**Chapter 3: PIC16F887 Microcontroller**

*The PIC16F887 is a well known product by Microchip. It features all the components which modern microcontrollers normally have. For its low price, wide range of application, high qual-ity and easy availability, it is an ideal solution in applications such as the control of different processes in industry, machine control devices, measurement of different values etc. Some of its main features are listed below.*

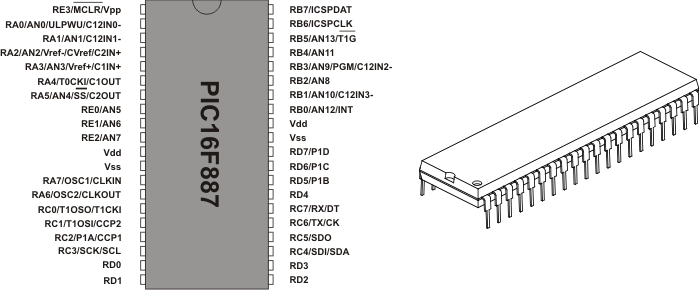
* [3.1 THE PIC16F887 BASIC FEATURES](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v1)
* [3.2 CORE SFRS](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v2)
* [3.3 INPUT/OUTPUT PORTS](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v3)
* [3.4 TIMER TMR0](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v4)
* [3.5 TIMER TMR1](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v5)
* [3.6 TIMER TMR2](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v6)
* [3.7 CCP MODULES](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v7)
* [3.8 SERIAL COMMUNICATION MODULES](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v8)
* [3.9 ANALOG MODULES](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v9)
* [3.10 CLOCK OSCILLATOR](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v10)
* [3.11 EEPROM MEMORY](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v11)
* [3.12 RESET! BLACK-OUT, BROWN-OUT OR NOISES?](http://www.mikroe.com/eng/chapters/view/16/chapter-3-pic16f887-microcontroller/#c3v12)

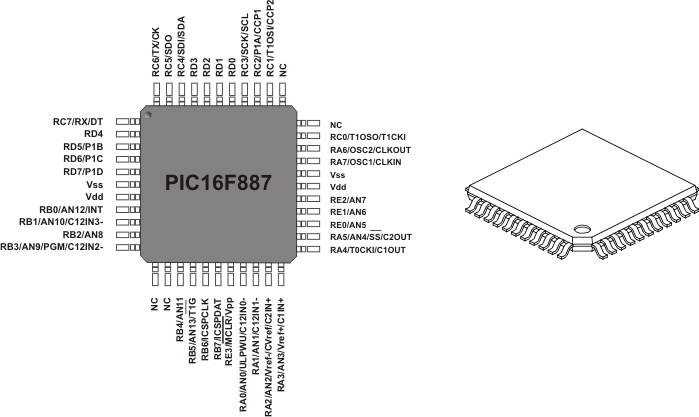
**3.1 THE PIC16F887 BASIC FEATURES**

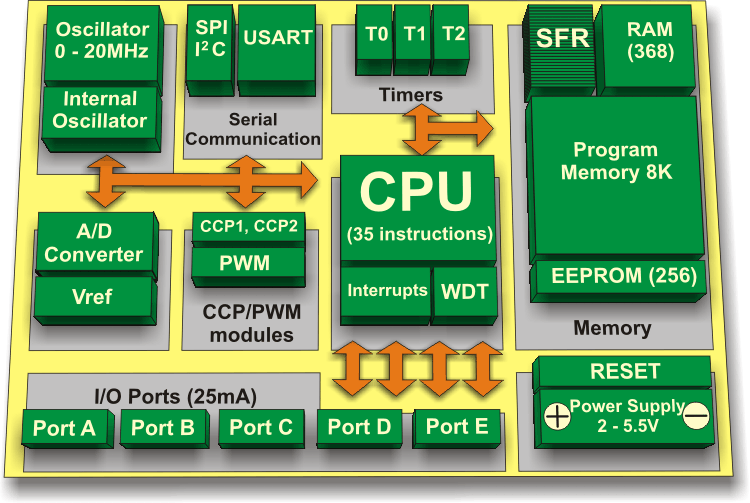
* **RISC architecture**
  + Only 35 instructions to learn
  + All single-cycle instructions except branches
* **Operating frequency 0-20 MHz**
* **Precision internal oscillator**
  + Factory calibrated
  + Software selectable frequency range of 8MHz to 31KHz
* **Power supply voltage 2.0-5.5V**
  + Consumption: 220uA (2.0V, 4MHz), 11uA (2.0 V, 32 KHz) 50nA (stand-by mode)
* **Power-Saving Sleep Mode**
* **Brown-out Reset (BOR) with software control option**
* **35 input/output pins**
  + High current source/sink for direct LED drive
  + software and individually programmable *pull-up* resistor
  + Interrupt-on-Change pin
* **8K ROM memory in FLASH technology**
  + Chip can be reprogrammed up to 100.000 times
* ***In-Circuit Serial Programming* Option**
  + Chip can be programmed even embedded in the target device
* **256 bytes EEPROM memory**

1. Data can be written more than 1.000.000 times
2. Bộ nhớ cho phép ghi xóa 1 triệu lần.
3. 368 bytes ram
4. Chuyển đổi A/D : 14 kênh độ phân giải 10bit
5. timer/counter độc lập.
6. **Watch-dog timer**

* **Analogue comparator module with**
  + Two analogue comparators
  + Fixed voltage reference (0.6V) – cố định điện áp tham chiếu
  + Programmable on-chip voltage reference – lập trình điện áp tham chiếu trên chip
* **PWM output steering control**
* **Enhanced USART module** – module truyền nối tiếp
  + Hỗ trợ giao tiếp RS-485, RS-232 and LIN2.0
  + Auto-Baud Detect
* **Master Synchronous Serial Port (MSSP)**
  + Hỗ trợ giao tiếp chuẩn SPI and I2C







**PINOUT DESCRIPTION**

Most pins of the PIC16F887 microcontroller are multi-functional as seen in figure above. For example, designator RA3/AN3/Vref+/C1IN+ for the fifth pin of the microcontroller indicates that it has the following functions:

Hầu hết các chân của vi điều khiển PIC16F887 là đa chức năng như trong hình trên. Ví dụ, vấn thiết kế RA3/AN3/Vref + / C1IN + cho pin thứ năm của vi điều khiển chỉ ra rằng nó có chức năng sau:

Nghe

Đọc ngữ âm

* + - * R/A là chân xuất nhập số của port A
      * AN3 là ngõ vào tương tự của kênh thứ 3
      * Chân dương của điện áp tham chiếu…
* RA3 Port A third digital input/output
* AN3 Third analog input
* Vref+ Positive voltage reference
* C1IN+ Comparator C1 positive input

• RA3 Port A thứ ba kỹ thuật số đầu vào / đầu ra  
• AN3 analog đầu vào thứ ba  
• Vref + Tích cực điện áp tham chiếu  
• C1IN + C1 so sánh tích cực đầu vào

* Nghe
* Đọc ngữ âm

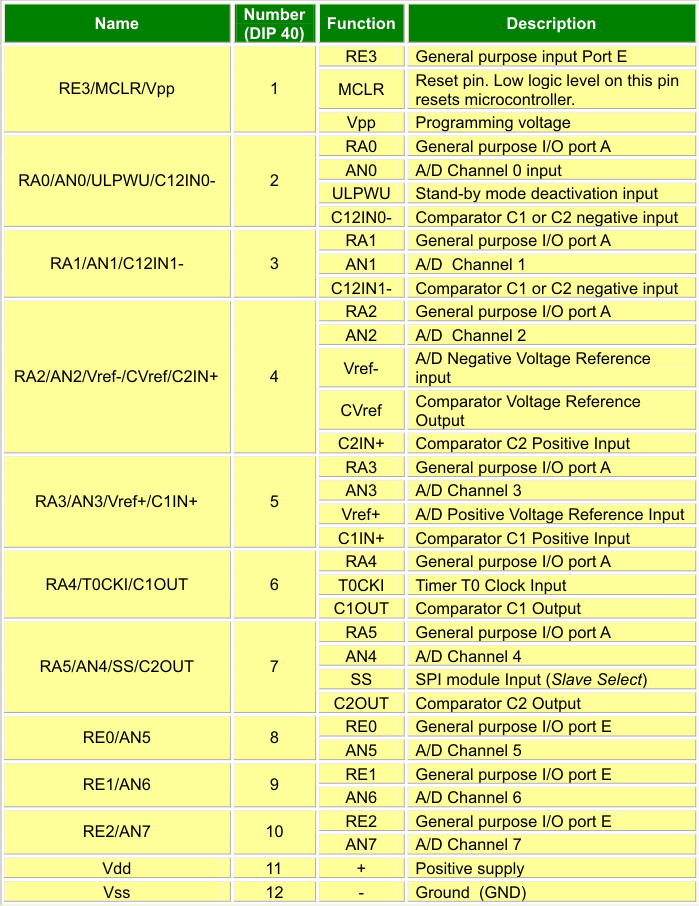
Such pin functionality is very useful as it makes the microcontroller package more compact without affecting its operation. These various pin functions cannot be used simultaneously, but can be changed at any point during operation.

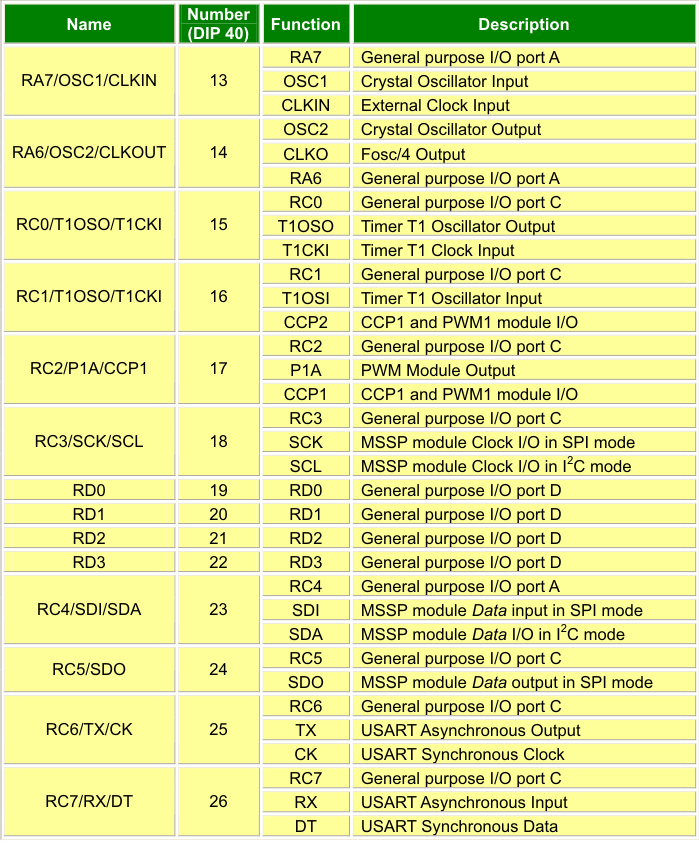
The following tables refer to the PDIP 40 microcontroller.

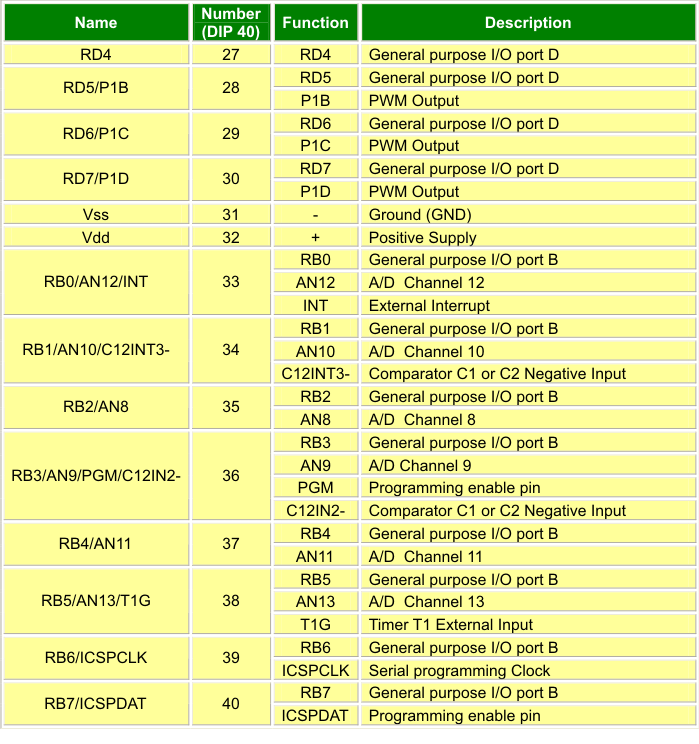
chức năng pin như vậy là rất hữu ích vì nó làm cho các việc đóng gói vi điều khiển nhỏ gọn hơn mà không ảnh hưởng đến hoạt động của mình. Các chức năng này pin khác nhau có thể không được sử dụng đồng thời, nhưng có thể thay tại mỗi thời điểm khác nhau trong quá trình hoạt động.  
Các bảng sau đây tham khảo các PDIP 40 vi điều khiển.

Nghe

Đọc ngữ âm







**CENTRAL PROCESSOR UNIT (CPU)**

**ĐƠN VỊ XỬ LÝ TRUNG TÂM:**

We are not going to bore you with the operation of the CPU at this stage. However, we will just state that the CPU is manufactured with RISC technology as it is an important factor when deciding which microcontroller to use.

RISC stands for *Reduced Instruction Set Computer*, which gives the PIC16F877 two great advantages:

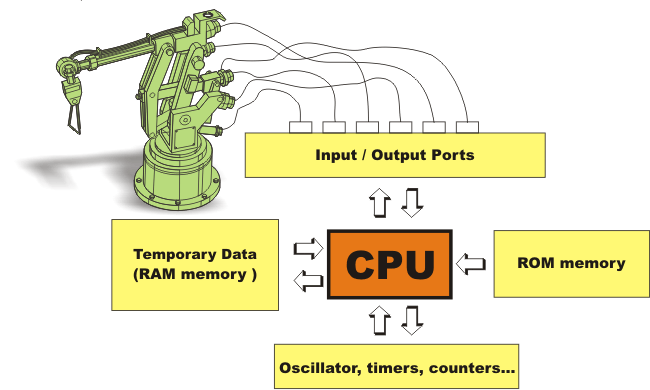
Chúng ta sẽ hãy khoan bàn về những hoạt động của CPU ở giai đoạn này. Tuy nhiên, chúng tôi sẽ chỉ rõ rằng các CPU được sản xuất với công nghệ chip RISC - một yếu tố quan trọng khi quyết định tại sao vi điều khiển để sử dụng.  
  
RISC là viết tắt của Reduced Instruction Set Computer, điều này làm cho PIC16F877 có 2 thuận lợi to lớn:

* The CPU only recognizes 35 simple instructions. Just to mention that in order to program other microcontrollers in assembly language it is necessary to know more than 200 instructions by heart.
* The execution time is the same for almost all instructions, and lasts for 4 clock cycles. The oscillator frequency is stabilized by a quartz crystal. The execution time of jump and branch instructions is 2 clock cycles. It means that if the microcontroller’s operating speed is 20MHz, the execution time of each instruction will be 200nS, i.e. the program will execute 5 million instructions per second!

• Thời gian thực hiện là như nhau cho hầu như tất cả các lệnh , mỗi lệnh kéo dài trong 4 chu kỳ xung nhịp. tần số dao động được ổn định bởi một tinh thể thạch anh.

Thời gian thực hiện lệnh nhảy và rẽ nhánh là 2 chu kỳ đồng hồ.

Điều này nghĩa là nếu tốc độ hoạt động của vi điều khiển là 20MHz, thời gian thực hiện mỗi lệnh sẽ là 200nS, tức là chương trình sẽ thực hiện 5.000.000 lênh trong một giây! (nhanh hơn so với 89C51)



**MEMORY**

**BỘ NHỚ:**

The PIC16F887 has three types of memory ROM, RAM and EEPROM. All of them will be separately discussed since each has specific functions, features and organization.

PIC16F887 có ba loại bộ nhớ ROM, RAM và EEPROM. Chúng được chia ra và có chức năng đặc biệt, cấu tạo kac nhau.

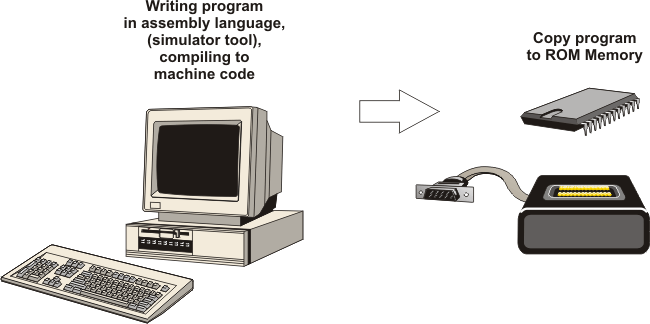
**ROM MEMORY**

ROM memory is used to permanently save the program being executed. This is why it is often called ‘program memory’. The PIC16F887 has 8Kb of ROM (in total of 8192 locations). Since the ROM memory is made with FLASH technology, its contents can be changed by providing a special programming voltage (13V).

bộ nhớ ROM được sử dụng để lưu các chương trình đang được thực thi 1 cách vĩnh viễn. Đây là lý do tại sao nó thường được gọi là 'chương trình bộ nhớ”. PIC16F887 có 8Kb bộ nhớ của ROM (trong tổng số 8.192 địa chỉ nhớ). từ khi bộ nhớ ROM được chế tạo với công nghệ FLASH, nội dung chứa trong nó có thể được thay đổi bằng cách cung cấp một điện áp lập trình đặc biệt (13V).

However, it is not necessary to explain it in detail as being automatically performed by means of a special program on the PC and a simple electronic device called the programmer.

Tuy nhiên, cũng không cần thiết phải giải thích 1 cách chi tiết rằng Nội dung trong ROM được tự động đổ vào do một chương trình đặc biệt trên máy tính và một thiết bị điện tử đơn giản gọi bộ lập trình



**EEPROM MEMORY**

Similar to program memory, the contents of EEPROM is permanently saved, even when the power goes off. However, unlike ROM, the contents of EEPROM can be changed during the operation of the microcontroller. This is why this memory (256 locations) is perfect for permanently saving some of the results created and used during the operation.

Bộ nhớ EEPROM  
Tương tự như bộ nhớ chương trình, nội dung chứa trong EEPROM là vĩnh viễn, ngay cả khi bị cúp điện. Tuy nhiên, không giống như ROM, các nội dung của EEPROM có thể được thay đổi trong khi vi điều khiển hoạt động.Đây là lý do tại sao bộ nhớ này (256 địa chỉ nhớ) là hoàn hảo cho việc lưu lại một số kết quả tạo ra và sử dụng trong suốt quá trình hoạt động.

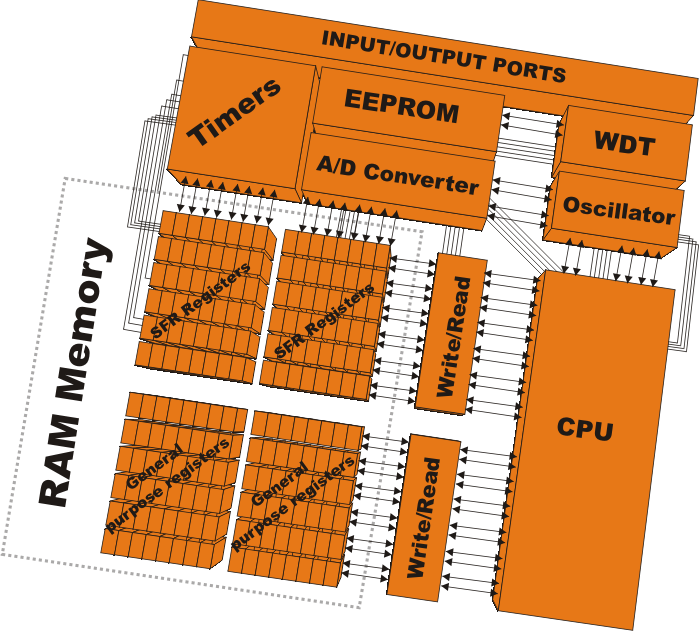
**RAM MEMORY**

This is the third and the most complex part of microcontroller memory. In this case, it consists of two parts: general-purpose registers and special-function registers (SFR). All these registers are divided in four memory banks to be explained later in the chapter.

Bộ nhớ RAM  
Đây là phần thứ ba và phần phức tạp nhất của bộ nhớ vi điều khiển. RAM được chia làm 2 phần: ram đa dụng ( general ) cho nhiều mục đích sử dụng và đặc và các thanh ghi đặc biệt (SFR). Tất cả các thanh ghi này nằm trong 4 bank thanh ghi và được đề cập trong chương kế tiếp.

Even though both groups of registers are cleared when power goes off and even though they are manufactured in the same manner and act in a similar way, their functions do not have many things in common.

Mặc dù nội dung của thanh ghi đa dụng và thanh ghi đặc biệt bị xóa khi tắt nguồn, Thậm chí ram đa dụng và ram đặc biệt cùng được chế tạo như nhau nhưng chức năng của chúng không giống nhau hoàn toàn.



**GENERAL-PURPOSE REGISTERS**

General-purpose registers are used for storing temporary data and results created during operation. For example, if the program performs counting (products on the assembly line), it is necessary to have a register which stands for what we in everyday life call ‘sum’. Since the microcontroller is not creative at all, it is necessary to specify the address of some general purpose register and assign it that function. A simple program to increment the value of this register by 1, after each product passes through a sensor, should be created.

Thanh ghi đa dụng:

Thanh ghi đa dụng được sử dụng để lưu trữ dữ liệu tạm thời và kết quả tạo ra trong quá trình hoạt động của vi điều khiển. Ví dụ, nếu chương trình thực hiện đếm (sản phẩm trên dây chuyền lắp ráp), cần phải có một thanh ghi làm chức năng ghi nhận kết quả của tổng số sản phẩm.

Now the microcontroller can execute the program as it knows what and where the sum to be incremented is. Similarly, each program variable must be preassigned some of the general- purpose registers.

**SPECIAL FUNCTION REGISTERS (SFRS)**

Special-function registers are also RAM memory locations, but unlike general-purpose registers, their purpose is predetermined during manufacturing process and cannot be changed. Since their bits are connected to particular circuits on the chip (A/D converter, serial communication module, etc.), any change of their contents directly affects the operation of the microcontroller or some of its circuits. For example, the ADCON0 register controls the operation of A/D converter. By changing its bits it is determined which port pin is to be configured as converter input, the moment conversion is to start as well as the speed of conversion.

Thanh ghi đặc biệt cũng là vùng nhớ RAM, nhưng nó không giống với thanh ghi đa dụng, mục đích của thanh ghi đặc biệt được xác định trước trong quá trình xử lý và không thể thay đổi. Kể từ khi bit của họ được kết nối với các mạch cụ thể trên chip (chuyển đổi A / D, module giao tiếp nối tiếp, vv), bất kỳ thay đổi nội dung của thanh ghi đặc biệt đều ảnh hưởng trực tiếp ảnh hưởng đến hoạt động của vi điều khiển hay một số ngoại vi của nó. Ví dụ, thanh ghi ADCON0 điều khiển hoạt động của bộ chuyển đổi Analog sang Degital. Bằng cách thay đổi các bit của thanh ghi đặc biệt, ta sẽ xác định cổng nào , chân nào là để được đặt là đầu vào chuyển đổi, thời điểm chuyển đổi cũng như tốc độ của chuyển đổi.

Nghe

Đọc ngữ âm

Another feature of these memory locations is that they have their names (both registers and their bits), which considerably simplifies the process of writing a program. Since high-level programming languages can use the list of all registers with their exact addresses, it is enough to specify the name of a register in order to read or change its contents.

Một tính năng của các vùng nhớ này chúng được định tên (tên cho từng byte và tên cho từng bit), điều này giúp đơn giản hoá 1 cách rất hiệu quả quá trình viết một chương trình. ngôn ngữ lập trình cấp cao có thể sử dụng danh sách các thanh ghi đặc biệt với địa chỉ chính xác - Hay còn gọi là ngôn ngữ gợi nhớ, từ đó có thể đọc hay ghi một nội dung khác vào các byte hoặc từng bit.

**RAM MEMORY BANKS**

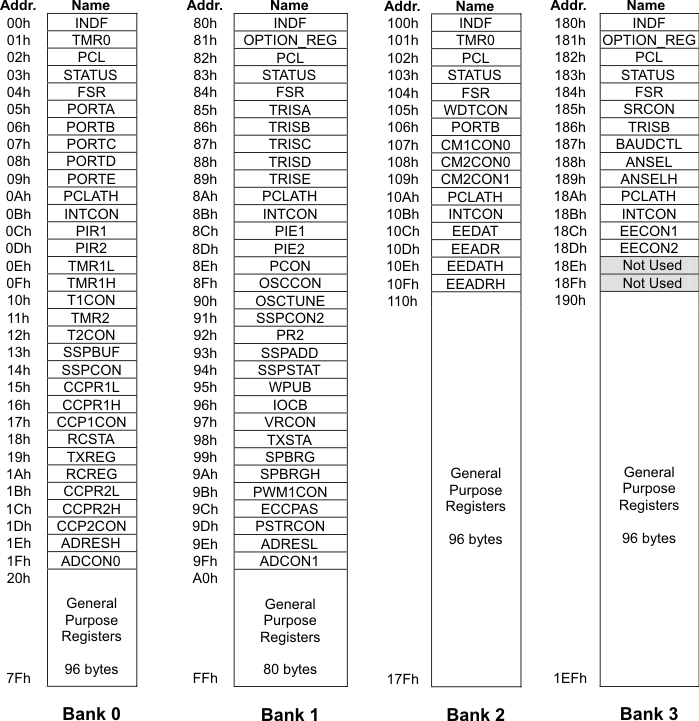
**Các bank thanh ghi:**

The RAM memory is partitioned into four banks. Prior to accessing any register during program writing (in order to read or change its contents), it is necessary to select the bank which contains that register. Two bits of the STATUS register are used for bank selection to be discussed later. In order to simplify the operation, the most commonly used SFRs have the same address in all banks, which enables them to be easily accessed.

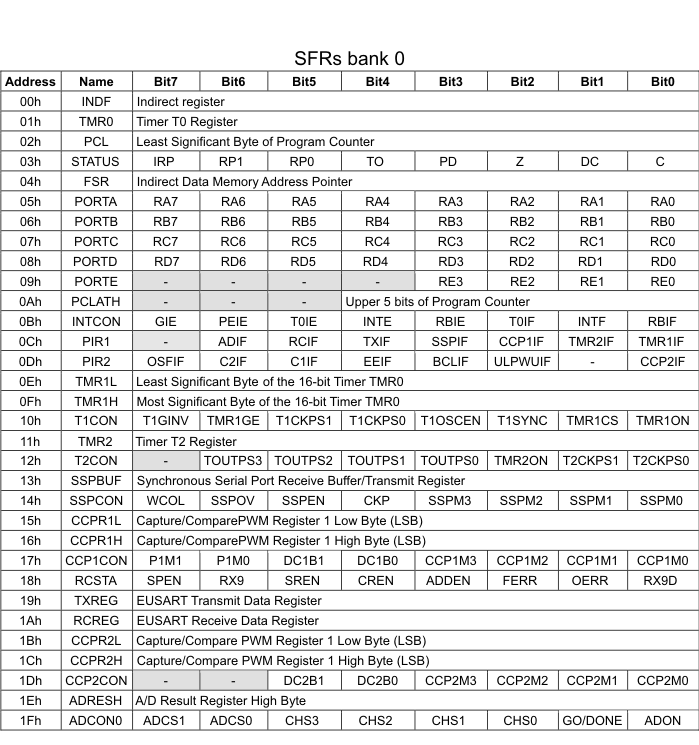
Bộ nhớ RAM được chia làm 4 bank thanh ghi.. Trước khi truy cập vào bất kỳ đăng ký trong chương trình bằng văn bản (theo thứ tự để đọc hoặc thay đổi nội dung của nó), nó là cần thiết để lựa chọn bank nào trong chúng , trong đó có thể đăng ký. Hai bit của STATUS đăng ký được sử dụng để lựa chọn ngân hàng sẽ được thảo luận sau đó. Để đơn giản hóa hoạt động, SFRs thông dụng nhất có địa chỉ tương tự trong tất cả các ngân hàng, cho phép họ được dễ dàng truy cập.

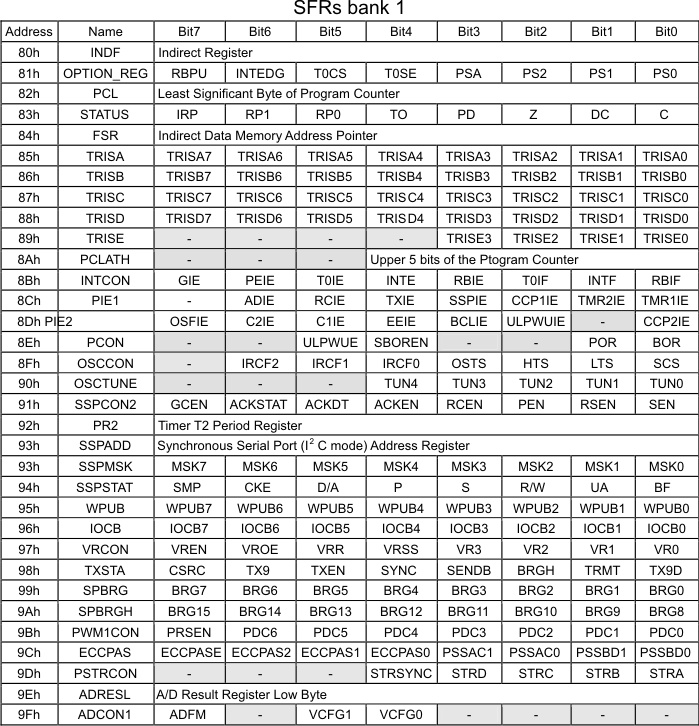
Nghe

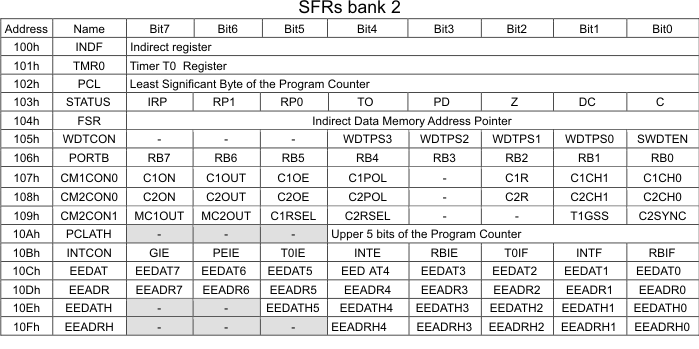
Đọc ngữ âm

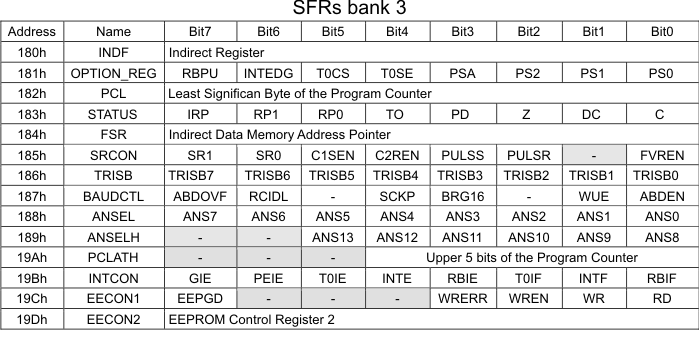


Handling banks may be difficult only if you write a program in assembly language. When using higher programming languages such as C and compilers such as **mikroC PRO for PIC**, all you have to do is to specify the register name. On the basis of that, the compiler selects necessary bank and appropriate instructions used for bank selection will be built in the code during the process of compilation. You have been using only assembly language so far and this is the first time you use the C compiler, haven’t you? Isn’t this a wonderful news?









**STACK**

A part of RAM used as stack consists of eight 13-bit registers. Before the microcontroller starts to execute a subroutine (**CALL** instruction) or when an interrupt occurs, the address of the first next instruction to execute is pushed onto the stack, i.e. one of its registers. Thanks to that the microcontroller knows from where to continue regular program execution upon a subroutine or an interrupt execution. This address is cleared after returning to the program because there is no need to save it any longer, and one location of the stack becomes automatically available for further use.

It is important to bear in mind that data is always circularly pushed onto the stack. It means that after the stack has been pushed eight times, the ninth push overwrites the value that was stored with the first push. The tenth push overwrites the second push and so on. Data overwritten in this way is not recoverable. In addition, the programmer cannot access these registers for write or read and there is no Status bit to indicate stack overflow or stack underflow conditions. For this reason, it is necessary to take special care of it during program writing.

**Let's do it in mikroC...**

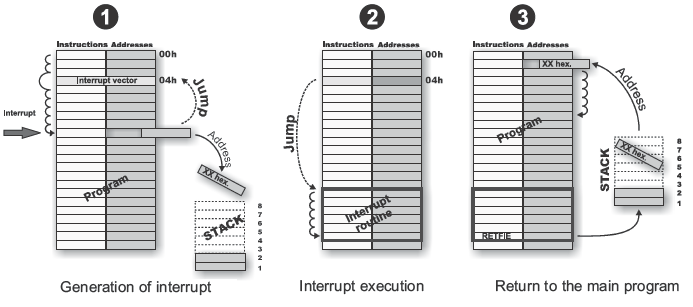
*/\* When entering or exiting an assembly instruction in the program, the compiler  
doesn’t save data on the currently active RAM bank. It means that in this program  
section, bank selection depends on the SFR registers in use. When switching back  
to the program section written in C, the control bits RP0 and RP1 must return the  
state they had before assembly language code execution. In this example, the problem  
is solved by using the saveBank auxiliary variable which saves the state of  
these two bits. \*/*  
  
 saveBank = STATUS & 0b01100000; *// Save the state of bits RP0 and RP1  
 // (bits 5 and 6 of the STATUS register)*  
 **asm** { *// Start of assembly sequence*  
 ...  
 ... *// Assembly code*  
 ...  
 } *// End of assembly sequence*  
 STATUS &= 0b10011111; *// Bits RP0 and RP1 return their original state*  
 STATUS |= saveBank;  
 ...  
 ...

**INTERRUPT SYSTEM**

The first thing the microcontroller does when an interrupt request arrives is to execute the current instruction and then stops the regular program execution. As a result, the current program memory address is automatically pushed onto the stack and the default address (predefined by the manufacturer) is written to the program counter. The location from where the program proceeds with execution is called an interrupt vector. For the PIC16F887 microcontroller, this address is 0004h. As seen in figure below, the location containing the interrupt vector is passed over during regular program execution.

A part of the program to be executed when an interrupt request arrives is called an interrupt routine. Its first instruction is located at the interrupt vector. How long will it take to execute this subroutine and what it will be like depends on the skills of the programmer as well as on the interrupt source itself. Some of the microcontrollers have more interrupt vectors (every interrupt request has its vector), but in this case there is only one. Consequently, the first part of the interrupt routine consists in interrupt source detection.

Finally, when the interrupt source is recognized and the interrupt routine is executed, the microcontroller reaches the **RETFIE** instruction, pops the address from the stack and proceeds with program execution from where it left off.



mikroC recognizes an interrupt routine to be executed as the **void interrupt()** function. The body of that function, i.e. interrupt routine, should be written by the user.

**void** interrupt() { *// Interrupt routine*  
 cnt++ ; *// Interrupt causes variable cnt to be incremented by 1*  
}

**In Short: How to Use SFRs**

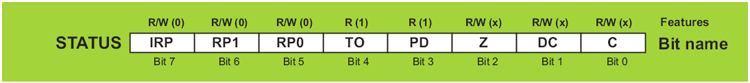
You have bought the microcontroller and have a good idea how to use it... There is a long list of SFRs and their bits. Each of them controls some process. All in all, it looks like a big control table with a lot of instruments and switches. Now you are concerned about whether you will manage to learn how to use them all? You will probably not, but don’t worry, you don’t have to! Such powerful microcontrollers are similar to supermarkets: they offer so many things at low prices and it is up to you to choose those you need. Therefore, select the field you are interested in and study only what you need to know. When you completely understand hardware operation, study SFRs which are in control of it (there are usually a few of them).

*As all devices have a sort of control system, the microcontroller has its ‘levers’ which you have to be familiar with in order to be able to use it properly. Of course, we are talking about SFRs from which the process of programming begins and where it ends.*

**3.2 CORE SFRS**

The following text describes the core SFRs of the PIC16F887 microcontroller. Bits of each of these registers control different circuits within the chip, so that it is not possible to classify them in some special groups. For this reason, they are described along with the processes they are in control of.

**STATUS Register**



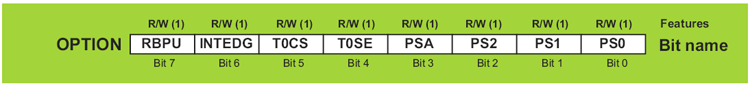
The STATUS register contains: the arithmetic status of data in the W register, the RESET status and the bank select bits for data memory.

* **IRP** - Bit selects register bank. It is used for indirect addressing.
  + **1** - Banks 0 and 1 are active (memory locations 00h-FFh)
  + **0** - Banks 2 and 3 are active (memory locations 100h-1FFh)
* **RP1,RP0** - Bits select register bank. They are used for direct addressing.

|  |  |  |
| --- | --- | --- |
| **RP1** | **RP0** | **Active Bank** |
| 0 | 0 | Bank0 |
| 0 | 1 | Bank1 |
| 1 | 0 | Bank2 |
| 1 | 1 | Bank3 |

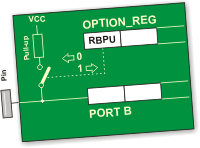
* **TO - Time-out bit.**
  + **1** - After power-on, after executing the **CLRWDT** instruction which resets the watch-dog timer or the SLEEP instruction which sets the microcontroller into low-consumption mode.
  + **0** - After watch-dog timer time-out has occurred.
* **PD - Power-down bit.**
  + **1** - After power-on or after executing the **CLRWDT** instruction which resets the watchdog timer.
  + **0** - After executing the **SLEEP** instruction which sets the microcontroller into low-consumption mode.
* **Z - Zero bit**
  + **1** - The result of an arithmetic or logic operation is zero.
  + **0** - The result of an arithmetic or logic operation is different from zero.
* **DC - Digit carry/borrow bit** is changed during addition and subtraction if an ‘overflow’ or a ‘borrow’ of the result occurs.
  + **1** - A carry-out from the 4th low-order bit of the result has occurred.
  + **0** - No carry-out from the 4th low-order bit of the result has occurred.
* **C - Carry/Borrow bit** is changed during addition and subtraction if an ‘overflow’ or a ‘borrow’ of the result occurs, i.e. if the result is greater than 255 or less than 0.
  + **1** - A carry-out from the most significant bit (MSB) of the result has occurred.
  + **0** - No carry-out from the most significant bit (MSB) of the result has occurred.

**OPTION\_REG Register**

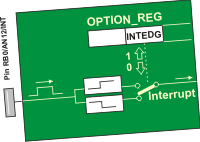


**Legend: R/W - Readable/Writable Bit, (1) After reset, bit is set**

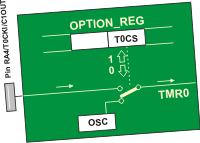
The OPTION\_REG register contains various control bits to configure Timer0/WDT prescaler, timer TMR0, external interrupt and pull-ups on PORTB.



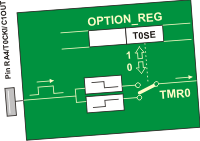
* **RBPU - Port B Pull up Enable bit.**
  + **1** - PortB pull-ups are disabled.
  + **0** - PortB pull-ups are enabled.



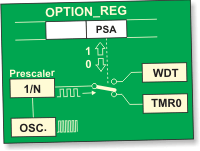
* **INTEDG - Interrupt Edge Select bit.**
  + **1** - Interrupt on rising edge of RB0/INT pin.
  + **0** - Interrupt on falling edge of RB0/INT pin.



* **T0CS - TMR0 Clock Source Select bit.**
  + **1** - Transition on TOCKI pin.
  + **0** - Internal instruction cycle clock (Fosc/4).



* **T0SE - TMR0 Source Edge Select bit** selects pulse edge (rising or falling) counted by the timer TMR0 through the RA4/T0CKI pin.
  + **1** - Increment on high-to-low transition on TOCKI pin.
  + **0** - Increment on low-to-high transition on TOCKI pin.



* **PSA - Prescaler Assignment bit** assigns prescaler (only one exists) to the timer or watchdog timer.
  + **1** - Prescaler is assigned to the WDT.
  + **0** - Prescaler is assigned to the TMR0.

**PS2, PS1, PS0 Prescaler Rate Select bits**

Prescaler rate is selected by combining these three bits. As shown in the table below, prescaler rate depends on whether prescaler is assigned to the timer (TMR0) or watch-dog timer (WDT).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PS2** | **PS1** | **PS0** | **TMR0** | **WDT** |
| 0 | 0 | 0 | 1:2 | 1:1 |
| 0 | 0 | 1 | 1:4 | 1:2 |
| 0 | 1 | 0 | 1:8 | 1:4 |
| 0 | 1 | 1 | 1:16 | 1:8 |
| 1 | 0 | 1 | 1:64 | 1:32 |
| 1 | 1 | 0 | 1:128 | 1:64 |
| 1 | 1 | 1 | 1:256 | 1:128 |

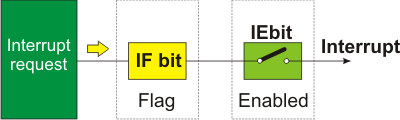
In order to achieve 1:1 prescaler rate when the timer TMR0 counts up pulses, the prescaler should be assigned to the WDT. As a result, the timer TMR0 does not use the prescaler, but directly counts pulses generated by the oscillator, which was the objective.

**Let's do it in mikroC...**

*/\* If the CLRWDT command is not executed, WDT will reset the microcontroller   
 every 32.768 uS (f = 4 MHz) \*/*  
  
**void** main() {  
 OPTION\_REG = 0b00001111; *// Prescaler is assigned to WDT (1:128)*  
   
 **asm** CLRWDT; *// Assembly command to reset WDT*  
 ...  
 ... *// Time between these two CLRWDT commands must not   
 ... // exceed 32.768 microseconds (128x256)*  
   
 **asm** CLRWDT; *// Assembly command to reset WDT*  
 ...   
 ... *// Time between these two CLRWDT commands must not   
 ... // exceed 32.768 microseconds (128x256)*  
   
 **asm** CLRWDT; *// Assembly command to reset WDT*  
 ...

**INTERRUPT SYSTEM REGISTERS**

When an interrupt request arrives, it doesn’t mean that an interrupt will automatically occur, because it must also be enabled by the user (from within the program). Because of this, there are special bits used to enable or disable interrupts. It is easy to recognize them by the letters IE contained in their names (stands for *Interrupt Enable*). Besides, each interrupt is associated with another bit called the flag which indicates that an interrupt request has arrived regardless of whether it is enabled or not. They are also easily recognizable by the last two letters contained in their names- IF (*Interrupt Flag*).



As seen, everything is based on a simple and efficient idea. When an interrupt request arrives, the flag bit is set first.

If the appropriate IE bit is not set (0), this condition will be completely ignored. Otherwise, an interrupt occurs! If several interrupt sources are enabled, it is necessary to detect the active one before the interrupt routine starts execution. Source detection is performed by checking flag bits.

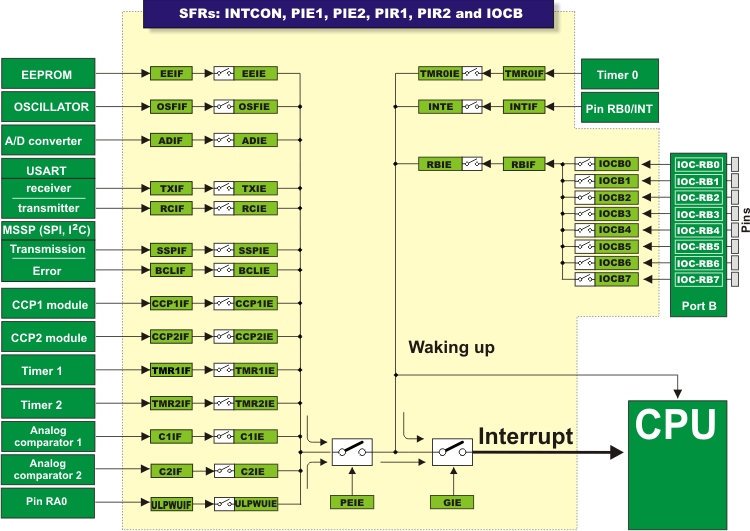
It is important to know that the flag bits are not automatically cleared, but by software while the interrupt routine execution is in progress. If we neglect this detail, another interrupt will occur immediately after returning to the main program, even though there are no more requests for its execution. Simply put, the flag, as well as the IE bit, remain set.

All interrupt sources typical of the PIC16F887 microcontroller are shown on the next page. Note several things:

The **GIE** bit enables all unmasked interrupts and disables all interrupts simultaneously.

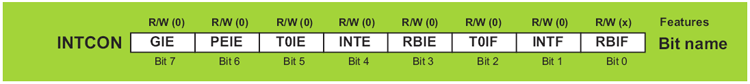
The **PEIE bit** enables all unmasked peripheral interrupts and disables all peripheral interrupts. This doesn’t concern Timer TMR0 and PORTB interrupt sources.

To enable an interrupt caused by changing logic state on PORTB, it is necessary to enable it for each bit separately. In this case, bits of the **IOCB** register act as control IE bits.



**INTCON Register**

The INTCON register contains various enable and flag bits for TMR0 register overflow, PORTB change and external INT pin interrupts.



**Legend: R/W - Readable/Writable Bit, (0) After reset, bit is cleared, (X) After reset, bit is unknown**

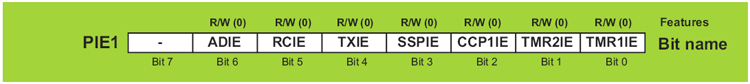
* **GIE - Global Interrupt Enable bit** - controls all possible interrupt sources simultaneously.
  + **1** - Enables all unmasked interrupts.
  + **0** - Disables all interrupts.
* **PEIE - Peripheral Interrupt Enable bit** acts similar to the GIE it, but controls interrupts enabled by peripherals. It means that it has no impact on interrupts triggered by the timer TMR0 or by changing the state of PORTB or the RB0/INT pin.
  + **1** - Enables all unmasked peripheral interrupts.
  + **0** - Disables all peripheral interrupts.
* **T0IE - TMR0 Overflow Interrupt Enable bit** controls interrupt enabled by TMR0 overflow.
  + **1** - Enables the TMR0 interrupt.
  + **0** - Disables the TMR0 interrupt.
* **INTE - RB0/INT External Interrupt Enable bit** controls interrupt caused by changing the logic state of the RB0/INT input pin (external interrupt).
  + **1** - Enables the INT external interrupt.
  + **0** - Disables the INT external interrupt.
* **RBIE - RB Port Change Interrupt Enable bit**. When configured as inputs, PORTB pins may cause an interrupt by changing their logic state (no matter whether it is high-to-low transition or vice versa, the fact that something is changed only matters). This bit determines whether an interrupt is to occur or not.
  + **1** - Enables the port B change interrupt.
  + **0** - Disables the port B change interrupt.
* **T0IF - TMR0 Overflow Interrupt Flag bit** registers the timer TMR0 register overflow, when counting starts at zero.
  + **1** - TMR0 register has overflowed (bit must be cleared from within the software).
  + **0** - TMR0 register has not overflowed.
* **INTF - RB0/INT External Interrupt Flag bit** registers the change of the RB0/INT pin logic state.
  + **1** - The INT external interrupt has occurred (must be cleared from within the software).
  + **0** - The INT external interrupt has not occurred.
* **RBIF - RB Port Change Interrupt Flag bit** registers any change of logic state of some PORTB input pins.
  + **1** - At least one of the PORTB general purpose I/O pins has changed state. Upon reading PORTB, the RBIF bit must be cleared from within the software.
  + **0** - None of the PORTB general purpose I/O pins has changed the state.

**Let's do it in mikroC...**

*// The PORTB.4 pin is configured as an input sensitive to logic state change*  
   
**void** initMain() {  
  
ANSEL = ANSELH = 0; *// All I/O pins are configured as digital*  
PORTB = 0; *// All PORTB pins are cleared*  
TRISB = 0b00010000; *// All PORTB pins except PORTB.4 are configured as outputs*  
RBIE = 1; *// Interrupts on PORTB change are enabled*  
IOCB4 = 1; *// Interrupt on PORTB pin4 change is enabled*  
GIE = 1; *// Global interrupt is enabled*  
...  
... *// From this point, any change of the PORTB.4 pin logic state   
 // will cause an interrupt*  
...

**PIE1 Register**

The PIE1 register contains peripheral interrupt enable bits.



**Legend: (-) Unimplemented bit, (R/W) - Readable/Writable Bit, (0) After reset, bit is cleared**

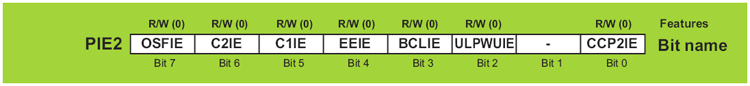
* **ADIE - A/D Converter Interrupt Enable bit.**
  + **1** - Enables the ADC interrupt.
  + **0** - Disables the ADC interrupt.
* **RCIE - EUSART Receive Interrupt Enable bit.**
  + **1** - Enables the EUSART receive interrupt.
  + **0** - Disables the EUSART receive interrupt.
* **TXIE - EUSART Transmit Interrupt Enable bit.**
  + **1** - Enables the EUSART transmit interrupt.
  + **0** - Disables the EUSART transmit interrupt.
* **SSPIE - Master Synchronous Serial Port (MSSP) Interrupt Enable bit** - enables an interrupt request to be generated upon each data transmission via synchronous serial communication module (SPI or I2C mode).
  + **1** - Enables the MSSP interrupt.
  + **0** - Disables the MSSP interrupt.
* **CCP1IE - CCP1 Interrupt Enable bit** enables an interrupt request to be generated in CCP1 module used for PWM signal processing.
  + **1** - Enables the CCP1 interrupt.
  + **0** - Disables the CCP1 interrupt.
* **TMR2IE - TMR2 to PR2 Match Interrupt Enable bit**
  + **1** - Enables the TMR2 to PR2 match interrupt.
  + **0** - Disables the TMR2 to PR2 match interrupt.
* **TMR1IE - TMR1 Overflow Interrupt Enable bit** enables an interrupt request to be generated upon each timer TMR1 register overflow, i.e. when the counting starts from zero.
  + **1** - Enables the TMR1 overflow interrupt.
  + **0** - Disables the TMR1 overflow interrupt.

**Let's do it in mikroC...**

*/\* Each overflow in the Timer1 register consisting of TMR1H and TMR1L, causes an interrupt  
to occur. In every interrupt rutine, variable cnt will be incremented by 1. \*/*  
  
**unsigned short** cnt; *// Define variable cnt*   
  
**void** interrupt() {  
 cnt++ ; *// Interrupt causes cnt to be incremented by 1*  
 PIR1.TMR1IF = 0; *// Reset bit TMR1IF*  
 TMR1H = 0x80; *// TMR1H and TMR1L timer registers are returned*  
 TMR1L = 0x00; *// their initial values*  
}  
  
**void** main() {  
 ANSEL = ANSELH = 0; *// All I/O pins are configured as digital*  
 T1CON = 1; *// Turn on timer TMR1*  
 PIR1.TMR1IF = 0; *// Reset the TMR1IF bit*  
 TMR1H = 0x80; *// Set initial value for timer TMR1*  
 TMR1L = 0x00;  
 PIE1.TMR1IE = 1; *// Enable an interrupt on overflow*  
 cnt = 0; *// Reset variable cnt*  
 INTCON = 0xC0; *// Enable interrupt (bits GIE and PEIE)*  
 ...

**PIE2 Register**

The PIE2 Register also contains various interrupt enable bits.



**Legend: (-) Unimplemented bit, (R/W) - Readable/Writable Bit, (0) After reset, bit is cleared**

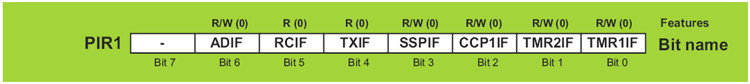
* **OSFIE - Oscillator Fail Interrupt Enable bit.**
  + **1** - Enables oscillator fail interrupt.
  + **0** - Disables oscillator fail interrupt.
* **C2IE - Comparator C2 Interrupt Enable bit.**
  + **1** - Enables Comparator C2 interrupt.
  + **0** - Disables Comparator C2 interrupt.
* **C1IE - Comparator C1 Interrupt Enable bit.**
  + **1** - Enables Comparator C1 interrupt.
  + **0** - Disables Comparator C1 interrupt.
* **EEIE - EEPROM Write Operation Interrupt Enable bit.**
  + **1** - Enables EEPROM write operation interrupt.
  + **0** - Disables EEPROM write operation interrupt.
* **BCLIE - Bus Collision Interrupt Enable bit.**
  + **1** - Enables bus collision interrupt.
  + **0** - Disables bus collision interrupt.
* **ULPWUIE - Ultra Low-Power Wake-up Interrupt Enable bit.**
  + **1** - Enables Ultra Low-Power Wake-up interrupt.
  + **0** - Disables Ultra Low-Power Wake-up interrupt.
* **CCP2IE - CCP2 Interrupt Enable bit.**
  + **1** - Enables CCP2 interrupt.
  + **0** - Disables CCP2 interrupt.

**Let's do it in mikroC...**

*/\* Comparator C2 is configured to use pins RA0 and RA2 as inputs.   
 Every change on the comparator's output will cause the PORTB.1 output pin   
 to change its logic state in interrupt routine. \*/*  
   
**void** interrupt() {  
 PORTB.F1 = ~PORTB.F1 ; *// Interrupt will invert logic state of the PORTB.1 pin*  
 PIR2.C2IF = 0; *// Interrupt flag bit C2IF is cleared*  
}  
  
**void** main() {  
 TRISB = 0; *// All PORTB pins are configured as outputs*  
 PORTB.1 = 1; *// The PORTB.1 pin is set*  
 ANSEL = 0b00000101; *// RA0/C12IN0- and RA2/C2IN+ pins are analog inputs*  
 ANSELH = 0; *// All other I/O pins are configured as digital*  
 C2CH0 = C2CH1 = 0; *// The RA0 pin is selected to be C2 inverting input*  
 C2IE = 1; *// Enables compatator C2 interrupt*  
 GIE = 1; *// Global interrupt is enabled*  
 C2ON = 1; *// Comparator C2 is enabled*  
 ...  
 ...

**PIR1 Register**

The PIR1 register contains the interrupt flag bits.

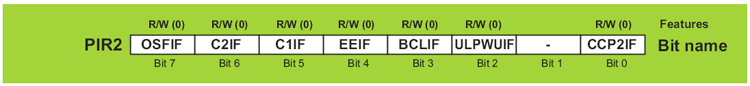


**Legend: (-) Unimplemented bit, (R/W) - Readable/Writable Bit, (R) - Readable Bit, (0) After reset, bit is cleared**

* **ADIF - A/D Converter Interrupt Flag bit.**
  + **1** - A/D conversion is completed (bit must be cleared from within the software).
  + **0** - A/D conversion is not completed or has not started.
* **RCIF - EUSART Receive Interrupt Flag bit.**
  + **1** - The EUSART receive buffer is full. Bit is cleared by reading the RCREG register.
  + **0** - The EUSART receive buffer is not full.
* **TXIF - EUSART Transmit Interrupt Flag bit.**
  + **1** - The EUSART transmit buffer is empty. The bit is cleared by any write to the TXREG register.
  + **0** - The EUSART transmit buffer is full.
* **SSPIF - Master Synchronous Serial Port (MSSP) Interrupt Flag bit.**
  + **1** - The MSSP interrupt conditions during data transmit/receive have occurred. They differ depending on MSSP operating mode (SPI or I2C). This bit must be cleared from within the software before returning from the interrupt service routine.
  + **0** - No MSSP interrupt condition has occurred.
* **CCP1IF - CCP1 Interrupt Flag bit.**
  + **1** - CCP1 interrupt condition has occurred (CCP1 is unit for capturing, comparing and generating PWM signal). Depending on operating mode, capture or compare match has occurred. In both cases, bit must be cleared in software. This bit is not used in PWM mode.
  + **0** - No CCP1 interrupt condition has occurred.
* **TMR2IF - Timer2 to PR2 Interrupt Flag bit**
  + **1** - TMR2 (8-bit register) to PR2 match has occurred. This bit must be cleared from within the software prior to returning from the interrupt service routine.
  + **0** - No TMR2 to PR2 match has occurred.
* **TMR1IF - Timer1 Overflow Interrupt Flag bit**
  + **1** - The TMR1 register has overflowed. This bit must be cleared from within the software.
  + **0** - The TMR1 register has not overflowed.

**PIR2 Register**

The PIR2 register contains the interrupt flag bits.



**Legend: (-) Unimplemented bit, (R/W) - Readable/Writable Bit, (0) After reset, bit is cleared**

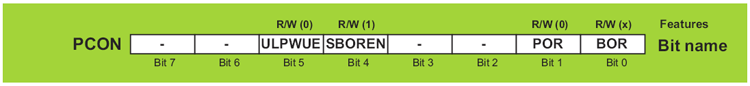
* **OSFIF - Oscillator Fail Interrupt Flag bit.**
  + **1** - System oscillator failed and clock input has changed to internal oscillator INTOSC. This bit must be cleared from within the software.
  + **0** - System oscillator operates normally.
* **C2IF - Comparator C2 Interrupt Flag bit.**
  + **1** - Comparator C2 output has changed (bit C2OUT). This bit must be cleared from within the software.
  + **0** - Comparator C2 output has not changed.
* **C1IF - Comparator C1 Interrupt Flag bit.**
  + **1** - Comparator C1 output has changed (bit C1OUT). This bit must be cleared from within the software.
  + **0** - Comparator C1 output has not changed.
* **EEIF - EE Write Operation Interrupt Flag bit.**
  + **1** - EEPROM write complete. This bit must be cleared from within the software.
  + **0** - EEPROM write is not complete or has not started yet.
* **BCLIF - Bus Collision Interrupt Flag bit.**
  + **1** - A bus collision has occurred in the MSSP when configured for I2C Master mode. This bit must be cleared from within the software.
  + **0** - No bus collision has occurred.
* **ULPWUIF - Ultra Low-power Wake-up Interrupt Flag bit.**
  + **1** - Wake-up condition has occurred. This bit must be cleared from within the software.
  + **0** - No Wake-up condition has occurred.
* **CCP2IF - CCP2 Interrupt Flag bit.**
  + **1** - CCP2 interrupt condition has occurred (unit for capturing, comparing and generating PWM signal). Depending on operating mode, capture or compare match has occurred. In both cases, the bit must be cleared from within the software. This bit is not used in PWM mode.
  + **0** - No CCP2 interrupt condition has occurred.

**Let's do it in mikroC...**

*// Module ULPWU activation sequence*  
   
**void** main() {  
 PORTA.0 = 1; *// PORTA.0 pin is set*  
 ANSEL = ANSELH = 0; *// All I/O pins are configured as digital*  
 TRISA = 0; *// PORTA pins are configured as outputs*  
 Delay\_ms(1); *// Charge capacitor*  
 ULPWUIF = 0; *// Clear flag*  
 PCON.ULPWUE = 1; *// Enable ULP Wake-up*  
 TRISA.0 = 1; *// PORTA.0 is configured as an input*  
 ULPWUIE = 1; *// Enable interrupt*  
 GIE = PEIE = 1; *// Enable peripheral interrupt*  
 **asm** SLEEP; *// Go to sleep mode*  
 ...  
 ...

**PCON register**

The PCON register contains only two flag bits used to differentiate between Power-on reset, Brown-out reset, Watchdog Timer reset and external reset over the MCLR pin.



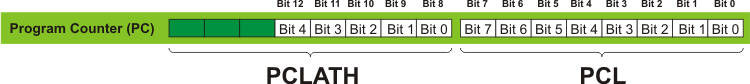
**Legend: (-) Unimplemented bit, (R/W) - Readable/Writable Bit, (1) - After reset, bit is set, (0) After reset, bit is cleared**

* **ULPWUE - Ultra Low-Power Wake-up Enable bit**
  + **1** - Ultra Low-Power Wake-up enabled.
  + **0** - Ultra Low-Power Wake-up disabled.
* **SBOREN - Software BOR Enable bit**
  + **1** - Brown-out Reset enabled.
  + **0** - Brown-out Reset disabled.
* **POR - Power-on Reset Status bit**
  + **1** - No Power-on reset has occurred.
  + **0** - Power-on reset has occurred. This bit must be set from within the software after a Power-on Reset occurs.
* **BOR - Brown-out Reset Status bit**
  + **1** - No Brown-out reset has occurred.
  + **0** - Brown-out reset has occurred. This bit must be set from within the software after a Brown-out Reset occurs.

**PCL AND PCLATH REGISTERS**

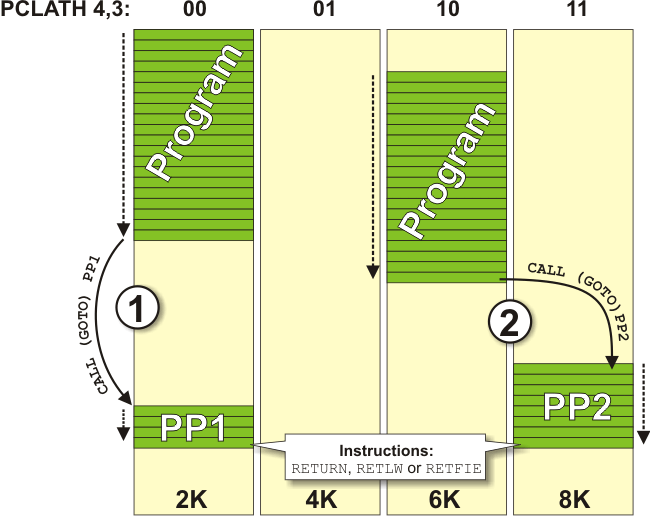
The size of the program memory of the PIC16F887 is 8K and has 8192 locations for program storing. For this reason, the program counter must be 13-bits wide (213 = 8192). To enable access to any program memory location during operation, it is necessary to access its address through SFRs. Since all SFRs are 8-bits wide, this addressing register is ‘artificially’ created by dividing its 13 bits into two independent registers PCLATH and PCL.

If the program execution doesn’t affect the program counter, the value of this register is automatically and constantly incremented +1, +1, +1, +1... In this way, the program is executed as it is written- instruction by instruction, followed by constant address increment.



If the program counter is changed from within the software, then there are several things that should be kept in mind in order to avoid problems:

* Eight lower bits (the low byte) come from the PCL register which is readable and writable, whereas five upper bits coming from the PCLATH register are write-only.
* The PCLATH register is cleared on any reset.
* In assembly language, the value of the program counter is marked with PCL and refers to 8 lower bits only. You should take care when using the ‘ADDWF PCL’ instruction. This is a jump instruction which specifies the target location by adding some number to the current address. It is often used when jumping into a look-up table or program branch table to read them. Aproblem arises if the current address is such that addition causes a change on some bit belonging to the higher byte of the PCLATH register.  
    
  Execution of any instruction upon the PCL register simultaneously causes the Program Counter bits to be replaced by the contents of the PCLATH register. However, the PCL register has access to only 8 lower bits of the instruction result and the following jump will be completely incorrect. The problem is solved by setting such instructions at addresses ending by xx00h. This enables the program to jump up to 255 locations. If longer jumps are executed by this instruction, the PCLATH register must be incremented by 1 for each PCL register overflow.
* On subroutine call or jump execution (instructions **CALL** and **GOTO**), the microcontroller is capable of providing only 11-bit addressing. Similar to RAM which is divided in ‘banks’, ROM is divided in four ‘pages’ in size of 2K each. Such instructions are executed within these pages without any problem. Simply put, since the processor is provided with 11-bit address from the program, it is capable of addressing any location within 2KB. Figure below illustrates the jump to the subroutine PP1 address.  
    
  However, if a subroutine or a jump address is not within the same page as the jump location, two ‘missing’- higher bits should be provided by writing to the PCLATH register. Figure below illustrates the jump to the subroutine PP2 address.

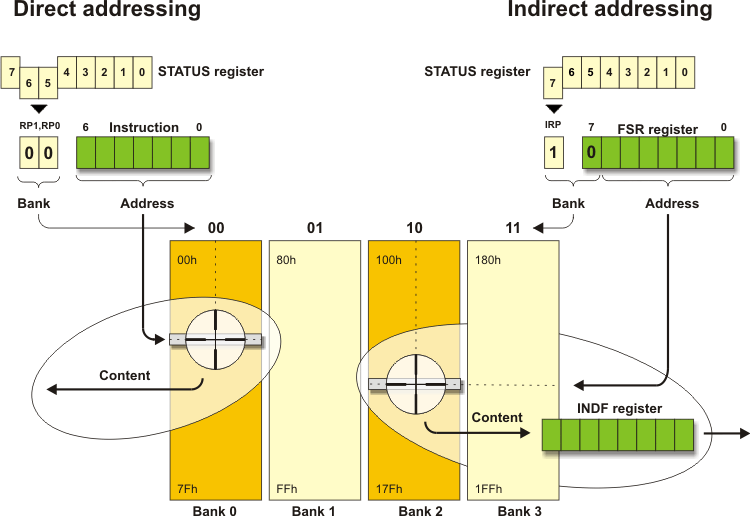


In both cases, when the subroutine reaches instructions **RETURN**, **RETLW** or **RETFIE** (return to the main program), the microcontroller will simply proceed with program execution from where it left off because the return address is pushed and saved onto the stack which, as mentioned, consists of 13-bit registers.

**INDIRECT ADDRESSING REGISTERS**

In addition to direct addressing, which is logical and clear (it is sufficient to specify the address of a register to read its contents), this microcontroller is capable of performing indirect addressing by means of the INDF and FSR registers. It sometimes makes the process of writing a program easier. The whole procedure is enabled because the INDF register is not true one (physically does not exist), but only specifies the register the address of which is located in the FSR register. For this reason, write or read from the INDF register actually means write or read from the register the address of which is located in the FSR register. In other words, registers’ addresses are specified in the FSR register, and their content is stored in the INDF register. The difference between direct and indirect addressing is illustrated in the figure below:

As seen, the problem with the ‘missing addressing bits’ is solved by a ‘borrow’ from another register. This time, it is the seventh bit, called the IRP bit of the STATUS register.



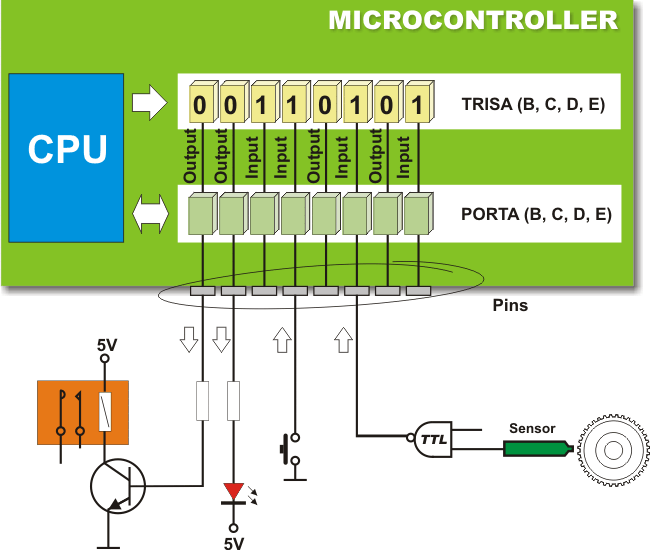
*One of the most important features of the microcontroller is a number of input/output pins which enable it to be connected to peripherals. The PIC16F887 has in total of 35 general-purpose I/O pins available, which is quite enough for most applications.*

**3.3 INPUT/OUTPUT PORTS**

In order to synchronize the operation of I/O ports with the internal 8-bit organization of the microcontroller, they are, similar to registers, grouped into five ports denoted by A, B, C, D and E. All of them have several features in common:

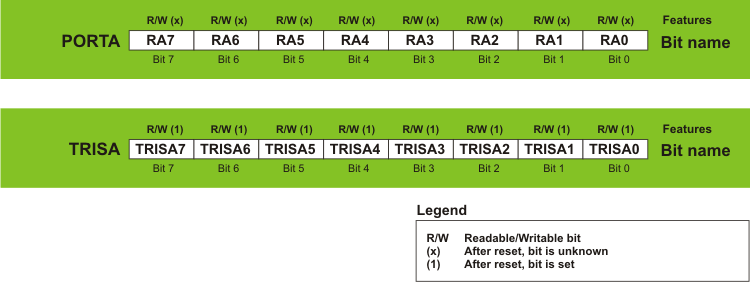
* For practical reasons, many I/O pins are multifunctional. If a pin performs any of these functions, it may not be used as a general-purpose input/output pin.
* Every port has its ‘satellite’, i.e. the corresponding TRIS register: TRISA, TRISB, TRISC etc. which determines the performance of port bits, but not their contents.

By clearing any bit of the TRIS register (bit=0), the corresponding port pin is configured as an output. Similarly, by setting any bit of the TRIS register (bit=1), the corresponding port pin is configured as an input. This rule is easy to remember 0 = **O**utput, 1 = **I**nput.



**PORTA and TRISA register**

Port A is an 8-bit wide, bidirectional port. Bits of the TRISA and ANSEL registers control the Port A pins. All Port A pins act as digital inputs/outputs. Five of them can also be analog inputs (denoted by AN):



RA0 = AN0 (determined by the ANS0 bit of the ANSELregister)  
  
RA1 = AN1 (determined by the ANS1 bit of the ANSELregister)  
  
RA2 = AN2 (determined by the ANS2 bit of the ANSELregister)  
  
RA3 = AN3 (determined by the ANS3 bit of the ANSELregister)  
  
RA5 = AN4 (determined by the ANS4 bit of the ANSELregister)

Similar to bits of the TRISA register determine which of the pins are to be configured as inputs and which ones as outputs, the appropriate bits of the ANSEL register determine whether pins are to be configured as analog inputs or digital inputs/outputs.

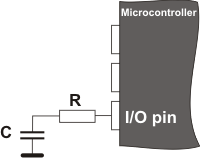
Each bit of this port has an additional function related to some of the built-in peripheral units, which will be described in later chapters. This chapter covers only the RA0 pin’s additional function since it is related to port A and the ULPWU unit.

**Let's do it in mikroC...**

*// The PORTA.2 pin is configured as a digital input.   
// All other PORTA pins are digital outputs*  
  
ANSEL = ANSELH = 0; *// All I/O pins are configured as digital*  
PORTA = 0; *// All PORTA pins are cleared*  
TRISA = 0b00000100; *// All PORTA pins except PORTA.2 are configured as outputs*  
...

**ULPWU UNIT**

The microcontroller is commonly used in devices which operate periodically and completely independently using a battery power supply. Minimum power consumption is one of the priorities here. Typical examples of such applications are: thermometers, fire detection sensors and the like. It is known that a reduction in clock frequency reduces the power consumption, thus one of the most convenient solutions to this problem is to slow down the clock, i.e. to use 32KHz quartz crystal instead of 20MHz.

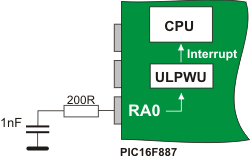


Setting the microcontroller to sleep mode is another step in the same direction. Still, the problem is how to wake up the microcontroller and set it to normal mode? It is obviously necessary to have an external signal to change the logic state of some of the pins. This signal must be generated by additional electronics, which causes higher power consumption of the entire device...

The ideal solution would be that the microcontroller wakes up periodically by itself, which is not impossible at all. The circuit which enables it is shown in figure on the left.

The principle of operation is simple:

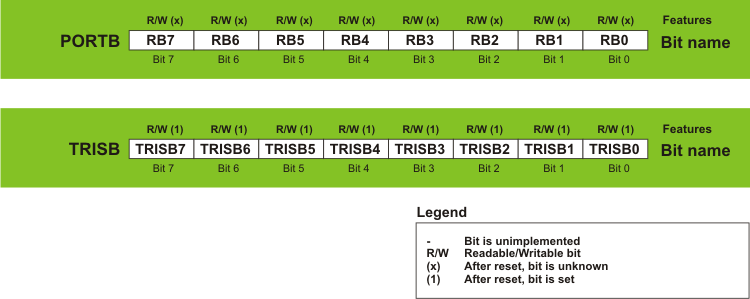
A pin is configured as an output and a logic one (1) is brought to it. This causes the capacitor to be charged. Immediately after this, the same pin is configured as an input. The change of logic state enables an interrupt and the microcontroller is set to *Sleep* mode. All that’s left now is to wait for the capacitor to discharge by the leakage current flowing out through the input pin. When it occurs, an interrupt takes place and the microcontroller proceeds with the program execution in normal mode. The whole procedure is repeated.



Theoretically, this is a perfect solution. The problem is that all pins able to cause an interrupt in this way are digital and have relatively large leakage current when their voltage is not close to the limit values Vdd (1) or Vss (0). In this case, the condenser is discharged for a short time since the current amounts to several hundreds of microamperes. This is why the ULPWU circuit, capable of registering slow voltage drops with minimum power consumption, was designed. Its output generates an interrupt, while its input is connected to one of the microcontroller pins. It is the RA0 pin. Referring to figure (R=200 ohms, C=1nF), discharge time is approximately 30mS, while a total consumption of the microcontroller is 1000 times lower (several hundreds of nanoamperes).

**PORTB and TRISB register**

Port B is an 8-bit wide, bidirectional port. Bits of the TRISB register determine the function of its pins.



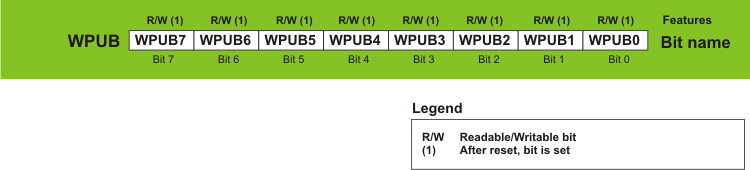
Similar to port A, a logic one (1) in the TRISB register configures the appropriate portB pin as an input and vice versa. Six pins of this port can act as analog inputs (AN). The bits of the ANSELH register determine whether these pins are to be configured as analog inputs or digital inputs/outputs:

RB0 = AN12 (determined by the ANS12 bit of the ANSELH register)  
  
RB1 = AN10 (determined by the ANS10 bit of the ANSELH register)  
  
RB2 = AN8 (determined by the ANS8 bit of the ANSELH register)  
  
RB3 = AN9 (determined by the ANS9 bit of the ANSELH register)  
  
RB4 = AN11 (determined by the ANS11 bit of the ANSELH register)  
  
RB5 = AN13 (determined by the ANS13 bit of the ANSELH register)

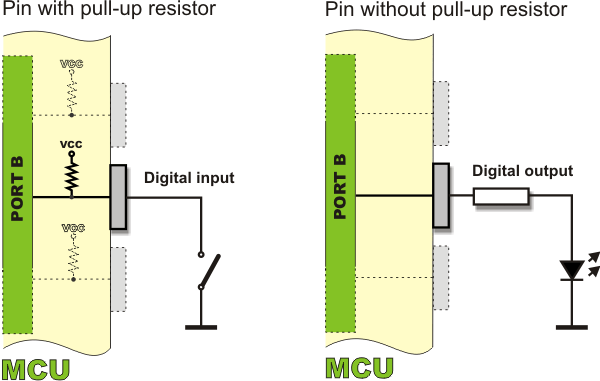
Each port B pin has an additional function related to some of the built-in peripheral units, which will be explained in later chapters.

This port has several features which distinguish it from other ports and make its pins commonly used:

* All the port B pins have built in *pull-up* resistors, which make them ideal for connection to push buttons (keyboard), switches and optocouplers. In order to connect these resistors to the microcontroller ports, the appropriate bit of the WPUB register should be set.\*

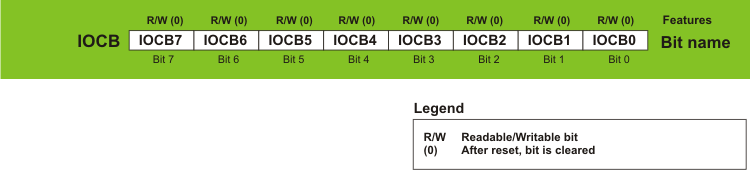


Having a high level of resistance (several tens of kiloohms), these ‘virtual’ resistors do not affect pins configured as outputs, but serves as a useful complement to inputs. As such, they are connected to the inputs of CMOS logic circuits. Otherwise, they would act as if they are floating due to their high input resistance.

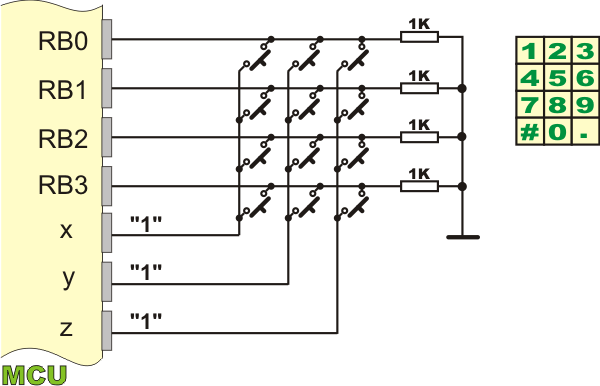


*\* Apart from the bits of the WPUB register, there is another bit affecting the installation of all pull-up resistors. It is the RBPU bit of the OPTION\_REG.*

* If enabled, each port B bit configured as an input may cause an interrupt by changing its logic state. In order to enable pins to cause an interrupt, the appropriate bit of the IOCB register should be set.



Thanks to these features, the port B pins are commonly used for checking push buttons on the keyboard because they unerringly register any button press. Thus, there is no need to ‘scan’ these inputs all the time.



When the X, Y and Z pins are configured as outputs set to logic one (1), it is only necessary to wait for an interrupt request which arrives upon any button press. After that, by combining zeros and ones on these outputs it is checked which push button is pressed.

**Let's do it in mikroC...**

*/\* The PORTB.1 pin is configured as a digital input. Any change of its logic state will cause  
an .i.n.terrupt. It also has a pull-up resistor. All other PORTB pins are digital outputs.\*/*  
  
ANSEL = ANSELH = 0; *// All I/O pins are configured as digital*  
PORTB = 0; *// All PORTB pins are cleared*  
TRISB = 0b00000010; *// All PORTB pins except PORTB.1 are configured as outputs*  
RBPU = 0; *// Pull-up resistors are enabled*  
WPUB1 = 1; *// Pull-up resistor is connected to the PORTB.1 pin*  
IOCB1 = 1; *// The PORTB.1 pin may cause an interrupt on logic state change*  
RBIE = GIE = 1; *// Interrupt is enabled*  
...

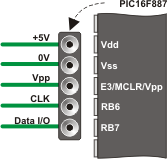
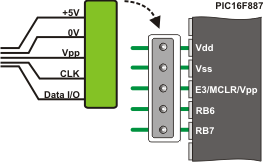
**PIN RB0/INT**

The RB0/INT pin is the only ‘true’ external interrupt source. It can be configured to react to signal raising edge (zero-to-one transition) or signal falling edge (one-to-zero transition). The INTEDG bit of the OPTION\_REG register selects the appropriate signal.

**RB6 AND RB7 PINS**

The PIC16F887 does not have any special pins for programming (the process of writing a program to ROM). Port pins, normally available as general-purpose I/O pins, are used for this purpose. To be more precise, it is about port B pins used for clock (RB6) and data transfer (RB7) during program loading. Besides, it is necessary to apply power supply voltage Vdd (5V) as well as appropriate voltage Vpp (12-14V) for FLASH memory programming. During programming, Vpp voltage is applied to the MCLR pin. You don’t have to think of all details concerning this process, nor which one of these voltages is applied first since the programmer’s electronics is in charge of that. What is very important here is that the program may be loaded to the microcontroller even after soldering it onto the target device. Normally, the loaded program can also be changed in the same way. This function is called ICSP (*In-Circuit Serial Programming*). In order to use it properly, it is necessary to plan ahead.

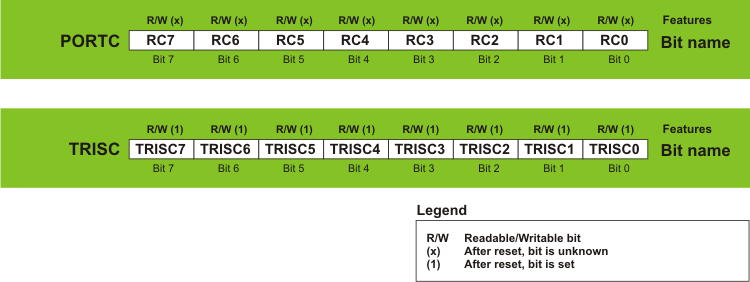
A piece of cake! It is only necessary to install a miniature 5-pin connector onto the target device so as to provide the microcontroller with necessary programming voltages. In order to prevent these voltages from interfering with other device electronics connected to microcontroller pins, all additional peripheral devices should be disconnected during the process of programming using resistors or jumpers.



As you can see, voltages applied to programmer's socket pins are the same as those used during ICSP programming

**PORTC and TRISC register**

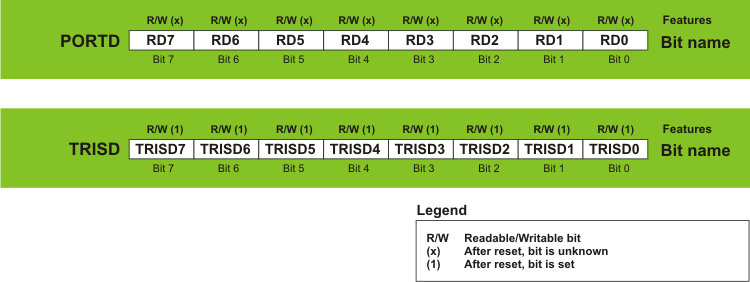
Port C is an 8-bit wide, bidirectional port. Bits of the TRISC register determine the function of its pins. Similar to other ports, a logic one (1) in the TRISC register configures the appropriate portC pin as an input.



All additional functions of port C bits will be explained later.

**PORTD and TRISD register**

Port D is an 8-bit wide, bidirectional port. Bits of the TRISD register determine the function of its pins. A logic one (1) in the TRISD register configures the appropriate portD pin as an input.

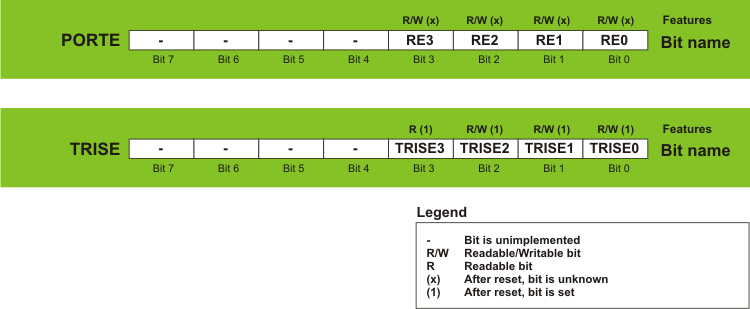


**PORTE and TRISE register**

Port E is a 4-bit wide, bidirectional port.

The TRISE register’s bits determine the function of its pins. Similar to other ports, a logic one (1) in the TRISE register configures the appropriate portE pin as an input.

The exception is the RE3 pin which is always configured as an input.



Similar to ports A and B, three pins can be configured as analog inputs in this case. The ANSELH register bits determine whether a pin will act as an analog input (AN) or digital input/output:

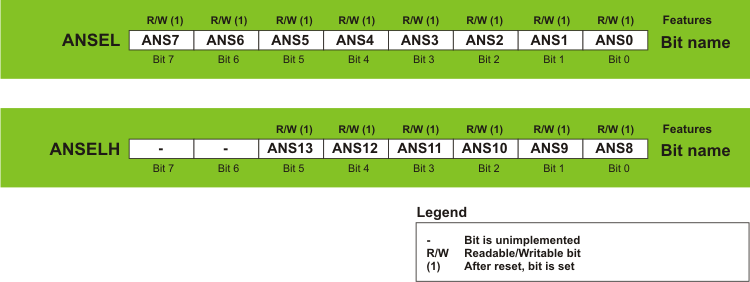
RE0 = AN5 (determined by the ANS5 bit of the ANSELregister);  
  
RE1 = AN6 (determined by the ANS6 bit of the ANSELregister); and  
  
RE2 = AN7 (determined by the ANS7 bit of the ANSELregister).

**Let's do it in mikroC...**

*/\* The PORTE.0 pin is configured as an analog input while another three pins of the same  
 port are configured as digital. \*/*  
...  
ANSEL = 0b00100000; *// The PORTE.0 pin is configured as analog*  
ANSELH = 0; *// All other I/O pins are configured as digital*  
TRISE = 0b00000001; *// All PORTE pins except PORTE.0 are configured as outputs*  
PORTE = 0; *// All PORTE pins are cleared*  
...

**ANSEL and ANSELH register**

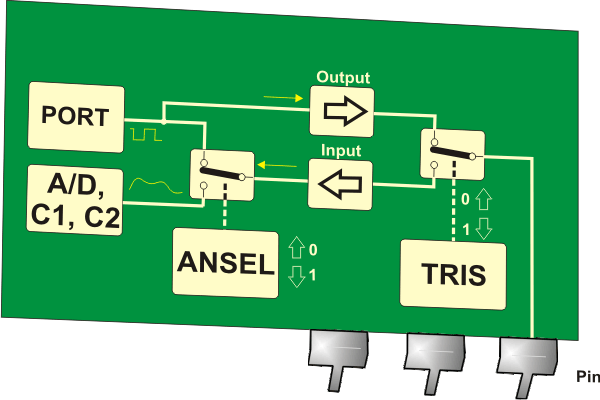
The ANSEL and ANSELH registers are used to configure the input mode of an I/O pin to analog or digital.



The rule is:

To configure a pin as an analog input, the appropriate bit of the ANSEL or ANSELH registers must be set (1). To configure a pin as a digital input/output, the appropriate bit must be cleared (0).

The state of the ANSEL bits has no influence on digital output functions. The result of any attempt to read a port pin configured as an analog input will be 0.



**In Short**

You will probably never write a program which doesn't use ports so the effort you make to learn all about them will definately pay off. Anyway, they are probaly the simplest modules within the microcontroller. This is how they are used:

* When designing a device, select a port through which the microcontroller will communicate to peripheral environment. If you use only digital inputs/outputs, select any port you want. If you intend to use some of the analog inputs, select the appropriate ports supporting such a pin configuration (AN0-AN13).
* Each port pin may be configured as either input or output. Bits of the TRISA, TRISB, TRISC, TRISD and TRISE registers determine how the appropriate port pins- PORTA, PORTB, PORTC, PORTD and PORTE will act. Simply...
* If you use some of the analog inputs, it is first necessary to set the appropriate bits of the ANSEL and ANSELH registers at the beginning of the program.
* If you use switches and push buttons as input signal source, connect them to port B pins because they have pull-up resistors. The use of these resistors is enabled by the RBPU bit of the OPTION\_REG register, whereas the installation of individual resistors is enabled by bits of the WPUB register.
* It is usually necessary to respond as soon as input pins change their logic state. However, it is not necessary to write a program for checking pins’ logic state. It is far simpler to connect such inputs to the PORTB pins and enable an interrupt to occur on every voltage change. Bits of the IOCB and INTCON registers are in charge of that.

*The PIC16F887 microcontroller has three completely separate timers/counters marked as TMR0, TMR1 and TMR2. If you want to learn more about them, read carefully this chapter.*

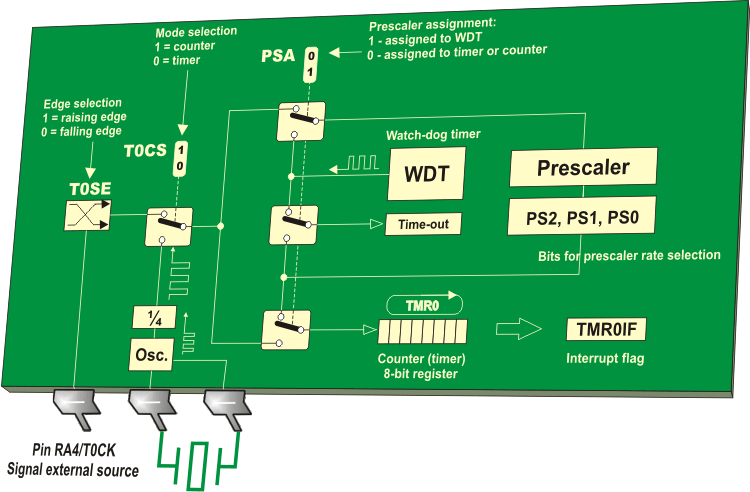
**3.4 TIMER TMR0**

The timer TMR0 has a wide range of application in practice. Very few programs don’t use it in some way. It is very convenient and easy to use for writing programs or subroutines for generating pulses of arbitrary duration, time measurement or counting external pulses (events) with almost no limitations.

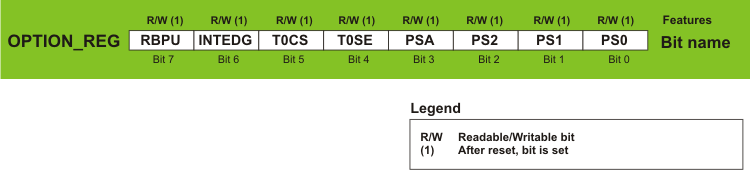
The timer TMR0 module is an 8-bit timer/counter with the following features:

* 8-bit timer/counter;
* 8-bit prescaler (shared with Watchdog timer);
* Programmable internal or external clock source;
* Interrupt on overflow; and
* Programmable external clock edge selection.

Figure below illustrates the timer TMR0 schematic with all bits which determine its operation. These bits are stored in the OPTION\_REG register.



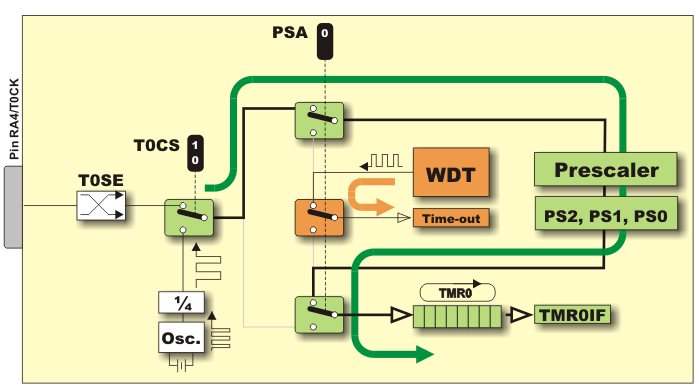
**OPTION\_REG Register**



* **RBPU - PORTB Pull-up enable bit**
  + 0 - PORTB pull-up resistors are disabled.
  + 1 - PORTB pins can be connected to pull-up resistors.
* **INTEDG - Interrupt Edge Select bit**
  + 0 - Interrupt on rising edge of the INT pin (0-1).
  + 1 - Interrupt on falling edge of the INT pin (1-0).
* **T0CS - TMR0 Clock Select bit**
  + 0 - Pulses are brought to TMR0 timer/counter input through the RA4 pin.
  + 1 - Timer uses internal cycle clock (Fosc/4).
* **T0SE - TMR0 Source Edge Select bit**
  + 0 - Increment on high-to-low transition on the TMR0 pin.
  + 1 - Increment on low-to-high transition on the TMR0 pin.
* **PSA - Prescaler Assignment bit**
  + 0 - Prescaler is assigned to the WDT.
  + 1 - Prescaler is assigned to the TMR0 timer/counter.
* **PS2, PS1, PS0 - Prescaler Rate Select bit**
  + Prescaler rate is adjusted by combining these bits. As seen in the table, the same combination of bits gives different prescaler rate for the timer/counter and watch-dog timer, respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **PS2** | **PS1** | **PS0** | **TMR0** | **WDT** |
| 0 | 0 | 0 | 1:2 | 1:1 |
| 0 | 0 | 1 | 1:4 | 1:2 |
| 0 | 1 | 0 | 1:8 | 1:4 |
| 0 | 1 | 1 | 1:16 | 1:8 |
| 1 | 0 | 0 | 1:32 | 1:16 |
| 1 | 0 | 1 | 1:64 | 1:32 |
| 1 | 1 | 0 | 1:128 | 1:64 |
| 1 | 1 | 1 | 1:256 | 1:128 |

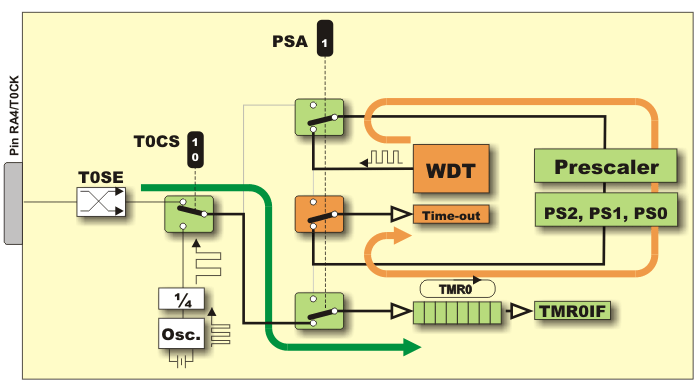
When PSA bit is cleared, prescaler is asigned to TMR0 timer/counter as ilustrated on the figure below:



**Let's do it in mikroC...**

*// In this example, TMR0 is configured as a timer and prescaler is assigned to it.*  
   
**unsigned** cnt; *// Define variable cnt*  
**void** interrupt() { *// Interrupt routine*  
 cnt++; *// Interrupt causes cnt to be incremented by 1*  
 TMR0 = 155; *// Timer (or counter) TMR0 returns its initial value*  
 INTCON = 0x20; *// Bit T0IE is set, bit T0IF is cleared*  
}  
  
**void** main() {  
 OPTION\_REG = 0x04; *// Prescaler (1:32) is assigned to the timer TMR0*  
 TMR0 = 155; *// Timer T0 counts from 155 to 255*  
 INTCON = 0xA0; *// Enable interrupt TMR0*  
 ...  
 ...  
   
 *// In the following example,TMR0 is configured as counter and prescaler is assigned to it*  
   
 OPTION\_REG = 0x20; *// Prescaler (1:2) is assigned to the counter TMR0*  
 TMR0 = 155; *// Counter T0 counts from 155 to 255*  
 INTCON = 0xA0; *// Enable interrupt TMR0*  
 ...  
 ...

When PSA bit is set, prescaler is asigned to watch-dog timer as ilustrated on the next figure:



**Let's do it in mikroC...**

*// In this example, prescaler (1:64) is assigned to Watch-dog timer.*  
   
**void** main() {  
 OPTION\_REG = 0x0E; *// Prescaler is assigned to WDT (1:64)*  
 **asm** CLRWDT; *// Assembly command to reset WDT*  
 ...  
 ...  
 **asm** CLRWDT; *// Assembly command to reset WDT*  
 ...

**Additionally it is also worth mentioning:**

* When the prescaler is assigned to the timer/counter, any write to the TMR0 register will clear the prescaler.
* When the prescaler is assigned to the watch-dog timer, the CLRWDT instruction will clear both the prescaler and WDT.
* Write to the TMR0 register, used as timer, will not cause the pulse counting to start immediately, but with two instruction cycles delay. Accordingly, it is necessary to adjust the value written to the TMR0 register.
* When the microcontroller is set in sleep mode, the clock oscillator is turned off. Overflow cannot occur since there are no pulses to count. This is why the TMR0 overflow interrupt cannot wake up the processor from Sleep mode.
* When used as an external clock counter, without prescaler, a minimal pulse length or a delay between two pulses must be 2 Tosc + 20 nS (Tosc is the oscillator clock signal period).
* When used as an external clock counter with prescaler, a minimal pulse length or interval between two pulses is only 10nS.
* The 8-bit prescaler register is not available to the user, which means that it cannot be directly read or written to.
* When changing the prescaler assignment from TMR0 to the watch-dog timer, the following instruction sequence written in assembly language must be executed in order to prevent the microcontroller from reset:

BANKSEL TMR0  
CLRWDT *;CLEAR WDT*  
CLRF TMR0 *;CLEAR TMR0 AND PRESCALER*  
BANKSEL OPTION\_REG  
BSF OPTION\_REG,PSA *;PRESCALER IS ASSIGNED TO WDT*  
CLRWDT *;CLEAR WDT*  
MOVLW b'11111000' *;SELECT BITS PS2,PS1,PS0 AND CLEAR*  
ANDWF OPTION\_REG,W *;THEM BY 'LOGIC AND' INSTRUCTION*  
IORLW b'00000101' *;BITS PS2, PS1, AND PS0 SET*  
MOVWF OPTION\_REG *;PRESCALER RATE TO 1:32*

* Likewise, when changing the prescaler assignment from the WDT to TMR0, the following instruction sequence, also written in assembly language, must be executed:

BANKSEL TMR0  
CLRWDT *;CLEAR WDT AND PRESCALER*  
BANKSEL OPTION\_REG  
MOVLW b'11110000' *;SELECT ONLY BITS PSA,PS2,PS1,PS0*  
ANDWF OPTION\_REG,W *;CLEAR THEM BY 'LOGIC AND' INSTRUCTION*  
IORLW b'00000011' *;PRESCALER RATE IS 1:16*  
MOVWF OPTION\_REG

**In Short**

In order to use TMR0 properly, it is necessary:

**Step 1: To select mode:**

* Timer mode is selected by the T0CS bit of the OPTION\_REG register, (T0CS: 0=timer, 1=counter).
* When used, the prescaler should be assigned to the timer/counter by clearing the PSA bit of the OPTION\_REG register. The prescaler rate is set by using the PS2-PS0 bits of the same register.
* When using interrupt, the GIE and TMR0IE bits of the INTCON register should be set.

**Step 2: Measuring and Counting**

**To measure time:**

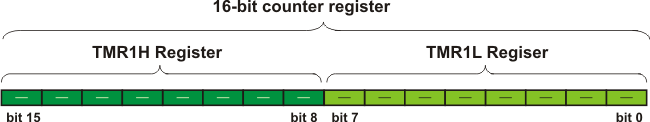
* Reset the TMR0 register or write some known value to it.
* Elapsed time (in microseconds when using 4MHz quartz) is measured by reading the TMR0 register.
* The flag bit TMR0IF of the INTCON register is automatically set every time the TMR0 register overflows. If enabled, an interrupt occurs.

**To count pulses:**

* The polarity of pulses are to be counted on the RA4 pin is selected by the TOSE bit of the OPTION\_REG register (T0SE: 0=positive, 1=negative pulses).
* Number of pulses may be read from the TMR0 register. The prescaler and interrupt are used in the same manner as in timer mode.

**3.5 TIMER TMR1**

Timer TMR1 module is a 16-bit timer/counter, which means that it consists of two registers (TMR1L and TMR1H). It can count up 65.535 pulses in a single cycle, i.e. before the counting starts from zero.

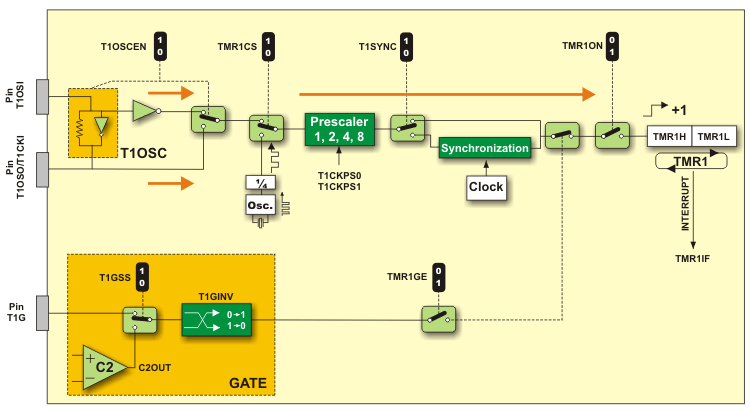


Similar to the timer TMR0, these registers can be read or written to at any moment. In case an overflow occurs, an interrupt is generated if enabled.

The timer TMR1 module may operate in one of two basic modes, that is as a timer or a counter. Unlike the TMR0 timer, both of these modes have additional functions.

The TMR1 timer has following features:

* 16-bit timer/counter register pair;
* Programmable internal or external clock source;
* 3-bit prescaler;
* Optional LP oscillator;
* Synchronous or asynchronous operation;
* Timer TMR1 gate control (count enable) via comparator or T1G pin;
* Interrupt on overflow;
* Wake-up on overflow (external clock); and
* Time base for Capture/Compare function.



**TIMER TMR1 CLOCK SOURCE SELECTION**

The TMR1CS bit of the T1CON register is used to select the clock source for this timer:

|  |  |
| --- | --- |
| **Clock Source** | **TMR1CS** |
| Fosc/4 | 0 |
| T1CKI pin | 1 |

When the internal clock source is selected, the TMR1H-TMR1L register pair will be incremented on multiples of Fosc pulses as determined by the prescaler.

When the external clock source is selected, this timer may operate as a timer or a counter. Clock in counter mode can be synchronized with the microcontroller internal clock or run asynchronously. In the event that an external clock oscillator is needed and the PIC16F887 microcontroller is using INTOSC with CLKOUT, timer TMR1 can use the LP oscillator as a clock source.

**TIMER TMR1 PRESCALER**

Timer TMR1 has a completely separate prescaler which allows 1, 2, 4 or 8 division of the clock input frequency. The prescaler is not directly readable or writable. However, the prescaler counter is automatically cleared after writing to the TMR1H or TMR1L register.

**TIMER TMR1 OSCILLATOR**

RC0/T1OSO and RC1/T1OSI pins are used to register pulses coming from peripheral electronics, but they also have an additional function. As seen in figure, they are simultaneously configured as both input (pin RC1) and output (pin RC0) of additional LP quartz oscillator (*Low Power*). This circuit is primarily designed for the operation at low frequencies (up to 200 KHz), more precisely, for the use of 32,768 KHz quartz crystal. Such crystals are used in quartz watches because it is easy to obtain one-second-long pulses by dividing this frequency.

Since this oscillator does not depend on internal clock, it can operate even in *sleep* mode. It is enabled by setting the T1OSCEN control bit of the T1CON register. The user must provide a software time delay (a few milliseconds) to enable the oscillator to start up properly.

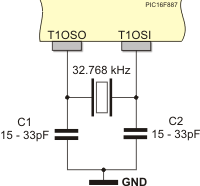


Table below shows the recommended values of the capacitors to suit the quartz oscillator. These values do not have to be exact. However, the general rule is: the higher the capacity, the higher the stability, which, at the same time, prolongs the time needed for oscillator stability.

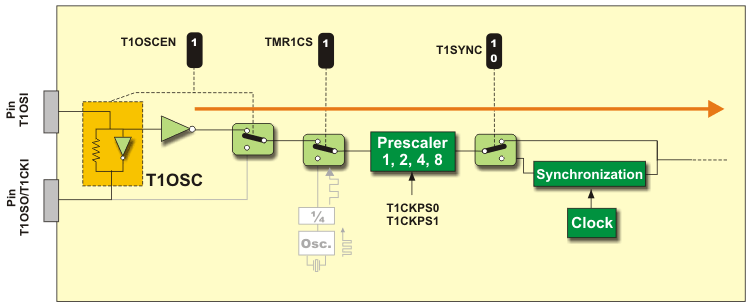
|  |  |  |  |
| --- | --- | --- | --- |
| **Oscillator** | **Frequency** | **C1** | **C2** |
| LP |  |  |  |
| 32 kHz | 33 pF | 33 pF |
| 100 kHz | 15 pF | 15 pF |
| 200 kHz | 15 pF | 15 pF |

**TIMER TMR1 GATE**

Timer TMR1 gate source is software configurable to be the T1G pin or the output of comparator C2. This gate allows the timer to directly time external events using the logic state of the T1G pin or analog events using the comparator C2 output. Refer to figure on the previous page. In order to measure a signal duration, it is sufficient to enable this gate and count pulses passing through it.

**THE USE OF TIMER TMR1 OSCILLATOR**

The power consumption of the microcontroller is reduced to the lowest level in Sleep mode since the main power consumer - the oscillator - doesn’t operate. It is easy to set the microcontroller in this mode- by executing the *SLEEP* instruction. The problem is how to wake up the microcontroller because only an interrupt can make it happen. Since the microcontroller 'sleeps', an interrupt must be triggered by external electronics. It gets incredibly complicated if it is necessary to wake up the microcontroller at regular time intervals...



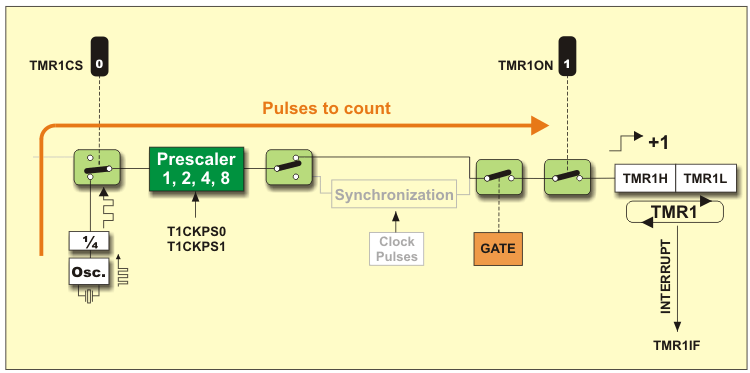
In order to solve this problem, a completely independent *Low Power* quartz oscillator, capable of operating in *sleep* mode, is built into the PIC16F887 microcontroller. Simply put, a previously separate circuit is now built into the microcontroller and assigned to the timer TMR1. The oscillator is enabled by setting the T1OSCEN bit of the T1CON register. The TMR1CS bit of the same register is then used to enable the timer TMR1 to use pulse sequences from that oscillator.

* A signal generated by this quartz oscillator is synchronized with the microcontroller clock by clearing the T1SYNC bit. In this case, the timer cannot operate in sleep mode because the circuit for synchronization uses the microcontroller clock.
* The TMR1 register overflow interrupt can be enabled. If the T1SYNC bit is set, such interrupts will also occur in *sleep* mode.

**TMR1 IN TIMER MODE**

In order to select this mode, it is necessary to clear the TMR1CS bit. After this, the 16-bit register will be incremented on every pulse generated by the internal oscillator. If the 4MHz quartz crystal is in use, it will be incremented every microsecond.

In this mode, the T1SYNC bit does not affect the timer because it counts internal clock pulses. Since the whole electronics uses these pulses, there is no need for synchronization.



The microcontroller’s clock oscillator does not operate during *sleep* mode so the timer register overflow cannot cause any interrupt.

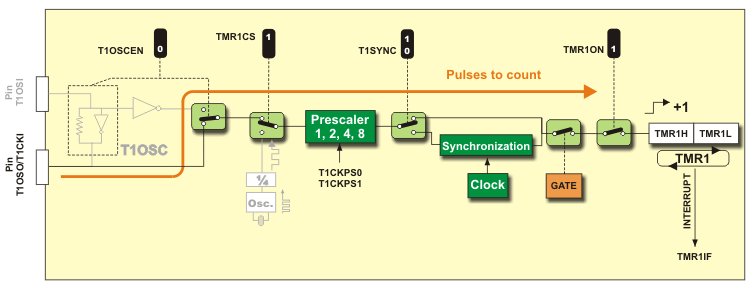
**Let's do it in mikroC...**

*// In this example, TMR1 is configured as a timer with the prescaler rate 1:8. Every time  
// TMR1H and TMR1L registers overflow occurs, an interrupt will be requested.*  
  
**void** main() {  
 PIR1.TMR1IF = 0; *// Reset the TMR1IF flag bit*  
 TMR1H = 0x22; *// Set initial value for the timer TMR1*  
 TMR1L = 0x00;  
 TMR1CS = 0; *// Timer1 counts pulses from internal oscillator*  
 T1CKPS1 = T1CKPS0 = 1; *// Assigned prescaler rate is 1:8*  
 PIE1.TMR1IE = 1; *// Enable interrupt on overflow*  
 INTCON = 0xC0; *// Enable interrupt (bits GIE and PEIE)*  
 TMR1ON = 1; *// Turn the timer TMR1 on*  
 ...

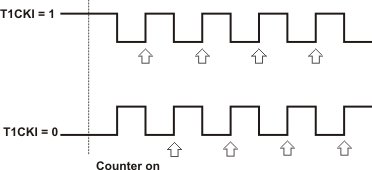
**TMR1 IN COUNTER MODE**

Timer TMR1 starts to operate as a counter by setting the TMR1CS bit. It counts pulses brought to the PC0/T1CKI pin and is incremented on the rising edge of the external clock input T1CKI. If the control bit T1SYNC of the T1CON register is cleared, the external clock inputs will be synchronized on their way to the TMR1 register. In other words, the timer TMR1 is synchronized to the microcontroller system clock and is called a synchronous counter.

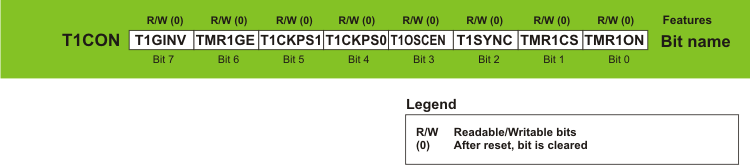
When the microcontroller, operating in this way, is set in *sleep* mode, the TMR1H and TMR1L timer registers are not incremented even though clock pulses appear on the input pins. Since the microcontroller system clock doesn’t run in this mode, there are no clock inputs to be used for synchronization. However, the prescaler will continue to run as far as there are clock pulses on the pins because it is just a simple frequency divider.



This counter registers a logic one (1) on input pins. It is important to know that at least one falling edge must be registered prior to starting pulse counting. Refer to figure on the left. The arrows in figure denote counter increments.



**T1CON Register**



**T1GINV** - Timer1 Gate Invert bit acts as logic state inverter on the T1G pin gate or the comparator C2 output (C2OUT) gate. It enables the timer to mea sure time whilst the gate is high or low.

* 1 - Timer 1 counts when the T1G pin or bit C2OUT gate is high (1).
* 0 - Timer 1 counts when the T1G pin or bit C2OUT gate is low (0).

**TMR1GE** - Timer1 Gate Enable bit determines whether the T1G pin or comparator C2 output (C2OUT) gate will be active or not. This bit is functional only in the event that the timer TMR1 is on (bit TMR1ON = 1). Otherwise, this bit is ignored.

* 1 - Timer TMR1 is on only if Timer1 gate is not active.
* 0 - Gate has no influence on the timer TMR1.

**T1CKPS1, T1CKPS0** - Determine the rate of the prescaler assigned to the timer TMR1.

|  |  |  |
| --- | --- | --- |
| **T1CKPS1** | **T1CKPS0** | **Prescaler Rate** |
| 0 | 0 | 1:1 |
| 0 | 1 | 1:2 |
| 1 | 0 | 1:4 |
| 1 | 1 | 1:8 |

**T1OSCEN** - LP Oscillator Enable Control bit

* 1 - LP oscillator is enabled for timer TMR1 clock (oscillator with low power consumption and frequency 32.768 kHz).
* 0 - LP oscillator is off.

**T1SYNC** - Timer1 External Clock Input Synchronization Control bit enables synchronization of the LP oscillator input or T1CKI pin input with the microcontroller internal clock. This bit is ignored while counting pulses from the main oscillator (bit TMR1CS = 0).

* 1 - Do not synchronize external clock input.
* 0 - Synchronize external clock input.

**TMR1CS** - Timer TMR1 Clock Source Select bit

* 1 - Count pulses on the T1CKI pin (on the rising edge 0-1).
* 0 - Count pulses of the microcontroller internal clock.

**TMR1ON** - Timer1 On bit

* 1 - Enable timer TMR1.
* 0 - Stop timer TMR1.

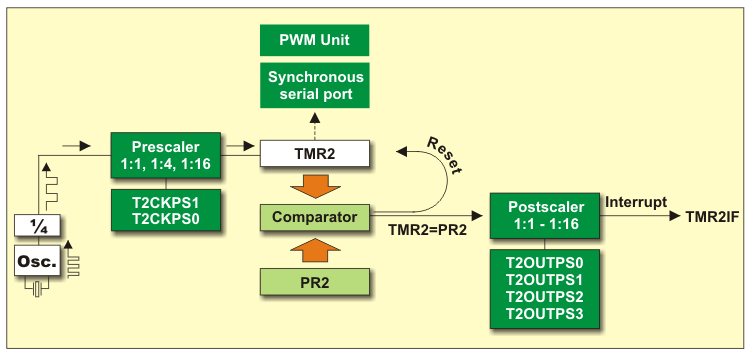
**In Short**

In order to use the timer TMR1 properly, it is necessary to perform the following:

* Since it is not possible to turn off the prescaler, its rate should be adjusted by using bits T1CKPS1 and T1CKPS0 of the register T1CON (Refer to table).
* Select the mode by the TMR1CS bit of the same register (TMR1CS: 0=the clock source is quartz oscillator, 1= the clock source is supplied externally).
* By setting the T1OSCEN bit of the same register, the oscillator is enabled and the TMR1H and TMR1L registers are incremented on every clock input. Counting stops by clearing this bit.
* The prescaler is cleared by clearing or writing to the counter registers.
* By filling both timer registers, the flag TMR1IF is set and counting starts from zero.

**3.6 TIMER TMR2**

Timer TMR2 module is an 8-bit timer which operates in a very specific way.



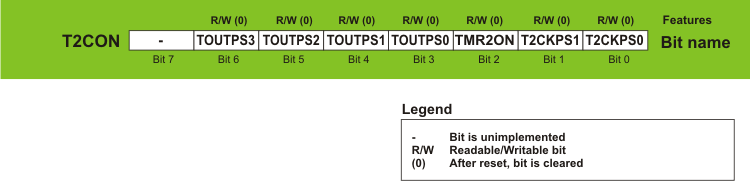
Pulses from the quartz oscillator first pass through the prescaler the rate of which may be changed by combining the T2CKPS1 and T2CKPS0 bits. The output of the prescaler is then used to increment the TMR2 register starting from 00h. The values of TMR2 and PR2 are constantly compared and the TMR2 register keeps on being incremented until it matches the value in PR2. When the match occurs, the TMR2 register is automatically cleared to 00h. The timer TMR2 postscaler is incremented and its output is used to generate an interrupt if it is enabled.

The TMR2 and PR2 registers are both fully readable and writable. Counting may be stopped by clearing the TMR2ON bit, which results in power saving.

The moment of TMR2 reset may also be used to determine the baud rate of synchronous serial communication.

The timer TMR2 is controlled by several bits of the T2CON register.

**T2CON Register**



**TOUTPS3 - TOUTPS0** - Timer2 Output Postcaler Select bits are used to determine the postscaler rate according to the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

**TMR2ON** - Timer2 On bit turns the timer TMR2 on.

* 1 - Timer TMR2 is on.
* 0 - Timer TMR2 is off.

**T2CKPS1, T2CKPS0** - Timer2 Clock Prescale bits determine the prescaler rate:

|  |  |  |
| --- | --- | --- |
| **T2CKPS1** | **T2CKPS0** | **Prescaler Rate** |
| 0 | 0 | 1:1 |
| 0 | 1 | 1:4 |
| 1 | x | 1:16 |

**In Short**

When using the TMR2 timer, you should know several specific details related to its registers:

* At the moment of powering on, the PR2 register contains the value FFh.
* Both prescaler and postscaler are cleared by writing to the TMR2 register.
* Both prescaler and postscaler are cleared by writing to the T2CON register.
* On any reset - you guess, both prescaler and postscaler are cleared.

*CCP modules can operate in many different modes, which makes them the most complicated to deal with. Just try to analyze their operation on the basis of the tables describing bit functions and you will understand what we are talking about. So, if you use some of the CCP module, first select the mode you need, analyze the appropriate figure and then change bits of the registers. Or else...*

**3.7 CCP MODULES**

The CCP module (*Capture/Compare/PWM*) is a peripheral which allows the user to time and control different events.

**Capture Mode** provides access to the current state of a register which constantly changes its value. In this case, it is the timer TMR1 register.

**Compare Mode** constantly compares values of two registers. One of them is the timer TMR1 register. This circuit also allows the user to trigger an external event when a predetermined amount of time has expired.

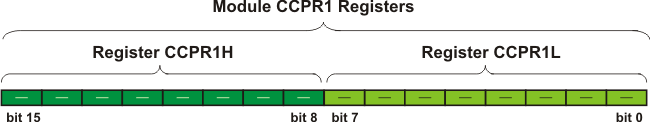
**PWM** (*Pulse Width Modulation*) can generate signals of varying frequency and duty cycle on one or more output pins.

The PIC16F887 microcontroller has two CCP modules- CCP1 and CCP2.

Both of them are identical in normal mode of operation, while the Enhanced PWM features are available on CCP1 only. This is why this chapter gives a detailed description of the CCP1 module. Concerning CCP2, only the features distinguishing it from CCP1 will be covered.

**CCP1 MODULE**

A central part of this circuit is a 16-bit register CCPR1 which consists of the CCPR1L and CCPR1H registers. It is used for capturing or comparing with binary numbers stored in the timer register TMR1 (TMR1H and TMR1L).



If enabled by software, the timer TMR1 reset may occur on match in Compare mode. Besides, the CCP1 module can generate PWM signals of varying frequency and duty cycle.

Bits of the CCP1CON register are in control of the CCP1 module.

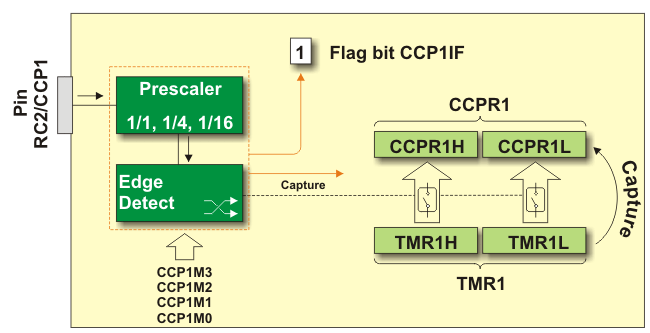
**CCP1 IN CAPTURE MODE**

In this mode, the timer register TMR1 (consisting of TMR1H and TMR1L) is copied to the CCP1 register (consisting of CCPR1H and CCPR1L) in the following situations:

* Every falling edge (1 -> 0) on the RC2/CCP1 pin;
* Every rising edge (0 -> 1) on the RC2/CCP1 pin;
* Every 4th rising edge (0 -> 1) on the RC2/CCP1 pin; and
* Every 16th rising edge (0 -> 1) on the RC2/CCP1 pin.

A combination of the four bits (CCP1M3 - CCP1M0) of the control register determines which of these events will cause 16-bit data to be transfered. In addition, the following conditions must be met:

* The RC2/CCP1 pin must be configured as an input; and
* TMR1 module must operate as timer or synchronous counter.



The flag bit CCP1IF is set when capture is made. If the CCP1IE bit of the PIE1 register is set when it happens, an interrupt occurs.

When the CCP1 module exsits the capture mode, an unwanted capture interrupt may be generated. In order to avoid this, both a bit enabling CCP1IE interrupt and flag bit CCP1IF should be cleared prior to any change occurs in the control register.

Unwanted interrupts may also be generated by switching from one capture prescaler to another. To avoid this, the CCP1 module should be temporarily switched off before changing the prescaler.

The following program sequence, written in assembly language, is recommended:

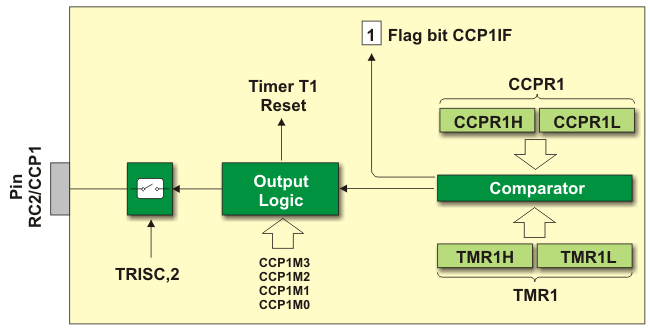
BANKESEL CCP1CON  
CLRF CCP1CON *;CONTROL REGISTER IS CLEARED*  
 *;CCP1 MODULE IS OFF*  
MOVLW XX *;NEW PRESCALER MODE IS SELECTED*  
MOVWF CCP1CON *;NEW VALUE IS LOADED TO THE CONTROL REGISTER*  
 *;CCP1 MODULE IS SIMULTANEOUSLY SWITCHED ON*

**Let's do it in mikroC...**

...  
**ASM** {  
 BANKESEL CCP1CON  
 CLRF CCP1CON *// CONTROL REGISTER IS CLEARED  
 // CCP1 MODULE IS OFF*  
 MOVLW XX *// NEW PRESCALER MODE IS SELECTED*  
 MOVWF CCP1CON *// NEW VALUE IS LOADED TO THE CONTROL REGISTER*  
} *// CCP1 MODULE IS SIMULTANEOUSLY SWITCHED ON*  
...

**CCP1 IN COMPARE MODE**

In this mode, the value stored in the CCP1 register is constantly compared to the value stored in the timer register TMR1. When a match occurs, the logic state of the RC2/CCP1 output pin may be changed, which depends on the state of bits in the control register (CCP1M3 - CCP1M0). The flag-bit CCP1IF will be simultaneously set.

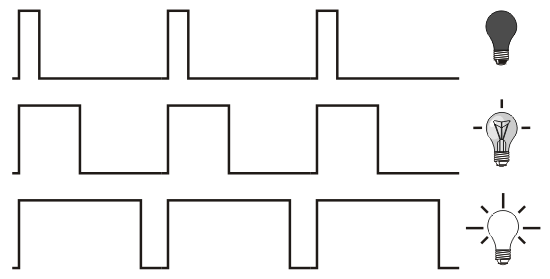


To set the CCP1 module to operate in this mode, two conditions must be met:

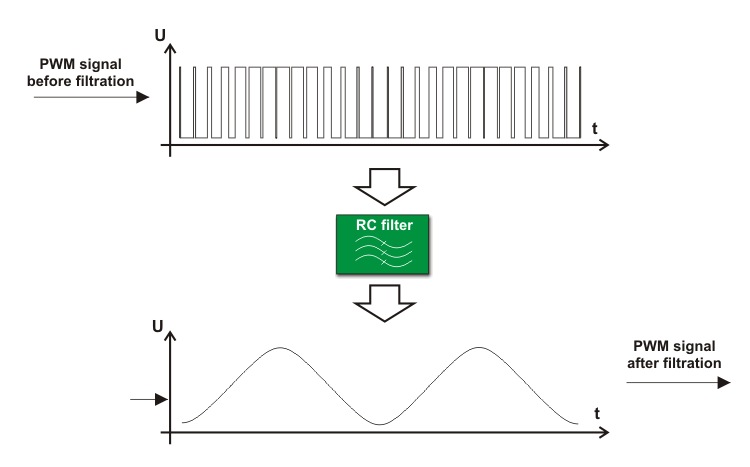
* The RC2/CCP1 pin must be configured as an output; and
* Timer TMR1 must be synchronized with internal clock.

**CCP1 IN PWM MODE**

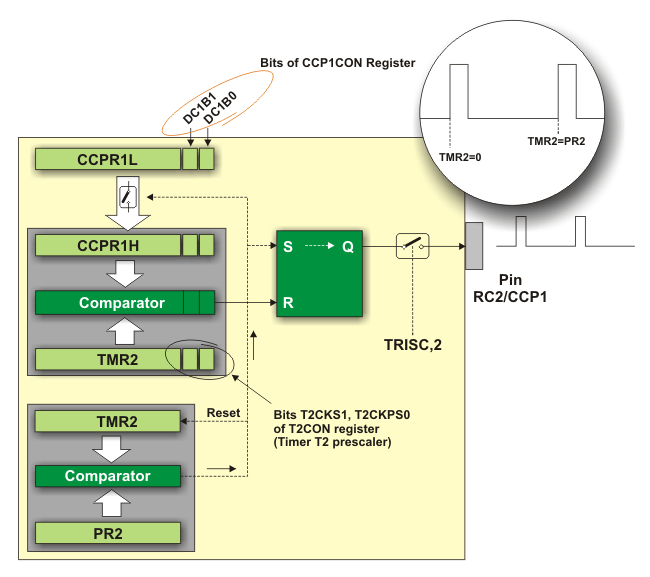
Signals of varying frequency and duty cycle have a wide range of application in automation. A typical example is a power control circuit. Refer to figure below. If a logic zero (0) indicates the switch-off and a logic one (1) indicates the switch-on, the electrical power that load consumers will be directly proportional to the pulse duration. This ratio is often called *Duty Cycle*.



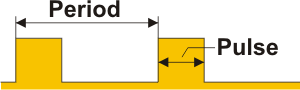
Another example, common in practice, is the use of PWM signals in the circuit for generating signals of arbitrary waveforms such as sinusoidal waveform. See figure below:



Devices which operate in this way are often used in practice as adjustable frequency drivers controlling the electric motor (speed, acceleration, deceleration etc.).



The figure above shows the block diagram of the CCP1 module set in PWM mode. In order to generate a pulse of arbitrary form on its output pin, it is necessary to set pulse period (frequency) and pulse duration.



**PWM PERIOD**

The output pulse period (T) is determined by the PR2 register of the timer TMR2. The PWM period can be calculated using the following equation:

PWM Period = (PR2 +1) \* 4Tosc \* TMR2 Prescale Value

If the PWM period (T) is known, then it is easy to determine the signal frequency F because these two values are related by equation F=1/T.

**PWM DUTY CYCLE**

The PWM duty cycle is specified by using in total of 10 bits: eight MSbs of the CCPR1L register and two additional LSbs of the CCP1CON register (DC1B1 and DC1B0). The result is a 10-bit number contained in the formula:

Pulse Width = (CCPR1L,DC1B1,DC1B0) \* Tosc \* TMR2 Prescale Value

The following table shows how to generate PWM signals of varying frequency if the microcontroller uses 20 MHz quartz-crystal (Tosc=50nS).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Frequency [KHz]** | **1.22** | **4.88** | **19.53** | **78.12** | **156.3** | **208.3** |
| TMR2 Prescaler | 16 | 4 | 1 | 1 | 1 | 1 |
| PR2 Register | FFh | FFh | FFh | 3Fh | 1Fh | 17h |

Just two more things:

* The output pin will be constantly set if the pulse width is by negligence determined to be larger than PWM period.
* In this application, the timer TMR2 postscaler cannot be used for generation of longer PWM periods.

**PWM RESOLUTION**

An PWM signal is nothing more than a pulse sequence with varying duty cycle. For one specified frequency (number of pulses per second), there is a limited number of duty cycle combinations. This number represents a resolution measured by bits. For example, a 10- bit resolution will result in 1024 discrete duty cycles, whereas an 8-bit resolution will result in 256 discrete duty cycles etc. In relation to this microcontroller, the resolution is determined by the PR2 register. The maximum value is obtained by writing number FFh.

PWM frequencies and resolutions (Fosc = 20MHz):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PWM Frequency** | **1.22kHz** | **4.88kHz** | **19.53kHz** | **78.12kHz** | **156.3kHz** | **208.3kHz** |
| Timer Prescale | 16 | 4 | 1 | 1 | 1 | 1 |
| PR2 Value | FFh | FFh | FFh | 3Fh | 1Fh | 17h |
| Maximum Resolution | 10 | 10 | 10 | 8 | 7 | 6 |

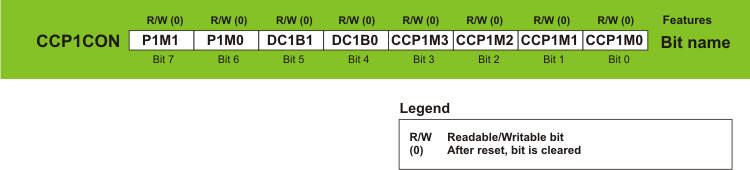
PWM frequencies and resolutions (Fosc = 8MHz):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **PWM Frequency** | **1.22kHz** | **4.90kHz** | **19.61kHz** | **76.92kHz** | **153.85kHz** | **200.0kHz** |
| Timer Prescale | 16 | 4 | 1 | 1 | 1 | 1 |
| PR2 Value | 65h | 65h | 65h | 19h | 0Ch | 09h |
| Maximum Resolution | 8 | 8 | 8 | 6 | 5 | 5 |

**Let's do it in mikroC...**

*/\* In this example, PWM module is initialized and set to give a pulse train of 50% dutycycle.  
For this purpose, functions PWM1\_Init(), PWM1\_Start() and PWM1\_Set\_Duty() are used.  
All of them are already contained in the mikroC PRO for PIC PWM library and just need to  
be copied to the program. \*/*  
  
**unsigned short** duty\_c; *// Define variable duty\_c*  
  
**void** initMain() {  
 ANSEL = ANSELH = 0; *// All I/O pins are configured as digital*  
 PORTC = TRISC = 0; *// Initial state of port C output pins*  
 PWM1\_Init(5000); *// PWM module initialization (5KHz)*  
}  
  
**void** main() {  
 initMain();  
 duty\_c = 127; *// Initial value of duty-cycle*  
 PWM1\_Start(); *// Start PWM1 module*  
 PWM1\_Set\_Duty(duty\_c); *// Set PWM duty-cycle to 50%*  
 ...  
 ...

**CCP1CON Register**



**P1M1, P1M0** - PWM Output Configuration bits - In all modes, except for PWM, the P1A pin is *Capture/Compare* module input. P1B, P1C and P1D pins act as input/output port D pins. In PWM mode, these bits affect the operation of the CCP1 module as shown in table below:

|  |  |  |
| --- | --- | --- |
| **P1M1** | **P1M0** | **Mode** |
| 0 | 0 | **PWM with single output** |
| Pin P1A outputs modulated signal. Pins P1B, P1C and P1D are port D input/output |
| 0 | 1 | **Full Bridge - Forward configuration** |
| Pin P1D outputs modulated signal Pin P1A is active Pins P1B and P1C are inactive |
| 1 | 0 | **Half Bridge configuration** |
| Pins P1A and P1B output modulated signal PinsP1C and P1D are port D input/output |
| 1 | 1 | **Full Bridge - Reverse configuration** |
| Pin P1B outputs modulated signal Pin P1C is active Pins P1A and P1D are inactive |

**DC1B1, DC1B0** - PWM Duty Cycle Least Significant bits - are only used in PWM mode in which they represent two least significant bits of a 10-bit number. This number determines duty cycle of the PWM signal. The rest of bits (8 in total) are stored in the CCPR1L register.

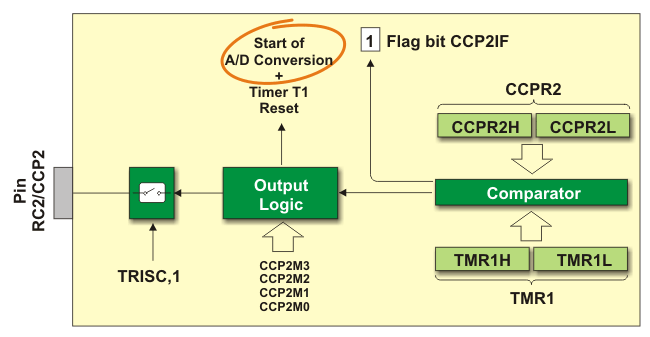
**CCP1M3 - CCP1M0** - CCP1 Mode Select bits determine the mode of the CCP1 module.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CCP1M3** | **CCP1M2** | **CCP1M1** | **CCP1M0** | **Mode** |
| 0 | 0 | 0 | 0 | Module is disabled (reset) |
| 0 | 0 | 0 | 1 | Unused |
| 0 | 0 | 1 | 0 | **Compare mode** |
| CCP1IF bit is set on match |
| 0 | 0 | 1 | 1 | Unused |
| 0 | 1 | 0 | 0 | **Capture mode** |
| Every falling edge on the CCP1 pin |
| 0 | 1 | 0 | 1 | **Capture mode** |
| Every rising edge on the CCP1 pin |
| 0 | 1 | 1 | 0 | **Capture mode** |
| Every 4th rising edge on the CCP1 pin |
| 0 | 1 | 1 | 1 | **Capture mode** |
| Every 16th rising edge on the CCP1 pin |
| 1 | 0 | 0 | 0 | **Compare mode** |
| Output and CCP1IF bit are set on match |
| 1 | 0 | 0 | 1 | **Compare mode** |
| Output is cleared and CCP1IF bit is set on match |
| 1 | 0 | 1 | 0 | **Compare mode** |
| Interrupt request arrives and bit CCP1IF is set on match |
| 1 | 0 | 1 | 1 | **Compare mode** |
| Bit CCP1IF is set and timers 1 or 2 registers are cleared |
| 1 | 1 | 0 | 0 | **PWM mode** |
| Pins P1A and P1C are active-high Pins P1B and P1D are active-high |
| 1 | 1 | 0 | 1 | **PWM mode** |
| Pins P1A and P1C are active-high Pins P1B and P1D are active-low |
| 1 | 1 | 1 | 0 | **PWM mode** |
| Pins P1A and P1C are active-low Pins P1B and P1D are active-high |
| 1 | 1 | 1 | 1 | **PWM mode** |
| Pins P1A and P1C are active-low Pins P1B and P1D are active-low |

**CCP2 MODULE**

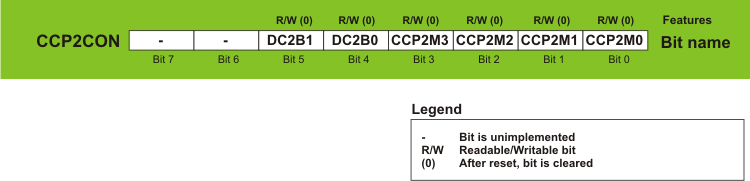
Excluding the different names of registers and bits, this module is a very good copy of the CCP1 module set in normal mode. There is only one true difference between them when CCP2 operates in Compare mode.

The difference refers to the timer T1 reset signal. Namely, if A/D converter is enabled, at the moment the values of the TMR1 and CCPR2 registers match, the timer T1 reset signal will automatically start A/D conversion.



Similar to the pervious module, this circuit is also under control of the control register bits. This time, it is the CCP2CON register.

**CCP2CON Register**



**DC2B1, DC2B0** - PWM Duty Cycle Least Significant bits - are only used in PWM mode representing two least significant bits of a 10-bit number. This number determines duty cycle of the PWM signal. The rest of bits (8 in total) are stored in the CCPR2L register.

**CCP2M3 - CCP2M0** - CCP2 Mode Select bits select CCP2 mode.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CCP2M3** | **CCP2M2** | **CCP2M1** | **CCP2M0** | **Mode** |
| 0 | 0 | 0 | 0 | Module is disabled (reset) |
| 0 | 0 | 0 | 1 | Unused |
| 0 | 0 | 1 | 0 | Unused |
| 0 | 0 | 1 | 1 | Unused |
| 0 | 1 | 0 | 0 | **Capture mode** |
| Every falling edge on the CCP2 pin |
| 0 | 1 | 0 | 1 | **Capture mode** |
| Every raising edge on the CCP2 pin |
| 0 | 1 | 1 | 0 | **Capture mode** |
| Every 4th rising edge on the CCP2 pin |
| 0 | 1 | 1 | 1 | **Capture mode** |
| Every 16th rising edge on the CCP2 pin |
| 1 | 0 | 0 | 0 | **Compare mode** |
| Output and CCP2IF bit are set on match |
| 1 | 0 | 0 | 1 | **Compare mode** |
| Output is cleared and CCP2IF bit is set on match |
| 1 | 0 | 1 | 0 | **Compare mode** |
| Interrupt is generated, CCP2IF bit is set and CCP2 pin is unaffected on match |
| 1 | 0 | 1 | 1 | **Compare mode** |
| CCP2IF bit is set, Timer 1 registers are cleared, A/D conversion is started if the A/D converter is on on match |
| 1 | 1 | x | x | **PWM mode** |

**In Short**

**Setting up CCP1 module for PWM operation**

In order to set up the CCP module for PWM operation, the following steps should be taken:

* Disable the CCP1 output pin. It should be configured as an input.
* Set the PWM period by loading the PR2 register.
* Configure the CCP module to operate in the PWM mode by combining bits of the CCP1CON register.
* Set duty cycle of the PWM signal by loading the CCPR1L register and using bits DC1B1 and DC1B0 of the CCP1CON register.
* Configure and start timer TMR2:
  + Clear the TMR2IF interrupt flag bit of the PIR1 register.
  + Set the timer TMR2 prescale value by loading bits T2CKPS1 and T2CKPS0 of the T2CON register.
  + Start the timer TMR2 by setting the TMR2ON bit of the T2CON register.
* Enable PWM output pins after one PWM cycle has been complete:
  + Wait for the timer TMR2 overflow (the TMR2IF bit of the PIR1register is set).
  + Configure the appropriate pin as an output by clearing the bit of the TRIS register.

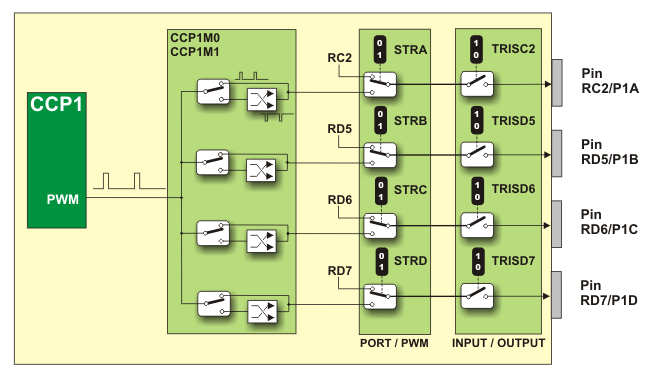
**CCP1 IN ENHANCED MODE**

The enhanced mode is available on CCP1 only. The CCP1 in enhanced mode basically doesn’t differ from the CCP1 in normal mode and enhancement refers to transmission of PWM signal to the output pins. Why is it so important? Because the microcontrollers are more frequently used in electric motor control systems. These devices are not described herein, but if you ever have had a chance to work on development of similar devices, you will recognize elements which, until quite recently, were used as external ones. We say ‘were used’ because all these elements are now integrated into the microcontroller and can operate in several different modes.

**SINGLE OUTPUT PWM MODE**

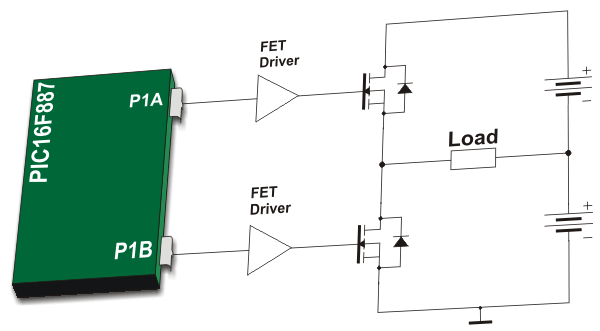
A single output PWM mode is enabled only in the event that the P1M1 and P1M0 bits of the CCP1CON register are cleared. In this case, one PWM signal can be simultaneously available on maximum of four different output pins. Besides, the PWM signal may appear in basic or inverted waveform. Signal distribution depends on the bits of the PSTRCON register, while its polarity depends on the CCP1M1 and CCP1M0 bits of the CCP1CON register.

When an inverted output is in use, pins are low-active and pulses having the same waveform are always generated in pairs: on the P1A and P1C pins and P1B and P1D pins, respectively.

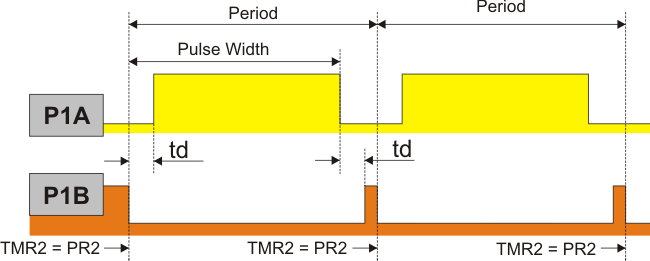


**HALF-BRIDGE MODE**

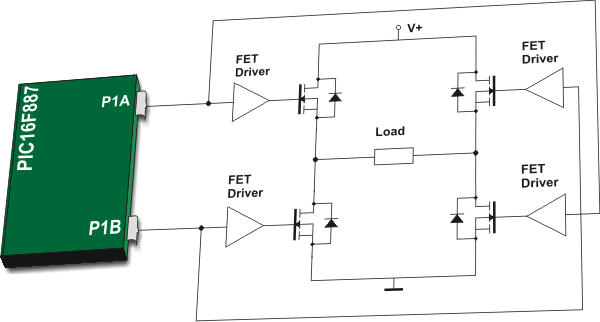
In relation to the half-bridge mode, the PWM signal is the output on the P1A pin, while at the same time the complementary PWM signal is the output on the P1B pin. Such pulses activate MOSFET drivers in *Half-Bridge* mode which enable/disable current flow through the device.



It is very dangerous to switch on MOSFET drivers simultaneously. The short circuit caused in that moment will be fatal. In order to avoid this, it is necessary to provide a short delay between switching drivers on and off. Such delay is marked as 'td' in figure below. The problem is solved by using the PDC0-PDC6 bits of the PWM1CON register.

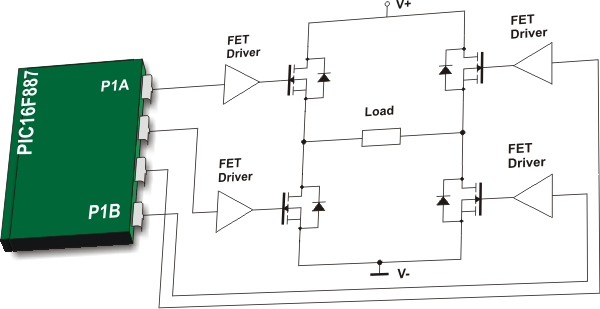


As shown in figure below, the half-bridge mode can also be used to activate MOSFET drivers in the *Full Bridge* configuration:



**FULL-BRIDGE MODE**

All four pins are used as outputs in the full-bridge mode. In practice, this mode is commonly used to run motors, thus providing a simple and full control of speed and rotation direction. There are two configurations of this mode: *Full Bridge-Forward* and *Full Bridge-Reverse*.

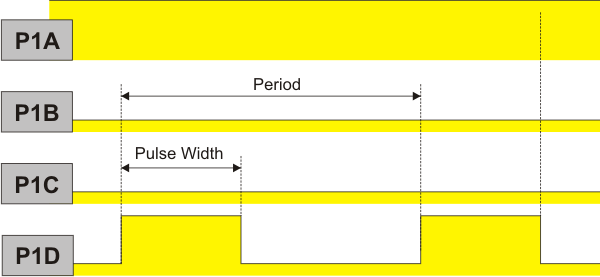


**FULL BRIDGE - FORWARD CONFIGURATION**

In *Forward* mode the following occurs:

* Logic one (1) appears on the P1A pin (pin is active-high);
* Pulse sequence appears on the P1D pin; and
* Logic zero (0) appears on the P1B and P1C pins (pins are active-low).

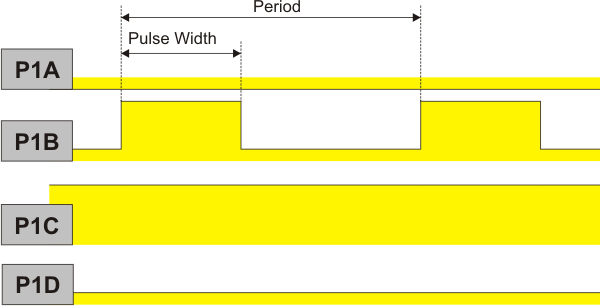
Figure below shows the state of the P1A-P1D pins during one full PWM cycle.



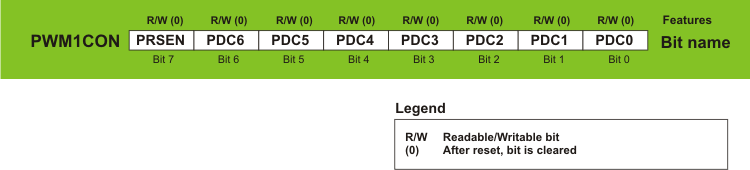
**FULL BRIDGE - REVERSE CONFIGURATION**

The similar occurs in *Reverse* mode, only that these pins have different functions:

* Logic one (1) appears on the P1C pin (pin is active-high);
* Pulse sequence appears on the P1B pin; and
* Logic zero (0) appears on the P1A and P1D pins (pins are active-low).



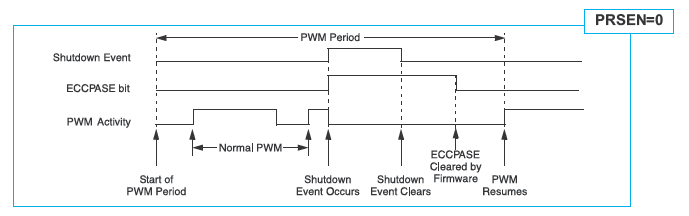
**PWM1CON Register**

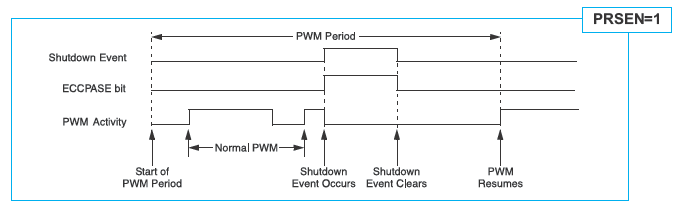


**STRC PWM Restart Enable bit**

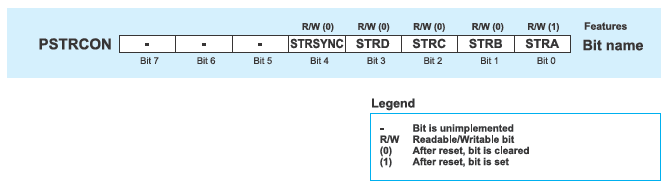
* 1 - Upon auto-shutdown, the PWM module automatically restarts, while the ECCPASE bit of the ECCPAS register is cleared.
* 0 - In order to restart PWM module upon auto-shutdown, the ECCPASE bit must be cleared in software.

**PDC6 - PDC0 PWM Delay Count bits** - 7-digit binary number determines the number of instruction cycles (4\*Tosc) added as a time delay during activation of PWM output pins.





**PSTRCON Register**



**STRSYNC - Steering Sync bit** determines the moment of PWM pulse steering:

* 1 - Steering occurs upon the PSTRCON register has been changed, but only if a PWM waveform is completed.
* 0 - Steering occurs upon the PSTRCON register has been changed. The PWM signal on the output pin is immediately changed with no regard to whether the previous cycle is completed or not. This operation is useful when it is needed to immediately remove a PWM signal from the pin.

**STRD - Steering Enable bit D** determines the P1D pin function.

* 1 - The P1D pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits.
* 0 - Pin is configured as a general port D input/output.

**STRC Steering Enable bit C** determines the P1C pin function.

* 1 - The P1C pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits.
* 0 - Pin is configured as a general port D input/output.

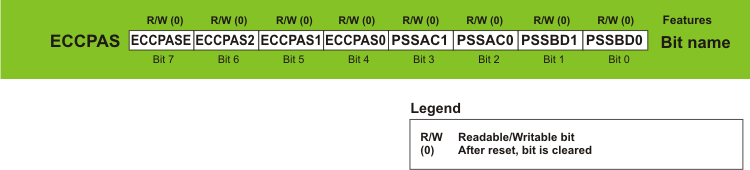
**STRB - Steering Enable bit B** determines the P1B pin function.

* 1 - The P1B pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits.
* 0 - Pin is configured as a general port D input/output.

**STRA - Steering Enable bit A** determines the P1A pin function.

* 1 - The P1D pin has the PWM waveform with polarity controlled by the CCP1M0 and CCP1M1 bits.
* 0 - Pin is configured as a general port C input/output.

**ECCPAS Register**



**ECCPASE - ECCP Auto-Shutdown Event Status bit** indicates whether shut-down of CCP module has occurred (*Shutdown* state):

* 1 - CCP module is in *Shutdown* state.
* 0 - CCP module operates normally.

**ECCPAS2 - ECCPAS0 - ECCP Auto-Shutdown Source Select bits** select auto shutdown source:

|  |  |  |  |
| --- | --- | --- | --- |
| **ECCPAS2** | **ECCPAS1** | **ECCPAS0** | **Shuthown state source** |
| 0 | 0 | 0 | Shutdown state disabled |
| 0 | 0 | 1 | Comparator C1 output change |
| 0 | 1 | 0 | Comparator C2 output change |
| 0 | 1 | 1 | Comparator C1 or C2 output change |
| 1 | 0 | 0 | Logic zero (0) on INT pin |
| 1 | 0 | 1 | Logic zero (0) on INT pin or comparator C1 output change |
| 1 | 1 | 0 | Logic zero (0) on INT pin or comparator C2 output change |
| 1 | 1 | 1 | Logic zero (0) on INT pin or comparator C1 or C2 output change |

**PSSAC1, PSSAC0 - Pins P1A, P1C Shutdown State Control bits** define the logic state of output pins P1A and P1C when CCP module is in shutdown state.

|  |  |  |
| --- | --- | --- |
| **PSSAC1** | **PSSAC0** | **Pins logic state** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | X | High impedance (Tri-state) |

**PSSBD1, PSSBD0 - Pins P1B, P1D Shutdown State Control bits** define the logic state of output pins P1B and P1D when CCP module is in shutdown state.

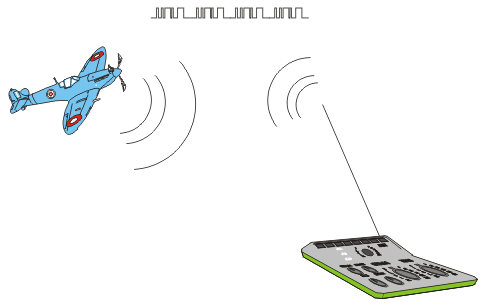
|  |  |  |
| --- | --- | --- |
| **PSSBD1** | **PSSBD0** | **Pins logic state** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | X | High impedance (Tri-state) |

*The PIC16F887 microcontroller has several independent serial communication modules, and each of them can be configured to operate in several different modes, which make them irreplaceable in many situations. Remember what we advised you about the CCP modules as the same applies here. Don’t burden yourself with details of the operation of all of them, but select one and use only what you really need.*

**3.8 SERIAL COMMUNICATION MODULES**

The USART is one of the oldest serial communication systems. The modern versions of this system are upgraded and called somewhat differently - EUSART.

**EUSART**



The *Enhanced Universal Synchronous Asynchronous Receiver Transmitter* (EUSART) *module* is a serial I/O communication peripheral unit. It is also known as *Serial Communications Interface* (SCI). It contains all clock generators, shift registers and data buffers necessary to perform an input/output serial data transfer independently of the device program execution. As its name states, apart from using the clock for synchronization, this module can also establish asynchronous connection, which makes it unique for some of the applications. For example, in the event that it is difficult or impossible to provide special channels for clock and data transfer (for example, radio or infrared remote control), the EUSART module is definitely the best possible solution.

The EUSART system integrated into the PIC16F887 microcontroller has the following features:

* *Full-duplex* asynchronous transmit and receive;
* Programmable 8- or 9-bit wide characters;
* Address detection in 9-bit mode;
* Input buffer overrun error detection; and
* *Half-duplex* communication in synchronous mode.

**EUSART ASYNCHRONOUS MODE**

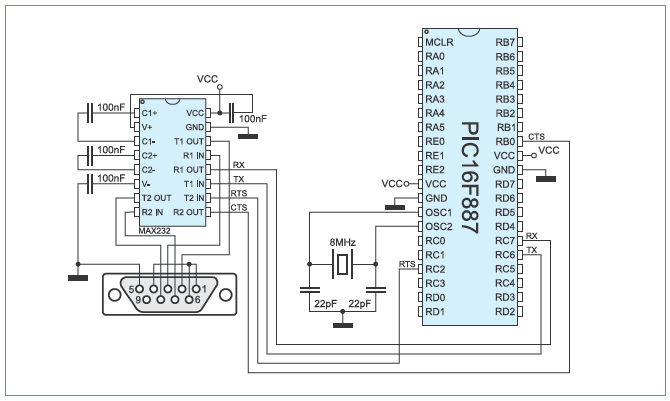
The EUSART transmits and receives data using a standard non-return-to-zero (NRZ) format. As seen in figure below, this mode doesn’t use clock signal, while the format of data being transferred is very simple:



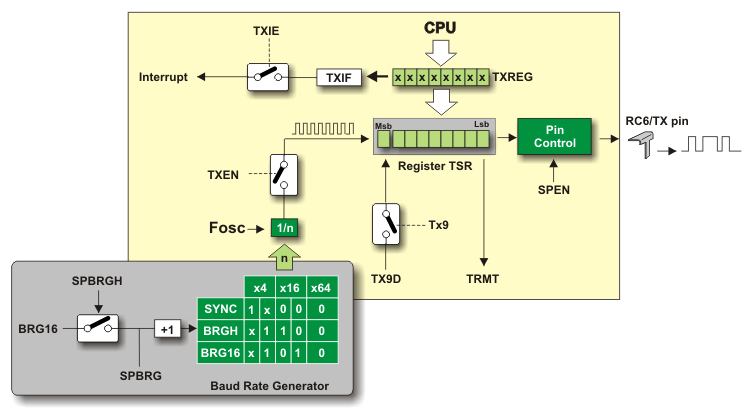
Briefly, each data is transferred in the following way:

* In idle state, data line has high logic level (1);
* Each data transmission starts with the START bit which is always a zero (0);
* Each data is 8- or 9-bit wide (the LSB bit is transferred first); and
* Each data transmission ends with the STOP bit which is always a one (1).

Figure below shows a common way of connecting PIC microcontroller that uses EUSART module. The RS-232 circuit is used as a voltage level converter.



**EUSART ASYNCHRONOUS TRANSMITTER**



In order to enable data transmission via EUSART module, it is necessary to configure it to operate as a transmitter. In other words, it is necessary to define the state of the following bits:

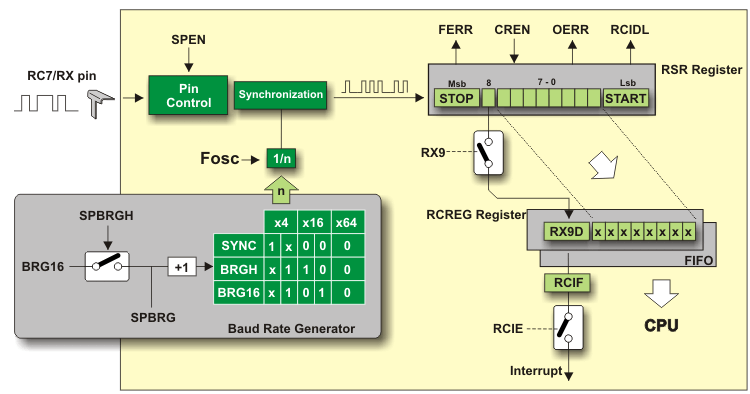
* **TXEN = 1** - EUSART transmitter is enabled by setting the TXEN bit of the TXSTA register.
* **SYNC = 0** - EUSART is configured to operate in asynchronous mode by clearing the SYNC bit of the TXSTA register.
* **SPEN = 1** - By setting the SPEN bit of the RCSTAregister, EUSART is enabled and the TX/CK pin is automatically configured as an output. If this bit is simultaneously used for some analogue function, it must be disabled by clearing the corresponding bit of the ANSEL register.

The central part of the EUSART transmitter is the shift register TSR which is not directly accessible by the user. In order to start data transfer, the module must be enabled by setting the TXEN bit of the TXSTA register. Data to be sent should be written to the TXREG register, which will cause the following sequence of events:

* Byte will be immediately transferred to the shift register TSR;
* TXREG register remains empty, which is indicated by setting the flag bit TXIF of the PIR1 register. If the TXIE bit of the PIE1 register is set, an interrupt will be generated. However, the flag is set regardless of whether an interrupt is enabled or not and it cannot be cleared by software, but by writing new data to the TXREG register.
* Control electronics 'pushes' data toward the TX pin in synchronization with internal clock: START bit (0) ... data ... STOP bit (1).
* When the last bit leaves the TSR register, the TRMT bit of the TXSTA register is automatically set.
* If the TXREG register has received a new character data in the meantime, the whole procedure will be immediately repeated after the STOP bit of the previous character has been transmitted.

9-bit data transfer is enabled by setting the TX9 bit of the TXSTA register. The TX9D bit of the TXSTA register is the ninth and most significant data bit. When transferring 9-bit data, the TX9D data bit must be written prior to writing the 8 least significant bits into the TXREG register. All nine bits of data will be transferred to the TSR shift register immediately after the TXREG write is complete.

**EUSART ASYNCHRONOUS RECEIVER**



In order to enable data transmission via EUSART module, it is necessary to configure it to operate as a transmitter. In other words, it is necessary to define the state of the following bits:

* **CREN = 1** - EUSART receiver is enabled by setting the CREN bit of the RCSTA register;
* **SYNC = 0** - EUSART is configured to operate in asynchronous mode by clearing the SYNC bit stored in the TXSTA register; and
* **SPEN = 1** - By setting the SPEN bit of the RCSTAregister, EUSART is enabled and the RX/DT pin is automatically configured as an input. If this bit is simultaneously used for some analogue function, it must be disabled by clearing the corresponding bit of the ANSEL register.

When this first and necessary step is accomplished and the START bit is detected, data is transferred to the shift register RSR through the RX pin. When the STOP bit has been received, the following occurs:

* Data is automatically transferred to the RCREG register (if empty);
* The flag bit RCIF is set and an interrupt, if enabled by the RCIE bit of the PIE1 register, occurs. Similarly to the transmitter, the flag bit is cleared by software only, i.e. by reading the RCREG register. Bear in mind that this is a two character FIFO memory (*first-in*, *first-out*) which allows reception of two characters simultaneously;
* If the RCREG register is occupied (contains two bytes) and the shift register detects new STOP bit, the overflow bit OERR will be set. In this case, a new coming data is lost, and the OEER bit must be cleared by software. It is done by clearing and resetting the CREN bit;  
  **Note: it is not possible to receive new data as far as the OERR bit is set.**
* If the STOP bit is a zero (0), the FERR bit of the RCSTA register detecting receive error will be set; and
* To enable 9-bit data reception, it is necessary to set the RX9 bit of the RCSTA register.

**RECEIVE ERROR DETECTION**

There are two types of errors which the microcontroller can automatically detect. The first one is called *Framing error* and occurs when the receiver does not detect the STOP bit at the expected time. Such an error is indicated by the FERR bit of the RCSTA register. If this bit is set, the last received data may be incorrect. Here are several things important to know:

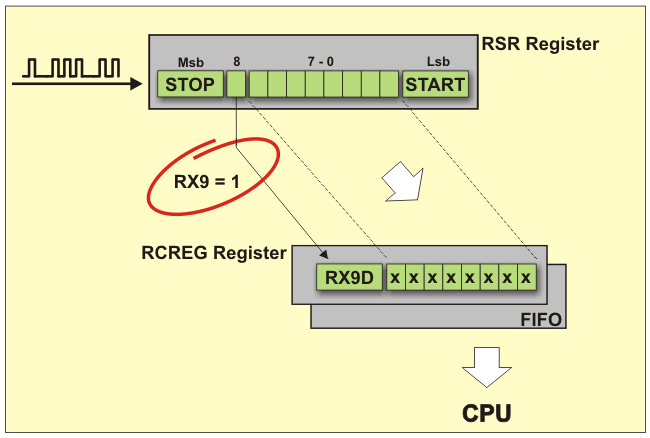
* A *Framing error* does not generate an interrupt by itself;
* If this bit is set, the last received data has an error;
* A framing error (bit set) does not prevent reception of new data;
* The FERR bit is cleared by reading received data, which means that check must be done prior to reading data; and
* The FERR bit cannot be cleared by software. If needed, it can be cleared by clearing the SPEN bit of the RCSTA register. It will simultaneously cause the whole EUSART system to be reset.

Another type of error is called *Overrun Error*. As previously mentioned, the FIFO memory can receive two characters only. An overrun error will be generated if the third character is received. Simply put, there is no space for another one byte and an error is unavoidable. When this happens the OERR bit of the RCSTA register is set. The consequences are the following:

* Data already stored in the FIFO registers (two bytes) can be normally read;
* No additional data will be received until the OERR bit is cleared; and
* This bit is not directly accessed. To clear it, it is necessary to clear the CREN bit of the RCSTAregister or reset the whole EUSART system by clearing the SPEN bit of the RCSTA register.

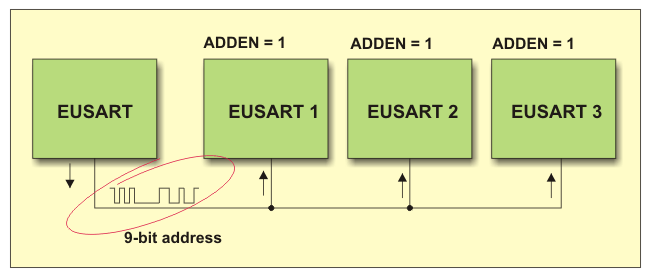
**9-BIT DATA RECEIVE**

Apart from receiving standard 8-bit data, the EUSART system supports 9-bit data reception. On the transmit side, the ninth bit is ‘attached’ to the original byte directly before the STOP bit. On the receive side, when the RX9 bit of the RCSTA register is set, the ninth data bit will be automatically written to the RX9D bit of the same register. After receiving this byte, it is necessary to take care of how to read its bits- the RX9D data bit must be read prior to reading 8 least significant bits of the RCREG register. Otherwise, the ninth data bit will be cleared.

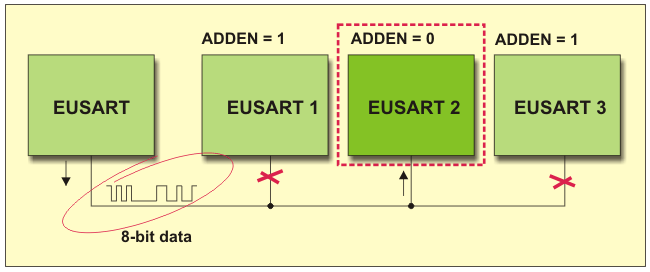


**ADDRESS DETECTION**

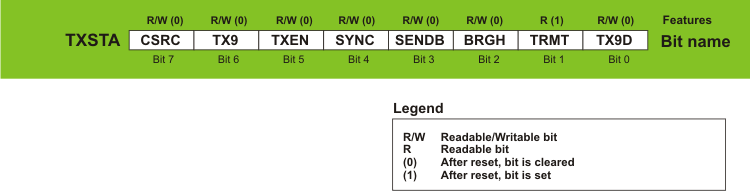
When the ADDEN bit of the RCSTA register is set, the EUSART module is able to receive only 9-bit data, whereas all 8-bit data will be ignored. Although it seems like a restriction, such modes enable serial communication between several microcontrollers. The principle of operation is simple. Master device sends a 9-bit data representing the address of one slave microcontroller. However, all of them must have the ADDEN bit set because it enables address detection. All slave microcontrollers, sharing the same transmission line, receive this data (address) and automatically check whether it matches their own address. Software, in which address match occurs, must disable address detection by clearing its ADDEN bit.



The master device keeps on sending 8-bit data. All data passing through the transmission line will be received by the addressed EUSART module only. When the last byte has been received, the slave device should set the ADDEN bit in order to enable new address detection.



**TXSTA Register**



**CSRC - Clock Source Select bit** - determines clock source. It is used only in synchronous mode.

* 1 - *Master* mode. Clock is generated internally from *Baud Rate* Generator.
* 0 - *Slave* mode. Clock is generated from external source.

**TX9 - 9-bit Transmit Enable bit**

* 1 - 9-bit data transmission via EUSART system.
* 0 - 8-bit data transmission via EUSART system.

**TXEN - Transmit Enable bit**

* 1 - Transmission enabled.
* 0 - Transmission disabled.

**SYNC - EUSART Mode Select bit**

* 1 - EUSART operates in synchronous mode.
* 0 - EUSART operates in asynchronous mode.

**SENDB - Send Break Character bit** is only used in asynchronous mode and when it is required to observe LIN bus standard.

* 1 - *Break* character transmission is enabled.
* 0 - *Break* character transmission is completed.

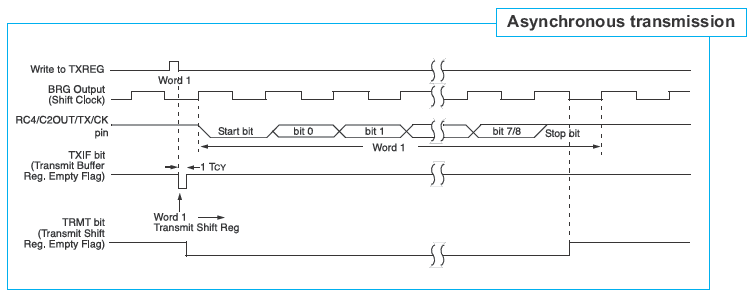
**BRGH - High Baud Rate Select bit** determines baud rate in asynchronous mode. It does not affect EUSART in synchronous mode.

* 1 - EUSART operates at high speed.
* 0 - EUSART operates at low speed.

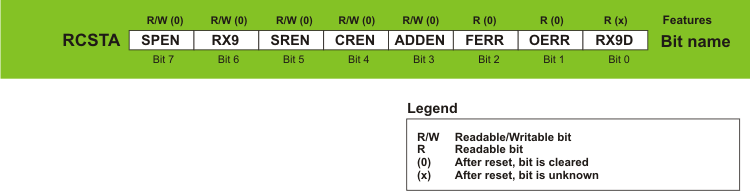
**TRMT - Transmit Shift Register Status bit**

* 1 - TSR register is empty.
* 0 - TSR register is full.

**TX9D - Ninth bit of Transmit Data** can be used as address or parity bit.



**RCSTA Register**



**SPEN - Serial Port Enable bit**

* 1 - Serial port enabled. RX/DT and TX/CK pins are automatically configured as input and output, respectively.
* 0 - Serial port disabled.

**RX9 - 9-bit Receive Enable bit**

* 1 - Reception of 9-bit data via EUSART system.
* 0 - Reception of 8-bit data via EUSART system.

**SREN - Single ReceiveEnable bit** is used only in synchronous mode when the microcontroller operates as *master*.

* 1 - Single receive enabled.
* 0 - Single receive disabled.

**CREN - Continuous Receive Enable bit** acts differently depending on EUSART mode.

Asynchronous mode:

* 1 - Receiver enabled.
* 0 - Receiver disabled.

Synchronous mode:

* 1 - Enables continuous receive until the CREN bit is cleared.
* 0 - Disables continuous receive.

**ADDEN - Address Detect Enable bit** is only used in address detect mode.

* 1 - Enables address detection on 9-bit data receive.
* 0 - Disables address detection. The ninth bit can be used as parity bit.

**FERR - Framing Error bit**

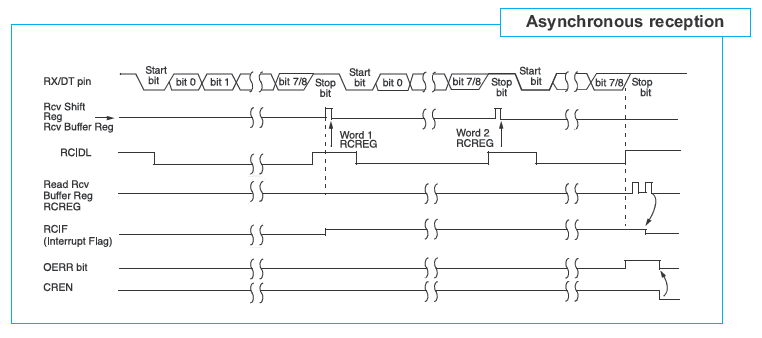
* 1 - On receive, *Framing Error* is detected.
* 0 - No framing error.

**OERR - Overrun Error bit.**

* 1 - On receive, *Overrun Error* is detected.
* 0 - No overrun error.

**RX9D - Ninth bit of Received Data** can be used as address or parity bit.

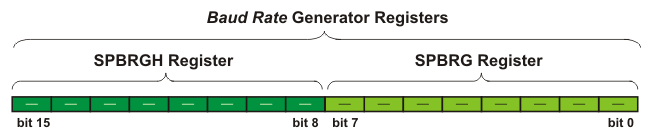
The next diagram shows three words appearing on the RX input. The receiving buffer is read after the third word, causing the OEER bit (overrun error bit) to be set.



**EUSART BAUD RATE GENERATOR (BRG)**

If you carefully look at asynchronous EUSART receiver or transmitter diagram, you will see that both of them use clock signal from the local timer BRG for synchronization. The same clock source is also used in synchronous mode.

The BRG timer consists of two 8-bit registers making one 16-bit register.

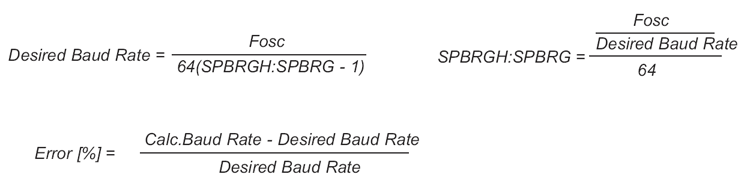


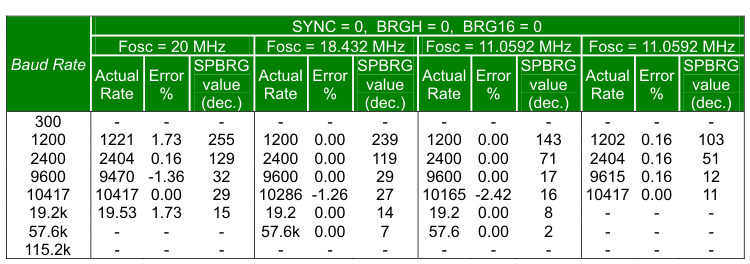
A number written to these two registers determines the baud rate. Besides, both the BRGH bit of the TXSTAregister and the BRGH16 bit of the BAUDCTL register affect clock frequency.

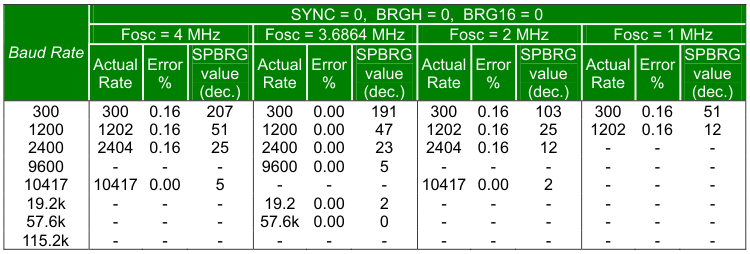
The formula used to determine *Baud Rate* is given in the table below.

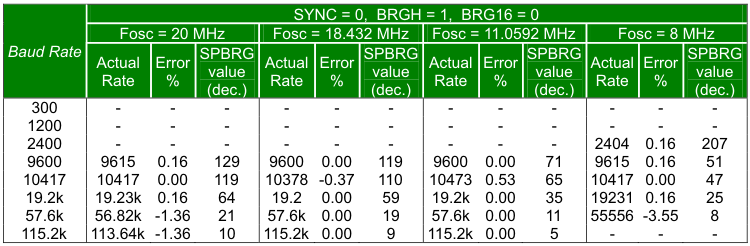
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bits** | | | **BRG / EUSART Mode** | **Baud Rate Formula** |
| **SYNC** | **BRG1G** | **BRGH** |
| 0 | 0 | 0 | 8-bit / asynchronous | Fosc / [64 (n + 1)] |
| 0 | 0 | 1 | 8-bit / asynchronous | Fosc / [16 (n + 1)] |
| 0 | 1 | 0 | 16-bit / asynchronous | Fosc / [16 (n + 1)] |
| 0 | 1 | 1 | 16-bit / asynchronous | Fosc / [4 (n + 1)] |
| 1 | 0 | X | 8-bit / asynchronous | Fosc / [4 (n + 1)] |
| 1 | 1 | X | 16-bit / asynchronous | Fosc / [4 (n + 1)] |

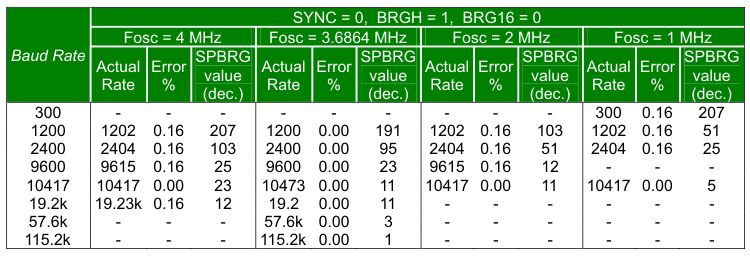
Tables on the following pages contain values that should be written to the 16-bit register SPBRG and assigned to the SYNC, BRGH and BRGH16 bits in order to obtain some of the standard baud rates. Use the following formulas to determine the *Baud Rate*:

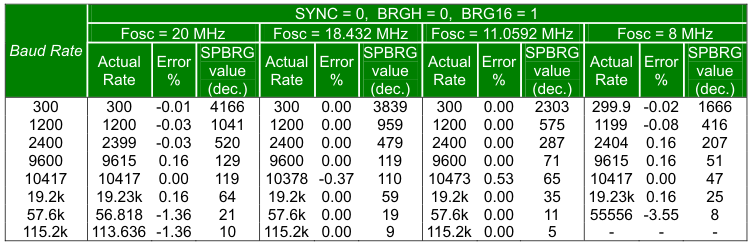


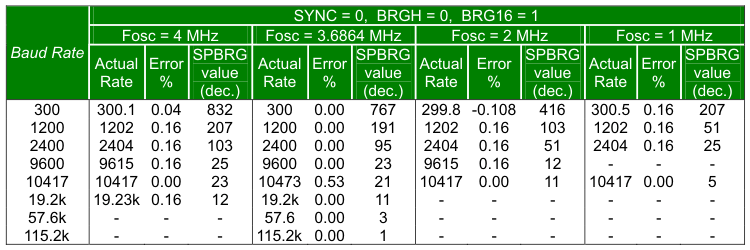


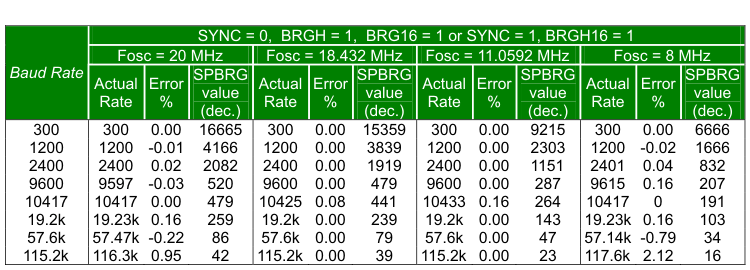


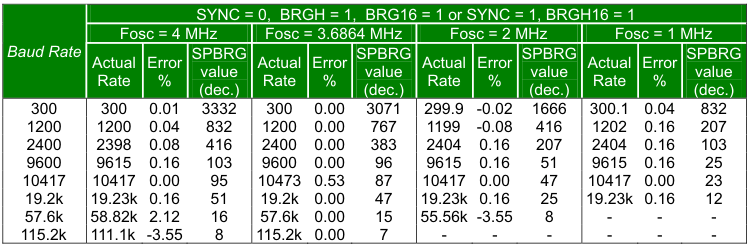




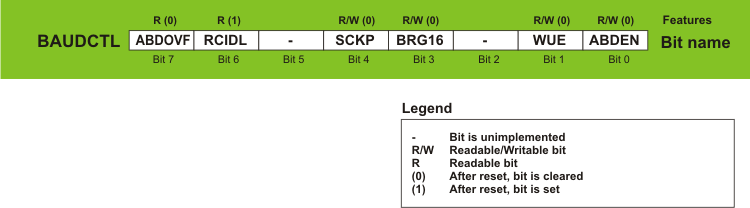








**BAUDCTL Register**



**ABDOVF - Auto-Baud Detect Overflow bit** is only used in asynchronous mode during baud rate detection.

* 1 - Auto-baud timer has overflowed.
* 0 - Auto-baud timer has not overflowed.

**RCIDL - Receive Idle Flag bit** is only used in asynchronous mode.

* 1 - Receiver is idle.
* 0 - START bit has been received and data receive is in progress.

**SCKP - Synchronous Clock Polarity Select bit**. The logic state of this bit varies depending on which EUSART mode is active.

Asynchronous mode:

* 1 - Transmit inverted data to the RC6/TX/CK pin.
* 0 - Transmit non-inverted data to the RC6/TX/CK pin.

Synchronous mode:

* 1 - Synchronization on the clock rising edge.
* 0 - Synchronization on the clock falling edge.

**BRG16 16-bit Baud Rate Generator bit** - determines whether the SPBRGH register will be used, i.e. whether the BRG timer will have 8 or 16 bits.

* 1 - 16-bit baud rate generator is used.
* 0 - 8-bit baud rate generator is used.

**WUE Wake-up Enable bit**

* 1 - Receiver waits for a falling edge on the RC7/RX/DT pin to wake up the microcontroller from sleep mode.
* 0 - Receiver operates normally.

**ABDEN - Auto-Baud Detect Enable bit** is used in asynchronous mode only.

* 1 - Auto-baud detect mode is enabled. Bit is automatically cleared on baud rate detection.
* 0 - Auto-baud detect mode is disabled.

**Let's do it in mikroC...**

*/\* In this example, internal EUSART module is initialized and set to send back the  
message immediately after receiving it. Baude rate is set to 9600 bps. The program  
uses UART library routines UART1\_init(), UART1\_Write\_Text(), UART1\_Data\_Ready(),  
UART1\_Write() and UART1\_Read().\*/*  
  
**char** uart\_rd;  
  
**void** main() {  
 ANSEL = ANSELH = 0; *// Configure AN pins as digital*  
 C1ON\_bit = C2ON\_bit = 0; *// Disable comparators*  
 UART1\_Init(9600); *// Initialize UART module at 9600 bps*  
 Delay\_ms(100); *// Wait for UART module to become stable*  
 UART1\_Write\_Text("Start");  
   
 **while** (1) { *// Endless loop*  
 **if** (UART1\_Data\_Ready()) { *// If data is received,*  
 uart\_rd = UART1\_Read(); *// read the received data,*  
 UART1\_Write(uart\_rd); *// and send data back via UART*  
 }`  
 }  
}

**In Short**

**Data transmission via asynchronous EUSART communication:**

1. The desired baud rate should be set by using bits BRGH (TXSTA register) and BRG16 (BAUDCTL register) and registers SPBRGH and SPBRG.
2. The SYNC bit (TXSTA register) should be cleared and the SPEN bit should be set (RCSTA register) in order to enable serial port.
3. The TX9 bit of the TXSTA register should be set on 9-bit data transmission.
4. Data transmission is enabled by setting the TXEN bit of the TXSTA register. The TXIF bit of the PIR1 register is automatically set.
5. The GIE and PEIE bits of the INTCON register should be set to enable the TXEN bit to cause an interrupt.
6. Value of the ninth bit should be written to the TX9D bit of the TXSTA register on 9-bit data transmission.
7. Transmission starts by writing 8-bit data to the TXREG register.

**Data reception via asynchronous EUSART communication:**

1. Baud Rate should be set by using bits BRGH (TXSTA register) and BRG16 (BAUDCTL register) and registers SPBRGH and SPBRG.
2. The SYNC bit (TXSTA register) should be cleared and the SPEN bit should be set (RCSTA register) in order to enable serial port.
3. Both the RCIE bit of the PIE1 register and bits GIE and PEIE of the INTCON register should be set when it is necessary to enable the data reception to cause an interrupt.
4. The RX9 bit of the RCSTA register should be set on 9-bit data receive.
5. Data reception is enabled by setting the CREN bit of the RCSTA register.
6. The RCSTA register should be read in order to check whether some errors have occurred during transmission. The ninth bit will be stored in this register on 9-bit data reception.
7. The received 8-bit data stored in the RCREG register should be read.

**Setting Address Detection Mode:**

1. Baud Rate should be set by using bits BRGH (TXSTA register) and BRG16 (BAUDCTL register) and registers SPBRGH and SPBRG.
2. The SYNC bit (TXSTA register) should be cleared and the SPEN bit should be set (RCSTA register) in order to enable serial port.
3. The RCIE bit of the PIE1 bit as well as bits GIE and PEIE of the INTCON register should be set when it is necessary to enable the data reception to cause an interrupt.
4. The RX9 bit of the RCSTA register should be set.
5. The ADDEN of the RCSTA register should be set, which enables data to be recognized as address.
6. Data reception should be enabled by setting the CREN bit of the RCSTA register.
7. As soon as the 9-bit data is received, the RCIF bit of the PIR1 register will be automatically set. If enabled, an interrupt occurs.
8. The RCSTA register should be read in order to check whether some errors have occurred during transmission. The ninth bit RX9D is always set.
9. The received 8-bit number stored in the RCREG register should be read. It should be checked whether the combination of these bits matches the predefined address. If the match occurs, it is necessary to clear the ADDEN bit of the RCSTA register, which enables 8-bit data to be received.

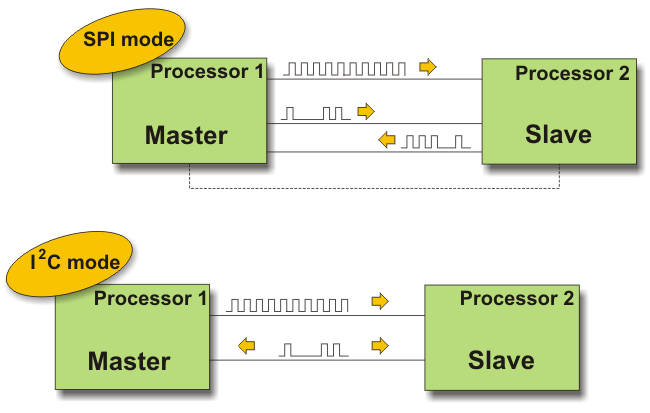
**MASTER SYNCHRONOUS SERIAL PORT MODULE**

MSSP module (*Master Synchronous Serial Port*) is a very useful, but at the same time one of the most complex circuits within the microcontroller. It enables high speed communication between the microcontroller and other peripherals or other microcontrollers by using few input/output lines (maximum two or three). Therefore, it is commonly used to connect the microcontroller to LCD displays, A/D converters, serial EEPROMs, shift registers etc. The main feature of this type of communication is that it is synchronous and suitable for use in systems with a single master and one or more slaves. A master device contains a circuit for baud rate generation and supplies all devices in the system with the clock. Slave devices may in this way eliminate the internal clock generation circuit. The MSSP module can operate in one out of two modes:

* SPI mode (*Serial Peripheral Interface*); and
* I2C mode (*Inter-Integrated Circuit*).

As seen in figure below, one MSSP module represents only a half of the hardware needed to establish serial communication, while the other half is stored in the device it exchanges data with. Even though the modules on both ends of the line are the same, their modes are essentially different depending on whether they operate as a *Master* or a *Slave*:

If the microcontroller to be programmed controls another device or circuit (peripherals), it should operate as a *master* device. It will generate clock when needed, i.e. only when data reception and transmission are required by the software. Obviously, connection establishment depends exclusively on the master device.



Otherwise, if the microcontroller to be programmed is integrated into a more complex device (for example, a PC) then it should operate as a *slave* device. As such, it always has to wait for data transmission request to be sent by the master device.

**SPI MODE**

The SPI mode allows 8 bits of data to be transmitted and received simultaneously using 3 input/output lines:

* **SDO** - *Serial Data Out* - transmit line;
* **SDI** - *Serial Data In* - receive line; and
* **SCK** - *Serial Clock* - synchronization line.

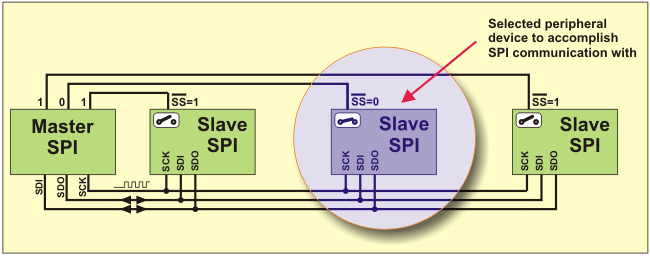
Apart from these three lines, there is the forth line (SS) as well which may be used if the microcontroller exchanges data with several peripheral devices. Refer to figure below.

**SS** - *Slave Select* - is additional pin used for specific device selection. It is active only when the microcontroller is in slave mode, i.e. when the external - master device requires data exchange.

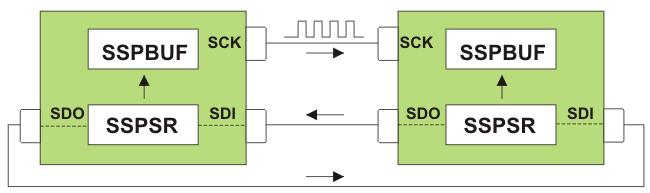
When operating in SPI mode, MSSP module uses in total of 4 registers:

* SSPSTAT - status register
* SSPCON - control register
* SSPBUF - buffer register
* SSPSR - shift register (not directly available)

The first three registers are writable/readable and can be changed at any moment, while the forth register, since not available, is used for converting data into ‘serial’ format.



As seen in figure below, the central part of the SPI module consists of two registers connected to pins for reception, transmission and synchronization.



The *Shift* register (SSPRS) is directly connected to the microcontroller pins and used for data transmission in serial format. The SSPRS register has its input and output so as to shift the data in and out of device. In other words, each bit appearing on the input (receive line) simultaneously shifts another bit toward the output (transmit line).

The SSPBUF register (*Buffer*) is part of memory used to temporarily hold the data prior to being sent or immediately after being received. After all 8 bits of data have been received, the byte is moved from the SSPRS to the SSPBUF register. This double buffering of the received data (SSPBUF) allows the next byte to start reception before reading the data that has just been received. Any write to the SSPBUF register during data transmission/ reception will be ignored. From the programmers’ point of view, this register is considered the most important as being most frequently accessed. Namely, if we neglect mode settings for a moment, data transfer via SPI actually comes to data write and read from this register, while another ‘acrobatics’ such as moving registers are automatically performed by hardware.

**Let's do it in mikroC...**

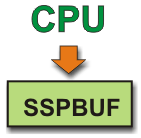
*/\* In this example, PIC microcontroller (master) sends data byte to peripheral chip  
(slave) via SPI. Program uses SPI library functions SPI1\_init() and SPI1\_Write. \*/*  
  
**sbit** Chip\_Select **at** RC0\_bit; *// Peripheral chip\_select pin is connected to RC0*  
**sbit** Chip\_Select\_Direction **at** TRISC0\_bit; *// TRISC0 bit defines RC0 pin to be input or output*  
**unsigned int** value; *// Data to be sent (value) is of unsigned int type*  
  
**void** main() {  
 ANSEL = ANSELH = 0; *// All I/O pins are digital*  
 TRISB0\_bit = TRISB1\_bit = 1; *// Configure RB0, RB1 pins as inputs*  
 Chip\_Select = 0; *// Select peripheral chip*  
 Chip\_Select\_Direction = 0; *// Configure the CS# pin as an output*  
 SPI1\_Init(); *// Initialize SPI module*  
 SPI1\_Write(value); *// Send value to peripheral chip*  
 ...

**In short**

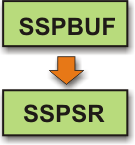
Prior to the SPI initialization, it is necessary to specify several options:

* Master mode TRISC.3=0 (the SCK pin is the clock output);
* Slave mode TRISC.3=1 (the SCK pin is the clock input);
* Data input phase- middle or end of data output time (the SMP bit of the SSPSTAT register);
* Clock edge (the CKE bit of the SSPSTAT register);
* Baud Rate, bits SSPM3-SSPM0 of the SSPCON register (only in Master mode);
* Slave select mode, bits SSPM3-SSPM0 of the SSPCON register (Slave mode only).

The module starts to operate by setting the SSPEN bit:



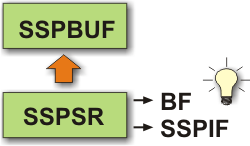
**Step 1.**  
Data to be transmitted should be written to the buffer register SSPBUF. If the SPI module operates in master mode, the microcontroller will automatically perform the following steps 2, 3 and 4. If the SPI module operates as Slave, the microcontroller will not perform these steps until the SCK pin detects clock signal.



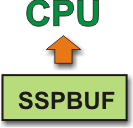
**Step 2.**  
The data is now moved to the SSPSR register and the SSPBUF register is not cleared.



**Step 3.**  
This data is then shifted to the output pin (MSB bit first) while the register is simultaneously being filled with bits through the input pin. In Master mode, the microcontroller itself generates clock, while the Slave mode uses external clock (the SCK pin).



**Step 4.**  
The SSPSR register is full once 8 bits of data have been received. It is indicated by setting the BF bit of the SSPSTAT register and the SSPIF bit of the PIR1 regis-ter. The received data (one byte) is automatically moved from the SSPSR register to the SSPBUF register. Since serial data transmission is performed automatically, the rest of the program is normally executed while the data transmission is in progress. In this case, the function of the SSPIF bit is to generate an interrupt when one byte transmission is completed.



**Step 5.**  
Finally, the data stored in the SSPBUF register is ready for use and should be moved to a desired register.

**I2C MODE**

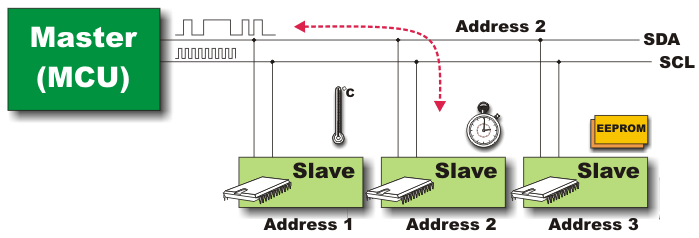
I2C mode (*Inter IC Bus*) is especially suitable when the microcontroller and an integrated circuit, which the microcontroller should exchange data with, are within the same device. It is usually another microcontrollers or specialized, cheap integrated circuits belonging to the new generation of so called 'smart peripheral components' (memories, temperature sensors, real-time clocks etc.)

Similar to serial communication in SPI mode, data transfer in I2C mode is synchronous and bidirectional. This time only two pins are used for data transmission. These are the SDA (Serial Data) and SCL (Serial Clock) pins. The user must configure these pins as inputs or outputs through the TRISC bits.

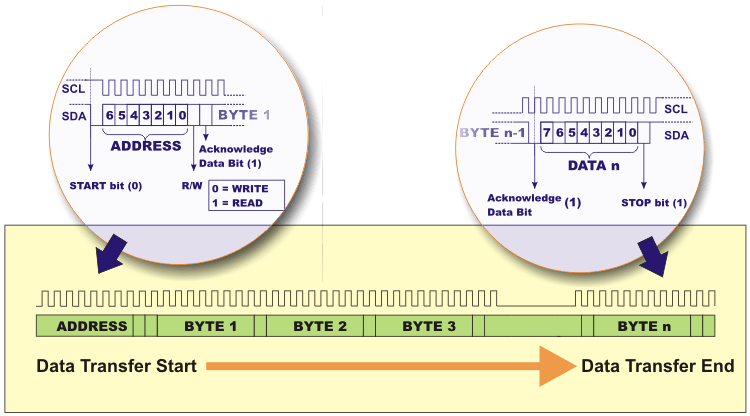
By observing particular rules (protocols), this mode enables up to 122 different components to be simultaneously connected in a simple way by using only two valuable I/O pins. Let’s take a look at how it works:

*Clock*, necessary to synchronize the operation of both devices, is always generated by a master device (a microcontroller) and its frequency directly affects the baud rate. Even though there is a protocol allowing maximum 3,4 MHz clock frequency (so called highspeed I2C bus), this book covers only the most frequently used protocol the clock frequency of which is limited to 100 KHz. Minimum frequency is not limited.

When *master* and *slave* components are synchronized by the clock, every data exchange is always initiated by the *master*. Once the MSSP module has been enabled, it waits for a Start condition to occur. The master device first sends the START bit (logic zero) through the SDA pin, then a 7-bit address of the selected slave device, and finally, the bit which requires data write (0) or read (1) to the device. In other words, the eight bits are shifted to the SSPSR register following the start condition. All *slave* devices sharing the same transmission line will simultaneously receive the first byte, but only one of them has the address to match and receives the whole data.

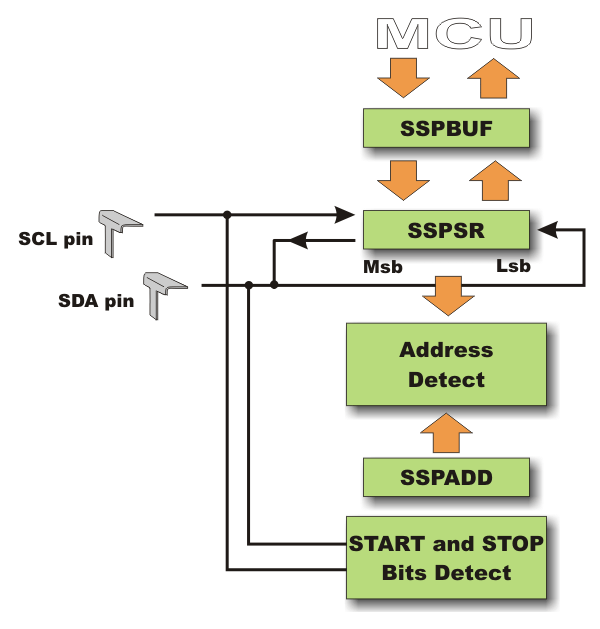


Once the first byte has been sent (only 8-bit data are transmitted), *master* goes into receive mode and waits for acknowledgment from the receive device that address match has occurred. If the *slave* device sends acknowledge data bit (1), data transfer will be continued until the *master* device (microcontroller) sends the Stop bit.



This is the simplest explanation of how two components communicate. Such a microcontroller is also capable of controlling more complicated situations when 1024 different components (10-bit address), shared by several different master devices, are connected. Such devices are rarely used in practice and there is no need to discuss them at greater length.

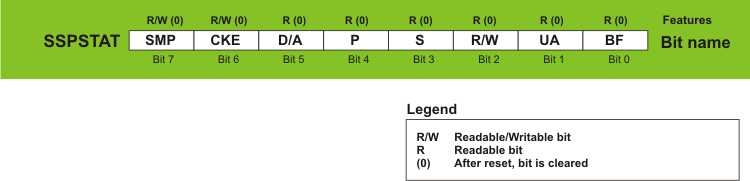
Figure below shows the block diagram of the MSSP module in I2C mode.



The MSSP module uses six registers for I2C operation. Some of them are shown in figure above:

* SSPCON
* SSPCON2
* SSPSTAT
* SSPBUF
* SSPSR
* SSPADD

**SSPSTAT Register**



**SMP Sample bit**

SPI master mode - This bit determines input data phase.

* 1 - Logic state is read at end of data output time.
* 0 - Logic state is read in the middle of data output time.

*SPI slave mode* - This bit must be cleared when SPI is used in Slave mode.

I²C mode (*master* or *slave*)

* 1 - Slew rate control disabled for standard speed mode (100kHz).
* 0 - Slew rate control enabled for high speed mode (400kHz).

**CKE - Clock Edge Select bit** selects synchronization mode.

CKP = 0:

* 1 - Data is transmitted on rising edge of clock pulse (0 - 1).
* 0 - Data is transmitted on falling edge of clock pulse (1 - 0).

CKP = 1:

* 1 - Data is transmitted on falling edge of clock pulse (1 - 0).
* 0 - Data is transmitted on rising edge of clock pulse (0 - 1).

**D/A - Data/Address bit** is used in I2C mode only.

* 1 - Indicates that the last byte received or transmitted was data.
* 0 - Indicates that the last byte received or transmitted was address.

**P - Stop bit** is used in I²C mode only.

* 1 - STOP bit was detected last.
* 0 - STOP bit was not detected last.

**S - Start bit** is used in I²C mode only.

* 1 - START bit was detected last.
* 0 - START bit was not detected last.

**R/W - Read Write bit** is used in I2C mode only. This bit holds the R/W bit information following the last address match. This bit is only valid from the address match to the next Start bit, Stop bit or not ACK bit.

In I²C slave mode

* 1 - Data read.
* 0 - Data write.

In I²C master mode

* 1 - Transmit is in progress.
* 0 - Transmit is not in progress.

**UA - Update Address bit** is used in 10-bit I2C mode only.

* 1 - The SSPADD register must be updated.
* 0 - Address in the SSPADD register is correct and doesn’t need to be updated.

**BF Buffer Full Status bit**

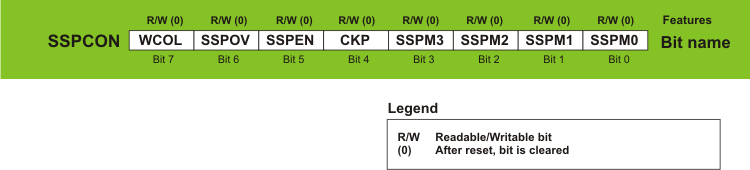
During data receive (in SPI and I²C modes)

* 1 - Receive complete. The SSPBUF register is full.
* 0 - Receive not complete. The SSPBUF register is empty.

During data transmit (in I²C mode only)

* 1 - Data transmit in progress (doesn’t include the ACK and STOP bits).
* 0 - Data transmit complete (doesn’t include the ACK and STOP bits).

**SSPCON Register**



**WCOL Write Collision Detect bit**

* 1 - Collision detected. Write to the SSPBUF register was attempted while the I2C conditions were not valid for transmission to start.
* 0 - No collision.

**SSPOV Receive Overflow Indicator bit**

* 1 - A new byte is received before reading the previously received data. Since there is no space for new data receive, one of these two bytes must be cleared. In this case, data stored in the SSPSR register is irretrievably lost.
* 0 - Serial data is correctly received.

**SSPEN - Synchronous Serial Port Enable bit** determines the microcontroller pins function and initializes MSSP module:

In SPI mode

* 1 - Enables MSSP module and configures pins SCK, SDO, SDI and SS as the source of the serial port pins.
* 0 - Disables MSSP module and configures these pins as I/O port pins.

In I²C mode

* 1 - Enables MSSP module and configures pins SDA and SCL as the source of the serial port pins.
* 0 - Disables MSSP module and configures these pins as I/O port pins.

**CKP - Clock Polarity Select bit** is not used in I²C master mode.

In SPI mode

* 1 - Idle state for clock is a high level.
* 0 - Idle state for clock is a low level.

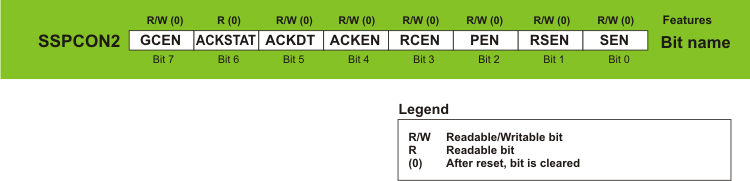
In I²C slave mode

* 1 - Enables clock.
* 0 - Holds clock low. Used to provide more time for data stabilization.

**SSPM3-SSPM0 - Synchronous Serial Port Mode Select bits**. SSP mode is determined by combining these bits:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SSPM3** | **SSPM2** | **SSPM1** | **SSPM0** | **Mode** |
| 0 | 0 | 0 | 0 | SPI master mode, clock = Fosc/4 |
| 0 | 0 | 0 | 1 | SPI master mode, clock = Fosc/16 |
| 0 | 0 | 1 | 0 | SPI master mode, clock = Fosc/64 |
| 0 | 0 | 1 | 1 | SPI master mode, clock = (output TMR)/2 |
| 0 | 1 | 0 | 0 | SPI slave mode, SS pin control enabled |
| 0 | 1 | 0 | 1 | SPI slave mode, SS pin control disabled, SS can be used as I/O pin |
| 0 | 1 | 1 | 0 | I²C slave mode, 7-bit address used |
| 0 | 1 | 1 | 1 | I²C slave mode, 10-bit address used |
| 1 | 0 | 0 | 0 | I²C master mode, clock = Fosc / [4(SSPAD+1)] |
| 1 | 0 | 0 | 1 | Mask used in I²C slave mode |
| 1 | 0 | 1 | 0 | Not used |
| 1 | 0 | 1 | 1 | I²C controlled master mode |
| 1 | 1 | 0 | 0 | Not used |
| 1 | 1 | 0 | 1 | Not used |
| 1 | 1 | 1 | 0 | I²C slave mode, 7-bit address used,START and STOP bits enable interrupt |
| 1 | 1 | 1 | 1 | I²C slave mode, 10-bit address used,START and STOP bits enable interrupt |

**SSPCON2 Register**



**GCEN - General Call Enable bit**

In I²C slave mode only

* 1 - Enables interrupt when a general call address (0000h) is received in the SSPSR.
* 0 - General call address disabled.

**ACKSTAT - Acknowledge Status bit**

In I²C Master Transmit mode only

* 1 - Acknowledge was not received from slave.
* 0 - Acknowledge was received from slave.

**ACKDT - Acknowledge data bit**

In I²C Master Receive mode only

* 1 - Not Acknowledge.
* 0 - Acknowledge.

**ACKEN - Acknowledge Sequence Enable bit**

In I²C Master Receive mode

* 1 - Initiate acknowledge condition on the SDA and SCL pins and transmit the ACKDT data bit. It is automatically cleared by hardware.
* 0 - Acknowledge condition is not initiated.

**RCEN - Receive Enable bit**

In I²C Master mode only

* 1 - Enables data receive in I2C mode.
* 0 - Receive disabled.

**PEN - STOP condition Enable bit**

In I²C Master mode only

* 1 - Initiates STOP condition on the SDA and SCL pins. Afterwards, this bit is automatically cleared by hardware.
* 0 - STOP condition is not initiated.

**RSEN - Repeated START Condition Enabled bit**

In I²C master mode only

* 1 - Initiates START condition on the SDA and SCL pins. Afterwards, this bit is automatically cleared by hardware.
* 0 - Repeated START condition is not initiated.

**SEN - START Condition Enabled/Stretch Enabled bit**

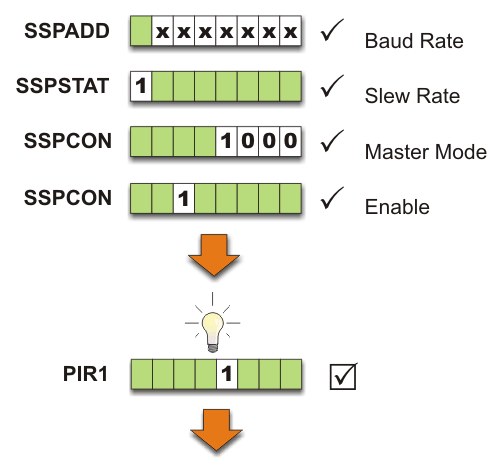
In I²C Master mode only

* 1 - Initiates START condition on the SDA and SCL pins. Afterwards, this bit is automatically cleared by hardware.
* 0 - START condition is not initiated.

**I2C in Master Mode**

The most common case is that the microcontroller operates as a *master* and a peripheral component as a *slave*. This is why this book covers just this mode. It is also considered that the address consists of 7 bits and device contains only one microcontroller (*single-master* device).

In order to enable MSSP module in this mode, it is necessary to do the following:

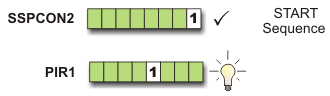


Set baud rate (SSPADD register), turn off *slew rate* control (by setting the SMP bit of the SSPSTAT register) and select *master* mode (SSPCON register). After all these preparations have been finished and the module has been enabled (SSPCON register : SSPEN bit), it is necessary to wait for internal electronics to signal that everything is ready for data transmission, i.e. the SSPIF bit of the PIR1 register is set.

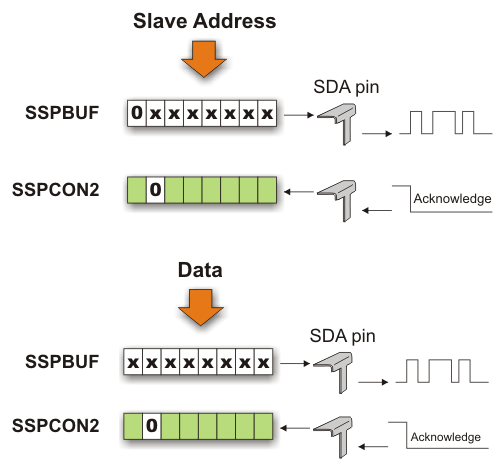
This bit should be cleared by software and after that the microcontroller is ready to exchange data with peripherals.

**Data Transmission in I2C Master Mode**

Data transmission on the SDA pin starts with a logic zero (0) which appears upon setting the SEN bit of the SSPCON2 register. Even enabled, the microcontroller has to wait a certain time before it starts communication. It is the so called 'Start condition' during which internal preparations and checks are performed. If all conditions are met, the SSPIF bit of the PIR1 is set and data transmission starts as soon as the SSPBUF register is loaded.



Maximum 112 integrated circuits (*slave* devices) may simultaneously share the same transmission line. The first data byte sent by the *master* device contains the address to match only one slave device. All addresses are listed in respective data sheets. The eighth bit of the first data byte specifies direction of data transmission, i.e. whether the microcontroller is to send or receive data. In this case, the eighth bit is cleared to logic zero (0), which means that it is data transmission.



When address match occurs, the microcontroller has to wait for the acknowledge data bit. The slave device acknowledges address match by clearing the ASKSTAT bit of the SSPCON2 register. If the match properly occurred, all data bytes are transmitted in the same way.

Data transmission ends by setting the SEN bit of the SSPCON2 register. The STOP condition occurs, which enables the SDA pin to receive pulse condition:

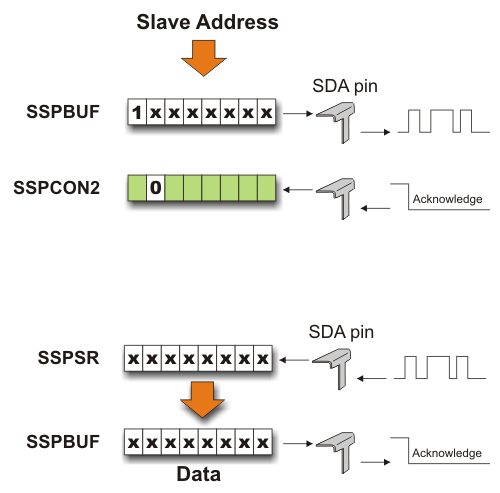
Start - Address - Acknowledge - Data - Acknowledge....Data - Acknowledge - Stop!

**Data Reception in I2C Master Mode**

Preparations for data reception are similar to those for data transmission, with exception that the last bit of the first sent byte (containing address) is set to logic one (1). It specifies that *master* expects to receive data from the addressed slave device. In relation to the microcontroller, the following occurs:

After internal preparations are finished and the START bit is set, *slave* device starts sending one byte at a time. These bytes are stored in the serial register SSPSR. Each data is, after receiving the last eighth bit, loaded to the SSPBUF register from where it can be read. Reading this register causes the acknowledge bit to be automatically sent, which means that the master device is ready to receive new data.

Likewise, data reception ends by setting the STOP bit:

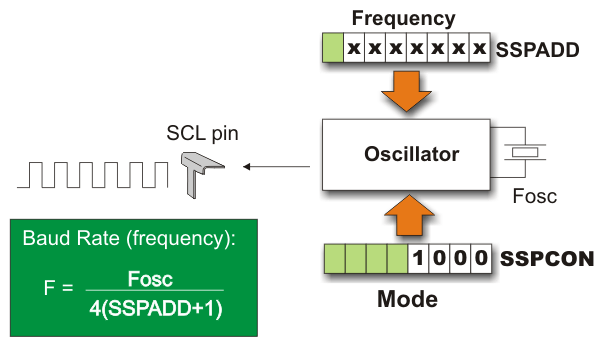


Start - Address - Acknowledge - Data - Acknowledge....Data - Acknowledge - Stop!

In this pulse sequence, the acknowledge bit is sent to *slave* device.

In order to synchronize data transmission, all events taking place on the SDA pin must be synchronized with the clock generated in the master device. This clock is generated by a simple oscillator the frequency of which depends on the microcontroller’s main oscillator frequency, the value written to the SSPADD register and the current SPI mode as well.

The clock frequency of the mode described in this book depends on selected quartz crystal and the SPADD register. Figure below shows the formula used to calculate it.



**Let's do it in mikroC...**

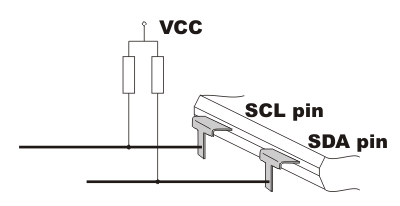
*/\* In this example, PIC MCU is connected to 24C02 EEPROM via SCL and SDA pins. The program  
sends one byte of data to the EEPROM address 2. Then, it reads that data via I2C from  
EEPROM and sends it to PORTB in order to check if the data was successfully written. \*/*  
  
**void** main(){  
 ANSEL = ANSELH = PORTB = TRISB = 0; *// All pins are digital. PORTB pins are outputs.*  
 I2C1\_Init(100000); *// Initialize I2C with desired clock*  
 I2C1\_Start(); *// I2C start signal*  
 I2C1\_Wr(0xA2); *// Send byte via I2C (device address + W)*  
 I2C1\_Wr(2); *// Send byte (address of EEPROM location)*  
 I2C1\_Wr(0xF0); *// Send data to be written*  
 I2C1\_Stop(); *// I2C stop signal*  
   
 Delay\_100ms();  
   
 I2C1\_Start(); *// I2C start signal*  
 I2C1\_Wr(0xA2); *// Send byte via I2C (device address + W)*  
 I2C1\_Wr(2); *// Send byte (data address)*  
 I2C1\_Repeated\_Start(); *// Issue I2C signal repeated start*  
 I2C1\_Wr(0xA3); *// Send byte (device address + R)*  
 PORTB = I2C1\_Rd(0u); *// Read the data (NO acknowledge)*  
 I2C1\_Stop(); *// I2C stop signal*  
}

**USEFUL NOTES ...**

When the microcontroller communicates with peripheral components, it may happen that data transmission fails for some reason. In that case, it is recommended to check the state of some of the bits which can clarify the problem. In practice, the status of these bits is checked by executing a short subroutine after each byte transmission and reception (just in case).

**WCOL (SPCON,7)** - If you try to write a new data to the SSPBUF register while another data transmission/reception is in progress, the WCOL bit will be set and the contents of the SSPBUF register remains unchanged. Write does not occur. After this, the WCOL bit must be cleared in software.

**BF (SSPSTAT,0)** - In transmission mode, this bit is set when the CPU writes to the SSPBUF register and remains set until the byte in serial format is shifted from the SSPSR register. In reception mode, this bit is set when data or address is loaded to the SSPBUF register. It is cleared after reading the SSPBUF register.



**SSPOV (SSPCON,6)** - In reception mode, this bit is set when a new byte is received by the SSPSR register via serial communication, whereas the previously received data has not been read from the SSPBUF register yet.

**SDA and SCL Pins** - When SPP module is enabled, these pins turn into *Open Drain* outputs. It means that they must be connected to the resistors which are by the other end connected to positive power supply.

**In short**

In order to establish serial communication in I2C mode, the following should be done:

**Setting Module and Sending Address:**

* Value to determine baud rate should be written to the SSPADD register.
* SlewRate control should be turned off by setting the SMP bit of the SSPSTAT register.
* In order to select Master mode, binary value 1000 should be written to the SSPM3-SSPM0 bits of the SSPCON1 register.
* The SEN bit of the SSPCON2 register (START sequence) should be set.
* The SSPIF bit is automatically set at the end of START sequence when the module is ready to operate. It should be cleared.
* Slave address should be written to the SSPBUF register.
* When the byte is sent, the SSPIF bit (interrupt) is automatically set after receiving the acknowledge bit from the Slave device.

**Data Transmit:**

* Data to be send should be written to the SSPBUF register.
* When the byte is sent, the SSPIF bit (interrupt) is automatically set after receiving the acknowledge bit from Slave device.
* In order to inform the Slave device that data transmission is complete, STOP condition should be initiated by setting the PEN bit of the SSPCON register.

**Data Receive:**

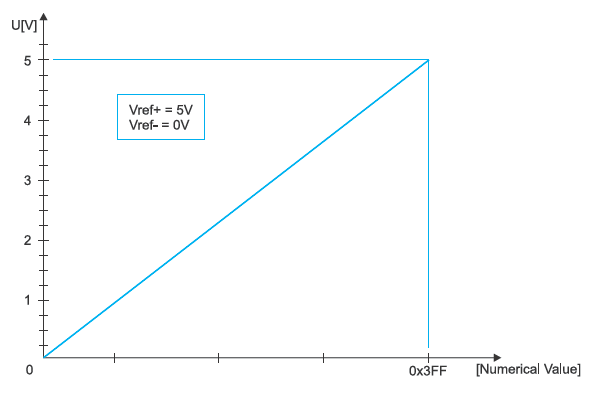
* In order to enable reception, the RSEN bit of the SSPCON2 register should be set.
* The SSPIF bit signals data reception. When data is read from the SSPBUF register, the ACKEN bit of the SSPCON2 register should be set in order to enable acknowledge bit to be sent.
* In order to inform the Slave device that data transmission is complete, the STOP condition should be initiated by setting the PEN bit of the SSPCON register.

*In addition to a large number of digital I/O lines used for communication with peripherals, the PIC16F887 contains 14 analog inputs. They enable the microcontroller to recognize not only whether a pin is driven to logic zero or one (0 or +5V), but to precisely measure its voltage and convert it into numerical value, i.e. digital format.*

**3.9 ANALOG MODULES**

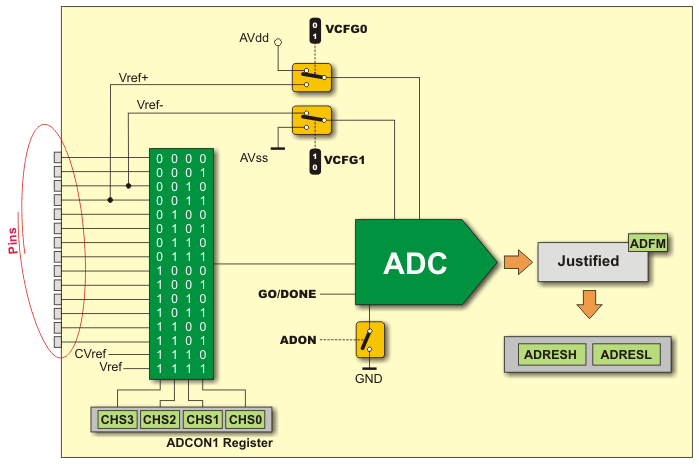
The A/D converter module has the following features:

* The converter generates a 10-bit binary result using the method of successive approximation and stores the conversion results into the ADC registers (ADRESL and ADRESH);
* There are 14 separate analog inputs;
* The A/D converter converts an analog input signal into a 10-bit binary number;
* The minimum resolution or quality of conversion may be adjusted to various needs by selecting voltage references Vref- and Vref+.



**A/D CONVERTER**

Even though the use of A/D converter seems to be very complicated, it is basically very simple, simpler than using timers and serial communication module, anyway.

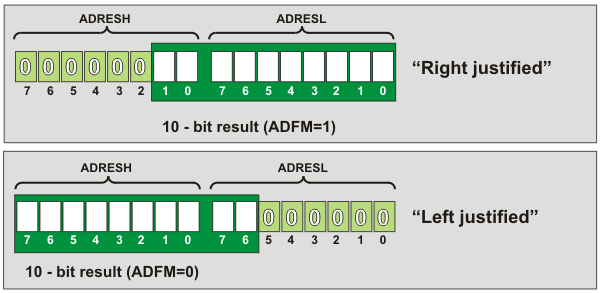


The operation of A/D converter is in control of the bits of four registers:

* ADRESH Contains high byte of conversion result;
* ADRESL Contains low byte of conversion result;
* ADCON0 Control register 0; and
* ADCON1 Control register 1.

**ADRESH and ADRESL Registers**

The result obtained after converting an analog value into digital is a10-bit number that is to be stored in the ADRESH and ADRESL registers. There are two ways of handling it - left and right justification which simplifies its use to a great extent. The format of conversion result depends on the ADFM bit of the ADCON1 register. In the event that the A/D converter is not used, these registers may be used as general-purpose registers.



**A/D ACQUISITION REQUIREMENTS**

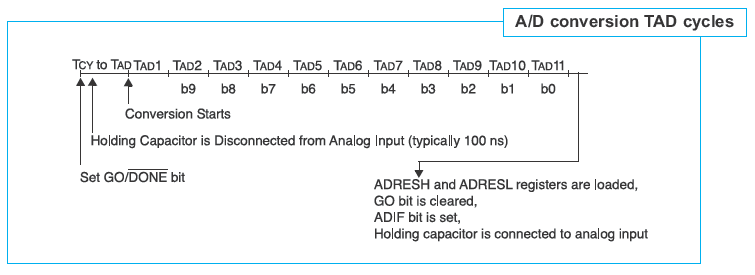
In order to enable the ADC to meet its specified accuracy, it is necessary to provide a certain time delay between selecting specific analog input and measurement itself. This time is called 'acquisition time' and mainly depends on the source impedance. There is an equation used to calculate this time accurately, which in the worst case amounts to approximately 20uS. So, if you want the conversion to be accurate, don’t forget this important detail.

**ADC CLOCK PERIOD**

The time needed to complete a one-bit conversion is defined as TAD. It is required to be at least 1,6 uS. One full 10-bit A/D conversion is slightly longer than expected and amounts to 11 TAD periods. Since both clock frequency and source of A/D conversion are specified by software, it is necessary to select one of the available combinations of bits ADCS1 and ADCS0 before the voltage measurement on some of the analog inputs starts. These bits are stored in the ADCON0 register.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ADC Clock Source** | **ADCS1** | **ADCS0** | **Device Frequency (Fosc)** | | | |
| **20 Mhz** | **8 Mhz** | **4 Mhz** | **1 Mhz** |
| Fosc/2 | 0 | 0 | 100 nS | 250 nS | 500 nS | 2 uS |
| Fosc/8 | 0 | 1 | 400 nS | 1 uS | 2 uS | 8 uS |
| Fosc/32 | 1 | 0 | 1.6 uS | 4 uS | 8 uS | 32 uS |
| Frc | 1 | 1 | 2 - 6 uS | 2 - 6 uS | 2 - 6 uS | 2 - 6 uS |

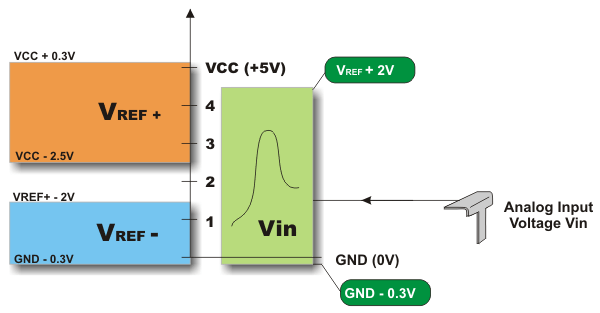
Any change in the system clock frequency will affect the ADC clock frequency, which may adversely affect the ADC result. Device frequency characteristics are shown in the table above. The values in the shaded cells are outside of the range recommended.



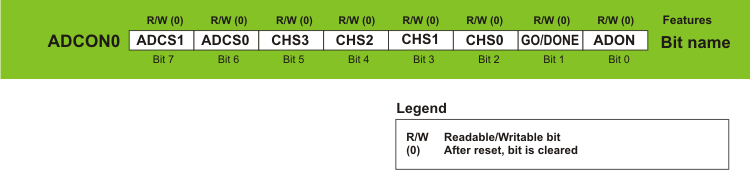
**HOW TO USE THE A/D CONVERTER?**

In order to enable the A/D converter to run without problems as well as to avoid unexpected results, it is necessary to consider the following:

* A/D converter does not differ between digital and analog signals. In order to avoid errors in measurement or chip damage, pins should be configured as analog inputs before the process of conversion starts. Bits used for this purpose are stored in the TRIS and ANSEL (ANSELH) registers;
* When reading the port with analog inputs, the state of the corresponding bits will be read as a logic zero (0); and
* Roughly speaking, voltage measurement in the converter is based on comparing input voltage with internal scale which has 1024 marks (210 = 1024). The lowest scale mark stands for the Vref- voltage, whilst its highest mark stands for the Vref+ voltage. Figure below shows selectable voltage references as well as their minimum and maximum values.



**ADCON0 Register**



**ADCS1, ADCS0 - A/D Conversion Clock Select bits** select clock frequency used for internal synchronization of A/D converter. It also affects duration of conversion.

|  |  |  |
| --- | --- | --- |
| **ADCS1** | **ADCS2** | **Clock** |
| 0 | 0 | Fosc/2 |
| 0 | 1 | Fosc/8 |
| 1 | 0 | Fosc/32 |
| 1 | 1 | RC \* |

\* Clock is generated by internal oscillator which is built in the converter.

**CHS3-CHS0 - Analog Channel Select bits** select a pin or an analog channel for A/D conversion, i.e. voltage measurement:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CHS3** | **CHS2** | **CHS1** | **CHS0** | **Channel** | **Pin** |
| 0 | 0 | 0 | 0 | 0 | RA0/AN0 |
| 0 | 0 | 0 | 1 | 1 | RA1/AN1 |
| 0 | 0 | 1 | 0 | 2 | RA2/AN2 |
| 0 | 0 | 1 | 1 | 3 | RA3/AN3 |
| 0 | 1 | 0 | 0 | 4 | RA5/AN4 |
| 0 | 1 | 0 | 1 | 5 | RE0/AN5 |
| 0 | 1 | 1 | 0 | 6 | RE1/AN6 |
| 0 | 1 | 1 | 1 | 7 | RE2/AN7 |
| 1 | 0 | 0 | 0 | 8 | RB2/AN8 |
| 1 | 0 | 0 | 1 | 9 | RB3/AN9 |
| 1 | 0 | 1 | 0 | 10 | RB1/AN10 |
| 1 | 0 | 1 | 1 | 11 | RB4/AN11 |
| 1 | 1 | 0 | 0 | 12 | RB0/AN12 |
| 1 | 1 | 0 | 1 | 13 | RB5/AN13 |
| 1 | 1 | 1 | 0 | CVref | | |
| 1 | 1 | 1 | 1 | Vref = 0.6V | | |

**GO/DONE - A/D Conversion Status bit** determines current status of conversion:

* 1 - A/D conversion is in progress.
* 0 - A/D conversion is complete. This bit is automatically cleared by hardware when the A/D conversion is complete.

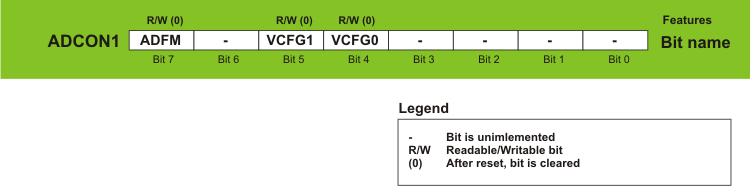
**ADON - A/D On bit** enables A/D converter.

* 1 - A/D converter is enabled.
* 0 - A/D converter is disabled.

**Let's do it in mikroC...**

*/\* This example code reads analog value from channel 2 and displays it on PORTB and  
PORTC as 10-bit binary number.\*/*  
  
#include <built\_in.h>  
**unsigned int** adc\_rd;  
  
**void** main() {  
 ANSEL = 0x04; *// Configure AN2 as analog pin*  
 TRISA = 0xFF; *// PORTA is configured as input*  
 ANSELH = 0; *// Configure all other AN pins as digital I/O*  
 TRISC = 0x3F; *// Pins RC7 and RC6 are configured as outputs*  
 TRISB = 0; *// PORTB is configured as an output*  
   
 **do** {  
 temp\_res = ADC\_Read(2); *// Get 10-bit result of AD conversion*  
 PORTB = temp\_res; *// Send lower 8 bits to PORTB*  
 PORTC = temp\_res >> 2; *// Send 2 most significant bits to RC7, RC6*  
 } **while**(1); *// Remain in the loop*  
}

**ADCON1 Register**



**ADFM - A/D Result Format Select bit**

* 1 - Conversion result is right justified. Six most significant bits of the ADRESH are not used.
* 0 - Conversion result is left justified. Six least significant bits of the ADRESL are not used.

**VCFG1 - Voltage Reference bit** selects negative voltage reference source needed for the operation of A/D converter.

* 1 - Negative voltage reference is applied to the Vref- pin.
* 0 - Power supply voltage Vss is used as negative voltage reference source.

**VCFG0 - Voltage Reference bit** selects positive voltage reference source needed for the operation of A/D converter.

* 1 - Positive voltage reference is applied to the Vref+ pin.
* 0 - Power supply voltage Vdd is used as positive voltage reference source.

**In Short**

In order to measure voltage on an input pin by the A/D converter, the following should be done:

**Step 1** - Port configuration:

* Write a logic one (1) to a bit of the TRIS register, thus configuring the appropriate pin as an input.
* Write a logic one (1) to a bit of the ANSEL register, thus configuring the appropriate pin as an analog input.

**Step 2** - ADC module configuration:

* Configure voltage reference in the ADCON1 register.
* Select ADC conversion clock in the ADCON0 register.
* Select one of input channels CH0-CH13 of the ADCON0 register.
* Select data format using the ADFM bit of the ADCON1 register.
* Enable A/D converter by setting the ADON bit of the ADCON0 register.

**Step 3** - ADC interrupt configuration (optionally):

* Clear the ADIF bit.
* Set the ADIE, PEIE and GIE bits.

**Step 4** - Wait for the required acquisition time to pass (approximately 20uS).

**Step 5** - Start conversion by setting the GO/DONE bit of the ADCON0 register.

**Step 6** - Wait for ADC conversion to complete.

* It is necessary to check in the program loop whether the GO/DONE pin is cleared or wait for an A/D interrupt (must be previously enabled).

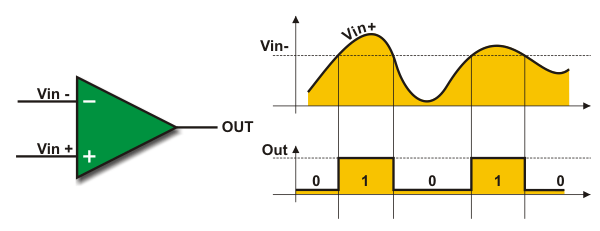
**Step 7** - Read ADC results:

* Read the ADRESH and ADRESL registers.

**ANALOG COMPARATOR**

In addition to A/D converter, there is another module, which until quite recently has been embedded only in integrated circuits belonging to the so called analog electronics. Owing to the fact that it is hardly possible to find any more complex automatic device which in some way does not use these circuits, two high quality comparators, along with additional electronics, are integrated into the microcontroller and connected to its pins.

How does a comparator operate? Basically, the analog comparator is an amplifier which compares the magnitude of voltages at two inputs. It has two inputs and one output. Depending on which input has a higher voltage (analog value), a logic zero (0) or logic one (1) (digital values) will appear on its output:



* When the analog voltage at Vin- is higher than that at Vin+, the output of the comparator is a digital low level.
* When the analog voltage at Vin+ is higher than that at Vin-, the output of the comparator is a digital high level.

The PIC16F887 microcontroller has two such voltage comparators the inputs of which are connected to I/O pins RA0-RA3, whereas the outputs are connected to the RA4 and RA5 pins. There is also a voltage reference internal source on the chip itself, which will be discussed later.

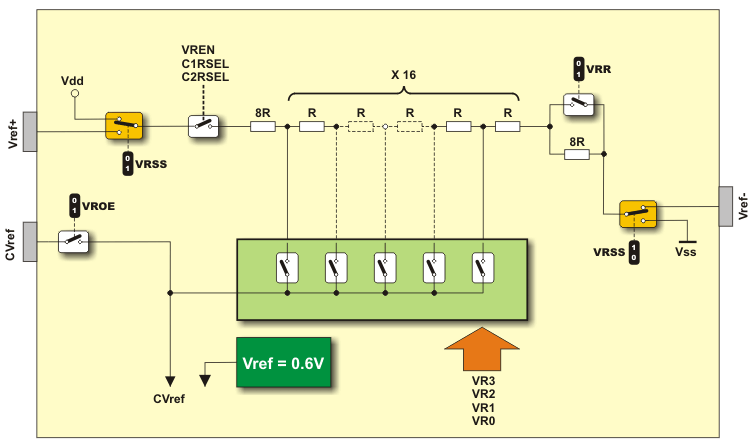
These two circuits are under control of the bits stored in the following registers:

* CM1CON0 is in control of comparator C1;
* CM2CON0 is in control of comparator C2;
* CM2CON1 is in control of comparator C2;

**VOLTAGE REFERENCE INTERNAL SOURCE**

One of two analog voltages provided on the comparator inputs is usually stable and unchangeable. It is called *'voltage reference'*(Vref). To generate it, both external and special internal voltage source can be used. When the voltage source is selected, Vref is derived from it by means of a ladder network consisting of 16 resistors which form a voltage divider. The voltage source is selectable through the both ends of the divider by the VRSS bit of the VRCON register.

In addition, the voltage fraction provided by the resistor ladder network may be selected through the bits VR0-VR3 and used as a voltage reference. See figure below.



The comparator voltage reference has 2 ranges each containing 16 voltage levels. Range selection is controlled by the VRR bit of the VRCON register. The selected voltage reference CVref may be output to the RA2/AN2 pin.

Even though the main idea was to obtain varying voltage reference for the operation of analog modules, a simple A/D converter is obtained thereby as well. This converter is very useful in some situations. Its operation is under control of the VRCON register.

**COMPARATORS AND INTERRUPT**

Every change of the logic state of any comparator’s output causes the flag bit CMIF of the register PIR to be set. Such changes will also cause an interrupt if the following bits are set:

* The CMIE bit of the PIE register = 1;
* The PEIE bit of the INTCON register = 1; and
* The GIE bit of the INTCON register = 1.

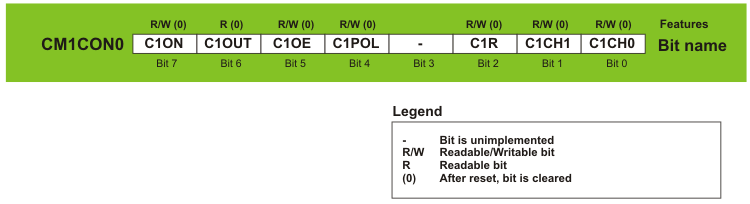
If an interrupt is enabled, any change on the comparator’s output when the microcontroller is set in *Sleep* mode can cause the microcontroller to exit that mode and proceed with normal operation.

**OPERATION DURING SLEEP**

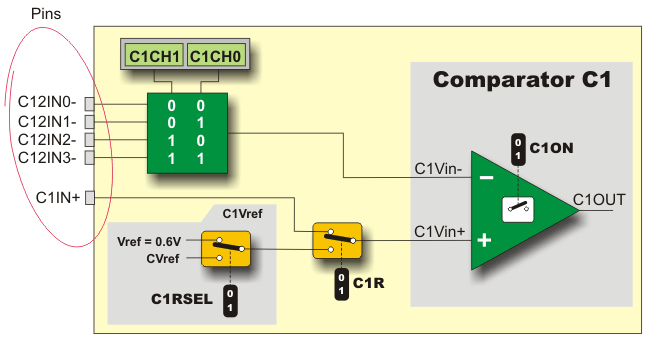
The comparator, if enabled before entering the *Sleep* mode, remains active during *Sleep*. If the comparator is not used to wake up the device, power consumption can be minimized in the *Sleep* mode by turning the comparator off. It is performed by clearing the CxON bit of the CMxCON0 register.

To enable the comparator to wake up the microcontroller from sleep, the CxIE bit of the IE2 register and the PEIE bit of the INTCON register must be set. The instruction following the *Sleep* instruction is always executed after exiting the *Sleep* mode. If the GIE bit of the INTCON register is set, the device will execute the *Interrupt Service Routine*.

**CM1CON0 Register**



Bits of this register are in control of the comparator C1. It mainly affects the configuration of its inputs. To understand it better, look at figure below which shows only a part of electronics directly affected by the bits of this register.



**C1ON - Comparator C1 Enable bit** enables comparator C1.

* 1 - Comparator C1 is enabled.
* 0 - Comparator C1 is disabled.

**C1OUT - Comparator C1 Output bit** is the output of the comparator C1.

If C1POL = 1 (comparator output is inverted)

* 1 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-.
* 0 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-.

If C1POL = 0 (comparator output is non-inverted)

* 1 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-.
* 0 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-.

**C1OE Comparator C1 Output Enable bit.**

* 1 - Comparator C1OUT output is connected to the C1OUT pin.\*
* 0 - Comparator output is internal only.

\* In order to enable the C1OUT bit to be present on the pin, two conditions must be met: C1ON = 1 (comparator must be on) and the corresponding TRIS bit = 0 (pin must be configured as an output).

**C1POL - Comparator C1 Output Polarity Select bit** enables the state of the comparator C1 output to be inverted.

* 1 - Comparator C1 output is inverted.
* 0 - Comparator C1 output is non-inverted.

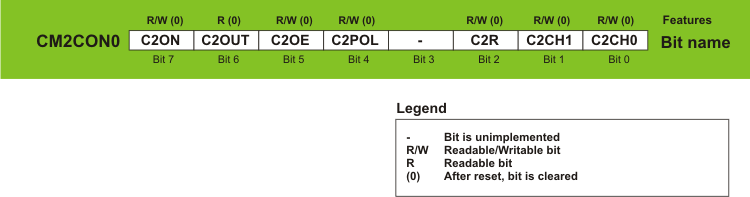
**C1R - Comparator C1 Reference Select bit**

* 1 - Non-inverting input C1Vin+ is connected to the reference voltage C1Vref.
* 0 - Non-inverting input C1Vin+ is connected to the C1IN+ pin.

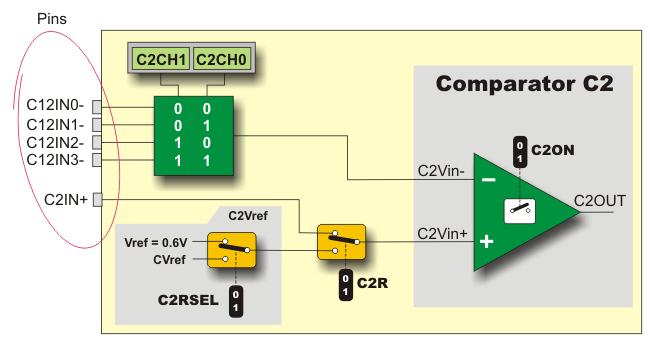
**C1CH1, C1CH0 - Comparator C1 Channel Select bit**

|  |  |  |
| --- | --- | --- |
| **C1CH1** | **C1CH0** | **Comparator C1Vin- input** |
| 0 | 0 | Input C1Vin- is connected to the C12IN0- pin |
| 0 | 1 | Input C1Vin- is connected to the C12IN1- pin |
| 1 | 0 | Input C1Vin- is connected to the C12IN2- pin |
| 1 | 1 | Input C1Vin- is connected to the C12IN3- pin |

**CM2CON0 Register**



Bits of this register are in control of the comparator C2. Similar to the previous case, figure below shows a simplified schematic of the circuit affected by the bits of this register.



**C2ON - Comparator C2 Enable bit** enables comparator C2.

* 1 - Comparator C2 is enabled; and
* 0 - Comparator C2 is disabled.

**C2OUT - Comparator C2 Output bit** is the output of the comparator C2.

If C2POL = 1 (comparator output inverted)

* 1 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-.
* 0 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-.

If C2POL = 0 (comparator output non-inverted)

* 1 - Analog voltage at C1Vin+ is higher than analog voltage at C1Vin-.
* 0 - Analog voltage at C1Vin+ is lower than analog voltage at C1Vin-.

**C2OE - Comparator C2Output Enable bit**

* 1 - Comparator C2OUT output is connected to the C2OUT pin.\*
* 0 - Comparator output is internal only.

\* In order to enable the C2OUT bit to be present on the pin, two conditions must be met: C2ON = 1 (comparator must be on) and the corresponding TRIS bit = 0 (pin must be configured as an output).

**C2POL - Comparator C2 Output Polarity Select bit** enables the state of the comparator C2 output to be inverted.

* 1 - Comparator C2 output is inverted.
* 0 - Comparator C2 output is non-inverted.

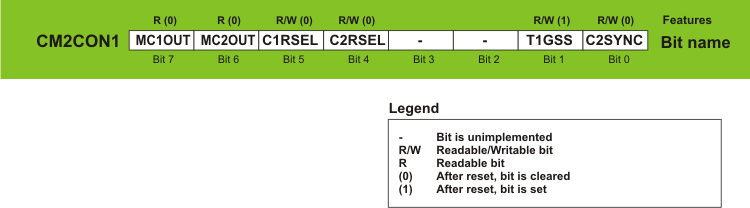
**C2R - Comparator C2 Reference Select bit**

* 1 - Non-inverting input C2Vin+ is connected to the reference voltage C2Vref.
* 0 - Non-inverting input C2Vin+ is connected to the C2IN+ pin.

**C2CH1, C2CH0 Comparator C2 Channel Select bit**

|  |  |  |
| --- | --- | --- |
| **C2CH1** | **C2CH0** | **Comparator C2Vin- input** |
| 0 | 0 | Input C2Vin- is connected to the C12IN0- pin |
| 0 | 1 | Input C2Vin- is connected to the C12IN1- pin |
| 1 | 0 | Input C2Vin- is connected to the C12IN2- pin |
| 1 | 1 | Input C2Vin- is connected to the C12IN3- pin |

**CM2CON1 Register**



**MC1OUT Mirror Copy of C1OUT bit**

**MC2OUT Mirror Copy of C2OUT bit**

**C1RSEL Comparator C1 Reference Select bit**

* 1 - Selectable voltage CVref is used in the voltage reference C1Vref source.
* 0 - Fixed voltage reference 0.6V is used in the voltage reference C1Vref source.

**C2RSEL - Comparator C2 Reference Select bit**

* 1 - Selectable voltage CVref is used in the voltage reference C2Vref source.
* 0 - Fixed voltage reference 0.6V is used in the voltage reference C2Vref source.

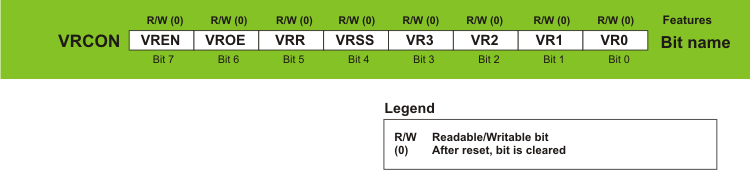
**T1GSS - Timer1 Gate Source Select bit**

* 1 - Timer T1gate source is T1G.
* 0 - Timer T1gate source is SYNCC2OUT.

**C2SYNC - Comparator C2 Output Synchronization bit**

* 1 - Comparator C2 output is synchronized to the falling edge of Timer TMR1 clock.
* 0 - Comparator output is asynchronous signal.

**VRCON Register**



**VREN Comparator C1 Voltage Reference Enable bit**

* 1 - Voltage reference CVref source is powered on.
* 0 - Voltage reference CVref source is powered off.

**VROE Comparator C2 Voltage Reference Enable bit**

* 1 - Voltage reference CVref is connected to the pin.
* 0 - Voltage reference CVref is disconnected from the pin.

**VRR - CVref Range Selection bit**

* 1 - Voltage reference source is set to low range.
* 0 - Voltage reference source is set to high range.

**VRSS - Comparator Vref Range selection bit**

* 1 - Voltage reference source is in the range of Vref+ to Vref-.
* 0 - Voltage reference source is in the range of Vdd to Vss (power supply voltage).

**VR3 - VR0 CVref Value Selection**

If VRR = 1 (low range)

Voltage reference is calculated using the formula: CVref = ([VR3:VR0]/24)Vdd

If VRR = 0 (high range)

Voltage reference is calculated using the formula: CVref = Vdd/4 + ([VR3:VR0]/32)Vdds

**In Short**

In order to properly use built-in comparators, it is necessary to do the following:

**Step 1** - Module Configuration:

* In order to select the appropriate mode, bits of the CM1CON0 and CM2CON0 registers should be configured. Interrupt should be disabled on any change of mode.

**Step 2** - Internal voltage reference Vref source configuration (only when used). In the VRCON register it is necessary to:

* Select one of two voltage ranges using the VRR bit.
* Configure necessary Vref using bits VR3 - VR0.
* Set the VROE bit if needed.
* Enable voltage Vref source by setting the VREN bit.

Formula used to calculate voltage reference:

**VRR = 1 (low range)**  
CVref = ([VR3:VR0]/24)VLADDER

**VRR = 0 (high range)**  
CVref = (VLADDER/4) + ([VR3:VR0]VLADDER/32)

Vladder = Vdd or ([Vref+] - [Vref-]) or Vref+

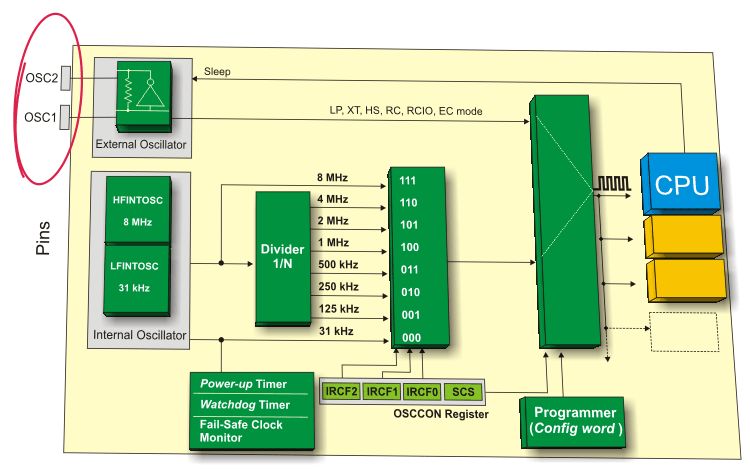
**Step 3** - Start of operation:

* Enable an interrupt by setting bits CMIE (PIE register), PEIE and GIE (INTCON register).
* Read the C1OUT and C2OUT bits of the CMCON register.
* Read the CMIF flag bit of the PIR register. After being set, this bit must be cleared in software.

*In order to synchronize all the processes taking place within the microcontroller, a clock signal must be used, while in order to generate the clock signal, a clock oscillator must be used. As simple as that. This microcontroller has several oscillators capable of working in different modes and this is where the story becomes interesting...*

**3.10 CLOCK OSCILLATOR**

As seen in figure below, the clock signal may be generated by one out of two built-in oscillators.



An external oscillator is installed within the microcontroller and connected to the OSC1 and OSC2 pins. It is called ‘external’ because it relies on an external circuit for the clock signal and frequency stabilization, such as a stand-alone oscillator, quartz crystal, ceramic resonator or resistor-capacitor circuit. The oscillator mode is selected by the bits of bytes, called Config Word, sent during programming.

**Internal oscillator** consists of two separate internal oscillators:

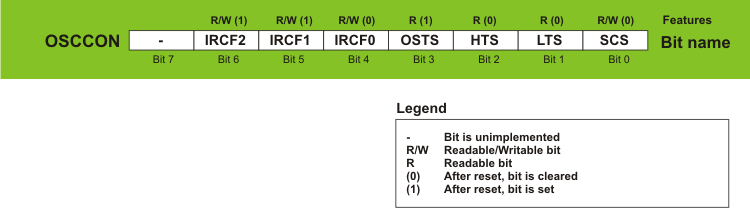
The HFINTOSC is a high-frequency internal oscillator which operates at 8MHz. The microcontroller can use clock source generated at this frequency or after being divided in prescaler.

The LFINTOSC is a low-frequency internal oscillator which operates at 31 kHz. Its clock sources are used for *watch-dog* and *power-up* timing, but it can also be used as a clock source for the operation of the entire microcontroller.

The system clock can be selected between external or internal clock source via the System Clock Select (SCS) bit of the OSCCON register.

**OSCCON Register**

The OSCCON register controls the system clock and frequency selection options. It contains the following bits: frequency selection bits (IRCF2, IRCF1, IRCF0), frequency status bits (HTS, LTS), system clock control bits (OSTA, SCS).



**IRCF2-0 - Internal Oscillator Frequency Select bits.** The divider rate depends on the combination of these three bits. The clock frequency of internal oscillator is determined in the same way.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **IRCF2** | **IRCF1** | **IRCF0** | **Frequency** | **OSC.** |
| 1 | 1 | 1 | 8 MHz | HFINTOSC |
| 1 | 1 | 0 | 4 MHz | HFINTOSC |
| 1 | 0 | 1 | 2 MHz | HFINTOSC |
| 1 | 0 | 0 | 1 MHz | HFINTOSC |
| 0 | 1 | 1 | 500 kHz | HFINTOSC |
| 0 | 1 | 0 | 250 kHz | HFINTOSC |
| 0 | 0 | 1 | 125 kHz | HFINTOSC |
| 0 | 0 | 0 | 31 kHz | LFINTOSC |

**OSTS - Oscillator Start-up Time-out Status bit** indicates which clock source is currently in use. It is read-only.

* 1 - External clock oscillator is in use.
* 0 - One of internal clock oscillators is in use (HFINTOSC or LFINTOSC).

**HTS - HFINTOSC Status bit** (8 MHz - 125 kHz) indicates whether the high-frequency internal oscillator operates in a stable way.

* 1 - HFINTOSC is stable.
* 0 - HFINTOSC is not stable.

**LTS - LFINTOSC Stable bit** (31 kHz) indicates whether the low-frequency internal oscillator operates in a stable way.

* 1 - LFINTOSC is stable.
* 0 - LFINTOSC is not stable.

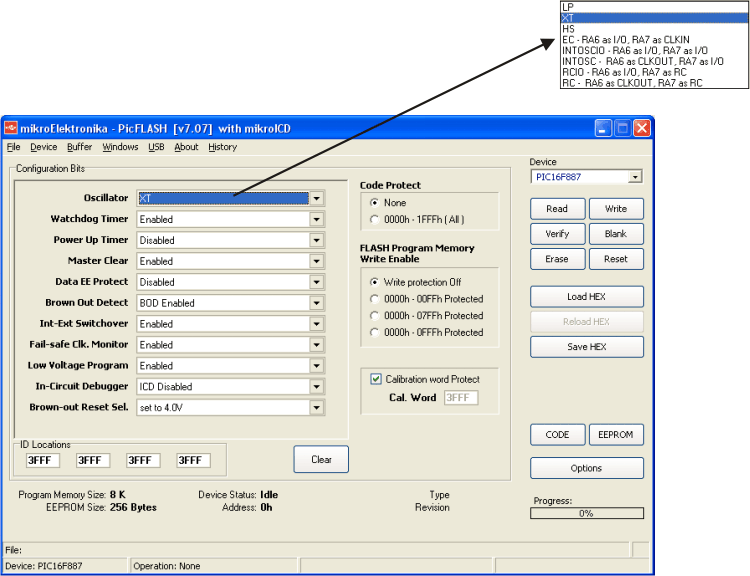
**SCS - System Clock Select bit** determines which oscillator is to be used as a clock source.

* 1 - Internal oscillator is used for system clock.
* 0 - External oscillator is used for system clock.  
  The oscillator mode is set by bits in *Config Word* written to the microcontroller memory during the process of programming.

**EXTERNAL CLOCK MODES**

The external oscillator can be configured to operate in one out of several modes, which enables it to operate at different speeds and use different components for frequency stabilization. Mode of operation is selected during the process of writing a program into the microcontroller. First of all, it is necessary to activate the program on a PC to be used for programming. It is the *PICflash* program in this case. Click on the oscillator field and select one oscillator from the drop-down list. The appropriate bits will be automatically set, thus becoming a part of several bytes which together form a *Config Word*.

During the process of programming the microcontroller, these bytes of Config Word are written to the microcontroller’s ROM memory and stored in special registers which are not available to the user. On the basis of these bits, the microcontroller 'knows' what to do, although it is not explicitly stated in the program. Mode of operation is selected after the process of writing and compiling a program



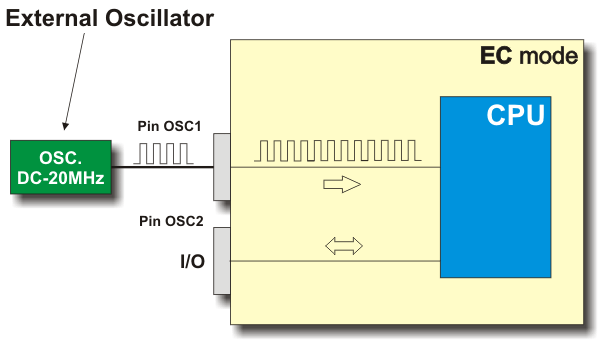
**EXTERNAL OSCILLATOR IN EC MODE**

The external clock (EC) mode uses external oscillator as a clock source. The maximum frequency of this clock is limited to 20 MHz.

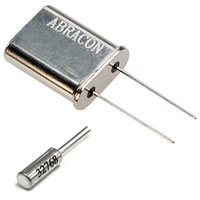


The advantages of the external oscillator when configured to operate in EC mode:

* The independent external clock source is connected to the OSC1 input and the OSC2 is available as a general purpose I/O;
* It is possible to synchronize the operation of the microcontroller with the rest of on-board electronics;
* In this mode the microcontroller starts operation immediately after the power is on. No time delay is required for frequency stabilization; and
* Temporary disabling the external clock source causes device to stop operation, while leaving all data intact. After restarting the external clock, the device proceeds with operation as if nothing has happened.

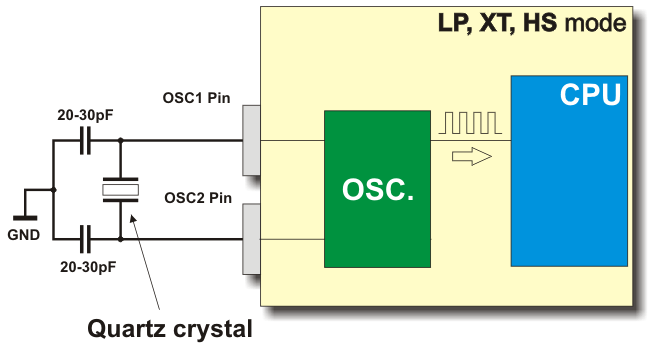


**EXTERNAL OSCILLATOR IN LP, XT OR HS MODE**



The LP, XT and HS modes use external oscillator as a clock source the frequency of which is determined by quartz crystal or ceramic resonators connected to the OSC1 and OSC2 pins. Depending on the features of the component in use, select one of the following modes:

* **LP mode** - (Low Power) is used for low-frequency quartz crystal only. This mode is designed to drive only 32.768 kHz crystals usually embedded in quartz watches. It is easy to recognize them by small size and specific cylindrical shape. The current consumption is the least of the three modes.
* **XT mode** is used for intermediate-frequency quartz crystals up to 8 MHz. The current consumption is the medium of the three modes.
* **HS mode** - (High Speed) is used for high-frequency quartz crystals over 8 MHz. The current consumption is the highest of the three modes.



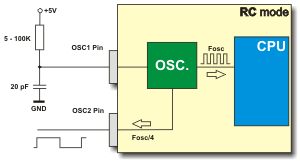
**CERAMIC RESONATORS IN XT OR HS MODE**

Ceramic resonators are by their features similar to quartz crystals and are connected in the same way, therefore. Unlike quartz crystals, they are cheaper and oscillators containing them have a bit poorer characteristics. They are used for clock frequencies ranging from 100 kHz to 20 MHz.

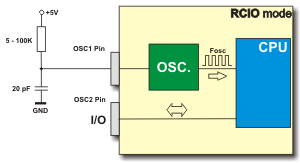


**EXTERNAL OSCILLATOR IN RC AND RCIO MODE**

There are certainly many advantages in using elements for frequency stabilization, but sometimes they are really unnecessary. In most cases the oscillator may operate at frequencies not precisely defined so that embedding of such elements is a waste of money. The simplest and cheapest solution in these situations is to use one resistor and one capacitor for the operation of oscillator. There are two modes:



**RC mode.** When the external oscillator is configured to operate in RC mode, the OSC1 pin should be connected to the RC circuit as shown in figure on the right. The OSC2 pin outputs the RC oscillator frequency divided by 4. This signal may be used for calibration, synchronization or other application requirements.



**RCIO mode.** Likewise, the RC circuit is connected to the OSC1 pin. This time, the available OSC2 pin is used as an additional general-purpose I/O pin.

In both cases, it is recommended to use components as shown in figure.  
The frequency of such an oscillator is calculated according to the formula f = 1/T in which:

* f = frequency [Hz];
* T = R \* C = time constant [s];
* R = resistor resistance [Ω]; and
* C = capacitor capacity [F].

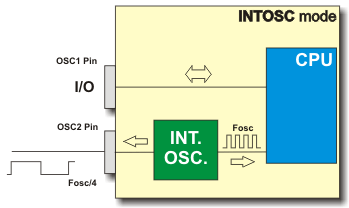
**INTERNAL CLOCK MODES**

The internal oscillator circuit consists of two separate oscillators that can be selected as the system clock source:

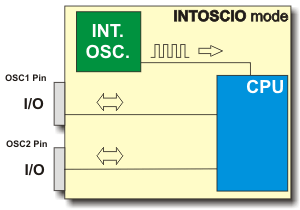
The **HFINTOSC** oscillator is factory calibrated and operates at 8 MHz. Its frequency can be set by the user via software using bits of the OSCTUNE register.

The **LFINTOSC** oscillator is not factory calibrated and operates at 31kHz.

Similar to the external oscillator, the internal one can also operate in several modes. The mode of operation is selected in the same way as with external oscillator - using bits of the *Config Word* register. In other words, everything is performed within PC software prior to writing a program into the microcontroller.



**INTERNAL OSCILLATOR IN INTOSC MODE**  
In this mode, the OSC1 pin is available as a general purpose I/O, while the OSC2 pin outputs selected internal oscillator frequency divided by 4.



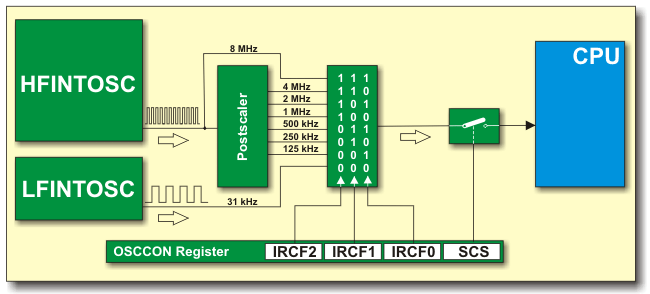
**INTERNAL OSCILLATOR IN INTOSCIO MODE**  
In this mode, both pins are available as a general purpose I/O.

**INTERNAL OSCILLATOR SETTINGS**

The internal oscillator consists of two separate circuits.

1. The high-frequency internal oscillator HFINTOSC is connected to the postscaler (frequency divider). It is factory calibrated and operates at 8MHz. By using postscaler, this oscillator can output clock sources at one out of seven frequencies. The frequency selection is performed within software using the IRCF2, IRCF1 and IRCF0 pins of the OSCCON register.

The HFINTOSC is enabled by selecting one out of seven frequencies (between 8 MHz and 125 kHz) and setting the System Clock Source (SCS) bit of the OSCCON register. As seen in figure below, everything is performed by using bits of the OSCCON register.



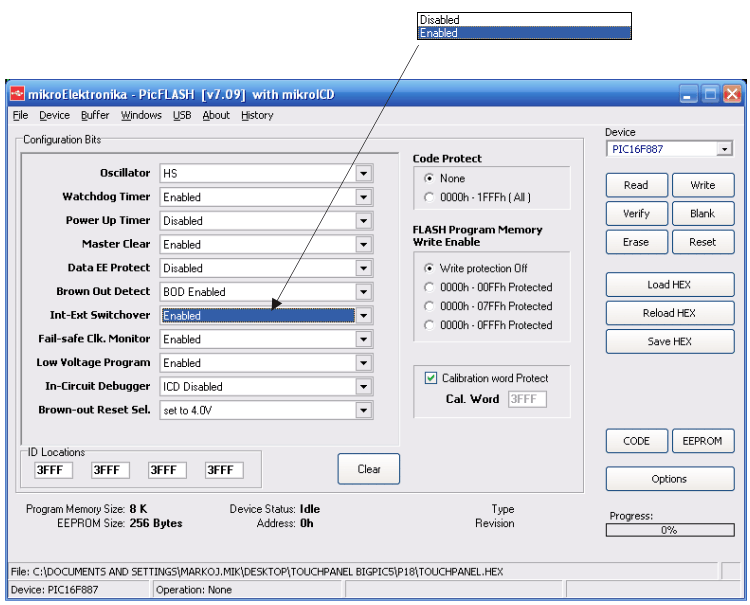
2. The low-frequency oscillator LFINTOSC is uncalibrated and operates at 31 kHz. It is enabled by selecting this frequency (bits of the OSCCON register) and setting the SCS bit of the same register.

**TWO-SPEED CLOCK START-UP MODE**

Two-Speed Clock Start-up mode is used to provide additional power savings when the microcontroller operates in sleep mode. What is this all about?

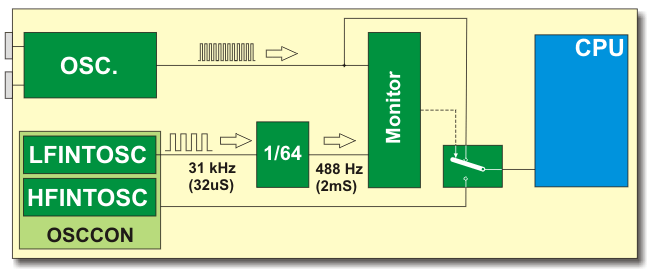
When configured to operate in LP, XT or HS mode, the external oscillator will be switched off on transition to *sleep* in order to reduce the overall power consumption of the device.

When the conditions for wake-up are met, the microcontroller will not immediately start to operate because it has to wait for the clock signal frequency to become stable. Such delay lasts for exactly 1024 pulses, then the microcontroller proceeds with program execution. It usually happens that only a few instructions are performed before the microcontroller is set back to *Sleep mode*. It means that most of time as well as most of power obtained from batteries is wasted. The problem is solved by using an internal oscillator for program execution while the counting of these 1024 pulses is in progress. As soon as the external oscillator frequency becomes stable, it will automatically take over the 'leading role'. The whole process is enabled by setting one bit of the configuration word. In order to program the microcontroller, it is necessary to select the *Int-Ext Switchover* option in software.



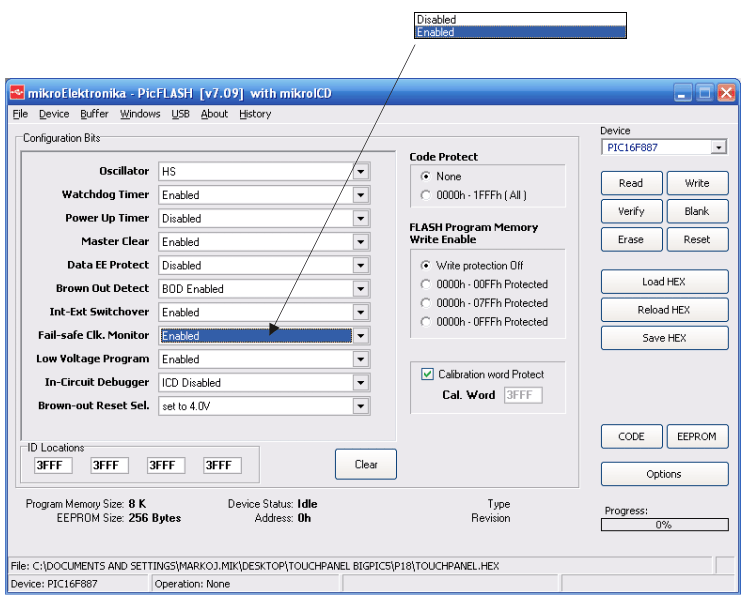
**FAIL-SAFE CLOCK MONITOR**

As its name suggests, the Fail-Safe Clock Monitor (FSCM) monitors the operation of external oscillator and allows the microcontroller to proceed with program execution even though the external oscillator fails for some reason. In this case, the internal oscillator takes over its role.



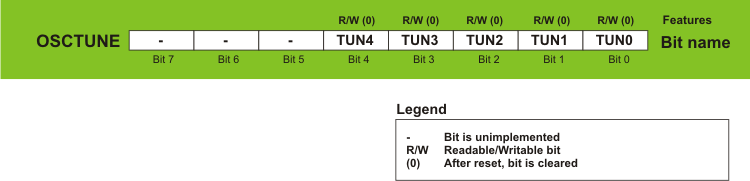
The fail-safe clock monitor detects the failure by comparing internal and external clock sources. If it takes more than 2mS for the external oscillator clock to come, the clock source will be automatically switched. The internal oscillator will thereby continue the operation controlled by the bits of the OSCCON register. When the OSFIE bit of the PIE2 register is set, an interrupt will be generated. The system clock will keep on being sourced from internal clock until the device successfully restarts the external oscillator and switches back to external operation.

Similarly, this module is enabled by changing configuration word directly before the process of programming chip starts. This time, it is done by selecting the Fail-Safe Clock Monitor option.



**OSCTUNE Register**

Modifications in the OSCTUNE register affect the HFINTOSC frequency, but not the LFINTOSC frequency. There is no indication during the operation that frequency shift has occurred.



**TUN4 - TUN0 Frequency Tuning bits.** By combining these five bits, the 8MHz oscillator frequency shifts. In this way, the frequencies obtained by its division in the postscaler shift too.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TUN4** | **TUN3** | **TUN2** | **TUN1** | **TUN0** | **Frequency** |
| 0 | 1 | 1 | 1 | 1 | Maximal |
| 0 | 1 | 1 | 1 | 0 |  |
| 0 | 1 | 1 | 0 | 1 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 0 | 0 | 0 | 0 | 1 |  |
| 0 | 0 | 0 | 0 | 0 | Calibrated |
| 1 | 1 | 1 | 1 | 1 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| 1 | 0 | 0 | 1 | 0 |  |
| 1 | 0 | 0 | 0 | 1 |  |
| 1 | 0 | 0 | 0 | 0 | Minimal |

*Eeprom is a separate memory segment, not part of program memory (ROM), nor data memory (RAM). Even though these memory locations are not easily and quickly accessed as other registers, their purpose is irreplaceable as the EEPROM data is permanently saved even after the loss of power and can be changed at any moment. These exceptional features make each byte of EEPROM valuable.*

**3.11 EEPROM MEMORY**

The PIC16F887 microcontroller has 256 locations of data EEPROM controlled by the bits of the following registers:

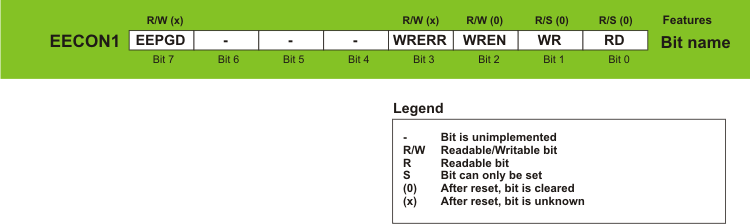
* EECON1 (control register);
* EECON2 (control register);
* EEDAT (saves data ready for write and read); and
* EEADR (saves address of EEPROM location to be accessed).

In addition, EECON2 is not true register, it does not physically exist. It is used in data write program sequence only.

The EEDATH and EEADRH registers are used during EEPROM write and read. Both of them are also used for program (FLASH) memory write and read.

Since this is considered a risk zone (you surely do not want your microcontroller to accidentally delete it’s own program), we will not discuss it further, but advise you to be careful.

**EECON1 Register**



**EEPGD - Program/Data EEPROM Select bit**

* 1 - Access program memory.
* 0 - Access EEPROM memory.

**WRERR - EEPROM Error Flag bit**

* 1 - Write operation is prematurely terminated and error has occurred.
* 0 - Write operation completed.

**WREN - EEPROM Write Enable bit.**

* 1 - Write to data EEPROM is enabled.
* 0 - Write to data EEPROM is disabled.

**WR - Write Control bit**

* 1 - Initiates write to data EEPROM.
* 0 - Write to data EEPROM is complete.

**RD - Read Control bit**

* 1 - Initiates read from data EEPROM.
* 0 - Read from data EEPROM is disabled.

**READ FROM EEPROM MEMORY**

In order to read data EEPROM memory, follow the procedure below:

* **Step 1**: Write the address (00h - FFh) to the EEADR register.
* **Step 2**: Select EEPROM memory block by clearing the EEPGD bit of the EECON1 register.
* **Step 3**: To read location, set the RD bit of the same register.
* **Step 4**: Data is stored in the EEDAT register and is ready for use.

The following example illustrates the above procedure when writing a program in assembly language:

BSF STATUS,RP1 *;*  
BCF STATUS,RP0 *; Access bank 2*  
MOVF ADDRESS,W *; Move address to the W register*  
MOVWF EEADR *; Write address*  
BSF STATUS,RP0 *; Access bank 3*  
BCF EECON1,EEPGD *; Select EEPROM*  
BSF EECON1,RD *; Read data*  
BCF STATUS,RP0 *; Access bank 2*  
MOVF EEDATA,W *; Data is stored in the W register*

The same program sequence written in C language looks as follows:

W = EEPROM\_Read(ADDRESS);

The advantages of C language becomes more obvious, don’t they?

**WRITE DATA TO EEPROM MEMORY**

Prior to writing data to EEPROM memory it is necessary to write the address to the EEADR register and data to the EEDAT register. All that’s left is to follow a special sequence to initiate write for each byte. Interrupts must be disabled as long as this procedure is in progress.

The example below illustrates the above procedure when writing a program in assembly language:

BSF STATUS,RP1  
BSF STATUS,RP0  
BTFSC EECON,WR1 ; Wait for the previous write to complete  
GOTO $-1 ;  
BCF STATUS,RP0 ; Bank 2  
MOVF ADDRESS,W ; Move address to W  
MOVWF EEADR ; Write address  
MOVF DATA,W ; Move data to W  
MOVWF EEDATA ; Write data  
BSF STATUS,RP0 ; Bank 3  
BCF EECON1,EEPGD ; Select EEPROM  
BSF EECON1,WREN ; Write to EEPROM enabled  
BCF INCON,GIE ; All interrupts disabled  
  
;Required Sentence  
MOVLW 55h  
MOVWF EECON2  
MOVLW AAh  
MOVWF EECON2  
BSF EECON1,WR  
  
BSF INTCON,GIE ; Interrupts enabled  
BCF EECON1,WREN ; Write to EEPROM disabled

The same program sequence written in C language looks as follows:

W = EEPROM\_Write(ADDRESS, W);

Need a comment?

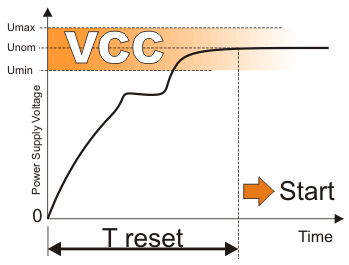
**Let's do it in mikroC...**

*// This example demonstrates the use of EEPROM Library in mikroC PRO for PIC.*  
  
**char** ii; *// Loop variable*  
  
**void** main(){  
 ANSEL = 0; *// Configure AN pins as digital I/O*  
 ANSELH = 0;  
 PORTB = 0;  
 PORTC = 0;  
 PORTD = 0;  
 TRISB = 0;  
 TRISC = 0;  
 TRISD = 0;  
   
 **for**(ii = 0; ii < 32; ii++) *// Fill data buffer*  
 EEPROM\_Write(0x80+ii, ii); *// Write data to address 0x80+ii*  
   
 EEPROM\_Write(0x02,0xAA); *// Write some data to EEPROM address 2*  
 EEPROM\_Write(0x50,0x55); *// Write some data to EEPROM address 0x50*  
   
 Delay\_ms(1000); *// Blink PORTB and PORTC diodes*  
 PORTB = 0xFF; *// to indicate start of reading*  
 PORTC = 0xFF;   
 Delay\_ms(1000);  
   
 PORTB = 0x00;  
 PORTC = 0x00;  
 Delay\_ms(1000);  
   
 PORTB = EEPROM\_Read(0x02); *// Read data from EEPROM address 2 and display it on PORTB*  
 PORTC = EEPROM\_Read(0x50); *// Read data from EEPROM address 0x50 and display it on PORTC*  
 Delay\_ms(1000);  
   
 **for**(ii = 0; ii < 32; ii++) { *// Read 32 bytes block from address 0x80*  
 PORTD = EEPROM\_Read(0x80+ii); *// and display data on PORTD*  
 Delay\_ms(250);  
 }  
}

*At first glance, it is sufficient to turn the power on to make the microcontroller operate. At first glance, it is sufficient to turn the power off to make it stop operating. Only at first glance... In reality, start and end of operation are critical phases of which a special signal called RESET takes care.*

**3.12 RESET! BLACK-OUT, BROWN-OUT OR NOISES?**

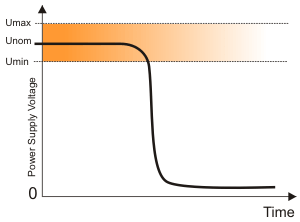
Reset condition causes the microcontroller to immediately stop operation and clear its registers. Areset signal may be generated externally at any moment (low logic level on the MCLR pin). If needed, it can also be generated by internal control logic. Power-on always causes reset. Since there are many transitional events taking place when power supply is turned on (switch contact flashing and sparkling, slow voltage rise, gradual clock frequency stabilization etc.), it is necessary to provide a certain time delay for the microcontroller before it starts to operate. Two internal timers- PWRT and OST are in charge of that. The first one can be enabled or disabled during the process of writing a program. Let’s take a look what happens then:



When the power supply voltage reaches 1.2 - 1.7V, a circuit called *Power-up timer* resets the microcontroller within approximately 72mS. As soon as this time expires, another timer called *Oscillator start-up timer* generates another reset signal within 1024 quartz oscillator periods. When this delay expires (marked as T *reset* in figure) and the MCLR pin is set high, all conditions are met and the microcontroller starts to execute the first instruction in the program.

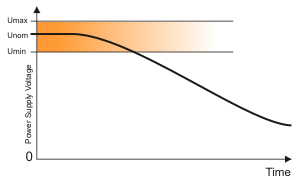
Apart from this 'controlled' reset which occurs at the moment power goes on, there are another two resets called *Black-out* and *Brown-out* which may occur during the operation as well as at the moment the power supply goes off.

**BLACK-OUT RESET**



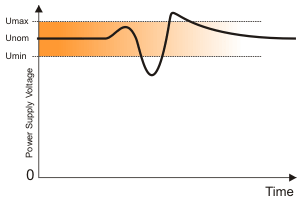
Black-out reset takes place when the power supply normally goes off. The microcontroller then has no time to do anything unpredictable simply because the voltage drops very fast beneath its minimum value. In other words the light goes off, curtain falls down and the show is over!

**BROWN-OUT RESET**



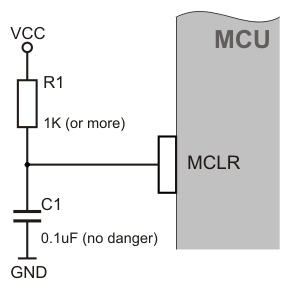
When the power supply voltage drops slowly (typical example is battery discharge, although the microcontroller experiences far faster voltage drops as slow processes), the internal electronics gradually stops to operate and the so called Brown-out reset occurs. Here, before the microcontroller completely stops the operation there is a real danger that circuits which operate at higher voltages start to perform unpredictably. Brown-out reset can also cause fatal changes in the program because it is saved in on-chip flash memory.

**NOISE**



This is a special type of Brown-out reset which occurs in industrial environment when the power supply voltage 'blinks' for a moment and drops beneath minimum level. Even short, such noise in power line may considerably affect the operation of the device.

**MCLR PIN**



A logic zero (0) on the MCLR pin causes an immediate and regular reset. It is recommended to connect it as per figure on the right. The function of additional components is to sustain 'pure' logic one (1) during normal operation. If their values are selected so as to provide high logic level on the pin after T reset is over, the microcontroller will immediately start the operation. This may be very useful when it is necessary to synchronize the operation of the microcontroller with additional electronics or the operation of several microcontrollers.

In order to avoid any error which may occur on *Brown-out reset*, the PIC 16F887 has built in 'protection mechanism'. It is a simple, but effective circuit which responds every time the power supply voltage drops below 4V and keeps this level for more than 100 micro seconds. This circuit generates a reset signal and since that moment the whole microcontroller operates as if it has just been turned on.