Polled IO versus Interrupt Driven IO

- Polled Input/Output (IO) – processor continually checks IO device to see if it is ready for data transfer
  - Inefficient, processor wastes time checking for ready condition
  - Either checks too often or not often enough
- Interrupt Driven IO – IO device interrupts processor when it is ready for data transfer
  - Processor can be doing other tasks while waiting for last data transfer to complete – very efficient.
  - All IO in modern computers is interrupt driven.

Polled IO: getch()

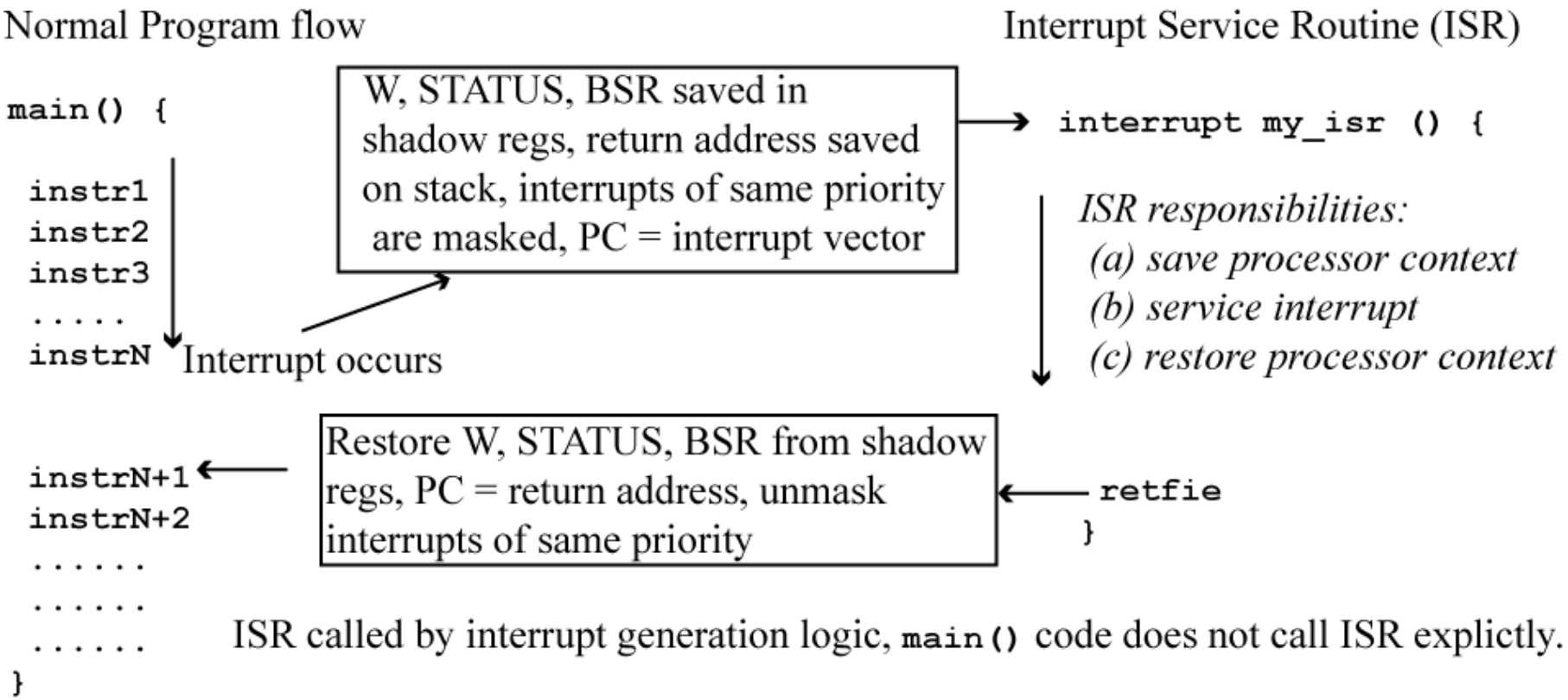
```
/* return 8 bit char
from Receive port */

unsigned char getch ()
{
  unsigned char c;
  /* wait until character is received */
  while (!RCIF);
  c = RCREG;
  return(c);
}
```

This is **polled** IO. Note that the processor is continually polling to see if data is available.

Time spent checking for data availability is WASTED time. In some applications this time can be spent doing something else.

Interrupt-driven IO on the PIC18



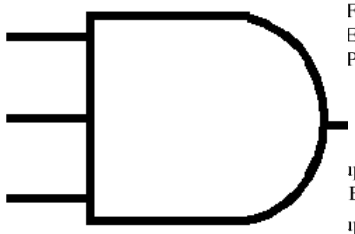
The normal program flow (main) is referred to as the foreground code. The interrupt service routine (ISR) is referred to as the background code.

PIC18Fxx2 Interrupts (Chap 8 of datasheet)

- Many PIC18Fxx2 interrupt sources
  - When character is received on serial port
  - When character is finished transmitting on serial port
  - When A/D conversion is finished
  - When external pin RB0 is pulled low
  - Many more
- Interrupt Enable, Flag bits
  - Each interrupt source has an **ENABLE** bit that allows an interrupt to be generated if interrupt condition is met. By default, interrupts are NOT enabled.
  - Each interrupt source also has a **FLAG** bit that indicates if the interrupt has occurred.
  - Each interrupt source also has a **PRIORITY** bit that allows it to be assigned a low or high priority.

Interrupt Logic

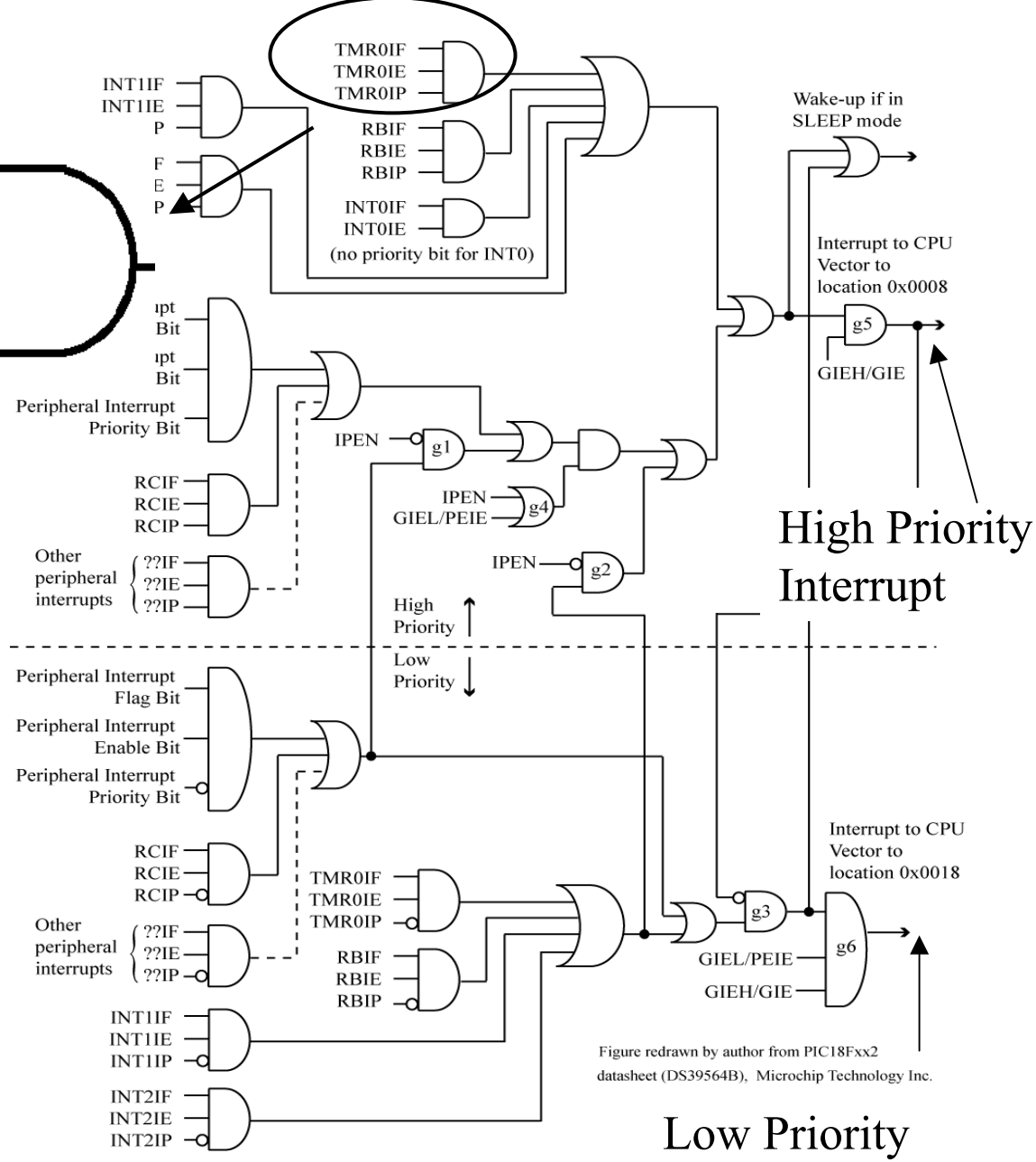
TMR0IF
TMR0IE
TMR0IP



TMR0IF : flag bit, ‘1’ indicates event is true.

TMR0IE : enable bit, must be ‘1’ for interrupt to be generated.

TMR0IP : priority bit, ‘1’ high priority, ‘0’ low priority.



Interrupt Priorities

- When an interrupt occurs, processor finishes current instruction, and calls an Interrupt Service Routine (ISR)
  - High priority interrupt routine at location 0x0008
  - Low priority interrupt routine at location 0x0018
  - An interrupt service routine should determine source of interrupt, and service it (i.e, do the interrupt function).
- A high priority interrupt can interrupt the ISR of a low priority interrupt, but not vice versa
  - A high priority interrupt cannot interrupt (should not) interrupt itself.

Interrupt Priorities (cont)

- Interrupt priorities can be disabled by setting the IPEN bit (RCON[7]) to ‘0’.
  - All priorities are classified as high priority (interrupt jumps to 0x0008).
- This is the mode that will be used for lab and class
  - This is compatible with the PIC16 family of micros
  - We have no need for priorities in our lab exercises.

GIE, PEIE

- Global interrupt enable (GIE, INTCON[7]) can be used to disable all interrupts
  - By default, all interrupts disabled
- Peripheral interrupt enable (PEIE, INTCON[6]) can be used to disable all peripheral interrupts
  - By default, all peripheral interrupts disabled
  - Peripheral interrupts are those associated with peripheral subsystems such as the USART, the Analog/Digital converter, timers, etc.
- If priorities are enabled,
  - the GIE bit is known as GIEH (GIE for high priority interrupts)
  - PEIE bit is known as GIEL (GIE for low priority interrupts).

When an enabled interrupt occurs...

- GIE bit is CLEARED – this disables further interrupts (do not want to get caught in an infinite loop!)
  - If priorities are enabled, either GIEH or GIEL is cleared, depending on interrupt priority
- Return address is pushed on the stack
- W, STATUS, BSR saved in shadow registers
- Jump to 0x0008 (high priority) or 0x0018 low priority interrupt is done

ISR Responsibilities

- Must save the processor *context* (W register, and Status Register, and BSR if necessary)
  - If not saved, normal program execution will become unpredictable since interrupt can happen at anytime
  - If ISR uses registers such as FSRx or PRODH/PRODL, must save these also!!!!!
- Must determine the source of the interrupt
  - If multiple interrupts are enabled, check flag bit status
- Must ***service*** the interrupt (clear interrupt flag)
  - E.g., for received character interrupt, read the RCREG, this clears the RCIF bit automatically.
- Restore processor context
- Execute RETFIE (return from interrupt)
  - Sets GIE to enable interrupts, reads PC from stack

Shadow Registers

- An interrupt can occur at any point in the execution of program.
- The ISR will, at a minimum, change W and STATUS, and probably the BSR
  - These must be saved on ISR entry, and restored on ISR exit
- Shadow registers were added with the PIC18
  - W, STATUS, and BSR are registers are saved automatically on interrupt
  - Will be restored on exit by the RETFIE (return from interrupt) if the ‘s’ bit is set in the instruction word - the PICC18 C compiler uses ‘RETFIE 1’ which restores the W, STATUS, BSR on return.
- Shadow registers can only be used with high priority interrupts
  - If low priority interrupt uses shadow registers, then can be overwritten by high priority interrupt

ISR Assembly

```
      CBLOCK 0x7D
      w_temp, status_temp, bsr_temp
      endc
      org 0x008
       goto  isr_high_priority
      org 0x0018
       goto  isr_low_priority
      org 0x????
isr_high_priority
      ;;; ISR high priority code
      retfie 1  ;; use shadow reg

isr_low_priority
      movwf w_temp          ; context
      movff  STATUS,status_temp
      movff BSR, bsr_temp
      ;;....ISR CODE ...
      ;;.... . . .
      movff bsr_temp,bsr  ; restore context
      movf  w_temp,w
      movff  status_temp,STATUS
      retfie
```

space for w, status, bsr

high priority ISR can use shadow registers (fast stack)

low priority must explicitly save the processor context

restore context, do not use shadow registers on ‘retfie’

PIC18 ISR in *C*

```
volatile unsigned int got_char_flag;
volatile unsigned char received_char;

// interrupt service routine
void interrupt
pic_isr(void)
{
  // see if this interrupt was
  // generated by a receive character
   if (RCIF) {
     // reading RCREG clears interrupt bit
    received_char = RCREG;
    got_char_flag = 1;
  }
}
```

Use `volatile` qualifier for any variables modified within ISR;
notifies compiler that variable can be modified between accesses.

`interrupt` qualifier for function notifies the compiler that this function is an ISR (high priority assumed).

If receive character interrupt, read character, save it, set flag to signal `main()` that interrupt occurred

Do not have to clear RCIF as it is cleared automatically by reading RCREG.

Read character, set **semaphore** got_char_flag for main() function

Enabling Interrupts in C

```
// code for serial port setup not shown
// enable interrupts
  IPEN = 0;    // disable priorities IPEN = RCON(7)
  RCIE = 1;    // enable receive int, RCIE = PIE1(5)
  PEIE = 1;  // enable peripheral intr, PEIE = INTCON(6)
  GIE = 1;  // enable global intr, GIE = INTCON(7)


while(1) {
    // wait for interrupt
    while (!got_char_flag);
    c = received_char;
    got_char_flag =0; // clear flag
    c++;  // increment Char
    putch (c);  // send the char

  }
```

Set by interrupt service routine – this is a ***semaphore***.

Get character read by interrupt service routine

What is a semaphore?

The main() code (i.e. foreground code) must be signaled when an interrupt action has occurred.

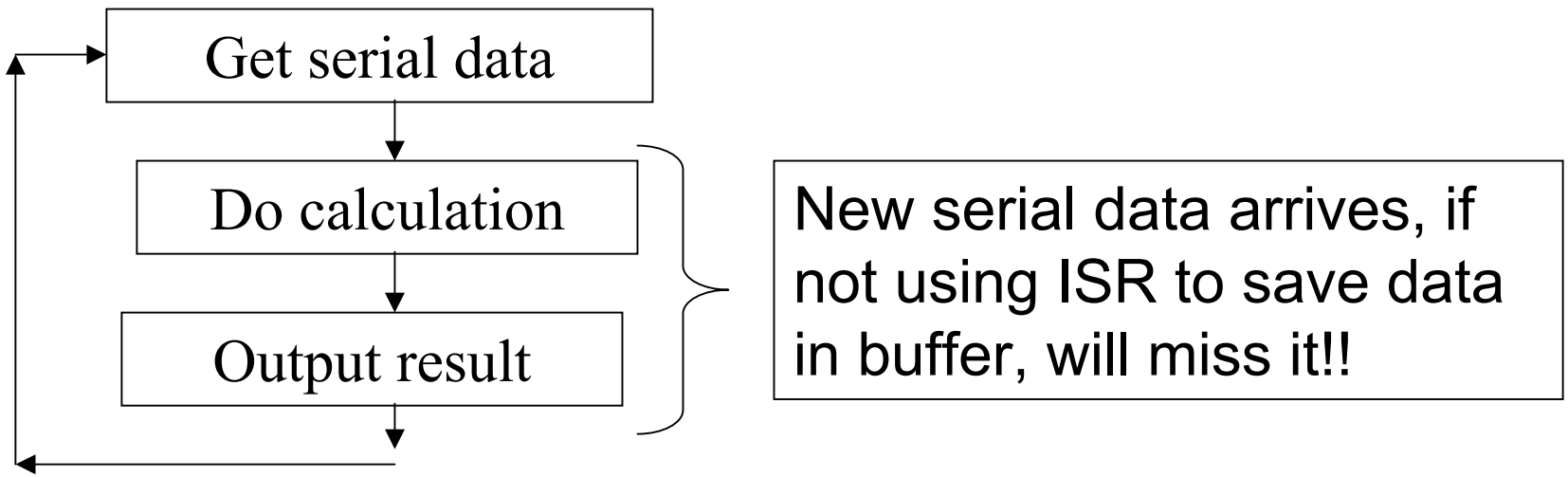
The isr() code (i.e, background code) must perform the IO action that is caused by the interrupt. After servicing the IO, it must signal the main() code via some variable to indicate that the interrupt occurred.

A variable used by an ISR to signal to main() code that an interrupt has occurred is called a ***semaphore***.

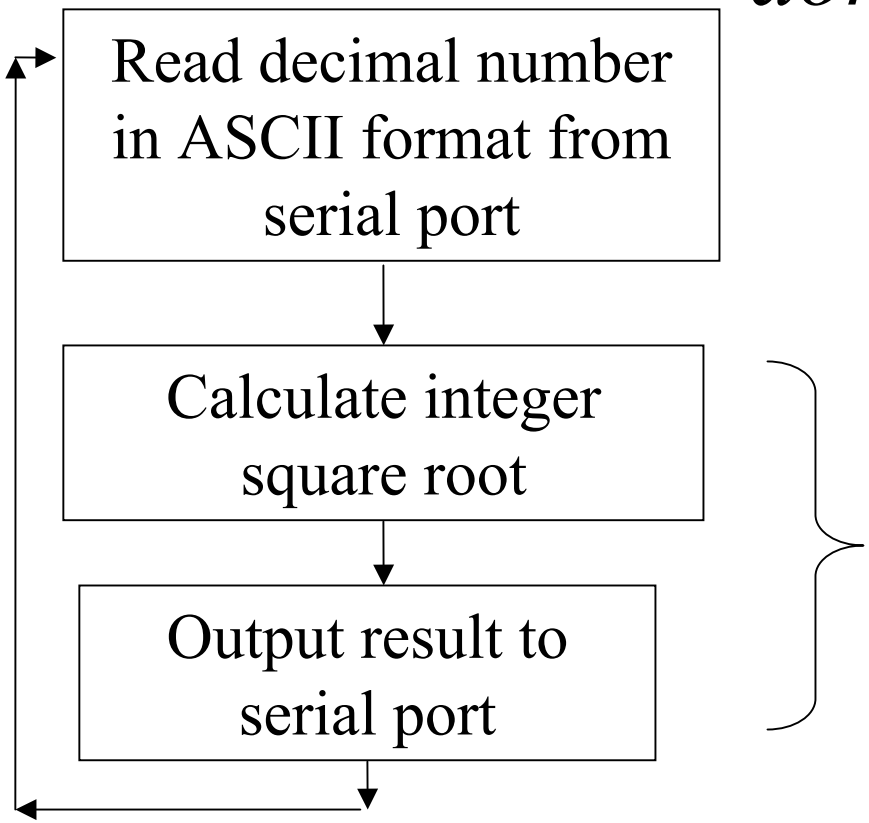
Soooooo…when are interrupts useful?

Previous example simply illustrated the use of interrupts for reading serial data. Interrupt usage was not really needed since main routine just waited for data to arrive so no advantage over polled IO.

Interrupts for serial IO useful when cannot poll serial port often enough!



doroot.c



New data can arrive. Current ISR only has room to save ONE character. Depending on baud rate, can have overrun error (input FIFO fills up before we read data again).

While the result is printing, more data is being sent, and the data is lost unless buffered.

`'The square root of ?? is ??'`
ASCII result stream

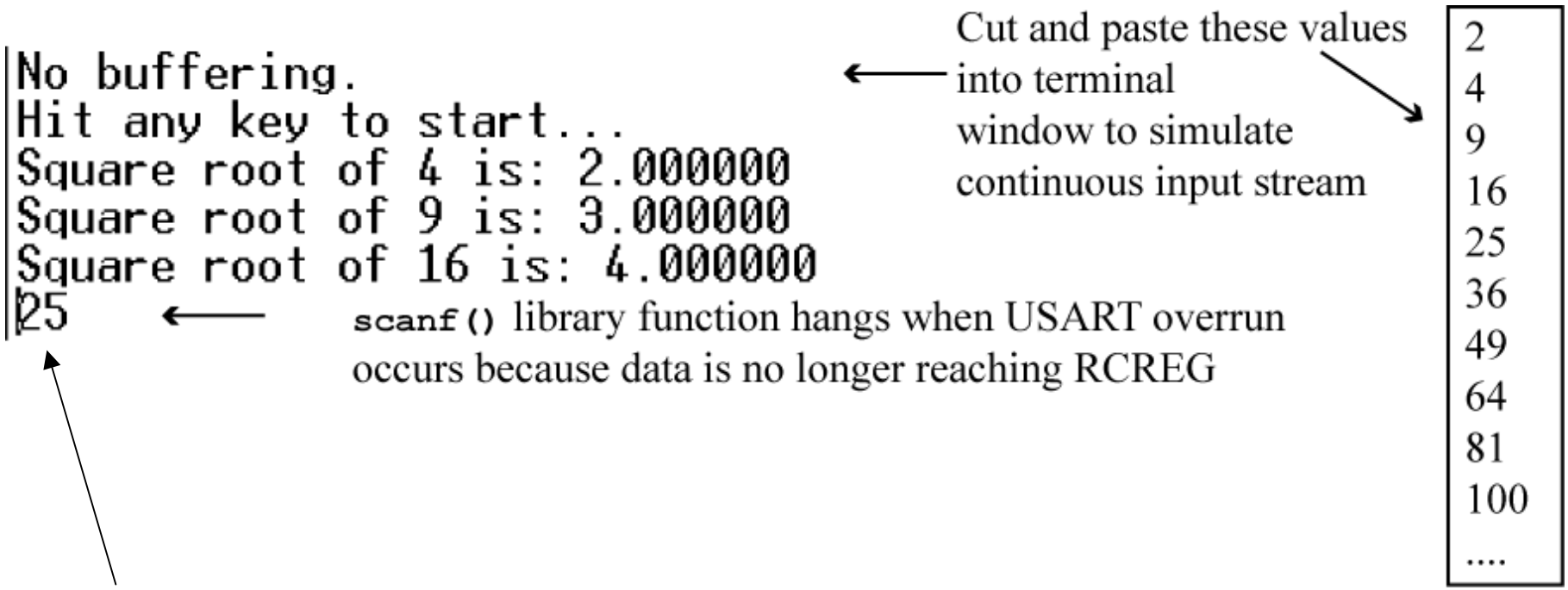
RS232 Serial link

`'2','4','9','16','25','36', ...`
ASCII-decimal input stream

PIC18F242

```
main(){
  while(1) {
    // read ASCII-decimal number
   // compute square root
   // print square root
  }
```

Data lost when no buffering is done



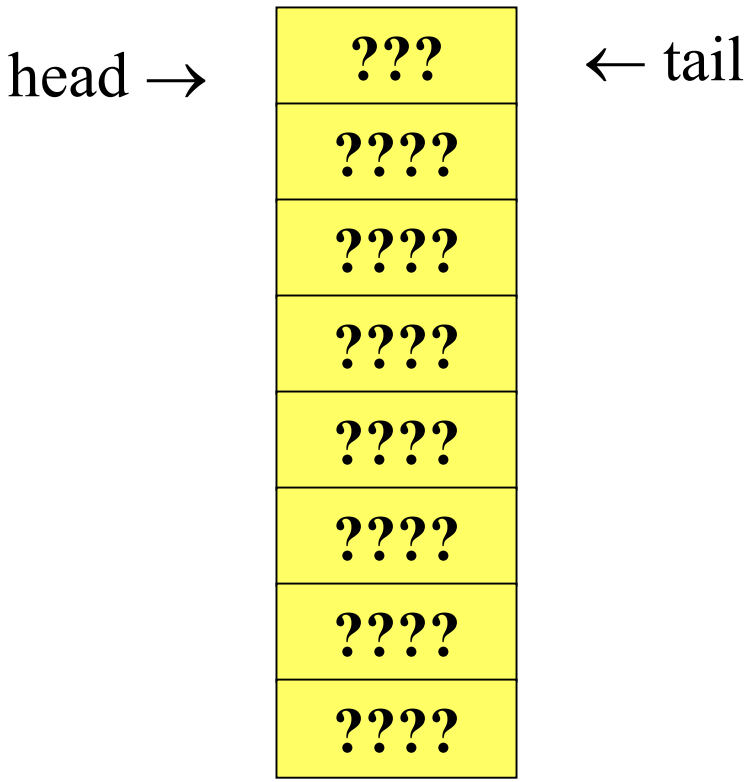
USART overrun occurs, application hangs.

FIFO Buffering of Data for Interrupt IO

- A *circular buffer* is most often used to handle interrupt driven INPUT.
- A circular buffer requires the following pointers
  - base address of memory buffer
  - head index (head pointer)
  - tail index (tail pointer)
  - size of buffer
- A circular buffer is simply another name for a *FIFO (First-In-First-Out)* buffer.
  - The name *circular buffer* helps to visualize the wraparound conditon

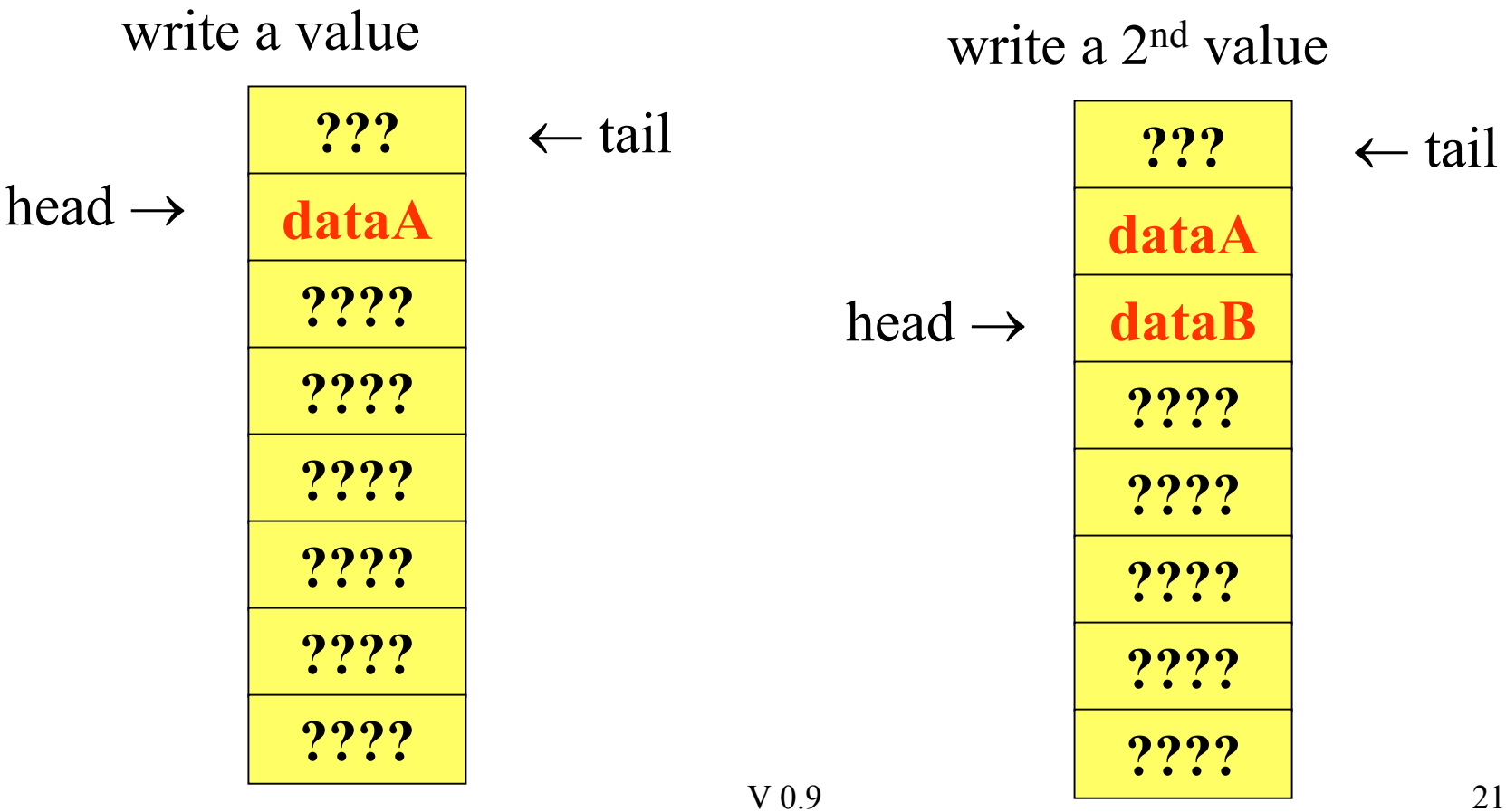
Circular buffer, 8 locations long

When buffer is empty, head = tail index



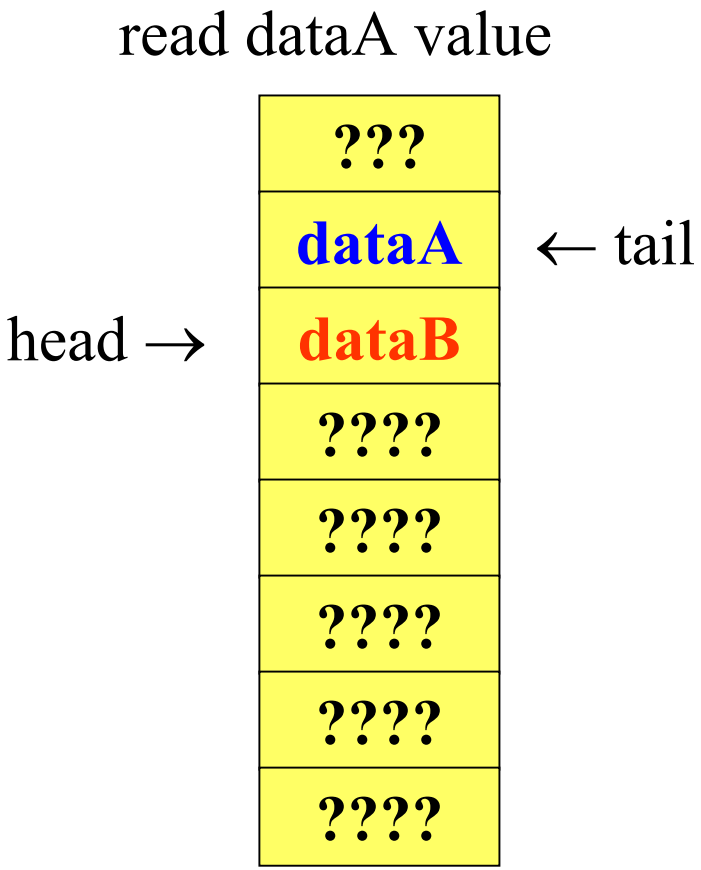
Circular buffer, write operation

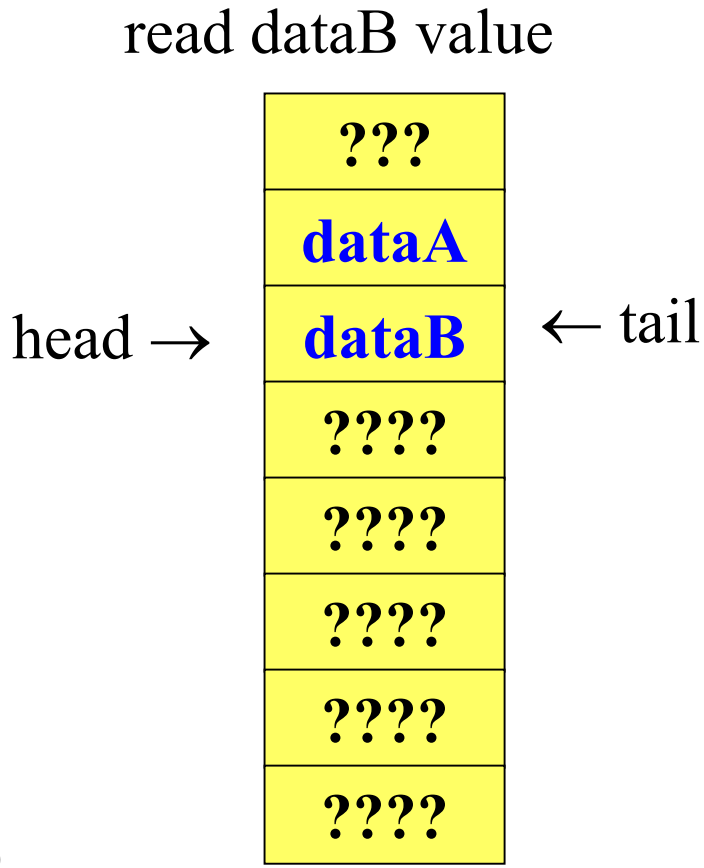
Interrupt service routine places items in memory buffer by incrementing head index, then storing value



Circular buffer, read operation

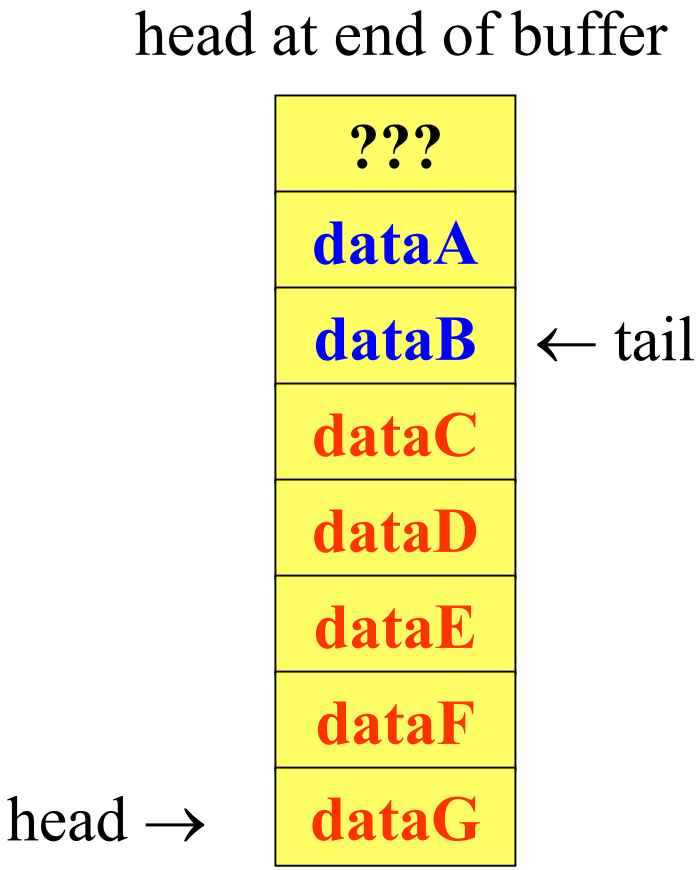
Input function occasionally checks to see if head not equal to tail, if true, then read value by incrementing tail, then reading memory.

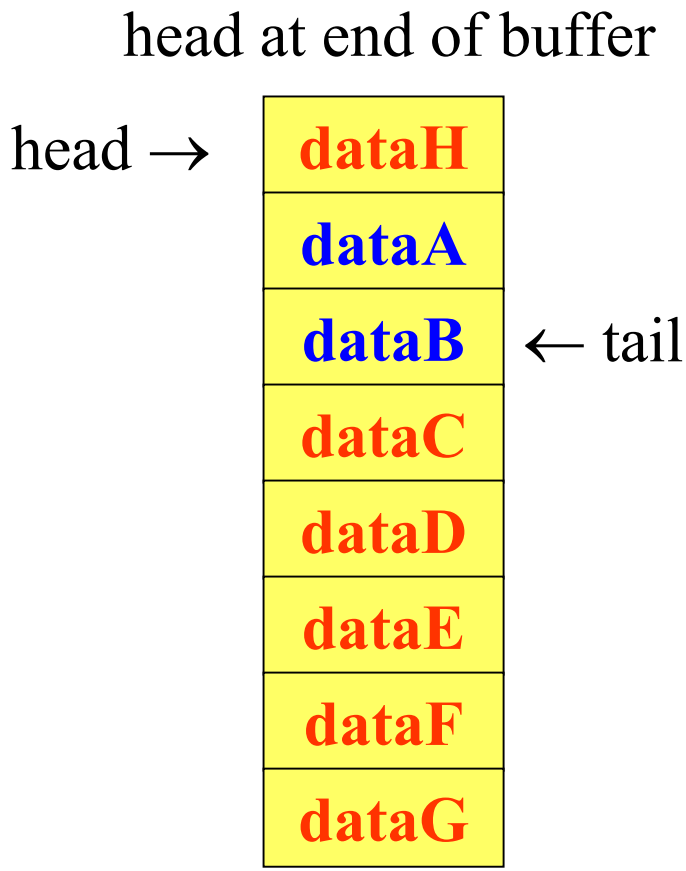




Circular buffer, wraparound

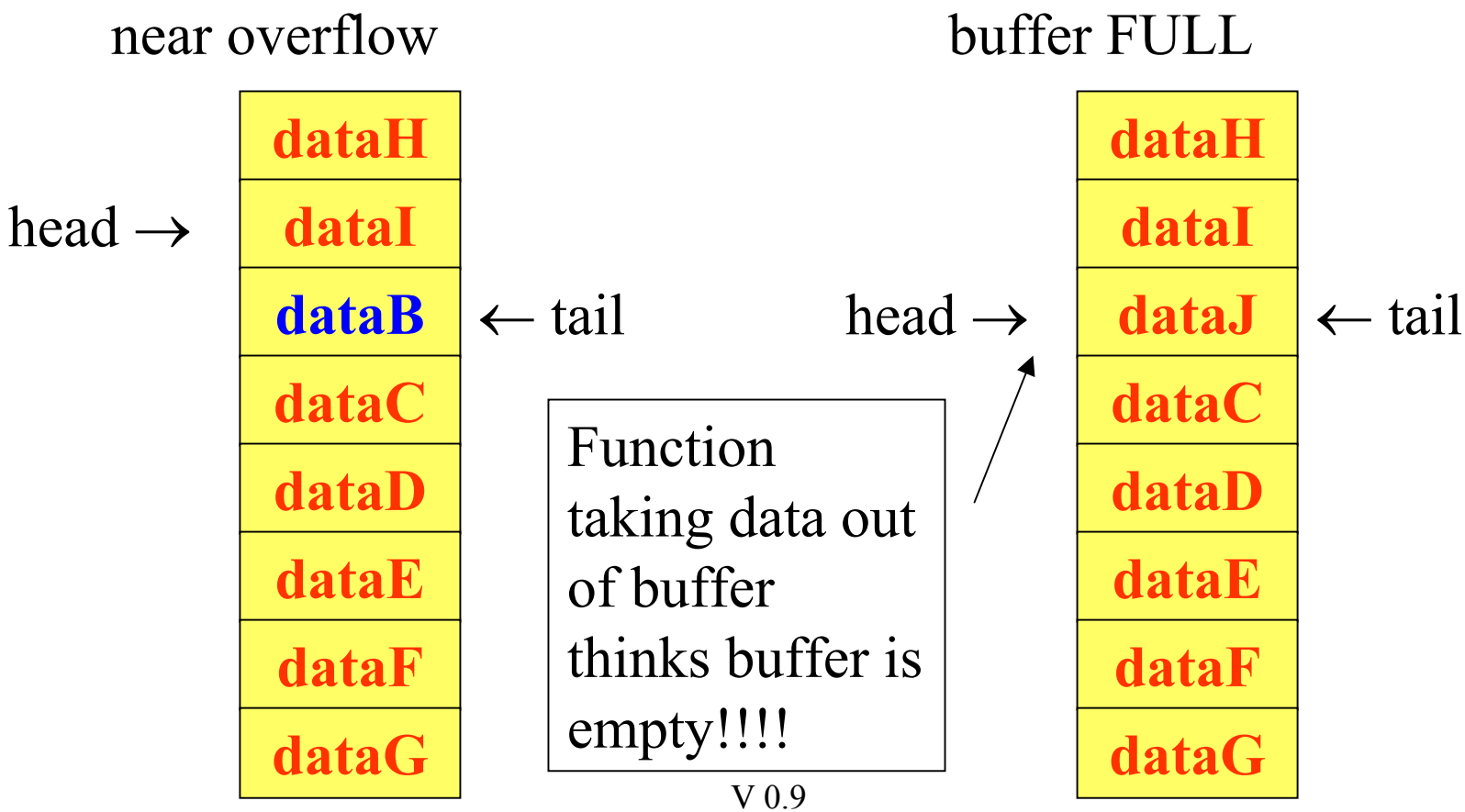
when head pointer gets to end of buffer, set back to top of buffer (wraparound)





Circular buffer, buffer FULL

buffer FULL occurs if interrupt service routines increments head pointer to place new data, and head = tail!!!!



How to pick size of circular buffer?

- Must be big enough so that buffer full condition never occurs
- Routine that is taking data out of buffer must check it often enough to ensure that buffer full condition does not occur.
  - If buffer fills up because not checking often enough, then increase the size of the buffer
  - No matter how large buffer is, must periodically read the data.
- Buffer must be big enough so that bursts of data into buffer does not cause buffer full condition.

ISR for Interrupt Driven Serial IO

```
#define BUFSIZE 2
unsigned char ibuf[BUFSIZE];
unsigned char head,tail;

void interrupt pic_isr(void){

  // see if this interrupt was generated by
  // receive character
  if (RCIF) {  // check RCIF bit
    head = head + 1;
    if (head == BUFMAX) head = 0;
    // reading this register clears interrupt bit
    ibuf[head] = RCREG;
  }
}
```

triggered when serial data arrives. Save in buffer.

Increasing buffer size will increase amount of time that *main{}* code can wait before reading input buff.

New *getch()* for Interrupt Receive

```
unsigned char getch (void){
  unsigned char c;
  while (head == tail) {
    asm("clrwdt");
    };
  tail = tail + 1;
  if (tail == BUFMAX) tail = 0;
  c = ibuf[tail];
  return(c);
}
```

Wait for ISR to trigger and save data in buffer. Combination of **head/tail** is the **semaphore**!!!

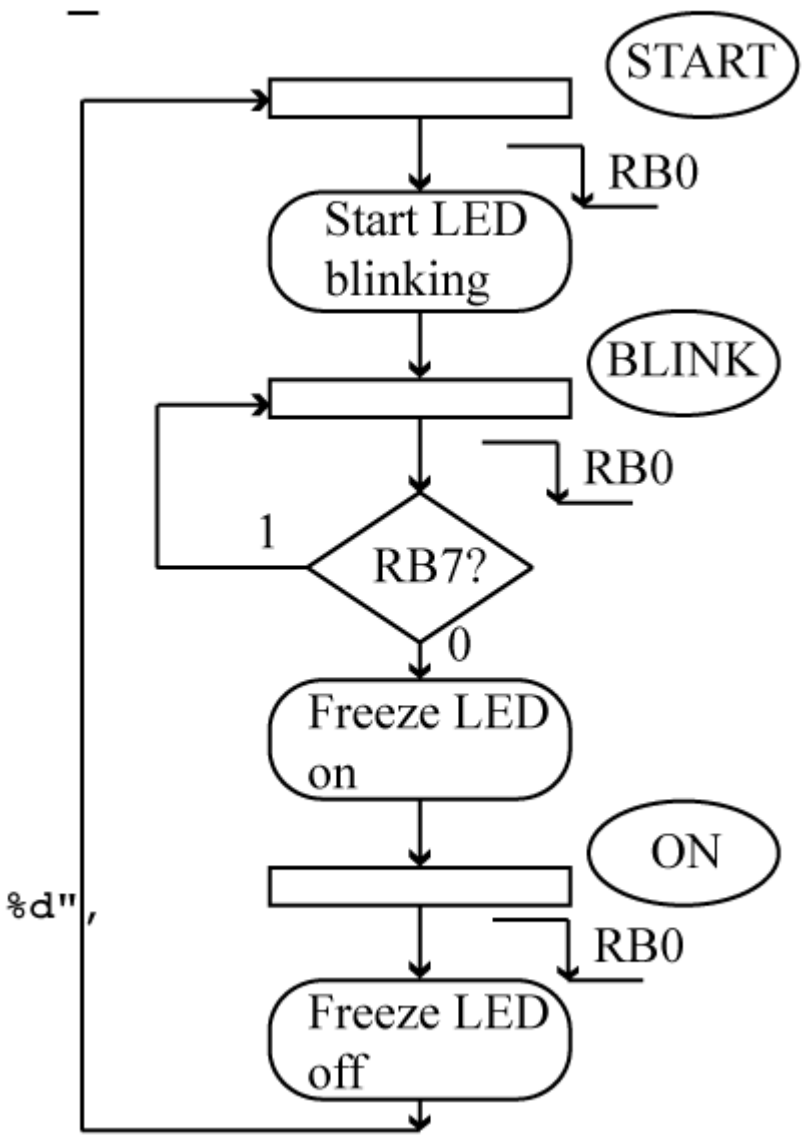
Must wrap tail pointer if at end of buffer.

Read data from buffer

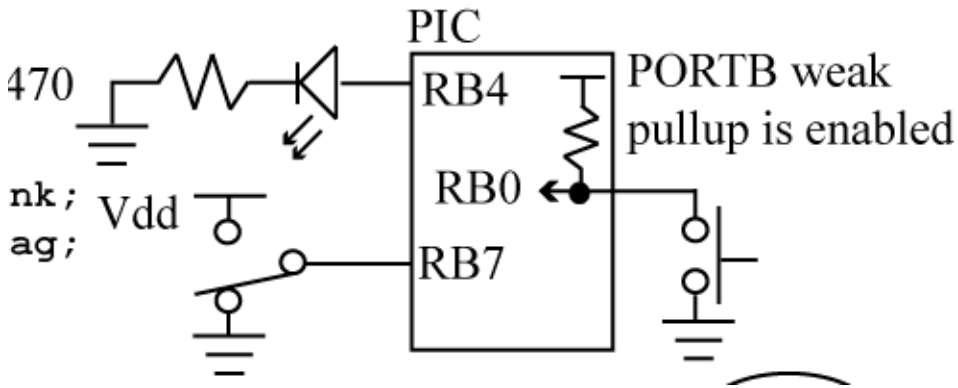
INT0/INT1/INT2 Interrupts

- The RB0/RB1/RB2 inputs can generate an interrupt on either a rising or falling edge.
- These are called the INT0/INT1/INT2 interrupts.
- The INTEDG0/INTEDG1/INTEDG2 bits determine the active edge
  - ‘0’ for falling edge, ‘1’ for rising edge
- These interrupts can wake the processor from sleep mode.

Interrupt Driven LED/Switch IO



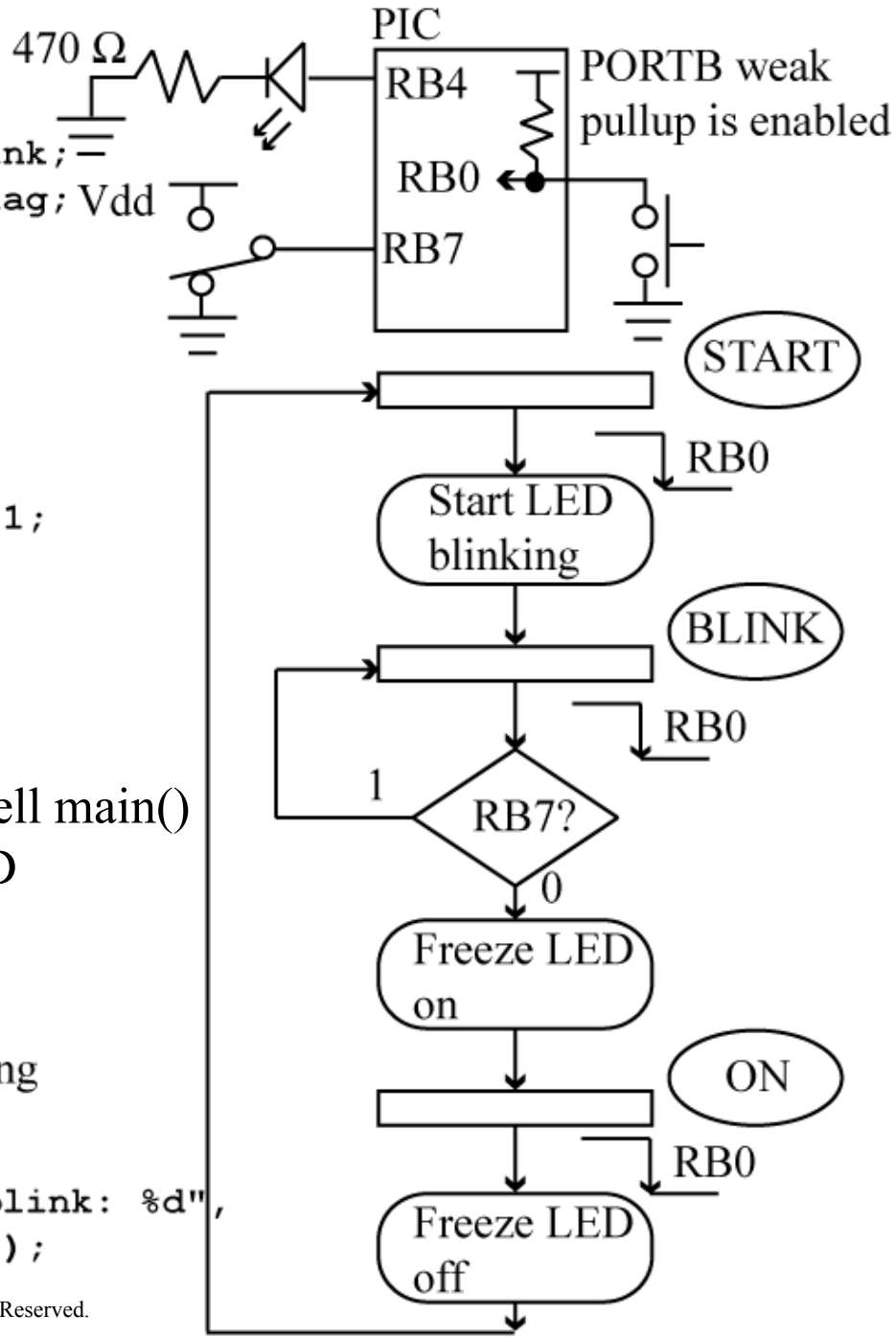
implement this with interrupt driven IO



```
#define START_STATE 0
#define BLINK_STATE 1
#define ON_STATE 2
volatile unsigned char state, led_blink;
volatile unsigned char led_on, int_flag;
void interrupt pic_isr(void){
 if (INT0IF) {
  ISR_DelayMs(30); //debounce
  INT0IF = 0; int_flag = 1;
  switch(state) {
   case START_STATE:
    state = BLINK_STATE; led_blink = 1;
    break;
   case  BLINK_STATE:
    if (!RB7) {
     led_blink = 0; led_on = 1;
     state = ON_STATE;
    }
    break;
   case ON_STATE:
    led_on = 0; state = START_STATE;
    break;
  }
 }
}
void print_debug(void){
 printf("State: %d, Led_on: %d, led_blink: %d",
      state,led_on,led_blink); pcrlf();
}
main(void){
```

led_blink, led_on tell main() how to control LED

print state for debugging



```
main(void){
  serial_init(95,1);// 19200 in HSPLL mode, crystal = 7.3728 MHz
  RBPU = 0; // weak pullups enabled
  // set RB0 for input, rising edge interrupt initially
  TRISB = 0xEF;  //RB4 is output, others inputs
  INTEDG0 = 0; // falling edge
  RB4 = 0; // turn LED off
  printf("INT0 FSM started (with debug).");pcrlf();
  // enable interrupts
  IPEN = 0; INT0IF = 0; INT0IE = 1; PEIE = 1; GIE = 1;
  print_debug();
  while(1) {
    if (int_flag) {
        print_debug(); int_flag = 0;
    }
    if (led_blink) {
      //LED toggle, delay
      if (LATB4) RB4 = 0; else RB4 = 1;
      DelayMs(250); DelayMs(250);
    }
    else if (led_on) RB4 = 1;
    else RB4 = 0;
  }
}
```

Configure ports, INT0 is falling edge trigerred

Enable interrupts

Loop is free-running, not synchronized to interrupt

Debug

Update LED based on `led_blink` and `led_on` semaphores

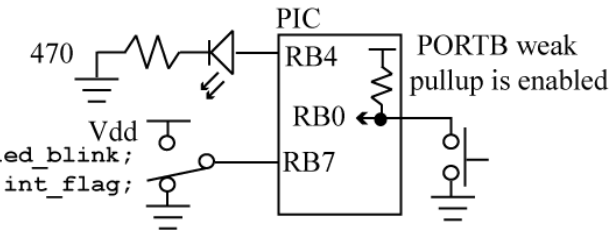
Semaphores *led_blink*, *led_on* control LED state in main() loop.

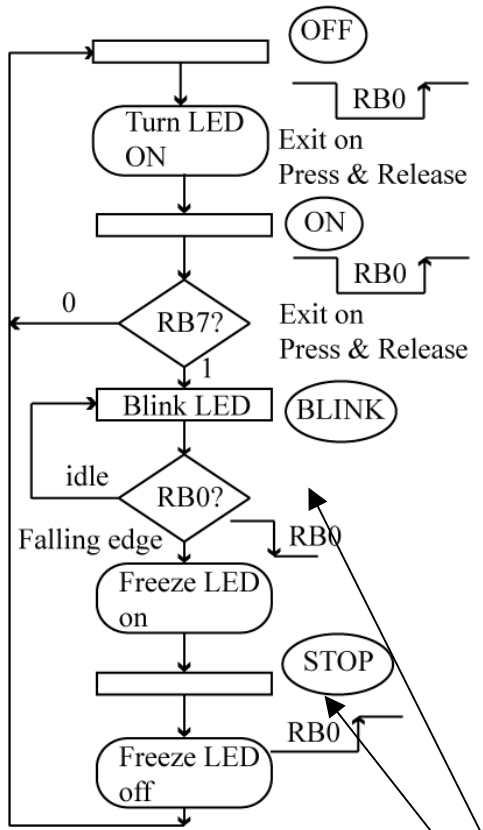
Interrupt Driven LED/Switch IO, example #2

```
#define OFF_STATE 0
#define ON_STATE 1
#define BLINK_STATE 2
#define STOP_STATE 3

volatile unsigned char state, led_blink;
volatile unsigned char led_on, int_flag;

void interrupt pic_isr(void){
 if (INT0IF) {
 INT0IF = 0; int_flag = 1;
 switch(state) {
  case OFF_STATE:
   // exited rising edge
   state = ON_STATE;
   led_on = 1; // turn on led
   break;
  case ON_STATE:
   // exited rising edge
   if (RB7) {
    // change to falling edge trigger
    INTEDG0 = 0;
    led_blink = 1; state = BLINK_STATE;
   }else {
    led_on = 0; state = OFF_STATE;
   }break;
   case  BLINK_STATE:
    // exited on falling edge
    led_blink = 0;led_on = 1;
    // change to rising edge trigger
    INTEDG0 = 1; state = STOP_STATE;
    break;
   case STOP_STATE:
    // exited on rising edge
    led_on = 0;state = OFF_STATE;
    break;
   }//end switch
 }// end if(INT0IF)
}//end
```





Remainder of code not shown, same as previous example except INT0 initialized for rising edge trigerred interrupt within `main()`.

Use Interrupt service routine to watch for falling/rising edges on RB0 input indicating a button press!!!!

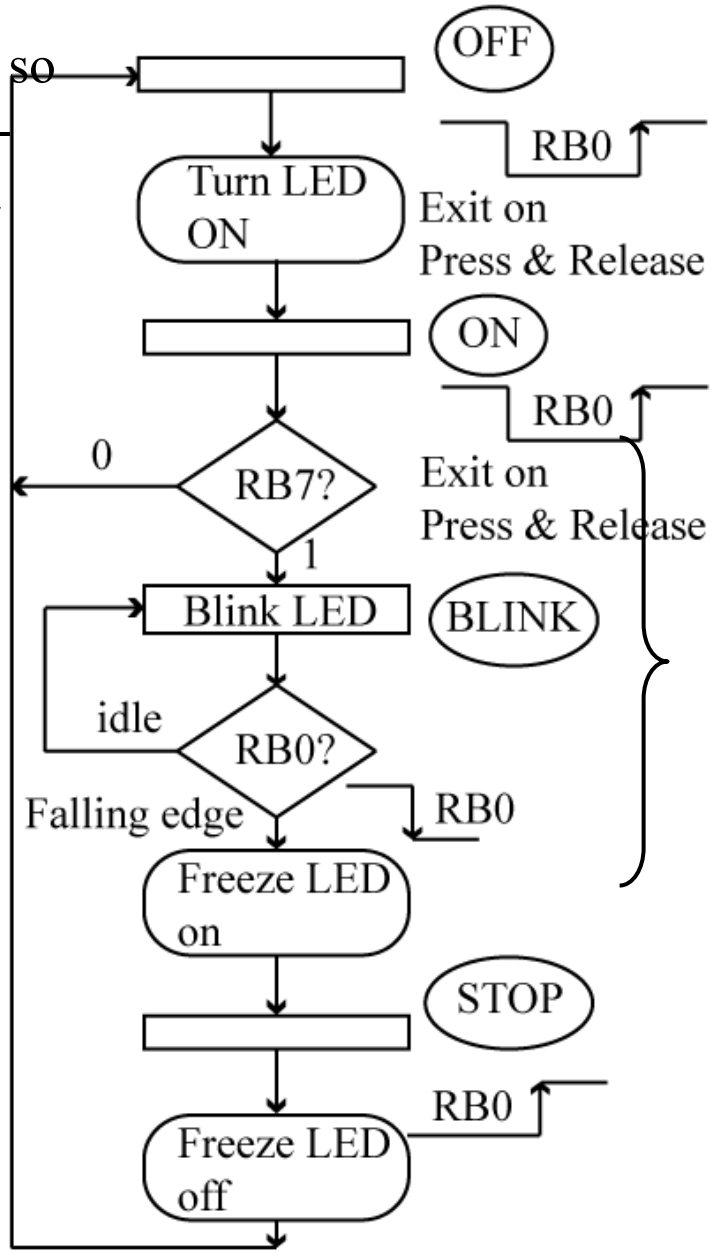
Use variables *led_on*, *led_blink* to tell *main()* that the LED should be turned on or blink.

Change active interrupt edge in ISR

```
void interrupt pic_isr(void){
 if (INT0IF) {
 INT0IF = 0; int_flag = 1;
 switch(state) {
  case OFF_STATE:
   // exited rising edge
   state = ON_STATE;
   led_on = 1; // turn on led
   break;
  case ON_STATE:
   // exited rising edge
   if (RB7) {
    // change to falling edge trigger
    INTEDG0 = 0;
    led_blink = 1; state = BLINK_STATE;
   }else {
    led_on = 0; state = OFF_STATE;
   }break;
   case  BLINK_STATE:
    // exited on falling edge
    led_blink = 0;led_on = 1;
    // change to rising edge trigger
    INTEDG0 = 1; state = STOP_STATE;
    break;
   case STOP_STATE:
    // exited on rising edge
    led_on = 0;state = OFF_STATE;
    break;
   }//end switch
 }// end if(INT0IF)
}//end
```

Initialize RB0 so initially rising-edge triggered

Active edge changed!!!



active interrupt edge is changed when go from ON to BLINK state

main() For Example #2

Basically the same as for example #1 except INT0 is configured for rising edge interrupt initially.

What do you have to know?

- How interrupts behave on the PIC18 for serial IO
- Function of PEIE, GIE bits
- Responsibilities of ISR
- Assembly language structure of ISR in PIC18
- ISR in PICC C
- Circular buffer operation
- Interrupt-driven LED/Switch IO