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()

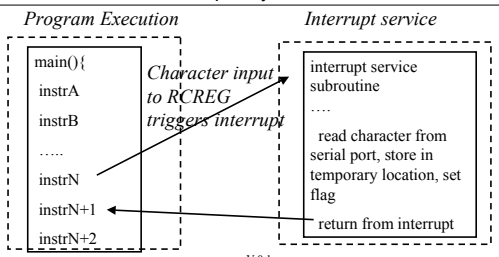
```
/* return 8 bit char
from Receive port */

unsigned char getch ()
{
  unsigned char c;
  /* wait until character is received */
  /* while (!RCIF) */
  while (!bittst(PIR1,5));
  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.

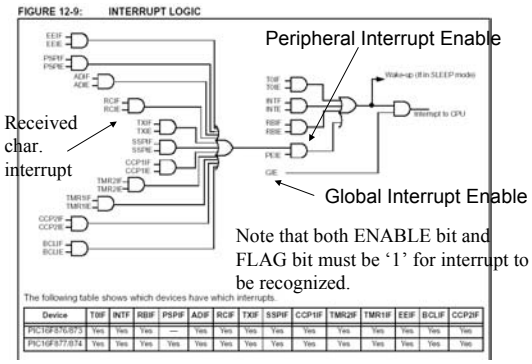
Interrupt-driven IO

In this example, the arrival of a character to RCREG triggers an interrupt - the normal program execution is halted, interrupt service routine executed which reads the character, saves in a temporary location.



PIC16 Interrupts (Chap 12.10 of datasheet)

- Many PIC16 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… 14 interrupt sources total
- 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.



GIE, PEIE

- Global interrupt enable (GIE) can be used to disable all interrupts
  - By default, all interrupts disabled
- Peripheral interrupt enable (PEIE) 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.

Interrupt Service Routine (ISR)

- The interrupt service routine (ISR) is called when an enabled interrupt occurs
  - Must be located at location 0x4
- When an enabled interrupt occurs…
  - GIE bit is CLEARED – this disables further interrupts (do not want to get caught in an infinite loop!)
  - Return address is pushed on the stack
  - PC loaded with 0x4
- Interrupt response is done *behind the scenes*
  - An interrupt can happen at any time, cannot predict where normal program execution will be interrupted

ISR Responsibilities

- Must save the status of the W register, and Status Register
  - If not saved, normal program execution will become unpredictable since interrupt can happen at anytime
  - Also save PCLATH if using pages 1,2,3 (not necessary PIC16873)
- 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 W, Status registers
- Execute RETFIE (return from interrupt)
  - Sets GIE to enable interrupts, reads PC from stack

PIC16F873 ISR

```
org 0x4

movwf W_TEMP         ; save W to filereg
movf  STATUS,w       ; save status in w
clrf  STATUS
movwf STATUS_TEMP    ; save status to filereg
 . . .
. . .ISR code . . .
. . .
movf  STATUS_TEMP,w
movwf STATUS          ; restore status, restores bank
swapf W_TEMP,f        ;restore W, use swap
swapf W_TEMP,w        ;because swap changes no flags
retfie                ; return from interrupt
```

W_TEMP location must be reserved in both banks because do not know what bank is selected when interrupt occurs.

PIC16F873 ISR in *C*

```
unsigned char received_char;
unsigned char got_char_flag;

/* interrupt service routine */
void interrupt pic_isr(void){

   /* see if this interrupt was generated by a
  receive character */

   if (bittst(PIR1,5)) {  /* check RCIF bit */
/* reading this register clears interrupt bit */

     received_char = RCREG;
     got_char_flag = 1;

   }
}
```

'interrupt' keyword indicates this is ISR

Check if receive char generated interrupt

Read character, set flag

Enabling Interrupts in C

```
/* code for serial port setup not shown */
/* enable interrupts */
  bitset(PIE1, 5);  /* enable receive int, RCIE */
  bitset(INTCON,6); /* enable peripheral ints, PEIE */
  bitset(INTCON,7); /* enable global interrupts, GIE */

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

  }
```

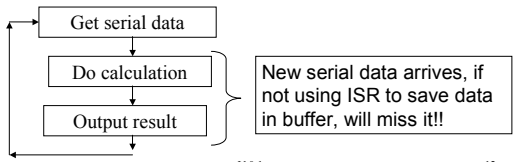
Set by interrupt service routine

Get character read by interrupt service routine

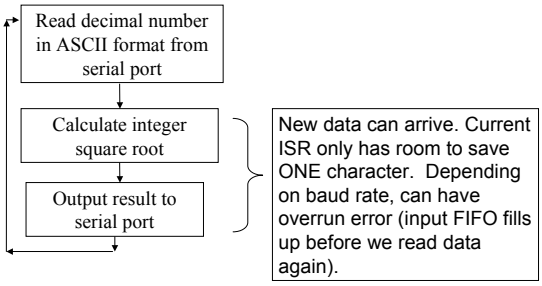
Soooooo…when are interrupts useful?

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

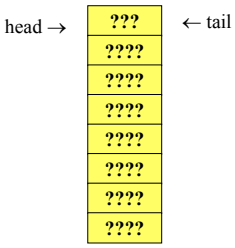


Buffering Data for Interrupt IO

- 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

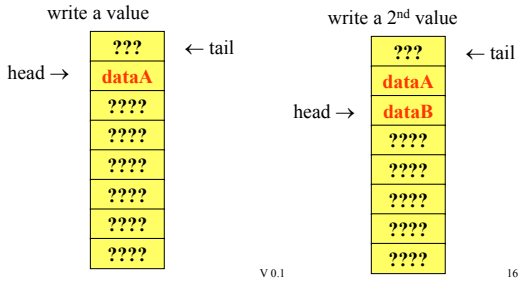
Circular buffer, 8 locations long

When buffer is empty, head = tail index



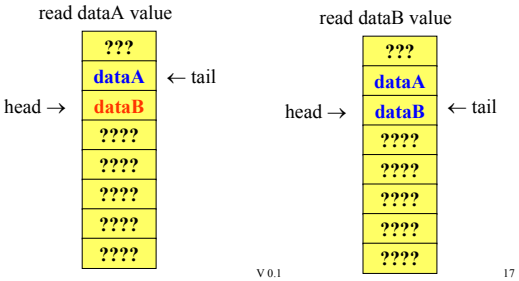
Circular buffer, write operation

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



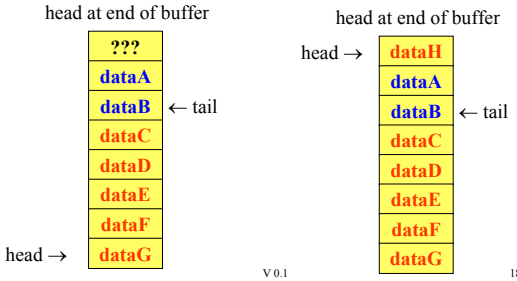
Circular buffer, read operation

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



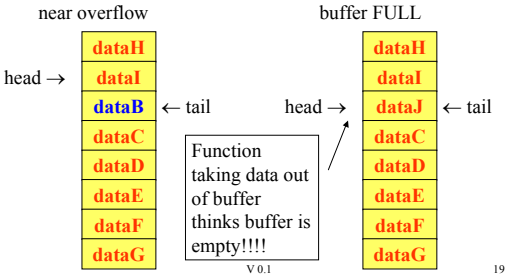
Circular buffer, wraparound

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.

Circular Buffers in *C*

Storage allocation

```
#define BUFSIZE 8
unsigned char buf[BUFSIZE];
unsigned char head,tail;
```

ISR places data into buffer, normal program execution reads data from buffer.

This code goes in ISR.

Placing data in buffer:

```
head++;
if (head == BUFSIZE) head=0;
buf[head] = c;
```

Wrap the pointers if at end of buffer!

Checking for data in buffer, reading data:

```
if (tail != head){
 /* data is available!*/
  tail++;
  if (tail == BUFSIZE) tail=0;
  c = buf[tail];
 }
```

This code goes in getch().

Allocating space in Bank1

May need to allocate buffer space and/or variables in bank1 if run out of space in bank0. Use bank1 tag for variable allocation

```
#define BUFSIZE 8
bank1 char buf[BUFSIZE];
bank1 unsigned char head,tail;
```

By default, the PICC compiler attempts to allocate all variable space in bank0. It will NOT use bank1 space unless you explicitly tag variables to be stored in bank1. PICC compiler will indicate error when bank0 space is exhausted.

What do you have to know?

- How interrupts behave on the PIC16 for serial IO
- Function of PEIE, GIE bits
- Responsiblilies of ISR
- Assembly language structure of ISR in PIC16
- ISR in PICC C
- Circular buffer operation