CS/ECE 5780/6780: Embedded System Design

John Regehr

Lecture 9: 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 main and interrupts, but must be accessed carefully.
### 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>
FIFO with Infinite Memory
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

- Allocated memory: GetPt → Valid, PutPt → Valid
- After many PUTs: GetPt → Valid
- After one more PUT: PutPt → Valid, GetPt → Valid
- After one more PUT: PutPt → Valid, GetPt → Valid
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 = begin_critical();
    PutPt=GetPt=&Fifo[0]; /* Empty when PutPt=GetPt */
    end_critical (SaveSP);
}
```
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 tap
}
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[FIFOSIZE]) Ppt = &Fifo[0]; /* Wrap */
    if (Ppt == GetPt ) {
        end_critical (SaveSP);
        return(0);    /* Failed: fifo was full */
    } else {
        PutPt=Ppt;
        end_critical (SaveSP);
        return(1);    /* Successful */
    }
}
Put for a Two-Pointer FIFO Example

| data = 0x04 | 0xXX |
| GetPt → | 0x01 |
| 0x02 |
| 0x03 |
| PutPt → | 0xXX |
| 0xXX |

int Fifo_Put(char data) {
    char *Ppt;
    unsigned char SaveSP = begin_critical();
    Ppt=PutPt;
    *(Ppt++)=data;
    if (Ppt == &Fifo[FIFOSIZE])
        Ppt = &Fifo[0];
    if (Ppt == GetPt ) {
        end_critical (SaveSP);
        return(0);
    } else {
        PutPt=Ppt;
        end_critical (SaveSP);
        return(1);
    }
}
Put for a Two-Pointer FIFO Example

<table>
<thead>
<tr>
<th>data = 0x04</th>
<th>0xXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetPt</td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td>0x02</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td>PutPt/Ppt</td>
<td>0xXX</td>
</tr>
</tbody>
</table>

```c
int Fifo_Put(char data) {
    char *Ppt;
    unsigned char SaveSP = begin_critical();
    Ppt = PutPt;
    *(Ppt++) = data;
    if (Ppt == &Fifo[FIFOSIZE])
        Ppt = &Fifo[0];
    if (Ppt == GetPt)
        end_critical(SaveSP);
        return(0);
    else
        PutPt = Ppt;
        end_critical(SaveSP);
        return(1);
}
```
Put for a Two-Pointer FIFO Example

<table>
<thead>
<tr>
<th>data</th>
<th>0x04</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetPt</td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td>0x02</td>
</tr>
<tr>
<td></td>
<td>0x03</td>
</tr>
<tr>
<td>PutPt</td>
<td>0x04</td>
</tr>
<tr>
<td>Ppt</td>
<td>0xXX</td>
</tr>
</tbody>
</table>

```c
int Fifo_Put(char data) {
    char *Ppt;
    unsigned char SaveSP = begin_critical();
    Ppt=PutPt;
    *(Ppt++)=data;
    if (Ppt == &Fifo[FIFOSIZE])
        Ppt = &Fifo[0];
    if (Ppt == GetPt ) {
        end_critical (SaveSP);
        return(0);
    } else {
        PutPt=Ppt;
        end_critical (SaveSP);
        return(1);
    }
}
```
Put for a Two-Pointer FIFO Example

<table>
<thead>
<tr>
<th>data = 0x04</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ppt → 0xXX</td>
</tr>
<tr>
<td>GetPt → 0x01</td>
</tr>
<tr>
<td>0x02</td>
</tr>
<tr>
<td>0x03</td>
</tr>
<tr>
<td>PutPt → 0x04</td>
</tr>
<tr>
<td>0xXX</td>
</tr>
</tbody>
</table>

```c
int Fifo_Put(char data) {
    char *Ppt;
    unsigned char SaveSP = begin_critical();
    Ppt = PutPt;
    *(Ppt++) = data;
    if (Ppt == &Fifo[FIFOSIZE])
        Ppt = &Fifo[0];
    if (Ppt == GetPt)
        end_critical(SaveSP);
        return(0);
    else {
        PutPt = Ppt;
        end_critical(SaveSP);
        return(1);
    }
}
```
int Fifo_Put(char data) {
    char *Ppt;
    unsigned char SaveSP = begin_critical();
    Ppt=PutPt;
    *(Ppt++)=data;
    if (Ppt == &Fifo[FIFOSIZE])
        Ppt = &Fifo[0];
    if (Ppt == GetPt ) {
        end_critical (SaveSP);
        return(0);
    } else {
        PutPt=Ppt;
        end_critical (SaveSP);
        return(1);
    }
}
int Fifo_Get(char *datapt) {
    if (PutPt == GetPt) {
        return(0); /* Empty if PutPt=GetPt */
    } else {
        unsigned char SaveSP = begin_critical();
        *datapt = *(GetPt++);
        if (GetPt == &Fifo[FIFOSIZE])
            GetPt = &Fifo[0]; /* Wrap */
        end_critical(SaveSP);
        return(1);
    }
}
In the two-pointer FIFO code we’ve been looking at, the number of elements in the queue is *implicit*—not stored directly.

What is the formula for computing it?

An alternative implementation is to explicitly store the number of elements in the FIFO; we call the resulting data structure a “two-pointer counter/FIFO”.

This requires an extra variable but has compensating advantages.
#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 = begin_critical();
    PutPt=GetPt=&Fifo[0]; /* Empty when Size==0 */
    Size=0;
    end_critical (SaveSP);
}
```c
int Fifo_Put(char data) {
    if (Size == FIFOSIZE) {
        return(0); /* Failed, fifo was full */
    } else {
        unsigned char SaveSP = begin_critical();
        Size++;
        *(PutPt++) = data; /* put data into fifo */
        if (PutPt == &Fifo[FIFOSIZE]) {
            PutPt = &Fifo[0]; /* Wrap */
        }
        end_critical(SaveSP);
        return(1); /* Successful */
    }
}
```
int Fifo_Get (char *datapt) {
    if (Size == 0 ) {
        return(0); /* Empty if Size=0 */
    } else {
        unsigned char SaveSP = begin_critical();
        *datapt=* (GetPt++);
        Size--;
        if (GetPt == &Fifo[FIFOSIZE]) {
            GetPt = &Fifo[0]; /* Wrap */
        }
        end_critical (SaveSP);
        return(1);
    }
}
The FIFOs we have been looking at use pointers.
An alternative is the “index FIFO” which accesses elements using array indices.
Initialization of an Index FIFO

```c
#define FIFOSIZE 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=FIFOSIZE */
char Fifo[FIFOSIZE];    /* 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);
}
```
int Fifo_Put (char data)
{
    if (Size == FIFOSIZE ) {
        return(0); /* Failed, fifo was full */
    } else {
        unsigned char SaveSP = begin_critical();
        Size++;
        Fifo[PutI++]=data; /* put data into fifo */
        if (PutI == FIFOSIZE)
            PutI = 0; /* Wrap */
        end_critical (SaveSP);
        return(1); /* Successful */
    }
}
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 == FIFOSIZE)
            GetI = 0;
        end_critical (SaveSP);
        return(1);
    }
}
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 $\min t_p \geq \max t_g$, 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., $\bar{r}_p > \bar{r}_g$), then FIFO will overflow.

A full error is serious because ignored data is lost.
An empty error may or may not be serious.
SCI Data Flow Graph with Two FIFOs
FIFO Summary

The service provided by FIFOs is decoupling the rate of production from the rate of consumption.
The cost you pay for this service is some RAM and a few CPU cycles.
Real systems have FIFOs everywhere—What is the fundamental reason for this?
FIFOs are concurrent data structures: touched by main + interrupts and/or threads.
Writing concurrent data structures can be hard, but if they are done right, using them is easy.