# CENG 336
# INT. TO EMBEDDED SYSTEMS DEVELOPMENT
# 2022-2023 Spring
# Recitation-5

## Today's Outline

- RTOS (Real Time Operating Systems)
- PICos18
- Serial Communication
- Example Codes

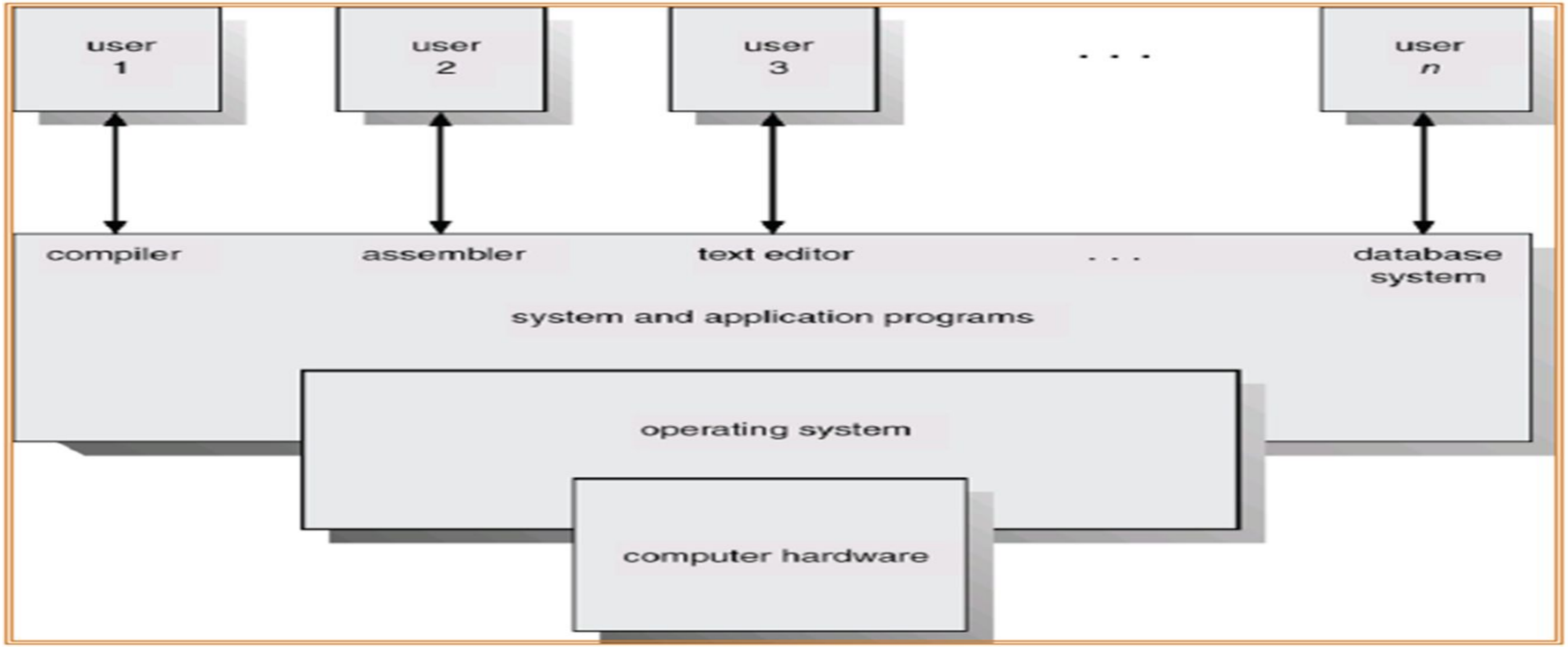
## Operating System

- An operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.
- Operating system goals:
  - ✔ Execute user programs and make solving user problems easier
  - ✔ Make the computer system convenient to use
- Use the computer hardware in an efficient manner.

## Computer System Components

- **Hardware:** CPU, memory, I/O devices
- **Operating system:** CPU management, Drivers, Memory Management
- **Applications programs:** compilers, database systems, video games, business programs (They define the ways in which the system resources are used to solve the computing problems of the users.)
- **Users:** people, machines, other computers

## Abstract View of System Components



## Real Time Operating System (RTOS)

- RTOS is the operating system which is capable of responding the events immediately. RTOSs are often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.
- They have well-defined fixed-time constraints.

## Terminology

- **Critical section,** or critical region, is code that needs to be treated indivisibly.
  - ✔ No interrupts
  - ✔ No context switch
- **Resource** is an entity used by a task.
  - ✔ Printer, keyboard, CAN bus, serial port
- **Shared resource** is a resource that can be used by more than one task.
  - ✔ Mutual exclusion
- **Multitasking** is the process of scheduling and switching the CPU between several tasks.

## Terminology

- **Kernel** is a set of functionalities, regroup under the term SERVICES for most of them.
  - ✔ For example, in the case of Linux, the kernel is composed of the task manager, the hardware access manager, the file system manager .
  - ✔ One of the most famous functionality of the kernel (without being a service) is the **SCHEDULER** in charge of the task processing in parallel.

## Terminology

***Non-preemptive* kernels, also cooperative multitasking:**

- The task needs to explicitly give up control of the CPU.
- Allows low interrupt latency, because they may be never disabled.
- Allows non-reentrant functions at the task level.
- Response time is determined by the longest task.
- No overhead for protecting shared data.
- Responsiveness may be low, because of low priority task requiring a lot of time until it releases the CPU.

## Terminology

***Preemptive* kernel:**

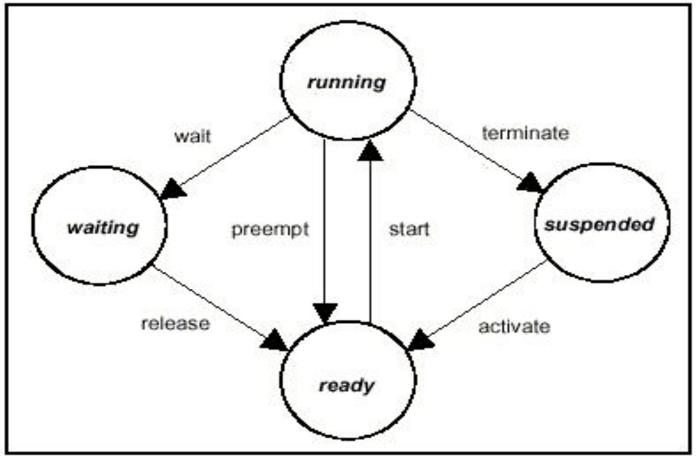
- Responsiveness is good, because tasks get preempted.
- A higher-priority task can preempt a lower priority task that still requires more time to compute.
- Response time becomes deterministic, because at the next tick, the OS switches to the other new task.
- Non-reentrant functions require careful programming.
- Periodic execution of the ‘tick’ adds to the overhead.

## PICos18

- PICos18 is a preemptive RTOS for the PIC18 series.
- PICos18 provides:
  - ✔ Core services: initialization, scheduling
  - ✔ Alarm and counter manager
  - ✔ Hook routines
  - ✔ Task manager
  - ✔ Event manager
  - ✔ Interrupt manager

## TASK

- **Task**, also called thread, is a user application.
  - ✔ Shares the CPU and resources with other tasks .
  - ✔ Follows a defined life cycle.
  - ✔ 4 possible task states in PICos18:
    - Suspended state
    - Ready state
    - Running state
    - Waiting state



## TASK

There are 4 task states in PICos18:

- **SUSPENDED:** The task is present in the application but is not taken into account by the kernel.
- **READY:** The task is available to be executed by the kernel then is taken into account by the scheduler.
- **WAITING:** The task is sleeping and so is temporarily SUSPENDED and will be READY as soon as an event occurs.
- **RUNNING:** There is only one task running at a certain time => this is the task in a READY state with the highest priority.

# TASK

***Context Switches***

- A context switch occurs whenever the multitasking kernel decides to run a different task.
  - ✔ Save the current task's context in the storage area.
  - ✔ Restores the new task's context from the storage area.
  - ✔ Resumes the new task.
- Context switching adds overhead.
- The more registers a processor has, the higher the overhead => irrelevant for RTOS as long as its known.
- Context switching is done by Kernel.

## TASK

### *Software Stack*

- PICos18 is compliant with the C18 software stack management. Each task has its own software stack so that each task of the application can work as alone in its own private space without disturbing the other processing. Only the 3 below values are possible:
  - ✔ A minimum software stack is 64.
  - ✔ A maximum software stack is 256.
  - ✔ A typical stack size is 128.

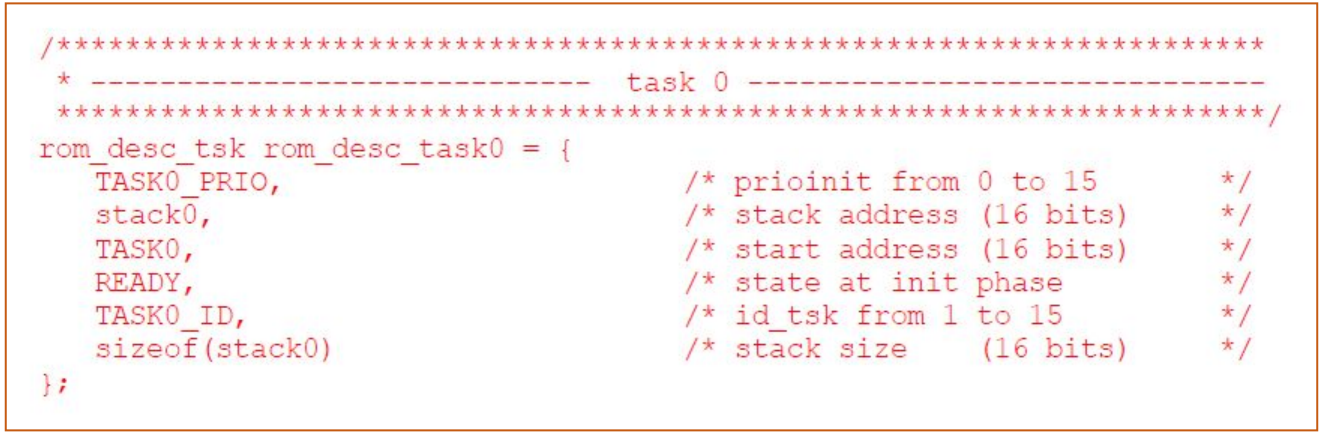
## TASK

***Software Stack***

- PICos18 has an automatic stack overflow detection that occurs when a software stack becomes bigger than expected **(kernelpanic flag).** Moreover a software stack has to be always in a same bank and not on 2 pieces of bank. If you don't know let all the stack size to 128.
- For example, if KernelPanic value is 0x53:
  -  The right nibble ("3") means the task with the ID 3 has a stack overflow.
  -  The left nibble ("5") means the task with the ID 5 has a stack trashed by an overflow.

# TASK

Task Definition (in *taskdesc.c* file):



```
/**********************************************************************
 * ----------------------------  task 0 ------------------------------
 **********************************************************************/
rom_desc_tsk rom_desc_task0 = {
   TASK0_PRIO,                          /* prioinit from 0 to 15      */
   stack0,                              /* stack address (16 bits)    */
   TASK0,                               /* start address (16 bits)    */
   READY,                               /* state at init phase        */
   TASK0_ID,                            /* id_tsk from 1 to 15        */
   sizeof(stack0)                       /* stack size    (16 bits)    */
};
```

## EVENTS

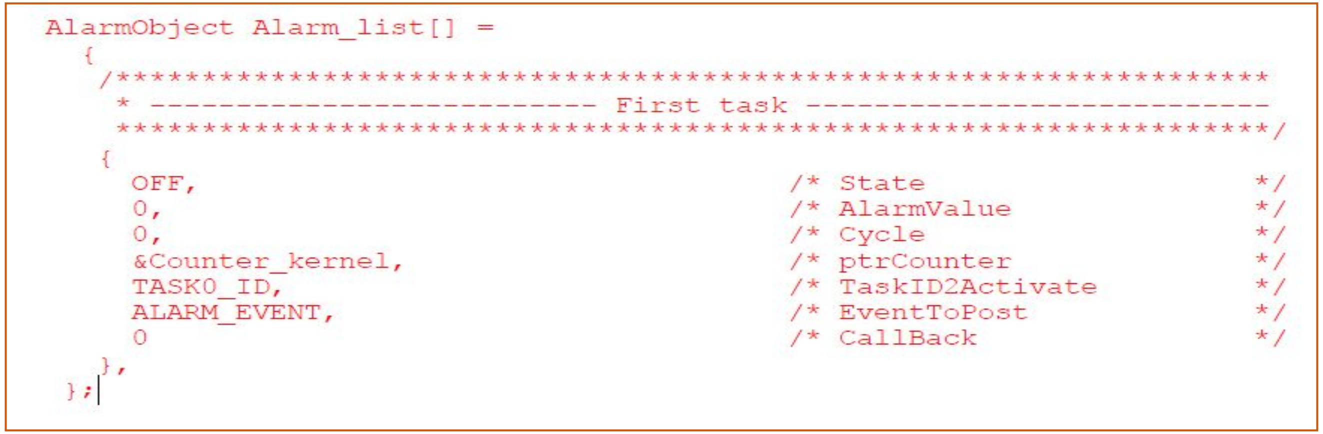
- Events are special signals to alert some task to make it in READY state while it was in WAITING state.
- They are similar to interrupts. When an event comes, it has to be cleared.
- The related functions:
  - ✔ SetEvent(*Event_Name*)
  - ✔ WaitEvent(*Event_Name*)
  - ✔ ClearEvent(*Event_Name*)

## EVENTS

- You can put to sleep your task on any event but keep in mind the event must be defined as a power of 2 in the "*define.h*" file. The allowed values are : 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02 et 0x01.

# ALARMS

- Alarm Definition (in *taskdesc.c* file):

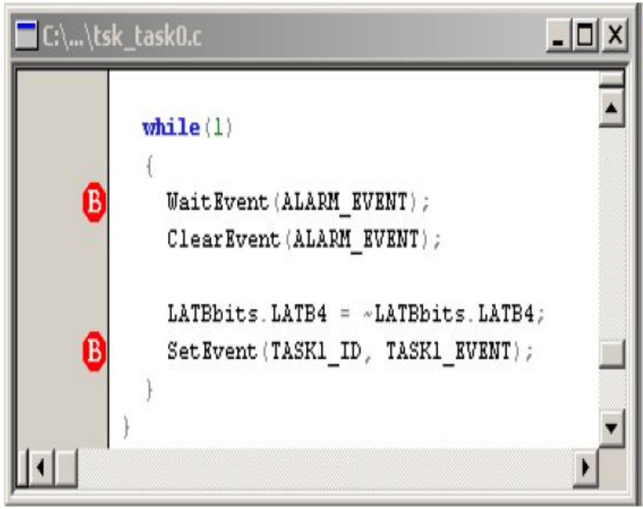


```
AlarmObject Alarm_list[] =
  {
   /*******************************************************************
    * -------------------------- First task ---------------------------
    *******************************************************************/
   {
     OFF,                                      /* State              */
     0,                                        /* AlarmValue         */
     0,                                        /* Cycle              */
     &Counter_kernel,                          /* ptrCounter         */
     TASK0_ID,                                 /* TaskID2Activate    */
     ALARM_EVENT,                              /* EventToPost        */
     0                                         /* CallBack           */
   },
 };
```

## ALARMS

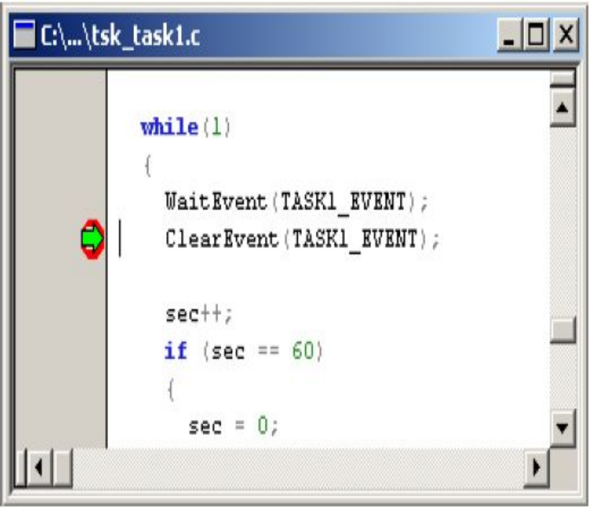
- At each tick the alarm counters get incremented by one.
- If the alarm value equals to the counter value, then the alarm will cause an event : ALARM_EVENT.
- The related functions:
  - ✔ SetRelAlarm(*Alarm_Name*, *Starting_Time*, *Interval*)
  - ✔ AddOneTick()
  - ✔ WaitEvent(*ALARM_EVENT*)
  - ✔ ClearEvent(*ALARM_EVENT*)

## PREEMPTION


C:\...\tsk_task0.c

```
while(1)
{
  WaitEvent(ALARM_EVENT);
  ClearEvent(ALARM_EVENT);

  LATBbits.LATB4 = ~LATBbits.LATB4;
  SetEvent(TASK1_ID, TASK1_EVENT);
}
}
```


C:\...\tsk_task1.c

```
while(1)
{
  WaitEvent(TASK1_EVENT);
  ClearEvent(TASK1_EVENT);

  sec++;
  if (sec == 60)
  {
    sec = 0;
```

Assume task0_priority > task1_priority. How does preemption occur?
Assume task1_priority > task0_priority. How does preemption occur?

## PREEMPTION

**If task0_priority > task1_priority:**

- Actually if the task-0 has a priority higher than the task-1 then when it posts an event  to the task-1 it continues to run until it falls asleep by the WaitEvent function.
- Then the task-1 is free to run and the pointer stops on the ClearEvent breakpoint.
- The pointer runs through the breakpoints in this order:
  - ✔ WaitEvent in the task-0
  - ✔ SetEvent in the task-0
  - ✔ WaitEvent in the task-0
  - ✔ ClearEvent in the task-1

## PREEMPTION

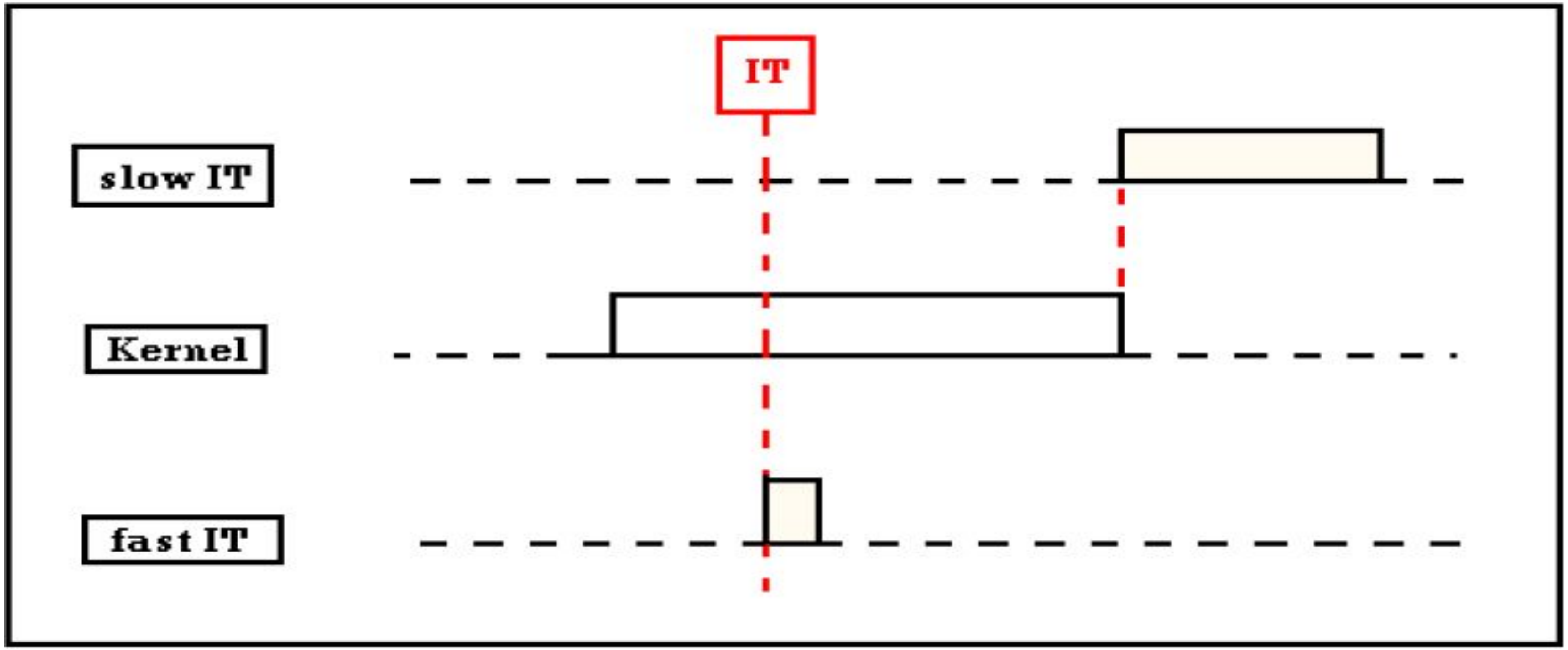
**If task1_priority > task0_priority:**

- Task-1 will be tried to run first but it is dependent on Task-0 for it to send TASK1_EVENT.
- Hence Task-0 will be run, but whenever it sends TASK1_EVENT to Task-1, the execution will be switched to Task-1. Until Task-1 finishes its job, Task-0 will not be run even if an alarm event occurs.
- The pointer runs through the breakpoints in this order:
  - ✔ WaitEvent in the task-0
  - ✔ SetEvent in the task-0
  - ✔ ClearEvent in the task-1

## PREEMPTION

- The preemption is not a sufficient reason to say that a kernel is a real-time kernel or not. To do it we have to talk about the **determinism**, it means that a kernel is able to warranty the time necessary to switch from one task to another one is a constant.
- With PICos18 the time (**latency time**) is 50 µs then each time the task-0 sends an event to the task-1 the time to switch from task-1 to task-0 is 50 µs.
- With 'Round Robin' algorithm the idle tasks (tasks with the lowest priority) can share the PIC18 processor during a time slice of 1ms if they have the equal priority.

## INTERRUPTS



## INTERRUPTS

- There are 2 types of interrupts in PICos18:
  - ✔ Slow Interrupts
  - ✔ Fast Interrupts
- In fast interrupts, you can implement very simple code since it is fast, it can not handle long codes.
- Generally, you need slow interrupts.

## Serial Communication

- There is a EUSART module in PIC18F4620, capable of asynchronous/synchronous communication.
- We will focus on asynchronous mode today.
- **We will also talk about some PICSimLab intricacies about asynchronous serial communication!**

## Serial Communication

Important Registers:

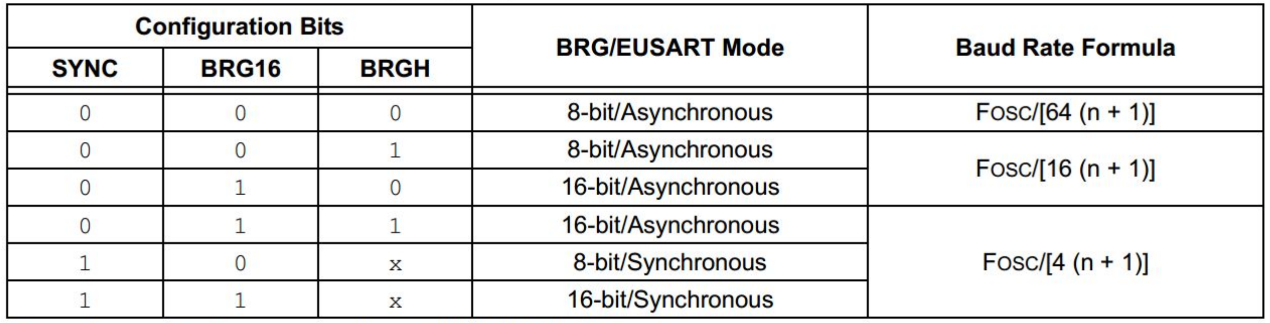
- RCSTA for reception, RCSTA.SPEN configures RX and TX as serial port pins. RCSTA.CREN enables reception.
- TXSTA for transmission. TXSTA.BRGH for setting the baudrate. TXSTA.TXEN to enable transmission. TXSTA.TRMT to poll if TXREG is transmitted (don’t use!).
- SPBRGH:SPBRG, BAUDCON for setting the baudrate.
- IPR1, PIE1, PIR1 for interrupt enable, priority and flag bits.
- INTCON.GIE, INTCON.PEIE for enabling global, peripheral interrupts.
- Lastly TRISC registers because RC6 is TX and RC7 is the RX pin.

## Initial Configurations

We will configure EUSART1 for asynchronous, 9600 bps 8-bit baudrate generator (high-speed mode), for a 10MHz crystal.

- Asynchronous, TXSTA.SYNC = 0;
- 8-bit generator, BAUDCON.BRG16 = 0; (SPBRGH ignored)
- High-speed mode, TXSTA.BRGH = 1;
- For SPBRG, we would normally use the formula here:

**TABLE 18-1: BAUD RATE FORMULAS**

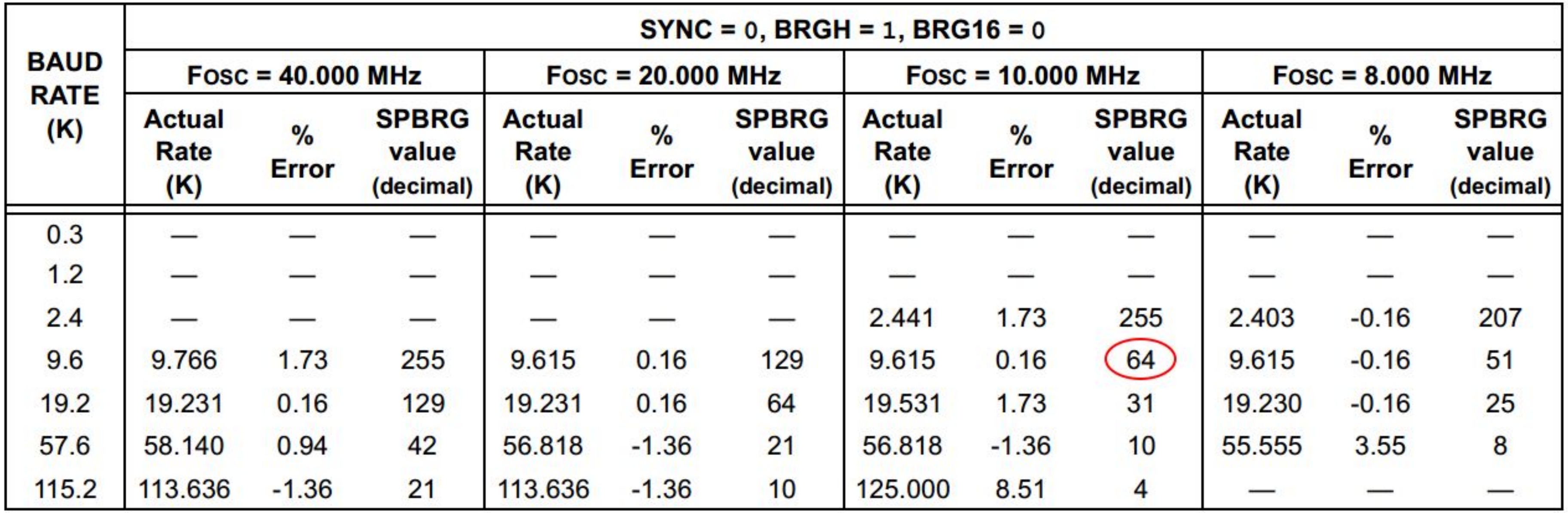


| Configuration Bits | | | BRG/EUSART Mode | Baud Rate Formula |
|---|---|---|---|---|
| SYNC | BRG16 | BRGH | | |
| 0 | 0 | 0 | 8-bit/Asynchronous | Fosc/[64 (n + 1)] |
| 0 | 0 | 1 | 8-bit/Asynchronous | Fosc/[16 (n + 1)] |
| 0 | 1 | 0 | 16-bit/Asynchronous | |
| 0 | 1 | 1 | 16-bit/Asynchronous | Fosc/[4 (n + 1)] |
| 1 | 0 | x | 8-bit/Synchronous | |
| 1 | 1 | x | 16-bit/Synchronous | |

**Legend:** x = Don't care, n = value of SPBRGH:SPBRG register pair

## Initial Configurations

- ...but it is already on the table as it is a common baudrate!
- SPBRG = 64;



| BAUD RATE (K) | SYNC = 0, BRGH = 1, BRG16 = 0 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Fosc = 40.000 MHz | | | Fosc = 20.000 MHz | | | Fosc = 10.000 MHz | | | Fosc = 8.000 MHz | | |
| | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) |
| 0.3 | — | — | — | — | — | — | — | — | — | — | — | — |
| 1.2 | — | — | — | — | — | — | — | — | — | — | — | — |
| 2.4 | — | — | — | — | — | — | 2.441 | 1.73 | 255 | 2.403 | -0.16 | 207 |
| 9.6 | 9.766 | 1.73 | 255 | 9.615 | 0.16 | 129 | 9.615 | 0.16 | [✓ 64] | 9.615 | -0.16 | 51 |
| 19.2 | 19.231 | 0.16 | 129 | 19.231 | 0.16 | 64 | 19.531 | 1.73 | 31 | 19.230 | -0.16 | 25 |
| 57.6 | 58.140 | 0.94 | 42 | 56.818 | -1.36 | 21 | 56.818 | -1.36 | 10 | 55.555 | 3.55 | 8 |
| 115.2 | 113.636 | -1.36 | 21 | 113.636 | -1.36 | 10 | 125.000 | 8.51 | 4 | — | — | — |

## Initial Configurations

- RCSTA.SPEN = 1; to set RC6/7 as serial comm. pins.
- PIE1.TXIE = 1; PIE1.RCIE = 1; to enable interrupts for transmission/reception. PIE1.TXIP = 0; PIE1.RCIP = 0; to make them low priority (high priority interrupts might cause trouble with PICOS).
- TRISC.RC6 = 0; TRISC.RC7 = 1; for pin I/O.
- INTCON.GIE = 1; INTCON.PEIE = 1; for enabling interrupts in general and also peripheral interrupts.

## Transmission

Once the TXREG register transfers the data to the TSR register (occurs in one TCY), the TXREG register is empty and the TXIF flag bit (PIR1<4>) is set. This interrupt can be enabled or disabled by setting or clearing the interrupt enable bit, TXIE (PIE1<4>). TXIF will be set regardless of the state of TXIE; it cannot be cleared in software. TXIF is also not cleared immediately upon loading TXREG, but becomes valid in the second instruction cycle following the load instruction. Polling TXIF immediately following a load of TXREG will return invalid results.

**Note 1:** The TSR register is not mapped in data memory so it is not available to the user.

**2:** Flag bit, TXIF, is set when enable bit, TXEN, is set.

To set up an Asynchronous Transmission:

1. Initialize the SPBRGH:SPBRG registers for the appropriate baud rate. Set or clear the BRGH and BRG16 bits, as required, to achieve the desired baud rate.
2. Enable the asynchronous serial port by clearing bit, SYNC, and setting bit, SPEN.
3. If interrupts are desired, set enable bit, TXIE.
4. If 9-bit transmission is desired, set transmit bit, TX9. Can be used as address/data bit.
5. Enable the transmission by setting bit, TXEN, which will also set bit, TXIF.
6. If 9-bit transmission is selected, the ninth bit should be loaded in bit, TX9D.
7. Load data to the TXREG register (starts transmission).
8. If using interrupts, ensure that the GIE and PEIE bits in the INTCON register (INTCON<7:6>) are set.

## Transmission

1. Whenever you want to send a character, set TXSTA.TXEN = 1;
2. This will trigger the interrupt (as TXREG is initially empty). Load your character to TXREG, in ISR.
3. For any regular interrupt, you want to clear it after everything is done. But PIR1.TXIF cannot be cleared in software. There are two ways of keeping it from triggering a meaningless interrupt again (done inside ISR):
    a. Loading a value to TXREG, which we did.
    b. setting TXSTA.TXEN = 0; i.e. terminating the transmission.
4. After the char is sent, the interrupt will trigger again. If this was the last char you wanted to send, do 3b in ISR. Otherwise load the next character to TXREG (3a - in ISR).

## Transmission in PICSimLab - (Bug?)

1. ~~Whenever you want to send a character, set TXSTA.TXEN = 1;~~
2. ~~This will not trigger the interrupt for your very first transmission! In addition to this, you need to load your character to TXREG outside the interrupt (before setting TXSTA.TXEN)!~~
3. ~~After this character is sent, the interrupt will be triggered. Clear the flag of the interrupt just like any other regular interrupt you need to handle after everything is done (very different from what datasheet says)! Inside ISR, two options:~~
    a. ~~If there is another char you want to send, load TXREG with this char.~~
    b. ~~If there is no more char you want to send, terminate it by setting TXSTA.TXEN = 0;~~

## Transmission in PICSimLab - (More Bugs?)

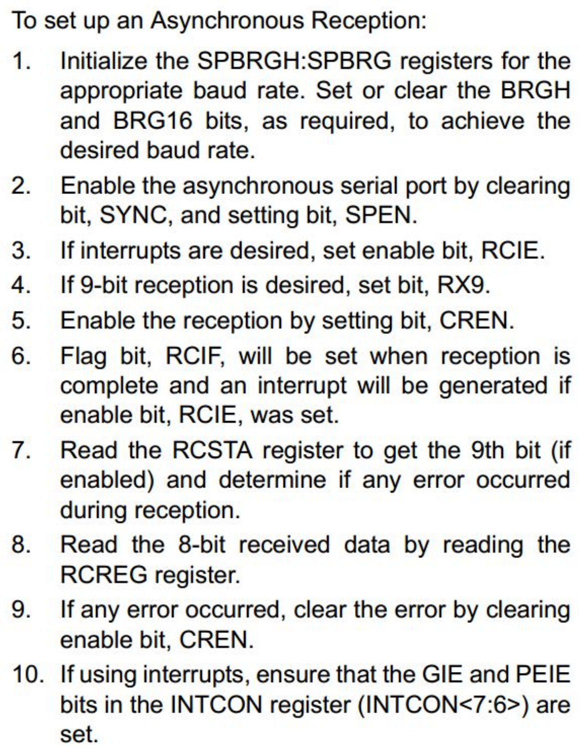
- Whenever you want to send a character, just load TXREG. That would not work for a real PIC18F4620, but it does for picsim (according to its source code as far as I understand)
- The last character loaded will be pushed to virtual serial port (TRMT bit will be 0 during this time).
- This takes a few clock cycles, however. That means you can’t get away with while looping TXREG loads or code bloating TXREG=’h’; TXREG=’i’; … (picsim refuses to push chars to virtual serial port if TRMT is 0)

## Transmission in PICSimLab - (More Bugs?)

- You will need to use the ISR to send a new char when TXREG content is pushed to virtual serial port. Two options in the ISR:
    a. Load TXREG with the next char you are going to send.
    b. Do nothing. <u>The last character loaded will not be sent again</u>.
- <u>It doesn’t care about TXEN bit.</u> It seems to auto clear TXIF. It definitely stops when SPEN is unset.

## Reception

- Set RCSTA.CREN = 1; to start receiving.
- RCIF will fire as soon as you receive a byte.
- The contents will be on RCREG. Copy them somewhere and use them.
- Sometimes errors might happen (RCSTA.OERR, RCSTA.FERR)
  - Unset and then set RCSTA.CREN to keep receiving.
- Do not forget to clear the interrupt flag.



To set up an Asynchronous Reception:

1. Initialize the SPBRGH:SPBRG registers for the appropriate baud rate. Set or clear the BRGH and BRG16 bits, as required, to achieve the desired baud rate.
2. Enable the asynchronous serial port by clearing bit, SYNC, and setting bit, SPEN.
3. If interrupts are desired, set enable bit, RCIE.
4. If 9-bit reception is desired, set bit, RX9.
5. Enable the reception by setting bit, CREN.
6. Flag bit, RCIF, will be set when reception is complete and an interrupt will be generated if enable bit, RCIE, was set.
7. Read the RCSTA register to get the 9th bit (if enabled) and determine if any error occurred during reception.
8. Read the 8-bit received data by reading the RCREG register.
9. If any error occurred, clear the error by clearing enable bit, CREN.
10. If using interrupts, ensure that the GIE and PEIE bits in the INTCON register (INTCON<7:6>) are set.

## Demo