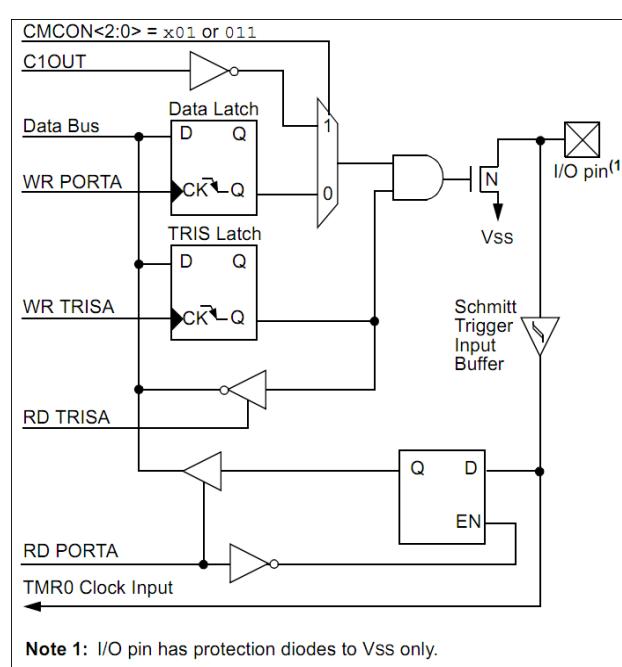
Legend: TTL = TTL input, ST = Schmitt Trigger input

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
05h	PORTA	—	—	RA5	RA4	RA3	RA2	RA1	RA0	--0x 0000	--0u 0000
85h	TRISA	—	—	PORTA Data Direction Register						--11 1111	--11 1111
9Ch	CMCON	C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0	0000 0111	0000 0111
9Dh	CVRCON	CVREN	CVROE	CVRR	—	CVR3	CVR2	CVR1	CVR0	000- 0000	000- 0000
9Fh	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	00-- 0000

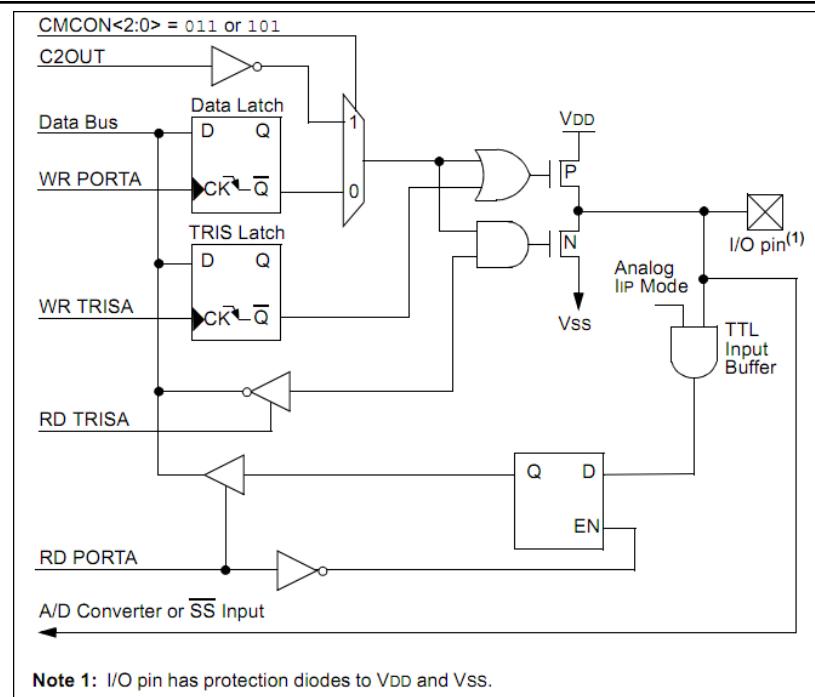
Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'. Shaded cells are not used by PORTA.



Sơ đồ khối RA3:RA0



Sơ đồ khối RA4



Sơ đồ khối RA5

#### 4.3.2. PORTB (địa chỉ 06h, 106h).

PORTB (RPB) gồm 8 pin I/O. Thanh ghi điều khiển xuất nhập tương ứng là TRISB. Bên cạnh đó một số chân của PORTB còn được sử dụng trong quá trình nạp chương trình cho vi điều khiển với các chế độ nạp khác nhau. PORTB còn liên quan đến ngắt ngoại vi và bộ Timer0. PORTB còn được tích hợp chức năng điện trở kéo lên được điều khiển bởi chương trình.

Các thanh ghi SFR liên quan đến PORTB bao gồm:

PORTB (địa chỉ 06h, 106h) : chứa giá trị các pin trong PORTB

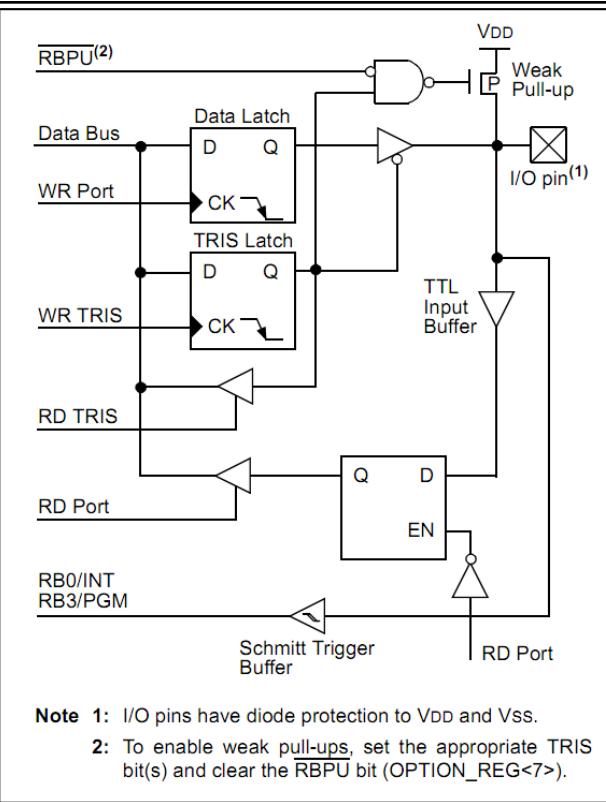
TRISB (địa chỉ 86h, 186h) : điều khiển xuất nhập

OPTION\_REG (địa chỉ 81h, 181h): điều khiển ngắt ngoại vi và bộ Timer0.

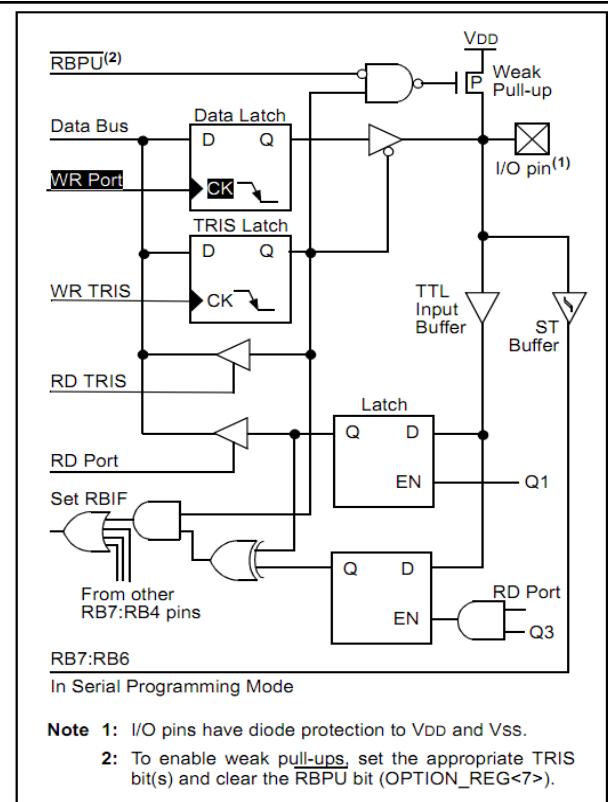
Name	Bit#	Buffer	Function
RB0/INT	bit 0	TTL/ST <sup>(1)</sup>	Input/output pin or external interrupt input. Internal software programmable weak pull-up.
RB1	bit 1	TTL	Input/output pin. Internal software programmable weak pull-up.
RB2	bit 2	TTL	Input/output pin. Internal software programmable weak pull-up.
RB3/PGM <sup>(3)</sup>	bit 3	TTL	Input/output pin or programming pin in LVP mode. Internal software programmable weak pull-up.
RB4	bit 4	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB5	bit 5	TTL	Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up.
RB6/PGC	bit 6	TTL/ST <sup>(2)</sup>	Input/output pin (with interrupt-on-change) or in-circuit debugger pin. Internal software programmable weak pull-up. Serial programming clock.
RB7/PGD	bit 7	TTL/ST <sup>(2)</sup>	Input/output pin (with interrupt-on-change) or in-circuit debugger pin. Internal software programmable weak pull-up. Serial programming data.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
06h, 106h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx	uuuu uuuu
86h, 186h	TRISB	PORTB Data Direction Register								1111 1111	1111 1111
81h, 181h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

Legend: x = unknown, u = unchanged. Shaded cells are not used by PORTB.



Sơ đồ khối RB3:RB0



Sơ đồ khối RB7:RB4

#### 4.3.3. PORTC (Địa chỉ 07h):

PORTC (RPC) gồm 8 pin I/O. Thanh ghi điều khiển xuất nhập tương ứng là TRISC. Bên cạnh đó PORTC còn chứa các chân chức năng của bộ so sánh, bộ Timer1, bộ PWM và các chuẩn giao tiếp nối tiếp I2C, SPI, SSP, USART.

Các thanh ghi điều khiển liên quan đến PORTC:

PORTC (địa chỉ 07h) : chứa giá trị các pin trong PORTC

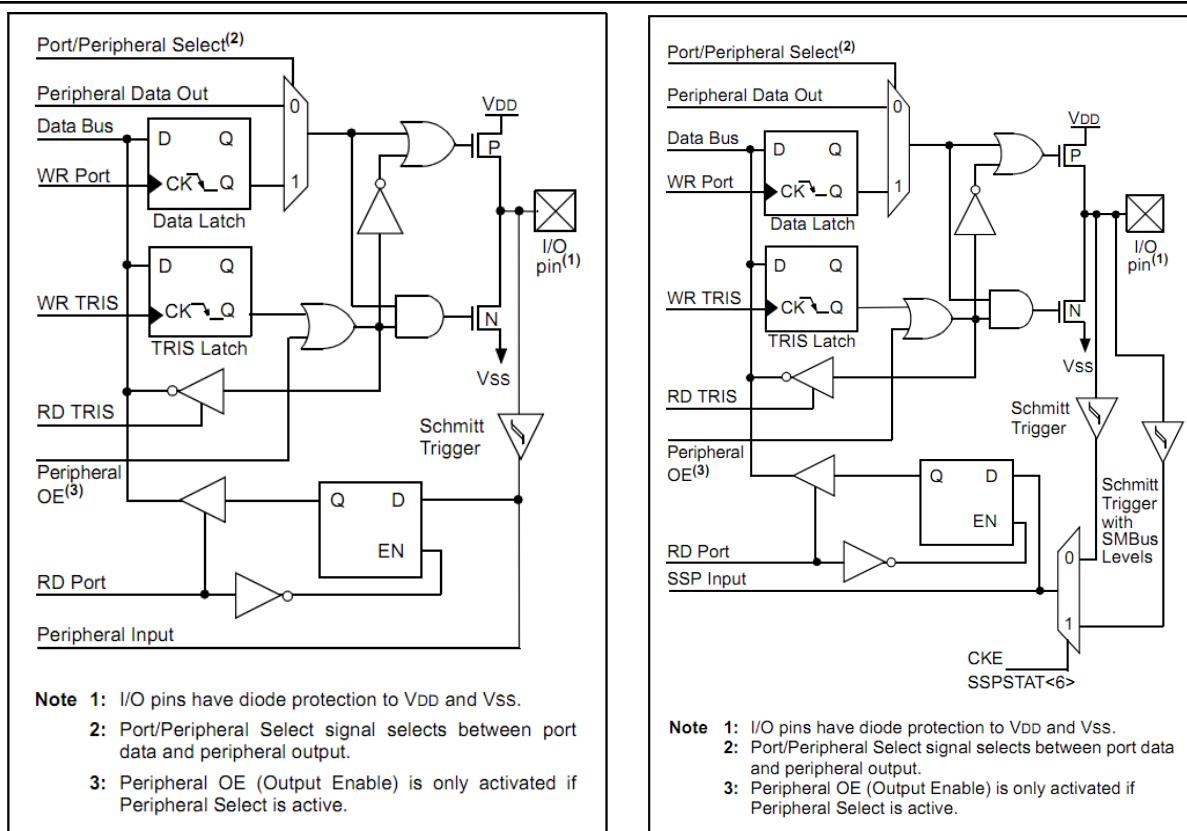
TRISC (địa chỉ 87h) : điều khiển xuất nhập.

Name	Bit#	Buffer Type	Function
RC0/T1OSO/T1CKI	bit 0	ST	Input/output port pin or Timer1 oscillator output/Timer1 clock input.
RC1/T1OSI/CCP2	bit 1	ST	Input/output port pin or Timer1 oscillator input or Capture2 input/ Compare2 output/PWM2 output.
RC2/CCP1	bit 2	ST	Input/output port pin or Capture1 input/Compare1 output/ PWM1 output.
RC3/SCK/SCL	bit 3	ST	RC3 can also be the synchronous serial clock for both SPI and I <sup>2</sup> C modes.
RC4/SDI/SDA	bit 4	ST	RC4 can also be the SPI data in (SPI mode) or data I/O (I <sup>2</sup> C mode).
RC5/SDO	bit 5	ST	Input/output port pin or Synchronous Serial Port data output.
RC6/TX/CK	bit 6	ST	Input/output port pin or USART asynchronous transmit or synchronous clock.
RC7/RX/DT	bit 7	ST	Input/output port pin or USART asynchronous receive or synchronous data.

Legend: ST = Schmitt Trigger input

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
07h	PORTC	RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0	xxxx xxxx	uuuu uuuu
87h	TRISC	PORTC Data Direction Register								1111 1111	1111 1111

Legend: x = unknown, u = unchanged



Sơ đồ khối RC7:RC5 và RC2:RC0

Sơ đồ khối RC4:RC3

#### 4.3.4. PORTD (Địa chỉ 08h):

PORTD (RPD) gồm 8 chân I/O, thanh ghi điều khiển xuất nhập tương ứng là TRISD. PORTD còn là cổng xuất dữ liệu của chuẩn giao tiếp PSP (Parallel Slave Port).

Các thanh ghi liên quan đến PORTD bao gồm:

PORTD : chứa giá trị các pin trong PORTD.

TRISD : điều khiển xuất nhập.

TRISE : điều khiển xuất nhập PORTE và chuẩn giao tiếp PSP.

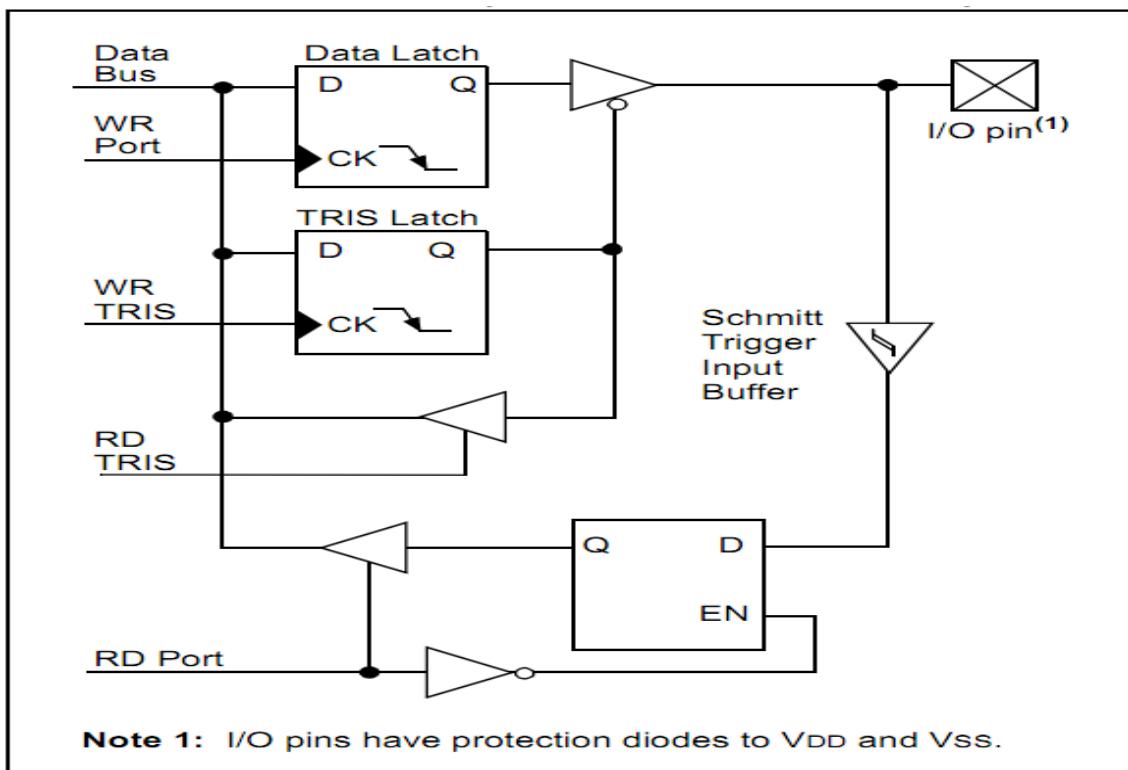
Name	Bit#	Buffer Type	Function
RD0/PSP0	bit 0	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 0.
RD1/PSP1	bit 1	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 1.
RD2/PSP2	bit2	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 2.
RD3/PSP3	bit 3	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 3.
RD4/PSP4	bit 4	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 4.
RD5/PSP5	bit 5	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 5.
RD6/PSP6	bit 6	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 6.
RD7/PSP7	bit 7	ST/TTL <sup>(1)</sup>	Input/output port pin or Parallel Slave Port bit 7.

Legend: ST = Schmitt Trigger input, TTL = TTL input

Note 1: Input buffers are Schmitt Triggers when in I/O mode and TTL buffers when in Parallel Slave Port mode.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
08h	PORTD	RD7	RD6	RD5	RD4	RD3	RD2	RD1	RD0	xxxx xxxx	uuuu uuuu
88h	TRISD	PORTD Data Direction Register								1111 1111	1111 1111
89h	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction Bits	0000 -111	0000 -111		

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PORTD.



Sơ đồ khối RD7:RD0

#### 4.3.5. PORTE (Địa chỉ 09h):

PORTE (RPE) gồm 3 chân I/O. Thanh ghi điều khiển xuất nhập tương ứng là TRISE. Các chân của PORTE có ngõ vào analog. Bên cạnh đó PORTE còn là các chân điều khiển của chuẩn giao tiếp PSP.

Các thanh ghi liên quan đến PORTE bao gồm:

PORTE : chứa giá trị các chân trong PORTE.

TRISE : điều khiển xuất nhập và xác lập các thông số cho chuẩn giao tiếp PSP.

ADCON1 : thanh ghi điều khiển khối ADC.

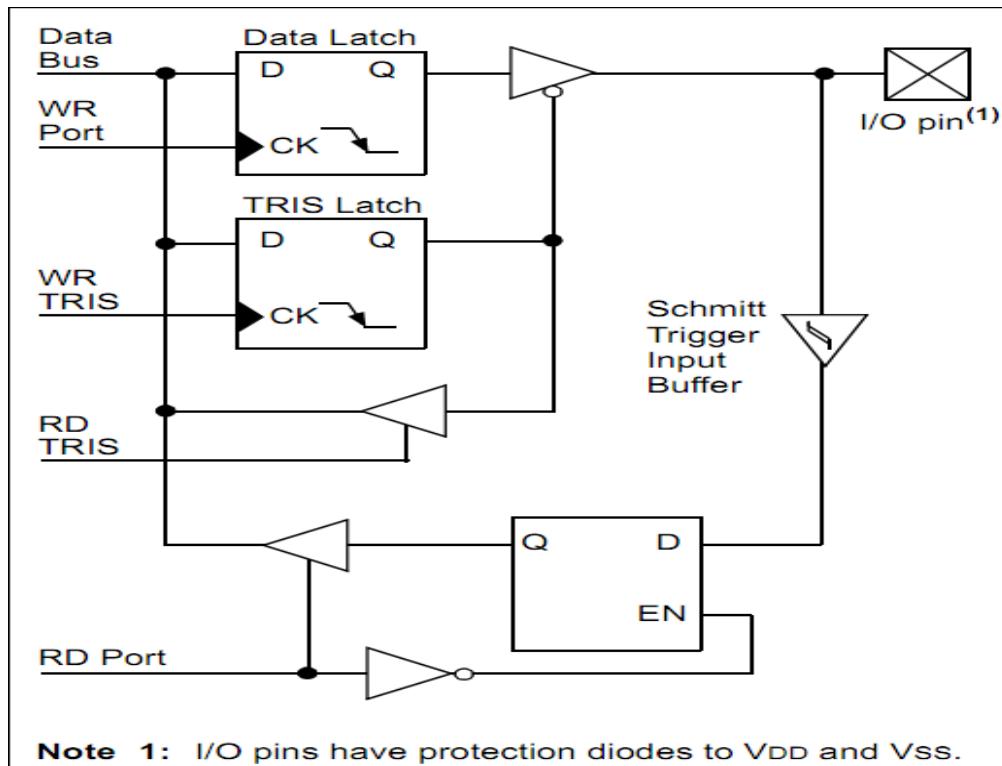
Name	Bit#	Buffer Type	Function
RE0/RD/AN5	bit 0	ST/TTL <sup>(1)</sup>	I/O port pin or read control input in Parallel Slave Port mode or analog input: RD 1 = Idle 0 = Read operation. Contents of PORTD register are output to PORTD I/O pins (if chip selected).
RE1/WR/AN6	bit 1	ST/TTL <sup>(1)</sup>	I/O port pin or write control input in Parallel Slave Port mode or analog input: WR 1 = Idle 0 = Write operation. Value of PORTD I/O pins is latched into PORTD register (if chip selected).
RE2/CS/AN7	bit 2	ST/TTL <sup>(1)</sup>	I/O port pin or chip select control input in Parallel Slave Port mode or analog input: CS 1 = Device is not selected 0 = Device is selected

Legend: ST = Schmitt Trigger input, TTL = TTL input

Note 1: Input buffers are Schmitt Triggers when in I/O mode and TTL buffers when in Parallel Slave Port mode.

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other Resets
09h	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -xxx	---- -uuu
89h	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction bits			0000 -111	0000 -111
9Fh	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	00-- 0000

Legend:  $x$  = unknown,  $u$  = unchanged,  $-$  = unimplemented, read as '0'. Shaded cells are not used by PORTE.

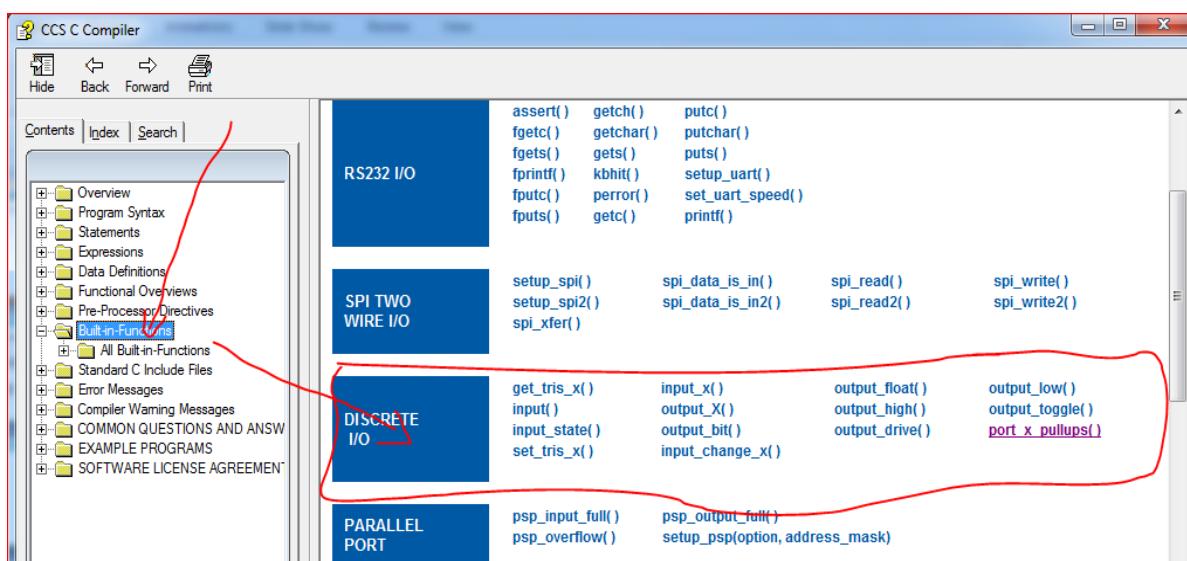


### Sơ đồ khói RE2:RE0

#### 4.3.6. Sử dụng hàm trong CCS:

Để sử dụng các hàm trong CCS ta vào phần Help (hoặc nhấn F12) sẽ hiện ra bảng CCS C Compiler. Đây là bảng chứa mọi câu lệnh, hướng dẫn giúp người dùng thao tác dễ dàng và tận dụng hết sức mạnh của phần mềm CCS mang lại.

Để sử dụng các hàm I/O của CCS bạn vào Built in-Funtion => Discrete I/O.



Hình 4.4. Các hàm CCS các PORT

➤ **Output\_low ( pin ) , Output\_high (pin) :**

- ✓ Dùng thiết lập mức 0 (low, 0V) hay mức 1 ( high, 5V ) cho chân IC , pin chỉ vị trí chân .
- ✓ Hàm này sẽ đặt pin làm ngõ ra.
- ✓ Hàm này dài 2-4 chu kỳ máy

➤ **Output\_bit (pin , value) :**

- ✓ **pin** : tên chân value : giá trị 0 hay 1
- ✓ Hàm này cũng xuất giá trị 0 / 1 trên pin , tương tự 2 hàm trên . Thường dùng nó khi giá trị ra tuỳ thuộc giá trị biến 1 bit nào đó , hay muốn xuất đảo của giá trị ngõ ra trước đó .
- ✓ **VD :** output\_bit( pin\_B0 , !x );

➤ **Output\_float ( pin ) :**

- ✓ Hàm này set pin như ngõ vào , cho phép pin ở mức cao như 1 cực thu hở.

➤ **Input ( pin ) :**

- ✓ Hàm này trả về giá trị 0 hay 1 là trạng thái của chân IC . Giá trị là 1 bit

➤ **Output\_X ( value ) :**

- ✓ X là tên port có trên chip . Value là giá trị 1 byte .
- ✓ Hàm này xuất giá trị 1 byte ra port . Tất cả chân của port đó đều là ngõ ra .
- ✓ **Ví dụ:** Output\_B ( 255 ) ; // xuất giá trị 11111111 ra port B

➤ **Input\_X () :**

- ✓ X : là tên port ( a, b ,c ,d e ).
- ✓ Hàm này trả về giá trị 8 bit là giá trị đang hiện hữu của port đó .
- ✓ **ví dụ :**

```
Int8 m;
m=input_E();
```

➤ **Port\_B\_pullups (value) :**

- ✓ Hàm này thiết lập ngõ vào port B pullup (điện trở kéo lên) . Value =1 sẽ kích hoạt tính năng này và value =0 sẽ ngừng .
- ✓ Đối pic 16F877A chỉ có port B có tính năng này mới dùng hàm này .

➤ **input\_change\_x();**

- ✓ X : là tên port ( a, b ,c ,d e ).
- ✓ Hàm này trả về giá trị 1byte. Sẽ so sánh với lần gọi hàm nhất. 1 nếu giá trị thay đổi, 0 nếu không thay đổi.

➤ **Set\_tris\_X ( value ) :**

- ✓ Hàm này định nghĩa chân IO cho 1 port là ngõ vào hay ngõ ra. Chỉ được dùng với #use fast\_IO . Sử dụng#byte để tạo biến chỉ đến port và thao tác trên biến này chính là thao tác trên port .
- ✓ Value là giá trị 8 bit . Mỗi bit đại diện 1 chân và bit=0 sẽ set chân đó là ngõ vào, bit= 1 set chân đó là ngõ ra .

➤ **#use FAST\_IO(X) ;**

- ✓ Thiết lập chế độ fast I/O cho PORTX, Yêu cầu phải chỉ rõ hướng vào ra bằng lệnh **Set\_tris\_X ( value )** cho các I/O. (Chỉ mất 1 chu kỳ máy).

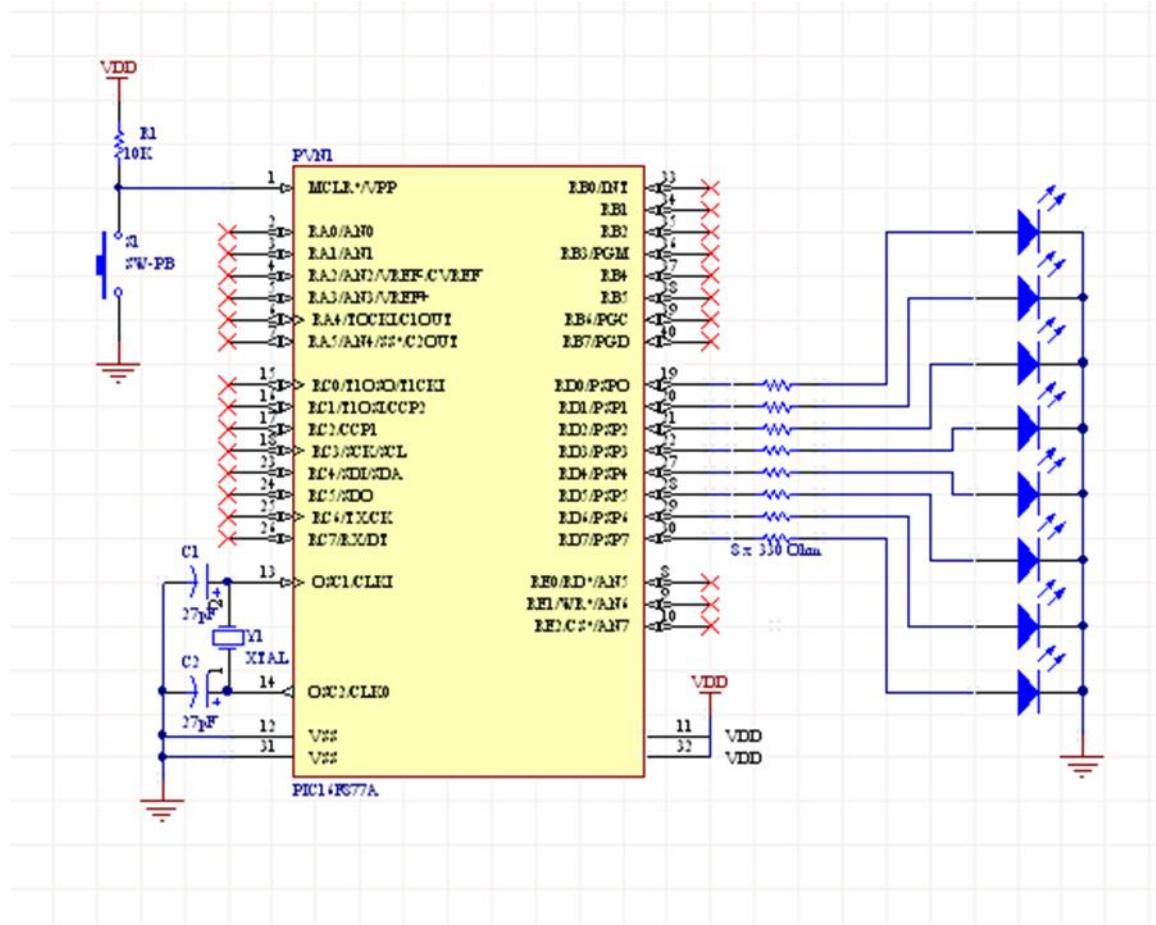
➤ **#use STANDARD\_IO(X)**

- ✓ Thiết lập chuẩn I/O không yêu cầu ngõ vào ra.(Các hàm I/O mất 3-4 chu kỳ lệnh).

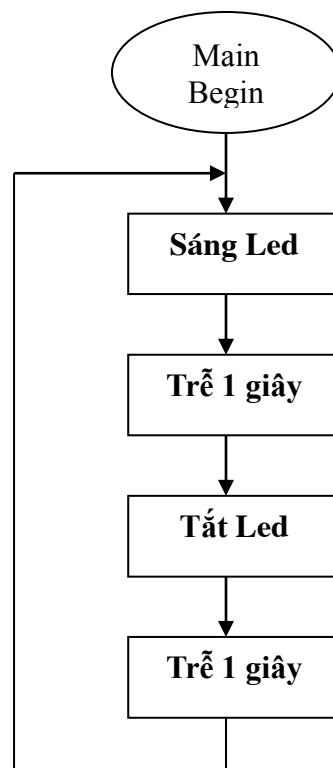
#### **4.3.7. Bài tập về I/O.**

a) Viết chương trình tạo tần số 1Hz tại PortD.

Ở ví dụ này tôi sẽ hướng dẫn các bạn làm theo 2 cách là sử dụng các câu lệnh của CCS đã cho như đã trình bày ở trên và cách khác tiện dụng hơn, thể hiện được đặc cách của người lập trình. Về phần mô phỏng thì các bạn xem thêm tài liệu về phần mềm Proteus nên tôi sẽ không hướng



✓ Lưu đồ giải thuật:



- ✓ Code cách 1 (ở đây tôi bỏ qua phần header để nhìn đơn giản. Xem lại phần giới thiệu CCS) Sauk hi viết xong nhấn F9 để Compiler và dung Proteus để mô phỏng.

```
#include <16F877A.h>
```

```
#device ADC=16
```

```
#FUSES NOWDT          //No Watch Dog Timer
#FUSES NOBROWNOUT    //No brownout reset
#FUSES NOLVP          //No low voltage prgming, B3(PIC16) or B5(PIC18) used
for I/O
```

```
#use delay(crystal=20000000)
```

```
#define DELAY 500
```

```
void main()
```

```
{
```

```
//Example blinking LED program
```

```
while(true)
```

```
{
```

```
    OUTPUT_D(0x0);
```

```
    delay_ms(DELAY);
```

```
    OUTPUT_D(0xff);
```

```
    delay_ms(DELAY);
```

```
}
```

```
}
```

- ✓ Code cách 2: Ở cách này cũng là cách mà tôi sẽ sử dụng để viết code sau này.

```
#include <16F877A.h>
```

```
#include "def_877a.h"
```

```
#device ADC=16
```

```
#FUSES NOWDT          //No Watch Dog Timer
```

```
#FUSES NOBROWNOUT    //No brownout reset
```

```
#FUSES NOLVP          //No low voltage prgming, B3(PIC16) or B5(PIC18) used
for I/O
```

```
#use delay(crystal=20000000)
```

```
#define DELAY 500
```

```
void main()
```

```
{
```

```
    Trisd=0x00;// cong ra
```

```
//Example blinking LED program
```

```
while(true)
```

```

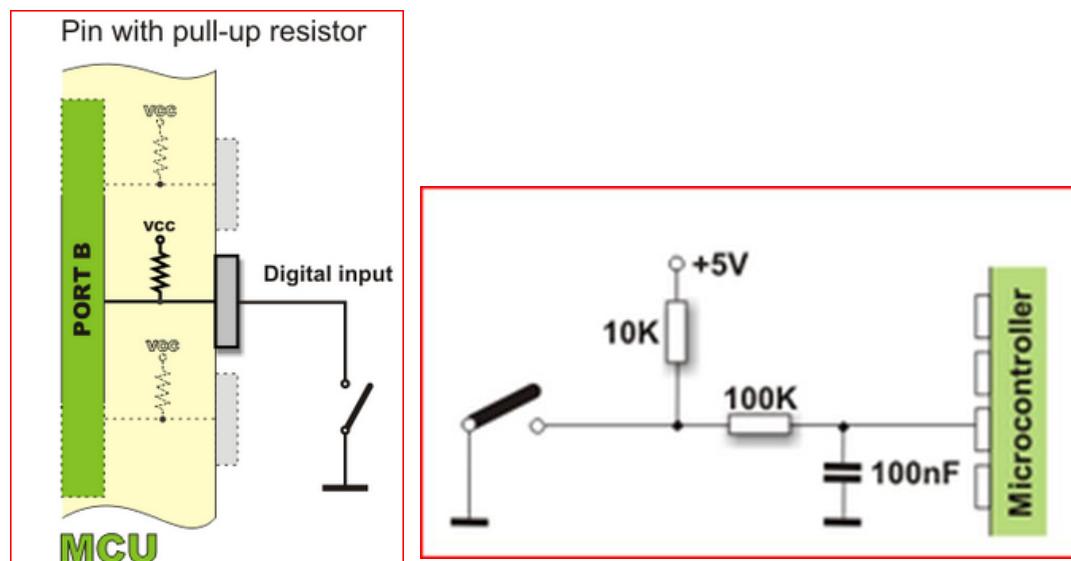
    }
    Portd=0x00;
    delay_ms(DELAY);
    portd=0xff;
    delay_ms(DELAY);
}
}

```

=>**Chú ý:** Như ở cách 2 các bạn chú ý cho tôi dòng: #include "def\_877a.h" đây là file mà người dùng đã định nghĩa hết các thanh ghi, các địa chỉ trong PIC. File này các bạn có thể xem thêm ở mục phụ lục.

Câu lệnh “Trisb=0x00;” đây là câu lệnh điều hướng cho port sử dụng xuất hay nhập các bạn xem lại phần xuất nhập port. Thực chất khi bạn sử dụng câu lệnh “OUTPUT\_B(0x0);” là phần mềm CCS đã chạy tới 2 câu lệnh “Trisb=0x00; và Portb=0x00;” nên sẽ làm tốn chu kỳ thực hiện của vi điều khiển vì câu lệnh trisb bạn chỉ cần chạy 1 lần là đủ cho đến khi có sự thay đổi I/O.

- b) Viết chương trình kiểm tra trạng thái Pin B7 nếu B7==0 thì sáng Led PortD, ngược lại tắt Led Portd.  
 ✓ Phần cứng input



TH1

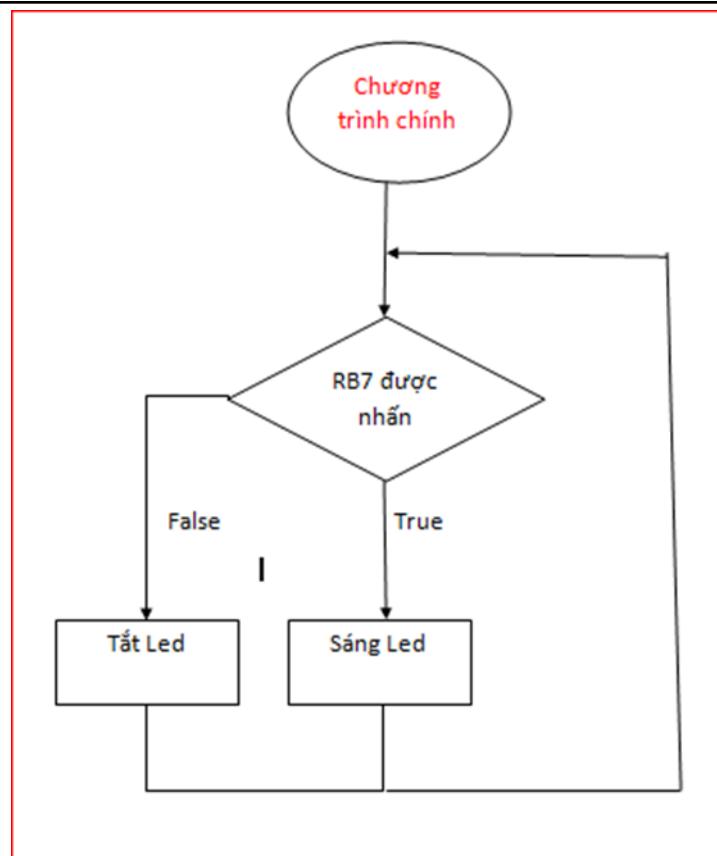
- ✓ Lưu ý khi kết nối phần cứng chế độ Input:

Đối với thiết kế phần cứng ở portb thì bạn chỉ cần kết nối theo TH1. Vì mặc định portb đã có điện trở nội kéo lên (pull-up) xem lại Portb.

Đối với các port khác khi dùng chế độ input các chân ở trạng thái nổi nên bạn cần thiết kế theo TH2. Tụ 100nF dùng để chống nhiễu cho vi điều khiển có thể có hoặc không.

- ✓ Lưu đồ giải thuật:

TH2



✓ Code chương trình:

```

#include <16F877A.h>
#include "def_877a.h"
#device ADC=16

#FUSES NOWDT          //No Watch Dog Timer
#FUSES NOBROWNOUT      //No brownout reset
#FUSES NOLVP           //No low voltage prgming, B3(PIC16) or B5(PIC18) used
for I/O
#use delay(crystal=20000000)
void main()
{
    Trisd=0x00;// cong ra
    TRISB=0XFF; // PORTB INPUT
    Port_B_pullups(True);
    //Example blinking LED program
    while(true)
    {
        If(RB7==0){
            Portd=0;
        }
        Else{
            Portd=0xff;
        }
    }
  
```

}

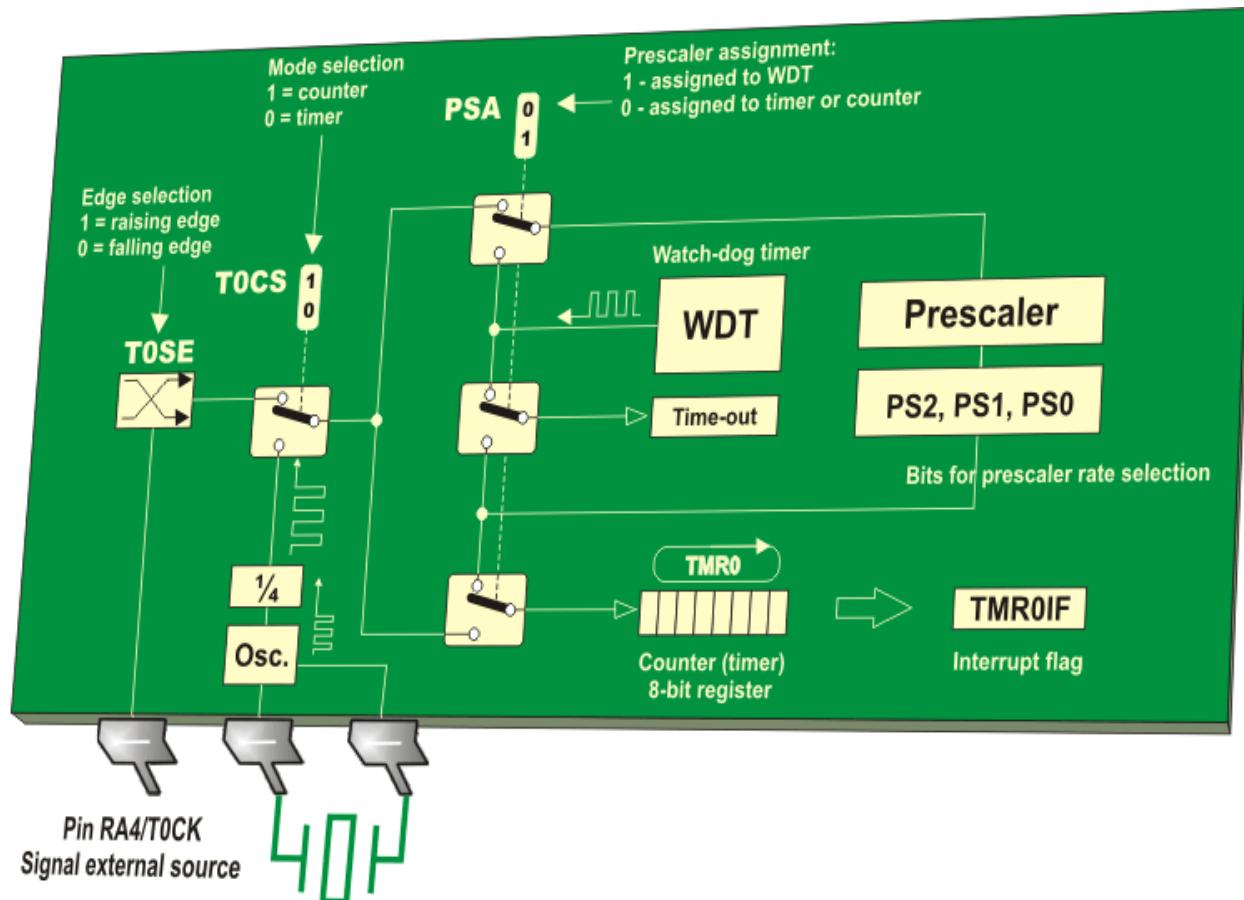
}

#### 4.4. Timer.

##### 4.4.1. Timer 0:

Đây là một trong ba bộ đếm hoặc bộ định thời của vi điều khiển PIC16F877A.

Timer0 là bộ đếm 8 bit được kết nối với bộ chia tần số (prescaler) 8 bit. Cấu trúc của Timer0 cho phép ta lựa chọn xung clock tác động và cạnh tích cực của xung clock. Ngắt Timer0 sẽ xuất hiện khi Timer0 bị tràn. Bit TMR0IE (INTCON<5>) là bit điều khiển của Timer0. TMR0IE=1 cho phép ngắt Timer0 tác động, TMR0IF= 0 không cho phép ngắt Timer0 tác động. Sơ đồ khối của Timer0 như sau:



Hình 4.5: Sơ đồ khối của Timer0.

Muốn Timer0 hoạt động ở chế độ Timer ta clear bit TOSC (OPTION\_REG<5>), khi đó giá trị thanh ghi TMR0 sẽ tăng theo từng chu kì xung đồng hồ (tần số vào Timer0 bằng  $\frac{1}{4}$  tần số oscillator). Khi giá trị thanh ghi TMR0 từ FFh trở về 00h, ngắt Timer0 sẽ xuất hiện. Thanh ghi TMR0 cho phép ghi và xóa được giúp ta ấn định thời điểm ngắt Timer0 xuất hiện một cách linh động.

Muốn Timer0 hoạt động ở chế độ counter ta set bit TOSC (OPTION\_REG<5>). Khi đó xung tác động lên bộ đếm được lấy từ chân RA4/TOCK1. Bit TOSE (OPTION\_REG<4>) cho phép lựa chọn cạnh tác động vào bộ đếm. Cạnh tác động sẽ là cạnh lên nếu TOSE=0 và cạnh tác động sẽ là cạnh xuống nếu TOSE=1.

Khi thanh ghi TMR0 bị tràn, bit TMR0IF (INTCON<2>) sẽ được set. Đây chính

là cờ ngắt của Timer0. Cờ ngắt này phải được xóa bằng chương trình trước khi bộ đếm bắt đầu thực hiện lại quá trình đếm. Ngắt Timer0 không thể đánh thức? vì điều khiển từ chế độ sleep.

Bộ chia tần số (prescaler) được chia sẻ giữa Timer0 và WDT (Watchdog Timer). Điều đó có nghĩa là nếu prescaler được sử dụng cho Timer0 thì WDT sẽ không có được hỗ trợ của prescaler và ngược lại. Prescaler được điều khiển bởi thanh ghi OPTION\_REG. Bit PSA (OPTION\_REG<3>) xác định đối tượng tác động của prescaler. Các bit PS2:PS0 (OPTION\_REG<2:0>) xác định tỉ số chia tần số của prescaler. Xem lại thanh ghi OPTION\_REG để xác định lại một cách chi tiết về các bit điều khiển trên. Các lệnh tác động lên giá trị thanh ghi TMR0 sẽ xóa chế độ hoạt động của prescaler. Khi đối tượng tác động là Timer0, tác động lên giá trị thanh ghi TMR0 sẽ xóa prescaler nhưng không làm thay đổi đối tượng tác động của prescaler. Khi đối tượng tác động là WDT, lệnh CLRWDT sẽ xóa prescaler, đồng thời prescaler sẽ ngưng tác vụ hỗ trợ cho WDT.

Các thanh ghi điều khiển liên quan đến Timer0 bao gồm:

- ✓ TMR0 (địa chỉ 01h, 101h) : chứa giá trị đếm của Timer0.
- ✓ INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): cho phép ngắt hoạt động (GIE và PEIE).
- ✓ OPTION\_REG (địa chỉ 81h, 181h): điều khiển prescaler.

#### **Thanh ghi INTCON: địa chỉ 0Bh, 8Bh, 10Bh, 18Bh.**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF

bit 7

bit 0

Bit 7 GIE Global Interrupt Enable bit

GIE = 1 cho phép tất cả các ngắt.  $\Leftrightarrow$  enable\_interrupts(GLOBAL);

GIE = 0 không cho phép tất cả các ngắt.  $\Leftrightarrow$  disable\_interrupts(GLOBAL);

#### **Thanh ghi OPTION\_REG.**

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

bit 7

bit 0

Bit 5 TOCS Timer0 Clock Source select bit

= 1 clock lấy từ chân RA4/TOCK1.

= 0 dùng xung clock bên trong

Bit 4 TOSE Timer0 Source Edge Select bit

= 1 tác động cạnh lên.

= 0 tác động cạnh xuống

Bit 3 PSA Prescaler Assignment Select bit

= 1 bộ chia tần số (prescaler) được dùng cho WDT

= 0 bộ chia tần số được dùng cho Timer0

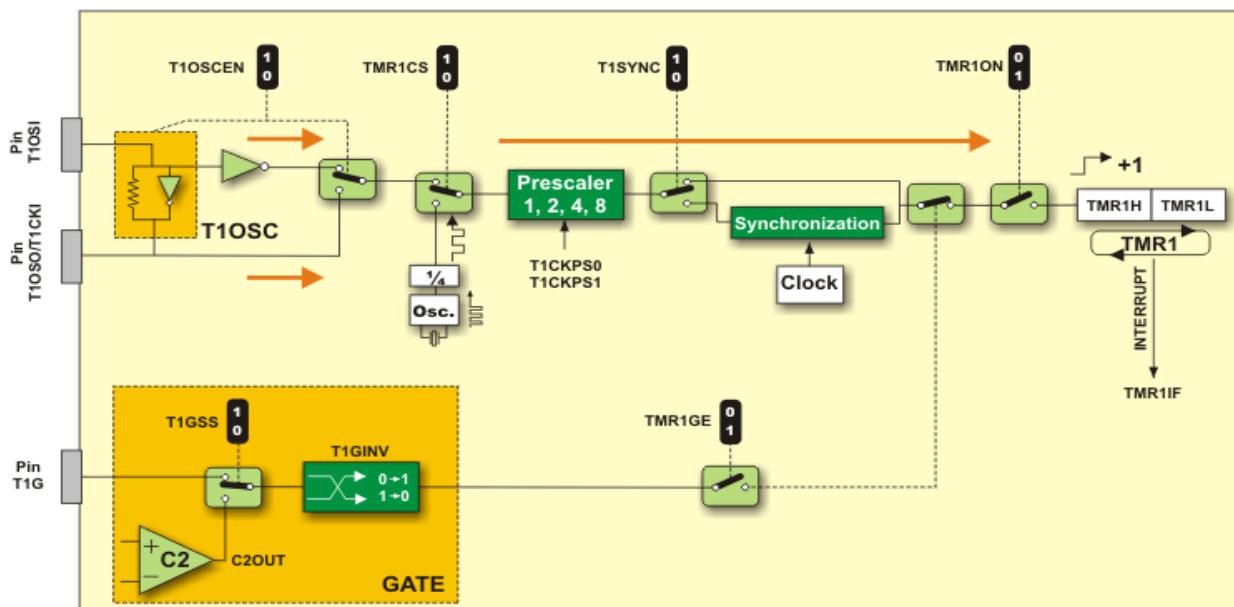
Bit 2:0 PS2:PS0 Prescaler Rate Select bit

Các bit này cho phép thiết lập tỉ số chia tần số của Prescaler

#### 4.4.2. Timer 1:

Timer1 là bộ định thời 16 bit, giá trị của Timer1 sẽ được lưu trong hai thanh ghi (TMR1H:TMR1L). Cờ ngắt của Timer1 là bit TMR1IF (PIR1<0>). Bit điều khiển của Timer1 sẽ là TMR1IE (PIE<0>). Tương tự như Timer0, Timer1 cũng có hai chế độ hoạt động: chế độ định thời (timer) với xung kích là xung clock của oscillator (tần số của timer bằng  $\frac{1}{4}$  tần số của oscillator) và chế độ đếm (counter) với xung kích là xung phản ánh các sự kiện cần đếm lấy từ bên ngoài thông qua chân RC0/T1OSO/T1CKI (cạnh tác động là cạnh lên). Việc lựa chọn xung tác động (tương ứng với việc lựa chọn chế độ hoạt động là timer hay counter) được điều khiển bởi bit TMR1CS (T1CON<1>).

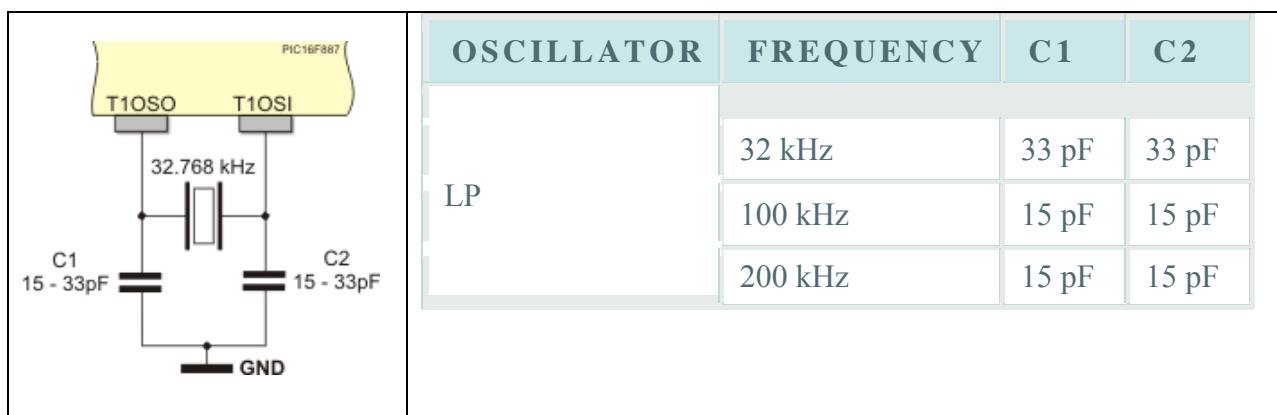
Sau đây là sơ đồ khái của Timer1:



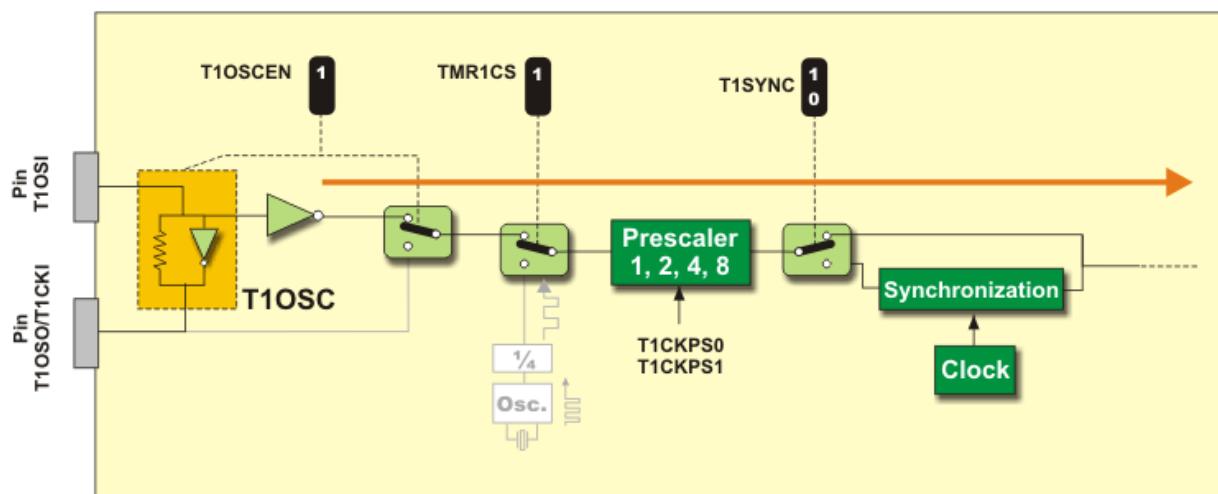
Hình 4.6. Sơ đồ khái của Timer1.

#### Đao động của Timer1

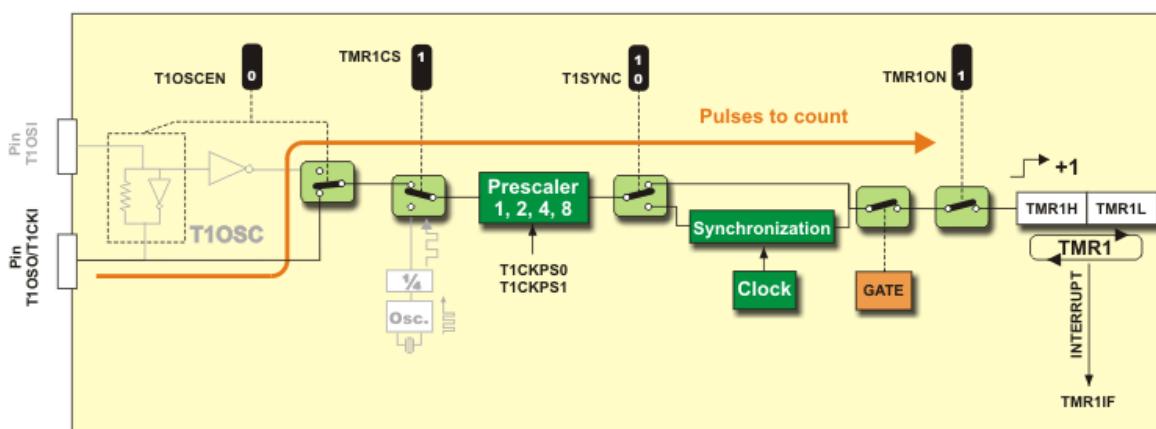
Mạch dao động thạch anh được xây dựng giữa 2 chân T1OSO và T1OSI. Khi dao động được cung cấp ở chế độ công suất thấp thì tần số cực đại của nó sẽ là 200kHz và trong nó ở chế độ SLEEP nó cung cấp ở tần số 32kHz



Hình 4.7. Dao động của Timer1.

**Bộ dao động Timer 1(Timer TMR1 Oscillator)**

Hình 4.8. Chế độ Timer1.

**TMR1 in counter mode**

Hình 4.9. Chế độ Counter của Timer1

Ngoài ra Timer1 còn có chức năng reset input bên trong được điều khiển bởi một trong hai khôi CCP (Capture/Compare/PWM). Khi bit T1OSCEN (T1CON<3>) được set, Timer1 sẽ lấy xung clock từ hai chân RC1/T1OSI/CCP2 và RC0/T1OSO/T1CKI làm xung đếm. Timer1 sẽ bắt đầu đếm sau cạnh xuống đầu tiên của xung ngõ vào. Khi đó PORTC sẽ bỏ qua sự tác động của hai bit TRISC<1:0> và PORTC<2:1> được gán giá trị 0. Khi clear bit T1OSCEN Timer1 sẽ lấy xung đếm từ oscillator hoặc từ chân RC0/T1OSO/T1CKI. Timer1 có hai chế độ đếm là đồng bộ (Synchronous) và bất đồng bộ (Asynchronous). Chế độ đếm được quyết định bởi bit điều khiển (T1CON:<2>).

Khi **T1SYNC** = 1 xung đếm lấy từ bên ngoài sẽ không được đồng bộ hóa với xung clock bên trong, Timer1 sẽ tiếp tục quá trình đếm khi vi điều khiển đang ở chế độ sleep và ngắt do Timer1 tạo ra khi bị tràn có khả năng “đánh thíc” vi điều khiển. Ở chế độ đếm bất đồng bộ, Timer1 không thể được sử dụng để làm nguồn xung clock cho khôi CCP(Capture/Compare/Pulse width modulation). Khi = 0 xung đếm vào Timer1 sẽ được đồng bộ hóa với xung clock bên trong. Ở chế độ này Timer1 sẽ không hoạt động khi vi điều khiển đang ở chế độ sleep.

Các thanh ghi liên quan đến Timer1 bao gồm:

- ✓ INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): cho phép ngắt hoạt động (GIE và PEIE) xem lại phân Timer 0.

- ✓ PIR1 (địa chỉ 0Ch): chứa cờ ngắt Timer1 (TMR1IF).
- ✓ PIE1 (địa chỉ 8Ch): cho phép ngắt Timer1 (TMR1IE).
- ✓ TMR1L (địa chỉ 0Eh): chứa giá trị 8 bit thấp của bộ đếm Timer1.
- ✓ TMR1H (địa chỉ 0Eh): chứa giá trị 8 bit cao của bộ đếm Timer1.
- ✓ T1CON (địa chỉ 10h): xác lập các thông số cho Timer1.

### Thanh ghi PIR1: địa chỉ 0Ch

Thanh ghi chứa cờ ngắt của các khôi ngoại vi.

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF <sup>(1)</sup>	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

Bit 0 TMR1IF TMR1 Overflow Interrupt Flag bit

TMR1IF = 1 thanh ghi TMR1 bị tràn (phải xóa bằng chương trình). ⇔ clear\_interrupt(INT\_TIMER1);

TMR1IF = 0 thanh ghi TMR1 chưa bị tràn.

### Thanh ghi PIE1: địa chỉ 8Ch

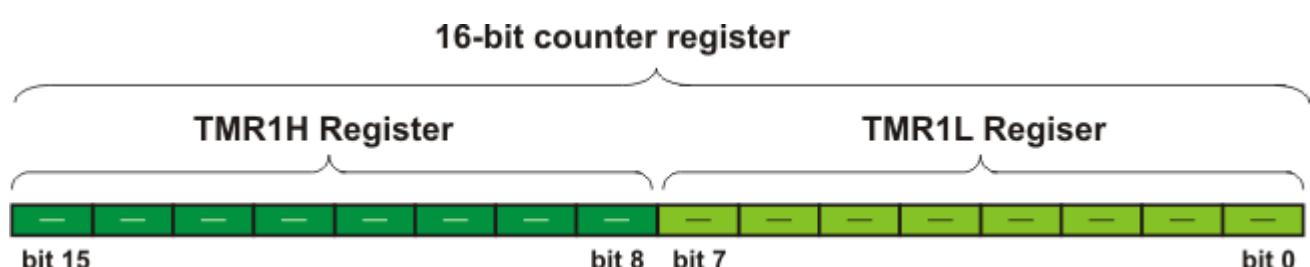
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE <sup>(1)</sup>	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

Bit 0 TMR1IE TMR1 Overflow Interrupt Enable bit

TMR1IE = 1 cho phép ngắt. ⇔ enable\_interrupts(INT\_TIMER1);

TMR1IE = 0 không cho phép ngắt. ⇔ disable\_interrupts(INT\_TIMER1);

### Thanh ghi TMR1:



### Thanh ghi T1CON: địa chỉ 10h.

Thanh ghi điều khiển Timer1.

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
bit 7							bit 0

Bit 7,6 Không quan tâm và mang giá trị mặc định bằng 0.

Bit 5,4 T1CKPS1:T1CKPS0 Timer1 Input Clock Prescaler Select bit

11 tỉ số chia tần số của prescaler là 1:8 ⇔ setup\_timer\_1(T1\_DIV\_BY\_8);

10 tỉ số chia tần số của prescaler là 1:4 ⇔ setup\_timer\_1(T1\_DIV\_BY\_4);

01 tỉ số chia tần số của prescaler là 1:2  $\Leftrightarrow$  setup\_timer\_1(T1\_DIV\_BY\_2);

00 tỉ số chia tần số của prescaler là 1:1  $\Leftrightarrow$  setup\_timer\_1(T1\_DIV\_BY\_1);

Bit 3 T1OSCEN Timer1 Oscillator Enable Control bit

T1OSCEN = 1 cho phép Timer1 hoạt động với xung do oscillator cung cấp.

T1OSCEN = 0 không cho phép Timer1 hoạt động với xung do oscillator cung cấp (tắt bộ chuyển đổi xung bên trong Timer1).

Bit 2 **T1SYNC** Timer1 ternal Clock Input Synchronization Control bit

Khi TMR1CS = 1:

**T1SYNC** = 1 không đồng bộ xung clock ngoại vi đưa vào Timer1.

**T1SYNC** = 0 đồng bộ xung clock ngoại vi đưa vào Timer1.

Khi TMR1CS = 0

Bit không được quan tâm do Timer1 sử dụng xung clock bên trong.

Bit 1 TMR1CS Timer1 Clock Source Select bit

TMR1CS = 1 chọn xung đếm là xung ngoại vi lấy từ pin RC0/T1OSC/T1CKI (cạnh tác động là cạnh lên).

TMR1CS = 0 chọn xung đếm là xung clock bên trong (FOSC/4).

Bit 0 TMR1ON Timer1 On bit

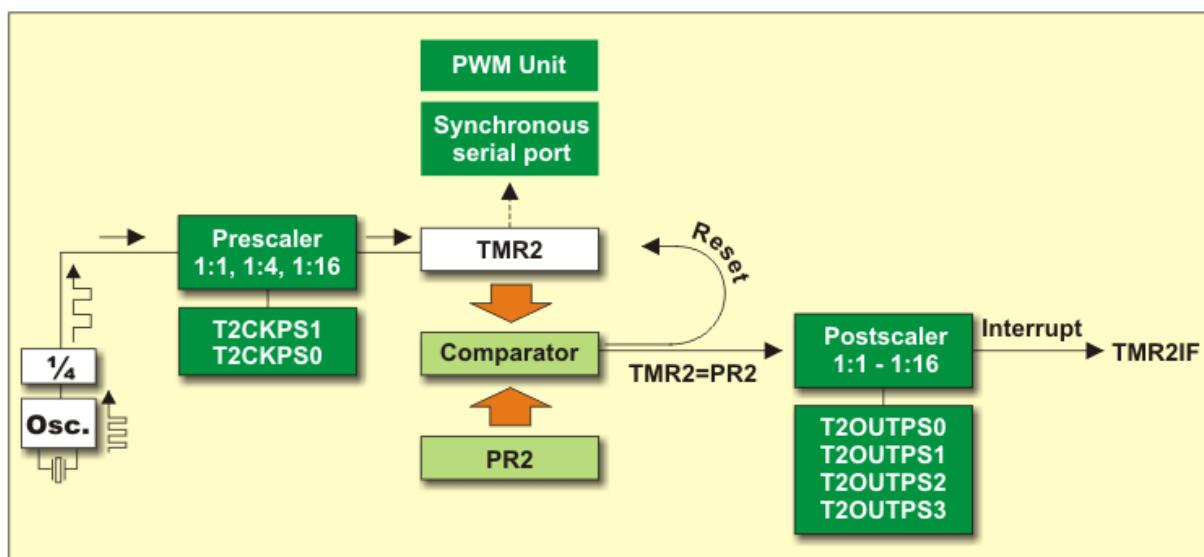
TMR1ON = 1 cho phép Timer1 hoạt động  $\Leftrightarrow$  setup\_timer\_1(T1\_INTERNAL);

TMR1ON = 0 Timer1 ngưng hoạt động.  $\Leftrightarrow$  setup\_timer\_1 ( T1\_DISABLED );

#### 4.4.3. Timer 2:

Timer2 là bộ định thời 8 bit và được hỗ trợ bởi hai bộ chia tần số prescaler và postscaler. Thanh ghi chứa giá trị đếm của Timer2 là TMR2. Bit cho phép ngắt Timer2 tác động là TMR2ON (T2CON<2>). Cờ ngắt của Timer2 là bit TMR2IF (PIR1<1>).

Xung ngoại vào (tần số bằng  $\frac{1}{4}$  tần số oscillator) được đưa qua bộ chia tần số prescaler 4 bit (với các tỉ số chia tần số là 1:1, 1:4 hoặc 1:16 và được điều khiển bởi các bit T2CKPS1:T2CKPS0 (T2CON<1:0>)).



Hình 4.10. Sơ đồ khái niệm Timer2.

Timer2 còn được hỗ trợ bởi thanh ghi PR2. Giá trị đếm trong thanh ghi TMR2 sẽ tăng từ 00h đến giá trị chép trong thanh ghi PR2, sau đó được reset về 00h. Khi reset thanh ghi PR2 được

nhận giá trị mặc định FFh. Ngõ ra của Timer2 được đưa qua bộ chia tần số postscaler với các mức chia từ 1:1 đến 1:16. Postscaler được điều khiển bởi 4 bit T2OUTPS3:T2OUTPS0. Ngõ ra của postscaler đóng vai trò quyết định trong việc điều khiển cờ ngắt. Ngoài ra ngõ ra của Timer2 còn được kết nối với khối SSP, do đó Timer2 còn đóng vai trò tạo ra xung clock đồng bộ cho khối giao tiếp SSP.

Các thanh ghi liên quan đến Timer2 bao gồm:

- ✓ INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): cho phép toàn bộ các ngắt (GIE và PEIE). xem lại phân Timer 0.
- ✓ PIR1 (địa chỉ 0Ch): chứa cờ ngắt Timer2 (TMR2IF).
- ✓ PIE1 (địa chỉ 8Ch): chứa bit điều khiển Timer2 (TMR2IE).
- ✓ TMR2 (địa chỉ 11h): chứa giá trị đếm của Timer2.
- ✓ T2CON (địa chỉ 12h): xác lập các thông số cho Timer2.
- ✓ PR2 (địa chỉ 92h): thanh ghi hỗ trợ cho Timer2.

#### **Thanh ghi PIR1: địa chỉ 0Ch**

Thanh ghi chứa cờ ngắt của các khối ngoại vi.

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF <sup>(1)</sup>	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF

bit 7

bit 0

Bit 1 TMR2IF TMR2 Overflow Interrupt Flag bit

TMR2IF = 1 thanh ghi TMR1 bị tràn (phải xóa bằng chương trình).  $\Leftrightarrow$  clear\_interrupt(INT\_TIMER2);

TMR2IF = 0 thanh ghi TMR1 chưa bị tràn.

#### **Thanh ghi PIE1: địa chỉ 8Ch**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE <sup>(1)</sup>	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE

bit 7

bit 0

Bit 1 TMR2IE TMR2 Overflow Interrupt Enable bit

TMR2IE = 1 cho phép ngắt.  $\Leftrightarrow$  enable\_interrupts(INT\_TIMER2);

TMR2IE = 0 không cho phép ngắt.  $\Leftrightarrow$  disable\_interrupts(INT\_TIMER2);

#### **Thanh ghi T2CON: địa chỉ 12h**

Thanh ghi điều khiển Timer2.

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0

bit 7

bit 0

Bit 7 Không quan tâm và mặc định mang giá trị 0

Bit 6-3 TOUTPS3:TOUTPS0 Timer2 Output Postscaler Select bit

Các bit này điều khiển việc lựa chọn tỉ số chia tần số cho postscaler.

TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	POSTSCALER RATE
0	0	0	0	1:1
0	0	0	1	1:2
0	0	1	0	1:3
0	0	1	1	1:4
0	1	0	0	1:5
0	1	0	1	1:6
0	1	1	0	1:7
0	1	1	1	1:8
1	0	0	0	1:9
1	0	0	1	1:10
1	0	1	0	1:11
1	0	1	1	1:12
1	1	0	0	1:13
1	1	0	1	1:14
1	1	1	0	1:15
1	1	1	1	1:16

Bit 2 TMR2ON Timer2 On bit

TMR2ON = 1 bật Timer2.

TMR2ON = 0 tắt Timer2.

Bit 1,0 T2CKPS1:T2CKPS0 Timer2 Clock Prescaler Select bit

Các bit này điều khiển tỉ số chia tần số của prescaler

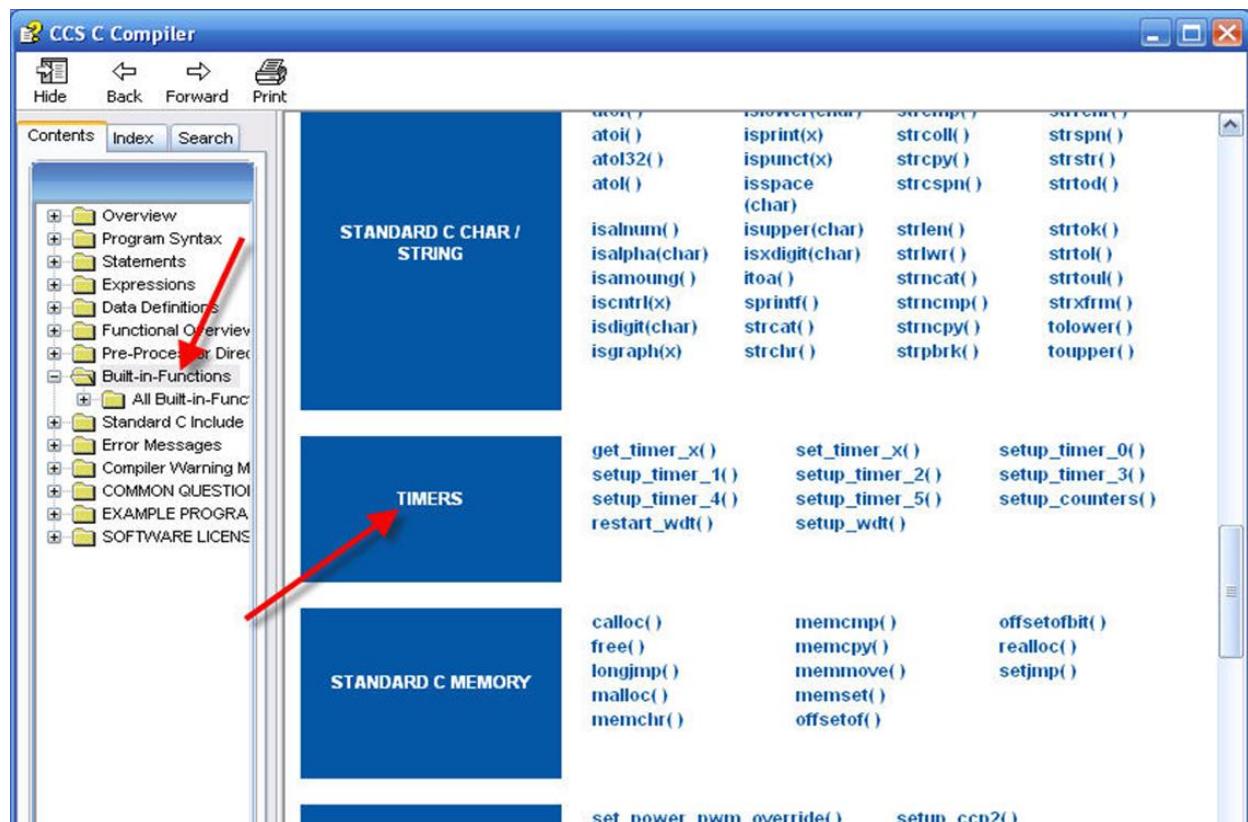
T2CKPS1	T2CKPS0	PRESALER RATE
0	0	1:1
0	1	1:4
1	x	1:16

Ta có một vài nhận xét về Timer0, Timer1 và Timer2 như sau:

Timer0 và Timer2 là bộ đếm 8 bit (giá trị đếm tối đa là FFh), trong khi Timer1 là bộ đếm 16 bit (giá trị đếm tối đa là FFFFh). Timer0, Timer1 và Timer2 đều có hai chế độ hoạt động là timer và counter. Xung clock có tần số bằng  $\frac{1}{4}$  tần số của oscillator. Xung tác động lên Timer0 được hỗ trợ bởi prescaler và có thể được thiết lập ở nhiều chế độ khác nhau (tần số tác động, cạnh tác động) trong khi các thông số của xung tác động lên Timer1 là cố định. Timer2 được hỗ trợ bởi hai bộ chia tần số prescaler và postcaler độc lập, tuy nhiên cạnh tác động vẫn được cố định là cạnh lên. Timer1 có quan hệ với khối CCP, trong khi Timer2 được kết nối với khối SSP. Một vài so sánh sẽ giúp ta dễ dàng lựa chọn được Timer thích hợp cho ứng dụng.

#### 4.4.4. Sử dụng hàm trong CCS:

Để xem các hàm CCS hỗ trợ dung Timer chúng ta vào CCS C Compiler Help => Built-in-Funtions => TIMERS.



Hình 4.11. Các hàm CCS của Timer

#### Hàm hỗ trợ Timer0:

- **setup\_timer\_0 (mode):** thiết đặt cho Timer0.
  - ✓ Mode có thể là:
    - Chọn nguồn xung cấp Timer0: RTCC\_INTERNAL, RTCC\_EXT\_L\_TO\_H, RTCC\_EXT\_H\_TO\_L (Xung cấp dung thạch anh, xung cấp ngoài vào chân RA4(T0) với cạnh từ thấp lên cao, xung cấp ngoài RA4 với cạnh cao xuống thấp).
    - Chọn bộ chia: RTCC\_DIV\_1, RTCC\_DIV\_2, RTCC\_DIV\_4, RTCC\_DIV\_8, RTCC\_DIV\_16, RTCC\_DIV\_32, RTCC\_DIV\_64, RTCC\_DIV\_128, RTCC\_DIV\_256 (Chọn bộ chia 1, chia 2, chia 4, chia 8, chia 16, chia 32, chia 64, chia 128, chia 256).
    - Khi muốn dung nhiều Mode ngăn các nhóm bởi dấu |.
    - **Ví dụ:** Ta muốn dung Timer0 với xung dung thạch anh và bộ chia 8 ta khai báo:  
setup\_timer\_0(RTCC\_INTERNAL| RTCC\_DIV\_8);
  - ✓ **set\_timer0(value) :** khởi tạo giá trị cho timer0. Đây là câu lệnh gán giá trị cho thanh ghi TMR0 8 bit không dấu 0-255.
  - ✓ **value=get\_timer0() :** Trả về giá trị của Timer0. Đọc giá trị trong thanh ghi TMR0.
  - ✓ **Ngắt Timer0: INT\_TIMER0( INT\_RTCC).** Khi dung hàm này ngắt Timer0 sẽ xảy ra ở #INT\_RTC hoặc #INT\_TIMER0. Do đó bạn phải bạn sẽ viết code trong địa chỉ đó.

**Ví dụ:** #INT\_RTCC

```
void ngat_timer0(void)
{
}
```

//Write code here!

}

- ✓ Ví dụ: Khi dùng thạch anh 20MHz ta khởi tạo:

```
setup_timer0(RTCC_INTERNAL|RTCC_DIV_2 )
set_timer0(0);
```

Hãy tính thời gian Timer tràn.

Ta có:  $f=20/4=5\text{MHz}$

$$T=1/f=1/(5*10^6)=2*10^{-7}(\text{S})=0.02(\text{us})$$

Timer0 sẽ tăng lên sau:  $2*0.2(\text{us})=0.4(\text{us})$

Timer0 sẽ tràn sau:  $0.4*((255+1)-TMR0)=0.4*(256-0)=102.4(\text{us})$

Chú ý: TMR0 là giá trị ta dùng lệnh set\_timer0(0);

### **Hàm hỗ trợ Timer1:**

- *setup\_timer\_1(mode)*: thiết đặt cho timer1.

- ✓ Mode có thể là

- Chọn nguồn xung cấp Timer1: T1\_DISABLED, T1\_INTERNAL, T1\_EXTERNAL, T1\_EXTERNAL\_SYNC
- T1\_CLK\_OUT
- Chọn bộ chia: T1\_DIV\_BY\_1, T1\_DIV\_BY\_2, T1\_DIV\_BY\_4, T1\_DIV\_BY\_8
- Tương tự như Timer0, khi muốn dùng nhiều Mode ngăn các nhóm bởi dấu |.

- *set\_timer1(value)* : khởi tạo giá trị cho timer1. Value kiểu 16 bit, không dấu 0-65535

- *value=get\_timer1()* : Trả về giá trị của Timer1, Value kiểu 16 bit.

- Ngắt: INT\_TIMER1. Khi dung hàm này ngắt Timer0 sẽ xảy ra ở #INT\_TIMER1.

- Ví dụ: Dùng thạch anh 20MHz khi khởi tạo:

```
setup_timer1(T1_INTERNAL|T1_DIV_BY_8);
set_timer1(0);
```

- Thì Timer 1 sẽ tăng sau mỗi 1.6us.
- Timer1 sẽ tràn sau 104.858ms.

### **Hàm hỗ trợ Timer2:**

- *setup\_timer\_2(mode, period, postscale)*. Trong đó mode có thể là:

- ✓ T2\_DISABLED, T2\_DIV\_BY\_1, T2\_DIV\_BY\_4, T2\_DIV\_BY\_16

- ✓ period : số nguyên từ 0-255 xác định khi giá trị bị reset

- ✓ postscale: số từ 1-16 là xác định bao nhiêu lần timer bị tràn trước 1 ngắt

- *set\_timer2(value)* : khởi tạo giá trị cho timer2, value kiểu số nguyên 8 bit.

- *value=get\_timer2()* : Trả về giá trị của Timer2 từ 0-255.

- Ngắt : INT\_TIMER2

- Ví dụ: Dùng thạch anh 20MHz khi khởi tạo:

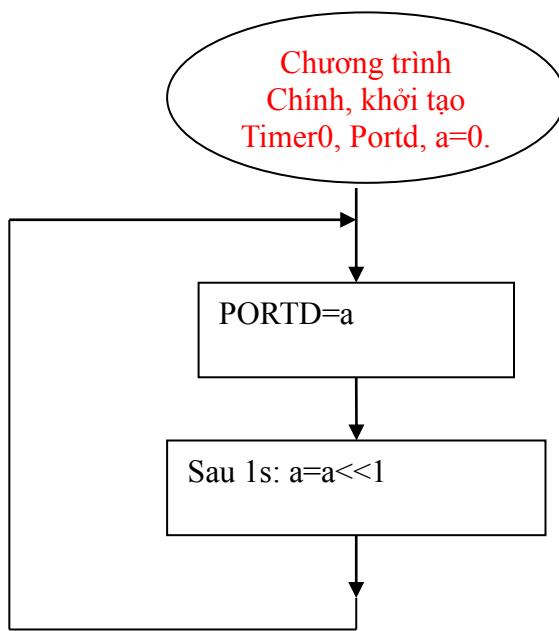
```
setup_timer2(T2_DIV_BY_4,192,2);
set_timer2(0);
```

- Timer2 sẽ tăng sau mỗi 0.8us ( $0.2\text{us}*4$ ).
- Timer2 sẽ tràn sau 154.4us ( $0.8*(192+1)$ )
- Ngắt sau 308.8us ( $154.4\text{us}*2$ )

#### **4.4.5. Bài tập Timer:**

Viết chương trình dịch led trên PORTD, thời gian dịch là 1s. Chương trình sử dụng Timer0 .

- Lưu đồ giải thuật:



- Chương trình:

```
*****
*
* PIC Training Course
* Nguyen Tat Thanh University
*
*****/
```

```
*****
*
* Module      : main.c
* Description  : Blink Led portd frequency 0,5HZ
* Tool        : ccs v5.00xx
* Chip        : 16F877A
* History     : 7/2011
* Version     : v1.0
* Author      : Nguyen Huu Luan
* Mail        : nvn.nhl@gmail.com
* Website     : ntt.edu.vn
* Notes       :
*****/
```

```
///////////
// Su dung portd output
// Sử dụng ngắt timer0 với chu kỳ 1s
// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh
// ///////////////////////////////////////////////////////////////////
```

```
/*
* Keywords Proteus
```

- \* Pic 16f877a.
- \* Led: Led-green, Led-red.....
- \* Dien tro: resistors.
- \*/

```
*****  

#include <16f877a.h>
#include <def_877a.h>
#device *=16 ADC=8
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
int8 a;
int16 count;
#int_timer0 // Khai bao chuong trinh ngat timer0
void ngat_timer0(){
    tmr0if=0; // xoa co ngat timer0
    set_timer0(130);
    count++;
    if(count==2500)
    {
        count=0;
        a=a<<1;
        if(a==256) a=1;
    }
}
void main(void)
{
    trisd=0;// dau ra
    ENABLE_INTERRUPTS(Global);//cho phep ngat Global
    ENABLE_INTERRUPTS(INT_TIMER0);//cho phep ngat timer0
    set_timer0(130);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_16);//125*16*0.2us=400us
    count =0;
    a=1;
    while(true){
        portd=a;
    }
}
```

#### **4.5. ADC**

ADC (Analog to Digital Converter) là bộ chuyển đổi tín hiệu giữa hai dạng tương tự và số. PIC16F877A có 8 ngõ vào analog (RA4:RA0 và RE2:RE0). Hiệu điện thế chuẩn VREF có thể được lựa chọn là VDD, VSS hay hiệu điện thế chuẩn được xác lập trên hai chân RA2 và RA3. Kết quả chuyển đổi từ tín hiệu tương tự sang tín hiệu số là 10 bit số tương ứng và được lưu trong hai thanh ghi ADRESH:ADRESL. Khi không sử dụng bộ chuyển đổi ADC, các thanh ghi này có

thể được sử dụng như các thanh ghi thông thường khác. Khi quá trình chuyển đổi hoàn tất, kết quả sẽ được lưu vào hai thanh ghi ADRESH:ADRESL, GO/DONE bit (ADCON0<2>) được xóa về 0 và cờ ngắt ADIF được set.

Qui trình chuyển đổi từ tương tự sang số bao gồm các bước sau:

1. Thiết lập các thông số cho bộ chuyển đổi ADC:

- ✓ Chọn ngõ vào analog, chọn điện áp mẫu (dựa trên các thông số của thanh ghi ADCON1)
- ✓ Chọn kênh chuyển đổi AD (thanh ghi ADCON0).
- ✓ Chọn xung clock cho kênh chuyển đổi AD (thanh ghi ADCON0).
- ✓ Cho phép bộ chuyển đổi AD hoạt động (thanh ghi ADCON0).

2. Thiết lập các cờ ngắt cho bộ AD

- ✓ Clear bit ADIF(ADC Interrupt Flag bit =0 chưa hoàn thành chuyển đổi)
- ✓ Set bit ADIE( ADC Interrupt Enable bit=1 cho phép ngắt ADC )
- ✓ Set bit PEIE( Peripheral Interrupt Enable bit = 1 cho phép tắt cả các ngắt ngoại vi)
- ✓ Set bit GIE(Global Interrupt Enable bit= 1 cho phép tắt cả các ngắt)

3. Đợi cho tới khi quá trình lấy mẫu hoàn tất (20us)t.

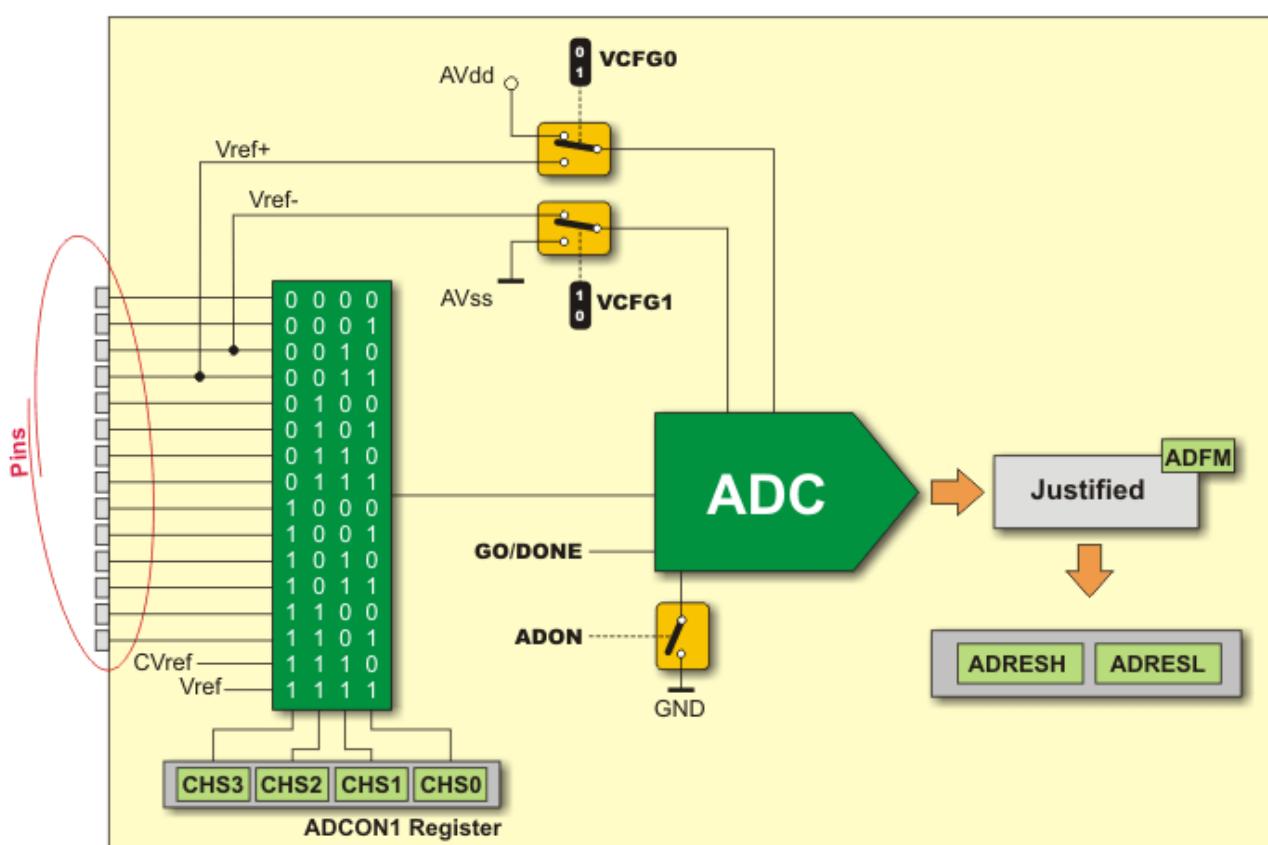
4. Bắt đầu quá trình chuyển đổi (set bit GO/DONE).

5. Đợi cho tới khi quá trình chuyển đổi hoàn tất bằng cách:

Kiểm tra bit GO/DONE. Nếu GO/DONE =0, quá trình chuyển đổi đã hoàn tất hoặc Kiểm tra cờ ngắt A/D( nếu dùng ngắt)

6. Đọc kết quả chuyển đổi và xóa cờ ngắt ADIF, set bit GO/DONE (nếu cần tiếp tục chuyển đổi).

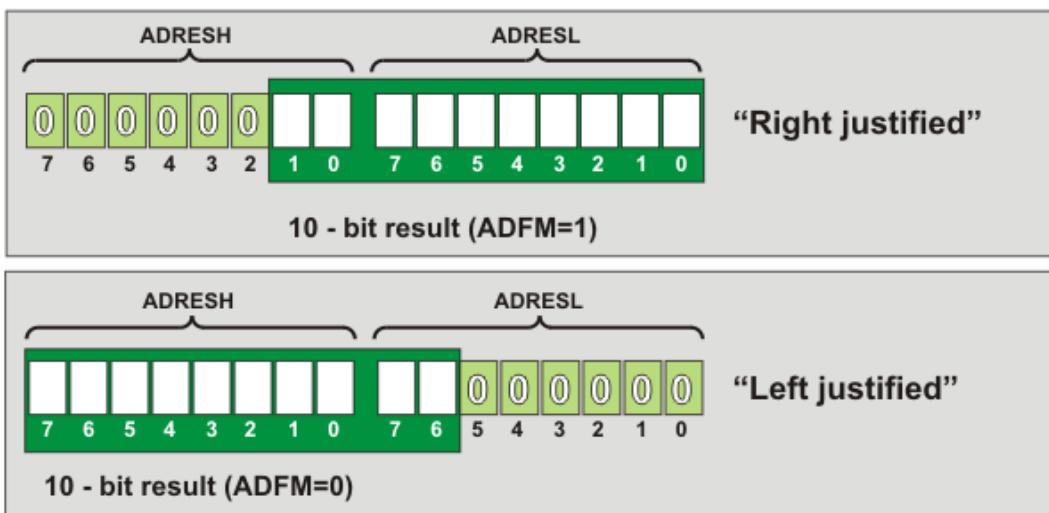
7. Tiếp tục thực hiện các bước 1 và 2 cho quá trình chuyển đổi tiếp theo.



Hình 4.12. Sơ đồ khái niệm bộ chuyển đổi ADC

Cần chú ý là có hai cách lưu kết quả chuyển đổi AD, việc lựa chọn cách lưu được điều

khiến bởi bit ADFM và được minh họa cụ thể trong hình sau:



Hình 4.13. Các cách lưu kết quả chuyển đổi AD

Các thanh ghi liên quan đến bộ chuyển đổi ADC bao gồm:

- ✓ INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): cho phép các ngắt (các bit GIE, PEIE).
- ✓ PIR1 (địa chỉ 0Ch): chứa cờ ngắt AD (bit ADIF).
- ✓ PIE1 (địa chỉ 8Ch): chứa bit điều khiển AD (ADIE).
- ✓ ADRESH (địa chỉ 1Eh) và ADRESL (địa chỉ 9Eh): các thanh ghi chứa kết quả chuyển đổi AD.
- ✓ ADCON0 (địa chỉ 1Fh) và ADCON1 (địa chỉ 9Fh): xác lập các thông số cho bộ chuyển đổi AD.
- ✓ PORTA (địa chỉ 05h) và TRISA (địa chỉ 85h): liên quan đến các ngõ vào analog ở PORTA, xem lại mục xuất nhập.
- ✓ PORTE (địa chỉ 09h) và TRISE (địa chỉ 89h): liên quan đến các ngõ vào analog ở PORTE.

**Thanh ghi INTCON (0Bh, 8Bh, 10Bh, 18Bh):** thanh ghi cho phép đọc và ghi, chứa các bit điều khiển và các bit cờ hiệu khi timer0 bị tràn, ngắt ngoại vi RB0/INT và ngắt interrupt-on-change tại các chân của PORTB.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
bit 7							bit 0

Bit 7 GIE Global Interrupt Enable bit

GIE = 1 cho phép tắt cả các ngắt.

GIE = 0 không cho phép tắt cả các ngắt.

Bit 6 PEIE Peripheral Interrupt Enable bit

PEIE = 1 cho phép tắt cả các ngắt ngoại vi

PEIE = 0 không cho phép tắt cả các ngắt ngoại vi

**Thanh ghi PIR1 (0Ch):** chứa cờ ngắt của các khối chức năng ngoại vi, các ngắt này được cho phép bởi các bit điều khiển chứa trong thanh ghi PIE1.

R/W-0 PSPIF <sup>(1)</sup>	R/W-0 ADIF	R-0 RCIF	R-0 TXIF	R/W-0 SSPIF	R/W-0 CCP1IF	R/W-0 TMR2IF	R/W-0 TMR1IF
bit 7				bit 0			

Bit 6 ADIF ADC Interrupt Flag bit  
 ADIF = 1 hoàn tất chuyển đổi ADC.  
 ADIF = 0 chưa hoàn tất chuyển đổi ADC.

**Thanh ghi PIE1 (8Ch):** chứa các bit điều khiển chi tiết các ngắt của các khối chức năng ngoại vi.

R/W-0 PSPIE <sup>(1)</sup>	R/W-0 ADIE	R/W-0 RCIE	R/W-0 TXIE	R/W-0 SSPIE	R/W-0 CCP1IE	R/W-0 TMR2IE	R/W-0 TMR1IE
bit 7				bit 0			

Bit 6 ADIE ADC (A/D converter) Interrupt Enable bit  
 ADIE = 1 cho phép ngắt ADC.  
 ADIE = 0 không cho phép ngắt ADC.

**Thanh ghi ADCON0 (1Fh):**

R/W-0 ADCS1	R/W-0 ADCS0	R/W-0 CHS2	R/W-0 CHS1	R/W-0 CHS0	R/W-0 GO/DONE	U-0	R/W-0 ADON
bit 7				bit 0			

Bit 7, 6 kết hợp với bit 6 (ADCS2) của thanh ghi ADCON1 để điều khiển việc chọn xung clock cho khối chuyển đổi ADC.

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	Fosc/4
1	01	Fosc/16
1	10	FOSC/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

Bit 5-3 CHS2:CHS0 - Analog Channel Select bit

Các bit này dùng để chọn kênh chuyển đổi ADC

- 000 = Channel 0 (AN0)
- 001 = Channel 1 (AN1)
- 010 = Channel 2 (AN2)
- 011 = Channel 3 (AN3)
- 100 = Channel 4 (AN4)
- 101 = Channel 5 (AN5)
- 110 = Channel 6 (AN6)

111 = Channel 7 (AN7)

Bit 2 GO/DONE: A/D Conversion Status bit

=1 A/D đang hoạt động (set bit này sẽ làm khởi động ADC và tự xóa khi quá trình chuyển đổi kết thúc)

=0 A/D không hoạt động

Bit 1: Không quan tâm và mặc định mang giá trị 0.

Bit 0: ADON:

=1: A/D cho phép quá trình chuyển đổi.

=0: A/D ngắt quá trình chuyển đổi.

#### Thanh ghi ADCON1 (9Fh):

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0

bit 7

bit 0

Bit 7 ADFM A/D Result Format Select bit

ADFM = 1 Kết quả được lưu về phía bên phải 2 thanh ghi ADRESH:ADRESL (6 bit cao mang giá trị 0).

ADFM = 0 Kết quả được lưu về phía bên trái 2 thanh ghi ADRESH:ADRESL (6 bit thấp mang giá trị 0).

Bit 6 ADCS2 A/D Conversion Clock Select bit

ADCS2 kết hợp với 2 bit ADCS1:ADCS0 trong thanh ghi ADCON0 để điều khiển việc chọn xung clock cho khối chuyển đổi ADC xem lại thanh ghi ADCON0.

Bit 5,4 Không cần quan tâm và mặc định mang giá trị 0.

Bit 3-0 PCFG3:PCFG0 A/D Port Configuration Control bit Các bit này điều khiển việc chọn cấu hình hoạt động các cổng của bộ chuyển đổi ADC.

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	Vss	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	Vss	7/1
0010	D	D	D	A	A	A	A	A	VDD	Vss	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	Vss	4/1
0100	D	D	D	D	A	D	A	A	VDD	Vss	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	Vss	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	Vss	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	Vss	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	Vss	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

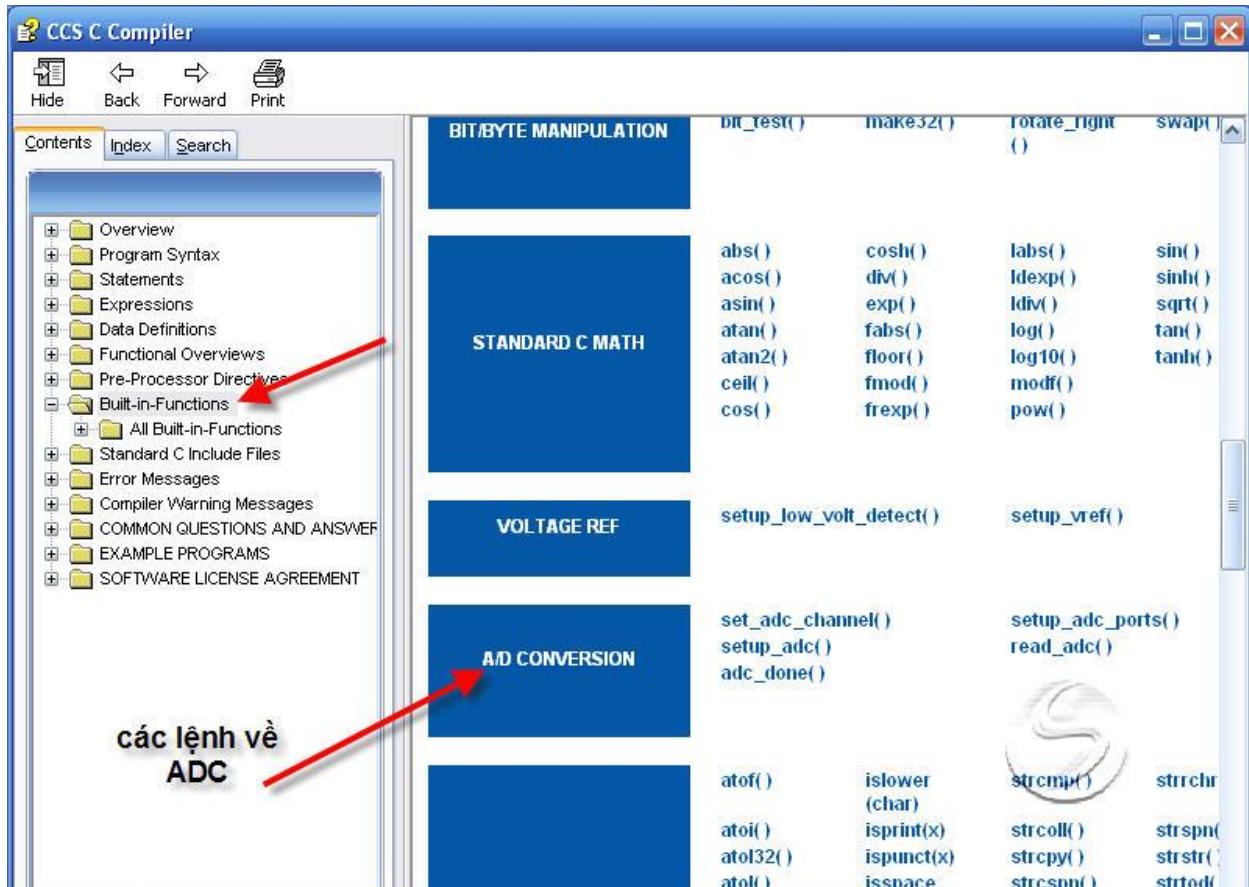
Trong đó A là ngõ vào Analog.

D là ngõ vào Digital.

C/R là số ngõ vào Analog/số điện áp mẫu.

### ❖ Các hàm ADC trong CCS.

Để xem các hàm CCS hỗ trợ dung Timer chúng ta vào CCS C Compiler Help => Built-in-Funtions => A/D CONVERSION.



➤ #device ADC=xx

✓ với xx là số bit .Với PIC16F877A thì xx=08 hoặc xx=10.

➤ Setup\_ADC(mode) Mode có thể là:

- ✓ ADC\_OFF // Tắt ADC
- ✓ ADC\_CLOCK\_DIV\_2 //thời gian lấy mẫu bằng xung Clock/2
- ✓ ADC\_CLOCK\_DIV\_4 //thời gian lấy mẫu bằng xung Clock/4
- ✓ ADC\_CLOCK\_DIV\_8 //thời gian lấy mẫu bằng xung Clock/8
- ✓ ADC\_CLOCK\_DIV\_16 //thời gian lấy mẫu bằng xung Clock/16
- ✓ ADC\_CLOCK\_DIV\_32 //thời gian lấy mẫu bằng xung Clock/32
- ✓ ADC\_CLOCK\_DIV\_64 //thời gian lấy mẫu bằng xung Clock/64
- ✓ ADC\_CLOCK\_INTERNAL //thời gian lấy mẫu 2-6us

➤ Setup\_ADC\_ports(Value)

Dùng để xác định chân lấy tín hiệu analog và điện thế chuẩn sử dụng. Value là:

Value	Tác dụng
NO_ANALOGS	Không dung analog
ALL_ANALOG	A0 A1 A2 A3 A5 E0 E1 E2
AN0_AN1_AN2_AN4_AN5_AN6_AN7_VSS_VREF	A0 A1 A2 A5 E0 E1 E2 VRefh=A3
AN0_AN1_AN2_AN3_AN4	A0 A1 A2 A3 A5
AN0_AN1_AN2_AN4_VSS_VREF	A0 A1 A2 A4 VRefh=A3
AN0_AN1_AN3	A0 A1 A3
AN0_AN1_VSS_VREF	A0 A1 VRefh=A3
AN0_AN1_AN4_AN5_AN6_AN7_VREF_VREF	A0 A1 A5 E0 E1 E2 VRefh=A3 VRefl=A2
AN0_AN1_AN2_AN3_AN4_AN5	A0 A1 A2 A3 A5 E0
AN0_AN1_AN2_AN4_AN5_VSS_VREF	A0 A1 A2 A5 E0 VRefh=A3
AN0_AN1_AN4_AN5_VREF_VREF	A0 A1 A5 E0 VRefh=A3 VRefl=A2
AN0_AN1_AN4_VREF_VREF	A0 A1 A4 VRefh=A3 VRefl=A2
AN0_AN1_VREF_VREF	A0 A1 VRefh=A3 VRefl=A2
AN0	A0
AN0_VREF_VREF	A0 VRefh=A3 VRefl=A2

➤ **Set\_ADC\_channel(channel):**

- ✓ Chọn chân để đọc vào giá trị analog bằng lệnh Read\_adc().
- ✓ Với PIC16F877A channel=0-7.

➤ **Read\_adc([mode]):** Dùng đọc giá trị ADC từ thanh ghi(cặp thanh ghi) chưa kết quả biến đổi ADC.

- ✓ Mode không bắt buộc.
- ✓ Mode có thể là:

- ADC\_START\_AND\_READ (Mặc định).
- ADC\_START\_ONLY
- ADC\_READ\_ONLY

➤ **adc\_done( ):** Nó trả về TRUE nếu đã thực hiện chuyển đổi A/D, FALSE nếu vẫn còn bận.

Ví dụ:

```

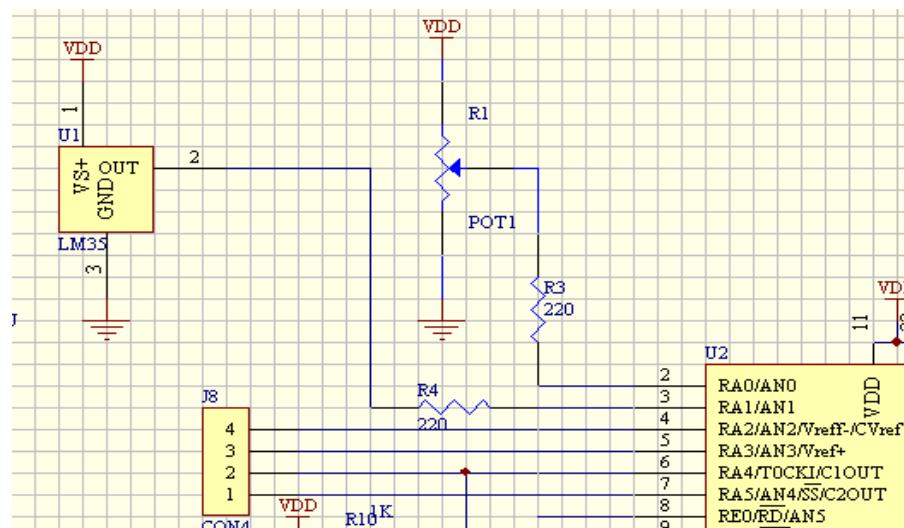
int16 value;
setup_adc_ports(sAN0|sAN1, VSS_VDD);
setup_adc(ADC_CLOCK_DIV_4|ADC_TAD_MUL_8);
set_adc_channel(0);
read_adc(ADC_START_ONLY);
int1 done = adc_done();
while(!done) {
    done = adc_done();
}
value = read_adc(ADC_READ_ONLY);
printf("A/C value = %LX\n\r", value);
}

```

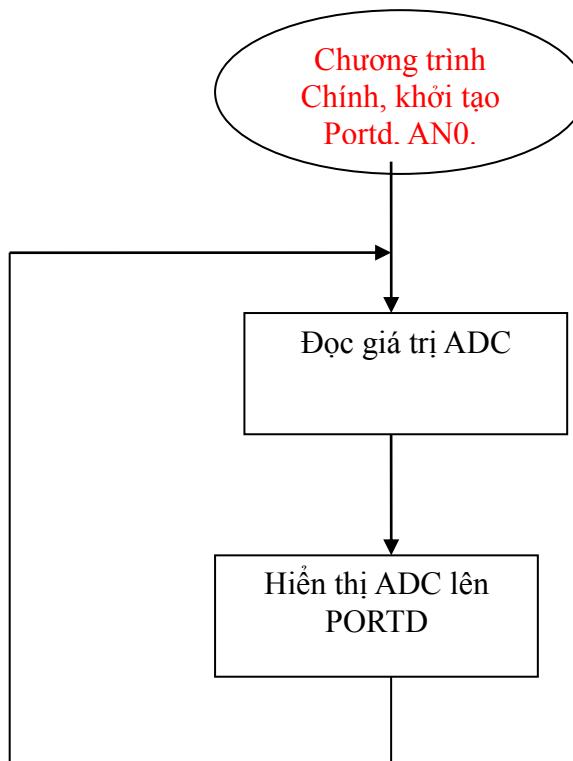
❖ **Bài tập ADC.**

Đọc giá trị ADC 8 bit chân AN0 (điều chỉnh bằng biến Trở 10k) xuất giá trị ra Led ở Portd.

- ✓ Kết nối phần cứng:



- ✓ Lưu đồ giải thuật:



- ✓ Chương trình:

```

 ****
 *
 * PIC Training Course
 * Nguyen Tat Thanh University
 *
 ****

```

```

 ****
 *

```

```
* Module      : main.c
* Description : Read ADC at AN0.
* Tool        : ccs v5.00xx
* Chip        : 16F877A
* History     : 7/2011
* Version     : v1.0
* Author      : Nguyen Huu Luan
* Mail        : nvn.nhl@gmail.com
* Website     : ntt.edu.vn
* Notes       :  
*****
```

```
//////////  
// Su dung portd output //  
// Doc gia tri adc tai chan an0 //  
// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh //  
//  
//////////
```

```
/*  
* Keywords Proteus  
* Pic 16f877a.  
* Led: Led-green, Led-red....  
* Dien tro: resistors.  
*/
```

```
*****
```

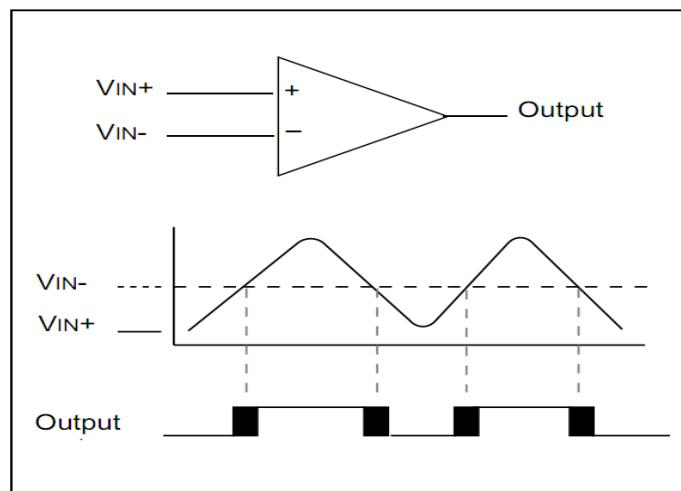
```
#include<16f877a.h>
#include <def_877a.h>
#fuses NOWDT,HS,PUT,NOPROTECT
#device 16f877a*=16 adc=8
#use delay(clock=20000000)
int8 adc;
main(){
    trisd=0;//dau ra
    setup_adcadc_clock_internal;//thoi gian lay mau 2-6us
    setup_adc_ports(an0);//AN0 nhan analog
    set_adc_channel(0);//chon chan de doc
    delay_ms(10);
    while(1)
    {
        adc=read_adc();
        portd=adc;
    }
}
```

#### 4.6. Comparator

Bộ so sánh bao gồm hai bộ so sánh tín hiệu analog và được đặt ở PORTA. Ngõ vào bộ so sánh là các chân RA3:RA0, ngõ ra là hai chân RA4 và RA5. Thanh ghi điều khiển bộ so sánh là CMCON. Các bit CM2:CM0 trong thanh ghi CMCON đóng vai trò chọn lựa các chế độ hoạt động cho bộ Comparator (hình 2.10). Cơ chế hoạt động của bộ Comparator như sau:

Tín hiệu analog ở chân VIN+ sẽ được so sánh với điện áp chuẩn ở chân VIN- và tín hiệu ở ngõ ra bộ so sánh sẽ thay đổi tương ứng như hình vẽ. Khi điện áp ở chân VIN+ lớn hơn điện áp ở chân VIN- ngõ ra sẽ ở mức 1 và ngược lại.

Dựa vào hình vẽ ta thấy đáp ứng tại ngõ ra không phải là tức thời so với thay đổi tại ngõ vào mà cần có một khoảng thời gian nhất định để ngõ ra thay đổi trạng thái (tối đa là 10 us). Cần chú ý đến khoảng thời gian đáp ứng này khi sử dụng bộ so sánh. Cực tính của các bộ so sánh có thể thay đổi dựa vào các giá trị đặt vào các bit C2INV và C1INV (CMCON<4:5>).



Hình 4.14. Nguyên tắc hoạt động của một bộ so sánh đơn giản

Các bit C2OUT và C1OUT (CMCON<7:6>) đóng vai trò ghi nhận sự thay đổi tín hiệu analog so với điện áp đặt trước. Các bit này cần được xử lý thích hợp bằng chương trình để ghi nhận sự thay đổi của tín hiệu ngõ vào. Cờ ngắt của bộ so sánh là bit CMIF (thanh ghi PIR1). Cờ ngắt này phải được reset về 0. Bit điều khiển bộ so sánh là bit CMIE (Tranh ghi PIE).

Các thanh ghi liên quan đến bộ so sánh bao gồm:

- ✓ CMCON (địa chỉ 9Ch) và CVRCON (địa chỉ 9Dh): xác lập các thông số cho bộ so sánh.
- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): chứa các bit cho phép các ngắt (GIE và PEIE).
- ✓ Thanh ghi PIR2 (địa chỉ 0Dh): chứa cờ ngắt của bộ so sánh (CMIF).
- ✓ Thanh ghi PIE2 (địa chỉ 8Dh): chứa bit cho phép bộ so sánh (CNIE).
- ✓ Thanh ghi PORTA (địa chỉ 05h) và TRISA (địa chỉ 85h): các thanh ghi điều khiển PORTA.

#### Thanh ghi CMCON (địa chỉ 9Ch)

Thanh ghi điều khiển và chỉ thị các trạng thái cũng như kết quả của bộ so sánh.

R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-1	R/W-1
C2OUT	C1OUT	C2INV	C1INV	CIS	CM2	CM1	CM0
bit 7							bit 0

Bit 7 C2OUT Comparator 2 (C2) Output bit

Khi C2INV = 0

C2OUT = 1 khi (pin VIN+ của C2) > (pin VIN- của C2).

C2OUT = 0 khi (pin VIN+ của C2) < (pin VIN- của C2).

Khi C2INV = 1

C2OUT = 1 khi (pin VIN+ của C2) < (pin VIN- của C2).

C2OUT = 0 khi (pin VIN+ của C2) > (pin VIN- của C2).

Bit 6 C1OUT Comparator 1 (C1) Output bit

Khi C1INV = 0

C1OUT = 1 khi (pin VIN+ của C1) > (pin VIN- của C1).

C1OUT = 0 khi (pin VIN+ của C1) < (pin VIN- của C1).

Khi C1INV = 1

C1OUT = 1 khi (pin VIN+ của C1) < (pin VIN- của C1).

C1OUT = 0 khi (pin VIN+ của C1) > (pin VIN- của C1).

Bit 5 C2INV Comparator 2 Output Conversion bit

C2INV = 1 ngõ ra C2 được đảo trạng thái.

C2INV = 0 ngõ ra C2 không đảo trạng thái.

Bit 4 C1INV Comparator 1 Output Conversion bit

C1INV = 1 ngõ ra C1 được đảo trạng thái.

C1INV = 0 ngõ ra C1 không đảo trạng thái.

Bit 3 CIS Comparator Input Switch bit

Bit này chỉ có tác dụng khi CM2:CM0 = 110

CIS = 1 khi pin VIN- của C1 nối với RA3/AN3 và pin VIN- của C2 nối với RA2/AN2

CIS = 0 khi pin VIN- của C1 nối với RA0/AN0 và pin VIN- của C2 nối với RA1/AN1

Bit 2-0 CM2:CM0 Comparator Mode bit

Các bit này đóng vai trò trong việc thiết lập các cấu hình hoạt động của bộ Comparator. Các dạng cấu hình của bộ Comparator được trình bày trong bảng sau:

<p><b>Two Common Reference Comparators</b> CM2:CM0 = 100</p>	<p><b>Two Common Reference Comparators with Outputs</b> CM2:CM0 = 101</p>
<p><b>One Independent Comparator with Output</b> CM2:CM0 = 001</p>	<p><b>Four Inputs Multiplexed to Two Comparators</b> CM2:CM0 = 110</p>

A = Analog Input, port reads zeros always. D = Digital Input. CIS (CMCON<3>) is the Comparator Input Switch.

<p><b>Comparators Reset</b> CM2:CM0 = 000</p>	<p><b>Comparators Off (POR Default Value)</b> CM2:CM0 = 111</p>
<p><b>Two Independent Comparators</b> CM2:CM0 = 010</p>	<p><b>Two Independent Comparators with Outputs</b> CM2:CM0 = 011</p>

Hình 5.6.2. Các chế độ hoạt động của bộ comparator. (xem các chế độ hoạt động trang 135 DATASHEET PIC16F87XA)

### Thanh ghi CVRCON (địa chỉ 9Dh)

Thanh ghi điều khiển bộ tạo điện áp so sánh khi bộ Comparator hoạt động với cấu hình ‘110’.

R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
CVREN	CVROE	CVRR	—	CVR3	CVR2	CVR1	CVR0

bit 7

bit 0

Bit 7 CVREN Comparator Voltage Reference Enable bit.

CVREN = 1 bộ tạo điện áp so sánh được cấp điện áp hoạt động.

CVREN = 0 bộ tạo điện áp so sánh không được cấp điện áp hoạt động.

Bit 6 CVROE Comparator VREF Output Enable bit

CVROE = 1 điện áp do bộ tạo điện áp so sánh tạo ra được đưa ra pin RA2.

CVROA = 0 điện áp do bộ tạo điện áp so sánh tạo ra không được đưa ra ngoài.

Bit 5 CVRR Comparator VREF Range Selection bit

CVRR = 1 một mức điện áp có giá trị VDD/24 (điện áp do bộ tạo điện áp so sánh tạo ra có giá trị từ 0 đến 0.75VDD).

CVRR = 0 một mức điện áp có giá trị VDD/32 (điện áp do bộ tạo điện áp so sánh tạo ra có giá trị từ 0.25 đến 0.75VDD).

Bit 4 Không cần quan tâm và mặc định mang giá trị 0.

Bit 3-0 CVR3:CVR0 Các bit chọn điện áp ngõ ra của bộ tạo điện áp so sánh.

Khi CVRR = 1:

Điện áp tại pin RA2 có giá trị CVREF = (CVR<3:0>/24)\*VDD.

Khi CVRR = 0

Điện áp tại pin RA2 có giá trị CVREF = (CVR<3:0>/32)\*VDD + ¼VDD.

**Thanh ghi INTCON (0Bh, 8Bh, 10Bh, 18Bh):** thanh ghi cho phép đọc và ghi, chứa các bit điều khiển và các bit cờ hiệu khi timer0 bị tràn, ngắt ngoại vi RB0/INT và ngắt interrupt-change tại các chân của PORTB.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF

bit 7

bit 0

Bit 7 GIE Global Interrupt Enable bit

GIE = 1 cho phép tắt cả các ngắt.

GIE = 0 không cho phép tắt cả các ngắt.

Bit 6 PEIE Peripheral Interrupt Enable bit

PEIE = 1 cho phép tắt cả các ngắt ngoại vi

PEIE = 0 không cho phép tắt cả các ngắt ngoại vi

**Thanh ghi PIR2 (0Dh):** chứa các cờ ngắt của các khối chức năng ngoại vi, các ngắt này được cho phép bởi các bit điều khiển chứa trong thanh ghi PIE2.

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF

bit 7

bit 0

Bit 6 CMIF Comparator Interrupt Flag bit

CMIF = 1 tín hiệu ngõ vào bộ so sánh thay đổi.

**Thanh ghi PIE2 (8Dh):** chứa các bit điều khiển các ngắt của các khôi chức năng CCP2, SSP bus, ngắt của bộ so sánh và ngắt ghi vào bộ nhớ EEPROM

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE

bit 7

bit 0

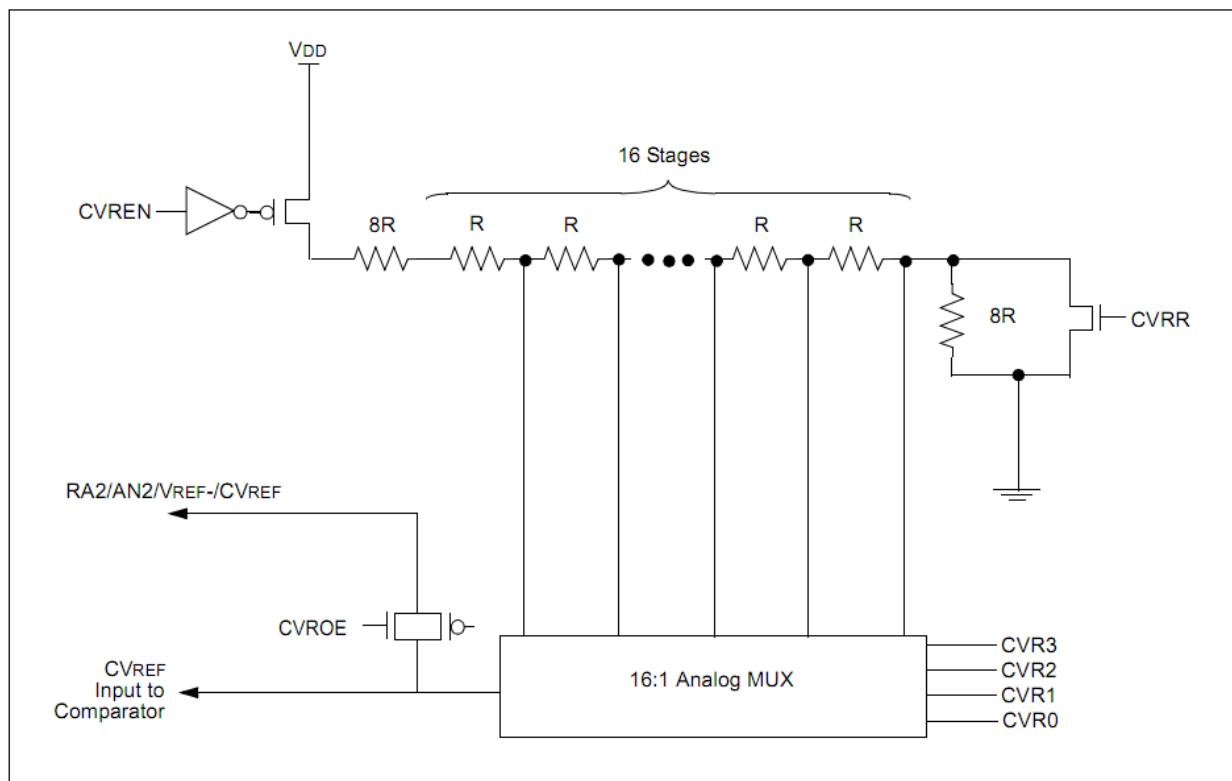
Bit 6 CMIE Comparator Interrupt Enable bit

CMIE = 1 Cho phép ngắt của bộ so sánh.

CMIE = 0 Không cho phép ngắt.

### Bộ tạo điện áp so sánh:

Bộ so sánh này chỉ hoạt động khi bộ Comparator được định dạng hoạt động ở chế độ '110'. Khi đó các pin RA0/AN0 và RA1/AN1 (khi CIS = 0) hoặc pin RA3/AN3 và RA2/AN2 (khi CIS = 1) sẽ là ngõ vào analog của điện áp cần so sánh đưa vào ngõ VIN- của 2 bộ so sánh C1 và C2 (xem chi tiết ở hình 2.10). Trong khi đó điện áp đưa vào ngõ VIN+ sẽ được lấy từ một bộ tạo điện áp so sánh. Sơ đồ khối của bộ tạo điện áp so sánh được trình bày trong hình vẽ sau:



Hình 4.15. Sơ đồ khối bộ tạo điện áp so sánh.

Bộ tạo điện áp so sánh này bao gồm một thang điện trở 16 mức đóng vai trò là cầu phân áp chia nhỏ điện áp VDD thành nhiều mức khác nhau (16 mức). Mỗi mức có giá trị điện áp khác nhau tùy thuộc vào bit điều khiển CVRR (CVRC<sub>CON<5></sub>). Nếu CVRR ở mức logic 1, điện trở 8R sẽ không có tác dụng như một thành phần của cầu phân áp (BJT dẫn mạnh và dòng điện không đi qua điện trở 8R), khi đó 1 mức điện áp có giá trị VDD/24. Ngược lại khi CVRR ở mức logic 0, dòng điện sẽ qua điện trở 8R và 1 mức điện áp có giá trị VDD/32. Các mức điện áp này được đưa qua bộ MUX cho phép ta chọn được điện áp đưa ra pin RA2/AN2/VREF-/CVREF để đưa vào ngõ VIN+ của bộ so sánh bằng cách đưa các giá trị thích hợp vào các bit CVR3:CVR0.

Bộ tạo điện áp so sánh này có thể xem như một bộ chuyển đổi D/A đơn giản. Giá trị điện áp cần so sánh ở ngõ vào Analog sẽ được so sánh với các mức điện áp do bộ tạo điện áp tạo ra

cho tới khi hai điện áp này đạt được giá trị xấp xỉ bằng nhau. Khi đó kết quả chuyển đổi xem như được chứa trong các bit CVR3:CVR0. Các thanh ghi liên quan đến bộ tạo điện áp so sánh này bao gồm:

- ✓ Thanh ghi CVRCON (địa chỉ 9Dh): thanh ghi trực tiếp điều khiển bộ so sánh điện áp.
- ✓ Thanh ghi CMCON (địa chỉ 9Ch): thanh ghi điều khiển bộ Comparator.

#### **4.7. CCP**

CCP (Capture/Compare/PWM) bao gồm các thao tác trên các xung đếm cung cấp bởi các bộ đếm Timer1 và Timer2. PIC16F877A được tích hợp sẵn hai khối CCP: CCP1 và CCP2 mỗi CCP có một thanh ghi 16 bit (CCPR1H : CCPR1L và CCPR2H : CCPR2L), pin điều khiển dùng cho khối CCPx là RC2/CCP1 và RC1/T1OSI/CCP2.

Các chức năng của CCP bao gồm:

- ✓ Capture.
- ✓ So sánh (Compare).
- ✓ Điều chế độ rộng xung PWM (Pulse Width Modulation).

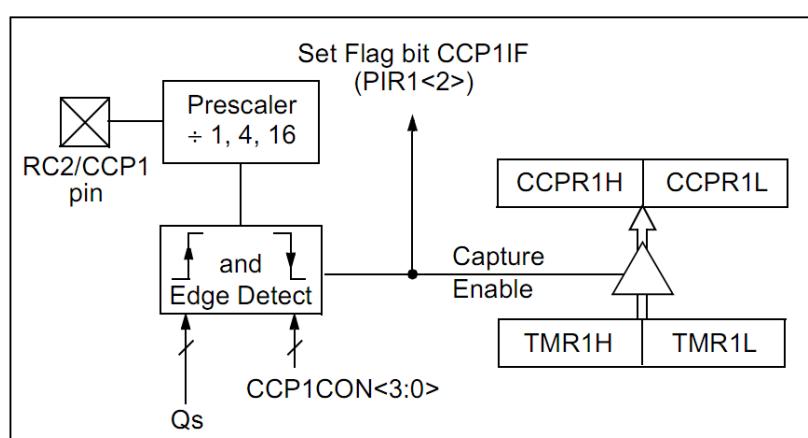
Cả CCP1 và CCP2 về nguyên tắc hoạt động đều giống nhau và chức năng của từng khối là khá độc lập. Tuy nhiên trong một số trường hợp ngoại lệ CCP1 và CCP2 có khả năng phối hợp với nhau để tạo ra các hiện tượng đặc biệt (Special event trigger) hoặc các tác động lên Timer1 và Timer2. Các trường hợp này được liệt kê trong bảng sau:

CCPx	CCPy	Tác động
Capture	Capture	Dùng chung nguồn xung clock từ TMR1
Capture	Compare	Tạo ra hiện tượng đặc biệt làm xóa TMR1
Compare	Compare	Tạo ra hiện tượng đặc biệt làm xóa TMR1
PWM	PWM	Dùng chung tần số xung clock và cùng chịu tác động của ngắt TMR2
PWM	Capture	Hoạt động độc lập
PWM	Compare	Hoạt động độc lập

##### **4.7.1. Capture Mode:**

Khi hoạt động ở chế độ Capture thì khi có một **hiện tượng** xảy ra tại pin RC2/CCP1 (hoặc RC1/T1OSI/CCP2), giá trị của thanh ghi TMR1 sẽ được đưa vào thanh ghi CCPR1 (CCPR2). Các **hiện tượng** được định nghĩa bởi các bit CCPxM3:CCPxM0 (CCPxCON<3:0>) và có thể là một trong các hiện tượng sau:

- ✓ Mỗi khi có cạnh xuống tại các pin CCP.
- ✓ Mỗi khi có cạnh lên.
- ✓ Mỗi cạnh lên thứ 4.
- ✓ Mỗi cạnh lên thứ 16.



Hình 4.15. Sơ đồ khối CCP (Capture mode).

Sau khi giá trị của thanh ghi TMR1 được đưa vào thanh ghi CCPRx, cờ ngắt CCPIF được set và phải được xóa bằng chương trình. Nếu hiện tượng tiếp theo xảy ra mà giá trị trong thanh ghi CCPRx chưa được xử lí, giá trị tiếp theo nhận được sẽ tự động được ghi đè lên giá trị cũ.

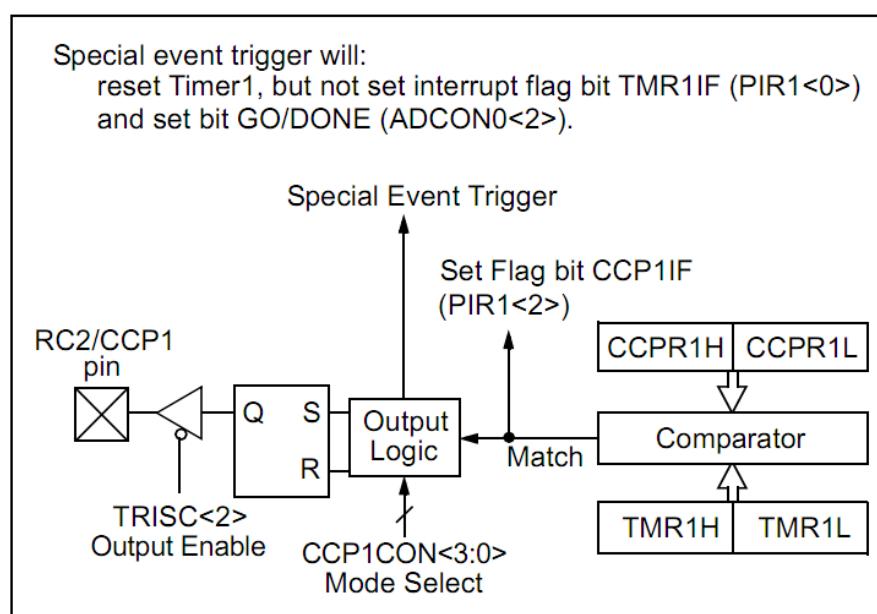
Một số điểm cần chú ý khi sử dụng CCP như sau:

- ✓ Các pin dùng cho khối CCP phải được án định là input (set các bit tương ứng trong thanh ghi TRISC). Khi án định các pin dùng cho khối CCP là output, việc đưa giá trị vào PORTC cũng có thể gây ra các hiện tượng? tác động lên khối CCP do trạng thái của pin thay đổi.
- ✓ Timer1 phải được hoạt động ở chế độ Timer hoặc chế độ đếm đồng bộ.
- ✓ Tránh sử dụng ngắt CCP bằng cách clear bit CCPXIE (thanh ghi PIE1), cờ ngắt CCPIF nên được xóa bằng phần mềm mỗi khi được set để tiếp tục nhận định được trạng thái hoạt động của CCP.
- ✓ CCP còn được tích hợp bộ chia tần số prescaler được điều khiển bởi các bit CCPxM3:CCPxM0. Việc thay đổi đối tượng tác động của prescaler có thể tạo ra hoạt động ngắt. Prescaler được xóa khi CCP không hoạt động hoặc khi reset.

Xem các thanh ghi điều khiển khối CCP.

#### 4.7.2. Compare Mode:

Khi hoạt động ở chế độ Compare, giá trị trong thanh ghi CCPRx sẽ thường xuyên được so sánh với giá trị trong thanh ghi TMR1. Khi hai thanh ghi chứa giá trị bằng nhau, các pin của CCP được thay đổi trạng thái (được đưa lên mức cao, đưa xuống mức thấp hoặc giữ nguyên trạng thái), đồng thời cờ ngắt CCPIF cũng sẽ được set. Sự thay đổi trạng thái của pin có thể được điều khiển bởi các bit CCPxM3:CCPxM0 (CCPxCON <3:0>).



Hình 4.16. Sơ đồ khối CCP (Compare mode).

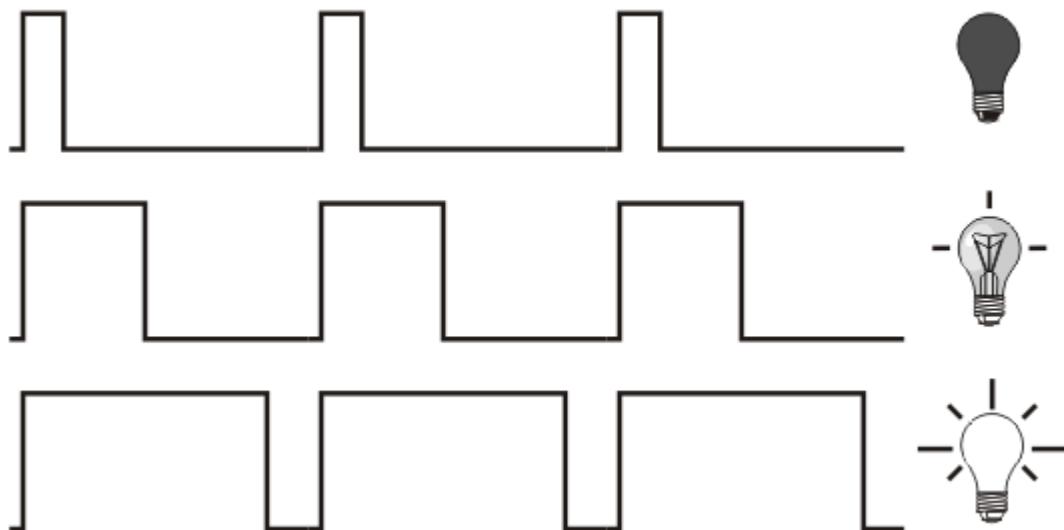
Tương tự như ở chế độ Capture, Timer1 phải được án định chế độ hoạt động là timer

hoặc đếm đồng bộ. Ngoài ra, khi ở chế độ Compare, CCP có khả năng tạo ra hiện tượng đặc biệt (Special Event trigger) làm reset giá trị thanh ghi TMR1 và khởi động bộ chuyển đổi ADC. Điều này cho phép ta điều khiển giá trị thanh ghi TMR1 một cách linh động hơn.

#### 4.7.3. PWM Mode:

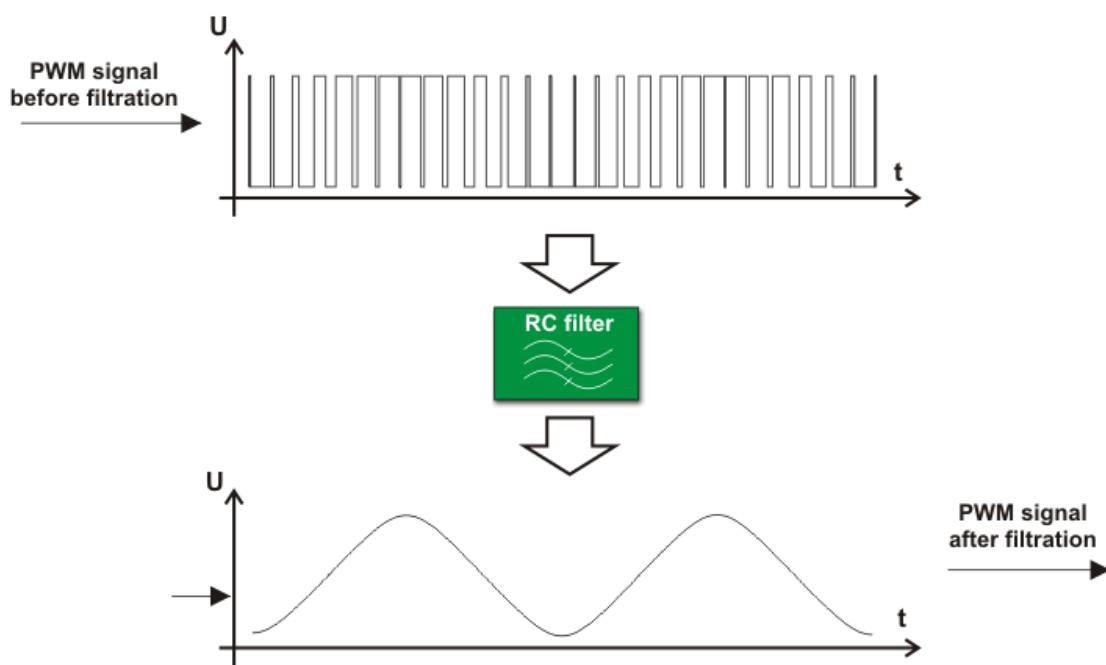
##### Phương pháp điều xung PWM là gì?

Phương pháp điều xung PWM (Pulse Width Modulation) là phương pháp điều chỉnh điện áp ra tải, hay nói cách khác, là phương pháp điều chế dựa trên sự thay đổi độ rộng của chuỗi xung vuông, dẫn đến sự thay đổi điện áp ra.



Hình 4.17. Nguyên lý hoạt động PWM

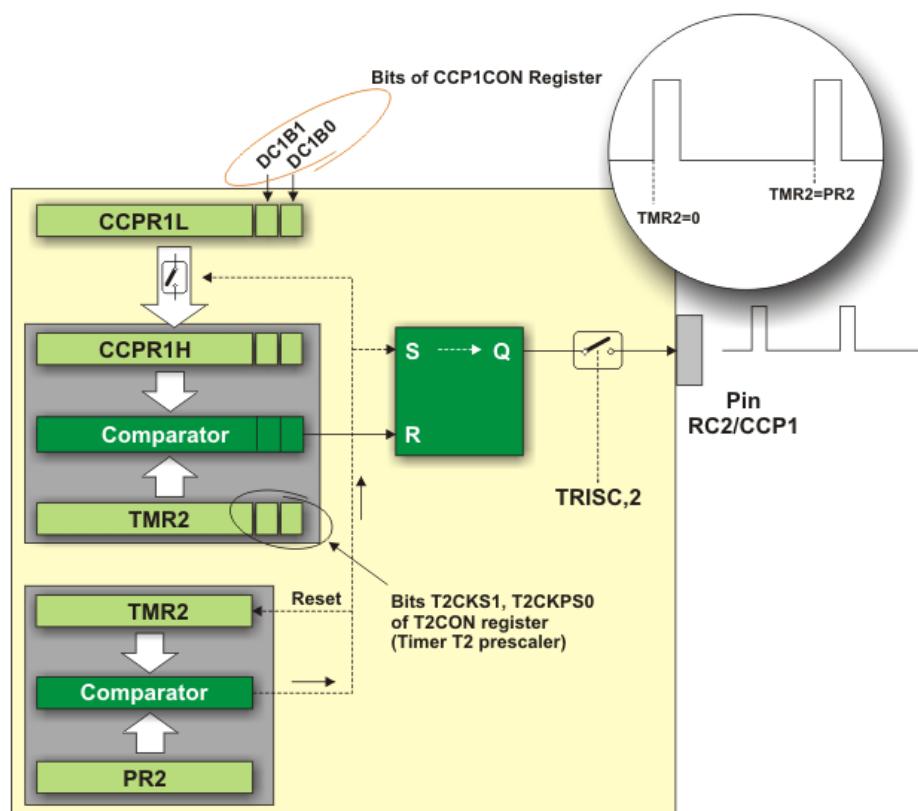
Tín hiệu PWM còn có thể tạo ra dạng sóng khác nhau nhờ mạch lọc:



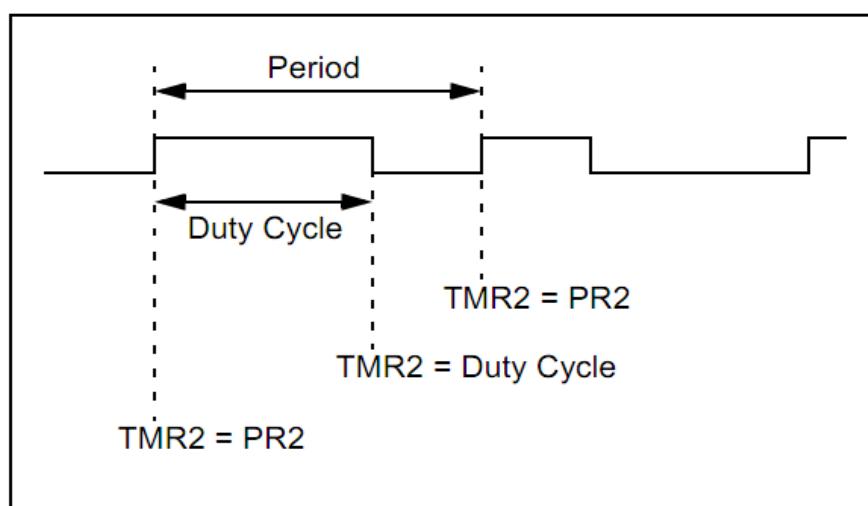
Hình 4.18. Chế độ PWM với mạch lọc.

Khi hoạt động ở chế độ PWM (Pulse Width Modulation \_ khói điều chế độ rộng xung), tín hiệu sau khi điều chế sẽ được đưa ra các pin của khói CCP (cần xác định các pin này là output). Để sử dụng chức năng điều chế này trước tiên ta cần tiến hành các bước cài đặt sau:

- Thiết lập thời gian của 1 chu kì của xung điều chế cho PWM (period) bằng cách đưa giá trị thích hợp vào thanh ghi PR2.
  - Thiết lập độ rộng xung cần điều chế (duty cycle) bằng cách đưa giá trị vào thanh ghi CCPRxL và các bit CCP1CON<5:4>.
  - Điều khiển các pin của CCP là output bằng cách clear các bit tương ứng trong thanh ghi TRISC.
  - Thiết lập giá trị bộ chia tần số prescaler của Timer2 và cho phép Timer2 hoạt động bằng cách đưa giá trị thích hợp vào thanh ghi T2CON.
  - Cho phép CCP hoạt động ở chế độ PWM.



Hình 4.19. Sơ đồ khối CCP (PWM mode).



Hình 4.20. Các tham số của PWM

Trong đó giá trị 1 chu kì (period) của xung điều chế được tính bằng công thức:

$$\text{PWM period} = [(PR2)+1]*4*TOSC*(\text{giá trị bộ chia tần số của TMR2-1,4,16}).$$

Bộ chia tần số prescaler của Timer2 chỉ có thể nhận các giá trị 1,4 hoặc 16 (xem lại Timer2 để biết thêm chi tiết). Khi giá trị thanh ghi PR2 bằng với giá trị thanh ghi TMR2 thì quá trình sau xảy ra:

- ✓ Thanh ghi TMR2 tự động được xóa.
- ✓ Pin của khối CCP được set.
- ✓ Giá trị thanh ghi CCPRxL (chứa giá trị án định độ rộng xung điều chế duty cycle) được đưa vào thanh ghi CCPRxH.

$$\text{PWM duty cycle} = (\text{CCPRxL:CCPxCON<5:4>})*TOSC*(\text{giá trị bộ chia tần số TMR2})$$

Như vậy 2 bit CCPxCON<5:4> sẽ chứa 2 bit LSB. Thanh ghi CCPRxL chứa byte cao của giá trị quyết định độ rộng xung. Thanh ghi CCPRxH đóng vai trò là buffer cho khối PWM. Khi giá trị trong thanh ghi CCPRxH bằng với giá trị trong thanh ghi TMR2 và hai bit CCPxCON<5:4> bằng với giá trị 2 bit của bộ chia tần số prescaler, pin của khối CCP lại được đưa về mức thấp, như vậy ta có được hình ảnh của xung điều chế tại ngõ ra của khối PWM như hình 2.14.

Một số điểm cần chú ý khi sử dụng khối PWM:

- ✓ Timer2 có hai bộ chia tần số prescaler và postscaler. Tuy nhiên bộ postscaler không được sử dụng trong quá trình điều chế độ rộng xung của khối PWM.
- ✓ Nếu thời gian duty cycle dài hơn thời gian chu kì xung period thì xung ngõ ra tiếp tục được giữ ở mức cao sau khi giá trị PR2 bằng với giá trị TMR2.

**Thanh ghi CCPR1L** (địa chỉ 15h): Thanh ghi chứa 8 bit thấp của khối CCP1.

**Thanh ghi CCPR1H** (địa chỉ 16h): Thanh ghi chứa 8 bit cao của khối CCP1.

**Thanh ghi CCP1CON và thanh ghi CCP2CON:** địa chỉ 17h (CCP1CON) và 1Dh (SSP2CON)

Thanh ghi điều khiển khối CCP1.

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CCPxX	CCPxY	CCPxM3	CCPxM2	CCPxM1	CCPxM0

bit 7

bit 0

Bit 7,6 Không có tác dụng và mặc định mang giá trị 0.

Bit 5,4 CCPxX:CCPxY: PWM least Significant bits (các bit này không có tác dụng ở chế độ Capture và Compare)

Ở chế độ PWM, đây là 2 bit MSB chứa giá trị tính độ rộng xung (duty cycle) của khối PWM (8 bit còn lại được chứa trong thanh ghi CCPRxL).

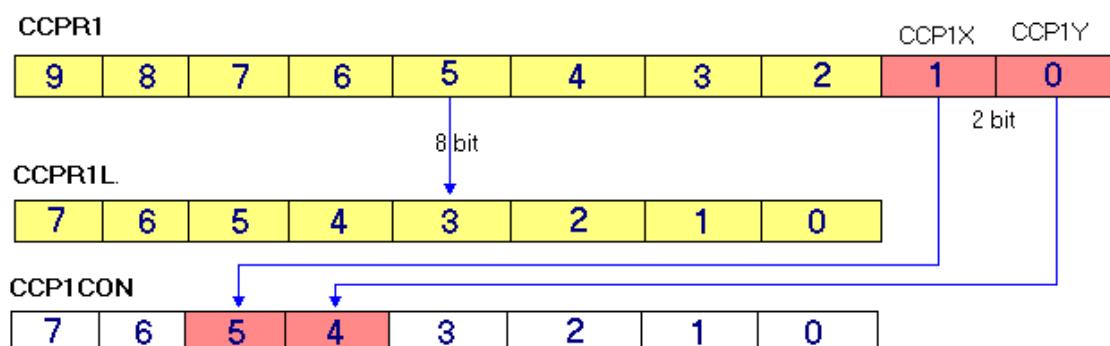
Bit 3-0 CCPxM3:CCPxM0 CCPx Mode Select bit

Các bit dùng để xác lập các chế độ hoạt động của khối CCPx

- 0000 không cho phép CCPx (hoặc dùng để reset CCPx)
- 0100 CCPx hoạt động ở chế độ Capture, “hiện tượng” được thiết lập là mỗi cạnh xuống tại pin dùng cho khối CCPx.

- 0101 CCPx hoạt động ở chế độ Capture, “hiện tượng” được thiết lập là mỗi cạnh lên tại pin dùng cho khối CCPx.
- 0110 CCPx hoạt động ở chế độ Capture, “hiện tượng” được thiết lập là mỗi cạnh lên thứ 4 tại pin dùng cho khối CCPx.
- 0111 CCPx hoạt động ở chế độ Capture, “hiện tượng” được thiết lập là mỗi cạnh lên thứ 16 tại pin dùng cho khối CCPx.
- 1000 CCPx hoạt động ở chế độ Compare, ngõ ra được đưa lên mức cao và bit CCPxIF được set khi các giá trị cần so sánh bằng nhau.
- 1001 CCPx hoạt động ở chế độ Compare, ngõ ra được xuống mức thấp và bit CCPxIF được set khi các giá trị cần so sánh bằng nhau.
- 1010 CCPx hoạt động ở chế độ Compare, khi các giá trị cần so sánh bằng nhau, ngắt xảy ra, bit CCPxIF được set và trạng thái pin output không bị ảnh hưởng.
- 1011 CCPx hoạt động ở chế độ Compare, khi các giá trị cần so sánh bằng nhau, xung trigger đặc biệt (Trigger Special Event) sẽ được tạo ra, khi đó cờ ngắt CCPxIF được set, các pin output không thay đổi trạng thái, CCp1 reset Timer1, CCP2 reset Timer1 và khởi động khối ADC. 11xx CCPx hoạt động ở chế độ PWM.

Ví dụ:



Xác định các thông số để tạo ra 1 sóng vuông có tần số 40Khz, duty=50%, thạch anh sử dụng 4Mhz, prescale=1

- + Chu kỳ T =  $1/40e3 = 2.5e-5$  sec
- + XTAL Tosc =  $1/4e6 = 2.5e-7$  sec
- + PR2=  $(2.5e-5 / (4 * 2.5e-7 * 1)) - 1 = 24$
- + Dpwm =  $(50 * 2.5e-5) / 100 = 1.25e-5$  sec
- + CCPR1=  $(1.25e-5) / (2.5e-7 * 1) = 50 \rightarrow 00001100\ 10$

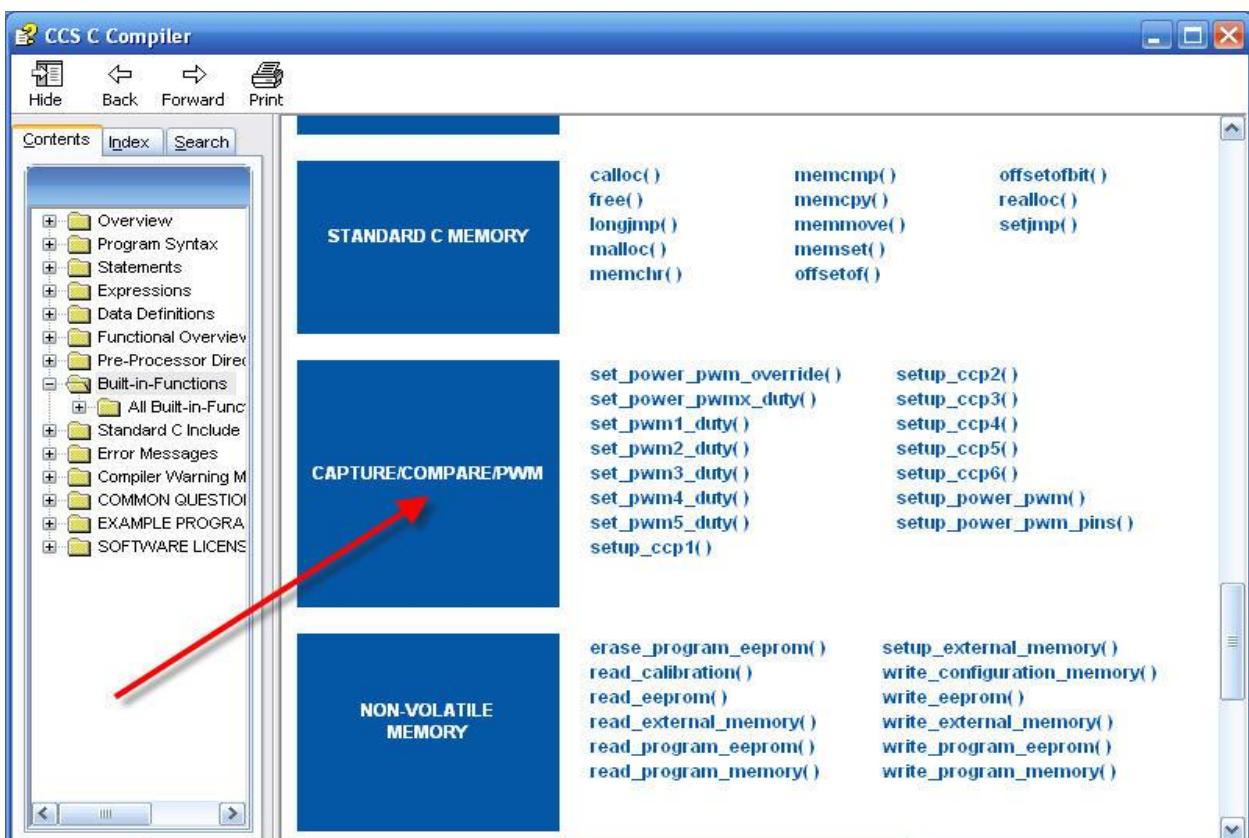
```

void Init_PWM(void)
{
    PR2      = 24;           //Set TIMER2 frequency
    CCPR1L = 0b00001100;   //Set TIMER2 duty cycle
    CCP1CON = 0B00101111; //Set x,y CCP1CON<5:4>
                           //CCP1CON<3:0> = 11xx =  PWM mode
    TMR2    = 0;            //Clear TMR2 first
    T2CON   = 0b01111000;
}

```

}

**4.7.4. Hàm trong CCS:** Để xem các hàm CCS hỗ trợ dung Timer chúng ta vào CCS C Compiler Help => Built-in-Funtions => A/D CONVERSION.



Hình 4.21. Các hàm CCS hỗ trợ dành cho CCP

CCS luôn tạo sẵn các tên danh định C như là các biến trỏ tới CCP1 và CCP2 là: CCP\_1(16 bit ), CCP\_2 (16 bit), CCP\_1\_HIGH (byte cao của CCP1), CCP\_1\_LOW, CCP\_2\_HIGH, CCP\_2\_LOW, bạn không cần khai báo . Dùng luôn các tên đó để lấy trị khi dùng module Cap, hay gán trị khi dùng Compare. Bạn có thể thấy điều này khi mở mục RAM symbol map quan sát phân bổ bộ nhớ.

➤ **Setup\_CCPx ( mode ):** Dùng trước tiên để thiết lập chế độ hoạt động hay vô hiệu tính năng CCP với:

- ✓ X= 1,2, . . . tên chân CCP có trên chip .
- ✓ Mode là 1 trong các hằng số sau: (các hằng số khác có thể có thêm trong file \*.h và tuỳ VDK). Mode có thể là:
  - CCP\_OFF: tắt chức năng CCP, RC sẽ là chân I/O .
  - CCP\_CAPTURE\_RE : capture khi có cạnh lên
  - CCP\_CAPTURE\_FE : capture khi có cạnh xuống
  - CCP\_CAPTURE\_DIV\_4 : chỉ capture sau khi đếm đủ 4 cạnh lên ( 4 xung ).
  - CCP\_CAPTURE\_DIV\_16 : chỉ capture sau khi đếm đủ 16 cạnh lên ( 16 xung ).

=> Sử dụng để làm dãn thời gian VDK để dành cho công việc khác thay vì cứ update từng xung .

✓ Chế độ compare:

- CCP\_COMPARE\_SET\_ON\_MATCH : xuất xung mức cao khi TMR1=CCPx
- CCP\_COMPARE\_CLR\_ON\_MATCH: xuất xung mức thấp khi TMR1=CCPx
- CCP\_COMPARE\_INT : ngắt khi TMR1=CCPx
- CCP\_COMPARE\_RESET\_TIMER : reset TMR1 =0 khi TMR1=CCPx

✓ Chế độ PWM:

- CCP\_PWM : bật chế độ PWM
- CCP\_PWM\_PLUS\_1 : không rõ chức năng
- CCP\_PWM\_PLUS\_2 : không rõ chức năng
- CCP\_PWM\_PLUS\_3 : không rõ chức năng

➤ **Set\_CCPx\_duty ( value ):** set\_pwm1\_duty(value) hay set\_pwm2\_duty(value)

Dùng set duty của xung trong chế độ PWM. Nó ghi 10 bit giá trị vào thanh ghi CCPx. Nếu value chỉ là 8 bit, nó dịch thêm 2 để đủ 10 bit nạp vào CCPx.

Tùy độ phân giải mà giá trị của value không phải lúc nào cũng đạt tới 1023

Do đó, value = 512 không có nghĩa là duty = 50 %

✓ Value: biến hay hằng, giá trị 8 hay 16 bit.

Nếu value là giá trị kiểu int 8bit: duty\_cycle = value / (period+1)

Nếu value là giá trị long int 16bit: duty\_cycle = value&1023 / [4\*(period+1)]

✓ x= 0 ,1 ,2 . . . :tên chân CCPx

➤ Ngắt INT\_CCP1, INT\_CCP2

**Ví dụ:** Ta muốn điều xung PWM với tần số 10kHz với tần số thạch anh (fosc) sử dụng là 20MHz (value 8bit).

$$f=fosc/[4*mode*(period+1)]$$

$$<=> 10000 = 20000000/[ 4*mode*(period+1) ]$$

$$<=> mode(period+1) = 500$$

Với mode = [1,4,16] và period = 0-255 ta có thể chọn:

$$mode = 4; period = 124$$

$$mode = 16; period = 32$$

Để cho việc điều xung được “mịn” (chọn được nhiều giá trị duty cycle) ta chọn mode = 4 và period = 124.

$$value = 30 => duty_cycle = 30 / ( 124+1 ) = 0.32 = 32\%$$

$$value = 63 => duty_cycle = 63 / ( 124+1 ) = 0.504 = 50.4\%$$

$$value = 113 => duty_cycle = 113 / ( 124+1 ) = 0.904 = 90.4\%$$

**Code:**

```
setup_timer_2(T2_DIV_BY_4,124,1);
setup_ccp1(CCP_PWM);
set_pwm1_duty(30);
```

#### 4.7.5. Bài tập: Thực hiện băm xung trên chân RC2/ CCP1

PWM Frequency	1.22 kHz
Timer Prescaler (1, 4, 16)	16
PR2 Value	0xFFh
Maximum Resolution (bits)	10

```
*****
*
```

\* PIC Training Course

\* Nguyen Tat Thanh University

\*

\*\*\*\*\*\*/

\*\*\*\*\*

\*

\* Module : main.c

\* Description : Create PWM at CCP1,CCP2

\* Tool : ccs v5.00xx

\* Chip : 16F877A

\* History : 7/2011

\* Version : v1.0

\* Author : Nguyen Huu Luan

\* Mail : nvn.nhl@gmail.com

\* Website : nt.edu.vn

\* Notes :

\*\*\*\*\*\*/

//////////

// Su dung rc1, rc2 output

//

// Sử dụng ccp dieu khien xuat pwm ra rc1, rc2

//

// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh //

//

//////////

/\*

\* Keywords Proteus

\* Pic 16f877a.

\* Led: Led-green, Led-red.....

\* Dien tro: resistors.

\* oscillate: kiem tra tan so pwm

\*/

\*\*\*\*\*\*/

```
#include <16f877a.h>
#include <def_877a.h>
#device *=16 ADC=8
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT,
NOLVP, NOCPD, NOWRT
#use delay(clock=20000000)
int16 j;
void main()
{
    trisc=0;//output
    portc=0xff;//dau tien led sang
    setup_ccp1(CCP_PWM); // che do PWM
    setup_timer_2(T2_DIV_BY_16, 255, 1);
        //chu ky PWM se la:
        //(1/20.000.000)*4*16*(255+1)=819.2(us)
```

//t2div=4

```

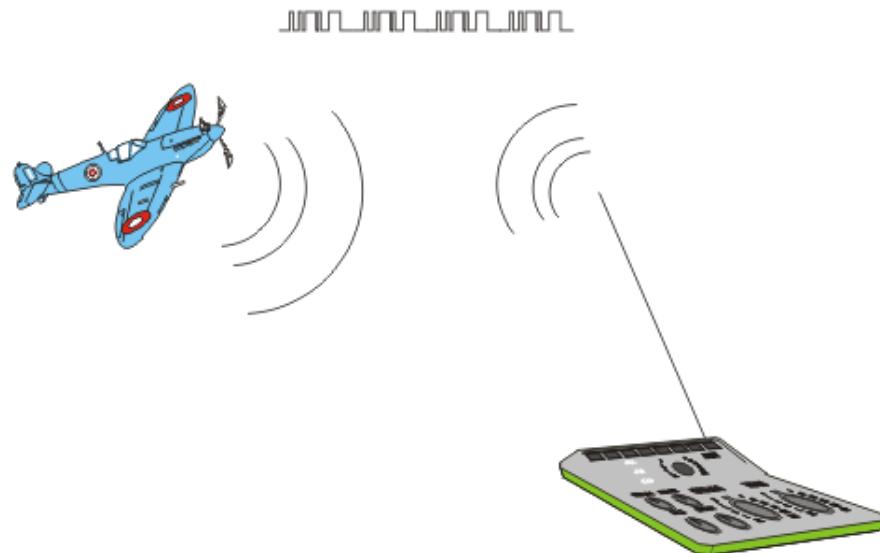
while(1)
{
    for(j=1023;j>=0;j--)
    {
        delay_ms(10);
        SET_PWM1_DUTY(j );//dat duty cycle
    }
}

```

#### 4.8. USART

USART (Universal Synchronous Asynchronous Receiver Transmitter) là một trong hai chuẩn giao tiếp nối tiếp. USART còn được gọi là giao diện giao tiếp nối tiếp SCI (Serial Communication Interface). Có thể sử dụng giao diện này cho các giao tiếp với các thiết bị ngoại vi, với các vi điều khiển khác hay với máy tính. Các dạng của giao diện USART ngoại vi bao gồm:

- ✓ Bất đồng bộ (Asynchronous).
- ✓ Đồng bộ\_ Master mode.
- ✓ Đồng bộ\_ Slave mode.



Hình 4.22. Giao tiếp USART

Hai pin dùng cho giao diện này là RC6/TX/CK và RC7/RX/DT, trong đó RC6/TX/CK dùng để truyền xung clock (baud rate) và RC7/RX/DT dùng để truyền data. Trong trường hợp này ta phải set bit TRISC<7:6> và SPEN (RCSTA<7>) để cho phép giao diện USART. PIC16F877A được tích hợp sẵn bộ tạo tốc độ baud BRG (Baud Rate Generator) 8 bit dùng cho giao diện USART. BRG thực chất là một bộ đếm có thể được sử dụng cho cả hai dạng đồng bộ và bất đồng bộ và được điều khiển bởi thanh ghi PSBRG. Ở dạng bất đồng bộ, BRG còn được điều khiển bởi bit BRGH (TXSTA<2>). Ở dạng đồng bộ tác động của bit BRGH được bỏ qua. Tốc độ baud do BRG tạo ra được tính theo công thức sau:

**Thanh ghi SPBRG: địa chỉ 99h**

SYNC	<b>BRGH = 0 (Low Speed)</b>	<b>BRGH = 1 (High Speed)</b>
0	(Asynchronous) Baud Rate = FOSC/(64 (X + 1))	Baud Rate = FOSC/(16 (X + 1))
1	(Synchronous) Baud Rate = FOSC/(4 (X + 1))	N/A

Trong đó X là giá trị của thanh ghi RSBRG ( X là số nguyên và  $0 < X < 255$ ).

Các thanh ghi liên quan đến BRG bao gồm:

- o TXSTA (địa chỉ 98h): chọn chế độ đồng bộ hay bất đồng bộ ( bit SYNC) và chọn mức tốc độ baud (bit BRGH).
- o RCSTA (địa chỉ 18h): cho phép hoạt động cổng nối tiếp (bit SPEN).
- o RSBRG (địa chỉ 99h): quyết định tốc độ baud.

**Thanh ghi TXSTA (địa chỉ 98h):**

Thanh ghi chứa các bit trạng thái và điều khiển việc truyền dữ liệu thông qua chuẩn giao tiếp USART.

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D

bit 7

bit 0

Bit 7 CSRC Clock Source Select bit

Ở chế độ bất đồng bộ: không cần quan tâm.

Ở chế độ đồng bộ:

CSRC = 1 Master mode (xung clock được lấy từ bộ tạo xung BRG).

CSRC = 0 Slave mode (xung clock được nhận từ bên ngoài).

Bit 6 TX-9 9-bit Transmit Enable bit

TX-9 = 1 truyền dữ liệu 9 bit.

TX-9 = 0 truyền dữ liệu 8 bit.

Bit 5 TXEN Transmit Enable bit

TXEN = 1 cho phép truyền.

TXEN = 0 không cho phép truyền.

Bit 4 SYNC USART Mode Select bit    SYNC = 1 dạng đồng bộ

SYNC = 0 dạng bất đồng bộ.

Bit 3 Không cần quan tâm và mặc định mang giá trị 0.

Bit 2 BRGH High Baud Rate Select bit

Bit này chỉ có tác dụng ở chế độ bất đồng bộ.

BRGH = 1 tốc độ cao.

BRGL = 0 tốc độ thấp.

Bit 1 TRMT Transmit Shift Register Status bit

TRMT = 1 thanh ghi TSR không có dữ liệu.

TRMT = 0 thanh ghi TSR có chứa dữ liệu.

Bit 0 TX9D

Bit này chứa bit dữ liệu thứ 9 khi dữ liệu truyền nhận là 9 bit.

**Thanh ghi RCSTA (địa chỉ 18h):**

Thanh ghi chứa các bit trạng thái và các bit điều khiển quá trình nhận dữ liệu qua chuẩn giao tiếp USART.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D

bit 7

bit 0

Bit 7 SPEN Serial Port Enable bit

SPEN = 1 Cho phép cổng giao tiếp USART (pin RC7/RX/DT và RC6/TX/CK).

SPEN = 0 không cho phép cổng giao tiếp USART.

Bit 6 RX9 9-bit Receive Enable bit

RX9 = 1 nhận 9 bit dữ liệu.

RX9 = 0 nhận 8 bit dữ liệu.

Bit 5 SREN Single Receive Enable bit

Ở chế độ USART bất đồng bộ: bit này không cần quan tâm.

Ở chế độ USART Master đồng bộ:

SREN = 1 cho phép chức năng nhận 1 byte dữ liệu (8 bit hoặc 9 bit).

SREN = 0 không cho phép chức năng nhận 1 byte dữ liệu.

Bit 4 CREN Continous Receive Enable bit

Ở chế độ bất đồng bộ:

CREN = 1 cho phép nhận 1 chuỗi dữ liệu liên tục.

CREN = 0 không cho phép nhận 1 chuỗi dữ liệu liên tục.

Ở chế độ bắt đồng bộ:

CREN = 1 cho phép nhận dữ liệu cho tới khi xóa bit CREN.

CREN = 0 không cho phép nhận chuỗi dữ liệu.

Bit 3 ADDEN Address Detect Enable bit

Ở chế độ USART bất đồng bộ 9 bit

ADDEN = 1 cho phép xác nhận địa chỉ, khi bit RSR<8> được set thì ngắt được cho phép thực thi và giá trị trong buffer được nhận vào.

ADDEN = 0 không cho phép xác nhận địa chỉ, các byte dữ liệu được nhận vào và bit thứ 9 có thể được sử dụng như là bit parity.

Bit 2 FERR Framing Error bit

FERR = 1 xuất hiện lỗi “Framing” trong quá trình truyền nhận dữ liệu.

FERR = 0 không xuất hiện lỗi “Framing” trong quá trình truyền nhận dữ liệu.

Bit 1 OERR Overrun Error bit,

OERR = 1 xuất hiện lỗi “Overrun”

OERR = 0 không xuất hiện lỗi “Overrun”

Bit 0 RX9D

Bit này chứa bit dữ liệu thứ 9 của dữ liệu truyền nhận.

#### **4.8.1. USART Bất đồng bộ:**

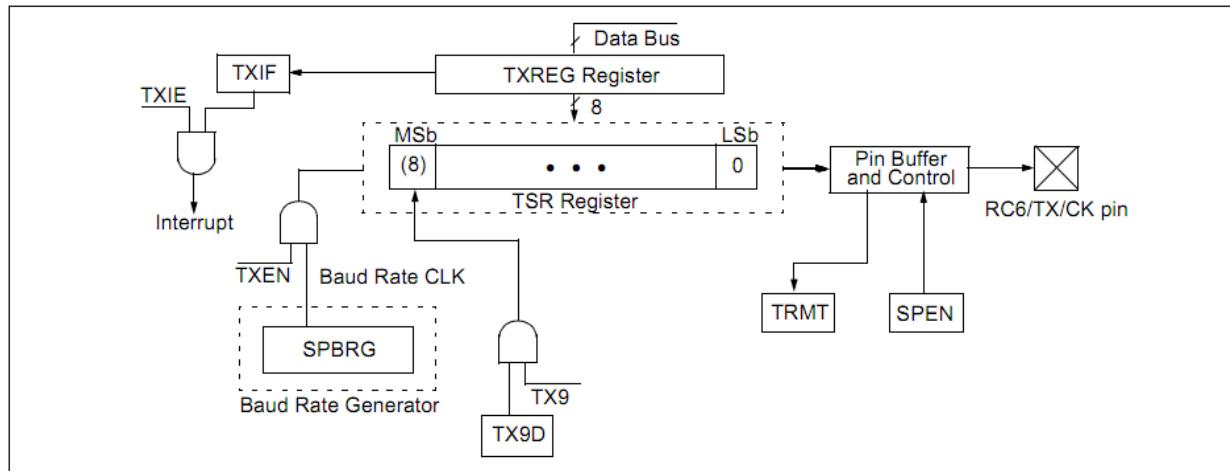
Ở chế độ truyền này USART hoạt động theo chuẩn NRZ (None-Return-to-Zero), nghĩa là các bit truyền đi sẽ bao gồm 1 bit Start, 8 hay 9 bit dữ liệu (thông thường là 8 bit) và 1 bit Stop.

Bit LSB sẽ được truyền đi trước. Các khối truyền và nhận data độc lập với nhau sẽ dùng chung tần số tương ứng với tốc độ baud cho quá trình dịch dữ liệu (tốc độ baud gấp 16 hay 64 lần tốc độ dịch dữ liệu tùy theo giá trị của bit BRGH), và để đảm bảo tính hiệu quả của dữ liệu thì hai khối

truyền và nhận phải dùng chung một định dạng dữ liệu.

#### 4.8.1.1. Truyền dữ liệu qua chuẩn giao tiếp USART bất đồng bộ:

Thành phần quan trọng nhất của khối truyền dữ liệu là thanh ghi dịch dữ liệu TSR (Transmit Shift Register). Thanh ghi TSR sẽ lấy dữ liệu từ thanh ghi đệm dùng cho quá trình truyền dữ liệu TXREG. Dữ liệu cần truyền phải được đưa trước vào thanh ghi TXREG. Ngay sau khi bit Stop của dữ liệu cần truyền trước đó được truyền xong, dữ liệu từ thanh ghi TXREG sẽ được đưa vào thanh ghi TSR, thanh ghi TXREG bị rỗng, ngắt xảy ra và cờ hiệu TXIF (PIR1<4>) được set. Ngắt này được điều khiển bởi bit TXIE (PIE1<4>). Cờ hiệu TXIF vẫn được set bất chấp trạng thái của bit TXIE hay tác động của chương trình (không thể xóa TXIF bằng chương trình) mà chỉ reset về 0 khi có dữ liệu mới được đưa vào thanh ghi TXREG.



Hình 4.23. Sơ đồ khái niệm của khối truyền dữ liệu USART

Trong khi cờ hiệu TXIF đóng vai trò chỉ thị trạng thái thanh ghi TXREG thì cờ hiệu TRMT (TXSTA<1>) có nhiệm vụ thể hiện trạng thái thanh ghi TSR. Khi thanh ghi TSR rỗng, bit TRMT sẽ được set. Bit này chỉ đọc và không có ngắt nào được gắn với trạng thái của nó. Một điểm cần chú ý nữa là thanh ghi TSR không có trong bộ nhớ dữ liệu và chỉ được điều khiển bởi CPU.

Khối truyền dữ liệu được cho phép hoạt động khi bit TXEN (TXSTA<5>) được set. Quá trình truyền dữ liệu chỉ thực sự bắt đầu khi đã có dữ liệu trong thanh ghi TXREG và xung truyền baud được tạo ra. Khi khối truyền dữ liệu được khởi động lần đầu tiên, thanh ghi TSR rỗng. Tại thời điểm đó, dữ liệu đưa vào thanh ghi TXREG ngay lập tức được load vào thanh ghi TSR và thanh ghi TXREG bị rỗng. Lúc này ta có thể hình thành một chuỗi dữ liệu liên tục cho quá trình truyền dữ liệu. Trong quá trình truyền dữ liệu nếu bit TXEN bị reset về 0, quá trình truyền kết thúc, khối truyền dữ liệu được reset và pin RC6/TX/CK chuyển đến trạng thái high-impedance.

Trong trường hợp dữ liệu cần truyền là 9 bit, bit TX9 (TXSTA<6>) được set và bit dữ liệu thứ 9 sẽ được lưu trong bit TX9D (TXSTA<0>). Nên ghi bit dữ liệu thứ 9 vào trước, vì khi ghi 8 bit dữ liệu vào thanh ghi TXREG trước có thể xảy ra trường hợp nội dung thanh ghi TXREG sẽ được load vào thanh ghi TSG trước, như vậy dữ liệu truyền đi sẽ bị sai khác so với yêu cầu.

Tóm lại, để truyền dữ liệu theo giao diện USART bất đồng bộ, ta cần thực hiện tuần tự các bước sau:

1. Tạo xung truyền baud bằng cách đưa các giá trị cần thiết vào thanh ghi RSBRG và bit điều khiển mức tốc độ baud BRGH.

2. Cho phép cổng giao diện nối tiếp nối tiếp bắt đồng bộ bằng cách clear bit SYNC và set bit PSEN.
3. Set bit TXIE nếu cần sử dụng ngắt truyền.
4. Set bit TX9 nếu định dạng dữ liệu cần truyền là 9 bit.
5. Set bit TXEN để cho phép truyền dữ liệu (lúc này bit TXIF cũng sẽ được set).
6. Nếu định dạng dữ liệu là 9 bit, đưa bit dữ liệu thứ 9 vào bit TX9D.
7. Đưa 8 bit dữ liệu cần truyền vào thanh ghi TXREG.
8. Nếu sử dụng ngắt truyền, cần kiểm tra lại các bit GIE và PEIE (thanh ghi INTCON).

Các thanh ghi liên quan đến quá trình truyền dữ liệu bằng giao diện USART bắt đồng bộ:

- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): cho phép tắt cả các ngắt.
- ✓ Thanh ghi PIR1 (địa chỉ 0Ch): chứa cờ hiệu TXIF.
- ✓ Thanh ghi PIE1 (địa chỉ 8Ch): chứa bit cho phép ngắt truyền TXIE.
- ✓ Thanh ghi RCSTA (địa chỉ 18h): chứa bit cho phép cổng truyền dữ liệu (hai pin RC6/TX/CK và RC7/RX/DT).
- ✓ Thanh ghi TXREG (địa chỉ 19h): thanh ghi chứa dữ liệu cần truyền.
- ✓ Thanh ghi TXSTA (địa chỉ 98h): xác lập các thông số cho giao diện. Xem lại phần USART
- ✓ Thanh ghi SPBRG (địa chỉ 99h): quyết định tốc độ baud.

**Thanh ghi INTCON (0Bh, 8Bh, 10Bh, 18Bh):** thanh ghi cho phép đọc và ghi, chứa các bit điều khiển và các bit cờ hiệu khi timer0 bị tràn, ngắt ngoại vi RB0/INT và ngắt interrupt-on-change tại các chân của PORTB.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
bit 7				bit 0			

Bit 7 GIE Global Interrupt Enable bit

GIE = 1 cho phép tắt cả các ngắt.

GIE = 0 không cho phép tắt cả các ngắt.

**Thanh ghi PIR1 (0Ch):** chứa cờ ngắt của các khối chức năng ngoại vi, các ngắt này được cho phép bởi các bit điều khiển chứa trong thanh ghi PIE1.

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF <sup>(1)</sup>	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7				bit 0			

Bit 4 TXIF USART Transmit Interrupt Flag bit

TXIF = 1 buffer truyền qua chuẩn giao tiếp USART rỗng.

TXIF = 0 buffer truyền qua chuẩn giao tiếp USART đầy.

**Thanh ghi PIE1 (8Ch):** chứa các bit điều khiển chi tiết các ngắt của các khối chức năng ngoại vi.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE <sup>(1)</sup>	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7				bit 0			

Bit 5 RCIE USART Receive Interrupt Enable bit

RCIE = 1 cho phép ngắt nhận USART

RCIE = 0 không cho phép ngắt nhận USART

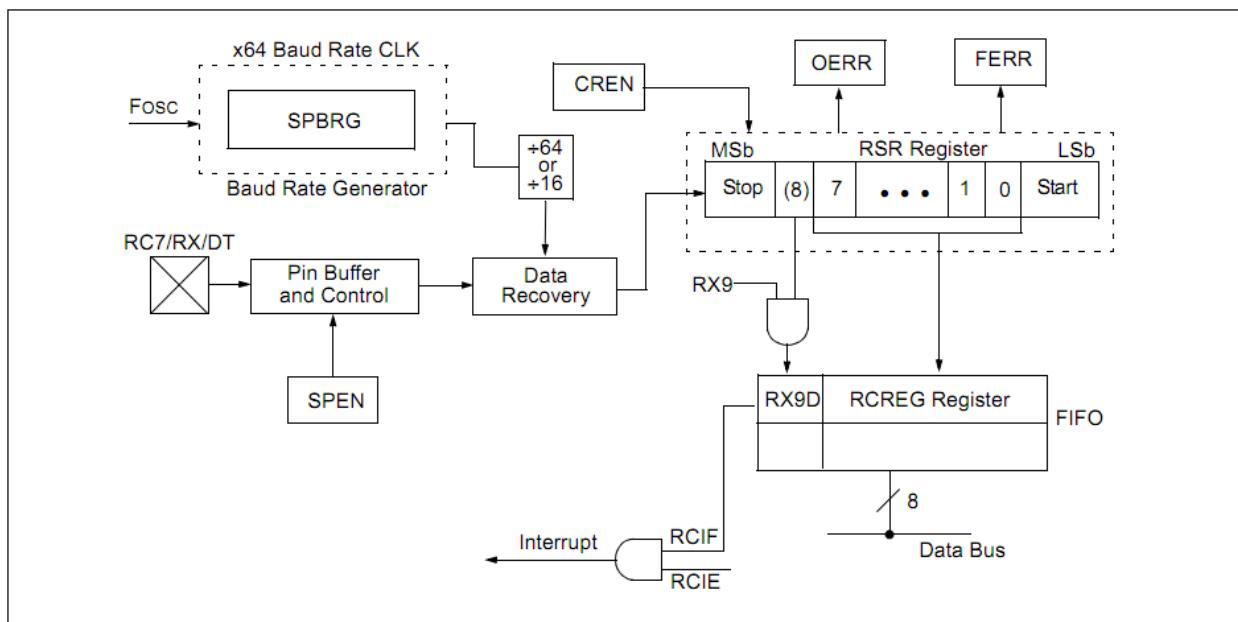
Bit 4 TXIE USART Transmit Interrupt Enable bit

TXIE = 1 cho phép ngắt truyền USART

TXIE = 0 không cho phép ngắt truyền USART

#### 4.8.1.2. Nhận dữ liệu qua chuẩn giao tiếp USART bắt đồng bộ:

Dữ liệu được đưa vào từ chân RC7/RX/DT sẽ kích hoạt khối phục hồi dữ liệu. Khối phục hồi dữ liệu thực chất là một bộ dịch dữ liệu có tốc độ cao và có tần số hoạt động gấp 16 lần hoặc 64 lần tần số baud. Trong khi đó tốc độ dịch của thanh ghi nhận dữ liệu sẽ bằng với tần số baud hoặc tần số của oscillator.



Hình 4.24. Sơ đồ khái niệm khái quát của khối nhận dữ liệu USART

Bit điều khiển cho phép khái quát nhận dữ liệu là bit RCEN (RCSTA<4>). Thành phần quan trọng nhất của khái quát nhận dữ liệu là thanh ghi nhận dữ liệu RSR (Receive Shift Register). Sau khi nhận diện bit Stop của dữ liệu truyền tới, dữ liệu nhận được

trong thanh ghi RSR sẽ được đưa vào thanh ghi RCGER, sau đó cờ hiệu RCIF (PIR1<5>) sẽ được set và ngắt nhận được kích hoạt. Ngắt này được điều khiển bởi bit RCIE (PIE1<5>). Bit cờ hiệu RCIF là bit chỉ đọc và không thể được tác động bởi chương trình. RCIF chỉ reset về 0 khi dữ liệu nhận vào ở thanh ghi RCGER đã được đọc và khi đó thanh ghi RCGER rỗng. Thanh ghi RCGER là thanh ghi có bộ đệm kép (double-buffered register) và hoạt động theo cơ chế FIFO (First In First Out) cho phép nhận 2 byte và byte thứ 3 tiếp tục được đưa vào thanh ghi RSR. Nếu sau khi nhận được bit Stop của byte dữ liệu thứ 3 mà thanh ghi RCGER vẫn còn đầy, cờ hiệu báo tràn dữ liệu (Overrun Error bit) OERR(RCSTA<1>) sẽ được set, dữ liệu trong thanh ghi RSR sẽ bị mất đi và quá trình đưa dữ liệu từ thanh ghi RSR vào thanh ghi RCGER sẽ bị gián đoạn.

Trong trường hợp này cần lấy hết dữ liệu ở thanh ghi RSREG vào trước khi tiếp tục nhận byte dữ liệu tiếp theo. Bit OERR phải được xóa bằng phần mềm và thực hiện bằng cách clear bit RCEN rồi set lại. Bit FERR (RCSTA<2>) sẽ được set khi phát hiện bit Stop của dữ liệu được nhận vào. Bit dữ liệu thứ 9 sẽ được đưa vào bit RX9D (RCSTA<0>). Khi đọc dữ liệu từ thanh ghi RCGER, hai bit FERR và RX9D sẽ nhận các giá trị mới. Do đó cần đọc dữ liệu từ thanh ghi RCSTA trước khi đọc dữ liệu từ thanh ghi RCGER để tránh bị mất dữ liệu.

Tóm lại, khi sử dụng giao diện nhận dữ liệu USART bắt đồng bộ cần tiến hành tuần tự các bước sau:

1. Thiết lập tốc độ baud (đưa giá trị thích hợp vào thanh ghi SPBRG và bit BRGH).
2. Cho phép cổng giao tiếp USART bắt đồng bộ (clear bit SYNC và set bit SPEN).
3. Nếu cần sử dụng ngắt nhận dữ liệu, set bit RCIE.
4. Nếu dữ liệu truyền nhận có định dạng là 9 bit, set bit RX9.
5. Cho phép nhận dữ liệu bằng cách set bit CREN.
6. Sau khi dữ liệu được nhận, bit RCIF sẽ được set và ngắt được kích hoạt (nếu bit RCIE được set).
7. Đọc giá trị thanh ghi RCSTA để đọc bit dữ liệu thứ 9 và kiểm tra xem quá trình nhận dữ liệu có bị lỗi không.
8. Đọc 8 bit dữ liệu từ thanh ghi RCREG.
9. Nếu quá trình truyền nhận có lỗi xảy ra, xóa lỗi bằng cách xóa bit CREN.
10. Nếu sử dụng ngắt nhận cần set bit GIE và PEIE (thanh ghi INTCON).

Các thanh ghi liên quan đến quá trình nhận dữ liệu bằng giao diện USART bắt đồng bộ:

- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): chứa các bit cho phép toàn bộ các ngắt (bit GIER và PEIE). Xem lại các bài trên.
- ✓ Thanh ghi PIR1 (địa chỉ 0Ch): chứa cờ hiệu RCIE. Xem lại các bài trên.
- ✓ Thanh ghi PIE1 (địa chỉ 8Ch): chứa bit cho phép ngắt RCIE. Xem lại các bài trên.
- ✓ Thanh ghi RCSTA (địa chỉ 18h): xác định các trạng thái trong quá trình nhận dữ liệu.
- ✓ Thanh ghi RCREG (địa chỉ 1Ah): Thanh ghi đóng vai trò là buffer đệm trong quá trình nhận dữ liệu qua chuẩn giao tiếp USART.
- ✓ Thanh ghi TXSTA (địa chỉ 98h): chứa các bit điều khiển SYNC và BRGH.
- ✓ Thanh ghi SPBRG (địa chỉ 99h): điều khiển tốc độ baud.

### **Thanh ghi RCSTA: địa chỉ 18h**

Thanh ghi chứa các bit trạng thái và các bit điều khiển quá trình nhận dữ liệu qua chuẩn giao tiếp USART.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D

bit 7

bit 0

Bit 7 SPEN Serial Port Enable bit

SPEN = 1 Cho phép cổng giao tiếp USART (pin RC7/RX/DT và RC6/TX/CK).

SPEN = 0 không cho phép cổng giao tiếp USART.

Bit 6 RX9 9-bit Receive Enable bit

RX9 = 1 nhận 9 bit dữ liệu.

RX9 = 0 nhận 8 bit dữ liệu.

Bit 5 SREN Single Receive Enable bit

Ở chế độ USART bắt đồng bộ: bit này không cần quan tâm.

Ở chế độ USART Master đồng bộ:

SREN = 1 cho phép chức năng nhận 1 byte dữ liệu (8 bit hoặc 9 bit).

SREN = 0 không cho phép chức năng nhận 1 byte dữ liệu.

Bit 4 CREN Continous Receive Enable bit

Ở chế độ bắt đồng bộ:

CREN = 1 cho phép nhận 1 chuỗi dữ liệu liên tục.

CREN = 0 không cho phép nhận 1 chuỗi dữ liệu liên tục.

Ở chế độ bắt đồng bộ:

CREN = 1 cho phép nhận dữ liệu cho tới khi xóa bit CREN.

CREN = 0 không cho phép nhận chuỗi dữ liệu.

Bit 3 ADDEN Address Detect Enable bit

Ở chế độ USART bắt đồng bộ 9 bit

ADDEN = 1 cho phép xác nhận địa chỉ, khi bit RSR<8> được set thì ngắt được cho phép thực thi và giá trị trong buffer được nhận vào.

ADDEN = 0 không cho phép xác nhận địa chỉ, các byte dữ liệu được nhận vào và bit thứ 9 có thể được sử dụng như là bit parity.

Bit 2 FERR Framing Error bit

FERR = 1 xuất hiện lỗi “Framing” trong quá trình truyền nhận dữ liệu.

FERR = 0 không xuất hiện lỗi “Framing” trong quá trình truyền nhận dữ liệu.

Bit 1 OERR Overrun Error bit,

OERR = 1 xuất hiện lỗi “Overrun”

OERR = 0 không xuất hiện lỗi “Overrun”

Bit 0 RX9D

Bit này chứa bit dữ liệu thứ 9 của dữ liệu truyền nhận.

#### **Thanh ghi TXSTA (địa chỉ 98h):**

Thanh ghi chứa các bit trạng thái và điều khiển việc truyền dữ liệu thông qua chuẩn giao tiếp USART.

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D

Bit 4 SYNC USART Mode Select bit    SYNC = 1 dạng đồng bộ

SYNC = 0 dạng bắt đồng bộ.

Bit 2 BRGH High Baud Rate Select bit

Bit này chỉ có tác dụng ở chế độ bắt đồng bộ.

BRGH = 1 tốc độ cao.

BRGL = 0 tốc độ thấp.

#### **4.8.2. USART Đồng bộ**

Giao diện USART đồng bộ được kích hoạt bằng cách set bit SYNC. Cổng giao tiếp nối tiếp vẫn là hai chân RC7/RX/DT, RC6/TX/CK và được cho phép bằng cách set bit SPEN. USART cho phép hai chế độ truyền nhận dữ liệu là Master mode và Slave mode. Master mode được kích hoạt bằng cách set bit CSRC (TXSTA<7>), Slave mode được kích hoạt bằng cách clear bit CSRC. Điểm khác biệt duy nhất giữa hai chế độ này là Master mode sẽ lấy xung clock đồng bộ từ bộ tao xung baud BRG còn Slave mode lấy xung clock đồng bộ từ bên ngoài qua chân RC6/TX/CK. Điều này cho phép Slave mode hoạt động cả khi vi điều khiển đang ở chế độ

sleep.

#### 4.8.2.1. Truyền dữ liệu qua chuẩn giao tiếp USART đồng bộ MASTER MODE

Tương tự như giao diện USART bắt đồng bộ, thành phần quan trọng nhất của hối truyền dữ liệu là thanh ghi dịch TSR (Transmit Shift Register). Thanh ghi này chỉ được điều khiển bởi CPU. Dữ liệu đưa vào thanh ghi TSR được chứa trong thanh ghi TXREG. Cờ hiệu của khôi truyền dữ liệu là bit TXIF (chỉ thị trạng thái thanh ghi TXREG), cờ hiệu này được gắn với một ngắt và bit điều khiển ngắt này là TXIE. Cờ hiệu chỉ thị trạng thái thanh ghi TSR là bit TRMT. Bit TXEN cho phép hay không cho phép truyền dữ liệu.

Các bước cần tiến hành khi truyền dữ liệu qua giao diện USART đồng bộ Master mode:

1. Tạo xung truyền baud bằng cách đưa các giá trị cần thiết vào thanh ghi RSBRG và bit điều khiển mức tốc độ baud BRGH.
2. Cho phép cổng giao diện nối tiếp nối tiếp đồng bộ bằng cách set bit SYNC, PSEN và CSRC.
3. Set bit TXIE nếu cần sử dụng ngắt truyền.
4. Set bit TX9 nếu định dạng dữ liệu cần truyền là 9 bit.
5. Set bit TXEN để cho phép truyền dữ liệu.
6. Nếu định dạng dữ liệu là 9 bit, đưa bit dữ liệu thứ 9 vào bit TX9D.
7. Đưa 8 bit dữ liệu cần truyền vào thanh ghi TXREG.
8. Nếu sử dụng ngắt truyền, cần kiểm tra lại các bit GIE và PEIE (thanh ghi INTCON).

Các thanh ghi liên quan đến quá trình truyền dữ liệu bằng giao diện USART đồng bộ Master mode:

- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): cho phép tắt cả các ngắt.
- ✓ Thanh ghi PIR1 (địa chỉ 0Ch): chứa cờ hiệu TXIF.
- ✓ Thanh ghi PIE1 (địa chỉ 8Ch): chứa bit cho phép ngắt truyền TXIE.
- ✓ Thanh ghi RCSTA (địa chỉ 18h): chứa bit cho phép cổng truyền dữ liệu (hai pin RC6/TX/CK và RC7/RX/DT).
- ✓ Thanh ghi TXREG (địa chỉ 19h): thanh ghi chứa dữ liệu cần truyền.
- ✓ Thanh ghi TXSTA (địa chỉ 98h): xác lập các thông số cho giao diện.
- ✓ Thanh ghi SPBRG (địa chỉ 99h): quyết định tốc độ baud.

Chi tiết về thanh ghi xem lại phần trên.

#### 4.8.2.2. Nhận dữ liệu qua chuẩn giao tiếp USART đồng bộ MASTER MODE

Cấu trúc khôi truyền dữ liệu là không đổi so với giao diện bắt đồng bộ, kể cả các cờ hiệu, ngắt nhận và các thao tác trên các thành phần đó. Điểm khác biệt duy nhất là giao diện này cho phép hai chế độ nhận dữ liệu, đó là chỉ nhận 1 word dữ liệu (set bit SCEN) hay nhận một chuỗi dữ liệu (set bit CREN) cho tới khi ta clear bit CREN. Nếu cả hai bit đều được set, bit điều khiển CREN sẽ được ưu tiên.

Các bước cần tiến hành khi nhận dữ liệu bằng giao diện USART đồng bộ Master mode:

1. Thiết lập tốc độ baud (đưa giá trị thích hợp vào thanh ghi SPBRG và bit BRGH).
2. Cho phép cổng giao tiếp USART bắt đồng bộ (set bit SYNC, SPEN và CSRC).
3. Clear bit CREN và SREN.
4. Nếu cần sử dụng ngắt nhận dữ liệu, set bit RCIE.
5. Nếu dữ liệu nhận có định dạng là 9 bit, set bit RX9.

6. Nếu chỉ nhận 1 word dữ liệu, set bit SREN, nếu nhận 1 chuỗi word dữ liệu, set bit CREN.
7. Sau khi dữ liệu được nhận, bit RCIF sẽ được set và ngắt được kích hoạt (nếu bit RCIE được set).
8. Đọc giá trị thanh ghi RCSTA để đọc bit dữ liệu thứ 9 và kiểm tra xem quá trình nhận dữ liệu có bị lỗi không.
9. Đọc 8 bit dữ liệu từ thanh ghi RCREG.
10. Nếu quá trình truyền nhận có lỗi xảy ra, xóa lỗi bằng cách xóa bit CREN.
11. Nếu sử dụng ngắt nhận cần set bit GIE và PEIE (thanh ghi INTCON).

Các thanh ghi liên quan đến quá trình nhận dữ liệu bằng giao diện USART đồng bộ Master mode:

- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): chứa các bit cho phép toàn bộ các ngắt (bit GIER và PEIE).
- ✓ Thanh ghi PIR1 (địa chỉ 0Ch): chứa cờ hiệu RCIE.
- ✓ Thanh ghi PIE1 (địa chỉ 8Ch): chứa bit cho phép ngắt RCIE.
- ✓ Thanh ghi RCSTA (địa chỉ 18h): xác định các trạng thái trong quá trình nhận dữ liệu.
- ✓ Thanh ghi RCREG (địa chỉ 1Ah): chứa dữ liệu nhận được.
- ✓ Thanh ghi TXSTA (địa chỉ 98h): chứa các bit điều khiển SYNC và BRGH.
- ✓ Thanh ghi SPBRG (địa chỉ 99h): điều khiển tốc độ baud.

Chi tiết về thanh ghi xem lại phần USART trên.

#### 4.8.2.3. Truyền dữ liệu qua chuẩn giao tiếp USART đồng bộ MASTER MODE

Quá trình này không có sự khác biệt so với Master mode khi vi điều khiển hoạt động ở chế độ bình thường. Tuy nhiên khi vi điều khiển đang ở trạng thái sleep, sự khác biệt được thể hiện rõ ràng. Nếu có hai word dữ liệu được đưa vào thanh ghi TXREG trước khi lệnh sleep được thực thi thì quá trình sau sẽ xảy ra:

1. Word dữ liệu đầu tiên sẽ ngay lập tức được đưa vào thanh ghi TSR để truyền đi.
2. Word dữ liệu thứ hai vẫn nằm trong thanh ghi TXREG.
3. Cờ hiệu TXIF sẽ không được set.
4. Sau khi word dữ liệu đầu tiên đã dịch ra khỏi thanh ghi TSR, thanh ghi TXREG tiếp tục truyền word thứ hai vào thanh ghi TSR và cờ hiệu TXIF được set.

5. Nếu ngắt truyền được cho phép hoạt động, ngắt này sẽ đánh thức vi điều khiển và nếu toàn bộ các ngắt được cho phép hoạt động, bộ đếm chương trình sẽ chỉ tới địa chỉ chia chương trình ngắt (0004h).

Các bước cần tiến hành khi truyền dữ liệu bằng giao diện USART đồng bộ Slave mode:

1. Set bit SYNC, SPEN và clear bit CSRC.
2. Clear bit CREN và SREN.
3. Nếu cần sử dụng ngắt, set bit TXIE.
4. Nếu định dạng dữ liệu là 9 bit, set bit TX9.
5. Set bit TXEN.
6. Đưa bit dữ liệu thứ 9 vào bit TX9D trước (nếu định dạng dữ liệu là 9 bit).
7. Đưa 8 bit dữ liệu vào thanh ghi TXREG.
8. Nếu ngắt truyền được sử dụng, set bit GIE và PEIE (thanh ghi INTCON).

Các thanh ghi liên quan đến quá trình truyền dữ liệu bằng giao diện USART đồng bộ Slave mode:

- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): cho phép tắt cả các ngắt.
- ✓ Thanh ghi PIR1 (địa chỉ 0Ch): chứa cờ hiệu TXIF.
- ✓ Thanh ghi PIE1 (địa chỉ 8Ch): chứa bit cho phép ngắt truyền TXIE.
- ✓ Thanh ghi RCSTA (địa chỉ 18h): chứa bit cho phép cổng truyền dữ liệu (hai pin RC6/TX/CK và RC7/RX/DT).
- ✓ Thanh ghi TXREG (địa chỉ 19h): thanh ghi chứa dữ liệu cần truyền.
- ✓ Thanh ghi TXSTA (địa chỉ 98h): xác lập các thông số cho giao diện.
- ✓ Thanh ghi SPBRG (địa chỉ 99h): quyết định tốc độ baud.

Chi tiết về các thanh ghi đã được trình bày các phần trên.

#### 4.8.2.4. Nhận dữ liệu qua chuẩn giao tiếp USART đồng bộ MASTER MODE

Sự khác biệt của Slave mode so với Master mode chỉ thể hiện rõ ràng khi vi điều khiển hoạt động ở chế độ sleep. Ngoài ra chế độ Slave mode không quan tâm tới bit SREN. Khi bit CREN (cho phép nhận chuỗi dữ liệu) được set trước khi lệnh sleep được thực thi, 1 word dữ liệu vẫn được tiếp tục nhận, sau khi nhận xong bit thanh ghi RSR sẽ chuyển dữ liệu vào thanh ghi RCREG và bit RCIF được set. Nếu bit RCIE (cho phép ngắt nhận) đã được set trước đó, ngắt sẽ được thực thi và vi điều khiển được “đánh thức”, bộ đếm chương trình sẽ chỉ đến địa chỉ 0004h và chương trình ngắt sẽ được thực thi.

Các bước cần tiến hành khi nhận dữ liệu bằng giao diện USART đồng bộ Slave mode:

1. Cho phép cổng giao tiếp USART bắt đồng bộ (set bit SYNC, SPEN clear bit CSRC).
2. Nếu cần sử dụng ngắt nhận dữ liệu, set bit RCIE.
3. Nếu dữ liệu truyền nhận có định dạng là 9 bit, set bit RX9.
4. Set bit CREN để cho phép quá trình nhận dữ liệu bắt đầu.
5. Sau khi dữ liệu được nhận, bit RCIF sẽ được set và ngắt được kích hoạt (nếu bit RCIE được set).
6. Đọc giá trị thanh ghi RCSTA để đọc bit dữ liệu thứ 9 và kiểm tra xem quá trình nhận dữ liệu có bị lỗi không.
7. Đọc 8 bit dữ liệu từ thanh ghi RCREG.
8. Nếu quá trình truyền nhận có lỗi xảy ra, xóa lỗi bằng cách xóa bit CREN.
9. Nếu sử dụng ngắt nhận cần set bit GIE và PEIE (thanh ghi INTCON).

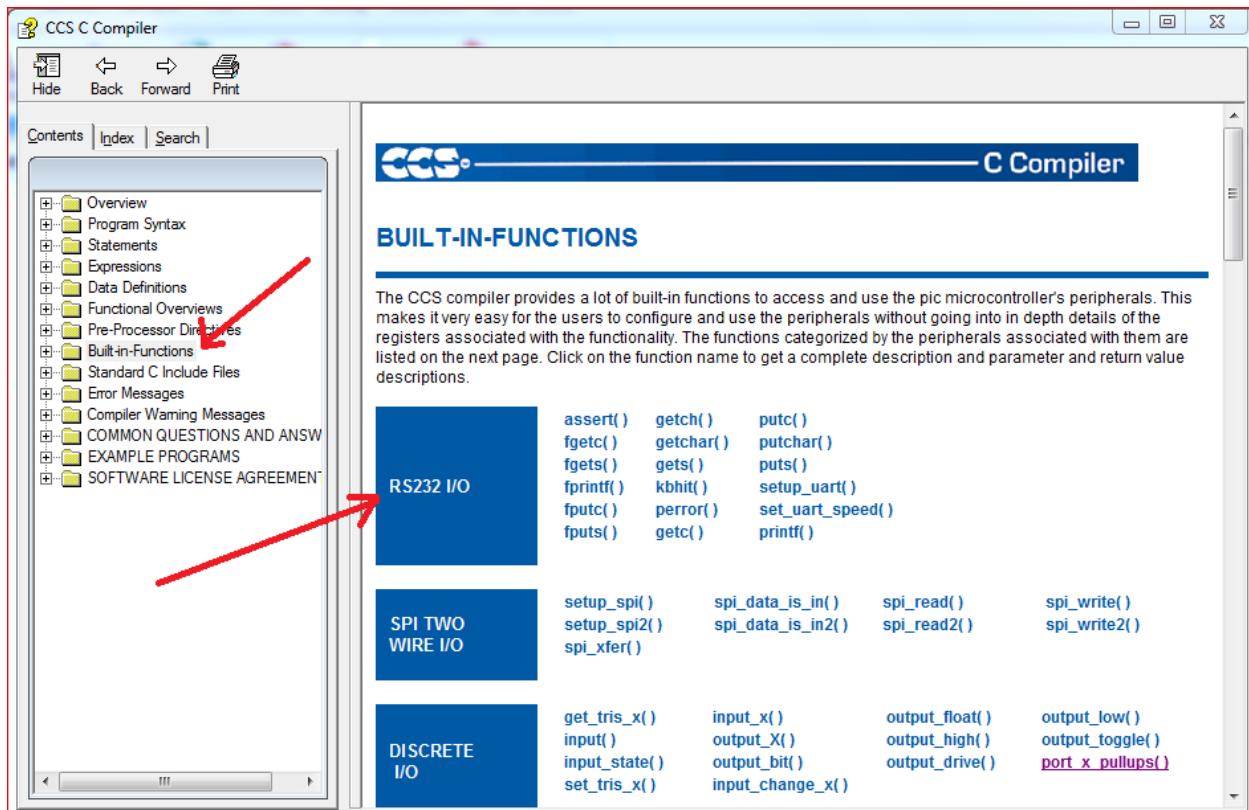
Các thanh ghi liên quan đến quá trình nhận dữ liệu bằng giao diện USART đồng bộ Slavemode:

- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): chứa các bit cho phép toàn bo các ngắt (bit GIER và PEIE).
- ✓ Thanh ghi PIR1 (địa chỉ 0Ch): chứa cờ hiệu RCIE.
- ✓ Thanh ghi PIE1 (địa chỉ 8Ch): chứa bit cho phép ngắt RCIE.
- ✓ Thanh ghi RCSTA (địa chỉ 18h): xác định các trạng thái trong quá trình nhận dữ liệu.
- ✓ Thanh ghi RCREG (địa chỉ 1Ah): chứa dữ liệu nhận được.
- ✓ Thanh ghi TXSTA (địa chỉ 98h): chứa các bit điều khiển SYNC và BRGH.
- ✓ Thanh ghi SPBRG (địa chỉ 99h): điều khiển tốc độ baud.

Chi tiết về các thanh ghi đã được trình bày các phần trên.

#### 4.8.3. Hàm trong CCS.

Để xem các hàm CCS hỗ trợ dung Timer chúng ta vào CCS C Compiler Help => Built-in-Funtions => RS232 I/O.



Hình 4.25. Các hàm USART trong CCS

➤ Để sử dụng được giao thức này phải có 2 khai báo:

#use delay (clock = 20000000) // nếu VDK đang dùng OSC 20Mhz

#use rs232 (options)

options được tách nhau bởi dấu phẩy, và có thể là:

- BAUD=x// Đặt baud rate
- parity=N// Đặt parity
- XMIT=pin// Đặt chân truyền
- RCV=pin// Đặt chân nhận
- Bits=x// Số bit

**Ví dụ:** #use rs232 (baud=9600,parity=N,xmit=PIN\_C6,rcv=PIN\_C7,bits=8) //dung tốc độ baud=9600, không chẵn lẻ, chân truyền C6, chân nhận C7.

➤ Hàm hỗ trợ của CCS: Xem trong 16f877a.h mục #use rs232() Prototypes:

- ✓ value = getc()
- ✓ value = fgetc(stream)
- ✓ value=getch()
- ✓ value=getchar()

=> Các hàm trên đều trả về giá trị 8 bit character. Hàm này có tác dụng nhận về 1 kí tự từ chân nhận. Đối với hàm value = fgetc(stream) thì stream là biến của USART (Mềm). Thường dùng trong trường hợp cần giao tiếp nhiều USART.

**Ví dụ:**

```
#use rs232(baud=9600,xmit=pin_c6,rcv=pin_c7,stream=HOSTPC)
#use rs232(baud=1200,xmit=pin_b1, rcv=pin_b0,stream=GPS)
#use rs232(baud=9600,xmit=pin_b3, stream=DEBUG)
```

```

...
while(TRUE) {
    c=fgetc(GPS);
    fputc(c,HOSTPC);
    if(c==13)
        fprintf(DEBUG,"Got a CR\r\n");
}

```

- ✓ putc (cdata)
- ✓ putchar (cdata)
- ✓ fputc(cdata, stream)

=> Các hàm này có tác dụng gửi 1 ký tự qua chân truyền. Hàm fputc(cdata, stream) với cdata là ký tự cần truyền, stream là địa chỉ USART truyền.

- ✓ gets (string)
- ✓ value = fgets (string, stream)

=> Nó có tác dụng nhận 1 xâu từ chân nhận.

Ví dụ:

```

char string[30];
printf("Password: ");
gets(string);
if(strcmp(string, password));
    printf("OK");

```

- ✓ puts (string)
- ✓ value = fputs (string, stream)

=> Nó có tác dụng đặt 1 xâu lên chân truyền

```

Ví dụ: puts( " ----- " );
puts( " | HI | " );
puts( " ----- " );

```

- ✓ printf (string)
- ✓ printf (cstring, values...)
- ✓ printf (fname, cstring, values...)
- ✓ fprintf (stream, cstring, values...)

=> Tương tự như hàm puts(string); Nhưng hàm này chủ yếu xuất chuỗi theo chuẩn RS232 ra PC  
+ string là 1 chuỗi hằng hay 1 mảng ký tự (kết thúc bởi ký tự null ).

+ value là danh sách các biến, cách nhau bởi dấu phẩy.

+ Bạn phải khai báo dạng format của value theo kiểu %wt. Trong đó w có thể có hoặc không, có giá trị từ 1-9 chỉ rõ có bao nhiêu ký tự được xuất ra (mặc định không có thì có bao nhiêu ra bấy nhiêu), hoặc 01-09 sẽ chèn thêm 0 cho đủ ký tự hoặc 1.1-1.9 cho trường hợp số thực. còn t là kiểu giá trị, t có thể là :

- C: 1 ký tự
- S: chuỗi hoặc ký tự
- U: số 8 bit không dấu
- x: số 8 bit kiểu hex ( ký tự viết thường ,VD : 1ef )
- X: số 8 bit kiểu hex (ký tự viết hoa, VD : 1EF )
- D: số 8 bit có dấu

e: số thực có luỹ thừa VD : e12

f: số thực

Lx: số hex 16 /32 bit (ký tự viết thường)

LX: hex 16 /32 bit (ký tự viết hoa)

Lu: số thập phân không dấu

Ld: số thập phân có dấu

%: ký hiệu %

Example formats:

Specifier	Value=0x12	Value=0xfe
%03u	018	254
%u	18	254
%2u	18	*
%5	18	254
%d	18	-2
%x	12	fe
%X	12	FE
%4X	0012	00FE
%3.1w	1.8	25.4

\* Result is undefined - Assume garbage.

### Ví dụ 1:

```
Int k =6;
Printf( " hello " );
Printf( "%u" , k );
```

### Ví dụ 2:

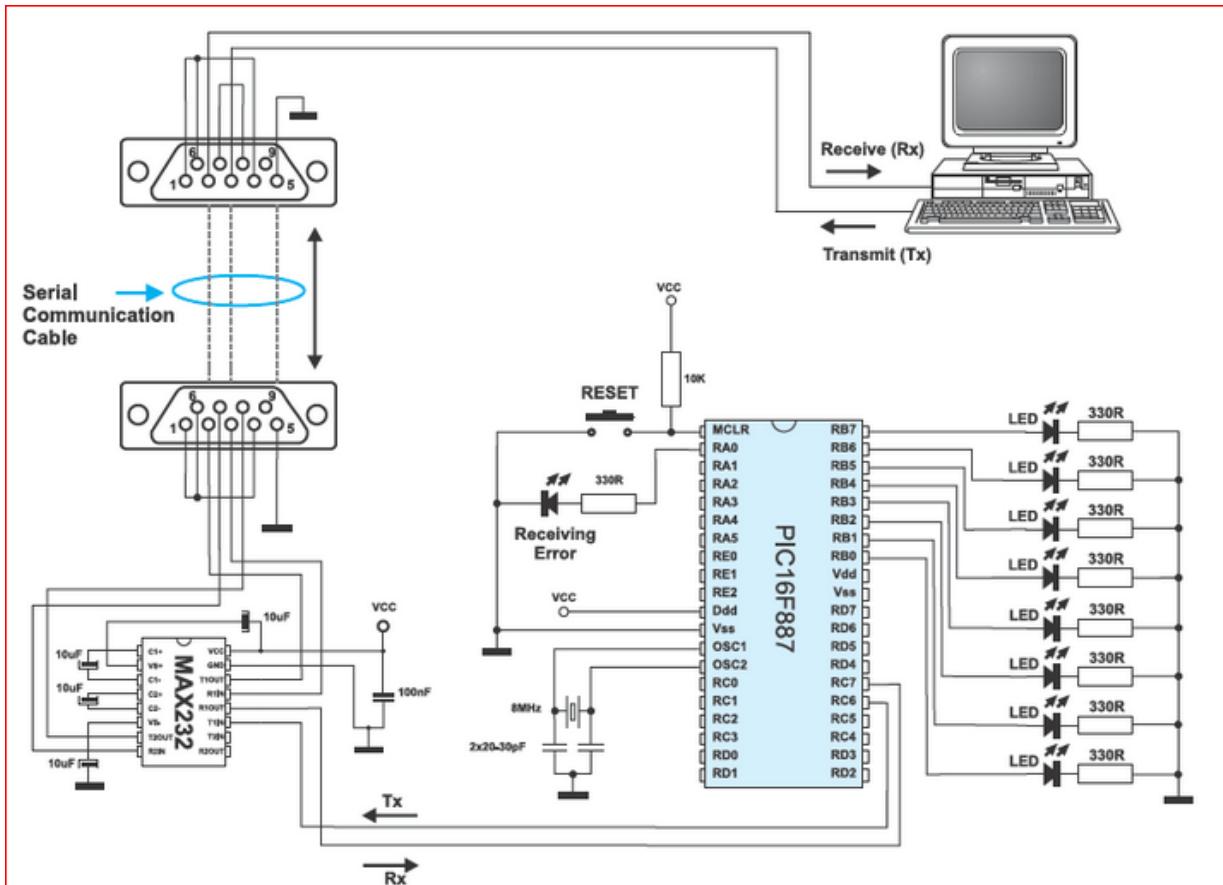
```
byte x,y,z;
printf("HiThere");
printf("RTCCValue=>%2x\n\r",get_rtcc());
printf("%2u %X %4X\n\r",x,y,z);
printf(LCD_PUTC, "n=%u",n);
✓ value = kbhit()
✓ value = kbhit (stream)
```

=> trả về TRUE (hay 1) nếu 1 ký tự đã được nhận (trong bộ đệm phàn cứng) và sẵn sàng cho việc đọc, và trả về FALSE (hay 0) nếu chưa sẵn sàng . Hàm này có thể dùng hỏi vòng xem khi nào có data nhận từ RS232 để đọc.

- ✓ setup\_uart(baud, stream)
- ✓ setup\_uart(baud)
- ✓ set\_uart\_speed (baud, [stream, clock])

- ✓ perror(string); //đòi hỏi khai báo them #include<errno.h>
- ✓ Assert (); //đòi hỏi khai báo them #include<assert.h>

#### 4.8.4. Sơ đồ mạch điện gió tiếp USART với PC thông qua cổng com.

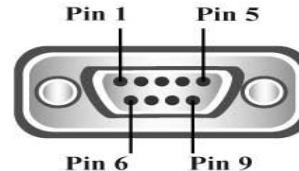


Hình 4.26. Sơ đồ gió tiếp USART với PC

## RS232

<b>Pin 1</b>	DCD
<b>Pin 2</b>	RXD
<b>Pin 3</b>	TXD
<b>Pin 4</b>	DTR
<b>Pin 5</b>	GND
<b>Pin 6</b>	DSR
<b>Pin 7</b>	RTS
<b>Pin 8</b>	CTS
<b>Pin 9</b>	RI

RS232 Pinout (9 Pin Male)



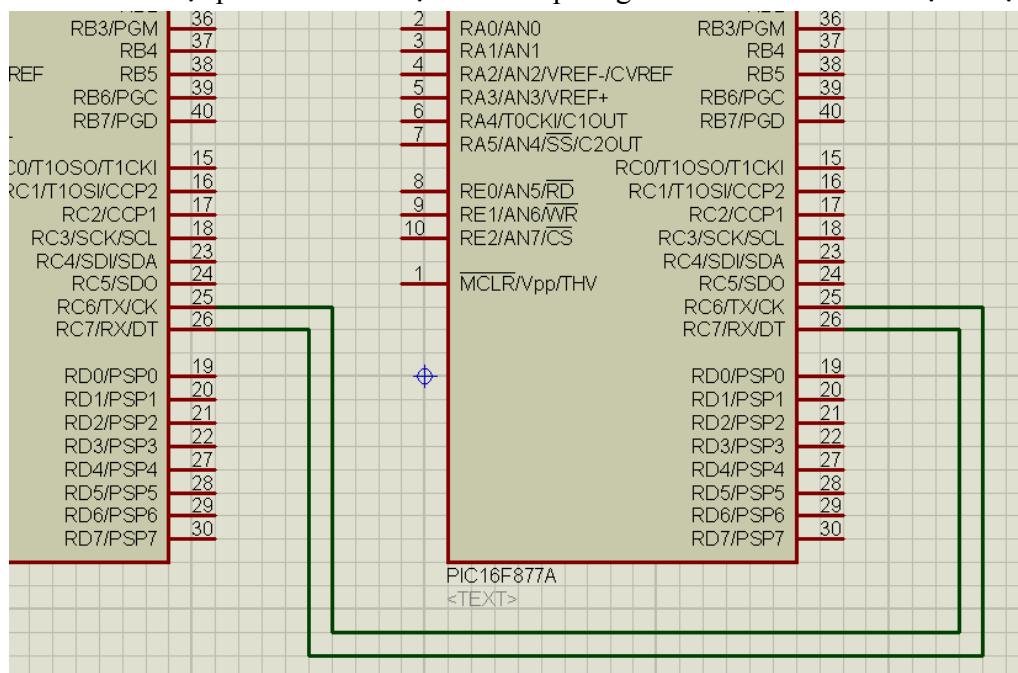
Hình 4.27. Sơ đồ chân cổng Com

- ✓ Muốn kết nối PIC với PC thì yêu cầu trên PC phải có cổng kết nối RS232 (Com). Nhưng hiện nay các máy tính hầu như đã bỏ cổng Com này rồi nên không thể kết nối trực tiếp bằng cách này được. Đối với laptop càng khó khăn.
- ✓ Để giải quyết vấn đề này các hang điện tử lớn đã phát triển ra những IC chuyên dụng nhằm chuyển đổi cổng usb sang cổng Com như: MAX3421EE, PL-2303TA, PL-2303HXD, FT232BL....và 1 số hang cũng đã sản xuất ra 1 số cable chuyên dụng cho việc này. Xem hình dưới:



Hình 4.28.Cable chuyển đổi usb to com - Hình 4.29. Mạch chuyển đổi usb to com dung FT232RL

- ✓ Trên sơ đồ mạch ta còn thấy MAX232. Đây là ic dùng để cách ly điện áp từ cổng com máy tính với vi điều khiển. nếu ta dùng giao tiếp qua usb thì không cần dùng đến thiết bị này.
- ✓ Để xem quá trình giao tiếp PIC với PC ta có thể dùng rất nhiều phần mềm có sẵn trên mạng hay chính các bạn tự thiết kế. Một trong những phần mềm để kiểm tra thông dụng nhất hiện nay là “Terminal” với giao diện rất dễ nhìn và đa chức năng, gọn nhẹ.
- ✓ Để mô phỏng quá trình giao tiếp trên máy tính, các bạn có thể dùng phần mềm “Virtual.Serial.Port.Driver.6.9.1.134” để tạo cổng com ảo trên máy tính và phần mềm “Proteus” là một phần mềm rất mạnh để mô phỏng vi điều khiển và các mạch điện tử.



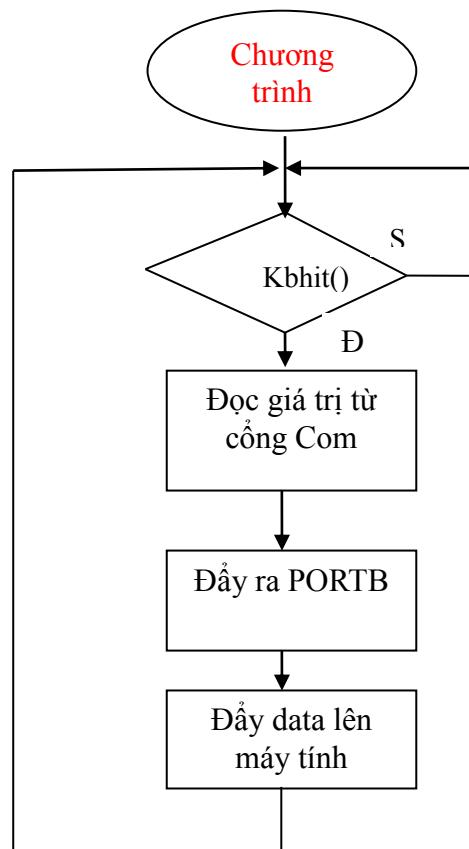
4.30. Sơ đồ giao tiếp giữa 2 vi điều khiển.

- ✓ Khi giao tiếp USART giữa 2 vi điều khiển ta chỉ cần kết nối chéo nhau giữa 2 chân TXD và RXD. Khi đó TXD của vi điều khiển này sẽ kết nối với RXD của con kia và ngược lại.

#### 4.8.5. Bài tập USART:

Viết chương trình nhận ký tự truyền từ máy tính, sau đó hiển thị giá trị đó lên PortB và gửi ngược lại máy tính:

➤ Lưu đồ thuật toán:



➤ Chương trình:

```

*****
*
* PIC Training Course
* Nguyen Tat Thanh University
*
*****/




*****
*
* Module      : main.c
* Description  : Receive character from PC, display Portb and transmit to PC.
* Tool        : ccs v5.00xx
* Chip        : 16F877A
* History     : 7/2011
* Version     : v1.0
* Author      : Nguyen Huu Luan
* Mail        : nvn.nhl@gmail.com
* Website     : ntt.edu.vn
* Notes       :
*****
/////////////////
// Su dung portB output
// Su dung USART1 Truyền C6, Nhận C7
// 
```

```
// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh //  
//  
///////////////////////////////
```

```
/*  
* Keywords Proteus  
* Pic 16f877a.  
* Led: Led-green, Led-red.....  
* Dien tro: resistors.  
* COMPIM : De lay cong Com.  
*/
```

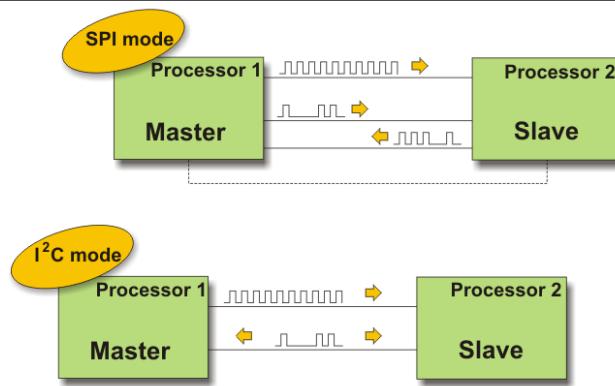
```
*****
```

```
#include <16f877a.h>  
#include <def_877a.h>  
#device *=16 ADC=8  
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT,  
NOLVP, NOCPD, NOWRT  
#use delay(clock=20000000)  
#use rs232(baud=9600,parity=N,xmit=PIN_C6,rcv=PIN_C7,bits=8)  
char h_thi;  
void main() {  
    trisb=0x00;//dau ra  
    portb=0x00;//ban dau led tat  
    while(true){  
        if(kbhit())  
        {  
            h_thi=getc();  
            portd=h_thi;  
            putc(h_thi);  
        };  
    }  
}
```

#### 4.9. MSSP

MSSP (Master Synchronous Serial Port) là giao diện đồng bộ nối tiếp dùng để giao tiếp với các thiết bị ngoại vi (EEPROM, ghi dịch, chuyển đổi ADC,...) hay các vi điều khiển khác. MSSP có thể hoạt động dưới hai dạng giao tiếp:

- o SPI (Serial Peripheral Interface).
- o I2C (Inter-Integrated Circuit).



Hình 4.31. Sơ đồ SPI mode và I2C mode

Các thanh ghi điều khiển giao chuẩn giao tiếp này bao gồm thanh ghi trạng thái SSPSTAT và hai thanh ghi điều khiển SSPCON1 và SSPCON2. Tùy theo chuẩn giao tiếp được sử dụng (SPI hay I2C) mà dùng chức năng các thanh ghi

#### **Thanh ghi SSPSTAT: (địa chỉ 94h)**

Thanh ghi chứa các bit trạng thái của chuẩn giao tiếp MSSP.

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/ $\bar{A}$	P	S	R/ $\bar{W}$	UA	BF
bit 7							bit 0

Bit 7 SMP Sample bit

SPI Master mode:

SMP = 1 dữ liệu được lấy mẫu (xác định trạng thái logic) tại thời điểm cuối xung clock.

SMP = 0 dữ liệu được lấy mẫu tại thời điểm giữa xung clock.

SPI Slave mode: bit này phải được xóa về 0.

#### Bit 6 CKE SPI Clock Select bit

CKE = 1 SPI Master truyền dữ liệu khi xung clock chuyển từ trạng thái tích cực đến trạng thái chờ.

CKE = 0 SPI Master truyền dữ liệu khi xung clock chuyển từ trạng thái chờ đến trạng thái tích cực.

(trạng thái chờ được xác định bởi bit CKP (SSPCON<4>).

Bit 5 **D/A**: Data/Address bit

Bit này chỉ có tác dụng ở chế độ I2C mode

Bit 4 P Stop bit

Bit này chỉ sử dụng khi MSSP ở chế độ I2C.

Bit 3 S Start bit

Bit này chỉ có tác dụng khi MSSP ở chế độ I<sub>2</sub>C.

Bit 2 R/W: Read/Write bit information

Bit này chỉ có tác dụng khi MSSP ở chế độ I<sub>2</sub>C.

Bit 1 UA Update Address bit

Bit này chỉ có tác dụng khi MSSP ở chế độ I<sub>2</sub>C.

Bit 0 BF Buffer Status bit

BF ≡ 1 thanh ghi đếm SSPBUE đã có dữ liệu

BE ≡ 0 thanh ghi đệm SSPBUF chưa có dữ liệu

**Thanh ghi SSPCON1 (địa chỉ 14h):**

Thanh ghi điều khiển chuẩn giao tiếp MSSP.

| R/W-0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| WCOL  | SSPOV | SSPEN | CKP   | SSPM3 | SSPM2 | SSPM1 | SSPM0 |
| bit 7 |       |       |       | bit 0 |       |       |       |

Bit 7 WCOL Write Collision Detect bit

WCOL = 1 dữ liệu mới được đưa vào thanh ghi SSPBUF trong khi chưa truyền xong dữ liệu trước đó.

WCOL = 0 không có hiện tượng xảy ra.

Bit 6 SSPOV Receive Overflow Indic平 bit (bit này chỉ có tác dụng ở chế độ SPI Slave mode).

SSPOV = 1 dữ liệu trong bufer đệm (thanh ghi SSPBUF) bị tràn (dữ liệu cũ chưa được đọc thì có dữ liệu mới gi đè lên).

SSPOV = 0 không có hiện tượng xảy ra.

Bit 5 SSPEN Synchronous Serial Port Enable bit

SSPEN = 1 cho phép công giao tiếp MSSP (các pin SCK, SDO, SDI và ).

SSPEN = 0 không cho phép công giao tiếp MSSP.

Bit 4 CKP Clock Polarity Select bit

CKP = 1 trạng thái chờ của xung clock là mức logic cao.

CKP = 0 trạng thái chờ của xung clock là mức logic thấp.

Bit 3-0 SSPM3:SSPM0 Synchronous Serial Mode Select bit

Các bit này đóng vai trò lựa chọn các chế độ hoạt động của MSSP.

- 0101 Slave mode, xung clock lấy từ pin SCK,  $\overline{SS}$  không cho phép pin điều khiển ( $\overline{SS}$  là pin I/O bình thường).
- 0100 SPI Slave mode, xung clock lấy từ pin SCK,  $\overline{SS}$  cho phép pin điều khiển .
- 0011 SPI Master mode, xung clock bằng (ngõ ra TMR2)/2.
- 0010 SPI Master mode, xung clock bằng (FOSC/64).
- 0001 SPI Master mode, xung clock bằng (FOSC/16).
- 0000 SPI Master mode, xung clock bằng (FOSC/4).

Các trạng thái không được liệt kê hoặc không có tác dụng điều khiển hoặc chỉ có tác dụng đổi với chế độ I2C mode.

**Khi MSSP ở chế độ I2C**

Bit 7 WCOL Write Collision Detect bit

Khi truyền dữ liệu ở chế độ I2C Master mode:

WCOL = 1 đưa dữ liệu truyền đi vào thanh ghi SSPBUF trong khi chế độ truyền dữ liệu của I2C chưa sẵn sàng.

WCOL = 0 không xảy ra hiện tượng trên.

khi truyền dữ liệu ở chế độ I2C Slave mode:

WCOL = 1 dữ liệu mới được đưa vào thanh ghi SSPBUF trong khi dữ liệu cũ chưa được truyền đi.

WCOL = 0 không có hiện tượng xảy ra.

**Ở chế độ nhận dữ liệu (Master hoặc Slave):**

Bit này không có tác dụng chỉ thi các trạng thái.

**Bit 6 SSPOV Receive Overflow Indicator Flag bit.**

Khi nhận dữ liệu:

SSPOV = 1 dữ liệu mới được nhận vào thanh ghi SSPBUF trong khi dữ liệu cũ chưa được đọc.

SSPOV = 0 không có hiện tượng trên xảy ra.

Khi truyền dữ liệu:

Bit này không có tác dụng chỉ thị các trạng thái.

**Bit 5 SSPEN Synchronous Serial Port Enable bit**

SSPEN = 1 cho phép công giao tiếp MSSP (các pin SDA và SCL).

SSPEN = 0 không cho phép công giao tiếp MSSP.

Cần chú ý là các pin SDA và SCL phải được điều khiển trạng thái bằng các bit tương ứng trong thanh ghi TRISC trước đó).

**Bit 4 CKP SCK Release Control bit**

Ở chế độ Slave mode:

CKP = 1 cho xung clock tác động.

CKP = 0 giữ xung clock ở mức logic thấp (để bảo đảm thời gian thiết lập dữ liệu).

**Bit 3,0 SSPM3:SSPM0**

Các bit này đóng vai trò lựa chọn các chế độ hoạt động của MSSP.

- 1111 I2C Slave mode 10 bit địa chỉ và cho phép ngắt khi phát hiện bit Start và bit Stop.
- 1110 I2C Slave mode 7 bit địa chỉ và cho phép ngắt khi phát hiện bit Start và bit Stop.
- 1011 Firmware Controlled Master mode (không cho phép chế độ Slave).
- 1000 I2C Master mode, xung clock = FOSC/(4\*(SSPADD+1)).
- 0111 I2C Slave mode 10 bit địa chỉ. Các trạng thái không được liệt kê hoặc không có tác dụng điều khiển hoặc chỉ có tác dụng đối với chế độ SPI mode.

#### **Thanh ghi SSPCON2 (địa chỉ 91h):**

Thanh ghi điều khiển các chế độ hoạt động của chuẩn giao tiếp I2C xem ở phần I2C Mode.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN

bit 7

bit 0

**Bit 7 GCEN General Call Enable bit**

GCEN = 1 Cho phép ngắt khi địa chỉ 0000h được nhận vào thanh ghi SSPSR (địa chỉ của chế độ General Call Address).

GCEN = 0 Không cho phép chế độ địa chỉ trên.

**Bit 6 ACKSTAT Acknowledge Status bit** (bit này chỉ có tác dụng khi truyền dữ liệu ở chế độ I2C Master mode).

ACKSTAT = 1 nhận được xung từ I2C Slave.

ACKSTAT = 0 chưa nhận được xung .

**Bit 5 ACKDT Acknowledge Data bit** (bit này chỉ có tác dụng khi nhận dữ liệu ở chế độ I2C Master mode).

ACKDT = 1 chưa nhận được xung .

ACKDT = 0 Đã nhận được xung .

Bit 4 ACKEN Acknowledge Sequence Enable bit (bit này chỉ có tác dụng khi nhận dữ liệu ở chế độ I2C Master mode)

ACKEN = 1 cho phép xung xuất hiện ở 2 pin SDA và SCL khi kết thúc quá trình nhận dữ liệu.

ACKEN = 0 không cho phép tác động trên.

Bit 3 RCEN Receive Enable bit (bit này chỉ có tác dụng ở chế độ I2C Master mode).

RCEN = 1 Cho phép nhận dữ liệu ở chế độ I2C Master mode.

RCEN = 0 Không cho phép nhận dữ liệu.

Bit 2 PEN Stop Condition Enable bit

PEN = 1 cho phép thiết lập điều kiện Stop ở 2 pin SDA và SCL.

PEN = 0 không cho phép tác động trên.

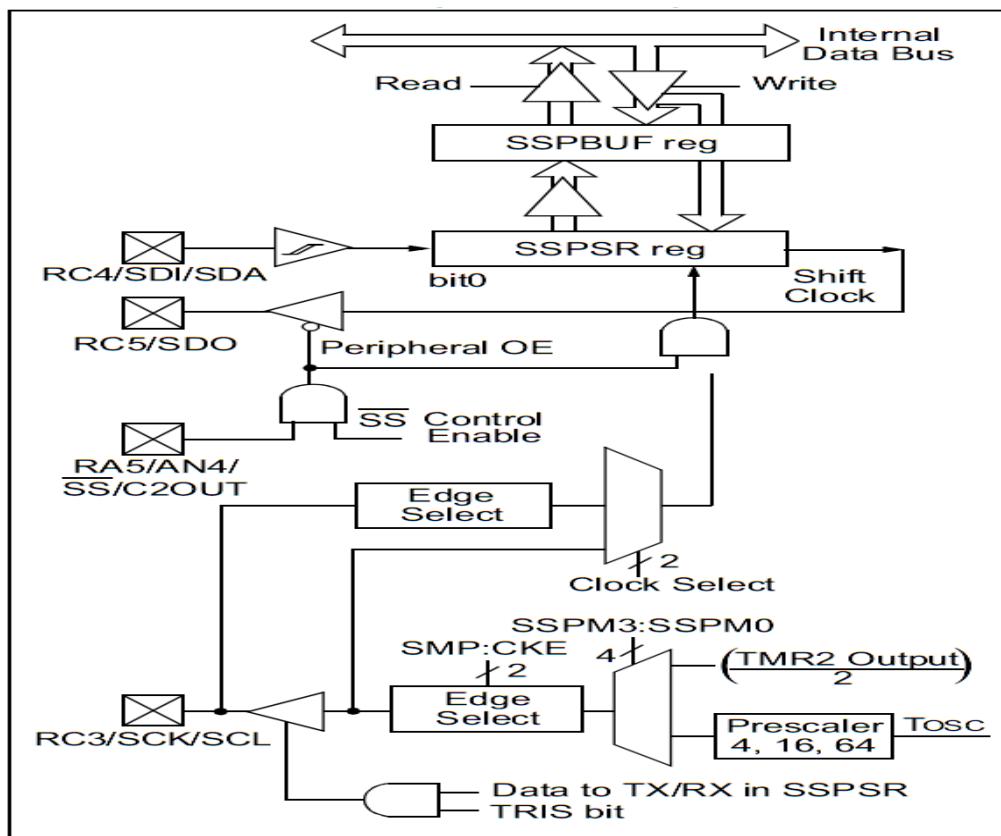
Bit 1 RSEN Repeated Start Condition Enable bit

RSEN = 1 cho phép thiết lập điều kiện Start lặp lại liên tục ở 2 pin SDA và SCL.

#### 4.9.1. SPI MODE

Chuẩn giao tiếp SPI cho phép truyền nhận đồng bộ. Ta cần sử dụng 4 pin cho chuẩn giao tiếp này:

- RC5/SDO: ngõ ra dữ liệu dạng nối tiếp (Serial Data output).
- RC4/SDI/SDA: ngõ vào dữ liệu dạng nối tiếp (Serial Data Input). Hình 5.10.1. Sơ đồ khối MSSP (giao diện SPI)
- RC3/SCK/SCL: xung đồng bộ nối tiếp (Serial Clock).
- RA5/AN4/SS/C2OUT: chọn đối tượng giao tiếp (Serial Select) khi giao tiếp ở chế độ Slave mode.



Hình 4.32. Sơ đồ khái niệm khối MSSP (giao diện SPI)

Các thanh ghi liên quan đến MSSP khi hoạt động ở chuẩn giao tiếp SPI bao gồm:

- ✓ Thanh ghi điều khiển SSPCON, thanh ghi này cho phép đọc và ghi.
- ✓ Thanh ghi trạng thái SSPSTAT, thanh ghi này chỉ cho phép đọc và ghi ở 2 bit trên, 6 bit còn lại chỉ cho phép đọc.
- ✓ Thanh ghi đóng vai trò là buffer truyền nhận SSPBUF, dữ liệu truyền đi hoặc nhận được sẽ được đưa vào trang ghi này. SSPBUF không có cấu trúc đệm hai lớp (doubled-buffer), do đó dữ liệu ghi vào thanh ghi SSPBUF sẽ lập tức được ghi vào thanh ghi SSPSR.
- ✓ Thanh ghi dịch dữ liệu SSPSR dùng để dịch dữ liệu vào hoặc ra. Khi 1 byte dữ liệu được nhận hoàn chỉnh, dữ liệu sẽ từ thanh ghi SSPSR chuyển qua thanh ghi SSPBUF và cờ hiệu được set, đồng thời ngắt sẽ xảy ra.

Chi tiết về các thanh ghi đã được trình bày ở MSSP.

Khi sử dụng chuẩn giao tiếp SPI trước tiên ta cần thiết lập các chế độ cho giao diện bằng cách đưa các giá trị thích hợp vào hai thanh ghi SSPCON và SSPSTAT. Các thông số cần thiết lập bao gồm:

Master mode hay Slave mode. Đối với Master mode, xung clock đồng bộ sẽ đi ra từ chân RC3/SCK/SCL. Đối với Slave mode, xung clock đồng bộ sẽ được nhận từ bên ngoài qua chân RC3/SCK/SCL. Các chế độ của Slave mode.

Mức logic của xung clock khi ở trạng thái tạm ngưng quá trình truyền nhận (Idle).

Cạnh tác động của xung clock đồng bộ (cạnh lên hay cạnh xuống).

Tốc độ xung clock (khi hoạt động ở Master mode).

Thời điểm xác định mức logic của dữ liệu (ở giữa hay ở cuối thời gian 1 bit dữ liệu được đưa vào).

Master mode, Slave mode và các chế độ của Slave mode được điều khiển bởi các bit SSPM3:SSPM0 (SSPCON<3:0>). Xem chi tiết ở thanh ghi **SSPCON1** Mục MSSP.

MSSP bao gồm một thanh ghi dịch dữ liệu SSPSR và thanh ghi đệm dữ liệu SSPBUF. Hai thanh ghi này tạo thành bộ đệm dữ liệu kép (doubled-buffer). Dữ liệu sẽ được dịch vào hoặc ra qua thanh ghi SSPSR, bit MSB được dịch trước. Đây là một trong những điểm khác biệt giữa hai giao diện MSSP và USART (USART dịch bit LSB trước). Trong quá trình nhận dữ liệu, khi dữ liệu đưa vào từ chân RC4/SDI/SDA trong thanh ghi SSPSR đã sẵn sàng (đã nhận đủ 8 bit), dữ liệu sẽ được đưa vào thanh ghi SSPBUF, bit chỉ thị trạng thái bộ đệm BF (SSPSTAT<0>) sẽ được set để báo hiệu bộ đệm đã đầy, đồng thời cờ ngắt SSPIF (PIR1<3>) cũng được set. Bit BF sẽ tự động reset về 0 khi dữ liệu trong thanh ghi SSPBUF được đọc vào. Bộ đệm kép cho phép đọc tiếp byte tiếp theo trước khi byte dữ liệu trước đó được đọc vào. Tuy nhiên ta nên đọc trước dữ liệu từ thanh ghi SSPBUF trước khi nhận byte dữ liệu tiếp theo.

Quá trình truyền dữ liệu cũng hoàn toàn tương tự nhưng ngược lại. Dữ liệu cần truyền sẽ được đưa vào thanh ghi SSPBUF đồng thời đưa vào thanh ghi SSPSR, khi đó cờ hiệu BF được set. Dữ liệu được dịch từ thanh ghi SSPSR và đưa ra ngoài qua chân RC5/SDO. Ngắt sẽ xảy ra khi quá trình dịch dữ liệu hoàn tất. Tuy nhiên dữ liệu trước khi được đưa ra ngoài phải được cho phép bởi tín hiệu từ chân . Chân này đóng vai trò chọn đối tượng giao tiếp khi SPI ở chế độ Slave mode.

Khi quá trình truyền nhận dữ liệu đang diễn ra, ta không được phép ghi dữ liệu vào thanh ghi SSPBUF. Thao tác ghi dữ liệu này sẽ set bit WCON (SSPCON<7>). Một điều cần chú ý nữa là thanh ghi SSPSR không cho phép truy xuất trực tiếp mà phải thông qua thanh ghi SSPBUF.

Công giao tiếp của giao diện SPI được điều khiển bởi bit SSPEN (SSPSON<5>). Bên

cạnh đó cần điều khiển chiều xuất nhập của PORTC thông qua thanh ghi TRISC sao cho phù hợp với chiều của giao diện SPI. Cụ thể như sau:

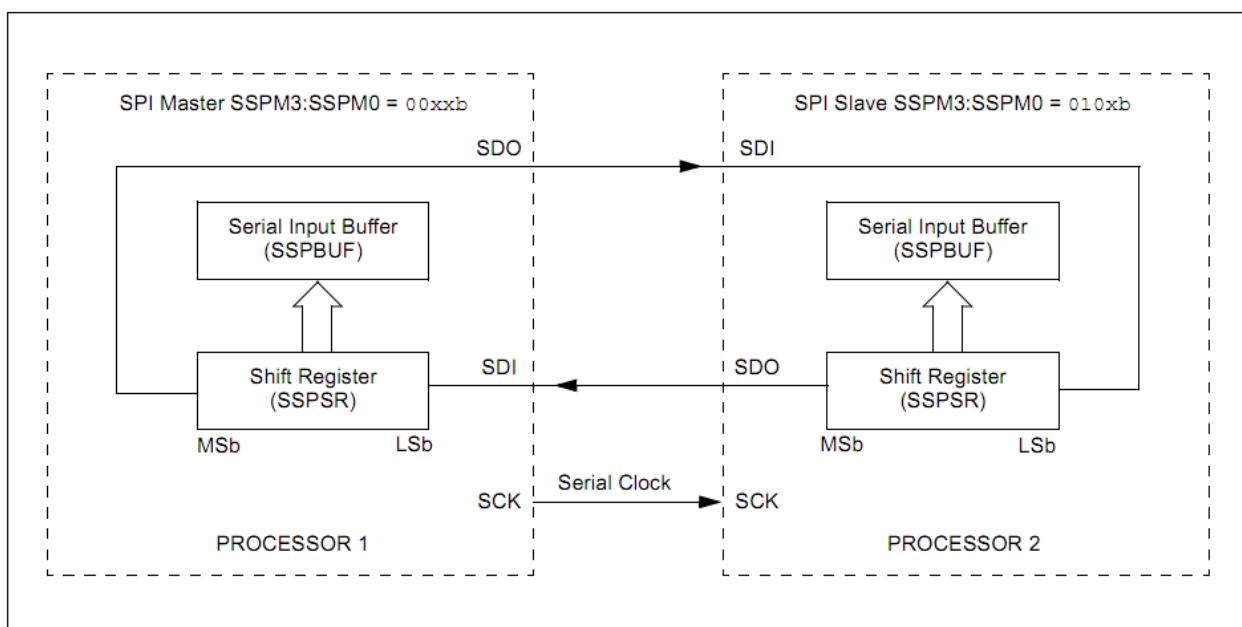
- RC4/SDI/SDA sẽ tự động được điều khiển bởi khối giao tiếp SPI.
- RS5/SDO là ngõ ra dữ liệu, do đó cần clear bit TRISC<5>.

Khi SPI ở dạng Master mode, cần clear bit TRISC<3> để cho phép đưa xung clock đồng bộ ra chân RC3/SCK/SCL.

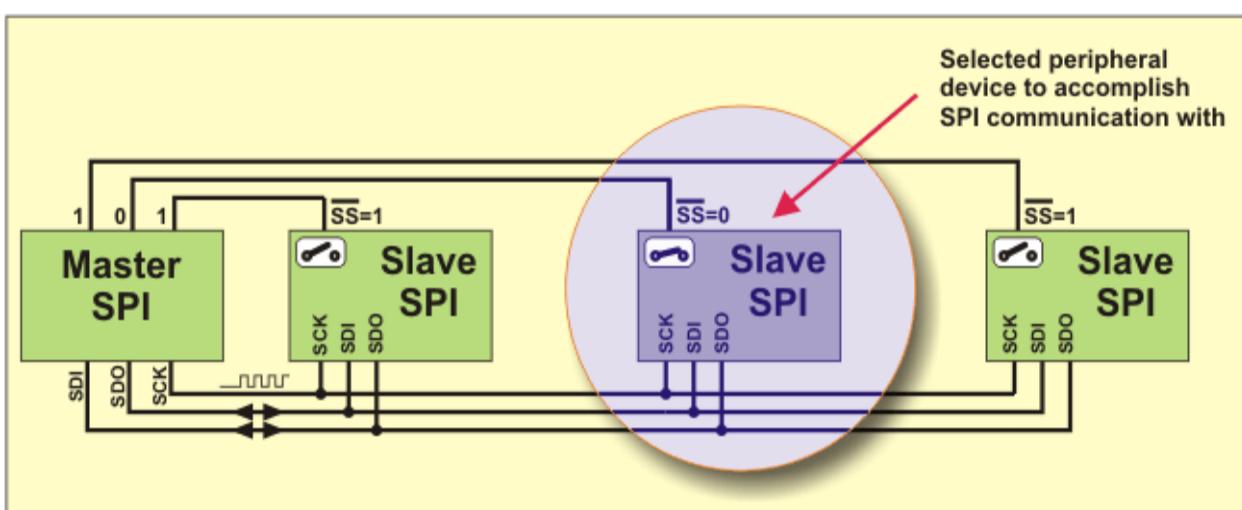
Khi SPI ở dạng Slave mode, cần set bit TRISC<3> để cho phép nhận xung clock đồng bộ từ bên ngoài qua chân RC3/SCK/SCL.

Set bit TRISC<4> để cho phép chân RA5/AN4/SS/C2OUT nhận tín hiệu điều khiển truy xuất dữ liệu khi SPI ở chế độ Slave mode.

Sơ đồ kết nối của chuẩn giao tiếp SPI như sau:



Hình 4.33. Sơ đồ kết nối của chuẩn giao tiếp SPI



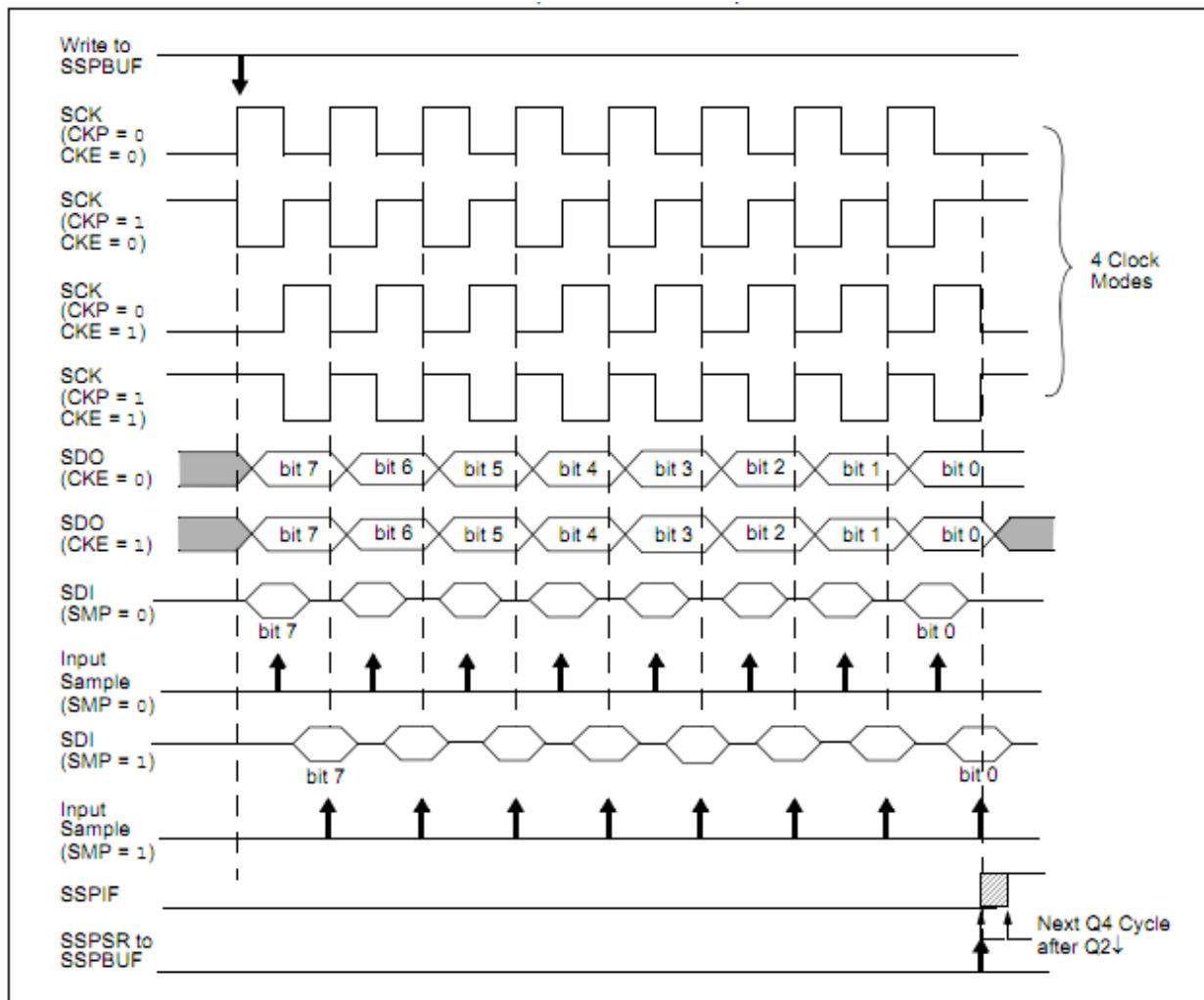
Hình 4.44. Sơ đồ kết nối nhiều Slave chuẩn giao tiếp SPI

Theo sơ đồ kết nối này, khôi Master sẽ bắt đầu quá trình truyền nhận dữ liệu bằng cách gửi tín hiệu xung đồng bộ SCK. Dữ liệu sẽ dịch từ cả hai thanh ghi SSPSR đưa ra ngoài nếu có một cạnh của xung đồng bộ tác động và ngưng dịch khi có tác động của cạnh còn lại. Cả hai khôi Master và Slave nên được ánh định chung các qui tắc tác động của xung clock đồng bộ để dữ liệu

có thể dịch chuyển đồng thời.

#### 4.9.1.1.SPI MASTER MODE:

Ở chế độ Master mode, vi điều khiển có quyền án định thời điểm trao đổi dữ liệu (và đổi tượng trao đổi dữ liệu nếu cần) vì nó điều khiển xung clock đồng bộ. Dữ liệu sẽ được truyền nhận ngay thời điểm dữ liệu được đưa vào thanh ghi SSPBUF. Nếu chỉ cần nhận dữ liệu, ta có thể án định chân SDO là ngõ vào (set bit TRISC<5>). Dữ liệu sẽ được dịch vào thanh ghi SSPSR theo một tốc độ được định sẵn cho xung clock đồng bộ. Sau khi nhận được một byte dữ liệu hoàn chỉnh, byte dữ liệu sẽ được đưa vào thanh ghi SSPBUF, bit BF được set và ngắt xảy ra. Khi lệnh SLEEP được thực thi trong quá trình truyền nhận, trạng thái của quá trình sẽ được giữ nguyên và tiếp tục sau khi vi điều khiển được đánh thức. Giản đồ xung của Master mode và các tác động của các bit điều khiển được trình bày trong hình vẽ sau:

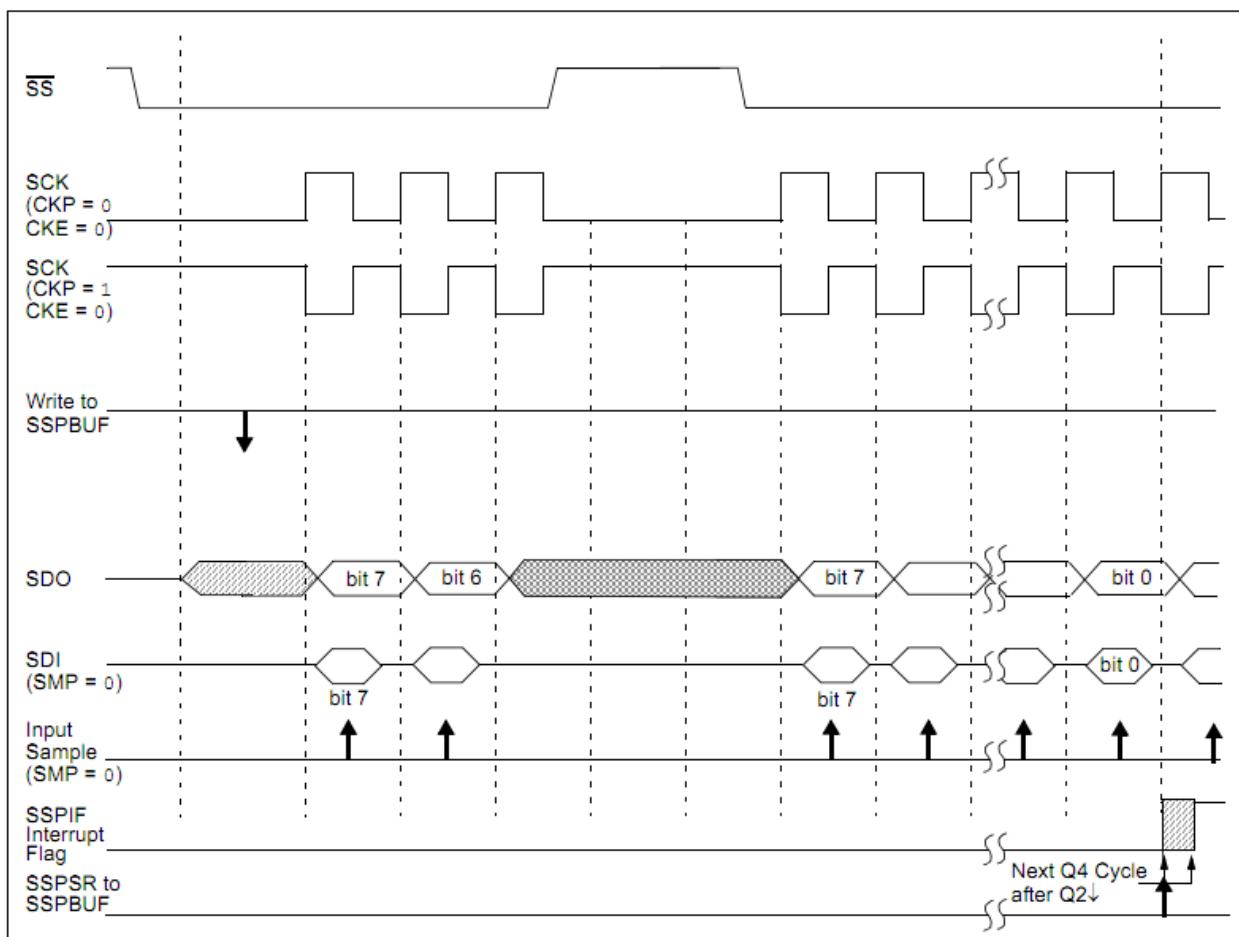


Hình 4.35. Giản đồ xung SPI ở chế độ Master

#### 4.9.1.2.SPI SLAVE MODE

Ở chế độ này SPI sẽ truyền và nhận dữ liệu khi có xung đồng bộ xuất hiện ở chân SCK. Khi truyền nhận xong bit dữ liệu cuối cùng, cờ ngắt SSPIF sẽ được set. Slave mode hoạt động ngay cả khi vi điều khiển đang ở chế độ sleep, và ngắt truyền nhận cho phép đánh thức? vi điều khiển. Khi chỉ cần nhận dữ liệu, ta có thể án định RC5/SDO là ngõ vào (set bit TRISC<5>). Slave mode cho phép sự tác động của chân điều khiển RA5/AN4/SS/C2OUT (SSPCON<3:0> = 0100).

Khi chân RA5/AN4/SS/C2OUT ở mức thấp, chân RC5/SDO được cho phép xuất dữ liệu và khi RA5/AN4/SS/C2OUT ở mức cao, dữ liệu ra ở chân RC5/SDO bị khóa, đồng thời SPI được reset (bộ đếm bit dữ liệu được gán giá trị 0)



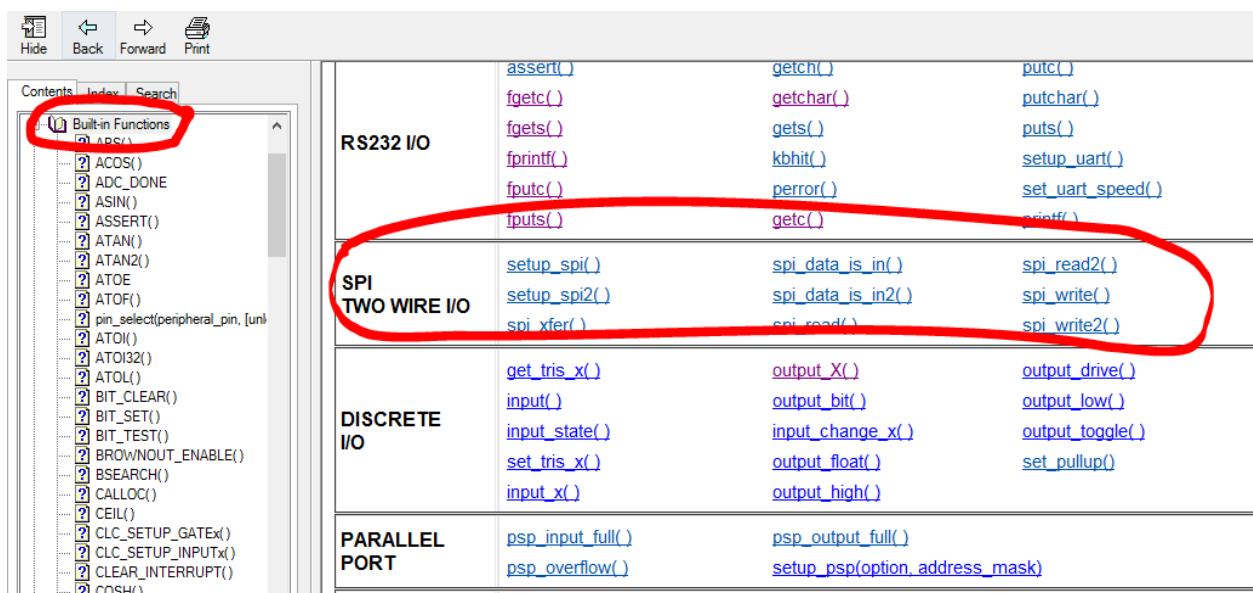
Hình 3.36. Giản đồ xung SPI ở chế độ Slave

Các thanh ghi liên quan đến chuẩn giao tiếp SPI bao gồm:

- ✓ Thanh ghi INTCON (địa chỉ 0Bh, 8Bh, 10Bh, 18Bh): chứa bit cho phép toàn bộ các ngắt (GIE và PEIE).
- ✓ Thanh ghi PIR1 (địa chỉ 0Ch): chứa ngắt SSPIE.
- ✓ Thanh ghi PIE1 (địa chỉ 8Ch): chứa bit cho phép ngắt SSPIE.
- ✓ Thanh ghi TRISC (địa chỉ 87h): điều khiển xuất nhập PORTC.
- ✓ Thanh ghi SSPBUF (địa chỉ 13h): thanh ghi bộ đếm dữ liệu.
- ✓ Thanh ghi SSPCON (địa chỉ 14h): điều khiển chuẩn giao tiếp SPI.
- ✓ Thanh ghi SSPSTAT (địa chỉ 94h): chứa các bit chỉ thị trạng thái chuẩn giao tiếp SPI.
- ✓ Thanh ghi TRISA (địa chỉ 85h): điều khiển xuất nhập chân .

#### 4.9.1.3. Các Hàm SPI Của CCS

Để xem các hàm CCS hỗ trợ dung Timer chúng ta vào CCS C Compiler Help => Built-in Functions => SPI TWO WIRE I/O.



Hình 3.37. Các hàm CCS hỗ trợ SPI

Để sử dụng các hàm sau trước tiên bạn phải khai báo: #use spi (*options*) trong đó Options là thành phần được ngăn cách bởi dấu phẩy có thể là:

- ✓ MASTER Thiết lập thiết bị là master. (mặc định)
  - ✓ SLAVE Thiết lập thiết bị là slave.
  - ✓ BAUD=n Tốc độ bit / giây, mặc định là nhanh nhất có thể thực hiện được.
  - ✓ CLOCK\_HIGH=n Thời gian mức cao của xung, đơn vị là us (Không được sử dụng nếu BAUD= được sử dụng). (Mặc định=0)
  - ✓ CLOCK\_LOW=n Thời gian mức thấp của xung, đơn vị là us (Không được sử dụng nếu BAUD= được sử dụng). (Mặc định=0)
  - ✓ DI=pin Chân chọn nhận data.
  - ✓ DO=pin Chân chọn phát data
  - ✓ CLK=pin Chân phát xung.
  - ✓ MODE=n Chọn chế độ SPI bus.
  - ✓ ENABLE=pin Chân lựa chọn hoạt động phát data.
  - ✓ LOAD=pin Chân lựa chọn hoạt động phát xung sau khi data được phát.
  - ✓ DIAGNOSTIC=pin Chân lựa chọn để thiết lập mức cao khi data được lấy mẫu.
  - ✓ SAMPLE\_RISE Lấy mẫu trên cạnh đi lên
  - ✓ SAMPLE\_FALL Lấy mẫu trên cạnh đi xuống (Mặc định).
  - ✓ BITS=n Số lớn nhất của các bit trong một lần phát . (Mặc định=32)
  - ✓ SAMPLE\_COUNT=n Số các mẫu để gọi(Sử dụng quyền data số). (Mặc định=1)
  - ✓ LOAD\_ACTIVE=n Trạng thái hoạt động chân LOAD(0, 1).
  - ✓ ENABLE\_ACTIVE=n Trạng thái hoạt động chân ENABLE(0, 1). (Mặc định=0)
  - ✓ IDLE=n Trạng thái không hoạt động chân CLK(0, 1). (Mặc định=0)
  - ✓ ENABLE\_DELAY=n Thời gian us chẽ sau khi ENABLE được hoạt động. (Mặc định=0)
  - ✓ DATA\_HOLD=n Thời gian giữa thay đổi data và xung thay đổi.
  - ✓ LSB\_FIRSTLSB gửi đi đầu tiên.
  - ✓ MSB\_FIRSTMSB gửi đi đầu tiên. (mặc định)
  - ✓ STREAM=id Chỉ rõ một tên stream cho giao thức này.
  - ✓ SPI1 Sử dụng các chân phần cứng cho port SPI port 1.

- ✓ SPI2 Sử dụng các chân phần cứng cho port SPI port 2.

- ✓ FORCE\_HW Sử dụng các chân pic phần cứng SPI.

**Ví dụ:**

```
#use spi(DI=PIN_B1, DO=PIN_B0, CLK=PIN_B2, ENABLE=PIN_B4, BITS=16) // uses software SPI
```

```
#use spi(FORCE_HW, BITS=16, stream=SPI_STREAM) // uses hardware SPI and gives this stream the name SPI_STREAM
```

➤ **Setup\_spi (mode);**

➤ **Setup\_spi2 (mode);**

- ✓ Dùng thiết lập giao tiếp SPI . Hàm thứ 2 dùng với VĐK có 2 bộ SPI .

- ✓ Tham số mode :là các hằng số sau , có thể OR giữa các nhómbởi dấu |

- SPI\_MASTER , SPI\_SLAVE , SPI\_SS\_DISABLED => xác định VĐK là master hay slave ,slave select.
- SPI\_L\_TO\_H , SPI\_H\_TO\_L => xác định clock cạnh lên hay xuống
- SPI\_CLK\_DIV\_4 , SPI\_CLK\_DIV\_16 , SPI\_CLK\_DIV\_64 , SPI\_CLK\_T2 => xác định tần số xung clock , SPI\_CLK\_DIV\_4 nghĩa là tần số = FOSC / 4 , tương ứng 1 chu kỳ lệnh / xung xem lại thanh ghi SSPCON1 (**địa chỉ 14h**).

=> Hàm không trả về trị .

=> Ngoài ra ,tùy VĐK mà có thêm 1 số tham số khác , xem file \*.h.

Ví dụ: Setup\_spi(SPI\_MASTER| SPI\_L\_TO\_H| SPI\_CLK\_DIV\_4) VĐK làm master, tác động khi xung lên, qua bộ chia FOSC/4.

➤ **Spi\_read ( data )**

➤ **Spi\_read2 ( data )**

- ✓ Hàm đọc giá trị SPI có giá trị 8 bit. Ham thứ 2 cho bộ SPI thứ 2 (nếu VĐK có nhiều hơn 1 bộ).

- ✓ Hàm trả về giá trị đọc bởi SPI . Nếu value phù hợp SPI\_read ( ) thì data sẽ được phát xung ngoài và data nhận được sẽ được trả về . Nếu không có data sẵn sàng , spi\_read ( ) sẽ đợi data .

- ✓ Hàm chỉ dùng cho SPI hardware ( SPI phần cứng ).

Ví dụ: int8 value;

```
Value = spi_read(data);
```

➤ **Spi\_write (value)**

➤ **Spi\_write2 (value)**

- ✓ Hàm không trả về trị. value là giá trị 8 bit .

- ✓ Hàm này gửi value (1 byte) tới SPI, đồng thời tạo 8 xung clock .

- ✓ Hàm chỉ dùng cho SPI hardware ( SPI phần cứng ).

➤ **spi\_data\_is\_in ()**

➤ **Spi\_data\_is\_in2 ()**

- ✓ Hàm trả về TRUE (1) nếu data nhận được đầy đủ (8 bit) từ SPI, trả về false nếu chưa nhận đủ .

- ✓ Hàm này dùng kiểm tra xem giá trị nhận về SPI đã đủ 1 byte chưa để dùng hàm spi\_read() đọc data vào biến.
- **spi\_xfer()**
- ✓ Phát dữ liệu tới và đọc data từ một thiết bị SPI.
- ✓ Đặc điểm đặc biệt của hàm này là vừa phát vừa đọc data thông qua thiết bị SPI trong cùng một thời điểm nó tương đương với hai hàm SPI\_WRITE() và SPI\_READ() cộng lại.

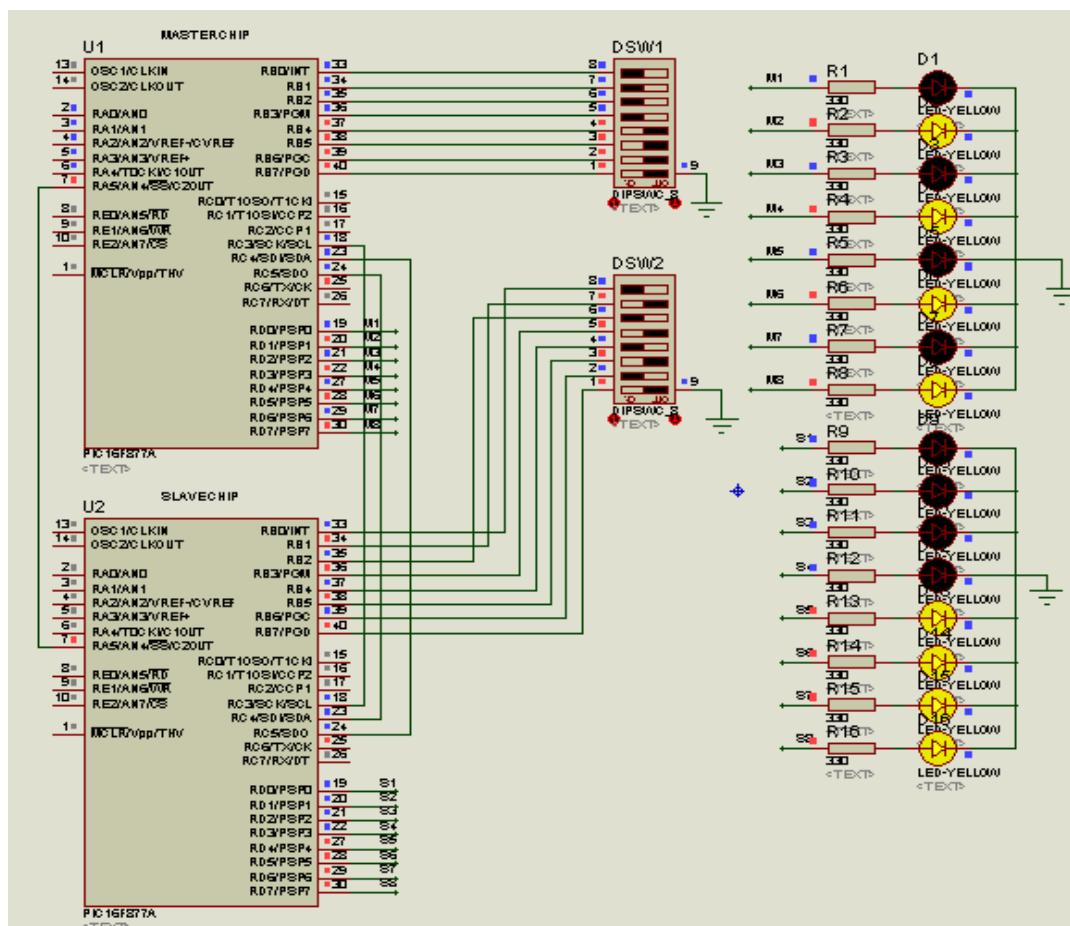
**Ví dụ:**

```
int i = 34;
spi_xfer(i);                                // Phát một số 34 qua SPI
int trans = 34, res;
res = spi_xfer(trans);
                                                // Phát một số 34 qua SPI
                                                // cũng đọc cả số đến từ SPI
```

#### 4.9.1.4. Bài tập về SPI

Giao tiếp SPI song công giữa 2 PIC: PIC Master ở trên truyền dữ liệu từ PortB (công tắc trên) qua PIC Slave ở dưới để hiển thị ra PortD (LED dưới), PIC Slave cũng lấy dữ liệu từ PortB (công tắc dưới) của mình, truyền qua PIC Master để hiển thị ra PortD (LED trên).

- Kết nối phần cứng và mô phỏng:



Hình 4.38. Sơ đồ kết nối SPI

- **Chương trình:**

**Code Master:**

```
#include <16f877a.h>
```

```
#include <def_877a.h>
#device *=16 ADC=8
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT,NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
#use fast_io(B)
#use fast_io(D)
#use fast_io(A)
#define REG_Write 0x80

#INT_SSP
void spi()
{
    PORTD=spi_read(PORTB);
    delay_ms(10);
}
void main()
{
    port_b_pullups(TRUE);
    setup_spi(spi_master|spi_1_to_h|spi_clk_div_16);
    enable_interrupts(INT_SSP);
    enable_interrupts(GLOBAL);
    SET_TRIS_B(0xff);
    SET_TRIS_D(0x00);
    SET_TRIS_A(0x00);
    while(1)
    {
        delay_ms(100);
        output_low(PIN_A5);      //Chân C2 dùng Select chip.
        delay_ms(10);           //Tạo tre de Slave chuẩn bị.
        spi_write(PORTB);
        output_high(PIN_A5);
    }
}
```

**Code Slave:**

```
#include <16f877a.h>
#include <def_877a.h>
#device *=16 ADC=8
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT,NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
#use fast_io(B)
#use fast_io(D)
```

```
#INT_SSP
void spi()
{
    PORTD=spi_read(PORTB); //Vua nhan vua truyen.
    delay_ms(10);
}

void main()
{
    port_b_pullups(TRUE);
    setup_spi(spi_slave|spi_l_to_h|spi_clk_div_16);
    enable_interrupts(INT_SSP);
    enable_interrupts(GLOBAL);
    TRISB=0xff;
    TRISD=0x00;
    while(1)
    {
    }
}
}
```

#### 4.9.2. I2C MODE

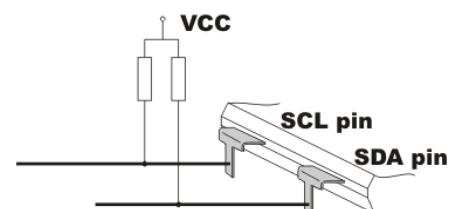
I2C-Inter Intergrated Circuit- do hãng Phillips phát triển Nhiều nhà sản xuất IC trên thế giới sử dụng: Texas Intrument (TI), Maxim Dallas, Analog Device, National Semiconductor ... Bus I2C được sử dụng làm bus giao tiếp ngoại vi cho rất nhiều loại IC như các loại VĐK 8051, PIC, AVR chip nhớ như RAM static, ADC, DAC, các IC điều khiển LCD, LED. Đây là một dạng khác của MSSP. Chuẩn giao tiếp I2C cũng có hai chế độ Master,Slave và cũng được kết nối với ngắt. I2C sẽ sử dụng 2 pin để truyền nhận dữ liệu:

- ✓ RC3/SCK/SCL: là đường truyền xung nhịp một hướng duy nhất do thiết bị Master làm chủ.
- ✓ RC4/SDI/SDA: chân truyền dẫn dữ liệu theo hai hướng.

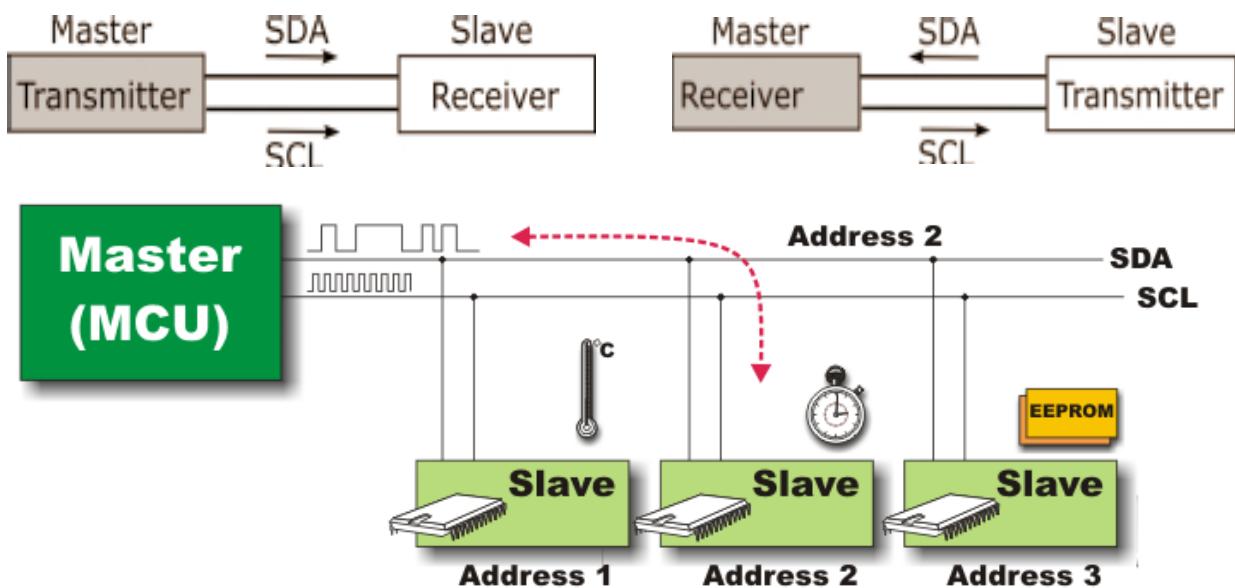
Mỗi dây SDA hay SCL đều được nối với điện áp dương của nguồn cấp thông qua một điện trở kéo lên do các chân giao tiếp có dạng cực máng hở. Giá trị các điện trở này tùy vào từng thiết bị thường giao động từ 1k đến 4k7, bên cạnh đó cần xác định các giá trị phù hợp cho các bit TRISC<4:3> (bit điều khiển xuất nhập các chân SCL và SDA).

Mỗi thiết bị sẽ được nhận ra bởi một địa chỉ duy nhất

I2C quy định một thiết bị Master và các thiết bị khác là Slave mỗi Slave có một địa chỉ riêng để Master có thể phân biệt. Xung nhịp SCL do Master làm chủ khi muốn giao tiếp với Slave, Master sẽ gửi byte địa chỉ kèm theo xung nhịp SCL, byte address này khớp với Slave nào



thì Slave đây sẽ phát ra một xung ACK (chấp nhận) báo rằng nó có thể truyền/nhận với Master. Việc Truyền/nhận dừng khi có một lệnh i2c\_stop() và bắt đầu khởi động bằng một lệnh i2c\_start(). Cấu hình là thiết bị chủ (master) hay tớ (slave), quyền điều khiển thuộc về thiết bị chủ.



Hình 4.39. Sơ đồ kết nối 1 master 3 slave

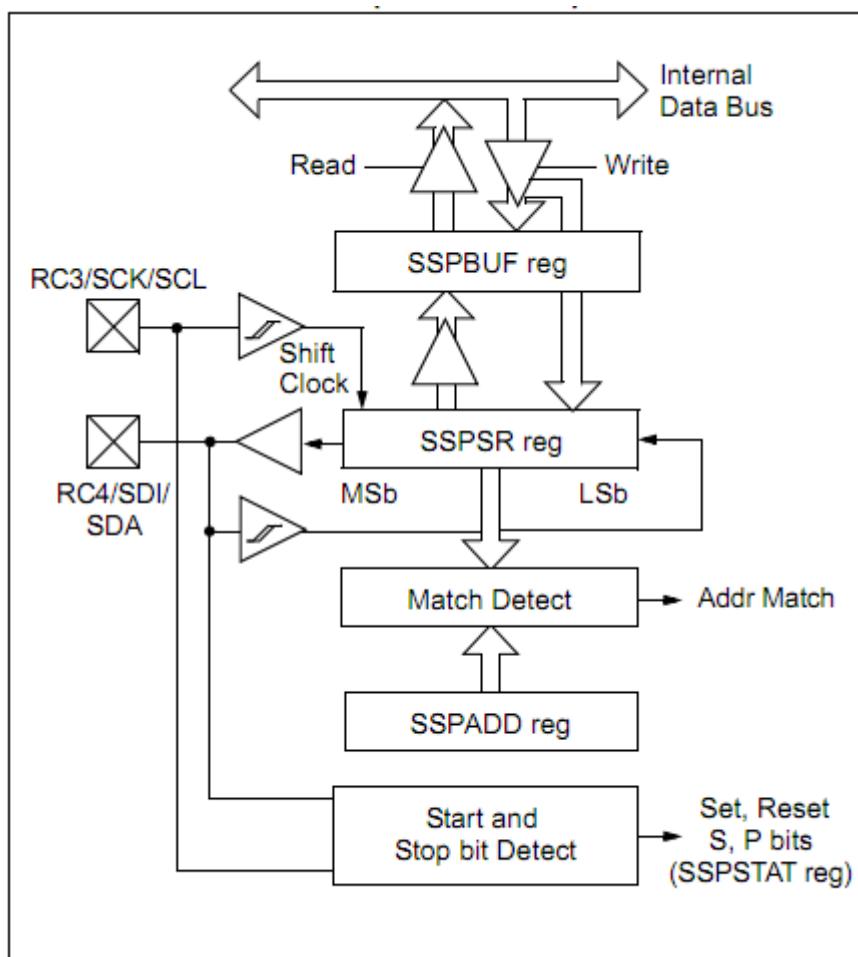
Các khái niệm cơ bản trong sơ đồ khái niệm của I2C không có nhiều khác biệt so với SPI. Tuy nhiên I2C còn có khái niệm bit Start và bit Stop của dữ liệu (Start and Stop bit detect) và khái niệm xác định địa chỉ (Match detect).

Các thanh ghi liên quan đến I2C bao gồm:

- ✓ Thanh ghi SSPCON1 và SSPCON2: điều khiển MSSP.
- ✓ Thanh ghi SSPSTAT: thanh ghi chứa các trạng thái hoạt động của MSSP.
- ✓ Thanh ghi SSPBUF: buffer truyền nhận nối tiếp.
- ✓ Thanh ghi SSPSR: thanh ghi dịch dùng để truyền nhận dữ liệu.
- ✓ Thanh ghi SSPADD: thanh ghi chứa địa chỉ của giao diện MSSP.

Các thanh ghi SSPCON1, SSPCON2 cho phép đọc và ghi. Thanh ghi SSPSTAT chỉ cho phép đọc và ghi ở 2 bit đầu, 6 bit còn lại chỉ cho phép đọc.

Thanh ghi SSPBUF chứa dữ liệu sẽ được truyền đi hoặc nhận được và đóng vai trò như một thanh ghi đệm cho thanh ghi dịch dữ liệu SSPSR. Thanh ghi SSPADD chứa địa chỉ của thiết bị ngoại vi cần truy xuất dữ liệu của I2C khi hoạt động ở Slave mode. Khi hoạt động ở Master mode, thanh ghi SSPADD chứa giá trị tạo ra tốc độ baud cho xung clock dùng để truyền nhận dữ liệu. Trong quá trình nhận dữ liệu, sau khi nhận được 1 byte dữ liệu hoàn chỉnh, thanh ghi SSPSR sẽ chuyển dữ liệu vào thanh ghi SSPBUF. Thanh ghi SSPSR không đọc và ghi được, quá trình truy xuất thanh ghi này phải thông qua thanh ghi SSPBUF. Trong quá trình truyền dữ liệu, dữ liệu cần truyền khi được đưa vào thanh ghi SSPBUF cũng đồng thời đưa vào thanh ghi SSPSR.



Hình 4.40. Sơ đồ khái niệm MSSP (I2C slave mode).

I2C có nhiều chế độ hoạt động và được điều khiển bởi các bit SSPCON<3:0>, bao gồm:

- ✓ I2C Master mode, xung clock = fosc/4\*(SSPADD+1).
- ✓ I2C Slave mode, 7 bit địa chỉ.
- ✓ I2C Slave mode, 10 bit địa chỉ.
- ✓ I2C Slave mode, 7 bit địa chỉ, cho phép ngắt khi phát hiện bit Start và bit Stop.
- ✓ I2C Slave mode, 10 bit địa chỉ, cho phép ngắt khi phát hiện bit Start và bit Stop.
- ✓ I2C Firmware Control Master mode.

Địa chỉ truyền đi sẽ bao gồm các bit địa chỉ và một bit R/W để xác định thao tác (đọc hay ghi dữ liệu) với đối tượng cần truy xuất dữ liệu.

#### Tốc độ :

- ✓ 100Kbits/s – Chế độ chuẩn (Standard mode).
- ✓ 400Kbits/s – Chế độ nhanh (Fast mode)
- ✓ 3,4Mbps – Chế độ cao tốc (High speed mode)

#### Kết nối:

- ✓ Một chủ một tớ (one master – one slave)
- ✓ Một chủ nhiều tớ (one master – multi slave)
- ✓ Nhiều chủ nhiều tớ (Multi master – multi slave)

Khi lựa chọn giao diện I2C và khi set bit SSPEN, các pin SCL và SDA sẽ ở trạng thái cực thu hồi. Do đó trong trường hợp cần thiết ta phải sử dụng điện trở kéo lên ở bên ngoài vi điều khiển, bên cạnh đó cần xác định các giá trị phù hợp cho các bit TRISC<4:3> (bit điều khiển xuất nhập các chân SCL và SDA).

#### 4.9.2.1. I2C SLAVE MODE:

Việc trước tiên là phải set các pin SCL và SDA là input (set bit TRISC<4:3>). I2C của vi điều khiển sẽ được điều khiển bởi một vi điều khiển hoặc một thiết bị ngoại vi khác thông qua các địa chỉ. Khi địa chỉ này chỉ đến vi điều khiển, thì tại thời điểm này và tại thời điểm dữ liệu đã được truyền nhận xong sau đó, vi điều khiển sẽ tạo ra xung  $\overline{\text{ACK}}$  để báo hiệu kết thúc dữ liệu, giá trị trong thanh ghi SSPSR sẽ được đưa vào thanh ghi SSPBUF. Tuy nhiên xung  $\overline{\text{ACK}}$  sẽ không được tạo ra nếu một trong các trường hợp sau xảy ra:

Bit BF (SSPSTAT<0>) báo hiệu buffer đầy đã được set trước khi quá trình truyền nhận xảy ra.

Bit SSPOV (SSPCON<6>) được set trước khi quá trình truyền nhận xảy ra (SSPOV được set trong trường hợp khi một byte khác được nhận vào trong khi dữ liệu trong thanh ghi SSPBUF trước đó vẫn chưa được lấy ra).

Trong các trường hợp trên, thanh ghi SSPSR sẽ không đưa giá trị vào thanh ghi SSPBUF, nhưng bit SSPIF (PIR1<3>) sẽ được set. Để quá trình truyền nhận dữ liệu được tiếp tục, cần đọc dữ liệu từ thanh ghi SSPBUF vào trước, khi đó bit BF sẽ tự động được xóa, còn bit SSPOV phải được xóa bằng chương trình. Khi MSSP được kích hoạt, nó sẽ chờ tín hiệu để bắt đầu hoạt động.

auk hi nhận được tín hiệu bắt đầu hoạt động (cạnh xuống đầu tiên của pin SDA), dữ liệu 8 bit sẽ được dịch vào thanh ghi SSPSR. Các bit đưa vào sẽ được lấy mẫu tại cạnh lên của xung clock. Giá trị nhận được từ thanh ghi SSPSR sẽ được so sánh với giá trị trong thanh ghi SSPADD tại cạnh xuống của xung clock thứ 8. Nếu kết quả so sánh bằng nhau, tức là I2C Master chỉ định đối tượng giao tiếp là vi điều khiển đang ở chế độ Slave mode (ta gọi hiện tượng này là address match), bit BF và SSPOV sẽ được xóa về 0 và gây ra các tác động sau:

1. Giá trị trong thanh ghi SSPSR được đưa vào thanh ghi SSPBUF.
2. Bit BF tự động được set.
3. Một xung được tạo ra.
4. Cờ ngắt SSPIF được set (ngắt được kích hoạt nếu được cho phép trước đó) tại cạnh xuống của xung clock thứ 9.

Khi MSSP ở chế độ I2C Slave mode 10 bit địa chỉ, vi điều khiển cần nhận vào 10 bit địa chỉ để so sánh. Bit (SSPSTAT<2>) phải được xóa về 0 để cho phép nhận 2 byte địa chỉ. Byte đầu tiên có định dạng là ‘11110 A9 A8 0’ trong đó A9, A8 là hai bit MSB của 10 bit địa chỉ. Byte thứ 2 là 8 bit địa chỉ còn lại.

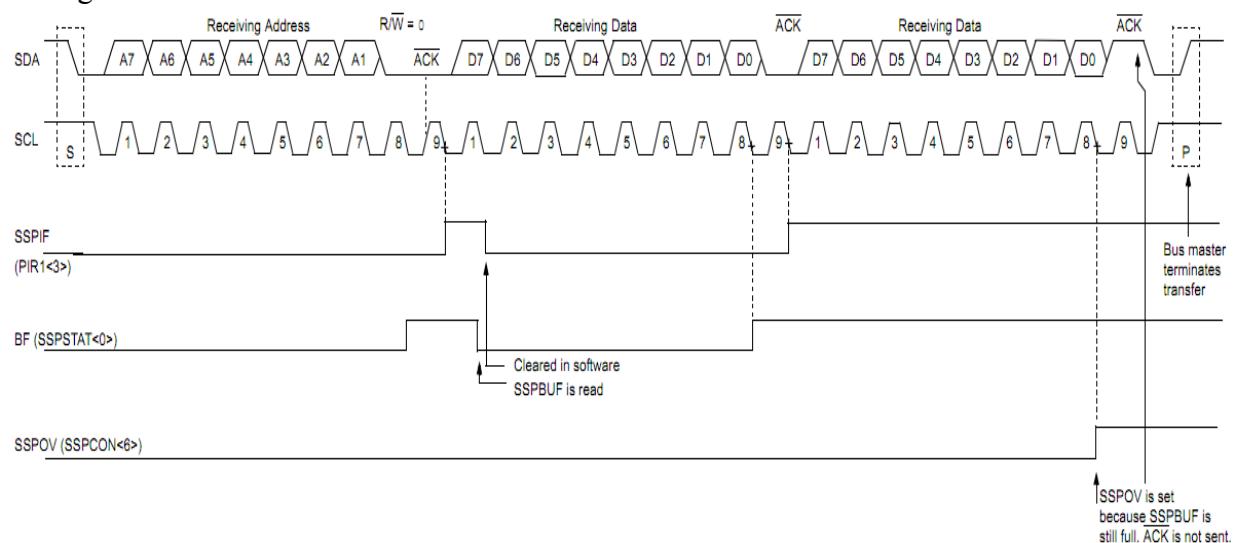
Quá trình nhận dạng địa chỉ của MSSP ở chế độ I2C Slave mode 10 bit địa chỉ như sau:

- ✓ Đầu tiên 2 bit MSB của 10 bit địa chỉ được nhận trước, bit SSPIF, BF và UA (SSPSTAT<1>) được set (byte địa chỉ đầu tiên có định dạng là ‘11110 A9 A8 0’).
- ✓ Cập nhật vào 8 bit địa chỉ thấp của thanh ghi SSPADD, bit UA sẽ được xóa bởi vi điều khiển để khởi tạo xung clock ở pin SCL sau khi quá trình cập nhật hoàn tất.
- ✓ Đọc giá trị thanh ghi SSPBUF (bit BF sẽ được xóa về 0) và xóa cờ ngắt SSPIF.
- ✓ Nhận 8 bit địa chỉ cao, bit SSPIF, BF và UA được set.
- ✓ Cập nhật 8 bit địa chỉ đã nhận được vào 8 bit địa chỉ cao của thanh ghi SSPADD, nếu địa chỉ nhận được là đúng (address match), xung clock ở chân SCL được khởi tạo và bit UA được set.
- ✓ Đọc giá trị thanh ghi SSPBUF (bit BF sẽ được xóa về 0) và xóa cờ ngắt SSPIF.
- ✓ Nhận tín hiệu Start.
- ✓ Nhận byte địa chỉ cao (bit SSPIF và BF được set).

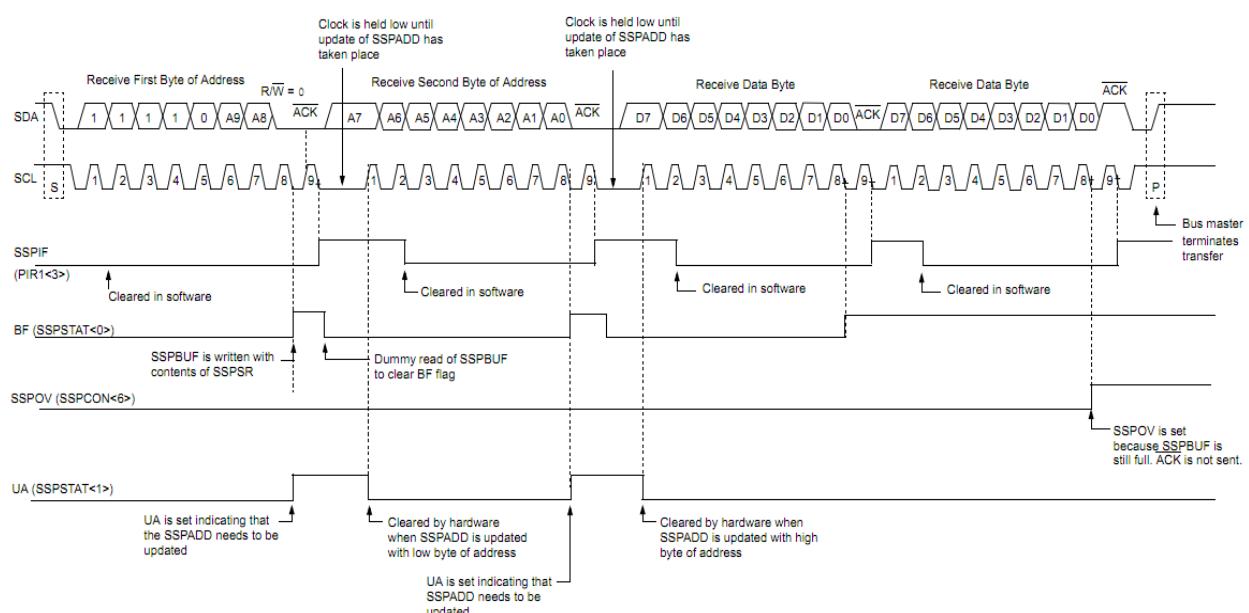
- ✓ Đọc giá trị thanh ghi SSPBUF (bit BF được xóa về 0) và xóa cờ ngắt SSPIF.

Trong đó các bước 7,8,9 xảy ra trong quá trình truyền dữ liệu ở chế độ Slave mode. Xem giản đồ xung của I2C để có được hình ảnh cụ thể hơn về các bước tiến hành trong quá trình nhận dạng địa chỉ.

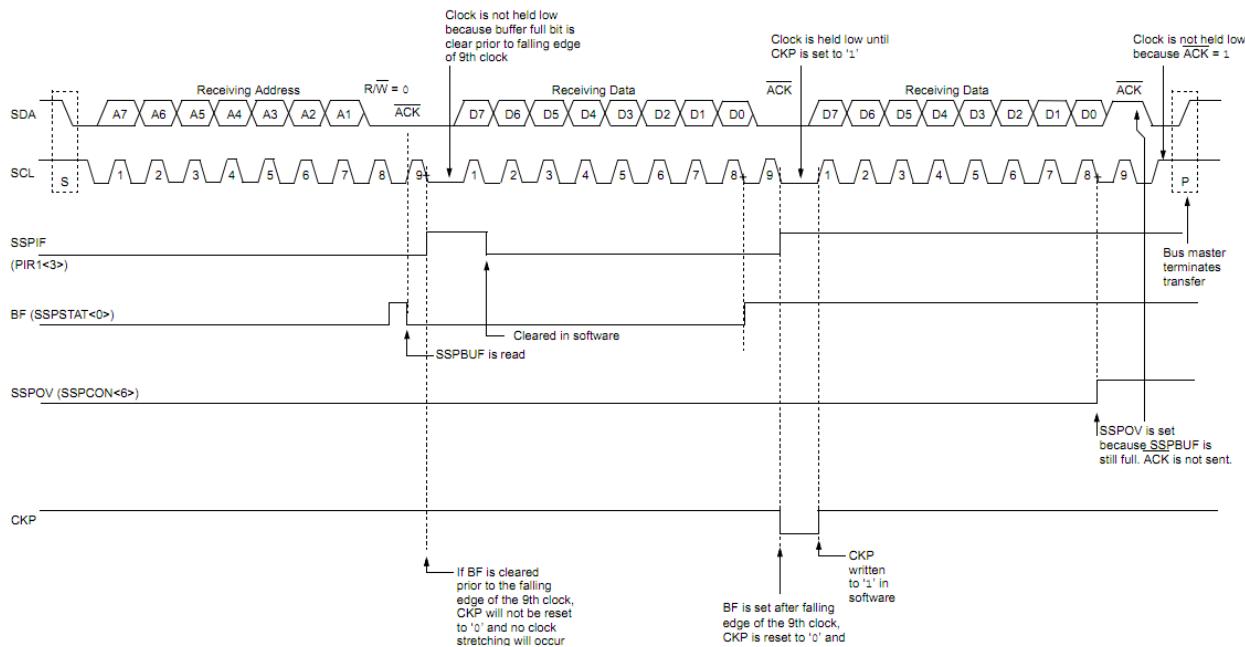
Xét quá trình nhận dữ liệu ở chế độ Slave mode, các bit địa chỉ sẽ được I2C Master đưa vào trước. Khi bit R/W trong các bit địa chỉ có giá trị bằng 0 (bit này được nhảy sau khi các bit địa chỉ đã được nhận xong) và địa chỉ được chỉ định đúng (address match), bit R/W của thanh ghi SSPSTAT được xóa về 0 và đường dữ liệu SDI được đưa về mức logic thấp (xung ACK). Khi bit SEN (SSPCON<0>) được set, sau khi 1 byte dữ liệu được nhận, xung clock từ chân RC3/SCK/SCL sẽ được đưa xuống mức thấp, muộn khởi tạo lại xung clock ta set bit CKP (SSPCON<4>). Điều này sẽ làm cho hiện tượng tràn dữ liệu không xảy ra vì bit SEN cho phép ta điều khiển được xung clock dịch dữ liệu thông qua bit CKP (tham khảo giản đồ xung để biết auk chi tiết). Khi hiện tượng tràn dữ liệu xảy ra, bit BF hoặc bit SSPOV sẽ được set. Ngắt sẽ xảy ra khi một byte dữ liệu được nhận xong, cờ ngắt SSPIF sẽ được set và phải được xóa bằng chương trình.



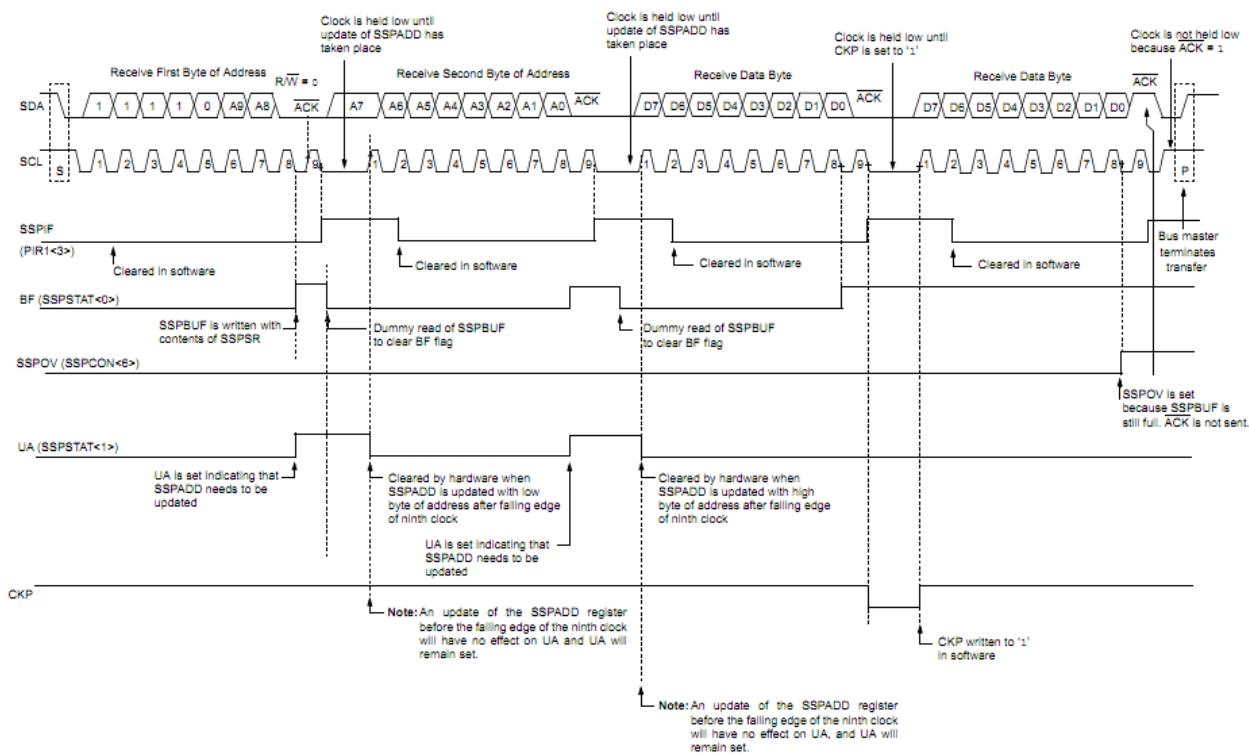
Hình 4.41. Gian đồ xung của I2C Slave mode 7 bit địa chỉ trong quá trình nhận dữ liệu (bit SEN = 0).



Hình 4.42. Giản đồ xung của I2C Slave mode 10 bit địa chỉ trong quá trình nhận dữ liệu (bit SEN= 0).



Hình 4.43. Giản đồ xung của I2C Slave mode 7 bit địa chỉ trong quá trình nhận dữ liệu (bit SEN = 1).

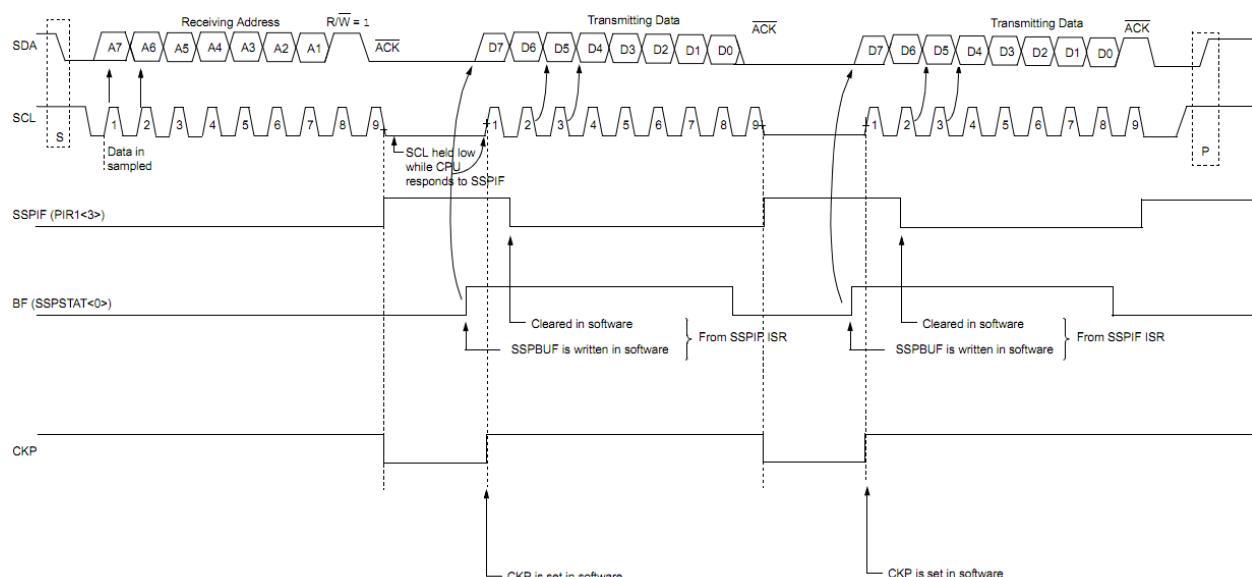


Hình 4.44. Giản đồ xung của I2C Slave mode 10 bit địa chỉ trong quá trình nhận dữ liệu (bit SEN = 1).

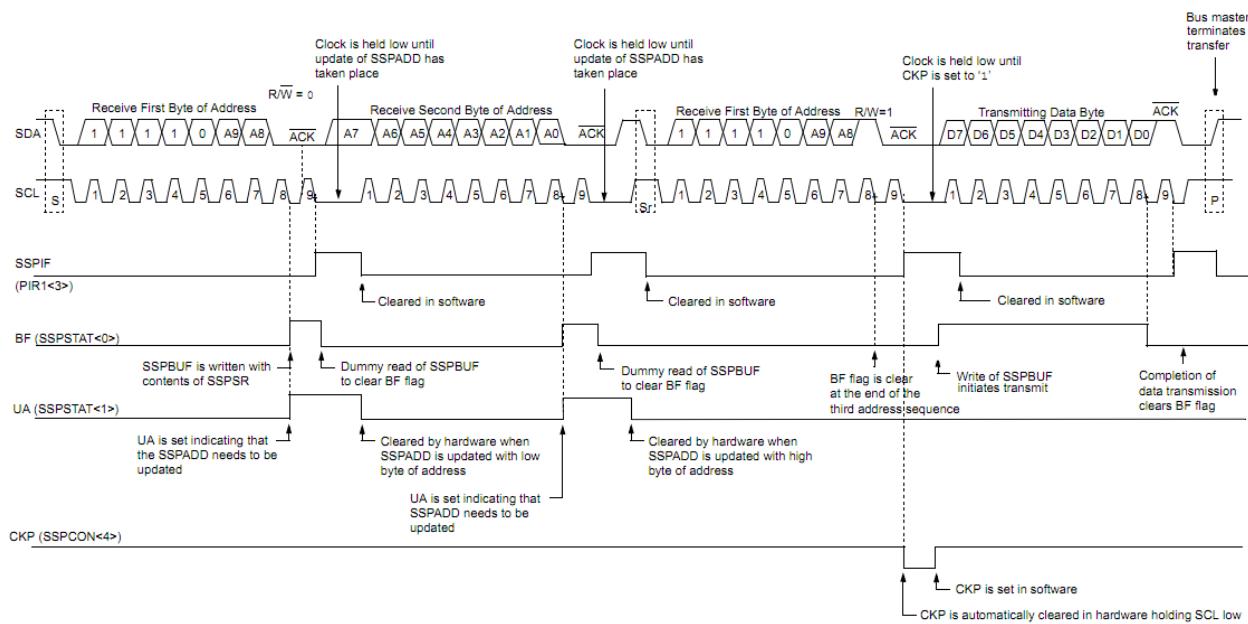
Xét quá trình truyền dữ liệu, khi bit  $R/W$  trong các bit dữ liệu mang giá trị 1 và địa chỉ được chỉ định đúng (address match), bit  $R/W$  của thanh ghi SSPSTAT sẽ được set. Các bit địa chỉ

được nhận trước và đưa vào thanh ghi SSPBUF. Sau đó xung  $\overline{\text{ACK}}$  được tạo ra, xung clock ở chân RC3/SCK/SCL được đưa xuống mức thấp bắt chấp trạng thái của bit SEN. Khi đó I2C Master sẽ không được đưa xung clock vào I2C Slave cho đến khi dữ liệu ở thanh ghi SSPSR ở trạng thái sẵn sàng cho quá trình truyền dữ liệu (dữ liệu đưa vào thanh ghi SSPBUF sẽ đồng thời được đưa vào thanh ghi SSPSR). Tiếp theo cho phép xung ở pin RC3/SCK/SCL bằng cách set bit CKP (SSPCON<4>). Từng bit của byte dữ liệu sẽ được dịch ra ngoài tại mỗi cạnh xuống của xung clock. Như vậy dữ liệu sẽ sẵn sàng ở ngõ ra khi xung clock ở mức logic cao, giúp cho I2C Master nhận được dữ liệu tại mỗi cạnh lên của xung clock. Như vậy trong quá trình truyền dữ liệu bit SEN không đóng vai trò quan trọng như trong quá trình nhận dữ liệu.

Tại cạnh lên xung clock thứ 9, dữ liệu đã được dịch hoàn toàn vào I2C Master, xung sẽ được tạo ra ở I2C Master, đồng thời pin SDA sẽ được giữ ở mức logic cao. Trong trường hợp xung  $\overline{\text{ACK}}$  được chốt bởi I2C Slave, thanh ghi SSPSTAT sẽ được reset. I2C Slave sẽ chờ tín hiệu của bit Start để tiếp tục truyền byte dữ liệu tiếp theo (đưa byte dữ liệu tiếp theo vào thanh ghi SSPBUF và set bit CKP). Ngắt MSSP xảy ra khi một byte dữ liệu kết thúc quá trình truyền, bit SSPIF được set tại cạnh xuống của xung clock thứ 9 và phải được xóa bằng chương trình để đảm bảo sẽ được set khi byte dữ liệu tiếp theo truyền xong.

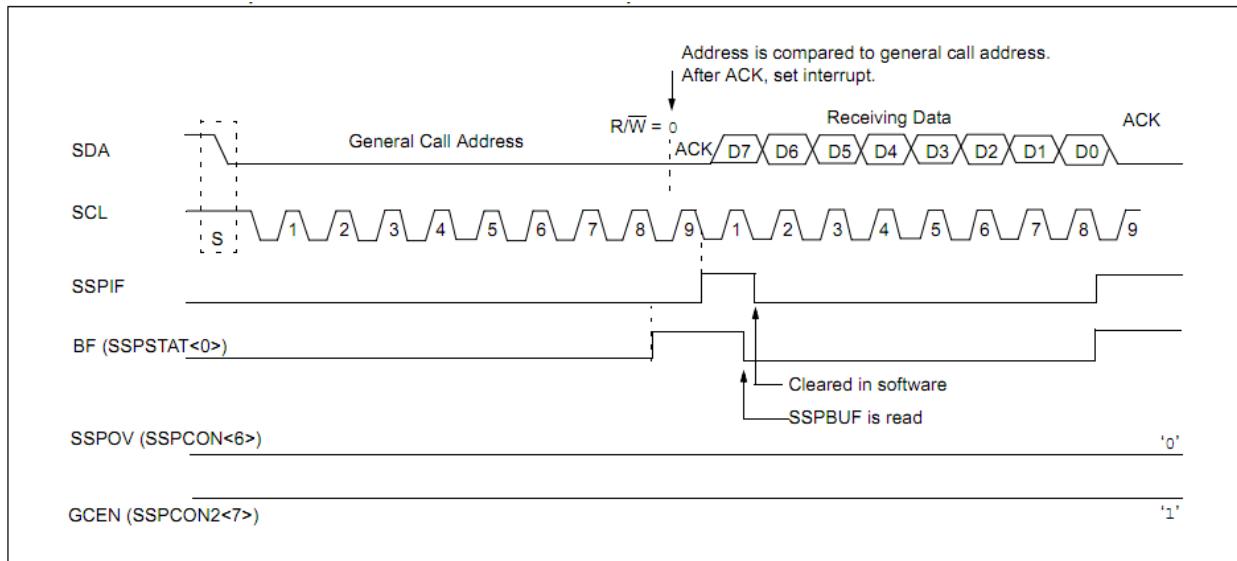


Hình 4.45. Giản đồ xung của I2C Slave mode 7 bit địa chỉ trong quá trình truyền dữ liệu.



Hình 4.46. Giản đồ xung của I2C Slave mode 10 bit địa chỉ trong quá trình truyền dữ liệu.

Quá trình truyền nhận các bit địa chỉ cho phép I2C Master chọn lựa đối tượng I2C Slave cần truy xuất dữ liệu. Bên cạnh đó I2C còn cung cấp một địa chỉ GCA (General Call Address) cho phép chọn tất cả các I2C Slave. Đây là một trong 8 địa chỉ đặc biệt của protocol I2C. Địa chỉ này được định dạng là một chuỗi ‘0’ với =0 và được cho phép bằng cách set bit GCEN (SSPCON2<7>). Khi đó địa chỉ nhận vào sẽ được so sánh với thanh ghi SSPADD và với địa chỉ GCA.

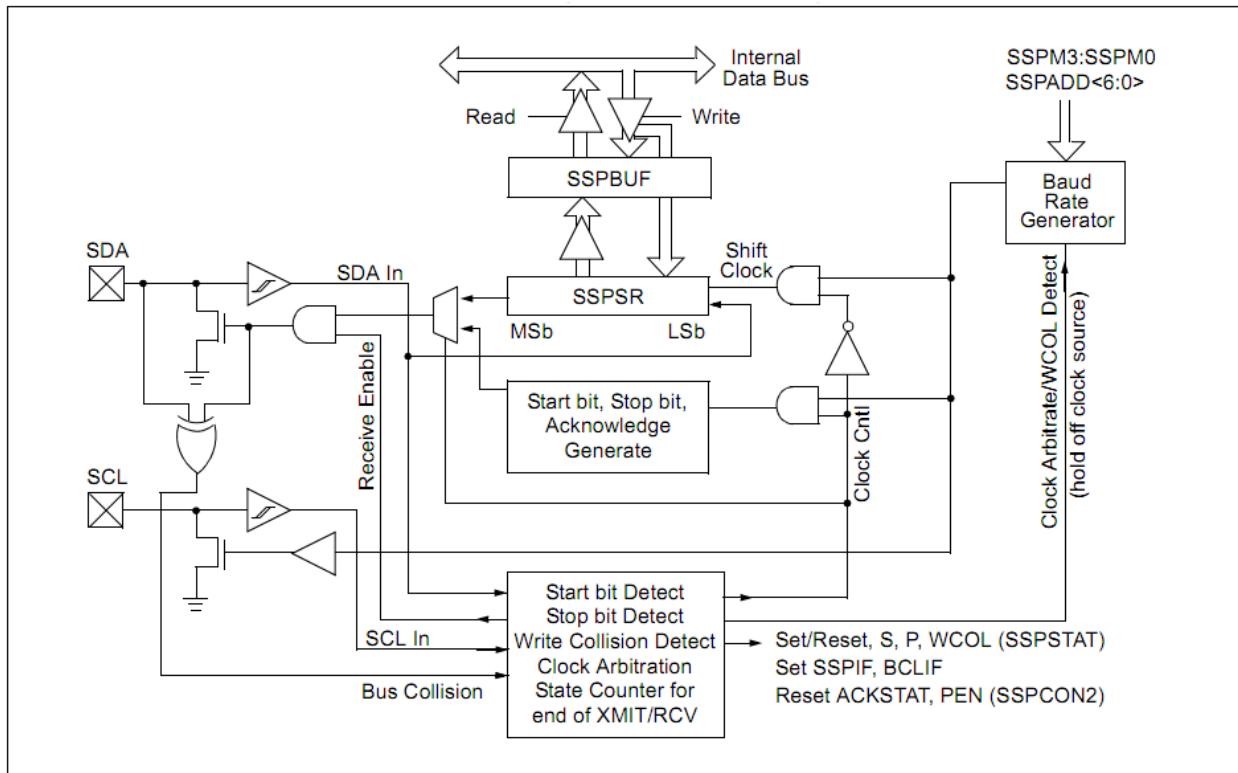


Hình 4.47. Giản đồ xung của I2C Slave khi nhận địa chỉ GCA.

Quá trình nhận dạng địa chỉ GCA cũng tương tự như khi nhận dạng các địa chỉ khác và không có sự khác biệt rõ ràng khi I2C hoạt động ở chế độ địa chỉ 7 bit hay 10 bit.

#### 4.9.2.2. I2C MASTER MODE

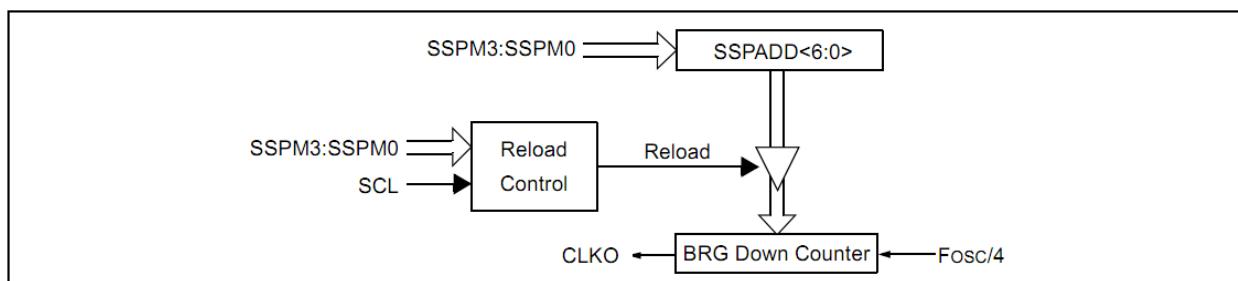
I2C Master mode được xác lập bằng cách đưa các giá trị thích hợp vào các bit SSPM của thanh ghi SSPCON và set bit SSPEN. Ở chế độ Master, các pin SCK và SDA sẽ được điều khiển bởi phần cứng của MSSP.



Hình 4.48. Sơ đồ khối MSSP (I2C Master mode).

I2C Master đóng vai trò tích cực trong quá trình giao tiếp và điều khiển các I2C Slave thông qua việc chủ động tạo ra xung giao tiếp và các điều kiện Start, Stop khi truyền nhận dữ liệu. Một byte dữ liệu có thể được bắt đầu bằng điều kiện Start, kết thúc bằng điều kiện Stop hoặc bắt đầu và kết thúc với cùng một điều kiện khởi động lặp lại (Repeated Start Condition).

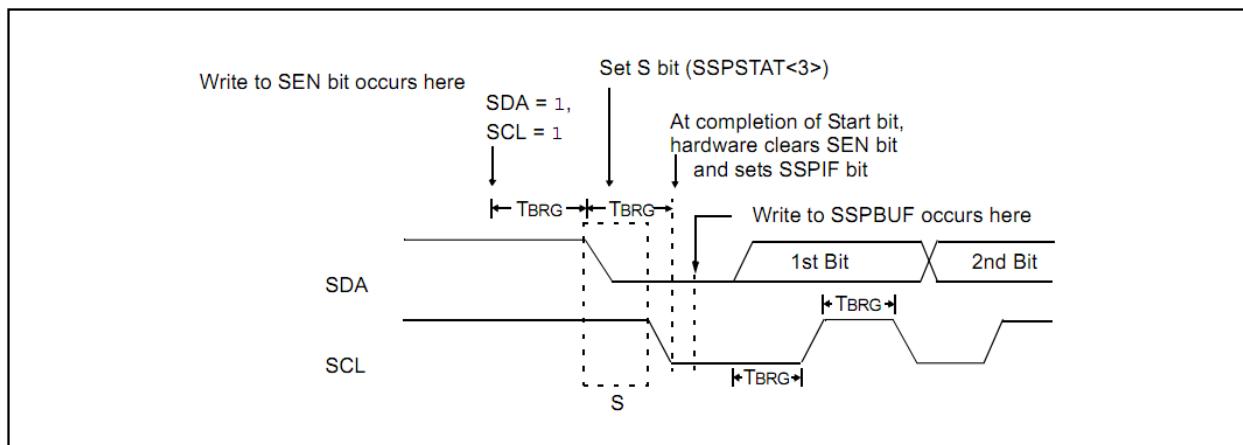
Xung giao tiếp nối tiếp sẽ được tạo ra từ BRG (Baud Rate Generator), giá trị ấn định tần số xung clock nối tiếp được lấy từ 7 bit thấp của thanh ghi SSPADD. Khi dữ liệu được đưa vào thanh ghi SSPBUF, bit BF được set và BRG tự động đếm ngược về 0 và dừng lại, pin SCL được giữ nguyên trạng thái trước đó. Khi dữ liệu tiếp theo được đưa vào, BRG sẽ cần một khoảng thời gian TBRG tự động reset lại giá trị để tiếp tục quá trình đếm ngược. Mỗi vòng lệnh (có thời gian TCY) BRG sẽ giảm giá trị 2 lần.



Hình 4.49. Sơ đồ khối BRG (Baud Rate Generator) của I2C Master mode.  
Các giá trị cụ thể của tần số xung nối tiếp do BRG tạo ra được liệt kê trong bảng sau:

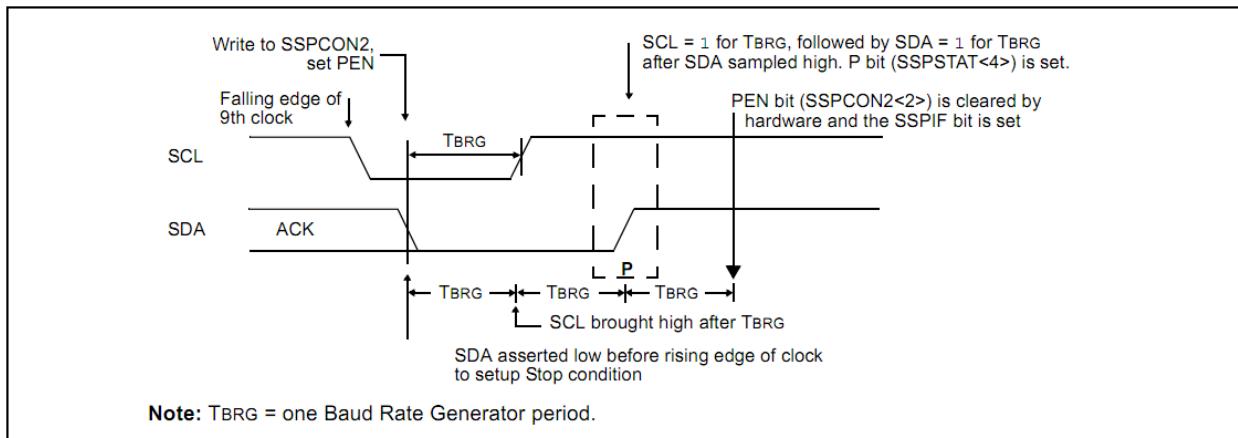
F <sub>CY</sub>	F <sub>CY</sub> *2	BRG Value	F <sub>SCL</sub> (2 Rollovers of BRG)
10 MHz	20 MHz	19h	400 kHz <sup>(1)</sup>
10 MHz	20 MHz	20h	312.5 kHz
10 MHz	20 MHz	3Fh	100 kHz
4 MHz	8 MHz	0Ah	400 kHz <sup>(1)</sup>
4 MHz	8 MHz	0Dh	308 kHz
4 MHz	8 MHz	28h	100 kHz
1 MHz	2 MHz	03h	333 kHz <sup>(1)</sup>
1 MHz	2 MHz	0Ah	100 kHz
1 MHz	2 MHz	00h	1 MHz <sup>(1)</sup>

Trong đó giá trị BRG là giá trị được lấy từ 7 bit thấp của thanh ghi SSPADD. Do I2C ở chế độ Master mode, thanh ghi SSPADD sẽ không được sử dụng để chứa địa chỉ, thay vào đó chức năng của SSPADD là thanh ghi chứa giá trị của BRG. Để tạo được điều kiện Start, trước hết cần đưa hai pin SCL và SDA lên mức logic cao và bit SEN (SSPCON2<0>) phải được set. Khi đó BRG sẽ tự động đọc giá trị 7 bit thấp của thanh ghi SSPADD và bắt đầu đếm. Sau khoảng thời gian TBRG, pin SDA được đưa xuống mức logic thấp. Trạng thái pin SDA ở mức logic thấp và pin SCL ở mức logic cao chính là điều kiện Start của I2C Master mode. Khi đó bit S (SSPSTAT<3>) sẽ được set. Tiếp theo BRG tiếp tục lấy giá trị từ thanh ghi SSPADD để tiếp tục quá trình đếm, bit SEN được tự động xóa và cờ ngắt SSPIF được set. Trong trường hợp pin SCL và SDA ở trạng thái logic thấp, hoặc là trong quá trình tạo điều kiện Start, pin SCL được đưa về trạng thái logic thấp trước khi pin SDA được đưa về trạng thái logic thấp, điều kiện Start sẽ không được hình thành, cờ ngắt BCLIF sẽ được set và I2C sẽ ở trạng thái tạm ngưng hoạt động (Idle).



Hình 4.50. Giản đồ xung I2C Master mode trong quá trình tạo điều kiện Start.

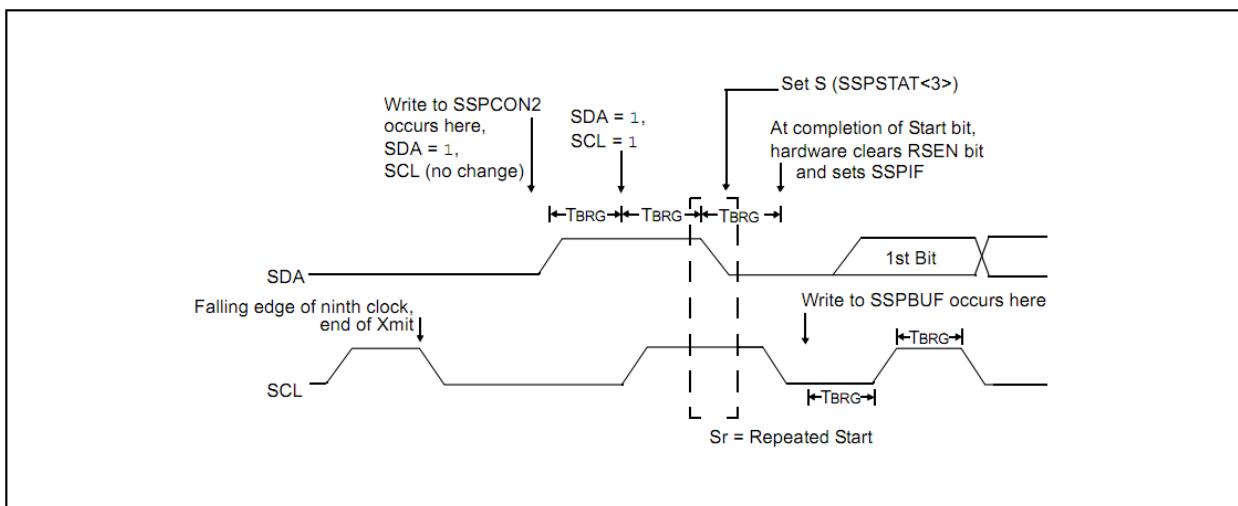
Tín hiệu Stop sẽ được đưa ra pin SDA khi kết thúc dữ liệu bằng cách set bit PEN (SSPCON2<2>). Sau cạnh xuống của xung clock thứ 9 và với tác động của bit điều khiển PEN, pin SDA cũng được đưa xuống mức thấp, BRG lại bắt đầu quá trình đếm. Sau một khoảng thời gian TBRG, pin SCL được đưa lên mức logic cao và sau một khoảng thời gian TBRG nữa pin SDA cũng được đưa lên mức cao. Ngay tại thời điểm đó bit P (SSPSTAT<4>) được set, nghĩa là điều kiện Stop đã được tạo ra. Sau một khoảng thời gian TBRG nữa, bit PEN tự động được xóa và cờ ngắt SSPIF được set.



Hình 4.51. Giản đồ xung I2C Master mode trong quá trình tạo điều kiện Stop.

Để tạo được điều kiện Start lặp lại liên tục trong quá trình truyền dữ liệu, trước hết cần set bit RSEN (SSPCON2<1>). Sau khi set bit RSEN, pin SCL được đưa xuống mức logic thấp, pin SDA được đưa lên mức logic cao, BRG lấy giá trị từ thanh ghi SSPADD vào để bắt đầu quá trình đếm. Sau khoảng thời gian TBRG, pin SCL cũng được đưa lên mức logic cao trong khoảng thời gian TBRG tiếp theo. Trong khoảng thời gian TBRG kế tiếp, pin SDA lại được đưa xuống mức logic thấp trong khi SCL vẫn được giữ ở mức logic cao. Ngay thời điểm đó bit S (SSPSTAT<3>) được set để báo hiệu điều kiện Start được hình thành, bit RSEN tự động được xóa và cờ ngắt SSPIF sẽ được set sau một khoảng thời gian TBRG nữa. Lúc này địa chỉ của I2C Slave có thể được đưa vào thanh ghi SSPBUF, sau đó ta chỉ việc đưa tiếp địa chỉ hoặc dữ liệu tiếp theo vào thanh ghi SSPBUF mỗi khi nhận được tín hiệu ACK từ I2C Slave, I2C Master sẽ tự động tạo tín hiệu Start lặp lại liên tục cho quá trình truyền dữ liệu liên tục.

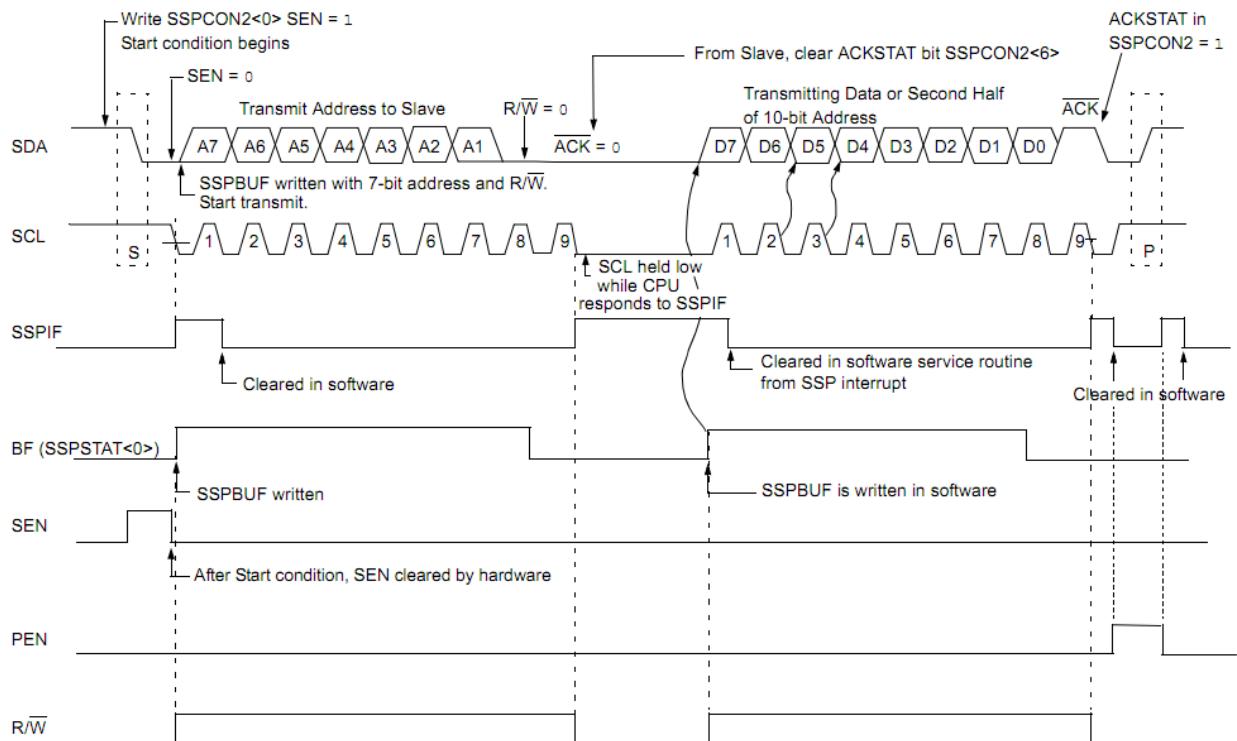
Cần chú ý là bắt cứ một trình tự nào sai trong quá trình tạo điều kiện Start lặp lại sẽ làm cho bit BCLIF được set và I2C được đưa về trạng thái “Idle”.



Hình 4.52. Giản đồ xung I2C Master mode trong quá trình tạo điều kiện Start liên tục.

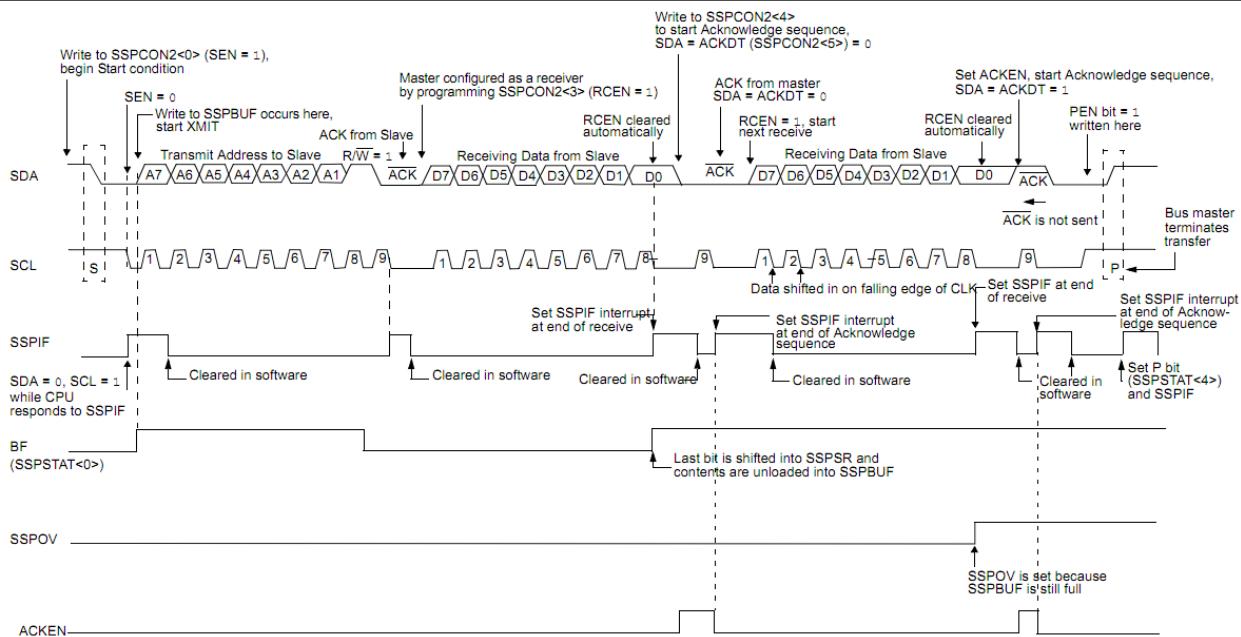
Xét quá trình truyền dữ liệu, xung clock sẽ được đưa ra từ pin SCL và dữ liệu được đưa ra từ pin SDA. Byte dữ liệu đầu tiên phải là byte địa chỉ xác định I2C Slave cần giao tiếp và bit (trong trường hợp này = 0). Đầu tiên các giá trị địa chỉ sẽ được đưa vào thanh ghi SSPBUF, bit BF tự động được set lên 1 và bộ đếm tạo xung clock nối tiếp BRG (Baud Rate Generator) bắt đầu hoạt động. Khi đó từng bit dữ liệu (hoặc địa chỉ và bit ) sẽ được dịch ra ngoài theo từng cạnh xuống của xung clock sau khi cạnh xuống đầu tiên của pin SCL được nhận diện (điều kiện Start), BRG bắt đầu đếm ngược về 0. Khi tất cả các bit của byte dữ liệu được đã được đưa ra ngoài, bộ

đếm BRG mang giá trị 0. Sau đó, tại cạnh xuống của xung clock thứ 8, I2C Master sẽ ngưng tác động lên pin SDA để chờ đợi tín hiệu từ I2C Slave (tín hiệu xung ). Tại cạnh xuống của xung clock thứ 9, I2C Master sẽ lấy mẫu tín hiệu từ pin SDA để kiểm tra xem địa chỉ đã được I2C Slave nhận dạng chưa, trạng thái được đưa vào bit ACKSTAT (SSPCON2<6>). Cũng tại thời điểm cạnh xuống của xung clock thứ 9, bit BF được tự động clear, cờ ngắt SSPIF được set và BRG tạm ngưng hoạt động cho tới khi dữ liệu hoặc địa chỉ tiếp theo được đưa vào thanh ghi SSPBUF, dữ liệu hoặc địa chỉ sẽ tiếp tục được truyền đi tại cạnh xuống của xung clock tiếp theo.



Hình 4.53. Giản đồ xung I2C Master mode trong quá trình truyền dữ liệu.

Xét quá trình nhận dữ liệu ở chế độ I2C Master mode. Trước tiên ta cần set bit cho phép nhận dữ liệu RCEN (SSPCON2<3>). Khi đó BRG bắt đầu quá trình đếm, dữ liệu sẽ được dịch vào I2C Master qua pin SDA tại cạnh xuống của pin SCL. Tại cạnh xuống của xung clock thứ 8, bit cờ hiệu cho phép nhận RCEN tự động được xóa, dữ liệu trong thanh ghi SSPSR được đưa vào thanh ghi SSPBUF, cờ hiệu BF được set, cờ ngắt SSPIF được set, BRG ngưng đếm và pin SCL được đưa về mức logic thấp. Khi đó MSSP ở trạng thái tạm ngưng hoạt động để chờ đợi lệnh tiếp theo. Sau khi đọc giá trị thanh ghi SSPBUF, cờ hiệu BF tự động được xóa. Ta còn có thể gửi tín hiệu ACK bằng cách set bit ACKEN (SSPCON2<4>).



Hình 4.54. Giản đồ xung I2C Master mode trong quá trình nhận dữ liệu.

#### 4.9.2.3. Các hàm CCS

Để xem các hàm CCS hỗ trợ dung Timer chúng ta vào CCS C Compiler Help => Built-in Functions => I2C I/O.

	spi_xfer()	spi_read()	spi_write2()
<b>DISCRETE I/O</b>	get_tris_x() input() input_state() set_tris_x() input_x()	output_X() output_bit() input_change_x() output_float() output_high()	output_drive() output_low() output_toggle() set_pullup()
<b>PARALLEL PORT</b>	ppsp_input_full() ppsp_overflow()	ppsp_output_full() setup_ppsp(option, address_mask)	
<b>I2C I/O</b>	i2c_isr_state() i2c_poll() i2c_read()	i2c_slaveaddr() i2c_start() i2c_stop()	i2c_write() i2c_speed()
<b>PROCESSOR CONTROLS</b>	clear_interrupt() disable_interrupts() enable_interrupts() ext_int_edge() getenv() brownout_enable()	goto_address() interrupt_active() jump_to_isr() label_address() read_bank() write_bank()	reset_cmu() restart_cause() setup_oscillator() sleep() write_bank()
<b>BIT/BYTE MANIPULATION</b>	bit_clear() bit_set() shift_left() bit_test()	make8() make16() shift_right() make32()	mul() rotate_left() swap() rotate_right()
	abs() acos() asin()	div() exp() fabs()	log() log10() modf()

Hình 4.55. Các hàm CCS hỗ trợ I2C

Để sử dụng các hàm sau trước tiên bạn phải khai báo: #use i2c (options)trong đó Options là thành phần được ngăn cách bởi dấu phẩy có thể là:

- ✓ MASTER                      Thiết lập chế độ master
- ✓ MULTI\_MASTER              Thiết lập chế độ multi\_master

- |              |   |
|--------------|---|
| ✓ SLAVE      | Thiết lập chế độ slave                        |
| ✓ SCL=pin    | Chỉ rõ chân SCL (chân này là một bit address) |
| ✓ SDA=pin    | Chỉ rõ chân SDA                               |
| ✓ ADDRESS=nn | Chỉ rõ địa chỉ của Slave                      |

**Chú ý:** chế độ địa chỉ 7-bit hoặc 10-bit Master gửi một byte trong đó 7bit sẽ là địa chỉ của Slave bít thứ 8 là bít R/W cho biết Master đọc(R/W = 1) hay ghi(R/W = 0) dữ liệu nên Slave.

- |                |  |
|----------------|--|
| ✓ FAST         | Miêu tả sử dụng tốc độ chậm baud = 400Kbs, chỉ khai báo trong Master   |
| ✓ FAST=nnnnnnn | Thiết lập tốc độ nnnnnnn hz  |
| ✓ SLOW         | Miêu tả sử dụng tốc độ chậm baud = 100Kbs, chỉ khai báo trong Master   |
| ✓ RESTART_WDT  | Restart WDT khi đang chờ I2C_READ đọc  |
| ✓ FORCE_HW     | Sử dụng các hàm phần cứng I2C (có sẵn trong phần cứng chip không cần viết ,code ngắn hơn FORCE_SW)                                 |
| ✓ FORCE_SW     | Sử dụng các hàm phần mềm I2C thanh ghi SSPCON2 sẽ không được sử dụng khi dùng option này.  |
| ✓ NOFLOAT_HIGH | Không cho phép các tín hiệu để ở mức cao, các tín hiệu điều khiển từ thấp đến cao  |
| ✓ SMBUS        | không được sử dụng bus I2C, nhưng rất tương tự   |
| ✓ STREAM=id    | Kết hợp một stream để định danh với port I2C này. Định danh này có thể khi được sử dụng trong các hàm như i2c_read hoặc i2c_write. |
| ✓ NO_STRETCH   | Không cho phép xung clock stretching   |
| ✓ MASK=nn      | Thiết lập một mặt address cho bộ phận hỗ trợ   |
| ✓ I2C1         | Thay thế SCL= và SDA= đây là các thiết lập các chân với module đầu tiên  |
| ✓ I2C2         | Thay thế SCL= và SDA= đây là các thiết lập các chân với module thứ 2   |

Tiếp theo là một vài chip cho phép:

- |                |   |
|----------------|---|
| ✓ DATA_HOLD    | gửi No ACK bao gồm I2C_READ được gọi cho các bytes data (chỉ slave) |
| ✓ ADDRESS_HOLD | gửi No ACK bao gồm I2C_READ được gọi cho các bytes data (chỉ slave) |
| ✓ SDA_HOLD     | Thời gian dữ nhỏ nhất 300ns trên SDA từ SCL tới low                 |

**Ví dụ:**

```
#use I2C(master, sda=PIN_B0, scl=PIN_B1)          //dùng i2c chế độ master, chân dữ liệu B0, Chân clock B1, sử dụng phần mềm.
```

```
#use I2C(slave,sda=PIN_C4,scl=PIN_C3 address=0xa0,FORCE_HW)//Sử dụng chế độ slave, chân data C4, Clock C3, dùng giao tiếp bằng phần cứng.
```

```
#use I2C(master, scl=PIN_B0, sda=PIN_B1, fast=450000)// //Sử dụng chế độ master, chân data B0, Clock B1, dùng giao tiếp bằng phần mềm, tốc độ 450 KBSP.
```

➤ **i2c\_isr\_state():**

➤ **i2c\_isr\_state(stream);**

⇒ Hàm này trả lại trạng thái của truyền thông của I2C trong chế độ I2C slave. Sau một ngắt SSP. Sự tăng dần giá trị trả lại với mỗi byte nhận được hoặc gửi đến. Nếu trả lại 0x00 hoặc 0x80, I2c\_READ() cần phải đọc được địa chỉ I2C mà được gửi (nó so khớp địa chỉ được cấu hình #I2C USE như vậy giá trị này có thể được bỏ qua).

**Ví dụ:**

```
#use i2c (slave,.. )
#INT_SSP
void i2c_isr() {
    state = i2c_isr_state();
    if(state >= 0x80)
        i2c_write(send_buffer[state - 0x80]);
    else if(state > 0)
        rcv_buffer[state - 1] = i2c_read();
}
```

➤ **I2C\_SlaveAddr(addr);**

➤ **I2C\_SlaveAddr(stream, addr);**

⇒ addr = 8 bit device address

stream – xác định trong phần khai báo #USE I2C

Hàm này thiết lập địa chỉ slave cho giao diện I2C trong chế độ I2C slave và chỉ sử dụng được ở chế độ slave

**Ví dụ:**

```
i2c_SlaveAddr(0x08);
i2c_SlaveAddr(i2cStream1, 0x08);
```

➤ **i2c\_start();**

➤ **i2c\_start(stream)**

➤ **i2c\_start(stream, restart)**

⇒ stream: được xác định trong phần khai báo #use I2C

Restart:

2- Khởi động lại mới thay vì bắt đầu. sẽ thêm một xung mới vào nhưng không có tác dụng gì, và một xung có tác dụng trong lần khởi động đầu tiên.

1- Thực hiện khởi động bình thường một lần duy nhất, nếu gặp i2c\_start tiếp theo sẽ không có tác dụng, và đây cũng là giá trị mặc định.

0- Sự khởi động lại được làm duy nhất nếu trình biên dịch gặp một lệnh i2c\_start không phải là i2c\_stop.

Hàm này khởi động bit start ở I2C master. Sau khi khởi động bit start, xung Clock ở mức thấp chờ đến khi lệnh I2C\_WRITE được gọi. Nếu i2c\_start được gọi trong cùng một hàm trước khi i2c\_stop được gọi thì điều kiện khởi động được thực hiện. Chú ý giao thức I2C phụ thuộc vào thiết bị Slave. Hiện tại I2C\_START chất nhận tham số cài đặt

Nếu là 1: Trình biên dịch sẽ giả thiết bus trong trạng thái dừng

Nếu là 2: Trình biên dịch sẽ xử lý I2C\_START như một sự khởi động lại, trình biên dịch

sẽ biên dịch từ I2C\_START tới I2C\_STOP cuối cùng. Ở chế độ master cờ SSPIF của master được bật nên khi lệnh i2c\_start, i2c\_stop được thực hiện, nhiệm vụ của người lập trình là phải quan sát được cờ SSPIF khi muốn thực hiện ngắt ở device master

Ví dụ:

```
i2c_start();
i2c_write(0xa0); // Device address
i2c_write(address); // Data to device
i2c_start(); // Restart
i2c_write(0xa1); // to change data direction
data=i2c_read(0); // Now read from slave
i2c_stop();
```

- **i2c\_stop();**
- **i2c\_stop(stream);**

=> stream – được xác định trong phần khai báo #USE I2C

Hàm này được sử dụng để tắt sử dụng I2C ở Master mode

Ví dụ:

```
i2c_start(); // Start condition
i2c_write(0xa0); // Device address
i2c_write(5); // Device command
i2c_write(12); // Device data
i2c_stop(); // Stop condition
```

- **data = i2c\_read();**
- **data = i2c\_read(ack);**
- **data = i2c\_read(stream, ack);**

=> Hàm này đọc một byte qua cổng I2C ở thiết bị Master. Lệnh này tạo xung clock ở chế độ Master và chờ xung ở chế độ Slave, không có thời gian chờ nào cho Slave, Sử dụng lệnh I2C\_POLL để ngăn cản trạng thái treo, sử dụng RESTART\_WDT trong #USE I2C để theo dõi sự nhập trong chế độ Slave.

Hàm này sẽ trả lại xung ACK nếu ACK=1 thì giá trị này được chấp nhận và chân SDA=1, ACK=0 thì giá trị không chấp nhận và chân SDA=0;

Trong chế độ đọc data từ Slave muốn kết hợp với lệnh i2c\_stop() để dừng giao tiếp với Slave thì lệnh cuối cùng phải như sau:

```
i2c_read(0); // xung ACK = 0
i2c_stop(); // dừng I2C
```

Ví dụ:

```
i2c_start();
i2c_write(0xa1);
data1 = i2c_read();
data2 = i2c_read();
i2c_stop();
```

- **i2c\_write (data);**

➤ **i2c\_write (stream, data);**

=> Hàm này được dùng để gửi một byte thông qua giao diện I2C ở chế độ Master nó sẽ phát xung và data, ở chế độ Slave nó sẽ chờ xung từ Master truyền đến. Không có thời gian chờ nào cung cấp bởi hàm này. Hàm này sẽ trả lại bít ACK. Nếu ACK = 0 giá trị địa chỉ Master gửi đến khớp với địa chỉ Slave và việc truyền/nhận data được thực hiện, Nếu ACK = 1 truyền/nhận không được thực hiện và ko có xung ACK. Bit LBS đầu tiên của lệnh ghi sau quyết định hướng của dữ liệu truyền (0 từ Master tới Slave). Chú ý chuẩn giao tiếp I2C phụ thuộc vào thiết bị Slave.

Chú ý:

Trong chế độ phát địa chỉ 7-bit hoặc 10-bit Master gửi một byte trong đó 7bit sẽ là địa chỉ của Slave bít thứ 8 là bít R/W cho biết Master đọc(R/W = 1) hay ghi(R/W = 0) dữ liệu nên Slave.

**Ví dụ:**

```
long cmd;
...
i2c_start(); // Start condition
i2c_write(0xa0);// Device address
i2c_write(cmd);// Low byte of command
i2c_write(cmd>>8);// High byte of command
i2c_stop(); // Stop condition
int1 ext_eeprom_ready() {
    int1 ack;
    i2c_start(); // Nếu lệnh ghi được chấp nhận
    ack = i2c_write(0xa0); // thì thiết bị sẵn sàng hoạt động giao tiếp
    i2c_stop();
    return !ack;
}

void write_ext_eeprom(EEPROM_ADDRESS address, BYTE data) {
    while(!ext_eeprom_ready());
    i2c_start();
    i2c_write((0xa0|(BYTE)(address>>7))&0xfe);
    i2c_write(address);
    i2c_write(data);
    i2c_stop();
}
```

➤ **i2c\_poll();**

➤ **i2c\_poll(stream);**

=> Hàm này dùng để hỏi vòng I2C, hàm chỉ dùng khi sử dụng SSP, hàm này trả về giá trị TRUE khi nhận được một giá trị trong bộ đệm, nếu hàm này trại lại giá trị TRUE nếu dùng hàm i2c\_read sẽ đọc được giá trị chuyển đến. Chỉ sử dụng hàm này trong chế độ I2C Master.

Hàm này muốn sử dụng trong chế độ Master phải khai báo sử dụng các hàm phần cứng NOFOCE\_SW hoặc FORCE\_HW trong #use. Trong chế độ Slave mặc định dùng các hàm này ko cần phải khai báo.

**Chú ý:**

Khi sử dụng các pin khác thay thế cho chức năng của RC3(SCL) và RC4(SDA) bắt buộc phải dùng tới các hàm phần mềm FORCE\_SW, khi dùng FORCE\_SW thì không thể dùng i2c\_poll(); ta sẽ dùng câu lệnh này để thay thế hàm i2c\_poll():

```
#use i2c(MASTER,SDA=PIN_c2,SCL=PIN_c1,force_sw)
#bit ACKSTAT=0x91.6
void readdata()
{
    i2c_start();
    i2c_write(0xa1);
    while(ACKSTAT);
    data=i2c_read(0);
    i2c_stop();
}
```

**Ví dụ:**

```
#use i2c (master,.. )
i2c_start(); // Start condition
i2c_write(0xc1); // Device address/Read
count=0;
while(count!=4) {
    while(!i2c_poll());
    buffer[count++]= i2c_read(); //Read Next
}
i2c_stop(); // Stop condition
```

**4.10. Tổng quan về một số đặc tính của CPU****4.10.1. CONFIGURATION BIT**

Đây là các bit dùng để lựa chọn các đặc tính của CPU. Các bit này được chứa trong bộ nhớ chương trình tại địa chỉ 2007h và chỉ có thể được truy xuất trong quá trình lập trình cho vi điều khiển. Chi tiết về các bit này như sau:

R/P-1	U-0	R/P-1	U-0	U-0	R/P-1	R/P-1	R/P-1	R/P-1						
CP	—	DEBUG	WRT1	WRT0	CPD	LVP	BOREN	—	—	PWRTE	WDTEN	Fosc1	Fosc0	bit0

Bit 13 CP: (Code Protection)

1: tắt chế độ bảo vệ mã chương trình.

0: bật chế độ bảo vệ mã chương trình.

Bit 12, 5, 4: không quan tâm và được mặc định mang giá trị 0.

Bit 11 DEBUG (In-circuit debug mode bit)

1: không cho phép, RB7 và RB6 được xem như các pin xuất nhập bình thường.

0: cho phép, RB7 và RB6 là các pin được sử dụng cho quá trình debug.

Bit 10-9 WRT1:WRT0 Flash Program Memory Write Enable bit

11: Tắt chức năng chống ghi, EECON sẽ điều khiển quá trình ghi lên toàn bộ nhớ chương trình.

10: chỉ chống từ địa chỉ 0000h: 00FFh.

01: chỉ chống ghi từ địa chỉ 0000h: 07FFh.

00: chỉ chống ghi từ địa chỉ 0000h: 0FFFh.

Bit 8 CPD Data EEPROM Memory Write Protection bit

1: Tắt chức năng bảo vệ mã của EEPROM.

0: Bật chức năng bảo vệ mã.

Bit 7 LVP Low-Voltage (Single supply) In-Circuit Serial Programming Enable

1: Cho phép chế độ nạp điện áp thấp, pin RB3/PGM được sử dụng cho chế độ này.

0: Không cho phép chế độ nạp điện áp thấp, điện áp cao được đưa vào từ pin , pin RB3 là pin I/O bình thường.

Bit 6 BODEN Brown-out Reset Enable bit

1: cho phép BOR (Brown-out Reset)

0: không cho phép BOR.

Bit 3 Power-up Timer Enable bit

1: không cho phép PWR.

0: cho phép PWR.

Bit 2 WDTEN Watchdog Timer Enable bit

1: cho phép WDT.

0: không cho phép WDT. Bit 1-0 FOSC1:FOSC0 lựa chọn loại oscillator

Bit 1-0: Oscillator Selection bits

11: sử dụng RC oscillator.

10: sử dụng HS oscillator.

01: sử dụng XT oscillator.

00: sử dụng LP oscillator.

Chi tiết về các đặc tính Oscillator sẽ được đề cập cụ thể trong phần tiếp theo.

#### 4.10.2. CÁC ĐẶC TÍNH CỦA OSCILLATOR

PIC16F877A có khả năng sử dụng một trong 4 loại oscillator, đó là:

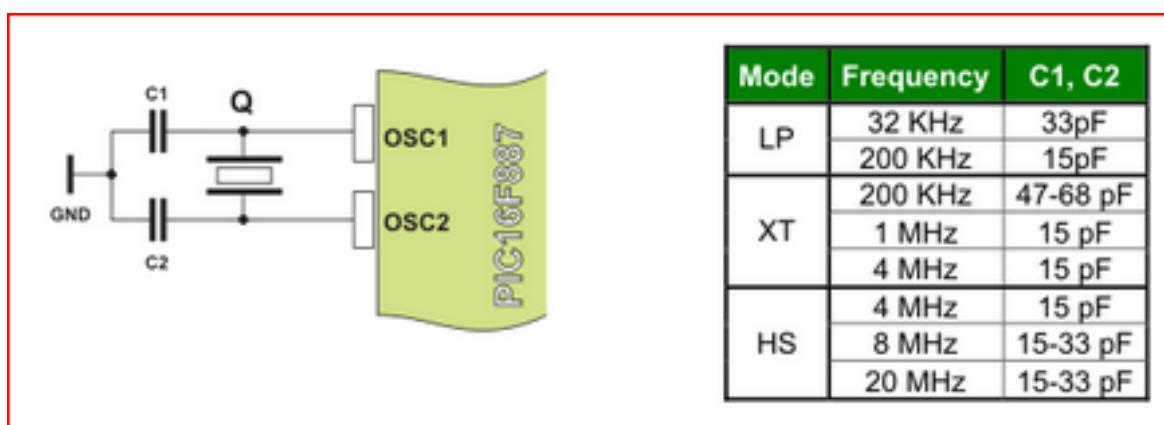
LP: (Low Power Crystal).

XT: Thạch anh bình thường.

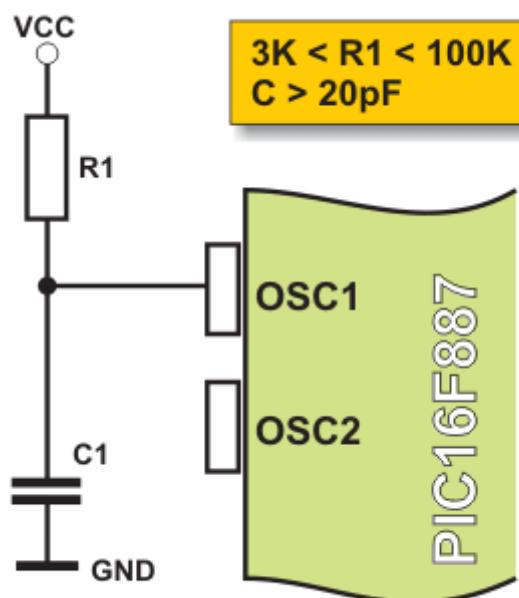
HS: (High-Speed Crystal).

RC: (Resistor/Capacitor) dao động do mạch RC tạo ra.

Đối với các loại oscillator LP, HS, XT, oscillator được gắn vào vi điều khiển thông qua các pin OSC1/CLKI và OSC2/CLKO.



Đối với các ứng dụng không cần các loại oscillator tốc độ cao, ta có thể sử dụng mạch dao động RC làm nguồn cung cấp xung hoạt động cho vi vi điều khiển. Tần số tạo ra phụ thuộc vào các giá trị điện áp, giá trị điện trở và tụ điện, bên cạnh đó là sự ảnh hưởng của các yếu tố như nhiệt độ, chất lượng của các linh kiện.



Hình 4.56. RC oscillator.

Các linh kiện sử dụng cho mạch RC oscillator phải bao đảm các giá trị sau:

$$3 \text{ K} < R_{EXT} < 100 \text{ K}$$

$$C_{EXT} > 20 \text{ pF}$$

#### 4.10.3. Các chế độ Reset

Có nhiều chế độ reset vi điều khiển, bao gồm: Power-on Reset POR (Reset khi cấp nguồn hoạt động cho vi điều khiển).

MCLR reset trong quá trình hoạt động.

MCLR từ chế độ sleep.

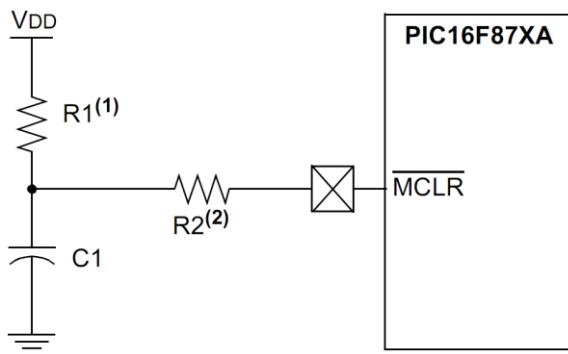
WDT reset (reset do khói WDT tạo ra trong quá trình hoạt động).

WDT wake up từ chế độ sleep.

Brown-out reset (BOR).

Ngoài trừ reset POR trạng thái các thanh ghi là không xác định và WDT wake up không ảnh hưởng đến trạng thái các thanh ghi, các chế độ reset còn lại đều đưa giá trị các thanh ghi về giá trị ban đầu được ấn định sẵn. Các bit và chỉ thị trạng thái hoạt động, trạng thái reset của vi điều khiển và được điều khiển bởi CPU.

**MCLR** reset: Khi pin **MCLR** ở mức logic thấp, vi điều khiển sẽ được reset. Tín hiệu reset được cung cấp bởi một mạch ngoại vi với các yêu cầu cụ thể sau: Không nối pin **MCLR** trực tiếp lên nguồn VDD. R1 phải nhỏ hơn 40K để đảm bảo các đặc tính điện của vi điều khiển. R2 phải lớn hơn 1 K để hạn dòng đi vào vi điều khiển.



Hình 4.57. Mạch reset qua pin.

**MCLR** reset còn được chống nhiễu bởi một bộ lọc để tránh các tín hiệu nhỏ tác động lên pin **MCLR**.

**Power-on reset (POR):** Đây là xung reset do vi điều khiển tạo ra khi phát hiện nguồn cung cấp VDD. Khi hoạt động ở chế độ bình thường, vi điều khiển cần được đảm bảo các thông số về dòng điện, điện áp để hoạt động bình thường. Nhưng nếu các tham số này không được đảm bảo, xung reset do POR tạo ra sẽ đưa vi điều khiển về trạng thái reset và chỉ tiếp tục hoạt động khi nào các tham số trên được đảm bảo.

**Power-up Timer (PWRT):** đây là bộ định thời hoạt động dựa vào mạch RC bên trong vi điều khiển. Khi PWRT được kích hoạt, vi điều khiển sẽ được đưa về trạng thái reset. PWRT sẽ tạo ra một khoảng thời gian delay (khoảng 72 ms) để VDD tăng đến giá trị thích hợp.

**Oscillator Start-up Timer (OST):** OST cung cấp một khoảng thời gian delay bằng 1024 chu kỳ xung của oscillator sau khi PWRT ngưng tác động (vi điều khiển đã đủ điều kiện hoạt động) để đảm bảo sự ổn định của xung do oscillator phát ra. Tác động của OST còn xảy ra đối với POR reset và khi vi điều khiển được đánh thức từ chế độ sleep. OST chỉ tác động đối với các loại oscillator là XT, HS và LP.

**Brown-out reset (BOR):** Nếu VDD hạ xuống thấp hơn giá trị VBOR (khoảng 4V) và kéo dài trong khoảng thời gian lớn hơn TBOR (khoảng 100 us), BOR được kích hoạt và vi điều khiển được đưa về trạng thái BOR reset. Nếu điện áp cung cấp cho vi điều khiển hạ xuống thấp hơn VBOR trong khoảng thời gian ngắn hơn TBOR, vi điều khiển sẽ không được reset. Khi điện áp cung cấp đủ cho vi điều khiển hoạt động, PWRT được kích hoạt để tạo ra một khoảng thời gian delay (khoảng 72ms). Nếu trong khoảng thời gian này điện áp cung cấp cho vi điều khiển lại tiếp tục hạ xuống dưới mức điện áp VBOR, BOR reset sẽ lại được kích hoạt khi vi điều khiển đủ điện áp hoạt động. Một điểm cần chú ý là khi BOR reset được cho phép, PWRT cũng sẽ hoạt động bất chấp trạng thái của bit PWRT.

U-0	U-0	U-0	U-0	U-0	U-0	R/W-0	R/W-1
—	—	—	—	—	—	$\overline{\text{POR}}$	$\overline{\text{BOR}}$

bit 7

bit 0

Tóm lại để vi điều khiển hoạt động được từ khi cấp nguồn cần trải qua các bước sau: POR tác động. PWRT (nếu được cho phép hoạt động) tạo ra khoảng thời gian delay TPWRT để ổn định nguồn cung cấp. OST (nếu được cho phép) tạo ra khoảng thời gian delay bằng 1024 chu kỳ xung của oscillator để ổn định tần số của oscillator. Đến thời điểm này vi điều khiển mới bắt đầu hoạt động bình thường. Thanh ghi điều khiển và chỉ thị trạng thái nguồn cung cấp cho vi điều khiển là thanh ghi PCON.

Bit 7, 6, 5, 4, 3, 2 Không cần quan tâm và mặc định mang giá trị 0.

Bit 1  $\overline{\text{POR}}$  Power-on Reset Status bit

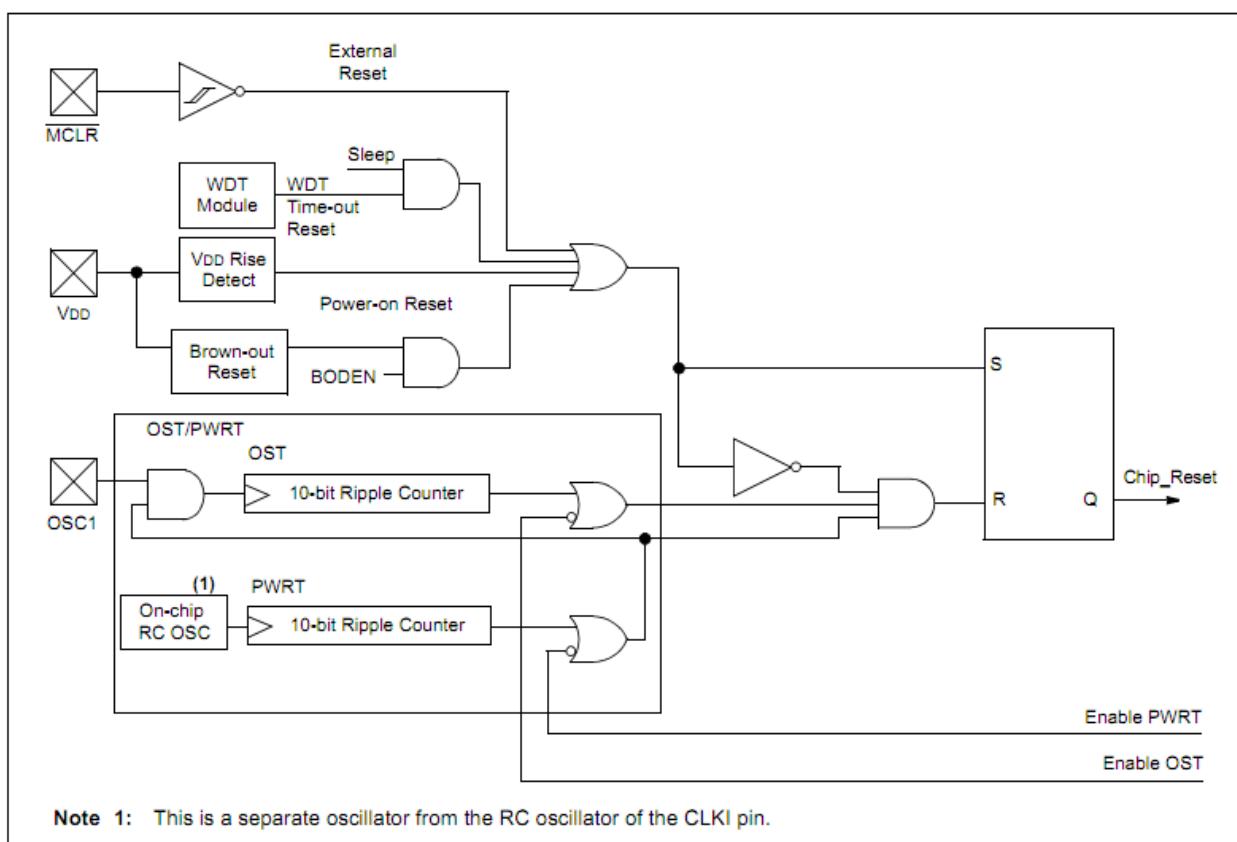
= 1 không có sự tác động của Power-on Reset.

= 0 có sự tác động của Power-on reset.

Bit 0  $\overline{\text{BOR}}$  Brown-out Reset Status bit

= 1 không có sự tác động của Brown-out reset.

= 0 có sự tác động của Brown-out reset.



Hình 4.58. Sơ đồ các chế độ reset của PIC16F877A.

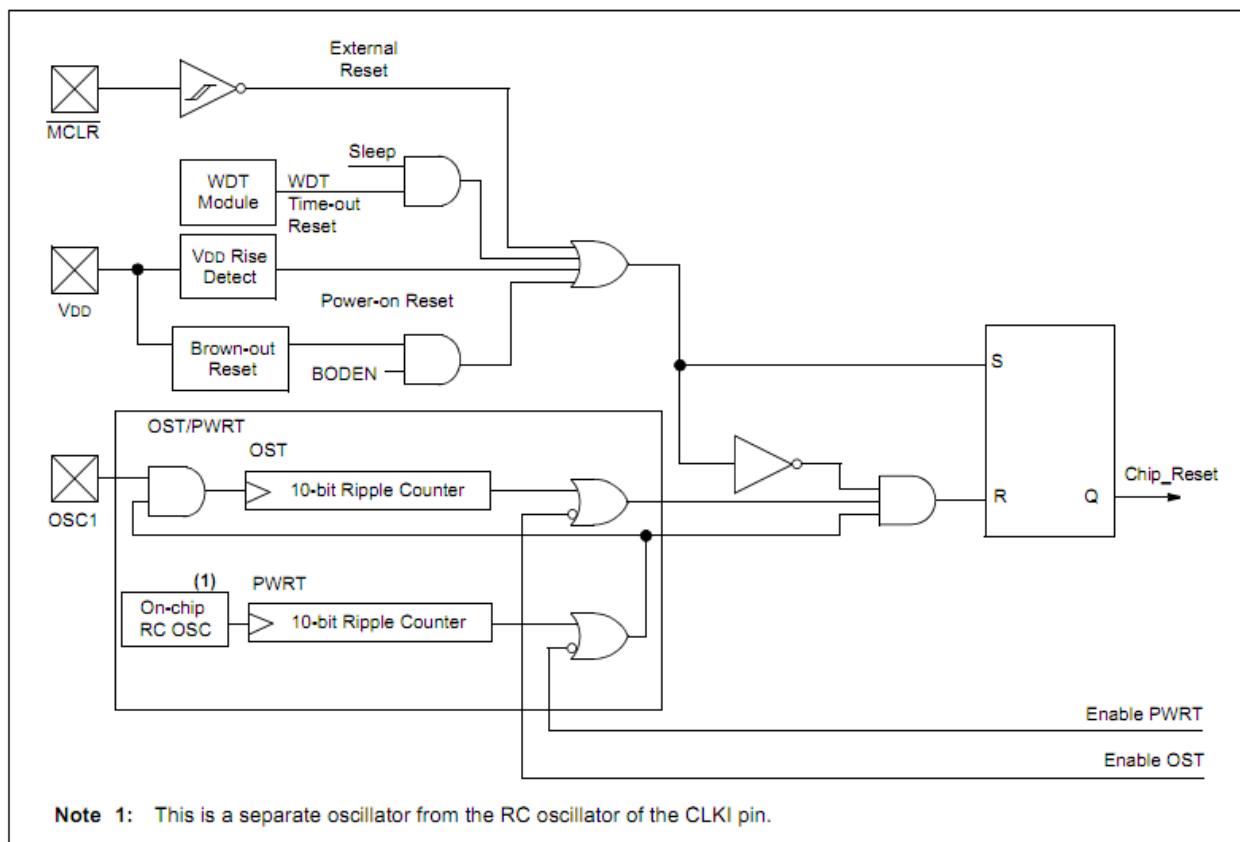
#### 4.10.4. Ngắt (Interrupt)

PIC16F877A có đến 15 nguồn tạo ra hoạt động ngắt được điều khiển bởi thanh ghi INTCON (bit GIE). Bên cạnh đó mỗi ngắt còn có một bit điều khiển và cờ ngắt riêng. Các cờ ngắt vẫn được set bình thường khi thỏa mãn điều kiện ngắt xảy ra bất chấp trạng thái của bit GIE, tuy nhiên hoạt động ngắt vẫn phụ thuộc vào bit GIE và các bit điều khiển khác. Bit điều khiển ngắt RB0/INT và TMR0 nằm trong thanh ghi INTCON, thanh ghi này còn chứa bit cho phép các ngắt

ngoại vi PEIE. Bit điều khiển các ngắt nằm trong thanh ghi PIE1 và PIE2. Cờ ngắt của các ngắt nằm trong thanh ghi PIR1 và PIR2. Trong một thời điểm chỉ có một chương trình ngắt được thực thi, chương trình ngắt được kết thúc bằng lệnh RETFIE. Khi chương trình ngắt được thực thi, bit GIE tự động được xóa, địa chỉ lệnh tiếp theo của chương trình chính được cất vào bộ nhớ Stack và bộ đếm chương trình sẽ chỉ đến địa chỉ 0004h. Lệnh RETFIE được dùng để thoát khỏi chương trình ngắt và quay trở về chương trình chính, đồng thời bit GIE cũng sẽ được set để cho phép các ngắt hoạt động trở lại. Các cờ hiệu được dùng để kiểm tra ngắt nào đang xảy ra và phải được xóa bằng chương trình trước khi cho phép ngắt tiếp tục hoạt động trở lại để ta có thể phát hiện được thời điểm tiếp theo mà ngắt xảy ra.

Đối với các ngắt ngoại vi như ngắt từ chân INT hay ngắt từ sự thay đổi trạng thái các pin của PORTB (PORTB Interrupt on change), việc xác định ngắt nào xảy ra cần 3 hoặc 4 chu kỳ lệnh tùy thuộc vào thời điểm xảy ra ngắt.

Cần chú ý là trong quá trình thực thi ngắt, chỉ có giá trị của bộ đếm chương trình được cất vào trong Stack, trong khi một số thanh ghi quan trọng sẽ không được cất và có thể bị thay đổi giá trị trong quá trình thực thi chương trình ngắt. Điều này nên được xử lý bằng chương trình để tránh hiện tượng trên xảy ra.



Hình 4.59. Sơ đồ logic của tất cả các ngắt trong vi điều khiển PIC16F877A.

#### 4.10.4.1. Ngắt INT

Ngắt này dựa trên sự thay đổi trạng thái của pin RB0/INT. Cạnh tác động gây ra ngắt có thể là cạnh lên hay cạnh xuống và được điều khiển bởi bit INTEDG (thanh ghi OPTION\_REG <6>). Khi có cạnh tác động thích hợp xuất hiện tại pin RB0/INT, cờ ngắt INTF được set bắt cháp trạng thái các bit điều khiển GIE và PEIE. Ngắt này có khả năng đánh thức vi điều khiển từ chế độ sleep nếu bit cho phép ngắt được set trước khi lệnh SLEEP được thực thi.

#### 4.10.4.2. Ngắt do sự thay đổi trạng thái các pin trong PORTB

Các pin PORTB<7:4> được dùng cho ngắt này và được điều khiển bởi bit RB (thanh ghi INTCON<4>). Cờ ngắt của ngắt này là bit RBIF (INTCON<0>).

#### 4.10.4.3. WatchDog Timer (WDT)

Watchdog timer (WDT) là bộ đếm độc lập dùng nguồn xung đếm từ bộ tạo xung được tích hợp sẵn trong vi điều khiển và không phụ thuộc vào bất kì nguồn xung clock ngoại vi nào. Điều đó có nghĩa là WDT vẫn hoạt động ngay cả khi xung clock được lấy từ pin OSC1/CLKI và pin OSC2/CLKO của vi điều khiển ngưng hoạt động (chẳng hạn như do tác động của lệnh sleep). Bit điều khiển của WDT là bit WDTE nằm trong bộ nhớ chương trình ở địa chỉ 2007h (Configuration bit). WDT sẽ tự động reset vi điều khiển (Watchdog Timer Reset) khi bộ đếm của WDT bị tràn (nếu WDT được cho phép hoạt động), đồng thời bit tự động được xóa. Nếu vi điều khiển đang ở chế độ sleep thì WDT sẽ đánh thức vi điều khiển (Watchdog Timer Wake-up) khi bộ đếm bị tràn. Như vậy WDT có tác dụng reset vi điều khiển ở thời điểm cần thiết mà không cần đến sự tác động từ bên ngoài, chẳng hạn như trong quá trình thực thi lệnh, vi điều khiển bị “kẹt” ở một chỗ nào đó mà không thoát ra được, khi đó vi điều khiển sẽ tự động được reset khi WDT bị tràn ê chương trình hoạt động đúng trở lại. Tuy nhiên khi sử dụng WDT cũng có sự phiền toái vì vi điều khiển sẽ thường xuyên được reset sau một thời gian nhất định, do đó cần tính toán thời gian thích hợp để xóa WDT (dùng lệnh CLRWDT). Và để việc xác định thời gian reset được linh động, WDT còn được hỗ trợ một bộ chia tần số prescaler được điều khiển bởi thanh ghi OPTION\_REG (prescaler này được chia sẻ với Timer0).

Một điểm cần chú ý nữa là lệnh sleep sẽ xóa bộ đếm WDT và prescaler. Ngoài ra lệnh xóa CLRWDT chỉ xóa bộ đếm chứ không làm thay đổi đối tượng tác động của prescaler (WDT hay Timer0).

Xem lại Timer0 và thanh ghi OPTION\_REG để biết thêm chi tiết.

#### 4.10.4.4. Chế độ Sleep

Đây là chế độ hoạt động của vi điều khiển khi lệnh SLEEP được thực thi. Khi đó nếu được cho phép hoạt động, bộ đếm của WDT sẽ bị xóa nhưng WDT vẫn tiếp tục hoạt động, bit (STATUS<3>) được reset về 0, bit được set, oscillator ngưng tác động và các PORT giữ nguyên trạng thái như trước khi lệnh SLEEP được thực thi. Do khi ở chế độ SLEEP, dòng cung cấp cho vi điều khiển là rất nhỏ nên ta cần thực hiện các bước sau trước khi vi điều khiển thực thi lệnh SLEEP:

Đưa tất cả các pin về trạng thái VDD hoặc VSS

Cần bảo đảm rằng không có mạch ngoại vi nào được điều khiển bởi dòng điện của vi điều khiển vì dòng điện nhỏ không đủ khả năng cung cấp cho các mạch ngoại vi hoạt động.

Tạm ngưng hoạt động củ khối A/D và không cho phép các xung clock từ bên ngoài tác động vào vi điều khiển.

Để ý đến chức năng kéo lên điện trở ở PORTB.

Pin MCLR phải ở mức logic cao.

#### “ĐÁNH THỨC” VI ĐIỀU KHIỂN:

Vi điều khiển có thể được “đánh thức” dưới tác động của một trong số các hiện tượng sau:

1. Tác động của reset ngoại vi thông qua pin .
2. Tác động của WDT khi bị tràn.
3. Tác động từ các ngắt ngoại vi từ PORTB (PORTB Interrupt on change hoặc pin INT).

Các bit và được dùng để thể hiện trạng thái của vi điều khiển và để phát hiện nguồn tác động làm reset vi điều khiển. Bit được set khi vi điều khiển được cấp nguồn và được reset về 0 khi vi điều khiển ở chế độ sleep. Bit được reset về 0 khi WDT tác động do bộ đếm bị tràn. Ngoài ra còn có một số nguồn tác động khác từ các chức năng ngoại vi bao gồm:

- ✓ Đọc hay ghi dữ liệu thông qua PSP (Parallel Slave Port).
- ✓ Ngắt Timer1 khi hoạt động ở chế độ đếm bắt đồng bộ.
- ✓ Ngắt CCP khi hoạt động ở chế độ Capture.
- ✓ Các hiện tượng đặc biệt làm reset Timer1 khi hoạt động ở chế độ đếm bắt đồng bộ dùng nguồn xung clock ở bên ngoài).
- ✓ Ngắt SSP khi bit Start/Stop được phát hiện.
- ✓ SSP hoạt động ở chế độ Slave mode khi truyền hoặc nhận dữ liệu.
- ✓ Tác động của USART từ các pin RX hay TX khi hoạt động ở chế độ Slave mode đồng bộ.
- ✓ Khối chuyển đổi A/D khi nguồn xung clock hoạt động ở dạng RC.
- ✓ Hoàn tất quá trình ghi vào EEPROM.
- ✓ Ngõ ra bộ so sánh thay đổi trạng thái.

Các tác động ngoại vi khác không có tác dụng đánh thức vi điều khiển vì khi ở chế độ sleep các xung clock cung cấp cho vi điều khiển ngưng hoạt động. Bên cạnh đó cần cho phép các ngắt hoạt động trước khi lệnh SLEEP được thực thi để bảo đảm tác động của các ngắt. Việc đánh thức vi điều khiển từ các ngắt vẫn được thực thi bát chấp trạng thái của bit GIE. Nếu bit GIE mang giá trị 0, vi điều khiển sẽ thực thi lệnh tiếp theo sau lệnh SLEEP của chương trình (vì chương trình ngắt không được cho phép thực thi). Nếu bit GIE được set trước khi lệnh SLEEP được thực thi, vi điều khiển sẽ thực thi lệnh tiếp theo của chương trình và sau đó nhảy tới địa chỉ chúa chương trình ngắt (0004h). Trong trường hợp lệnh tiếp theo không đóng vai trò quan trọng trong chương trình, ta cần đặt thêm lệnh NOP sau lệnh SLEEP để bỏ qua tác động của lệnh này, đồng thời giúp ta dễ dàng hơn trong việc kiểm soát hoạt động của chương trình ngắt. Tuy nhiên cũng có một số điểm cần lưu ý như sau: Nếu ngắt xảy ra trước khi lệnh SLEEP được thực thi, lệnh SLEEP sẽ không được thực thi và thay vào đó là lệnh NOP, đồng thời các tác động của lệnh SLEEP cũng sẽ được bỏ qua. Nếu ngắt xảy ra trong khi hay sau khi lệnh SLEEP được thực thi, vi điều khiển lập tức được đánh thức từ chế độ sleep, và lệnh SLEEP sẽ được thực thi ngay sau khi vi điều khiển được đánh thức. Để kiểm tra xem lệnh SLEEP đã được thực thi hay chưa, ta kiểm tra bit. Nếu bit vẫn mang giá trị 1 tức là lệnh SLEEP đã không được thực thi và thay vào đó là lệnh NOP. Bên cạnh đó ta cần xóa WDT để chắc chắn rằng WDT đã được xóa trước khi thực thi lệnh SLEEP, qua đó cho phép ta xác định được thời điểm vi điều khiển được đánh thức do tác động của WDT.

#### 4.10.4.5. Các hàm CCS phục vụ ngắt:

Mỗi dòng vi điều khiển có số lượng nguồn ngắt khác nhau: PIC 16 có 16 ngắt, PIC 18 có 35 ngắt.

Muốn biết CCS hỗ trợ những ngắt nào cho VDK của bạn, mở file \*.h tương ứng , ở cuối file là danh sách các ngắt mà CCS hỗ trợ nó. Cách khác là vào CCS -> View -> Valid interrupts, chọn VDK muốn xem , nó sẽ hiển thị danh sách ngắt có thể có cho VDK đó.

Sau đây là danh sách 1 số ngắt với chức năng tương ứng:

- ✓ #INT\_GLOBAL: ngắt chung, nghĩa là khi có ngắt xảy ra, hàm theo sau chỉ thị này được thực thi, bạn sẽ không được khai báo thêm chỉ thị ngắt nào khác khi sử dụng chỉ thị này. CCS không sinh bất kỳ mã lưu nào, hàm ngắt bắt đầu ngay tại vector ngắt. Nếu bật nhiều cờ cho phép ngắt, có thể bạn sẽ phải hỏi vòng để xác định ngắt nào. Dùng chỉ thị này tương đương viết hàm ngắt 1 cách thủ công mà thôi , như là viết hàm ngắt với ASM vậy .
- ✓ #INT\_AD : chuyển đổi A /D đã hoàn tất, thường thì không nên dùng
- ✓ #INT\_ADOF : I don't know
- ✓ #INT\_BUSCOL : xung đột bus
- ✓ #INT\_BUTTON : nút nhấn.
- ✓ #INT\_CCP1 : có Capture hay compare trên CCP1
- ✓ #INT\_CCP2 : có Capture hay compare trên CCP2
- ✓ #INT\_COMP : kiểm tra bằng nhau trên Comparator
- ✓ #INT\_EEPROM : hoàn thành ghi EEPROM
- ✓ #INT\_EXT : ngắt ngoài s
- ✓ #INT\_EXT1 : ngắt ngoài 1
- ✓ #INT\_EXT2 : ngắt ngoài 2
- ✓ #INT\_I2C : có hoạt động I 2C
- ✓ #INT\_LCD : có hoạt động LCD
- ✓ #INT\_LOWWOLT : phát hiện áp thấp
- ✓ #INT\_PSP : có data vào cổng Parallel slave
- ✓ #INT\_RB : bất kỳ thay đổi nào trên chân B4 đến B7
- ✓ #INT\_RC : bất kỳ thay đổi nào trên chân C4 đến C7
- ✓ #INT\_RDA : data nhận từ RS 232 sẵn sàng
- ✓ #INT\_RTCC : tràn Timer 0
- ✓ #INT\_SSP : có hoạt động SPI hay I 2C
- ✓ #INT\_TBE : bộ đếm chuyển RS 232 trống
- ✓ #INT\_TIMER0 : một tên khác của #INT\_RTCC
- ✓ #INT\_TIMER1 : tràn Timer 1
- ✓ #INT\_TIMER2 : tràn Timer 2
- ✓ #INT\_TIMER3 : tràn Timer 3
- ✓ #INT\_TIMER5 : tràn Timer 5
- ✓ #INT\_OSCF : lỗi OSC
- ✓ #INT\_PWMKB : ngắt của PWM time base
- ✓ #INT\_IC3DR : ngắt đổi hướng ( direct ) của IC 3
- ✓ #INT\_IC2QEI : ngắt của QEI
- ✓ #INT\_IC1 : ngắt IC 1

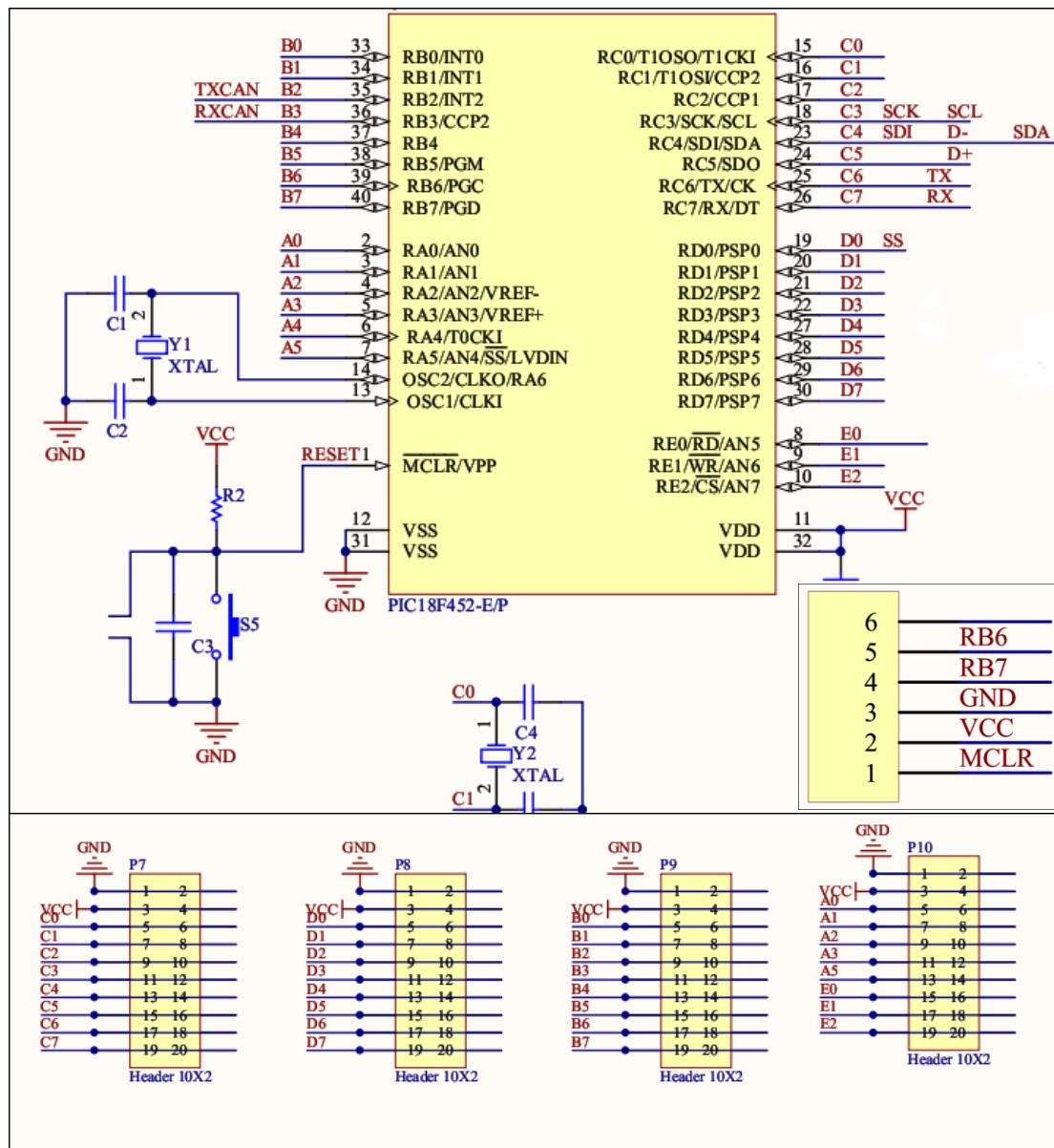
Hàm đi kèm phục vụ ngắt không cần tham số vì không có tác dụng.

Sử dụng NOCLEAR sau #int\_xxx để CCS không xoá cờ ngắt của hàm đó.

## CHƯƠNG V: MẠCH ĐIỆN ÚNG DỤNG

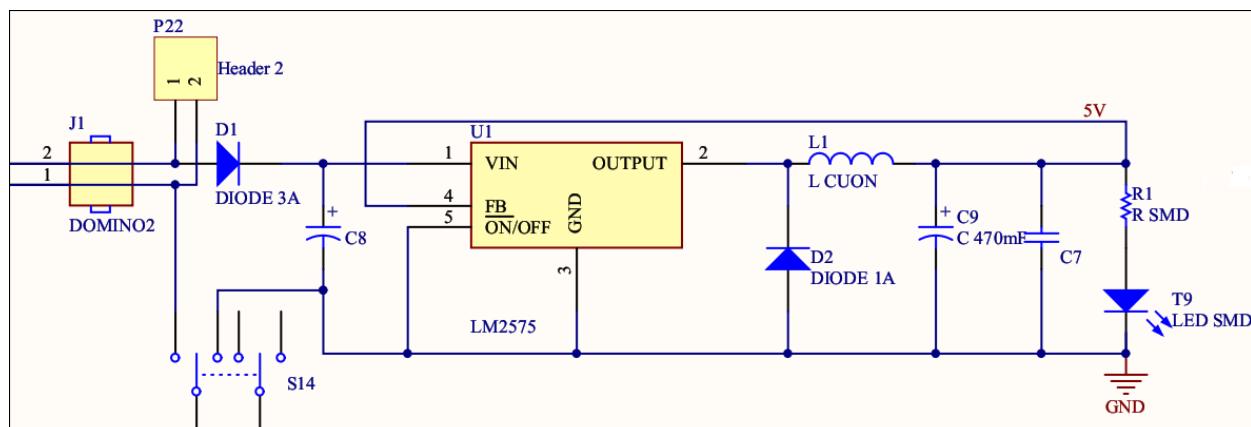
### 5.1. Sơ đồ vi điều khiển, mạch nạp:

Board mạch đưa ra các header dùng cho các ứng dụng mở rộng, cổng nạp isp... dùng cho dùng pic 16fxxx và 18fxxx.

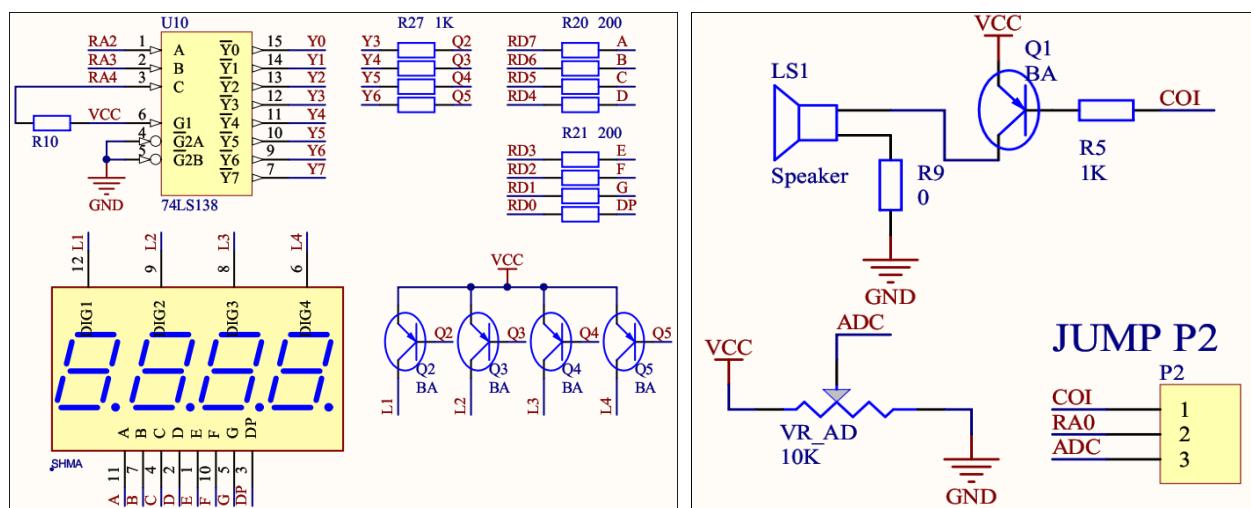


### 5.2. Sơ đồ khối nguồn sử dụng LM2578 (5V-3A)

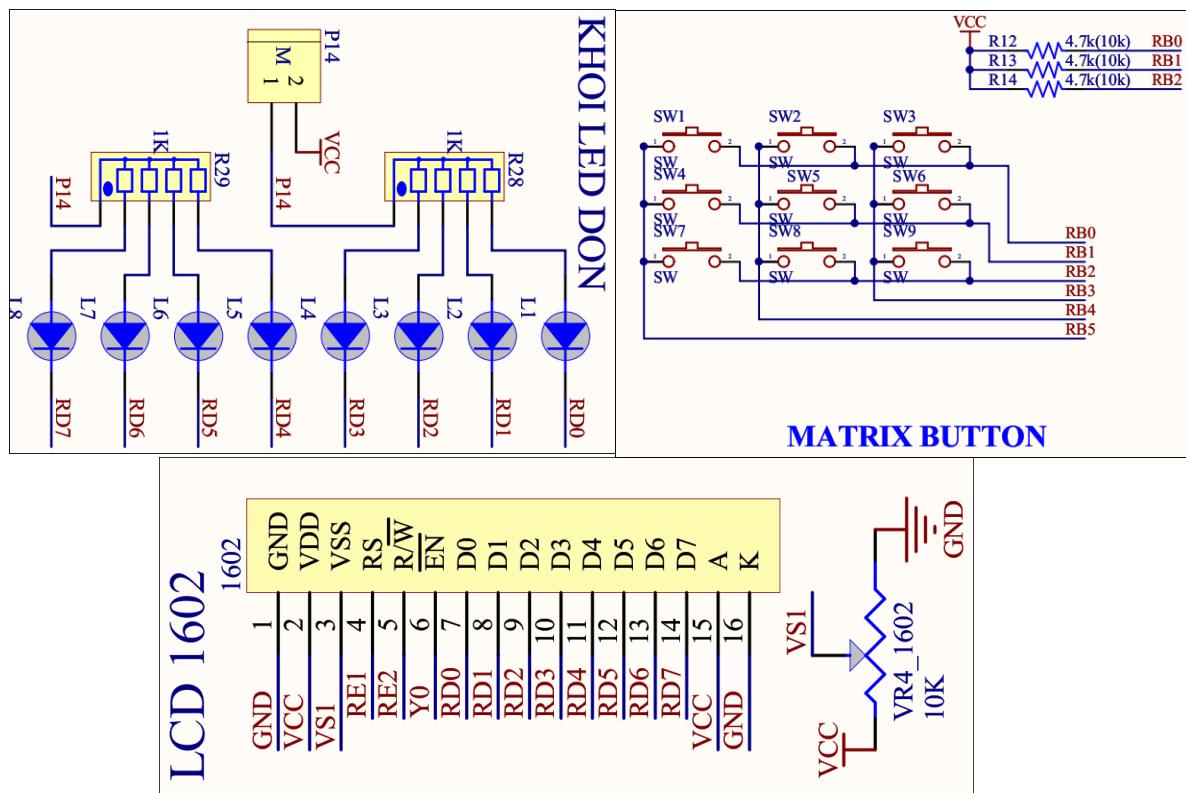
IC LM2578 là ic chuyên dụng cho điện áp ổn định, dòng cao (3A), tần số đáp ứng cao.



### 5.3. Sơ đồ kết nối led 7 đoạn, ADC, Còi:

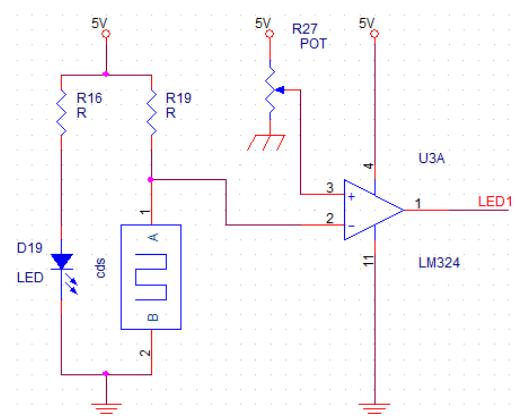


### 5.4. Khối Led đơn, Keypad 3x3, LCD1602:



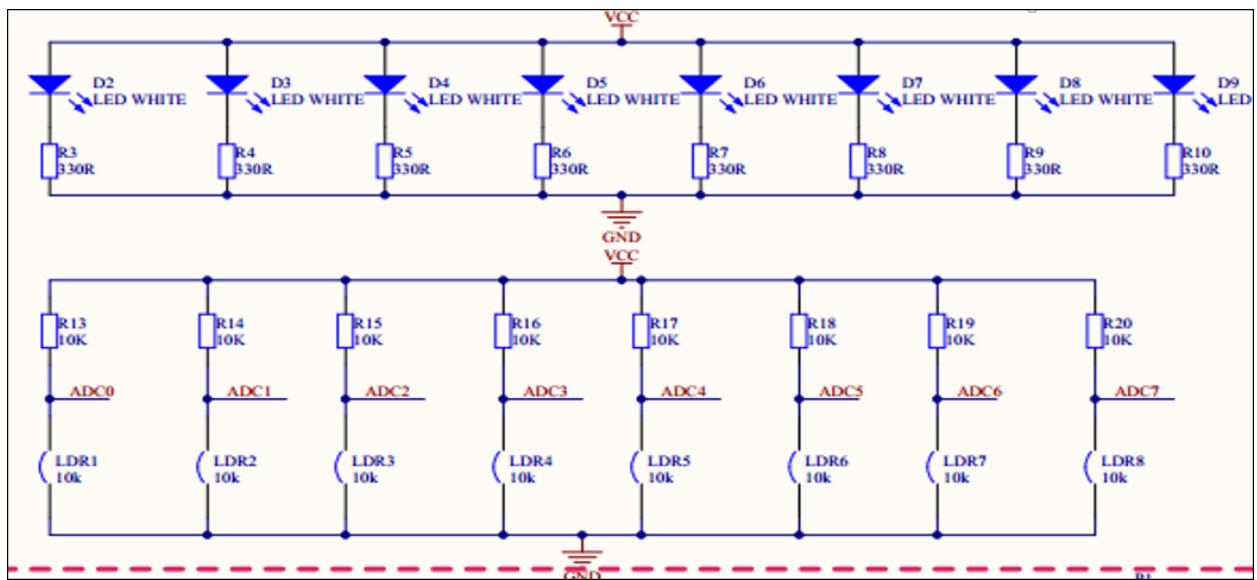
### 5.5. Mạch dò line dùng trong robot dùng opamp

Sử dụng opamp để so sánh điện áp trả về từ quang trỏ. Từ đó đưa ra tín hiệu vào vi điều khiển

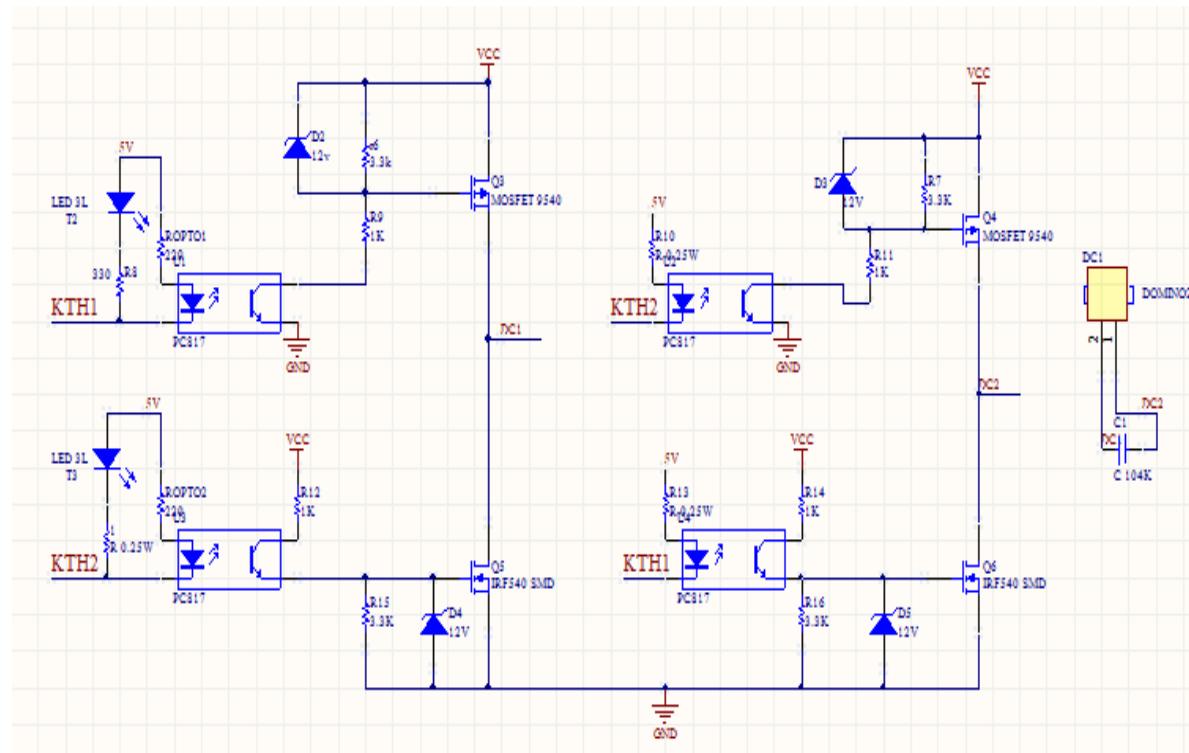


### 5.6. Mạch dò line trong robot dùng vi điều khiển:

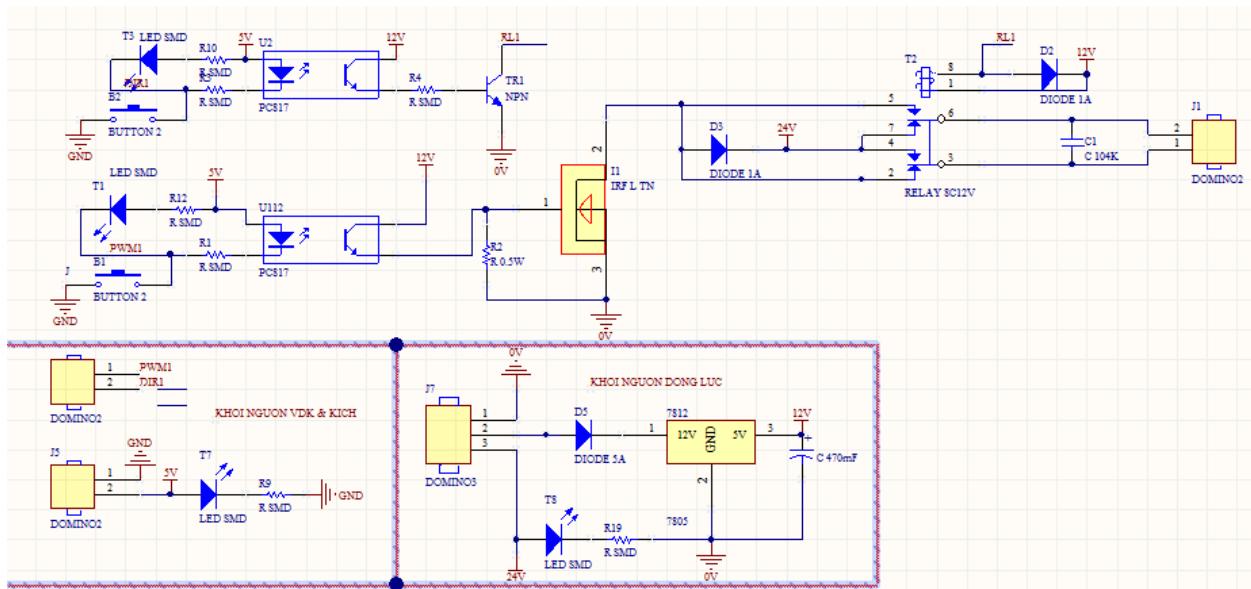
Tín hiệu ADCX sẽ đưa vào các chân ANx của vi điều khiển.



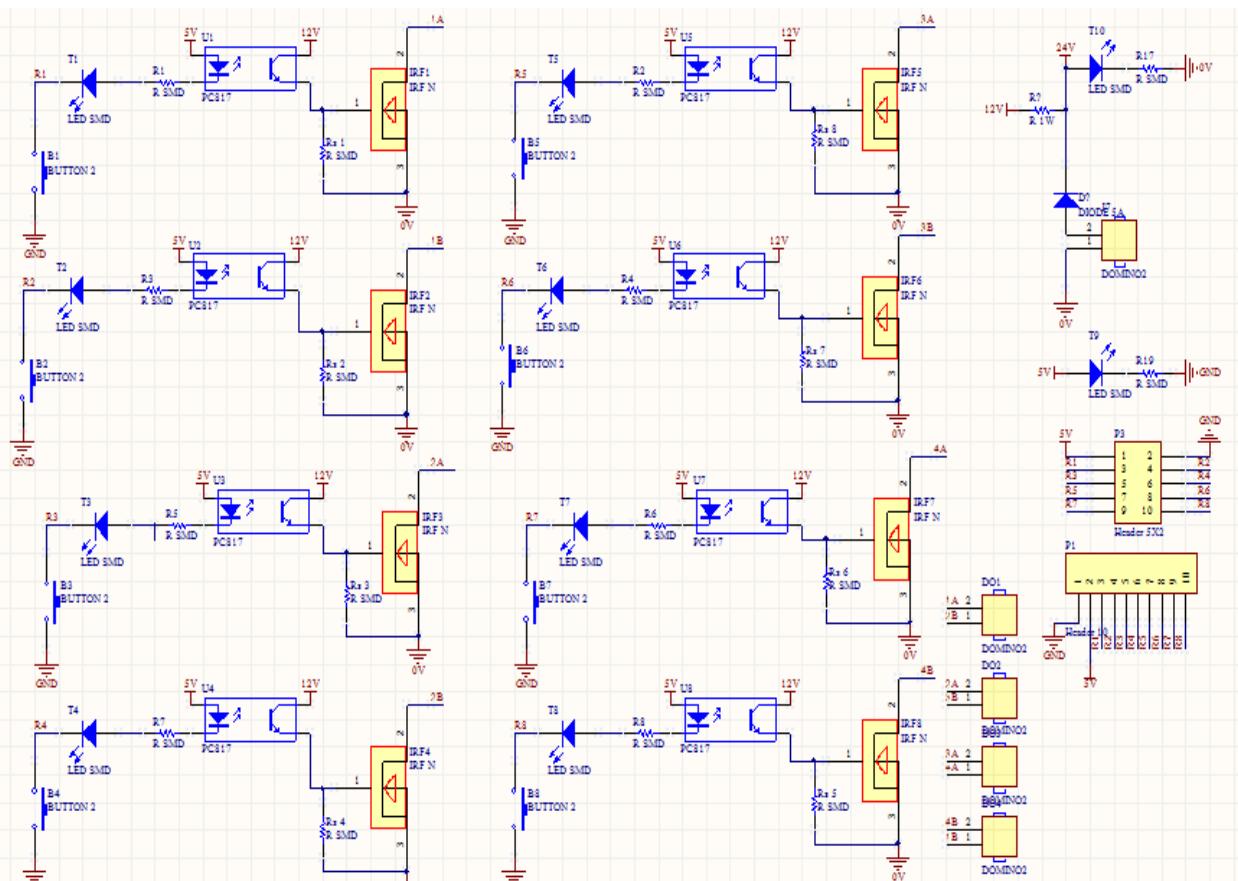
### 5.7. Mạch điều khiển động cơ DC cầu H dùng IRF9540-540:



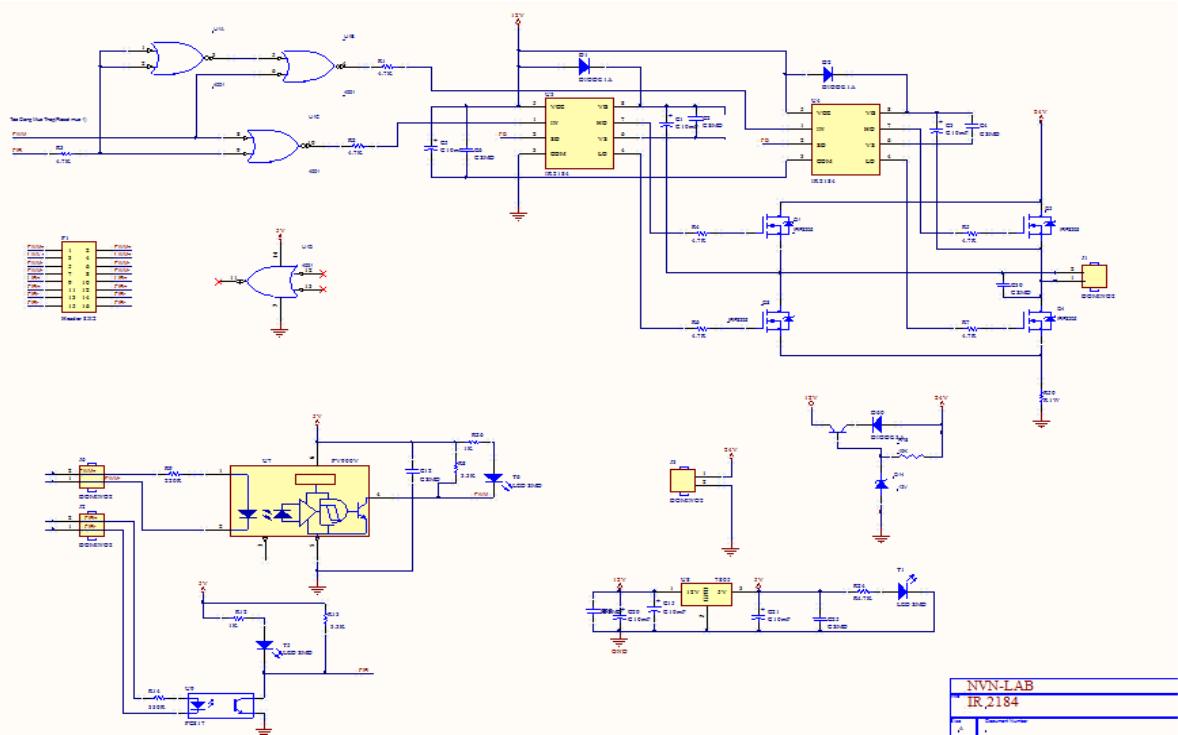
### 5.8. Mạch điều khiển động cơ dùng Fet-Relay:



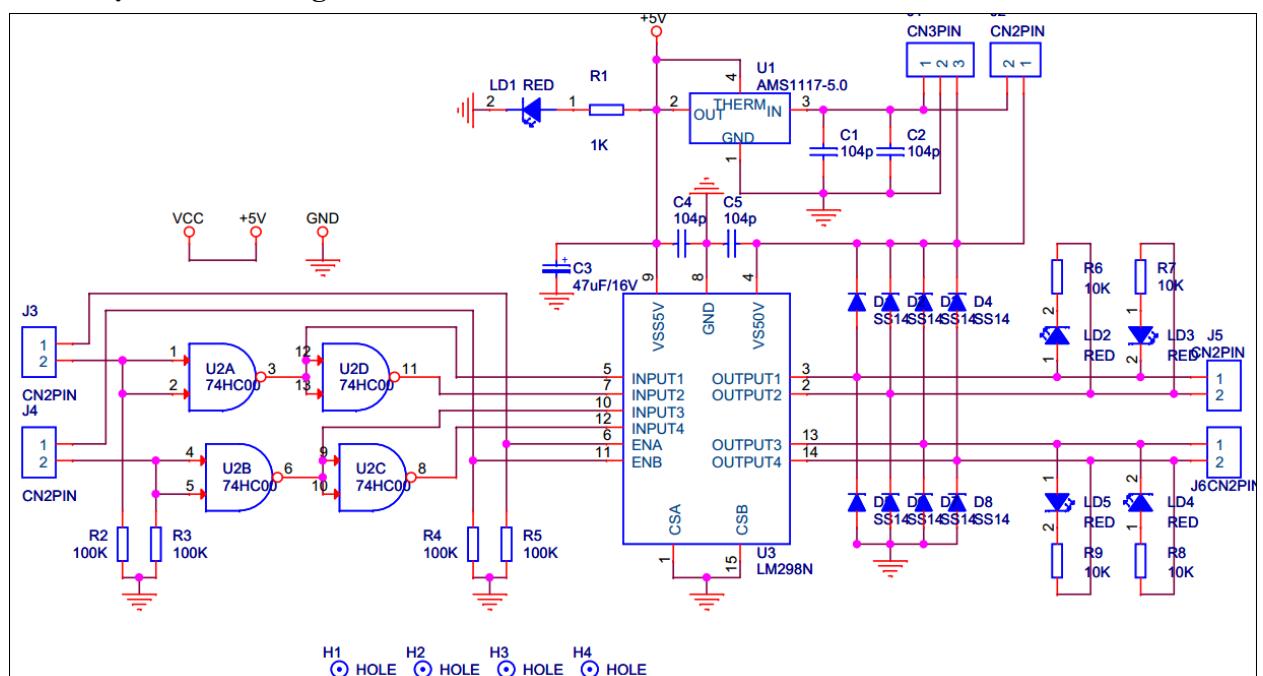
### 5.9. Mạch điều khiển xilanh, led vẫy...:



### 5.10. Mạch điều khiển động cơ dung ir2184:



### 5.11. Mạch cầu H Dung L298

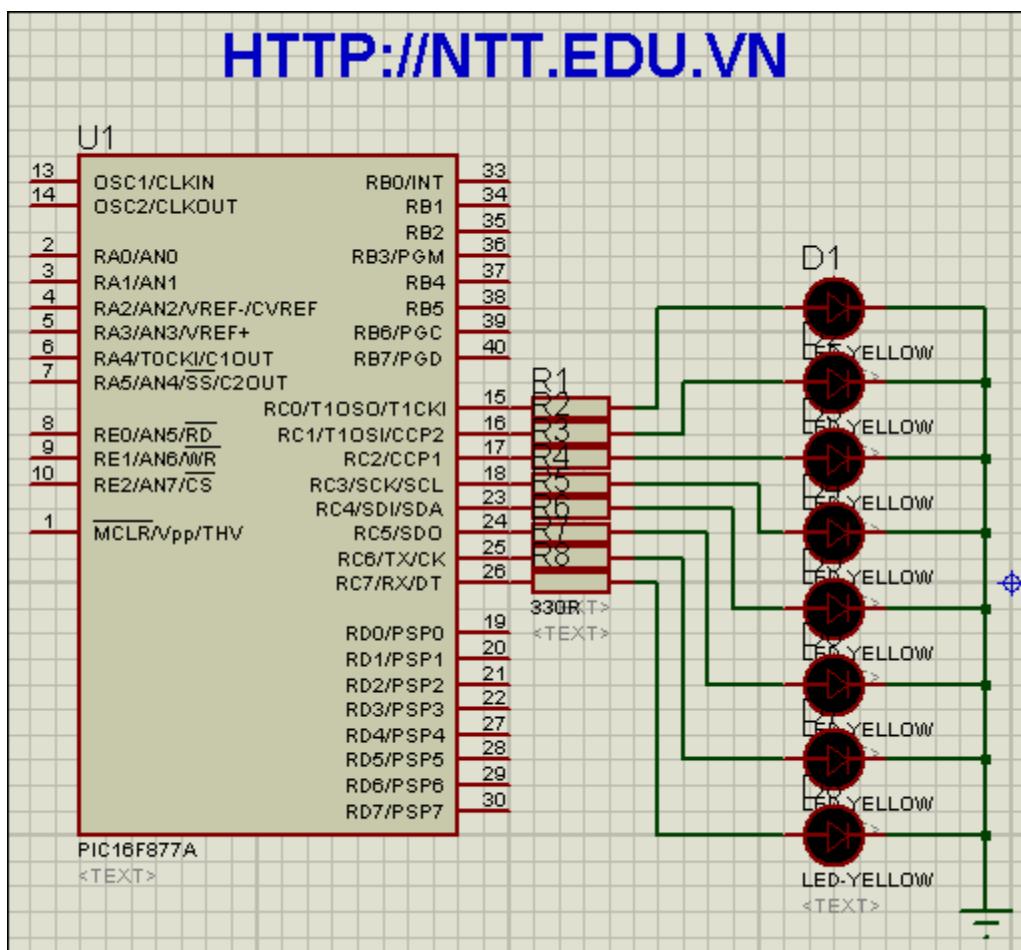


**CHƯƠNG VI: BÀI TẬP ỨNG DỤNG**

Ở phần bài tập này tôi sẽ không trình bày lại header miêu tả mà chỉ để phần keywords của proteus cho các bạn dễ lấy linh kiện. Nhưng khi viết các bạn nên viết thêm phần này cho dễ nâng cấp, chỉnh sửa sau này. Các file include trong code được trình bày ở phần phụ lục.

**6.1. Viết chương trình nháy led PORTC, sang 1s, tắt 1s.**

Phần cứng mô phỏng:



**Code:**

```
*****
```

```
*
```

```
* PIC Training Course
```

```
* Nguyen Tat Thanh University
```

```
*
```

```
*****
```

```
*****
```

```
*
```

```
* Module      : main.c
```

```
* Description : Display led portc.
```

```
* Tool        : ccs v5.00xx
```

\* Chip : 16F877A  
\* History : 7/2011  
\* Version : v1.0  
\* Author : Nguyen Huu Luan  
\* Mail : nvn.nhl@gmail.com  
\* Website : ntt.edu.vn  
\* Notes :  
\*\*\*\*\*\*/

```
//////////  
// Su dung portc output //  
//  
// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh //  
//  
//////////  
/*  
* Keywords Proteus  
* Lay vdk: PIC16F877A  
* Lay led don: Led-green, led-red....  
* Lay dien tro: Resistor  
*/  
  
*****  
* IMPORT  
*****  
#include "16f877A.h"  
#include "def_877a.h"  
#fuses NOWDT,NOPROTECT,PUT,NOLVP,HS  
#use delay(clock = 4000000) //Tan so thach anh 4MHz  
  
void main()  
{  
    setup_adc_ports(NO_ANALOGS|VSS_VDD);  
    setup_adc(ADC_OFF);  
    setup_spi(SPI_SS_DISABLED);  
    setup_timer_0(T0_DISABLED);  
    setup_timer_1(T1_DISABLED);  
    setup_timer_2(T2_DISABLED,0,1);  
    setup_comparator(NC_NC_NC_NC); // This device COMP currently not supported by the  
PICWizard  
//Setup_Oscillator parameter not selected from Intr Oscillotar Config tab  
  
set_tris_c(0x00); //PORTC xuat du lieu.  
PORTC = 0x00; //Cac led deu tat.
```

```
while(1)
{
    PORTC = ~PORTC;
    delay_ms(1000);
}
```

### 6.2. Viết chương trình sang tắt Led PORTC 5 lần, mỗi lần 2s.

- ✓ Phản ứng mô phỏng: xem bài 6.1.

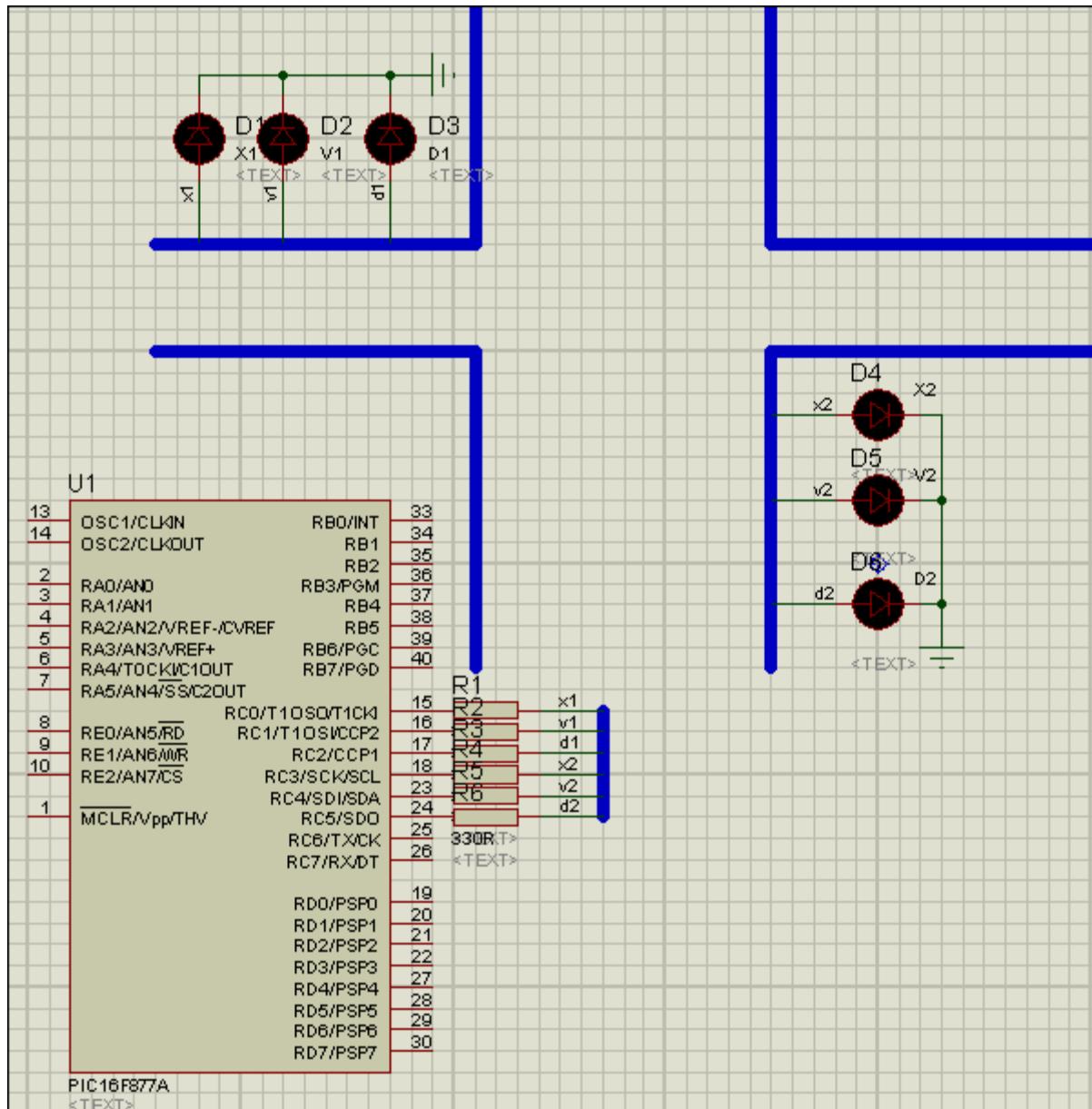
- ✓ **Code:**

```
#include "16f877A.h"
#include "def_877a.h"
#fuses NOWDT,NOPROTECT,PUT,NOLVP,HS
#use   delay(clock = 4000000)      //Tần số thạch anh 4MHz
int8 k;
void main()
{
    setup_adc_ports(NO_ANALOGS|VSS_VDD);
    setup_adc(ADC_OFF);
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_1(T1_DISABLED);
    setup_timer_2(T2_DISABLED,0,1);
    setup_comparator(NC_NC_NC_NC); // This device COMP currently not supported by the
PICWizard
//Setup_Oscillator parameter not selected from Intr Oscillotar Config tab

set_tris_c(0x00); //PORTC xuất đường.
PORTC = 0x00; //Các led đều tắt.
for(k=0;k<10;k++) //Led sáng và tắt 5 lần.
{
    PORTC = ~PORTC; //Đảo trạng thái các led.
    delay_ms(2000);
}
while(1); //VXL không làm gì nữa.
}
```

### 6.3. Viết chương trình đèn giao thông ngã 4, đèn xanh 30s, đèn vàng 5s.

- ✓ Phản ứng mô phỏng:



✓ Code:

```
#include "16f887.h"
#include "def_16f887.h"
#fuses NOWDT,NOPROTECT,PUT,NOLVP,HS
#use   delay(clock = 4000000)      //Tan so thach anh 4MHz
#define SW RA4
unsigned char const X1V2 = 0x11;
unsigned char const X1D2 = 0x21;
unsigned char const V1X2 = 0x0a;
unsigned char const D1X2 = 0x0c;

void main()
{
    setup_adc_ports(NO_ANALOGS|VSS_VDD);
    setup_adc(ADC_OFF);
```

```

setup_spi(SPI_SS_DISABLED);
setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
setup_timer_1(T1_DISABLED);
setup_timer_2(T2_DISABLED,0,1);
setup_comparator(NC_NC_NC_NC); // This device COMP currently not supported by
the PICWizard
//Setup_Oscillator parameter not selected from Intr Oscillotar Config tab
set_tris_c(0x00); //PORTC xuat du lieu.
PORTC = 0x03; //2 led trai sang.
while(1)
{
    PORTC = X1V2;
    delay_ms(5000);
    //-----
    PORTC = X1D2;
    delay_ms(25000);
    //-----
    PORTC = V1X2;
    delay_ms(5000);
    //-----
    PORTC = D1X2;
    delay_ms(25000);
    //-----
}
}

```

#### 6.4. Viết chương trình led sang dần trên PORTC.

- ✓ **Mô phỏng phần cứng:** Xem bài 6.1.
- ✓ **Code:**

```

#include "16f887.h"
#include "def_16f887.h"
#fuses NOWDT,NOPROTECT,PUT,NOLVP,HS
#use   delay(clock = 4000000) //Tần số thạch anh 4MHz
int8 k;
void main()
{
    setup_adc_ports(NO_ANALOGS|VSS_VDD);
    setup_adc(ADC_OFF);
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_1(T1_DISABLED);

```

```

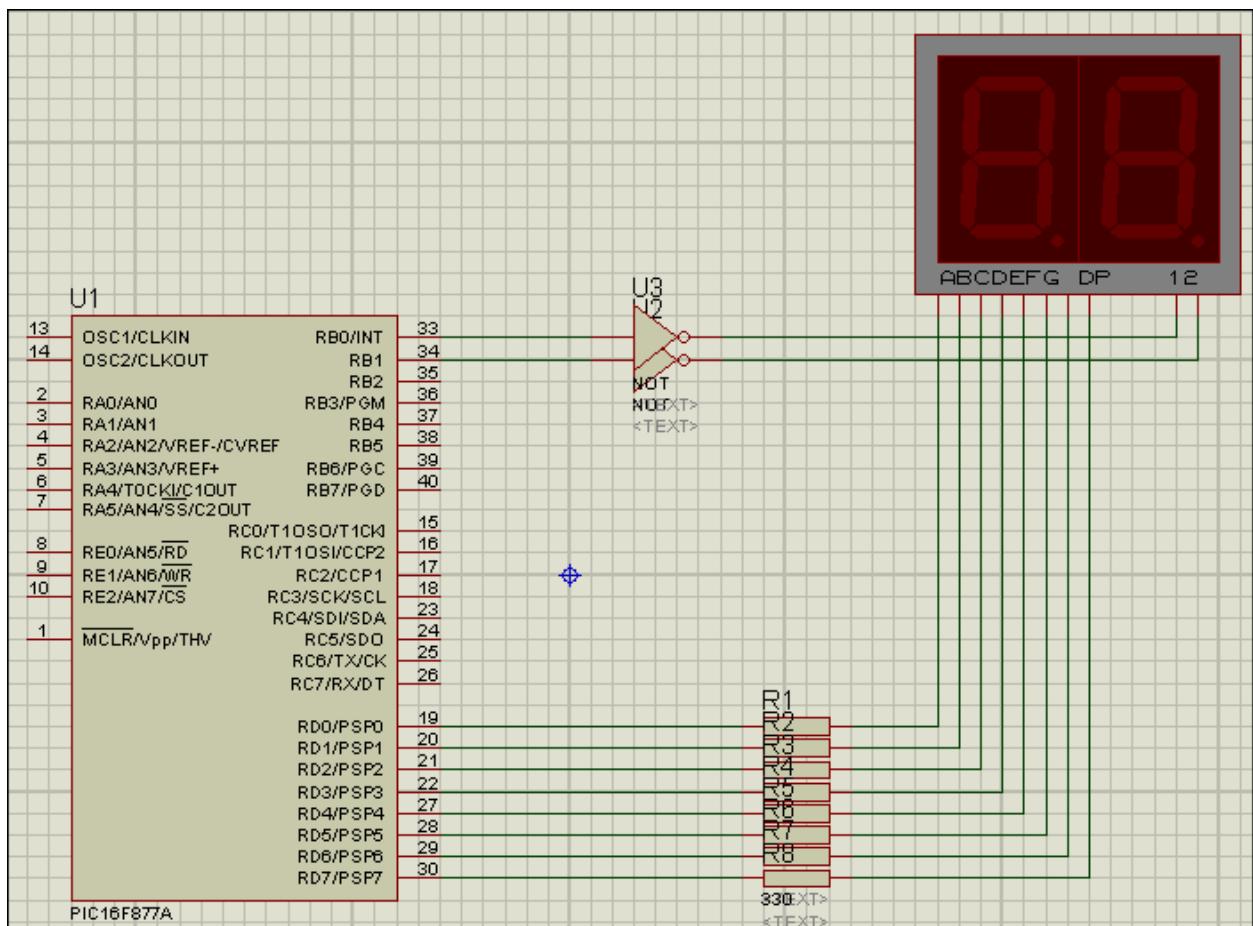
setup_timer_2(T2_DISABLED,0,1);
setup_comparator(NC_NC_NC_NC); // This device COMP currently not supported by
the PICWizard
//Setup_Oscillator parameter not selected from Intr Oscillotar Config tab

set_tris_c(0x00); //PORTC xuat du lieu.
PORTC = 0x00; //Cac led deu tat.
while(1)
{
    delay_ms(1000); //Delay nhin thay cac led deu tat.
    PORTC = (PORTC<<1)|0x01; //Led sang dan.
    if(PORTC == 0xff)
    {
        delay_ms(1000); //Delay nhin thay cac led deu sang.
        PORTC = 0x00;
    }
}
}

```

### 6.6. Viết chương trình hiển thị số 0=>>99 trên 2 led 7 đoạn dùng anôt chung.

- ✓ Mô tả phần cứng mô phỏng:



✓ **Code:**

```

///////////////////////////////
// Su dung portd output          //
//                                //
// Chuong trinh thiet ke boi Khoa Co Khi Truong DH Nguyen Tat Thanh // //
//                                //
///////////////////////////////
/*
 * Keywords Proteus
 * Lay vdk: PIC16F877A
 * Lay led 7 doan Anot chung: 7SEG
 * Lay dien tro: Resistor
 * Lay cong not (Thay transisto) : Not
 */

#include <16F877A.h>
#include <def_877a.h> //file header do nguoi dung dinh nghia
#fuses HS,NOWDT,NOPROTECT,NOLVP
#use    delay(clock=20000000)

//dinh nghia chan quet led
#define led0  rb1
#define led1  rb0

//dinh nghia portd dua data
#define data_led  portd

//char dig[]={192, 249, 164, 176, 153, 146, 130, 248, 128, 144}; // mang chua gia tri led 7
daon
char dig[]={0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};

void display0_99(int8 x){
    int8 i,j;
    // tach lay hang chu
    i = x/10;
    led1 = 0;
    data_led = dig[i];//=>>portd = dig[i];
    delay_ms(10);
    led1 = 1;
    // tach lay hang don vi
    i = x%10;// i=1
    led0 = 0;
    data_led = dig[i];
    delay_ms(10);
}

```

```

    led0 = 1;
}

void main()
{
    int8 i;
    // TODO: USER CODE!
    trisD = 0x00; //output
    portD = 0xff; //off led

    TRISB = 0x00; //output
    portB = 0xff; //off led

    while(true)
    {
        for(i=0;i<99,i++)
        {
            display0_99(75);
            delay_ms(500);
        }
    }
}

//end of main program

```

### 6.7. Viết chương trình nháy led nhiều chế độ trên portD

- ✓ Phần cứng mô phỏng giống bài 6.1. đổi portC=>portD.
- ✓ Code tham khảo:

```

#include <16F877A.h>
#include <def_877a.h>
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT,
NOLVP, NOCPD, NOWRT
#use delay(clock=20000000)

```

```

int8 mode,i;
byte temp;
void program1();
void program2();
void program3();
void program4();
void program5();

```

```
void program6();
void program7();
void program8();

void main() {
    trisd = 0x00; //output portd
    portd=0xff;   //Led is off
    mode = 0;
    while (1) {
        switch(mode) {
            case 0: program0(); break;
            case 1: program1(); break;
            case 2: program2(); break;
            case 3: program3(); break;
            case 4: program4(); break;
            case 5: program5(); break;
            case 6: program6(); break;
            case 7: program7(); break;
            case 8: program8(); break;
        }
        mode++;
        if (mode==9) mode = 0;
    }
}

Void program0(){
    //chuong trinh 1 led sang dan tu phai sang
    // for(i=0;i<=8;i++)
    // {
    //     b=e<<0;
    //     PORTD=b;
    //     delay_ms(3000);
    // };

    //chuong trinh 1 led sang tu trai sang
    for(i=0;i<=8;i++)
    {
        b=z>>i;
        PORTD=b;
        delay_ms(300);
    };

    /*dich led tu ngoai vao trong,dich led tu trong ra ngoai*/
    for(i=0;i<=8;i++)
    {
```

```
b=e<<i;  
c=z>>i;  
PORTD=b|c;  
delay_ms(300);  
};  
//chuong trinh led tat dan tu phai sang trai  
for(i=0;i<=8;i++)  
{  
    PORTD=a<<i;  
    delay_ms(300);  
};  
//chuong trinh led tat dan tu trai samh phai  
for(i=0;i<=8;i++)  
{  
    PORTD=a>>i;  
    delay_ms(300);  
};  
//chuong trinh led tat dan tu giua ra  
for(i=4;i<=8;i++)  
{  
    b=a>>i;  
    c=a<<i;  
    PORTD=b|c;  
    delay_ms(300);  
};  
//sang dan tu ngoai vao roi tat dan tu giua ra  
for(i=0;i<=8;i++)  
{  
    b=a>>i;  
    c=a<<i;  
    PORTD=b^c;  
    delay_ms(300);  
};  
//chuong trinh tat dan tu 2 ben vao giua  
for(i=0;i<=4;i++)  
{  
    b=a>>i;  
    c=a<<i;  
    PORTD=b&c;  
    delay_ms(300);  
};  
}
```

```
void program1() {
    PortD = 0x00;
    delay_ms(250);
    Portd = 0xFF;
    delay_ms(250);
}
void program2() {
    temp = 0xFF;
    for (i=0;i<=8;i++) {
        portd = temp;
        delay_ms(250);
        temp >= 1;
    }
}
void program3() {
    temp = 0xFF;
    for (i=0;i<=8;i++) {
        portd = temp;
        delay_ms(250);
        temp <= 1;
    }
}
void program4() {
    portd = 0xAA;
    delay_ms(500);
    portd = 0x55;
    delay_ms(500);
}
void program5() {
    Portd = 0x7E;
    delay_ms(150);
    Portd = 0xBD;
    delay_ms(250);
    Portd = 0xDB;
    delay_ms(150);
    Portd = 0xE7;
    delay_ms(150);
    Portd = 0xDB;
    delay_ms(150);
    Portd = 0xBD;
    delay_ms(150);
    Portd = 0x7E;
    delay_ms(150);
}
```

```
void program6() {
    temp = 0xFF;
    for (i=0;i<=8;i++) {
        portd = temp;
        delay_ms(250);
        temp = temp >> 1;
    }
}

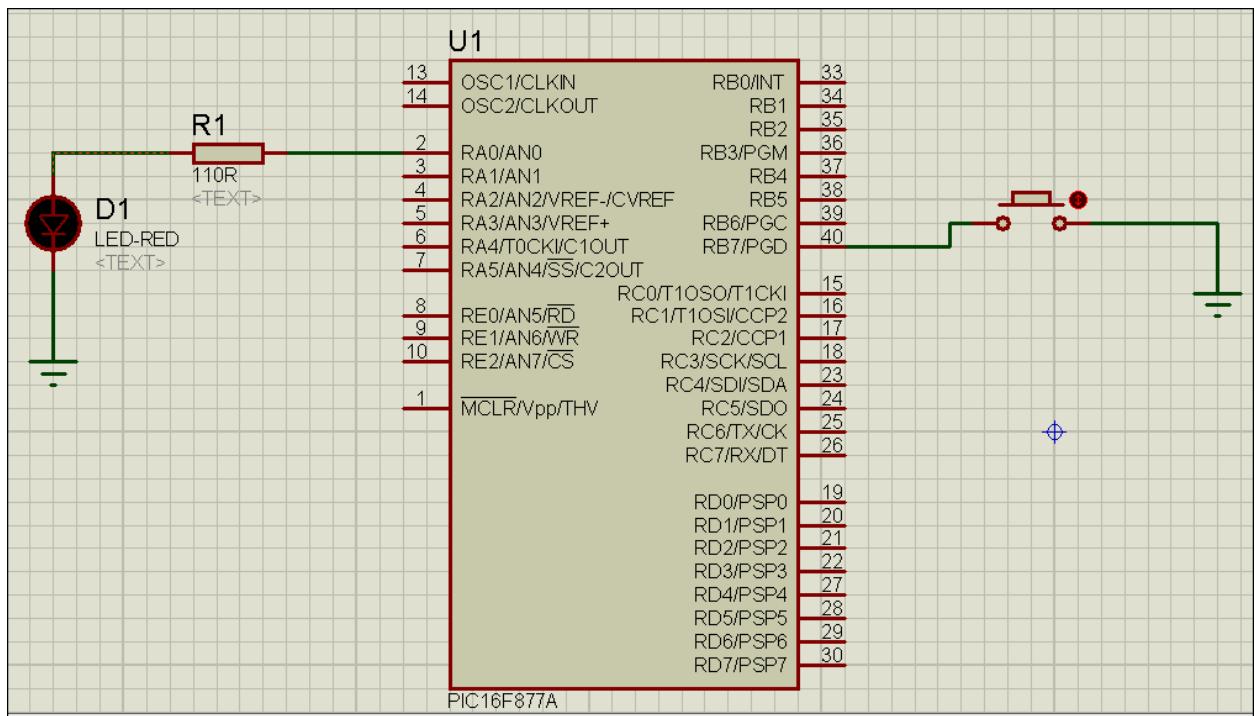
void program7() {

    Portd = 0xFE;
    delay_ms(150);
    Portd = 0xFD;
    delay_ms(150);
    Portd = 0xFB;
    delay_ms(150);
    Portd = 0xF7;
    delay_ms(150);
    Portd = 0xEF;
    delay_ms(150);
    PortD = 0xDF;
    delay_ms(150);
    Portd = 0xBF;
    delay_ms(150);
    Portd = 0x7F;
    delay_ms(150);
}

void program8() {
    Portd = 0x7F;
    delay_ms(150);
    Portd = 0xBF;
    delay_ms(150);
    PortD = 0xDF;
    delay_ms(150);
    Portd = 0xEF;
    delay_ms(150);
    Portd = 0xF7;
    delay_ms(150);
    Portd = 0xFB;
    delay_ms(150);
    Portd = 0xFD;
    delay_ms(150);
    Portd = 0xFE;
```

```
    delay_ms(150);
}
```

### 6.8. Viết chương trình kiểm tra nút nhấn PIN B7. Khi nhấn led RA0 tắt, ngược lại RA0 sáng.



✓ Mô tả phần cứng:

✓ Code:

```
/*
 * Keywords Proteus
 * Lay vdk: PIC16F877A
 * Lay Nut nhan: Button.
 * Lay dien tro: Resistor
 * Lay cong not (Thay transisto) : Not
 */
```

```
#include <16F877A.h>
#include "def_877a.h"
#device adc=16

#FUSES NOWDT          //No Watch Dog Timer
#FUSES HS              //High speed Osc (> 4mhz for PCM/PCH) (>10mhz for PCD)
#FUSES NOPUT           //No Power Up Timer
#FUSES NOBROWNOUT      //No brownout reset
#FUSES NOLVP            //No low voltage prgming, B3(PIC16) or B5(PIC18) used
for I/O
#FUSES NOCPD           //No EE protection
```

```

#FUSES NOWRT          //Program memory not write protected
#FUSES NODEBUG        //No Debug mode for ICD
#FUSES NOPROTECT     //Code not protected from reading

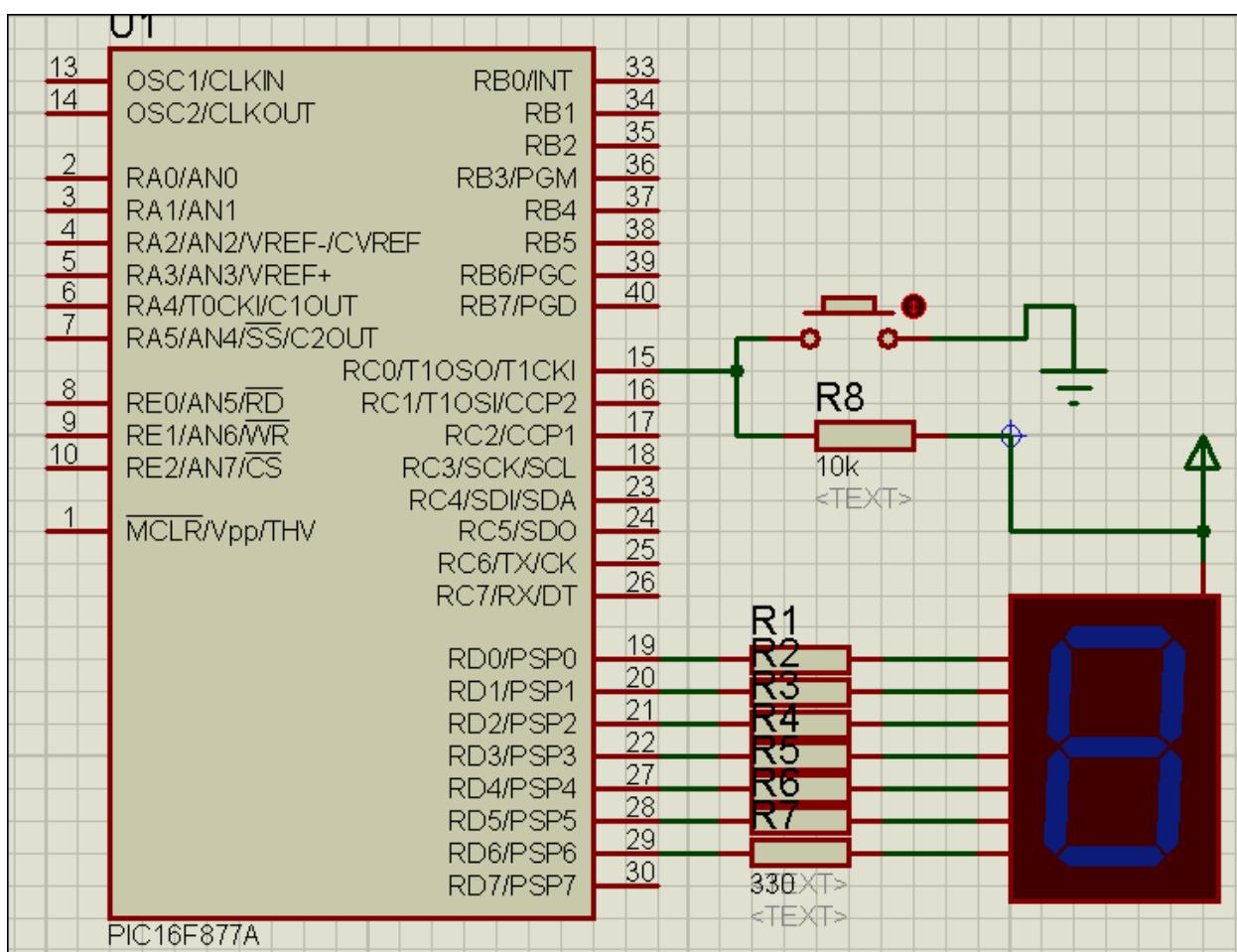
#use delay(clock=20000000)

void main()
{
    TRISA0=0;
    TRISB=255;
    port_b_pullups (0xff);
    while(TRUE){
        if(RB7==0)
        {
            RA0=0;
        }
        else RA0=1;
    }
}

```

### 6.9. Viết chương trình hiển thị số lần ấn nút PINC0 hiển thị ra led 7 đoạn.

- ✓ Phản ứng mô phỏng:



**⇒ Chú ý:** Như tôi đã trình bày với dòng fpic 16f877a thì portb bên trong phần cứng đã có điện trở kéo lên nên khi thiết kế phần cứng input thì không cần dùng điện trở ngoài (bài 6.8.). Đối với các port còn lại vì cực máng để hở nên khi dùng chế độ input ta buộc phải dùng điện trở kéo lên (bài 6.9).

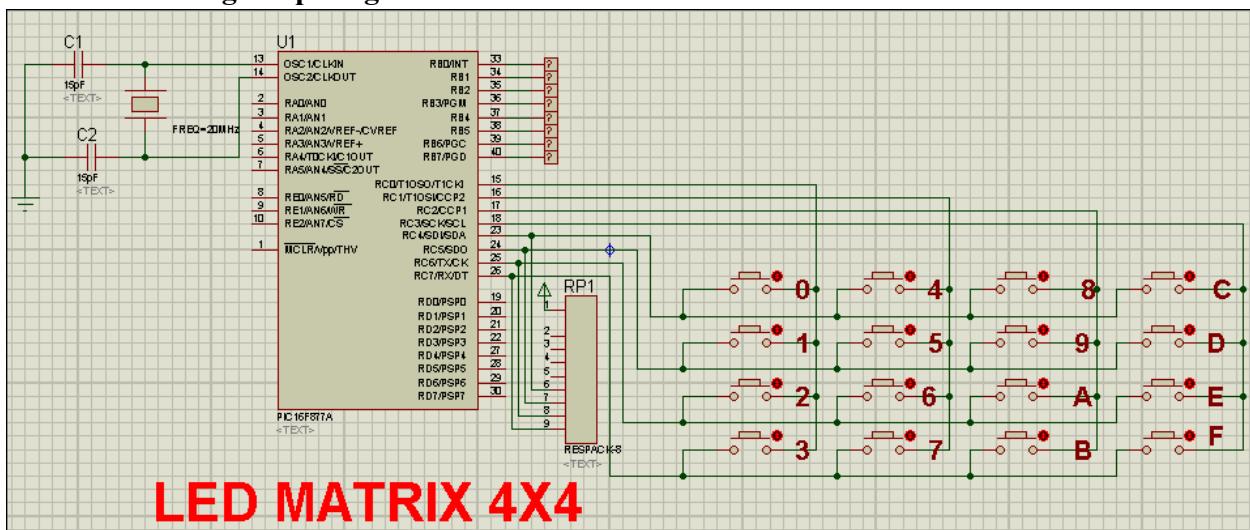
✓ **Code:**

```
#include <16F886.h>
#include <def_877a.h> //file header do người dùng định nghĩa
#fuses HS,NOWDT,NOPROTECT,NOLVP
#use    delay(clock=20000000)
//char dig[] = {192, 249, 164, 176, 153, 146, 130, 248, 128, 144}; // mang chứa giá trị led 7
daon
char dig[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};

void main()
{
    int8 i;
    // TODO: USER CODE!!
    trisD = 0x00; //output
    portD = 0xff; //off led
    trisC = 0b00000001;// RC0 INPUT
    while(true)
    {
        if(RC0==0) i++;           //Kiem tra chan RC0, neu RC0=0 se tang bien I len 1.
        while(!RC0); //Cho chan RC0=1 (den khi nha phim nhan)
        portD=dig[i];           //gan portD=mang dig[i].
    }
}
//end of main program
```

### 6.10. Viết chương trình đọc dữ liệu phím ma trận 4x4.

✓ **Phần cứng mô phỏng:**



**✓ Giải thuật:**

- Các chân 0, 1, 2, 3 được set như các chân Output và giữ ở mức cao, các chân 4, 5, 6, 7 là Input và có điện trở kéo lên. Lần lượt kéo chân 0, 1, 2, 3 xuống thấp (lần lượt xuất giá trị 0 ra từng chân), đọc trạng thái các chân 4, 5, 6, 7 để kết luận nút nào được nhấn. Ví dụ như hình trên, nút '0' được nhấn thì quá trình quét sẽ cho kết quả như sau:

Bước 1: kéo chân 0 xuống 0 (các chân 1,2,3 vẫn ở mức cao), kiểm tra 4 chân 4, 5, 6, 7 thu được kết quả D4=1, D5=1, D6=1, D7=1. (giá trị đọc về của PINB là 00001111 nhị phân)

Bước 2: kéo chân 1 xuống 0, kiểm tra lại D4, D5, D6, D7, kết quả thu được D4=1, D5=0, D6=1, D7=1 (giá trị đọc về của PINB là 0b00001011 nhị phân). Chân D5=0 tức có 1 nút ở hàng thứ 2 được nhấn, chúng ta lại đang ở Bước thứ 2 tức nút nhấn thuộc cột thứ 2. Chúng ta có thể dừng quá trình quét tại đây và kết quả thu về nút ở hàng 2, cột 2 (tức nút '1' được) được nhấn.

Quá trình quét cho các nút khác cũng xảy ra tương tự.

Các bạn tham khảo thêm các code dùng mảng.

**✓ Code:**

```
/*
 * Keywords Proteus
 * Lay vdk: PIC16F877A
 * Nut nhan: Button.
 * Phim matrix: Keypad-smallcalc
 * Cong Logic hien thi: LOGICPROBE (BIG)
 * Dien tro thanh: RESPACK-8
 * Lay dien tro: Resistor
 * Lay cong not (Thay transisto) : Not
 */
#include <16f877a.h>
#include "def_877a.h"
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
#use fast_io(b)

//DE THAY DOI PORT
#define KEYPORT PORTC
#define TRISKEYPORT TRISC
#define BASEKEY PIN_C4

//Cac ham quet phim
char xoayTrai(char value);
char readKey();

void main(void)
{
    char key=0;
    TRISB=0X00;//OUPUT
    PORTB=0XF0;//4bit cao input
    delay_ms(1000);
```

```
while(true)
{
    key=readKey();//DOC PHIM
    if(key!=0) PORTB=key-1;//HIEN THI PHIM NEU CO MOT PHIM NHAN
}

//HAM CON
char xoayTrai(char value)
{
    char temp;
    temp=value;
    temp=temp>>7;
    value=value<<1;
    value=value|temp;
    return value;
}

char readKey()
{
    char temp;//LUU GIA TRI XOAY CHO PORT QUET PHIM
    char key;//LUU GIA TRI KEY DA NHAN
    //SAU NAY BAN MA HOA BIEN KEY DE HIEN THI RA LED HAY GI GI DO NHE
    int i;//BIEN DEM VONG

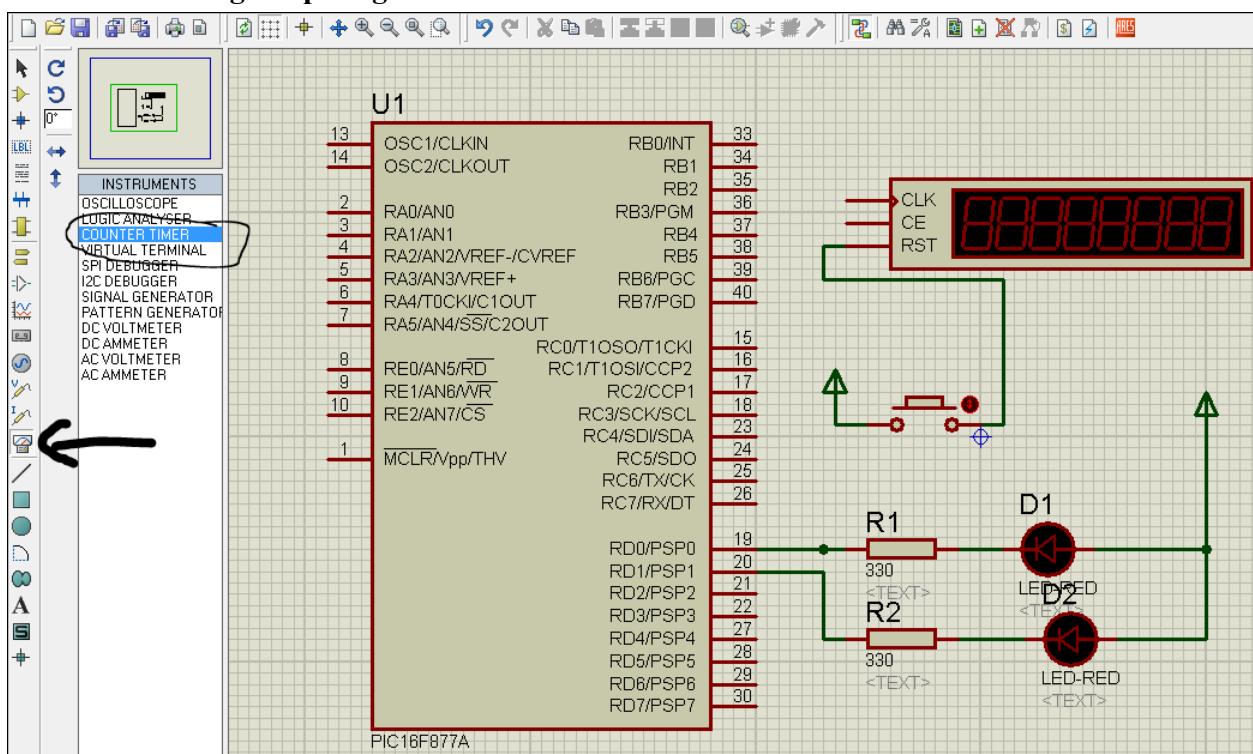
    TRISKEYPORT=0XF0;//BAN PHAI THIET LAP INPUT VA OUTPUT CHO PORT QUET
    NHE
    KEYPORT=0X00;

    temp=0x7F;
    for( i=0;i<4;i++)
    {
        temp=xoayTrai(temp);//XOAY MOT BIT
        KEYPORT=temp;//DUA RA PORT DE QUET QUA HANG/COT MOI
        KEY=0;//MINH CHO KEY LUC NAO CUNG BAT DAU BANG 0, NEU KHAC 0 THI LA
        CO PHIM NHAN
        if( input_state(BASEKEY)==0 )//PHAI DUNG HAM INPUT_STATE KHONG DUOC
        DUNG INPUT(PIN) BAN NHE, HAM INPUT(PIN) DUNG LA CO LOI
        {
            KEY=i*4+1;
            break;
        }
        else if( input_state(BASEKEY+1)==0 )
        {
```

```
KEY=i*4+2;
break;
}
else if( input_state(BASEKEY+2)==0 )
{
KEY=i*4+3;
break;
}
else if( input_state(BASEKEY+3)==0 )
{
KEY=i*4+4;
break;
}
}
return KEY;
}
```

#### 6.11. Dùng ngắt Timer0 nhấp nháy led 1s tắt, 1s sang tại chân RD0.

- ### ✓ Phân cứng mô phỏng:



Lấy COUNTER TIMER bằng cách: chọn biểu tượng đồng hồ => Counter timer.

## Dùng Thạch Anh 20MHz.

- ✓ Code:

```
#include <16F877A.h>
```

```
#include <def_877a.h> //file header do nguoi dung dinh nghia
```

#fuses HS,NOWDT,NOPROTECT,NOLVP

```
#use    delay(clock=20000000)

int16 count_1s=0, count_15s=0;
//dinh nghia chan quet led
#define led  rd0
#define led1  rd1

//chuong trinh phuc vu ngat timer 0
#int_TIMER0
void TIMER0_isr(void) {
    tmr0if=0;// xoa co ngat
    set_timer0(6); //0.2us*250=50us *64 =1600 us*625 =1 000000 us=1s
    ++count_1s;
    ++count_15s;
    if(count_1s==15){SAI PHAI =625
        led = !led;           //Dao trang thai chan Led
        count_1s=0;
        //rotate_left(&count_1s,1);
    }

    if(count_15s==9375)
    {
        led1=!led1;
        count_15s=0;
    }
}

void main()
{
    //Khoi tao T1 va ngat.
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_64); //T1 dem xung noi, ti le chia 1
    enable_interrupts(INT_TIMER0);
    enable_interrupts(GLOBAL);
    set_timer0(6);      //250*32*0.2 = 1600us
                        // 1600us*625= 1s

    //Thiet lap cac pin xuat nhap
    set_tris_d(0x00);    //PORTD xuat du lieu.
    while(1){
    }
}
//end of main program
```

**6.12. Viết chương trình bộ định thời xung vuông 1KHz duty 50% tại RD0, và xung vuông 2KHz 50% tại chân RD1.**

✓ Phản ứng mô phỏng: giống bài 6.11. Chú ý Thạch Anh sử dụng 20MHz

✓ Code:

```
#include "16F877A.h"
#include "def_877a.h"
#fuses HS,PUT,NOWDT,NOPROTECT,NOLVP
#use delay(Clock = 20000000) //Thach anh 20MHz

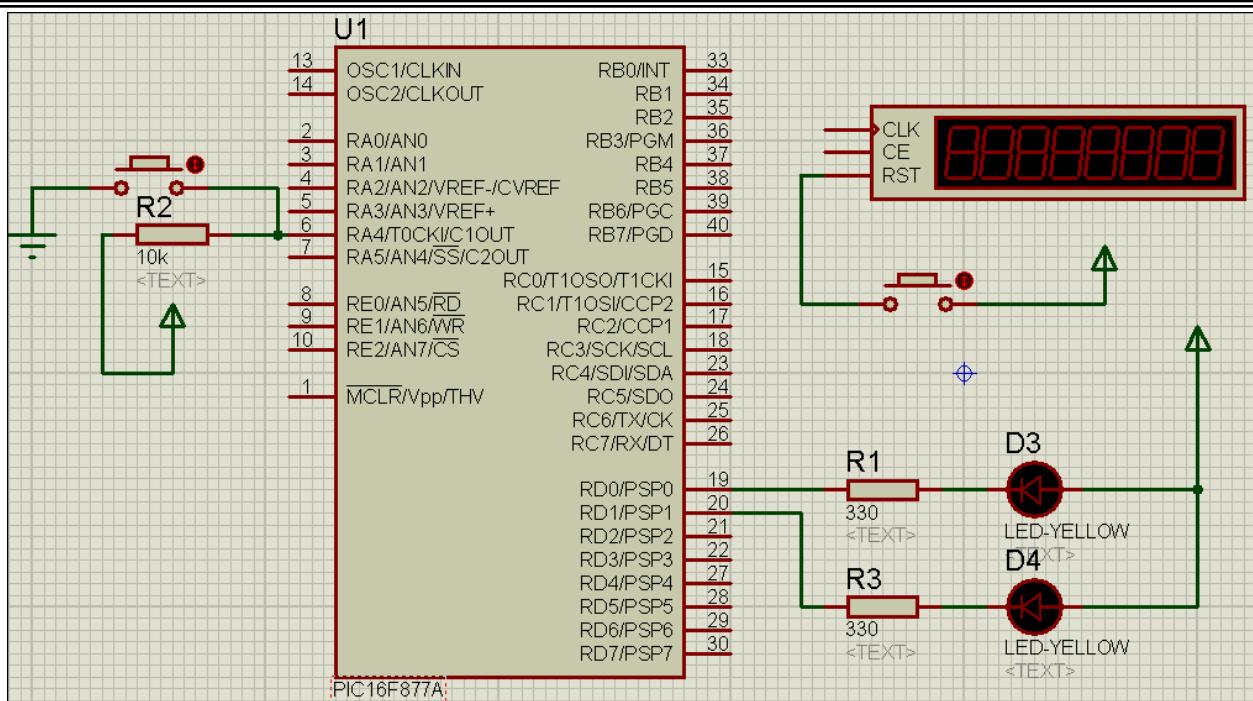
#define _XTAL_FREQ 20000000L

#int_TIMER1
void TIMER1_isr(void)
{
    RD1 = !RD1;           //Tao xung 2KHz.
    if(RD1==0)            //RD1 = 0 thi dao trang thai RD0.
        RD0 = !RD0;       //Tao xung 1KHz.
    set_timer1(65035);
}

void main()
{
    //Khoi tao T1 va ngat.
    setup_timer_1(T1_INTERNAL|T1_DIV_BY_1); //T1 dem xung noi, ti le chia 1
    enable_interrupts(INT_TIMER1);
    enable_interrupts(GLOBAL);
    set_timer1(65035);      //f = 2KHz => CK xung = 0,5ms, he so duty 50% => muc cao = muc
    thap = 0,25ms            //Thach anh 20 MHz => CK may = 4/20 = 0,2us
                            //0,25ms = 250us = 500xung = 65535 - 65035
    //Thiet lap cac pin xuat nhap
    set_tris_d(0x00);       //PORTD xuat du lieu.
    //-----
    while(1);
}
```

**6.13. Viết chương trình ngắt timer0 dùng ở chế độ Counter.**

✓ Phản ứng mô phỏng:



✓ Code:

```
#include <16F877A.h>
#include <def_877a.h> //file header do nguoi dung dinh nghia
#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock=20000000)

int16 count_1s=0;
//dinh nghia chan quet led
#define led rd0

/*chuong trinh phuc vu ngat timer 1-----*/
#int_TIMER0
void TIMER0_isr(void) {
    set_timer0(250);
    led = !led;           //Dao trang thai chan Led
}

void main()
{
    //Khoi tao T1 va ngat.
    setup_timer_0 (RTCC_DIV_1|RTCC_EXT_1_TO_H); //Timer0 is Counter bo chia 1
    enable_interrupts(INT_TIMER0); //Cho phep ngat Timer0
    enable_interrupts(GLOBAL); // Cho phep ngat toan cuc
    set_timer0(250); //Gia tri dem 255-250 =5

    set_tris_d(0x00); //PORTD xuat du lieu.
//-----
    while(1){
```

```

rd1=!rd1;
delay_ms(500);
};

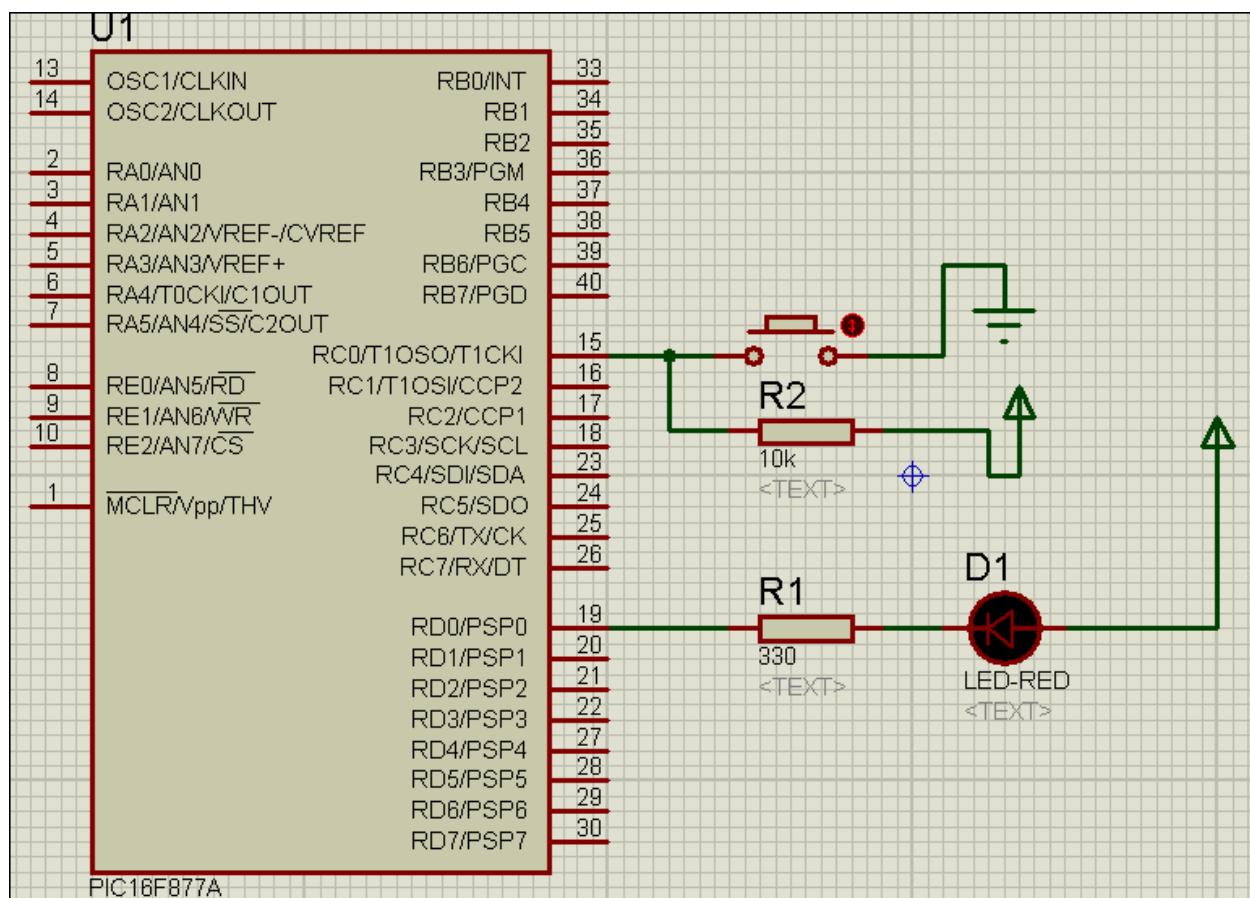
}

//end of main program

```

#### 6.14. Viết chương trình ngắt Timer1 dùng chế độ Counter.

✓ Phản ứng mô phỏng:



✓ Code:

```

#include <16F877A.h>
#include <def_877a.h> //file header do nguoi dung dinh nghia
#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock=20000000)

int16 count_1s=0;
//dinh nghia chan quet led
#define led rd0

//chuong trinh phuc vu ngat timer 1
#int_TIMER1

```

```

void TIMER1_isr(void) {
    //tmr1if=0;// xoa co ngat
    set_timer0(65535);
    led = !led;           //Dao trang thai chan Led
}

void main()
{
    //Khoi tao T1 va ngat.

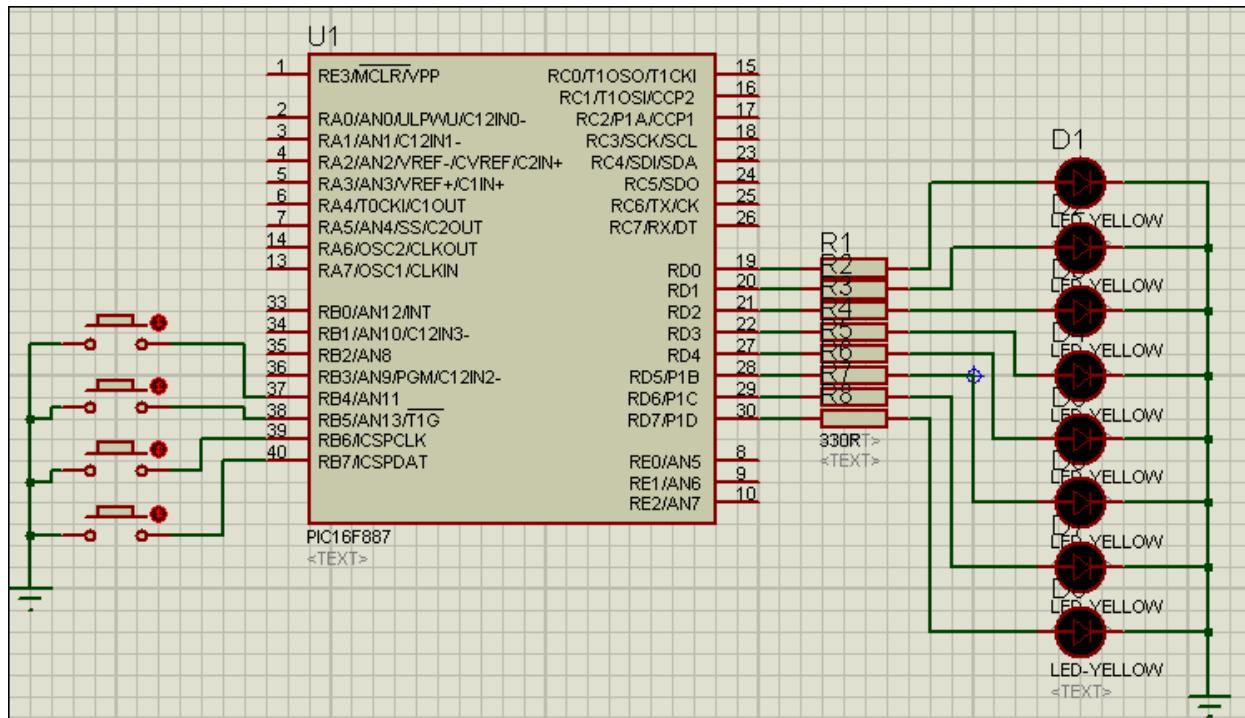
    setup_timer_1 (T1_EXTERNAL|T1_DIV_BY_1); // Timer0 is Counter bo chia 1
    enable_interrupts(INT_TIMER1);
    enable_interrupts(GLOBAL);
    set_timer1(65535);      //Gia tri dem 256-251 =5

    set_tris_d(0x00);       //PORTD xuat du lieu.
    //-----
    while(1);
}
//end of main program

```

### 6.15. Viết chương trình ngắt RB, đọc dữ liệu PORTB xuất PORTD.

- ✓ Phản ứng mô phỏng:



- ✓ Code:

```
#include "16f877A.h"
```

```
#include <def_877a.h>
#fuses HS,PUT,NOWDT,NOPROTECT,NOLVP
#use delay(Clock = 20000000) //Thach anh 20MHz

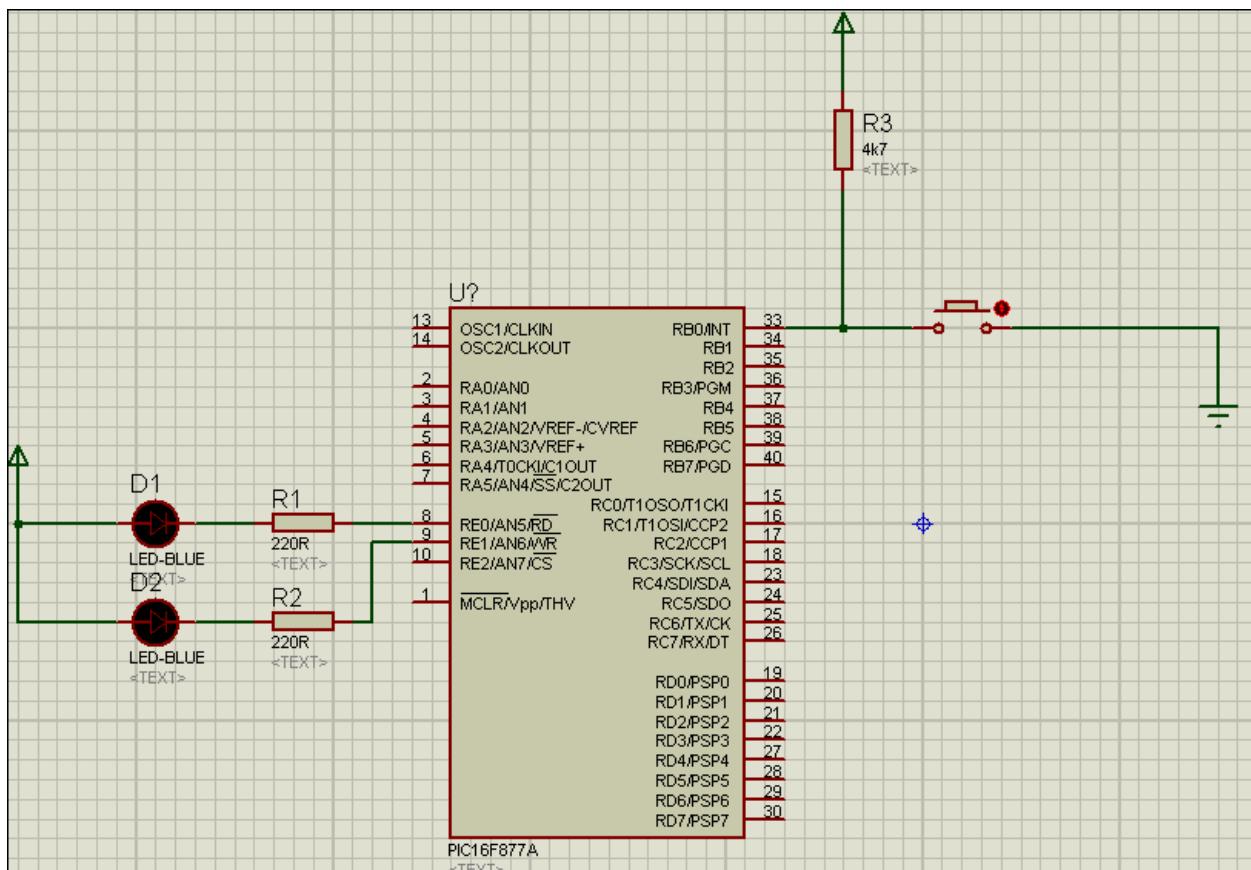
#define RB
void RB_isr(void)
{
    PORTD = PORTB;
    delay_ms(20);
}

void main()
{
    setup_adc_ports(NO_ANALOGS|VSS_VDD);
    setup_adc(ADC_OFF);
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_1(T1_DISABLED);
    setup_timer_2(T2_DISABLED,0,1);
    setup_comparator(NC_NC_NC_NC);
    enable_interrupts(INT_RB); //Khai báo ngắt RB
    enable_interrupts(GLOBAL);

    set_tris_b(0xff); //PORTB nhap du lieu.
    port_b_pullups(0xff); //Kich hoat dien tro treo.
    trisd = 0x00; //PORTD xuat data.
    while(1);
}
```

#### 6.16. Viết chương trình thay đổi trạng thái led dùng ngắt ngoài (INT0).

- ✓ Phản ứng mô phỏng:



✓ Code:

```
#include<16f877a.h>
#include <def_877a.h>
use delay(clock=2000000)
#fuses XT,NOWDT,NOPROTECT,NOLVP
#use fast_io(B)
#use fast_io(E)
int8 mode=0;

//chuong trinh ngat RB0
#int_ext
void ngatngoai(){
    mode++;
    if(mode==3)mode=0;
}

//chuong trinh con
void mode1(void);
void mode2(void);
void tat(void);

//chuong trinh chinh
void main(void){
    set_tris_B(0x01); //RB0 INPUT
    set_tris_E(0x00); //PORTE OUTPUT
```

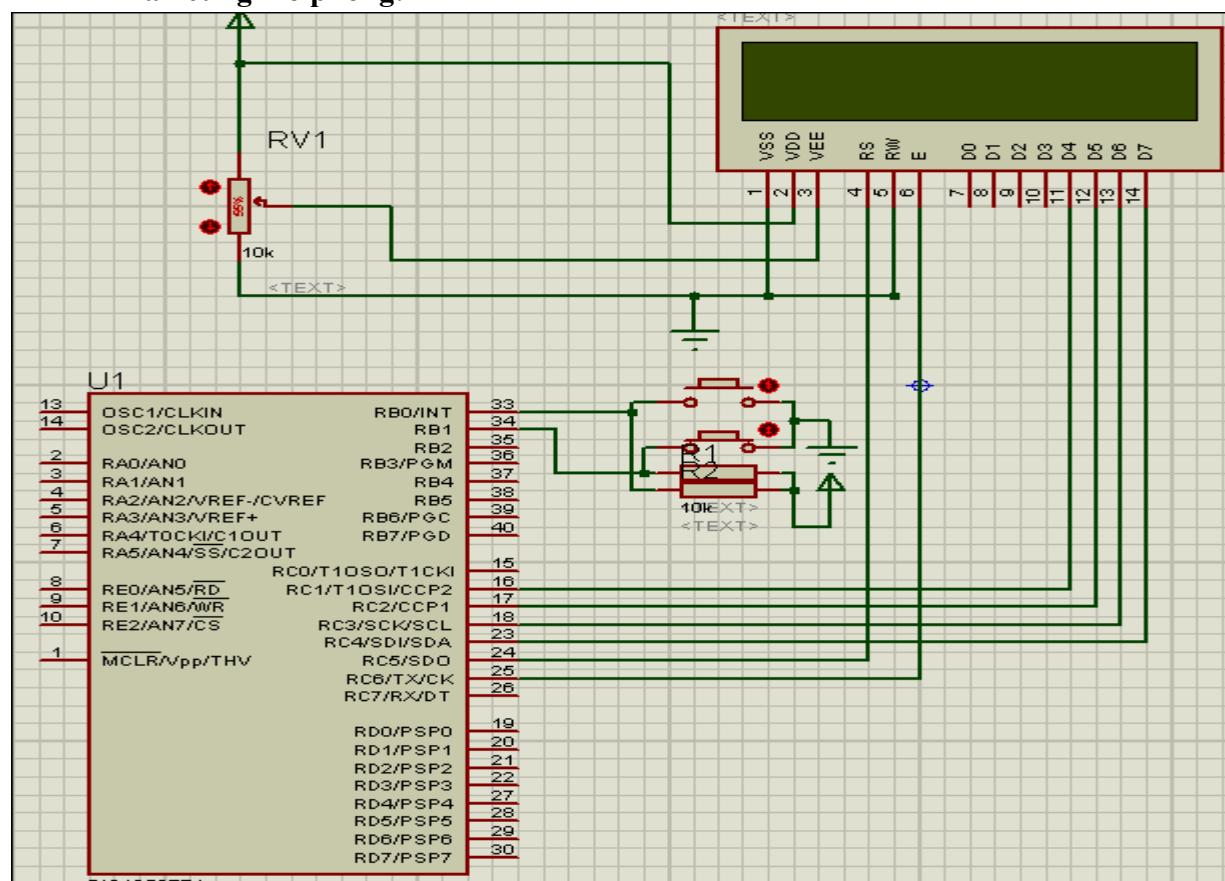
```

enable_interrupts(int_ext); //cho phep ngat int
enable_interrupts(global); //Bat ngat toan cuc
ext_int_edge(H_to_L); //Ngat canh thap
while(1){
    switch(mode){
        case 1:mode1();break;
        case 2:mode2();break;
        default:tat();break;
    }
}
void mode1(){
    output_low(pin_E0);
    output_high(pin_E1);
}
void mode2(){
    output_low(pin_E1);
    output_high(pin_E0);
}
void tat(){
    portE=0x00;
}

```

### 6.17. Viết chương trình đếm số xung encoder hiển thị lên LCD1602.

✓ Phản ứng mô phỏng:



Ứng dụng đếm số xung của encoder, hiển thị ra lcd về chiều quay thuận và nghịch. Các hàm và lệnh điều khiển LCD các bạn vui lòng xem trên mạng hoặc tài liệu đính kèm.

✓ **Code:**

```
#include <16F877A.h>
#include <def_877a.h>
//#include <pid.c>
#device *=16 adc=10
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
//#use rs232(baud=115200,parity=N,xmit=PIN_C6,recv=PIN_C7,bits=9)
#include <LCD_NVN.c>

// bien toan cuc
int pwm;
signed int16 xung;

//-----
#endif
#int_EXT
void ngat ngoai_int0(void)
{
    if(!input(pin_b1))
        xung++; // NEU QUAY CUNG CHIEU KIM DH THI TANG XUNG LEN 1
    else
        xung--; // NEU QUAY CUNG CHIEU KIM DH THI GIAM XUNG LEN 1
}

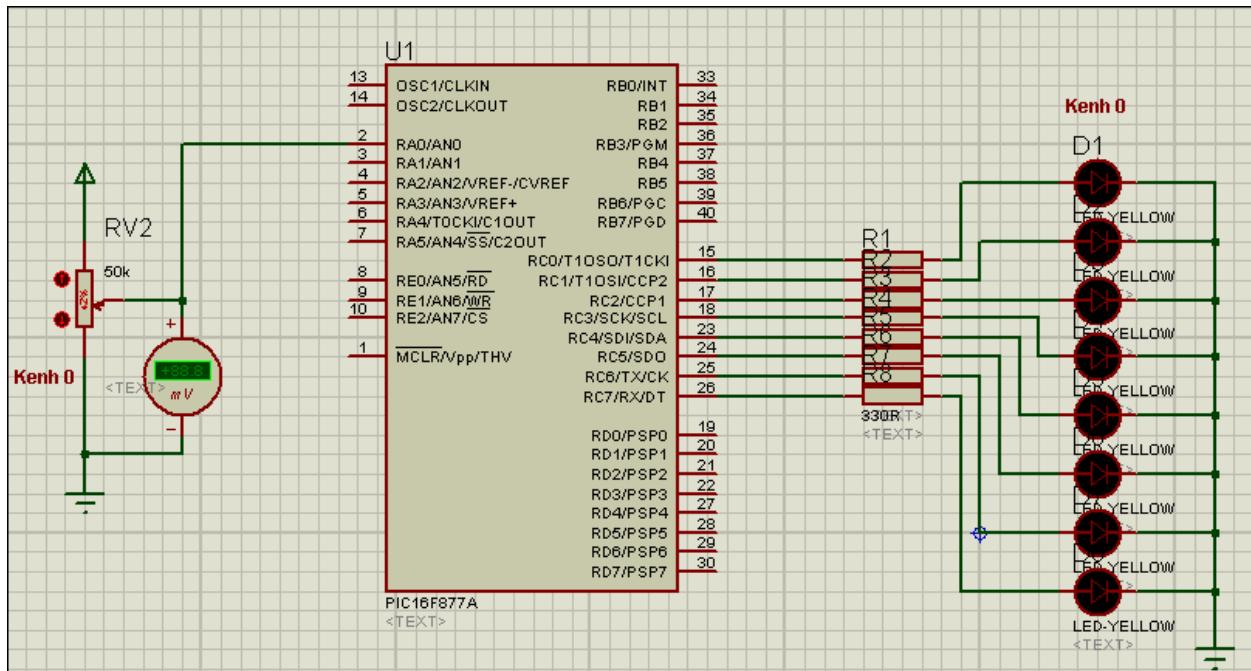
void int_int0(void)
{
    // Khoi tao cho ngat ngoai
    enable_interrupts (INT_EXT);
    ext_int_edge(H_TO_L);
    enable_interrupts (GLOBAL);
}

void main()
{
    trisb = 0x01; //RB0 INPUT
    //trisd = 0x01;
    trisc=0;//output
    portc=0xff;//dau tien led sang
    PORTB=0;
    PORTD=0XFF;
```

```
xung=0;  
int _int0(); // khai tao ngat ngoai  
  
lcd_init(); // khai tao dung LCD  
lcd_gotoxy(1,1); // cho toa do dau tien x:1, y:1  
Printf(lcd_putc," Init OK");  
delay_ms(1000);  
  
int1 flag_lcd=0;  
  
while(1)  
{  
  
    // if(flag_lcd==0){  
        lcd_gotoxy(1,1); // cho toa do dau tien x:1, y:1  
        printf(lcd_putc,"WWW.NTT.EDU.VN"); // in dong chu LCD  
        //lcd_moveRIGHT(20);  
        // DELAY_MS(200);  
    //}  
    // flag_lcd=1;  
    //----- hien thi 1 bien ra lcd-----//  
    lcd_gotoxy(0,2); // cho toa do dau tien x:1, y:1  
    printf(LCD_putc,"\nPluse:%4ld",xung);  
  
}  
}  
  
//end main-----
```

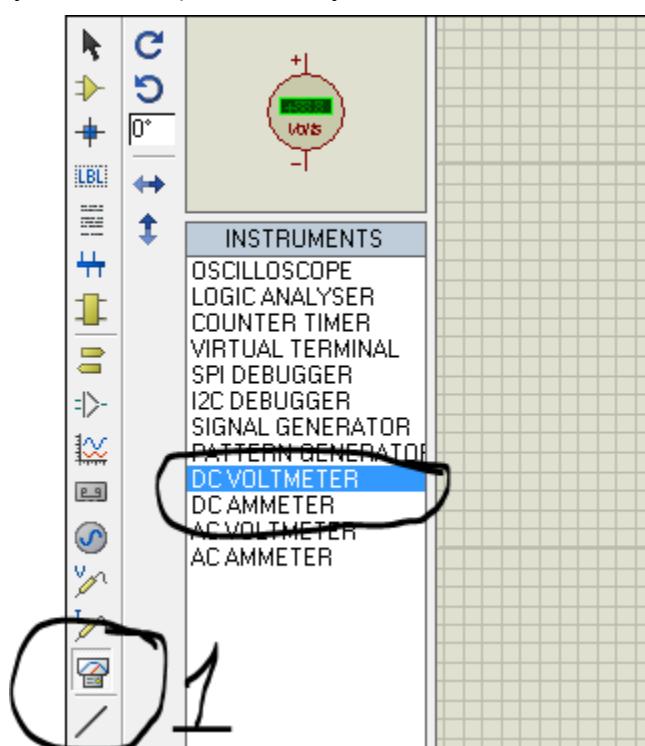
**6.18. Viết chương trình đọc ADC tại chân AN0 dùng điện áp tham chiếu là VDD (Nguồn) hiển thị ra PORTC.**

- ✓ **Phản ứng mô phỏng:**



Để lấy DC VOLTMETER bạn vào hình biểu tượng đồng hồ => DC VOLTMETER.

Lấy Biến trả ban nhấn Keywords: POT-HG



✓ Code:

```
#include <16F877A.h>
#include <def_877a.h>
#device *=16 adc=8
#FUSES NOWDT, HS, NOPUT, NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
```

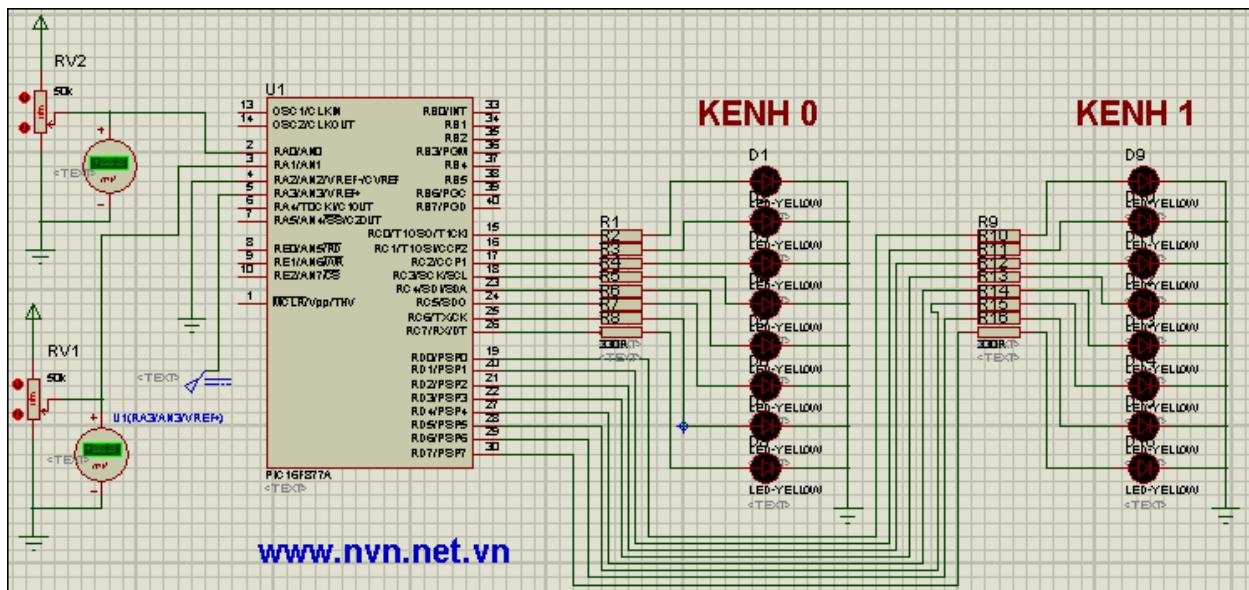
```

INT8 GT;
//chuong trinh chinh
void main(void)
{
    set_tris_c(0x00);      //PORTC xuat du lieu.
//***** khai tao cac modules *****/
    setup_adcadc_clock_internal);
    setup_adc_ports (AN0); // chon chan AN0 so voi Vdd
    //setup_adc_ports(ALL_ANALOG);

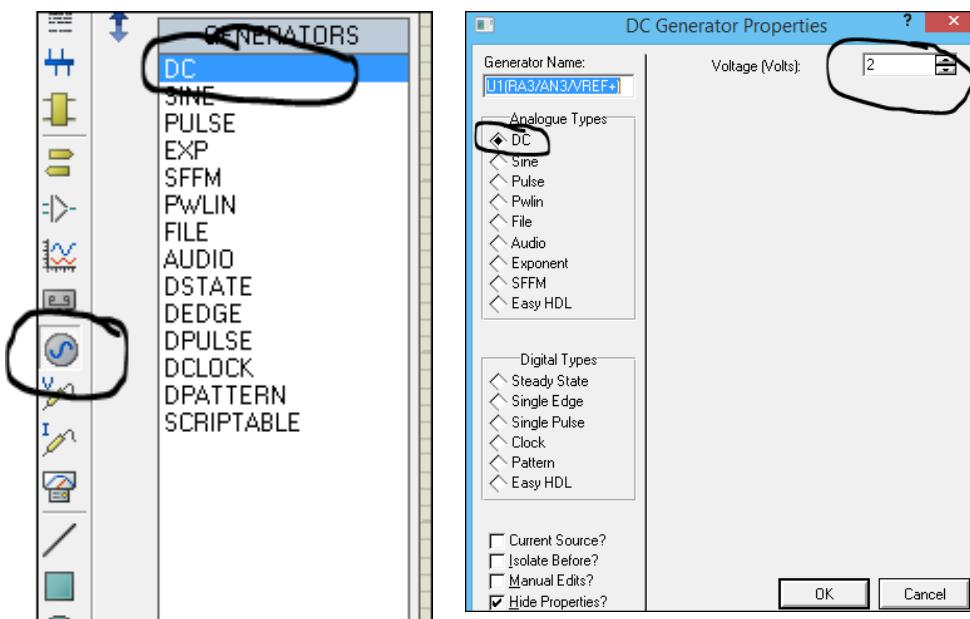
while(TRUE)
{
    set_adc_channel(0); // doc gia tri tai chan 0
    GT = READ_ADC(); // gan bien "GT" vao gt doc duoc
    DELAY_MS(20) ;
    PORTC=GT;
}
}

```

**6.19. Viết chương trình đọc giá trị analog của chân AN0, AN1 với điện áp tham chiếu với VREF- = 0V, VREF+ = 2V rồi xuất giá trị đọc được ra PORTC và PORTD.**



- ✓ **Phản cứng mô phỏng:**  
Để lấy được nguồn 2v bạn vào biểu tượng => DC sau đó điều chỉnh điện áp mong muốn tại Voltage.  
Ở đây bạn có thể chọn thêm nhiều nguồn khác nhau để mô phỏng.



✓ **Code:**

```
#include "16f877A.h"
#include <def_877a.h>
#device *=16 ADC = 10
#fuses HS,PUT,NOWDT,NOPROTECT,NOLVP
#use    delay(Clock = 20000000)      //Thach anh 20MHz
int16 temp;
void main()
{
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_2(T2_DISABLED,0,1);
    setup_comparator(NC_NC_NC_NC);
    setup_vref(FALSE);
    setup_timer_1(T1_INTERNAL);
    setup_adc_ports(sAN0|sAN1|VREF_VREF); //kenh A0,A1 nhan tin hieu, A2 = 0V; A3 = 2V;
    setup_adc(ADC_CLOCK_INTERNAL);      //Thoi gian lay mau bang clock he thong.
    delay_ms(5);                      //Cho thiet lap xong ADC.
    // TODO: USER CODE!!

    trisa = 0xff;                     //PORTA nhan du lieu.
    trisc = 0x00;                     //PORTC xuat du lieu.
    trisd = 0x00;                     //PORTD xuat du lieu.
    while(1)
    {
        //Doc kenh 0
        set_adc_channel(0);          //Chon kenh 0.
        delay_us(10);               //cho chon kenh xong.
```

```

temp = read_adc();           //doc gia tri analog da chuyen doi sang so.
temp = temp/10.24;          //stepsize = 50mV. Neu giao tiep LM35 thi chia 2.048 (stepsize =
10mv)

PORTC = temp;               //Value = Vol/0,05
delay_ms(1000);             //Thoi gian hien thi 1s.

//Doc kenh 1

set_adc_channel(1);         //Chon kenh 1.
delay_us(10);               //cho chon kenh xong.
temp = read_adc();           //doc gia tri analog da chuyen doi sang so.
temp = temp/10.24;          //stepsize = 50mV. Neu giao tiep LM35 thi chia 2.048 (stepsize =
10mv)

PORTD = temp;               //Value = Vol/0,05
delay_ms(1000);             //Thoi gian hien thi 1s.

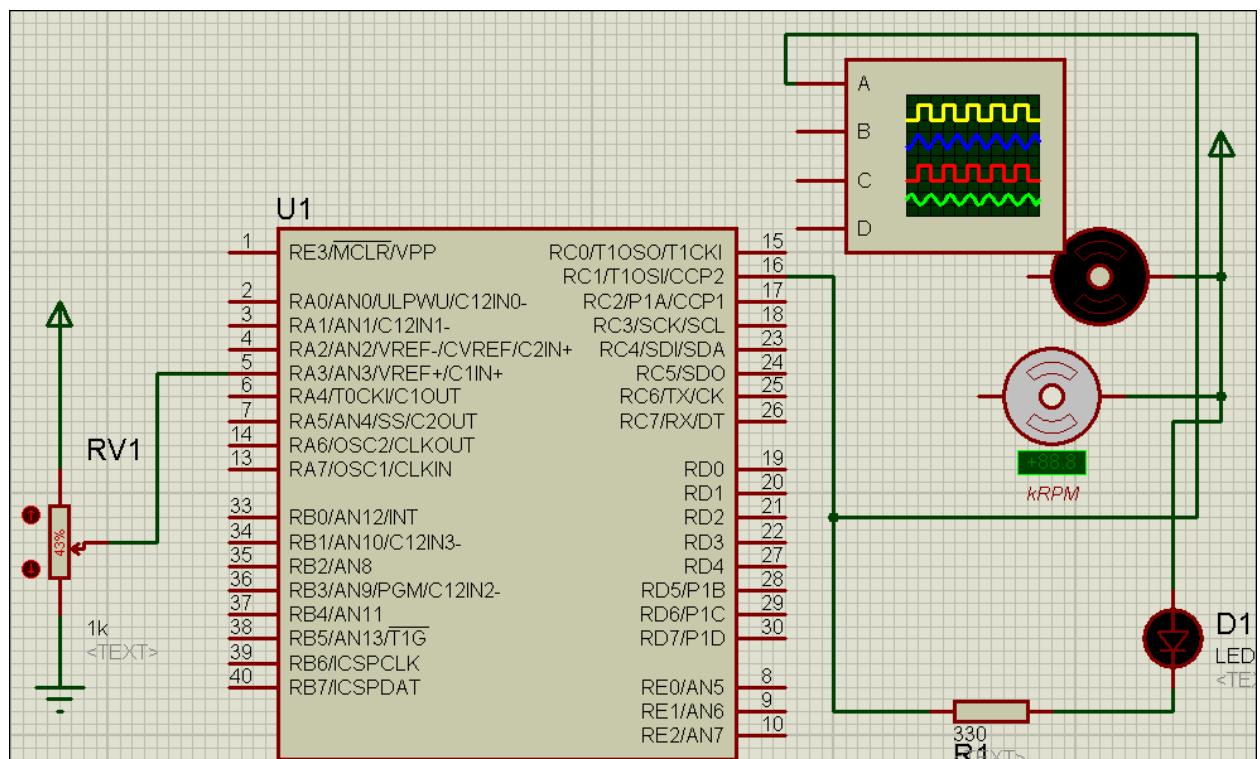
}

}

```

#### 6.19. Viết chương trình đọc tín hiệu ADC từ AN3 điều khiển độ rộng xung CCP2.

### ✓ Phản ứng mô phỏng:



Để lấy OSCILLOSCOPE bạn vào biểu tượng đồng hồ => OSCILLOSCOPE.

Để lấy Motor ban đánh keywords: Motor.

✓ Code:

// bai tap chuong trinh thay doi toc do dong co bang bien tro

```
#include "16f877A.h"
```

```
#include <def_877a.h>
```

```

#device *=16 ADC = 10
#fuses HS,PUT,NOWDT,NOPROTECT,NOLVP
#use    delay(Clock = 20000000)      //Thach anh 20MHz
int dc;

void main()
{
// vong lap mai mai
set_tris_a(0xff);
set_tris_b(0x00);
set_tris_c(0x0f);
setup_adc(ADC_CLOCK_INTERNAL);
setup_adc_ports(ALL_ANALOG); //kick hoat tat ca cac chan adc

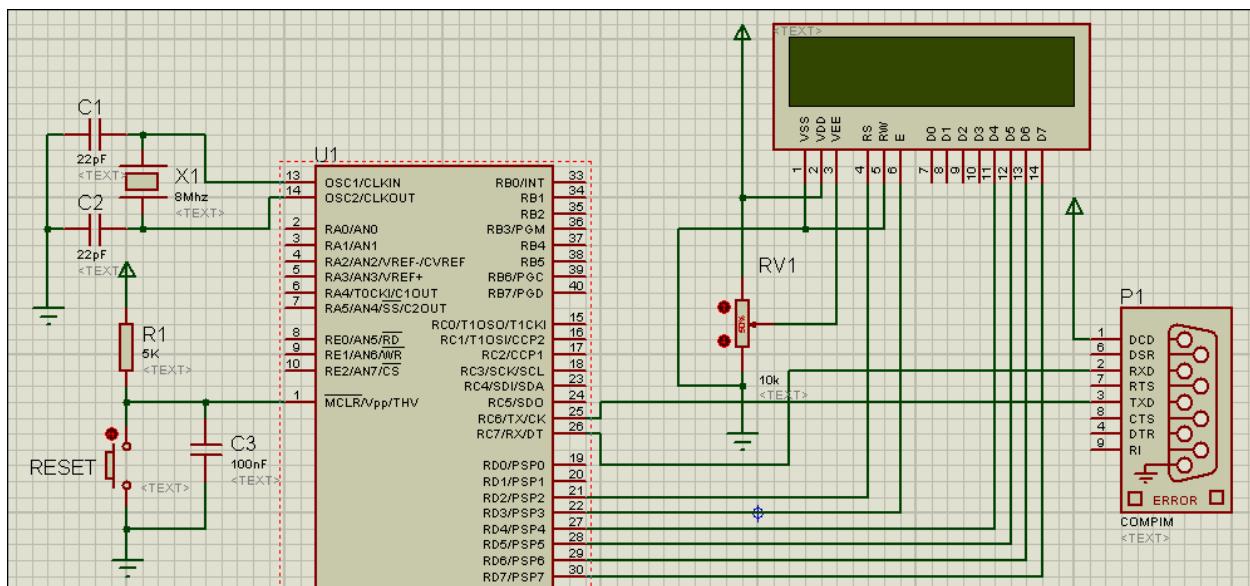
set_adc_channel(3);      // doc gia tri adc tai chan AN3
delay_us(10);           // cho 1 khoang thoi gian de chan chan nhan dc gia tri

setup_ccp2 (CCP_pwm);// khai bao chan PWM C2
setup_timer_2(T2_div_by_16,255,1);
while(1){
    dc=read_adc();
    set_pwm2_duty(dc);
}
}

```

**6.20. Viết chương trình nhận chuỗi ký tự từ PC hiển thị LCD1602 và gửi ngược lại PC thông qua chuẩn USART.**

#### ✓ Phản ứng mô phỏng:



✓ Code:

```
#include <16F877A.h>           // PIC16F877 header file
#use delay(clock=8000000)       // for 8Mhz crystal
#fuses HS, NOWDT, NOPROTECT, NOLVP // for debug mode
#include "LCD_NVN.c"
#use rs232(baud=9600, xmit=PIN_C6, recv=PIN_C7, stream=MYPC)

int8 temp;
char buffRev[16];
int8 idx = 0, i = 0;
void displayOnLCD(void);

void main(void)
{
    lcd_init();
    lcd_gotoxy(2,1);
    printf(lcd_putc,"NVN -LAB");
    lcd_gotoxy(3,2);
    printf(lcd_putc,"Nguyen Huu Luan");
    while(1)
    {
        if(0 == kbhit())
        {
            temp = fgetc(MYPC);
            if(temp=='$')
            { // Neu ky tu nhan dc bang ky tu : thi khai tao lai gia tri.
                for(i=0; i<sizeof(buffRev); i++) buffRev[i] = ' ';
                continue;
            }
            else if(temp=='#')// neu bang ky tu ket thuc thi truyen lai cho PC.
            {
                for(i=0; i<idx; i++)
                {
                    putc(buffRev[i]);
                }
                idx = 0;
                displayOnLCD(); // Hien thi ket qua thu duoc len LCD.
                continue;
            }
            if(idx<16)
            {
                buffRev[idx++] = temp;
            }
        }
    }
}
```

```

    }
}

```

```

void displayOnLCD()
{
    lcd_init(); // Init LCD.
    lcd_gotoxy(1,1);
    printf(lcd_putc, "%s", buffRev);
}

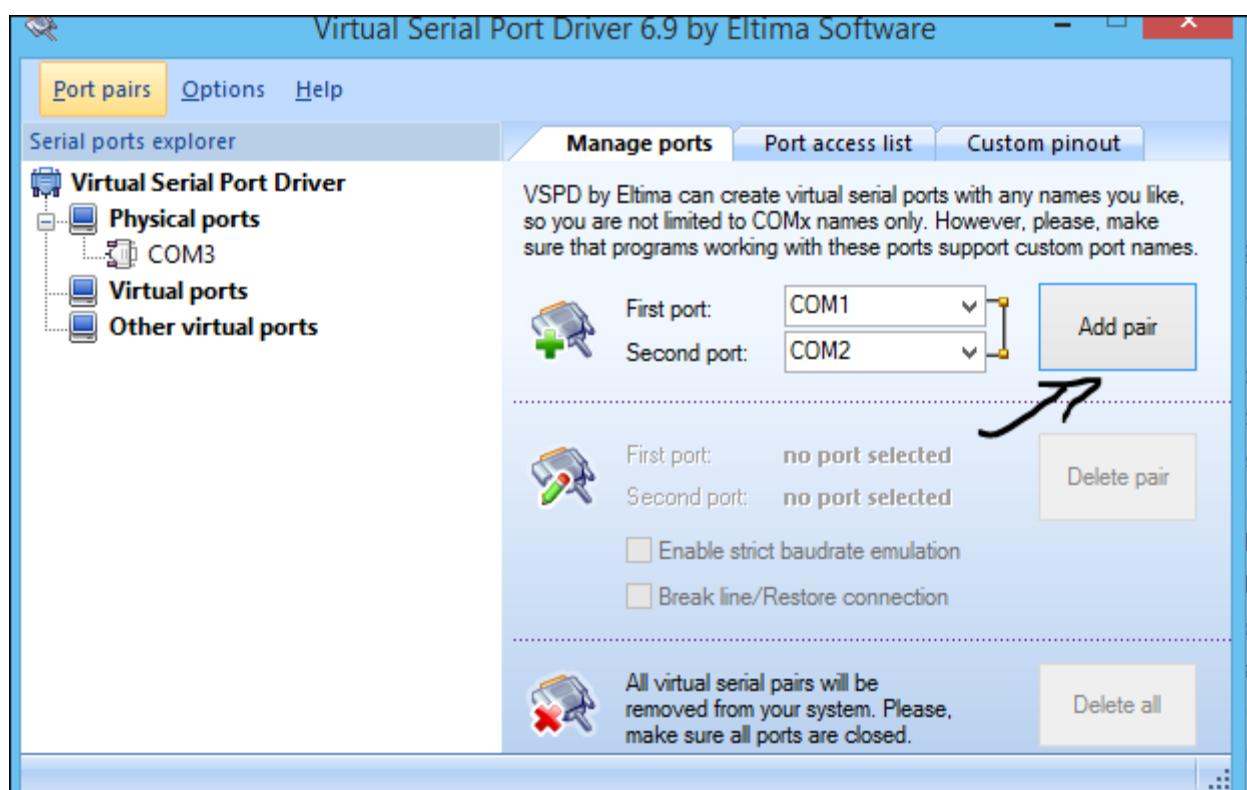
```

✓ **Kết nối Proteus với PC qua chuẩn USART.**

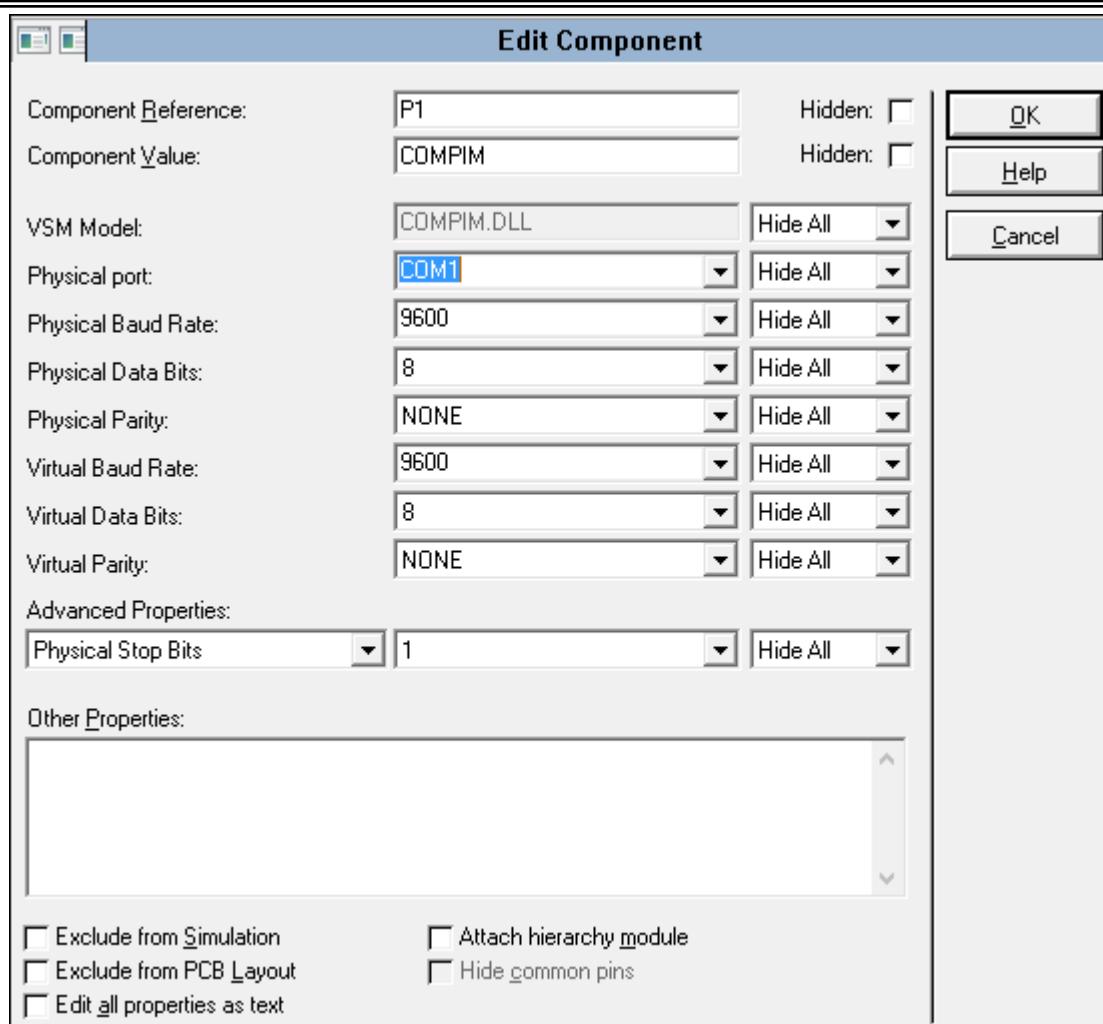
- **Bước 1:** Tạo công com ảo kết nối công Com của Proteus với Terminal.

Mở phần mềm “Configure Virtual Serial Port Driver”

Ở First port và Second port ta chọn 2 công com mà sẽ kết nối của công com Proteus và Terminal. Lưu ý công com Proteus chỉ giới hạn 1=>4 nên chỉ được chọn com1,2,3,4. Sau đó Nhấn Add pair để kết nối.

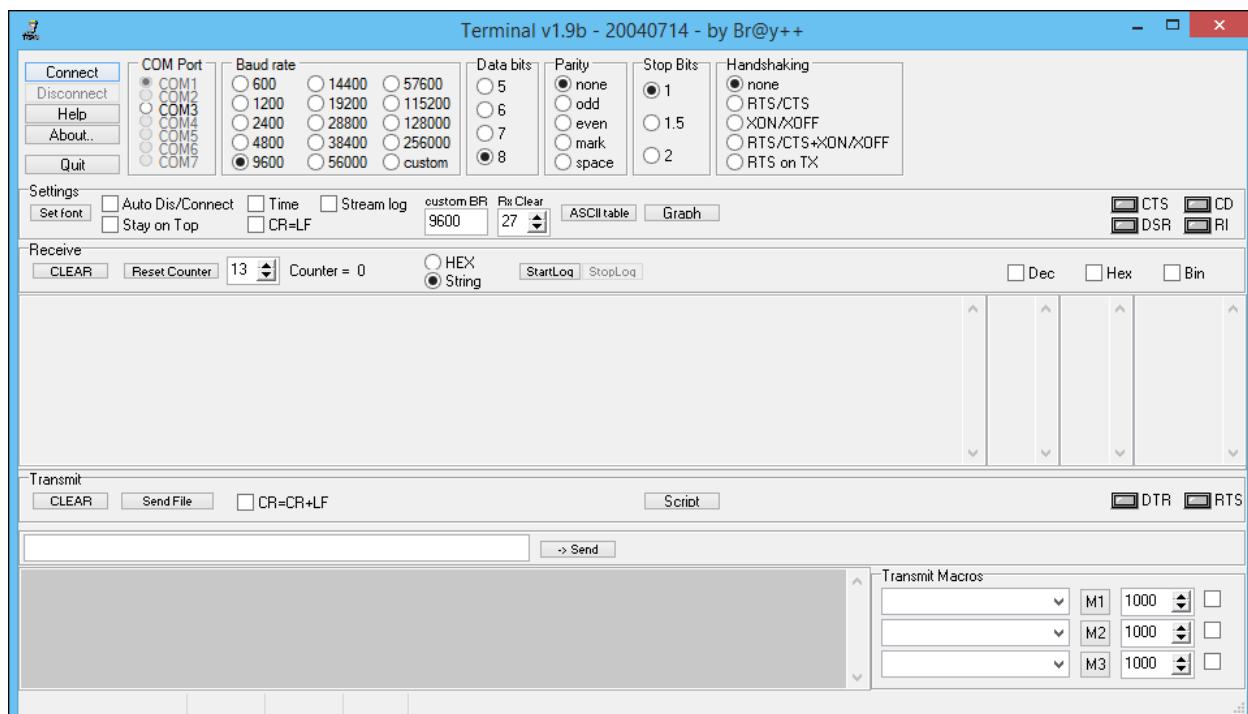


- **Bước 2:** Vào phần mềm Proteus nhập double vào ComPim điều chỉnh lại com1 (vì ta đã kết nối com với com2)



- **Bước 3:** Mở phần mềm Terminal lên chọn đúng các thông số đúng với các khai báo mà ta đã khai báo trên:
  - Com port: com2
  - Baud rate: 9600
  - Data bits: 8
  - Parity: none
  - Stop bits: 1

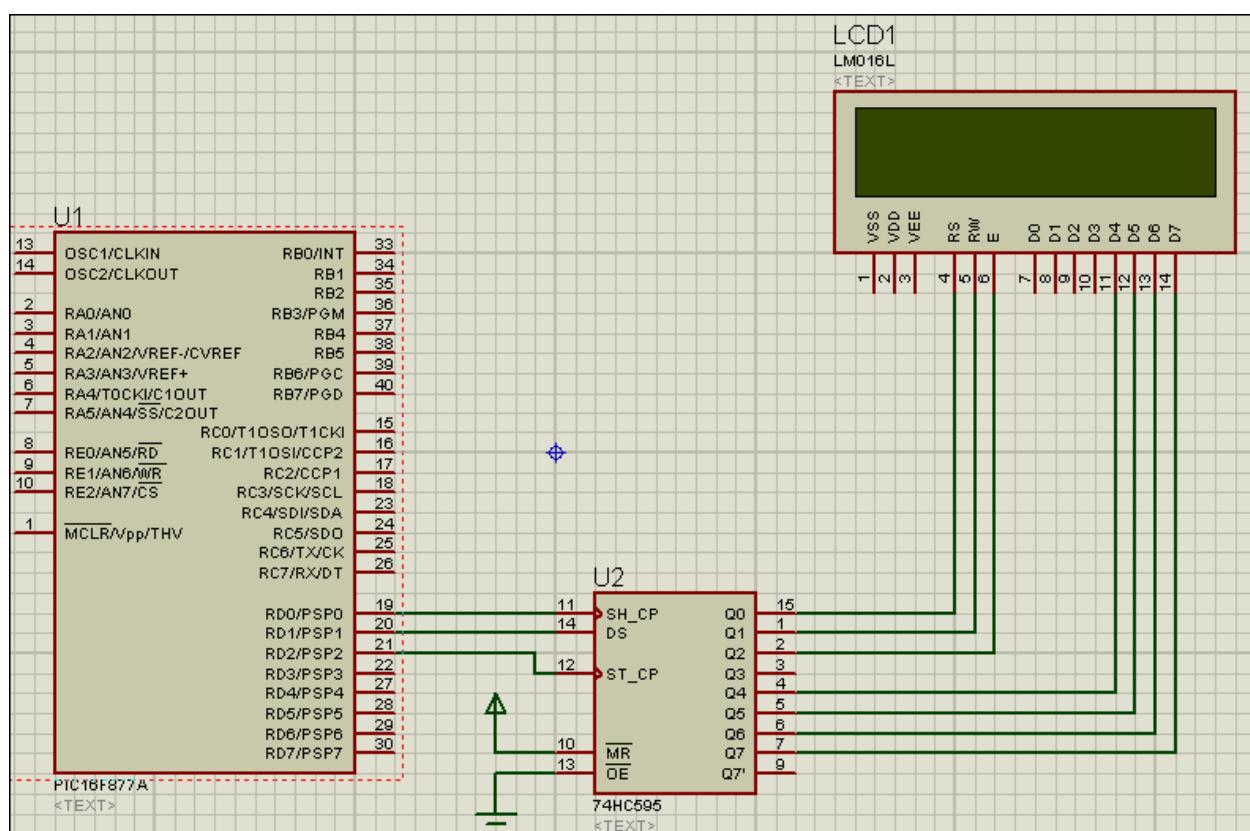
Sau đó bấm vào Connect để bắt đầu quá trình giao tiếp.



Muốn gửi dã liệu qua cổng com ta nhập vào ô textbox rồi nhấn send ở khung Transmit.  
Các ký tự nhận được từ cổng com sẽ hiển thị khung Receive.

### 6.21. Viết chương trình giải mã LCD1602 thông qua chuẩn SPI.

- ✓ Phản ứng mô phỏng:



**✓ Code:**

```
#include <16F877A.h>
#fuses HS, NOWDT ,PROTECT ,NOBROWNOUT ,NOPUT
#use delay(clock=20000000)
#include <def_877a.h>
#include <lcdspi.h> //thu vien lcd
    unsigned long      xung;

void main(void)
{
    TRISA=0xFF;
    TRISB=0x00;
    TRISC=0xFF;
    TRISD=0xF0 ;
    PORTB = 0x00;
    lcd_init();
    while( true )
    {
        lcd_gotoxy(0,0);
        printf(lcd_putchar," Hello");
        lcd_gotoxy(1,0);
        //printf(lcd_putchar,"nvn.net.vn");
        xung=-10;
        printf(lcd_putchar,"adc:%lu",xung);
    };
}
}
```

**❖ File: lcdspi.h.**

```
//#include <16F877A.h>
#define data74595 RD1
#define clock74595 RD0
#define latch74595 RD2
int8 buffer_data_lcd = 0;
void OutPut74595(int8 data);
void LCD_STROBE()
{
    bit_set(buffer_data_lcd,2);
    OutPut74595(buffer_data_lcd);
    bit_clear(buffer_data_lcd,2);
    OutPut74595(buffer_data_lcd);
}

/* write a byte to the LCD in 4 bit mode */
void LCD_DATA(unsigned char c)
{
    buffer_data_lcd = buffer_data_lcd & 0x0F;
    buffer_data_lcd = buffer_data_lcd | (c << 4);
    OutPut74595(buffer_data_lcd);
}

void lcd_write(unsigned char c)
{
    delay_us(40);
    LCD_DATA(c>>4);
    LCD_STROBE();
    LCD_DATA( c );
    LCD_STROBE();
}

void lcd_clear(void)
{
```

```

bit_clear(buffer_data_lcd,0);
OutPut74595(buffer_data_lcd);
lcd_write(0x1);
delay_ms(2);
}
/* write one character to the LCD */

void lcd_putchar(char c)
{
    bit_set(buffer_data_lcd,0);
    OutPut74595(buffer_data_lcd);
    lcd_write( c );
}

void lcd_gotoxy(unsigned char row,unsigned
char col)
{
    bit_clear(buffer_data_lcd,0);
    OutPut74595(buffer_data_lcd);
    switch(row)
    {
        case 0 : lcd_write( 0x80 + col ) ;
        break ;
        case 1 : lcd_write( 0xC0 + col ) ;
        break ;
        case 2 : lcd_write( 0x94 + col ) ;
        break ;
        case 3 : lcd_write( 0xD4 + col ) ;
        break ;
    };
}

/* initialise the LCD - put into 4 bit mode */
void lcd_init()
{
    buffer_data_lcd = 0;
    OutPut74595(buffer_data_lcd);
    delay_ms(15); // wait 15mSec after power
applied,
    LCD_DATA(0x03);
    LCD_STROBE();
    delay_ms(5);
    LCD_STROBE();
    delay_us(200);
    LCD_STROBE();
    delay_us(200);
    LCD_DATA(2); // Four bit mode
    LCD_STROBE();
    lcd_write(0x28); // Set interface length
    lcd_write(0x0C); // Display On, Cursor
On, Cursor Blink
    lcd_clear(); // Clear screen
    lcd_write(0x06); // Set entry Mode
}

//-----
void OutPut74595(int8 data)
{
int8 i ;
for(i=0;i<8;i++)
{
    if( bit_test(data,7) )
    {
        data74595=1 ;
    }
    else
    {
        data74595=0 ;
    };
    clock74595=0;
    clock74595=1;

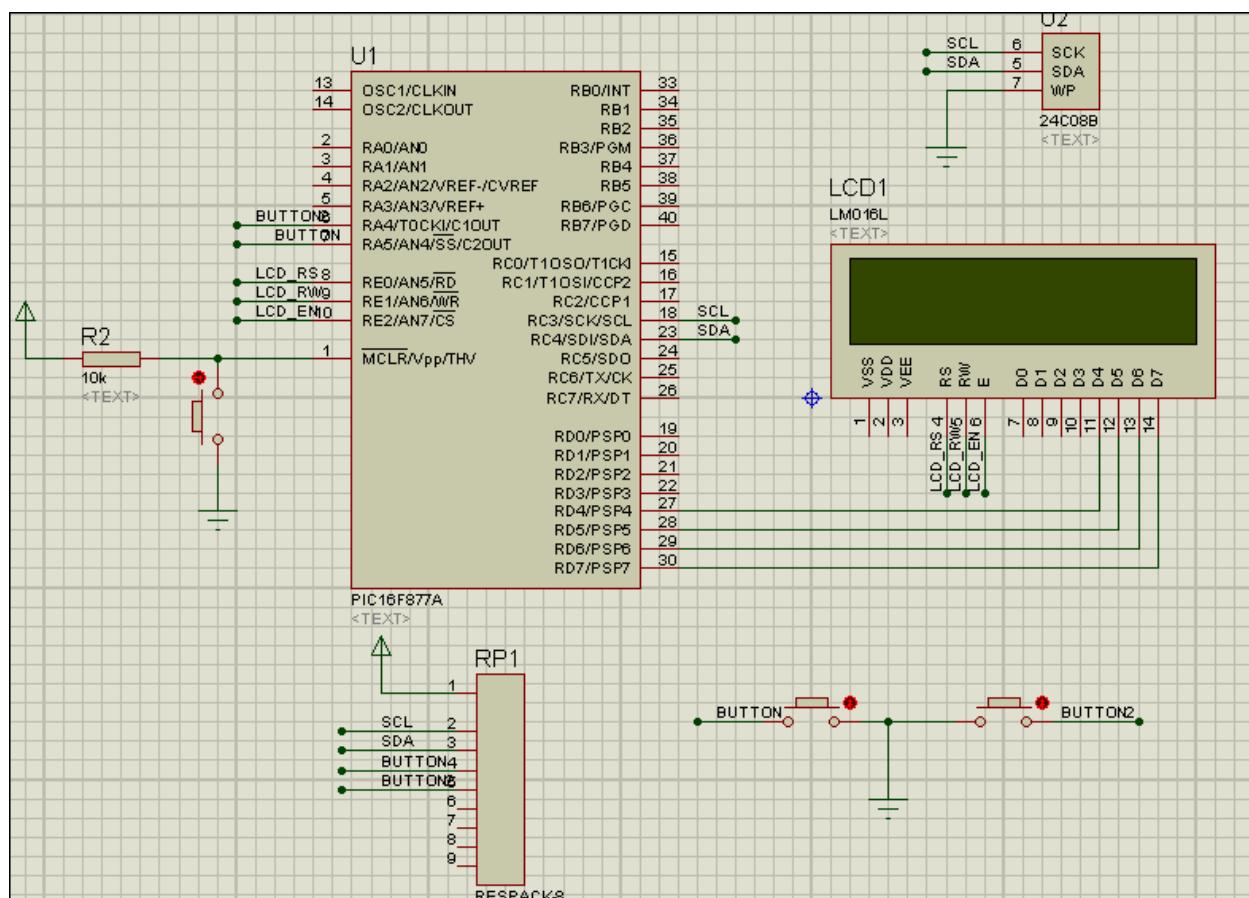
    data = data<<1 ;
};

}

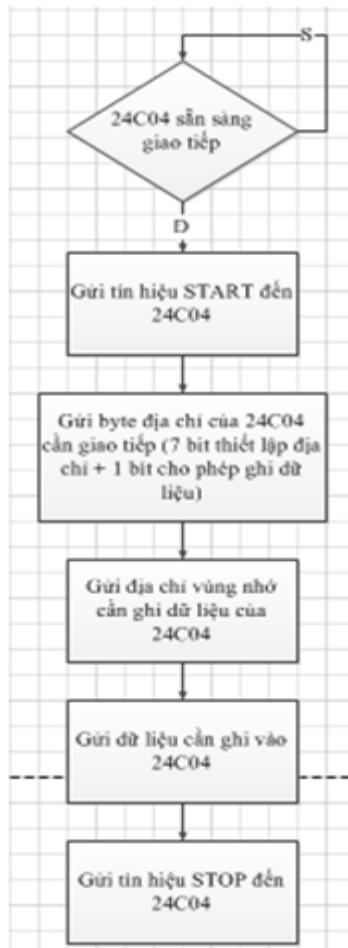
//-----
latch74595 = 1;
latch74595 = 0;
}
//-----
-----
```

### **6.22. Viết chương trình đọc và ghi dữ liệu vào EEPROM24XX.**

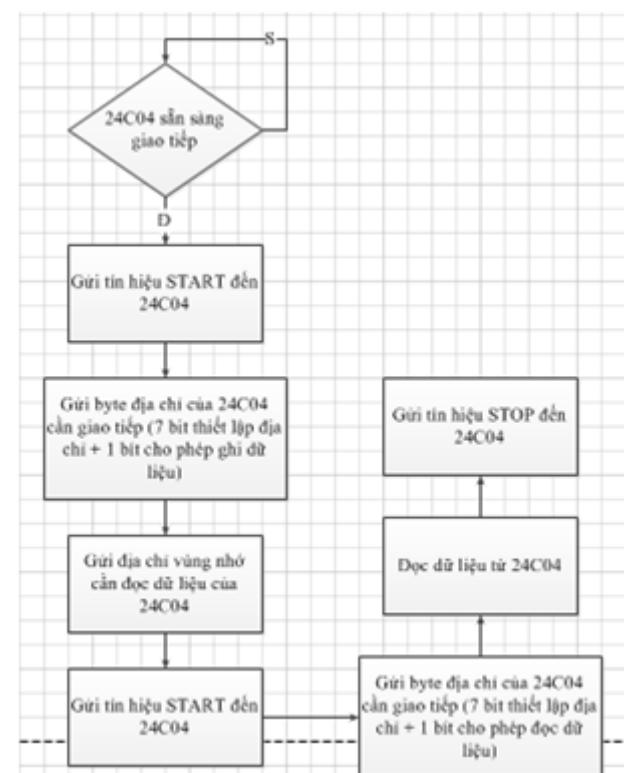
- ✓ **Phần cứng mô phỏng:**



Chú ý: khi kết nối EEPROM thực tế cần xem kỹ datasheet để kết nối cho đúng.



Các bước ghi dữ liệu vào EEPROM



Các bước đọc dữ liệu vào EEPROM

✓ **Code:**

```
#include <16F877A.h>
#fuses HS, NOWDT ,PROTECT ,NOBROWNOUT ,NOPUT
#use delay(clock=20000000)
#include <def_877a.h>
#define I2C_DATA  PIN_C4
#define I2C_CLK   PIN_C3
#use i2c(Master, Fast, sda=I2C_DATA, scl=I2C_CLK)
#include "LCD_NVN.c"
#include "24cxx.c"

void main()
{
    unsigned char Value;
    i2c_config();
    lcd_init(); // khai tao dung LCD
    lcd_gotoxy(1,1); // cho toa do dau tien x:1, y:1
    Printf(lcd_putc, " DEMO 24XX");
    Value=EEP_24CXX_Read(0x03);
    lcd_gotoxy(0,2); // cho toa do dau tien x:1, y:1
    printf(LCD_putc, "\nValue:%u", Value);

    while(TRUE)
    {
        if(input(BUTTON1)==0)
        {
            delay_ms(10);
            if(input(BUTTON1)==0)
            {
                Value++;
                EEP_24CXX_Write(0x03, Value);
                lcd_gotoxy(0,2); // cho toa do dau tien x:1, y:1
                printf(LCD_putc, "\nValue:%u", Value);
                while(input(BUTTON1)==0);
            }
        }
        if(input(BUTTON2)==0)
        {
            delay_ms(10);
            if(input(BUTTON2)==0)
            {
                Value=0;
                EEP_24CXX_Write(0x03, Value);
                lcd_gotoxy(0,2); // cho toa do dau tien x:1, y:1
            }
        }
    }
}
```

```
    printf(LCD_putc,"\\nValue:%u",Value);
    while(input(BUTTON2)==0);
}
}
}
}
```

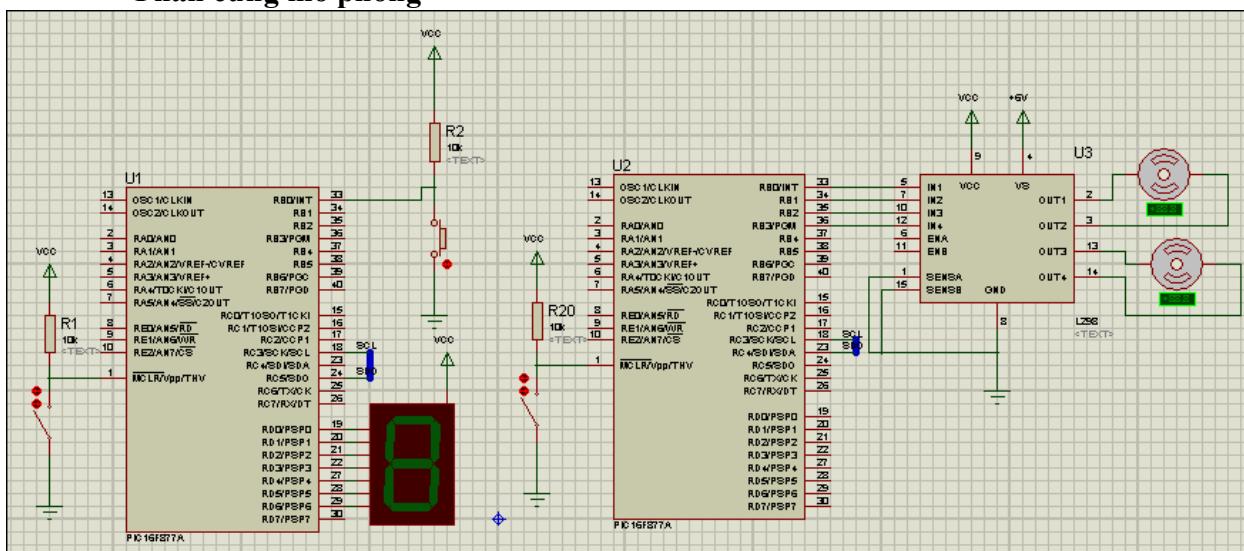
**File 24cxx.c:**

```
#ifndef __24CXX_
#define __24CXX_
int1 EEPROM_24C_Is_Ready(){
    int1 ack;
    i2c_start();
    ack = i2c_write(0xa0);
    i2c_stop();
    return !ack;
}
void EEP_24CXX_Write(unsigned int8 address, BYTE data)
{
    while(!EEPROM_24C_Is_Ready());
    i2c_start();
    i2c_write((0xa0|(BYTE)(address>>7))&0xfe);
    i2c_write(address);
    i2c_write(data);
    i2c_stop();
}
BYTE EEP_24CXX_Read(unsigned int8 address)
{
    BYTE data;
    while(!EEPROM_24C_Is_Ready());
    i2c_start();
    i2c_write((0xa0|(BYTE)(address>>7))&0xfe);
    i2c_write(address);
    i2c_start();
    i2c_write((0xa0|(BYTE)(address>>7))|1);
    data=i2c_read(0);
    i2c_stop();
    return(data);
}
void EEP_24CXX_WriteS(unsigned char address,unsigned char*s)
{
    while(*s)
    {
```

```
EEP_24CXX_Write(address++,*s);
s++;
}
}
void EEPROM_24CXX_ReadS(unsigned char address, unsigned char lenght, unsigned char *s)
{
    unsigned char i=0;
    while(lenght)
    {
        s[i++]=EEP_24CXX_Read(address++);
        lenght--;
    }
    s[++i]=0;
}
```

6.23. Viết chương trình điều khiển động cơ ở chip SLAVE thông qua chuẩn giao tiếp SPI 1 SLAVE – 1 MASTER.

## ✓ Phân cứng mô phỏng



✓ Code:

## Master:

```
#include <16f877a.h>
#FUSES NOWDT, HS, NOPUT,NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
int led7doan[10]={0b11000000,0b1111001,0b10100100,0b10110000,0b10011001,0b10010010,
0b10000010,0b11111000,0b10000000,0b10010000};
int i=0,t1,t2,t3,t4;
#endif
void ngat_negoai()
{
    i=i+1;
```

```
if(i>4)
{
    i=0;
}

switch (i){
    case 1:{  
        output_D(led7doan[1]);  
        t1=1;  
        ouput_low(pin_A5);  
        spi_write(t1);  
        delay_us(100);  
        break;  
    }  
  
    case 2:{  
        output_D(led7doan[2]);  
        t2=2;  
        ouput_low(pin_A5);  
        spi_write(t2);  
        delay_us(100);  
        break;  
    }  
    case 3:{  
        output_D(led7doan[3]);  
        t3=3;  
        ouput_low(pin_A5);  
        spi_write(t3);  
        delay_us(100);  
        break;  
    }  
  
    case 4:{  
        output_D(led7doan[4]);  
        t4=4;  
        ouput_low(pin_A5);  
        spi_write(t4);  
        delay_us(100);  
        break;  
    }  
  
    default:{  
        break;  
    }  
}
```

```
        }
    }

void main()
{
    setup_spi(spi_master |SPI_SS_DISABLED|spi_l_to_h |spi_clk_div_16);
    ext_int_edge(0,H_TO_L);
    enable_interrupts(INT_EXT);
    enable_interrupts(GLOBAL);
while(true)
{
}
```

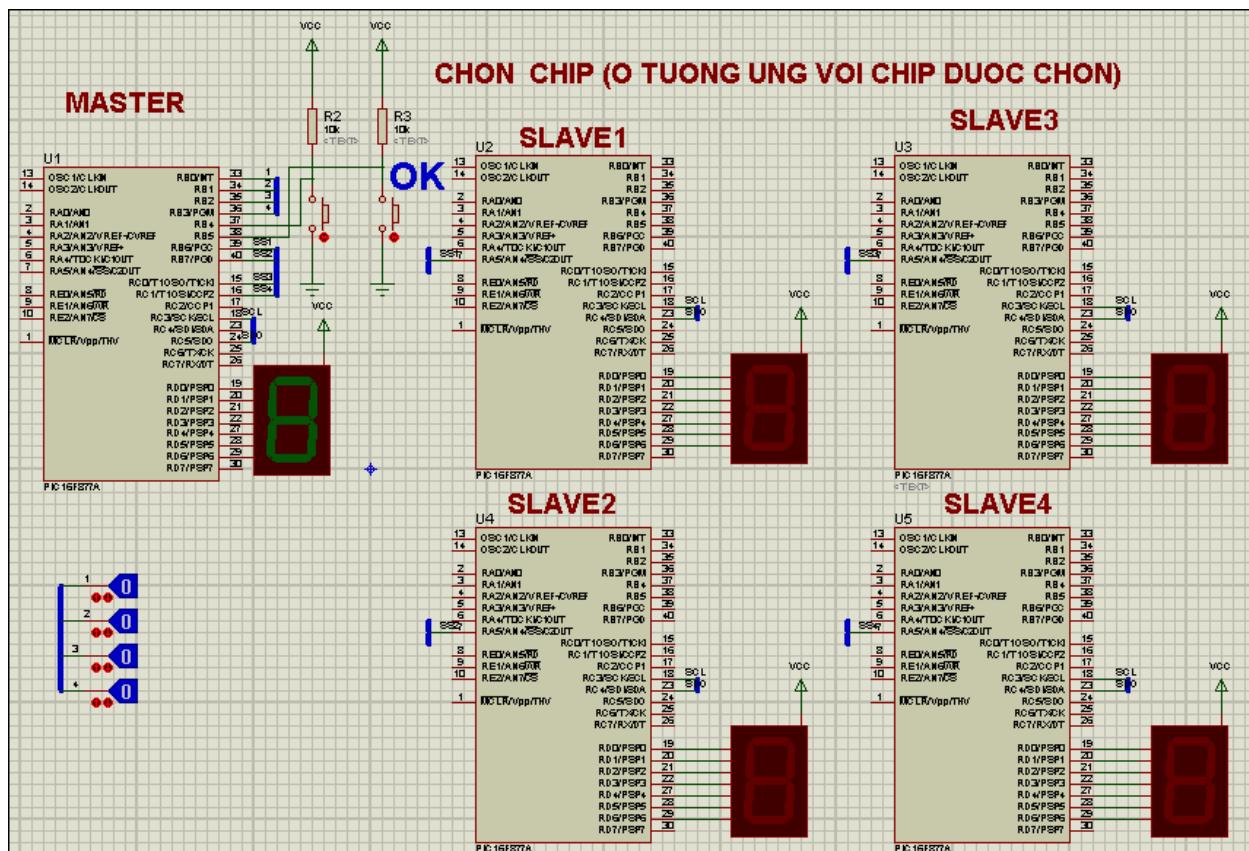
**Slave:**

```
#include <16f877a.h>
#FUSES NOWDT, HS, NOPUT,NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
int8 data_in;
#INT_SSP
void isr_ssp(){
    data_in=spi_read();
}
void main()
{
    setup_spi(spi_slave | spi_l_to_h |spi_clk_div_16);
    enable_interrupts(INT_SSP);
    enable_interrupts(GLOBAL);
while(true){
    switch (data_in){
        case 1:
            output_high(pin_B0);
            output_low(pin_B1);
            output_low(pin_B2);
            output_low(pin_B3);
            break;
        case 2:
            output_low(pin_B0);
            output_high(pin_B1);
            output_low(pin_B2);
            output_low(pin_B3);
            break;
        case 3:
```

```
        output_high(pin_B2);
        output_low(pin_B3);
        output_low(pin_B0);
        output_low(pin_B1);
        break;
    case 4:
        output_low(pin_B2);
        output_high(pin_B3);
        output_low(pin_B0);
        output_low(pin_B1);
        break;
    default:
        output_low(pin_B0);
        output_low(pin_B1);
        output_low(pin_B2);
        output_low(pin_B3);
        break;
    }
}
```

#### 6.24. Viết chương trình giao tiếp 1 Master 4 Slave thông qua chuẩn SPI.

- #### ✓ Phản ứng mô phỏng:



- ## ✓ Code

**Master:**

```
#include <16f877a.h>
#FUSES NOWDT, HS, NOPUT,NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
int8 a=0;
int led7doan[10]={0b11000000,0b1111001,0b10100100,0b10110000,0b10011001,0b10010010,
0b10000010,0b1111000,0b10000000,0b10010000};
#INT_RB
void ngat_ngoi(){
    if(input(pin_B4)==0){
        a=a+1;
        if(a>9){
            a=0;
        }
        output_D(led7doan[a]);
    }

    if(input(pin_B5)==0){
        if(input(pin_B0)==0){
            output_low(pin_B6);
        }
        else{
            output_high(pin_B6);
        }
    /////////////////
    if(input(pin_B1)==0){
        output_low(pin_B7);
    }
    else{
        output_high(pin_B7);
    }
    ///////////////
    if(input(pin_B2)==0){
        output_low(pin_C0);
    }
    else{
        output_high(pin_C0);
    }
    ///////////////
    if(input(pin_B3)==0){
        output_low(pin_C1);
    }
    else{
```

```
        output_high(pin_C1);
    }
    spi_write(a);
    delay_us(200);
}
}

void main()
{
    setup_spi(spi_master| spi_l_to_h |spi_clk_div_16);
    enable_interrupts(INT_RB);
    enable_interrupts(GLOBAL);
    output_D(led7doan[0]);
    while(true)
    {
    }
}
```

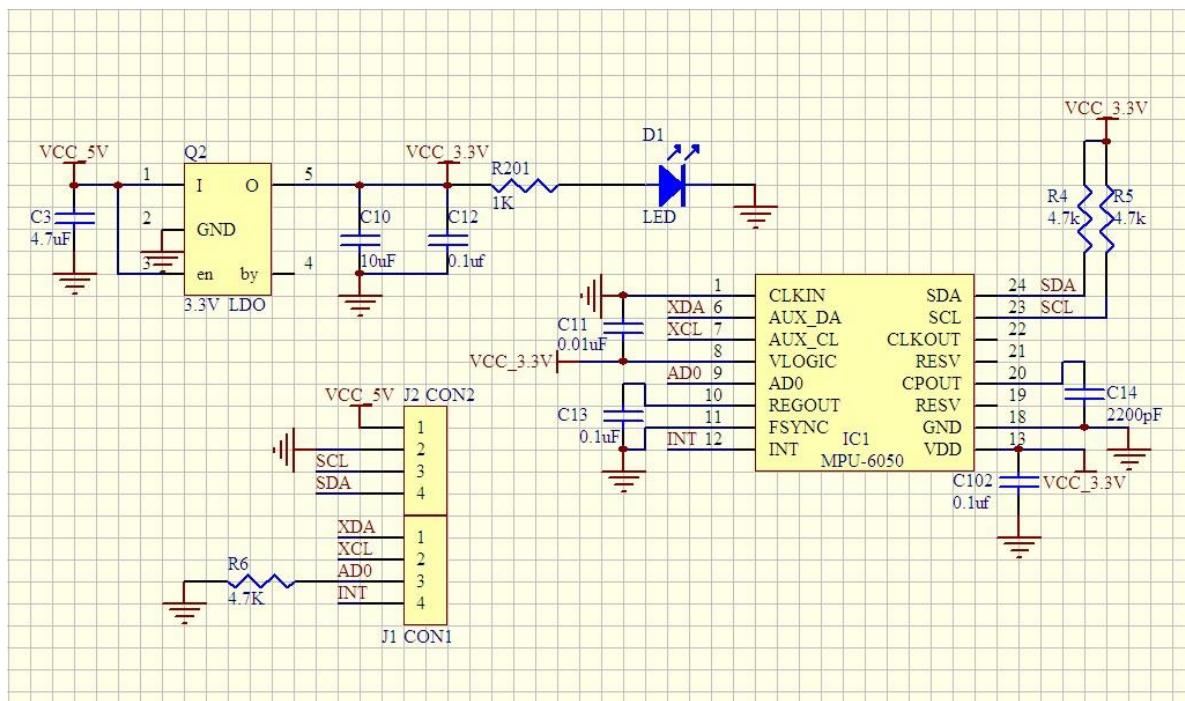
**Slave:**

```
#include <16f877a.h>
#FUSES NOWDT, HS, NOPUT,NOPROTECT, NODEBUG, NOBROWNOUT, NOLVP,
NOCPD, NOWRT
#use delay(clock=20000000)
int led7doan[10]={0b11000000,0b1111001,0b10100100,0b10110000,0b10011001,0b10010010,
0b10000010,0b1111000,0b10000000,0b10010000};
int8 data_in;
#INT_SSP
void isr_ssp(){
    data_in=spi_read();
    output_D(led7doan[data_in]);
}

void main(){
    setup_spi(spi_slave);
    enable_interrupts(INT_SSP);
    enable_interrupts(GLOBAL);
    while(true){
    }
}
```

**6.25. Viết chương trình đọc dữ liệu MCU6050 thông qua chuẩn giao tiếp I2C.**

- ✓ Phần cứng MCU 6050



✓ Kết nối với vi điều khiển:



MPU-6050 tích hợp 6 trục cảm biến bao gồm:

- + cảm biến quay hồi chuyển 3 trục (3-axis MEMS gyroscope)
  - + cảm biến gia tốc 3 chiều (3-axis MEMS accelerometer)

Ngoài ra, MPU-6050 còn có 1 đơn vị tăng tốc phần cứng chuyên xử lý tín hiệu (Digital Motion Processor - DSP) do cảm biến thu thập và thực hiện các tính toán cần thiết. Điều này giúp giảm bớt đáng kể phần xử lý tính toán của vi điều khiển, cải thiện tốc độ xử lý và cho ra phản hồi nhanh hơn. Đây chính là 1 điểm khác biệt đáng kể của MPU-6050 so với các cảm biến gia tốc và gyro khác.

MPU-6050 có thể kết hợp với cảm biến từ trường (bên ngoài) để tạo thành bộ cảm biến 9 góc đầy đủ thông qua giao tiếp I2C.

Các cảm biến bên trong MPU-6050 sử dụng bộ chuyển đổi tương tự - số (Analog to Digital Converter - ADC) 16-bit cho ra kết quả chi tiết về góc quay, tọa độ... Với 16-bit bạn sẽ có  $2^{16} = 65536$  giá trị cho 1 cảm biến.

Tùy thuộc vào yêu cầu của bạn, cảm biến MPU-6050 có thể hoạt động ở chế độ tốc độ xử lý cao hoặc chế độ đo góc quay chính xác (chậm hơn). MPU-6050 có khả năng đo ở phạm vi:

- + con quay hồi chuyển:  $\pm 250$  500 1000 2000 dps  
+ gia tốc:  $\pm 2 \pm 4 \pm 8 \pm 16$ g

Hơn nữa, MPU-6050 có sẵn bộ đệm dữ liệu 1024 byte cho phép vi điều khiển phát lệnh

cho cảm biến, và nhận về dữ liệu sau khi MPU-6050 tính toán xong.

Con quay hồi chuyển là cảm biến rất thông dụng nắm xác định vị trí, tọa độ robot, tích hợp trong các điện thoại Smartphone hiện nay.

- + Nguồn: 3-5V, trên module MPU-6050 đã có sẵn LDO chuyển nguồn 5V -> 3V
- + Giao tiếp I2C ở mức 3V
- + Khoảng cách chân cảm: 2.54mm
- + Địa chỉ: 0x68, có thể cấp mức cao vào chân AD0 để chuyển địa chỉ thành 0x69

✓ **Code:**

```
#include <main.h>
#include "def_887.h"
#include "lcdspi.h"
#include "GY_521.h"
void main() {
    set_tris_b(0x00);
    //set_tris_d(0x00);
    set_tris_a(0xff);
    //set_tris_c(0x00);
    // TRISB=0;
    init_sensor_Gyro_MP6050();
    lcd_init();
    lcd_gotoxy(1,1);
    printf(lcd_putchar, "OK");
    delay_ms(100);
    //***** int timer 0 *****/
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    set_timer0(6) ;
    enable_interrupts(INT_TIMER0);
    // enable_interrupts(global);
    char aay=0;
    while(true){
        team=(i2c_readreg(MPU6050_ADD,DATATH)*256) |
        (i2c_readreg(MPU6050_ADD,DATATL));
        t1 = (double)(team / 340.00);
        Ax=(i2c_readreg(MPU6050_ADD,AccXH)*256) |
        (i2c_readreg(MPU6050_ADD,AccXL));
        Ay=(i2c_readreg(MPU6050_ADD,AccYH)*256) |
        (i2c_readreg(MPU6050_ADD,AccYL));
        Az=(i2c_readreg(MPU6050_ADD,AccZH)*256) |
        (i2c_readreg(MPU6050_ADD,AccZL));
        Gx=(i2c_readreg(MPU6050_ADD,GyroXH)*256) |
        (i2c_readreg(MPU6050_ADD,GyroXL));
        Gy=(i2c_readreg(MPU6050_ADD,GyroYH)*256) |
        (i2c_readreg(MPU6050_ADD,GyroYL));
```

```
Gz=(i2c_readreg(MPU6050_ADD,GyroZH)*256) |  
(i2c_readreg(MPU6050_ADD,GyroZL));  
  
    lcd_gotoxy(0,0);  
    printf(lcd_putchar, "%4ld", ax/100);  
    lcd_gotoxy(0,6);  
    printf(lcd_putchar, "%4ld", ay/100);  
    lcd_gotoxy(0,12);  
    printf(lcd_putchar, "%4ld", az/100);  
    lcd_gotoxy(1,0);  
    printf(lcd_putchar, "%4ld", gx/100);  
    lcd_gotoxy(1,6);  
    printf(lcd_putchar, "%4ld", gy/100);  
    Delay_ms(200);  
}  
}
```

### File GY\_521.h

```
#define MPU6050_ADD (0x68)  
#include <stdio.h>  
#include <math.h>  
#define AccXH 59  
#define AccXL 60  
#define AccYH 61  
#define AccYL 62  
#define AccZH 63  
#define AccZL 64  
  
#define DATATH 65  
#define DATATL 66  
  
#define GyroXH 67 // dec -> hex 0x43  
#define GyroXL 68  
#define GyroYH 69  
#define GyroYL 70  
#define GyroZH 71  
#define GyroZL 72  
#define WHO_AM_I 0x75 //  
  
Signed int16 Ax,Ay,Az,Gx,Gy,Gz;  
Signed int32 goc1=0, sum_ay=0;  
Signed int16 t1, team;  
int16 goc=0, goc2=0;
```

```

void i2c_writereg(int8 device, int8 address, int8 val);
int8 i2c_readreg(int8 device, int8 address);

void init_sensor_Acc_MPU6050(void)
{
    delay_ms(10);
    //*****init_sensor_Acc_MPU6050*****
    // write(DEVICE1,0x6B,0x10);
    // i2c_writereg(MPU6050_ADD,0x1c,0x10);
    //0x00: +-2g 16384 LSB/g
    //0x08: +-4g 8192 LSB/g
    //0x10: +-8g 4096 LSB/g
    //0x18: +-16g 2048 LSB/g
    // acc_1G=512;
    // i2c_writereg(MPU6050_ADD,0x6a,0b00100000);
    // i2c_writereg(MPU6050_ADD,0x37,0x00);
}

void init_sensor_Gyro_MPU6050(void)
{
    //*****init_sensor_Gyro_MPU6050 *****
    i2c_writereg(MPU6050_ADD,0x6b,0x80); //reset device, reset cam bien
    delay_ms(5);
    i2c_writereg(MPU6050_ADD,0x6b,0x00); //cam bien hoat dong
    i2c_writereg(MPU6050_ADD,0x19,0x00); //Sample Rate Divider = 0; cai nay minh khong
    hieu lam. @@
    i2c_writereg(MPU6050_ADD,0x1a,0x00); //Configuration;
    i2c_writereg(MPU6050_ADD,0x1b,0x10); //Gyroscope Configuration; 0-8-10-18 ; do phan
    giao Gyro la 1000do/s
    i2c_writereg(MPU6050_ADD,0x6a,0x00); //User Control; Con MPU6050 co kha nang lam
    master
        //cua 1 so con co chuan truyen I2C khac, o day khong dung nen khong config
    i2c_writereg(MPU6050_ADD,0x37,0x00);
}

void i2c_writereg(int8 device, int8 address, int8 val)
{
    i2c_start();
    i2c_write(device<<1);
    i2c_write(address);
    i2c_write(val);
    i2c_stop();
}

```

}

```

int8 i2c_readreg(int8 device, int8 address)
{
    int8 value;
    i2c_start();
    i2c_write(device<<1);
    i2c_write(address);
    i2c_start();
    i2c_write(device<<1|0x01);
    value=i2c_read(0);
    i2c_stop();
    return value;
}

//ham getdata tra ve gia tri 16bit
int16 GetData(int8 address){
    char H,L;
    H=i2c_write(address);
    L=i2c_write(address+1);
    //return (H<<8)+L; // ccs ko chay lenh nayf
} // return (i2c_write(address)*256| i2c_write(address+1));

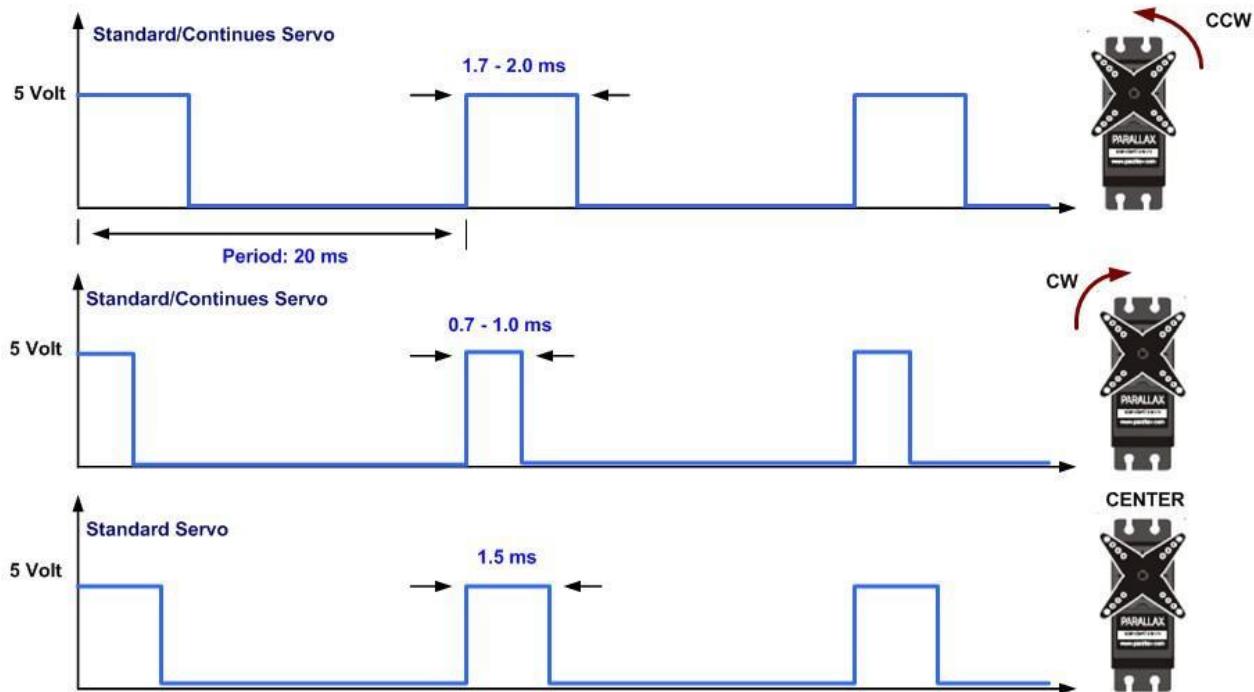
```

### 6.26. Viết chương trình điều khiển động cơ RCSERVO.

✓ Giới thiệu RCSERVO:  
 RC servo là thiết bị có gắn động cơ mini và các hệ bánh răng giảm tốc, mục đích giúp trực quay đến góc mong muốn mà vẫn giữ được góc chính xác và không bị trôi lệch, ngoài ra RC servo còn có khả năng chịu tải lớn hơn rất nhiều lần so với trọng lượng của nó. RC servo thường được ứng dụng trong robotic, đồ chơi mô hình RC (máy bay, xe, thuyền, ...)

Hoạt động của RC servo dựa trên nguyên lý nhận xung PWM và cho ra góc quay. RC servo chỉ có thể xoay ở góc cố định có nghĩa là không thể xoay quanh trực như những loại động cơ bình thường. Tùy loại RC servo mà góc quay hoạt động được 90o hay 180o, đa phần thì 90o.

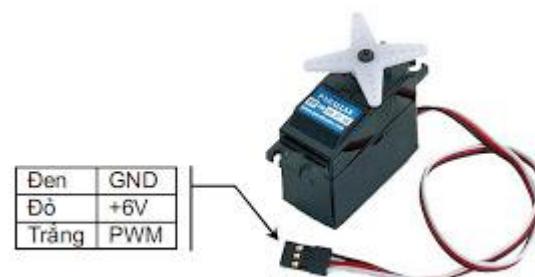




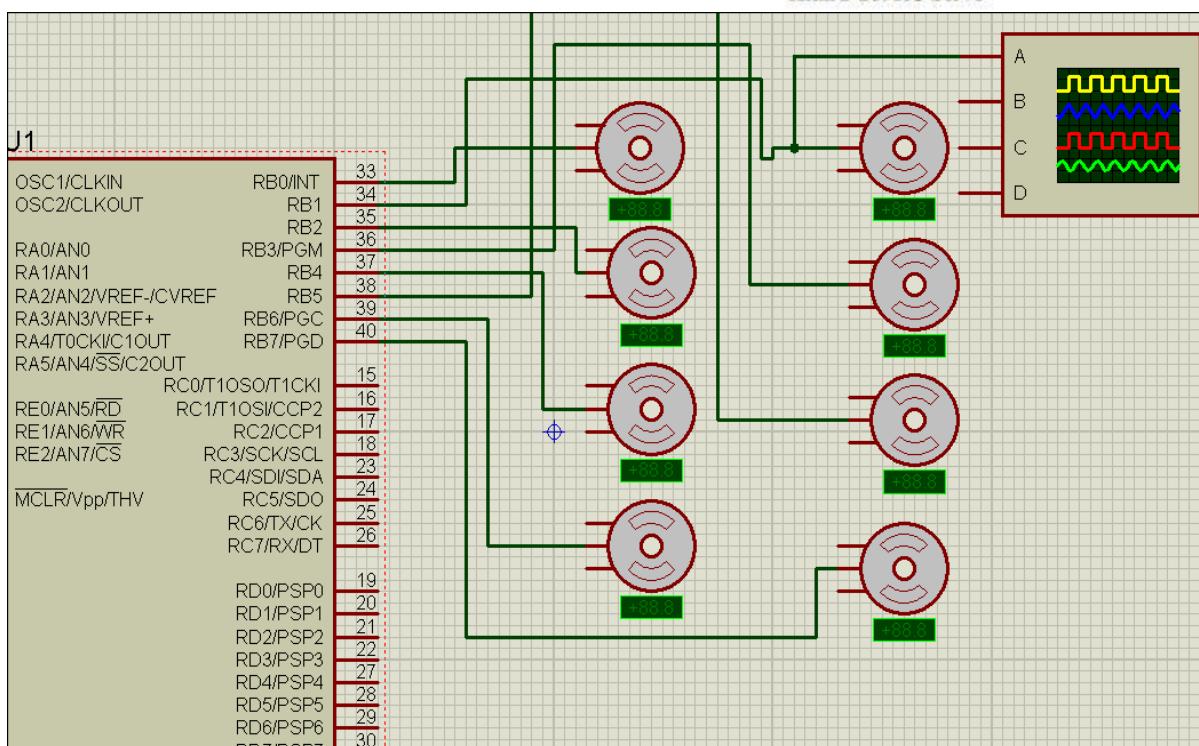
Nhìn hình trên ta thấy chu kỳ của PWM là 20ms (50 Hz), thời gian mức cao sẽ quyết định góc quay của RC servo dao động từ 1ms -> 2ms  $\Leftrightarrow$  0 -> 90o.

✓ Kết nối với vi điều khiển:

- + Dây đỏ: VCC từ 4-7v.
- + Dây đen: kết nối 0v.
- + Dây trắng kết nối chân tạo xung PWM.



Hình 2-28. RC Servo



✓ Code 1:

```

    {
        output_high(SERVO_PIN_0);
        output_low(SERVO_PIN_7);

        //set_servo_pin_high = FALSE;
        CCP_1 += pulse_high_duration_0;
        current_slot = 1;
        break;
    }

case 1:
{
    output_high(SERVO_PIN_1);
    output_low(SERVO_PIN_0);
    //set_servo_pin_high = FALSE;
    CCP_1 += pulse_high_duration_1;
    current_slot = 2;
    break;
}

case 2:
{
    output_high(SERVO_PIN_2);
    output_low(SERVO_PIN_1);
    //set_servo_pin_high = FALSE;
    CCP_1 += pulse_high_duration_2;
    current_slot = 3;
    break;
}

else // If high portion of signal is done, do the low part.
{
    output_low(SERVO_PIN_0);
    output_low(SERVO_PIN_1);
    output_low(SERVO_PIN_2);
    set_servo_pin_high = TRUE;
    CCP_1 += (PWM_PERIOD - pulse_high_duration_7);

    pulse_done_flag = TRUE;
}
}

//=====
void main()

```

```
{  
int16 i;  
  
setup_timer_1(T1_DIV_BY_8 | T1_INTERNAL);  
set_timer1(0);  
  
// Setup CCP to generate CCP1 interrupt when Timer1  
// matches the value in the CCP1 registers.  
setup_ccp1(CCP_COMPARE_INT);  
output_low(SERVO_PIN_0);  
output_low(SERVO_PIN_1);  
  
// Initialize the variables and CCP1 to give a 1 ms servo pulse.  
//pulse_high_duration = PULSE_1MS;  
CCP_1 = PWM_PERIOD - pulse_high_duration_1; // Pulse Off time  
pulse_done_flag = FALSE;  
  
clear_interrupt(INT_CCP1);  
enable_interrupts(INT_CCP1);  
enable_interrupts(GLOBAL);  
while(1)  
{  
//! for(i = PULSE_1MS; i < PULSE_2MS; i++)  
//! {  
//!     while(!pulse_done_flag); // Wait until pulse is done  
//!  
//!     pulse_done_flag = FALSE;  
//!     disable_interrupts(GLOBAL);  
//!     pulse_high_duration = i;  
//!     enable_interrupts(GLOBAL);  
//! }  
  
pulse_high_duration_0=625;  
pulse_high_duration_1=1000;  
pulse_high_duration_2=1000;  
delay_ms(500);  
while(!pulse_done_flag); // Wait until pulse is done  
  
pulse_high_duration_0=500;  
pulse_high_duration_1=700;  
pulse_high_duration_2=1200;  
  
while(!pulse_done_flag); // Wait until pulse is done  
delay_ms(500);
```

```

        output_toggle(PIN_C1);
    }

}

```

**✓ Code 2:**

```

// Software use timer_1 and CCP1 in MCU.

#include <16f877A.h>
#fuses HS, NOWDT, NOPROTECT, BROWNOUT, PUT, NOLVP
#use     delay(clock=20M)

//-----General config-----
#define SERVO_MAX      5           //Servo max
#define SERVO_MAX_VALUE 2500       //Max of pulse width micro sec(uS) =25ms
#define SERVO_MIN_VALUE 500        //Min of pulse width micro sec(uS) =0.5ms

typedef struct tServo{
    volatile unsigned pin;
    volatile unsigned Active;
    volatile unsigned int16 value;
}Servo;

//-----Khai bao ham con-----
void ServoActive(unsigned chanel,unsigned vPin,unsigned int16 vValue);
void ServoValue(unsigned chanel,unsigned int16 value); //value is uS.

//-----Bien toan cuc-----
signed ServoCnt=0;
Servo nServo[SERVO_MAX];
//-----ngat CCP1-----
#INT_CCP1
void ServoTask()
{
    //if enc of servo count ,reset timer1.
    if(ServoCnt<0){
        set_timer1(0);
    }else{
        //if server pin is active set output servo pin to low.
        if(nServo[ServoCnt].Active)
            output_low(nServo[ServoCnt].pin);
    }
    ServoCnt++;
    if(ServoCnt<SERVO_MAX)
    {

```

```
//set interrupt to next servo time.  
CCP_1 = get_timer1() + nServo[ServoCnt].value;  
//if servo pin is active set output servo pin to high.  
if(nServo[ServoCnt].Active)  
    output_high(nServo[ServoCnt].pin);  
}else{  
    //set fist chanel to start scan.  
    CCP_1 = get_timer1() + 100;  
    ServoCnt      = -1;  
}  
}  
  
/*  
Example : ServoActive(0,PIN_A0,1500);  
    set PIN_A0 to connect servo at chanel '0' and set pulse width 1500 uS.  
*/  
void ServoActive(unsigned channel,unsigned vPin,unsigned int16 value)  
{  
    nServo[channel].pin      =      vPin;  
    nServo[channel].value   =   ((value*5)/8);  
    nServo[channel].Active  =  1;  
}  
/*  
Example : ServoValue(0,2400);  
    set servo chanel '0' to pulse width 2400 uS. (2.4mS)  
*/  
void ServoValue(unsigned channel,unsigned int16 value)  
{  
    if(value>SERVO_MAX_VALUE)  
        value=SEROVO_MAX_VALUE;  
    if(value<SERVO_MIN_VALUE)  
        value=SEROVO_MIN_VALUE;  
  
    if (channel>=SERVO_MAX)  
        channel=SEROVO_MAX-1;  
  
    nServo[channel].value   =   ((Value*5)/8);  
}  
void main()  
{  
    unsigned i;  
    //-----Setup timer1 to servo task-----//  
    setup_timer_1 (T1_INTERNAL|T1_DIV_BY_4);  
    setup_ccp1      (CCP_COMPARE_INT);
```

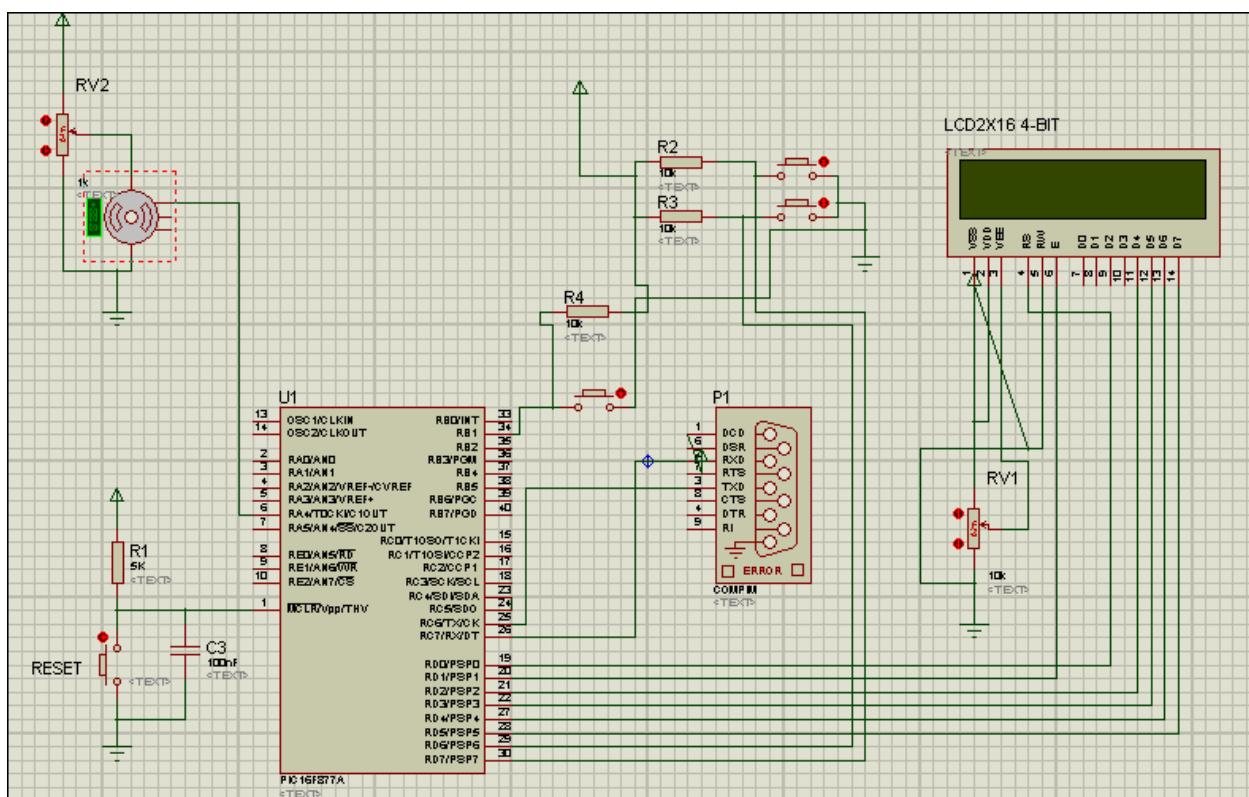
```
enable_interrupts(GLOBAL);
enable_interrupts(INT_CCP1);
//----- //

//Add pin_d0 to connect servo chanel '0'
ServoActive(0,PIN_A0,1500);

ServoActive(1,PIN_B0,1500);
ServoActive(2,PIN_B1,1500);
ServoActive(3,PIN_C1,1500);
delay_ms(1000);
while(true)
{
    ServoActive(0,PIN_A0,1500);
    delay_ms(500);
    ServoActive(0,PIN_A0,2500);
    delay_ms(500);
}
```

#### 6.27. Viết chương trình đo tốc độ động cơ dùng Timer0 chế độ counter.

- ✓ Phản ứng mô phỏng:



- ✓ Code:

```
#include <16F877A.h>
#fuses HS, NOWDT ,PROTECT ,NOBROWNOUT ,NOPUT
#use delay(clock=20000000)
```

```
#include <def_877a.h>
#use rs232(baud=9600,parity=N,xmit=PIN_A3,rcv=PIN_A2,bits=8)
#include <LCD_NVN.c>
#define INTS_PER_SECOND1 19;
signed int16 xung,so_vong,int_count1,t0;

void init_t0(void)
{
    //khai_bao_ngat();
    int_count1 = INTS_PER_SECOND1;//10 lan 1 giay
    setup_timer_0(RTCC_DIV_1|RTCC_EXT_H_TO_L); // Timer0 is Counter
    set_timer0(0);
    set_timer1(0);

    setup_timer_1(T1_INTERNAL | T1_DIV_BY_4); // Timer1 is Timer
    enable_interrupts(INT_RTCC);
    enable_interrupts(INT_TIMER1);
}

#define INT_TIMER0 // ngat timer0 tang bien dem len 1
void timer0_int()
{
    xung++;
}

#define INT_TIMER1 // Chuong trinh ngat Timer 1
void ngat_timer_1 ()
{
    // Ham duoc goi khi Timer1 tran (65535->0)
    // Xap xi 19 lan / giay
    if(--int_count1==0)
    {
        int_count1 = INTS_PER_SECOND1;
        so_vong = xung*256*10^6+get_timer0();// xung tran =256-0, +get_timer0(thanh ghi Trm0)
        do thoi gian..
        xung = 0;
        set_timer0(0);
        set_timer1(0);
    }
}
void main()
{
    lcd_init();// khai tao dung LCD
    // WELLCOME
    lcd_gotoxy(17,1);// cho toa do dau tien x:1, y:1
```

```

printf(lcd_putc,"PHONG NGHIEN CUU NVN"); // in dong chu LCD
lcd_moveRIGHT(40);
DELAY_MS(200);
output_b(0x0ff);
xung=(0);
output_high(pin_d3);
port_b_pullups(true);
//CHO PHEP NGAT TOAN CUC
enable_interrupts(INT_RDA);//CHO NGAT RD
enable_interrupts(GLOBAL);//CHO PHEP NGAT
xung= 0;
init_t0();// khai bao dung ngat timer

while(true) {
    lcd_gotoxy(1,1);
    printf(lcd_putc,"PIC Training NVN");
    //----- hien thi 1 bien ra lcd-----
    lcd_gotoxy(1,2);// cho toa do dau tien x:1, y:1
    printf(LCD_putc,"Speed:%5ld v/pd",so_vong);
}

```

6.27. Viết chương trình điều khiển tốc độ động cơ dùng thuật toán PID.

```

#include <16F877A.h>
#fuses HS, NOWDT ,PROTECT ,NOBROWNOUT ,NOPUT
#use delay(clock=20000000)
#include <def_877a.h>

//Dinh nghia cac duong dieu khien motor
#define Sampling_time      50 //thoi gian lay mau (ms)
#define inv_Sampling_time  20 // 1/Sampling_time
#define PWM_Period         1023

unsigned int16 Pulse=0,pre_Pulse=0;
signed long rSpeed=0,Err=0 , pre_Err=0; //thanh phan PID
float Kp=5, Kd=0.6, Ki=2.4; //thanh phan PID
float pPart=0, iPart=0, dPart=0; //thanh phan PID
unsigned long Ctrl_Speed=10;   //van toc can dieu khien (desired speed)
unsigned int16 Output;// absrSpeed;
unsigned char sample_count=0;
void Motor_Speed_PID(long int des_Speed);

#define int_EXT
void EXT_isr(void) {

```

```
if(!input(PIN_B0))Pulse++;
else Pulse--;
}

#define TIMER1
void TIMER1_isr(void)
{
    Set_timer1(3035);//25ms
    sample_count++;
    Motor_Speed_PID(Ctrl_Speed);
}

void Motor_Speed_PID(unsigned long des_Speed){
    rSpeed=Pulse-pre_Pulse; //tinh van toc (trong sampling time)
    pre_Pulse=Pulse;
    Err=des_Speed-abs(rSpeed); //tinh error (loi)
    pPart=Kp*Err;
    dPart=Kd*(Err-pre_Err)*inv_Sampling_time;
    iPart+=Ki*Sampling_time*(Err+pre_Err)/2000;
    Output +=pPart+dPart+iPart; //cong thuc duoc bien doi vi la dieu khien van toc
    if (Output >PWM_Period) Output=PWM_Period-1;
    if (Output <=0) Output=1;
    set_pwm1_duty((int16)Output); //gan duty cycle cho CCP1 update PWM
    pre_Err=Err; //luu lai gia tri error
}

void main()
{
    set_tris_a(0xff);
    //set_tris_b(0x01);
    setup_spi(SPI_SS_DISABLED);
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
    setup_timer_1(T1_INTERNAL|T1_DIV_BY_2);
    setup_timer_2(T2_DIV_BY_16,255,1);
    setup_ccp1(CCP_PWM);
    setup_comparator(NC_NC_NC_NC);
    setup_vref(FALSE);
    enable_interrupts(INT_EXT);
    ext_int_edge( H_TO_L ); // Sets up EXT
    enable_interrupts(INT_TIMER1);
    enable_interrupts(GLOBAL);
    while(true) {
    }
}
```

**PHỤ LỤC****File def\_877a.h**

```

//DINH NGHIA BIT
// register definitions
#define W 0           #bit RA5      =0x05.5
#define F 1           #bit RA4      =0x05.4
// register files
#define INDF         =0x00       #bit RA3      =0x05.3
#define TMR0         =0x01       #bit RA2      =0x05.2
#define PCL          =0x02       #bit RA1      =0x05.1
#define STATUS        =0x03       #bit RA0      =0x05.0
#define FSR          =0x04
#define PORTA        =0x05       #bit RB7      =0x06.7
#define PORTB        =0x06       #bit RB6      =0x06.6
#define PORTC        =0x07       #bit RB5      =0x06.5
#define PORTD        =0x08       #bit RB4      =0x06.4
#define PORTE        =0x09       #bit RB3      =0x06.3
#define EEDATA        =0x10C
#define EEADR        =0x10D      #bit RB2      =0x06.2
#define EEDATH        =0x10E      #bit RB1      =0x06.1
#define EEADRH        =0x10F      #bit RB0      =0x06.0
#define ADCON0        =0x1F
#define ADCON1        =0x9F
#define ADRESH        =0x9F
#define ADSESL        =0x9F
#define PCLATH        =0x0a
#define INTCON        =0x0b       #bit RC7      =0x07.7
#define PIR1          =0x0c       #bit RC6      =0x07.6
#define PIR2          =0x0d       #bit RC5      =0x07.5
#define PIE1          =0x8c       #bit RC4      =0x07.4
#define PIE2          =0x8d       #bit RC3      =0x07.3
#define RC2           =0x9F       #bit RC2      =0x07.2
#define RC1           =0x9F       #bit RC1      =0x07.1
#define RC0           =0x9F       #bit RC0      =0x07.0
#define OPTION_REG    =0x81
#define TRISA          =0x85       #bit RD7      =0x08.7
#define TRISB          =0x86       #bit RD6      =0x08.6
#define TRISC          =0x87       #bit RD5      =0x08.5
#define TRISD          =0x88       #bit RD4      =0x08.4
#define TRISE          =0x89       #bit RD3      =0x08.3
#define RD2           =0x8a       #bit RD2      =0x08.2
#define RD1           =0x8b       #bit RD1      =0x08.1
#define RD0           =0x8c       #bit RD0      =0x08.0
#define RE2           =0x09.2
#define RE1           =0x09.1
#define RE0           =0x09.0
#define TRISA5         =0x85.5
#define TRISA4         =0x85.4
#define TRISA3         =0x85.3
#define TRISA2         =0x85.2
#define TRISA1         =0x85.1

```

#bit TRISA0 = 0x85.0	#bit tmr2if = 0x0c.1
#bit TRISB7 = 0x86.7	#bit tmrlif = 0x0c.0
#bit TRISB6 = 0x86.6	
#bit TRISB5 = 0x86.5	//PIR2 for C
#bit TRISB4 = 0x86.4	#bit cmif = 0x0d.6
#bit TRISB3 = 0x86.3	#bit eeif = 0x0d.4
#bit TRISB2 = 0x86.2	#bit bclif = 0x0d.3
#bit TRISB1 = 0x86.1	#bit ccp2if = 0x0d.0
#bit TRISB0 = 0x86.0	
#bit TRISC7 = 0x87.7	// PIE1 for C
#bit TRISC6 = 0x87.6	#bit adie = 0x8c.6
#bit TRISC5 = 0x87.5	#bit rcie = 0x8c.5
#bit TRISC4 = 0x87.4	#bit txie = 0x8c.4
#bit TRISC3 = 0x87.3	#bit ssPie = 0x8c.3
#bit TRISC2 = 0x87.2	#bit CCP1IE = 0x8c.2
#bit TRISC1 = 0x87.1	#bit tmr2ie = 0x8c.1
#bit TRISC0 = 0x87.0	#bit tmrlie = 0x8c.0
#bit TRISD7 = 0x88.7	
#bit TRISD6 = 0x88.6	//PIE2 for C
#bit TRISD5 = 0x88.5	#bit osfie = 0x8d.7
#bit TRISD4 = 0x88.4	#bit cmie = 0x8d.6
#bit TRISD3 = 0x88.3	#bit eeie = 0x8d.4
#bit TRISD2 = 0x88.2	// OPTION Bits
#bit TRISD1 = 0x88.1	#bit not_rbpU = 0x81.7
#bit TRISD0 = 0x88.0	#bit intedg = 0x81.6
#bit TRISE2 = 0x89.2	#bit t0cs = 0x81.5
#bit TRISE1 = 0x89.1	#bit t0se = 0x81.4
#bit TRISE0 = 0x89.0	#bit psa = 0x81.3
// INTCON Bits for C	#bit ps2 = 0x81.2
#bit gie = 0x0b.7	#bit ps1 = 0x81.1
#bit peie = 0x0b.6	#bit ps0 = 0x81.0
#bit tmr0ie = 0x0b.5	
#bit int0ie = 0x0b.4	// EECON1 Bits
#bit rbie = 0x0b.3	#bit eepgd = 0x18c.7
#bit tmr0if = 0x0b.2	#bit free = 0x18C.4
#bit int0if = 0x0b.1	#bit wrerr = 0x18C.3
#bit rbif = 0x0b.0	#bit wren = 0x18C.2
// PIR1 for C	#bit wr = 0x18C.1
#bit pspif = 0x0c.7	#bit rd = 0x18C.0
#bit adif = 0x0c.6	
#bit rcif = 0x0c.5	//ADCON0
#bit txif = 0x0c.4	#bit CHS0 = 0x1F.3
#bit sspif = 0x0c.3	#bit CHS1 = 0x1F.4
#bit ccp1if = 0x0c.2	#bit CHS2 = 0x1F.5

**File LCD\_NVN.h**

```

// Lcd 16x2 in 4 bit mode
// LCD.C file diver
// As defined in the following structure the pin
connection is as follows:
// RB3 enable
// RB2 rs
// RB4 DB4
// RB5 DB5
// RB6 DB6
// RB7 DB7
//
// LCD pins B0-B3 are not used and RW is not used.

#define rs PIN_C5
//rw connect gnd
#define enabled PIN_C6
#define DB4 PIN_C1
#define DB5 PIN_C2
#define DB6 PIN_C3
#define DB7 PIN_C4
char const lcd_type=2;
char const LCD_INIT_STRING[4] = {0x20 |
(lcd_type << 2), 0xc, 1, 6};

=====
==

void make_out_data(char buffer_data)
{
    output_bit(DB4,bit_test(buffer_data,0));
    output_bit(DB5,bit_test(buffer_data,1));
    output_bit(DB6,bit_test(buffer_data,2));
    output_bit(DB7,bit_test(buffer_data,3));
}

=====
==

void lcd_send_nibble(char buffer_nibble)
{ make_out_data(buffer_nibble);
delay_us(10);
output_high(enabled);
delay_us(10);
output_low(enabled);
}

=====

void lcd_send_byte( char address, char n )
{
    output_low(rs); //rs= 0;
delay_ms(1);
output_bit(rs,address);//lcd.rs = address;
delay_us(20);
delay_us(20);
output_low(enabled);//lcd.enable = 0;
lcd_send_nibble(n >> 4);
lcd_send_nibble(n & 0xf);
}

=====

void lcd_init()
{
    char i;
//set_tris_lcd(LCD_WRITE);
output_low(rs); //lcd.rs = 0;
//output_low(rw); //lcd.rw = 0;
output_low(enabled); //lcd.enable = 0;
delay_ms(200);
for(i=1;i<=3;++i)
{
    lcd_send_nibble(3);
delay_ms(10);
}
lcd_send_nibble(2);
for(i=0;i<=3;++i)
lcd_send_byte(0,LCD_INIT_STRING[i]);
}

=====

=====

void lcd_gotoxy( char x, char y)
{char address;
switch(y) {
case 1 : address=0x80;break;
case 2 : address=0xc0;break;
case 3 : address=0x94;break;
}
}

```

```

case 4 : address=0xd4;break;
}
address=address+(x-1);
lcd_send_byte(0,0x80|address);
}

=====

=====

void lcd_putc( char c ) {
switch (c) {
case '\f' : lcd_send_byte(0,1);
delay_ms(2); break;
case '\n' : lcd_gotoxy(1,2); break;
case '\b' : lcd_send_byte(0,0x10); break;
default : lcd_send_byte(1,c); break;
}
}

=====

=====

void LCD_Command(int cm);
void LCD_ShiftLeft(void);
void LCD_ShiftRight(void);
void LCD_MoveRight(char p){
char i;
for(i=0;i<p;i++){
LCD_ShiftRight();
delay_ms(200);
}
}

void LCD_MoveLeft(char p){
char i;
for(i=0;i<p;i++){
LCD_ShiftLeft();
delay_ms(200);
}
}

void LCD_String(char*s,int dly);

```

## HARDWARE PIC16F877A

Indirect add. Bank0	0h	Indirect add. Bank1	80h	Indirect add. Bank2	100h	Indirect add. Bank3	180h
TMR0	1h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	2h	PCL	82h	PCL	102h	PCL	182h
STATUS	3h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	4h	FSR	84h	FSR	104h	FSR	184h
PORTA	5h	RTISA	85h	105h			185h
PORTB	6h	RTISB	86h	PORTB	106h	RTISB	186h
PORTC	7h	RTISC	87h	107h			187h
PORTD	8h	RTISD	88h	108h			188h

PORTE	9h	RTISE	89h		109h		189h
PCLATH	Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	Eh	PCON	8Eh	EEDATH	10Eh	Reserved	18Eh
TMR1H	Fh		8Fh	EEADRH	10Fh	Reserved	18Fh
T1CON	10h		90h	General purpose register 16byte	110h	General purpose register 16byte	190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h		95h		115h		195h
CCPR1H	16h		96h		116h		196h
CCP1CON	17h		97h		117h		197h

RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah		9Ah		11Ah		19Ah
CCPR2L	1Bh		9Bh		11Bh		19Bh
CCPR2H	1Ch		9Ch		11Ch		19Ch
CCPR2CON	1Dh		9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON0	9Fh		11Fh		19Fh
General purpose register 96byte	20h	General purpose register 80byte	A0h	General purpose register 80byte	120h	General purpose register 80byte	1A0h
	7Fh	Accesses 70h – 7Fh	FFh	Accesses 70h – 7Fh	1Fh	Accesses 70h – 7Fh	1FFh

## TÀI LIỆU THAM KHẢO

1. DATASHEET PIC16F877A
2. CCS-C-cho-PIC16F877A-2008 – Thang 8831.
3. Diễn đàn Picvietnam.net
4. Diễn đàn Codientu.org.
5. Diễn đàn Dientuvietnam.net.
6. Website Banlinhkien.vn
7. Bai giang C++ -Học Viện Bưu Chính Viễn Thông.
8. Tài liệu sử dụng CCS tiếng việt của tác giả Trần Xuân Trường, ĐH BK HCM.
9. Teach\_Yourself\_PIC\_Microcontroller\_Programming- Dr. Amer Iqbal.
10. Embedded C Programming & Microchip Pic – Richard Barnett, Larry O'Cull, Sarah Cox.
11. <http://www.ccsinfo.com/>
12. <http://www.mikroe.com/>
13. <http://www.labcenter.com/>
14. <http://wikipedia.org>