FIFO's

- Useful interface
  * provide slack to decouple producer and consumer rates
  * provides order preserving buffering
    - for the case where all produced values are important
    - alternative - single memory location
    - for the case when only the most recent value is needed
  * circular queue is a useful buffered I/O interface
    - statically allocated global memory
      * aids in controlling memory footprint when resources are limited
        - e.g., as in your lab kits
      * can be shared by main and ISR's
        - access must be carefully controlled to get it right however

Producer Consumer Examples

<table>
<thead>
<tr>
<th>Source/producer</th>
<th>Sink/consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard input</td>
<td>Program that interprets</td>
</tr>
<tr>
<td>Program with data</td>
<td>Printer output</td>
</tr>
<tr>
<td>Program sends message</td>
<td>Program receives message</td>
</tr>
<tr>
<td>Microphone and ADC</td>
<td>Program that saves sound data</td>
</tr>
<tr>
<td>Program that has sound data</td>
<td>DAC and speaker</td>
</tr>
</tbody>
</table>
Basic Code Model

* Not robust however

```c
char static volatile *PutPt; // put next
char static volatile *GetPt; // get next
// call by value
int Fifo_Put(char data){
  *PutPt = data; // Put
  PutPt++; // next
  return(1); // true if success
}
// call by reference
int Fifo_Get(char *datapt){
  *datapt = *GetPt; // return by reference
  GetPt++; // next
  return(1); // true if success
}
```

What’s missing?

2-pointer Finite FIFO Initialization

```c
#define FIFO_SIZE 10 /* can hold 9 */
char static volatile *PutPt; /* Pointer to put next */
char static volatile *GetPt; /* Pointer to get next */
// FIFO is empty if PutPt == GetPt */
// FIFO is full if PutPt+1 == GetPt (with wrap) */
char static Fifo[FIFO_SIZE];

void Fifo_Init(void)
{
  unsigned char SaveSP = begin_critical();
  PutPt=GetPt=0; // Empty when PutPt=GetPt */
  end_critical(SaveSP);
}
```

Atomicity Functions

```c
unsigned char begin_critical (void)
{
  unsigned char SaveSP;
  asm tpa
  asm staa SaveSP
  asm sei
  return SaveSP;
}

void end_critical (unsigned char SaveSP)
{
  asm ldaa SaveSP
  asm tpa
}
```

Put for a 2-pointer Circular FIFO

```c
int Fifo_Put(char data)
{
  char *Ppt; /* Temp put pointer */
  unsigned char SaveSP = begin_critical();
  Ppt=PutPt; /* Copy of put pointer */
  *(Ppt++)=data; /* Try to put data into fifo */
  if (Ppt == &Fifo[FIFO_SIZE]) Ppt = &Fifo[0]; /* Wrap */
  if (Ppt == GetPt )
  { // end_critical(SaveSP);
    return(0); /* Failed: fifo was full */
  }
  else { // Ppt=PutPt;
    end_critical (SaveSP);
    return(1); /* Successful */
  }
}
```
**Put Example**

Initially

```
data = 0x04
GetPt = 0x01
PutPt = 0x02
```

```c
int FifsPut(char data) {
    char *Ptmp;
    unsigned char SaveSP = begin_critical();
    Ptmp = PutPt;
    *Ptmp = data;
    if (Ptmp == &Fifo[FIFOSIZE])
        Ptmp = &Fifo[0];
    if (Ptmp == &GetPt) {
        end_critical(SaveSP);
        return(0);
    } else {
        Ptmp = Ptmp + 1;
        end_critical(SaveSP);
        return(1);
    }
}
```

**Get for a 2-pointer Circular FIFO**

```
data = 0x04
GetPt = 0x01
PutPt = 0x02
Ptmp = 0x03
```

```c
int FifoGet(char *data) {
    char *Ptmp;
    unsigned char SaveSP = begin_critical();
    Ptmp = GetPt;
    *data = Ptmp;
    if (Ptmp == &Fifo[FIFOSIZE])
        Ptmp = &Fifo[0];
    if (Ptmp == &GetPt) {
        end_critical(SaveSP);
        return(0);
    } else {
        Ptmp = Ptmp + 1;
        end_critical(SaveSP);
        return(1);
    }
```
2-pointer vs. Counter FIFO's

- 2 pointer version
  - Implicit number of elements
    - How do you calculate how many values are in the queue?
- Alternative is explicit store of current size
  - 2-pointer counter FIFO
    - Requires an extra variable, e.g., Size
    - But has compensating advantages

Initialization of a 2-pointer Counter FIFO

```c
#define FIFO_SIZE 10 /* can hold 10 */
char static volatile *PutPt; /* Pointer to put next */
char static volatile *GetPt; /* Pointer to get next */
char *Fifo[FIFO_SIZE];
unsigned char Size; /* Number of elements */
void Fifo_Init(void)
{  unsigned char SaveSP = begin_critical();
   PutPt=GetPt=Fifo[0]; /* Empty when Size=0 */
   Size=0;
   end_critical (SaveSP);
}
```

Put Function

```c
int Fifo_Put(char data)
{  if (Size == FIFO_SIZE )
     return(0);     /* Failed, fifo was full */
   else
     {  unsigned char SaveSP = begin_critical();
        Size++;
        *(PutPt++)=data; /* put data into fifo */
        if (PutPt == &Fifo[FIFO_SIZE])
         {  PutPt = &Fifo[0];   /* Wrap */
            end_critical (SaveSP);
            return(1);     /* Successful */
         }
     }
}
```

Get Function

```c
int Fifo_Get (char *data) 
{  if (Size == 0 )
     return(0);     /* Empty if Size=0 */
   else
     {  unsigned char SaveSP = begin_critical();
        *data=(GetPt++);
        Size--;
        if (GetPt == &Fifo[FIFO_SIZE])
         {  GetPt = &Fifo[0];   /* Wrap */
             end_critical (SaveSP);
             return(1);
         }
     }
    What advantages come from the Size variable?
}
```
Yet Another FIFO Option

- First two options
  - used pointers
- Index FIFO
  - accesses elements via array indices

Index FIFO Initialization

Same basic idea but w/o pointer weirdness

```c
#define FIFO_SIZE 10 /* Number of 8 bit data in the fifo */
unsigned char PutI; /* Index of where to put next */
unsigned char GetI; /* Index of where to get next */
unsigned char Size; /* Number of elements in the FIFO */
/* FIFO is empty if Size=0 */
/* FIFO is full if Size=FIFO_SIZE */
char Fifo[FIFO_SIZE]; /* The statically allocated fifo data */
void Fifo_Init(void)
{
    unsigned char SaveSP = begin_critical();
    PutI=GetI=Size=0; /* Empty when Size=0 */
    end_critical (SaveSP);
}
```

Index FIFO Put

```c
int Fifo_Put (char data)
{
    if (Size == FIFO_SIZE ) {
        return(0); /* Failed, fifo was full */
    } else {
        unsigned char SaveSP = begin_critical();
        Size++;
        Fifo[PutI++] = data; /* put data into fifo */
        if (PutI == FIFO_SIZE)
            PutI = 0; /* Wrap */
        end_critical (SaveSP);
        return(1); /* Successful */
    }
}
```

Index FIFO Get

```c
int Fifo_Get (char *datapt)
{
    if (Size == 0 ) {
        return(0); /* Empty if Size=0 */
    } else {
        unsigned char SaveSP = begin_critical();
        *datapt=Fifo[GetI++];
        Size--;
        if (GetI == FIFO_SIZE)
            GetI = 0;
        end_critical (SaveSP);
        return(1);
    }
}
```
FIFO Dynamics

Rates of production/consumption vary dynamically.
- \( t_p \) is time between Put calls, \( r_p \) is arrival rate \( (r_p = \frac{1}{t_p}) \).
- \( t_k \) is time between Get calls, \( r_k \) is service rate \( (r_k = \frac{1}{t_k}) \).
If \( \min t_p \geq \max t_k \), FIFO is not necessary.
If arrival rate can temporarily increase or service rate temporarily decrease, then a FIFO is necessary.
If average production rate exceeds average consumption rate \( (\text{i.e., } r_p > r_k) \), then FIFO will overflow.
A full error is serious because ignored data is lost.
An empty error may or may not be serious.

Concluding Remarks

- **Basic FIFO service**
  - decouple rate of production from rate of consumption
  - ideal size depends on maximum slack between the rates
- **Cost**
  - some RAM utilization and a few CPU cycles
  - note critical section occupancy
    - If it’s longer the \( t_p \) or \( t_k \) then there is a problem
- **Real systems have FIFO’s everywhere**
  - main reason why this lecture had such a narrow focus
  - what’s the fundamental reason for this?
- **FIFO’s are concurrent data structures**
  - touched by main + ISRs or threads
- **Writing correct concurrent data structures can be hard**
  - if done right then using them is easy