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>

Introduction to FIFOs

- FIFO circular queue is useful for a buffered I/O interface.
- This order-preserving data structure temporarily saves data created by a producer before being processed by a consumer.
- Decouples the producer from the consumer.
- Use statically allocated global memory, so they can be shared by threads, but must be accessed carefully.

Basic Idea of a FIFO

```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
}
```

Two-Pointer FIFO

```
```

FIFO with Infinite Memory
Two-Pointer FIFO

Initialization of a Two-Pointer FIFO

```c
#define FIFOSIZE 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[FIFOSIZE];
void Fifo_Init(void) {
    unsigned char SaveSP;
    asm tpa
    asm staa SaveSP
    asm sei /* make atomic, entering critical */
    PutPt=GetPt=&Fifo[0]; /* Empty when PutPt=GetPt */
    asm ldaa SaveSP
    asm tap /* end critical section */
}
```

Put for a Two-Pointer FIFO

```c
int Fifo_Put(char data) { char *Ppt; /* Temp put pointer */
unsigned char SaveSP;
asm tpa
asm staa SaveSP
asm sei /* make atomic, entering critical */
Ppt=PutPt; /* Copy of put pointer */
*(Ppt++)=data; /* Try to put data into fifo */
if (Ppt == &Fifo[FIFOSIZE]) Ppt = &Fifo[0]; /* Wrap */
if (Ppt == GetPt) {
    asm ldaa SaveSP
    asm tap /* end critical section */
    return(0); /* Failed, fifo was full */
} else {
    PutPt=Ppt;
    asm ldab %SaveSP\n tap*); /* end critical section */
    return(1); /* Successful */
}
```

Get for a Two-Pointer FIFO

```c
int Fifo_Get(char *datapt) {
unsigned char SaveSP;
if (PutPt == GetPt) {
    return(0); /* ... critical section */
    return(1); }
```}

Initialization of a Two-Pointer/Counter FIFO

```c
#define FIFOSIZE 10 /* can hold 10 */
char static volatile *PutPt; /* Pointer to put next */
char static volatile *GetPt; /* Pointer to get next */
char Fifo[FIFOSIZE];
unsigned char Size; /* Number of elements */
void Fifo_Init(void) {
    unsigned char SaveSP;
    asm tpa
    asm staa SaveSP
    asm sei /* make atomic, entering critical */
    PutPt=GetPt=&Fifo[0]; /* Empty when Size==0 */
    Size=0;
    asm ldaa SaveSP
    asm tap /* end critical section */
}
```

Put for a Two-Pointer/Counter FIFO

```c
int Fifo_Put(char data) {
unsigned char SaveSP;
if (Size == FIFOSIZE) {
    return(0); /* Failed, fifo was full */
} else {
    asm tpa
    asm staa SaveSP
    asm sei /* make atomic, entering critical */
    PutPt=GetPt=&Fifo[0]; /* Transparent data */
    if (GetPt == &Fifo[FIFOSIZE])
        GetPt = &Fifo[0]; /* Wrap */
    asm ldaa SaveSP
    asm tap /* end critical section */
    return(1); /* Successful */
}
```
Get for a Two-Pointer/Counter FIFO

```c
int Fifo_Get (char *datapt) {
    unsigned char SaveSP;
    if (Size == 0){
        return(0); /* Empty if Size=0 */
    } else {
        asm tpa
        asm staa SaveSP
        asm sei /* make atomic, entering critical */
        *datapt=*(GetPt++);
        Size--;
        if (GetPt == &Fifo[FIFOSIZE])
            GetPt = &Fifo[0]; /* Wrap */
        asm ldaa SaveSP
        asm tap /* end critical section */
        return(1); }}
```

FIFO Dynamics

- Rates of production/consumption vary dynamically.
- \( t_p \) is time between Put calls, \( r_p \) is arrival rate (\( r_p = \frac{1}{t_p} \)).
- \( t_g \) is time between Get calls, \( r_g \) is service rate (\( r_g = \frac{1}{t_g} \)).
- If \( t_p \geq t_g \), 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 (i.e., \( T_p > T_g \)), then FIFO will overflow.
- A full error is serious because ignored data is lost.
- An empty error is usually not serious.

SCI Data Flow Graph with Two FIFOs