the join are complete. As an
e neighbors over and gives
the farmer working alone to
o accomplish the single goal
eration causes the neighbors

le threads, but only one runs
ency on real-time systems.
nterrupt is a parameter-less
mbols for interrupts are also
it is time to do something.
new input data has arrived,
f an interrupt-driven system
l. The **foreground** thread is
nd threads are executions of
g a book. Reading a book is
ding at the beginning of the
You might jump to the back
m where, which is analogous
mes, which is analogous to a
of pages you read follows a
gs, you place a bookmark in
hone conversation, you hang
t off. The ringing phone is
xecuting the ISR.

l by two (or more) threads. In
m starts executing a function,
l. In order for two threads to
entrant software, place local
lobal memory variables. The

## 3.7. First In First Out Queue

### 3.7.1. Classical definition of a FIFO

The first in first out circular queue (**FIFO**) is quite useful for implementing a buffered I/O interface (Figure 3.9). It can be used for both buffered input and buffered output. The order preserving data structure temporarily saves data created by the source (producer) before it is processed by the sink (consumer). The class of FIFOs studied in this section will be statically allocated global structures. Because they are global variables, it means they will exist permanently and can be carefully shared by more than one program. The advantage of using a FIFO structure for a data flow problem is that we can decouple the producer and consumer threads. Without the FIFO we would have to produce 1 piece of data, then process it, produce another piece of data, then process it. With the FIFO, the producer thread can continue to produce data without having to wait for the consumer to finish processing the previous data. This decoupling can significantly improve system performance.

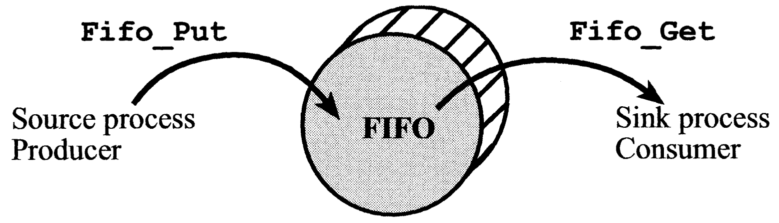

*Figure 3.9. The FIFO is used to buffer data between the producer and consumer.*

You have probably already experienced the convenience of FIFOs. For example, a FIFO is used while streaming audio from the Internet. As sound data are received from the Internet they are put (calls **`Fifo_Put`**) in a FIFO. When the sound board needs data it calls **`Fifo_Get`**. As long as the FIFO never comes full or empty, the sound is played in a continuous manner. A FIFO is also used when you ask the computer to print a file. Rather than waiting for the actual printing to occur character by character, the print command will put the data in a FIFO. Whenever the printer is free, it will get data from the FIFO. The advantage of the FIFO is it allows you to continue to use your computer while the printing occurs in the background. To implement this magic of background printing we will need interrupts. There are many producer/consumer applications. In Table 3.3 the processes on the left are producers that create or input data, while the processes on the right are consumers which process or output data.

| *Source/Producer* | *Sink/Consumer* |
|---|---|
| Keyboard input | Program that interprets |
| Program with data | Printer output |
| Program sends message | Program receives message |
| Microphone and ADC | Program that saves sound data |
| Program that has sound data | DAC and speaker |

**Table 3.3. Producer consumer examples.**

The producer puts data into the FIFO. The **`Fifo_Put`** operation does not discard information already in the FIFO. If the FIFO is full and the user calls **`Fifo_Put`**, the **`Fifo_Put`** routine will return a full error signifying the last (newest) data was not properly saved. The sink process removes data from the FIFO. The **`Fifo_Get`** routine will modify the FIFO. After a get, the particular information returned from the get routine is no longer saved on the FIFO. If the FIFO is empty and the user tries to get, the **`Fifo_Get`** routine will return an empty error signifying no data could be retrieved. The FIFO is order preserving, such that the information is returned by repeated calls of **`Fifo_Get`** in the same order as the data was saved by repeated calls of **`Fifo_Put`**.

There are many ways to implement a statically-allocated FIFO. We can use either a pointer or an index to access the data in the FIFO. We can use either two pointers (or two indices) or two pointers (or two indices) and a counter. The counter specifies how many entries are currently stored in the FIFO. There are even hardware implementations of FIFO queues. We begin with the two-pointer implementation. It is a little harder to implement, but it does have some advantages over the other implementations.

## 3.7.2. Two-pointer FIFO implementation

The two-pointer implementation has, of course, two pointers. If we were to have infinite memory, a FIFO implementation is easy (Figure 3.10). **`GetPt`** points to the data that will be removed by the next call to **`Fifo_Get`**, and **`PutPt`** points to the empty space where the data will stored by the next call to **`Fifo_Put`**, see Program 3.6.

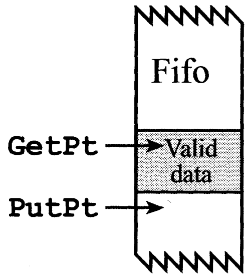

*Figure 3.10. The FIFO implementation with infinite memory.*

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

*Program 3.6. Code fragments showing the basic idea of a FIFO.*

There are four modifications that are required to the above subroutines. If the FIFO is full when **`Fifo_Put`** is called then the function should return a full error. Similarly, if the FIFO is empty when **`Fifo_Get`** is called, then the function should return an empty error. **`PutPt`** must be wrapped back up to the top when it reaches the bottom (Figure 3.11).

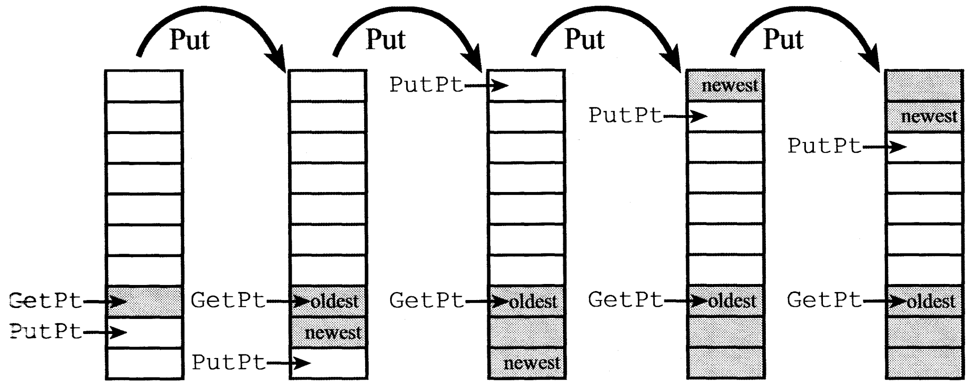

*Figure 3.11. The FIFO **`Fifo_Put`** operation showing the pointer wrap.*

The **`GetPt`** must also be wrapped back up to the top when it reaches the bottom (Figure 3.12).

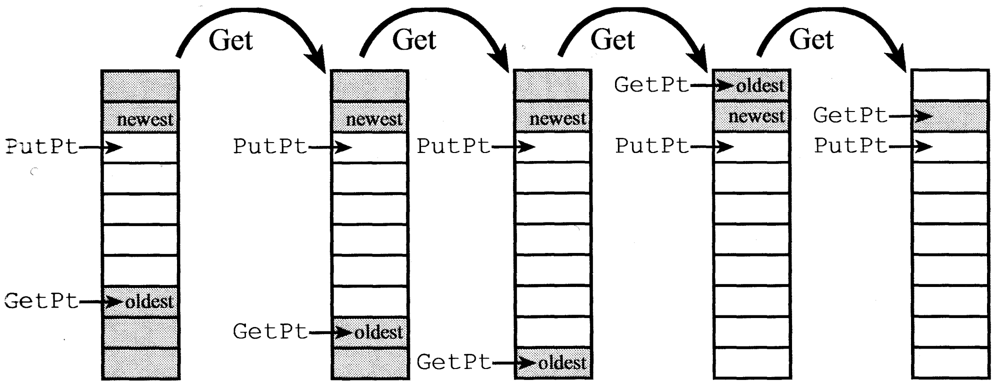

*Figure 3.12. The FIFO **`Fifo_Get`** operation showing the pointer wrap.*

There are two mechanisms to determine whether the FIFO is empty or full. A simple method is to implement a counter containing the number of bytes currently stored in the FIFO. **`Fifo_Get`** would decrement the counter and **`Fifo_Put`** would increment the counter. We will not implement a counter because incrementing and decrementing a counter causes a race condition, meaning the counter could become incorrect when shared in a multithreaded environment. Race conditions and critical sections will be presented in Chapter 5.

The second method is to prevent the FIFO from being completely full. The implementation of this FIFO module is shown in Program 3.7. You can find all the FIFOs of this section on the book web site as **FIFO_xxx.zip**, where xxx refers to the specific microcontroller on which the example was tested.

```
#define FIFOSIZE 10    // can be any size
#define FIFOSUCCESS 1
#define FIFOFAIL    0
typedef char DataType;
DataType volatile *PutPt; // put next
DataType volatile *GetPt; // get next
DataType static Fifo[FIFOSIZE];
// initialize FIFO
void Fifo_Init(void){
  PutPt = GetPt = &Fifo[0]; // Empty
}
// add element to FIFO
int Fifo_Put(DataType data){
  DataType volatile *nextPutPt;
  nextPutPt = PutPt+1;
  if(nextPutPt == &Fifo[FIFOSIZE]){
    nextPutPt = &Fifo[0];  // wrap
  }
  if(nextPutPt == GetPt){
    return(FIFOFAIL);      // Failed, FIFO full
  }
  else{
    *(PutPt) = data;       // Put
    PutPt = nextPutPt;     // Success, update
    return(FIFOSUCCESS);
  }
}
// remove element from FIFO
int Fifo_Get(DataType *datapt){
  if(PutPt == GetPt ){
    return(FIFOFAIL);      // Empty if PutPt=GetPt
  }
  *datapt = *(GetPt++);
  if(GetPt == &Fifo[FIFOSIZE]){
     GetPt = &Fifo[0];   // wrap
  }
  return(FIFOSUCCESS);
}
```

*Program 3.7. Two-pointer implementation of a FIFO.*

For example, if the FIFO had 10 bytes allocated, then the **`Fifo_Put`** subroutine would allow a maximum of 9 bytes to be stored. If there were already 9 bytes in the FIFO and another **`Fifo_Put`** were called, then the FIFO would not be modified and a full error would be returned. See Figure 3.13. In this way if **`PutPt`** equals **`GetPt`** at the beginning of **`Fifo_Get`**, then the FIFO is empty. Similarly, if **`PutPt+1`** equals **`GetPt`** at the beginning of **`Fifo_Put`**, then the FIFO is full. Be careful to wrap the **`PutPt+1`** before comparing it to **`Fifo_Get`**. This method does not require the length to be stored or calculated.

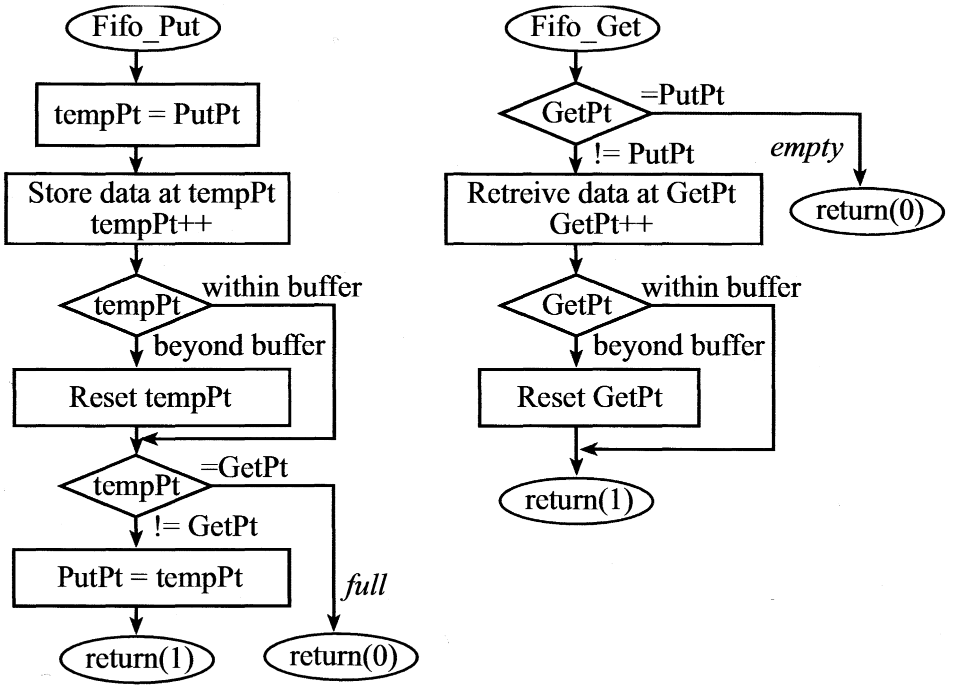

*Figure 3.13. Flowcharts of the pointer implementation of the* FIFO *queue.*

To check for FIFO full, the following **`Fifo_Put`** routine attempts to put using a temporary **`PutPt`**. If putting makes the FIFO look empty, then the temporary **`PutPt`** is discarded and the routine is exited without saving the data. This is why a FIFO with 10 allocated bytes can only hold 9 data points. If putting doesn't make the FIFO look empty, then the temporary **`PutPt`** is stored into the actual **`PutPt`** saving the data as desired.

To check for FIFO empty, the **`Fifo_Get`** routine in Program 3.7 simply checks to see if **`GetPt`** equals **`PutPt`**. If they match at the start of the routine, **`then Fifo_Get`** returns with the "empty" condition signified.

Since **`Fifo_Put`** and **`Fifo_Get`** have read modify write accesses to global variables they are themselves not reentrant. Similarly **`Fifo_Init`** has a multiple step write access to global variables. Therefore **`Fifo_Init`** is not reentrant.

One advantage of this pointer implementation is that if you have a single thread that calls the **`Fifo_Get`** (e.g., the main program) and a single thread that calls the **`Fifo_Put`** (e.g., the serial port receive interrupt handler), then this **`Fifo_Put`** function can interrupt this **`Fifo_Get`** function without loss of data. So in this particular situation, interrupts would not have to be disabled. It would also operate properly if there were a single interrupt thread calling **`Fifo_Get`** (e.g., the serial port transmit interrupt handler) and a single thread calling **`Fifo_Put`** (e.g., the main program.) On the other hand, if the situation is more general, and multiple threads could call **`Fifo_Put`** or multiple threads could call **`Fifo_Get`**, then the interrupts would have to be temporarily disabled.

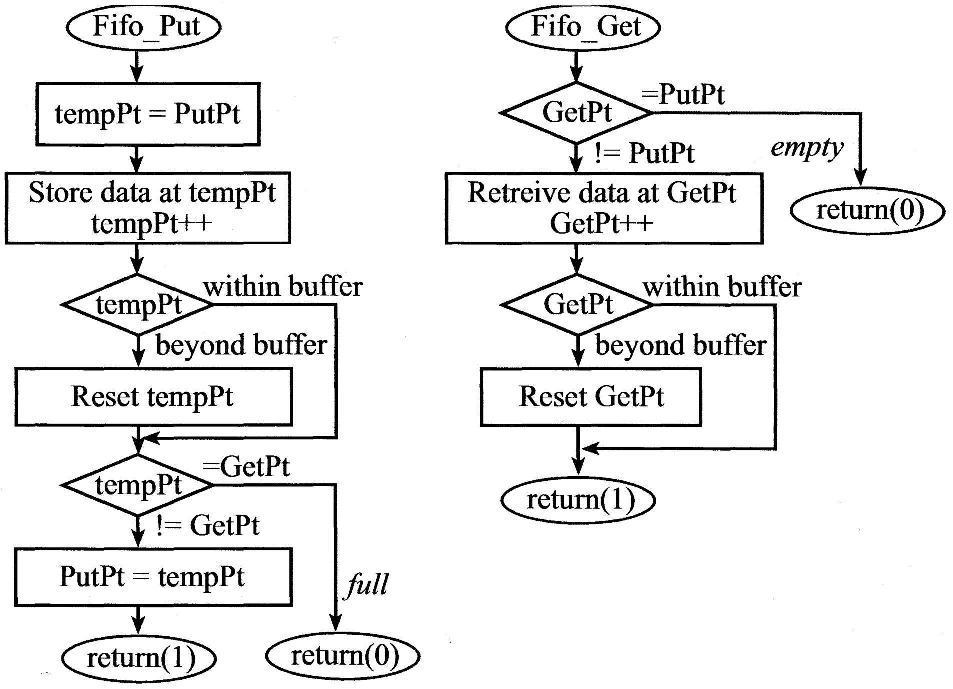

*Figure 3.13. Flowcharts of the pointer implementation of the FIFO queue.*

To check for FIFO full, the following **`Fifo_Put`** routine attempts to put using a temporary **`PutPt`**. If putting makes the FIFO look empty, then the temporary **`PutPt`** is discarded and the routine is exited without saving the data. This is why a FIFO with 10 allocated bytes can only hold 9 data points. If putting doesn't make the FIFO look empty, then the temporary **`PutPt`** is stored into the actual **`PutPt`** saving the data as desired.

To check for FIFO empty, the **`Fifo_Get`** routine in Program 3.7 simply checks to see if **`GetPt`** equals **`PutPt`**. If they match at the start of the routine, **`then Fifo_Get`** returns with the "empty" condition signified.

Since **`Fifo_Put`** and **`Fifo_Get`** have read modify write accesses to global variables they are themselves not reentrant. Similarly **`Fifo_Init`** has a multiple step write access to global variables. Therefore **`Fifo_Init`** is not reentrant.

One advantage of this pointer implementation is that if you have a single thread that calls the **`Fifo_Get`** (e.g., the main program) and a single thread that calls the **`Fifo_Put`** (e.g., the serial port receive interrupt handler), then this **`Fifo_Put`** function can interrupt this **`Fifo_Get`** function without loss of data. So in this particular situation, interrupts would not have to be disabled. It would also operate properly if there were a single interrupt thread calling **`Fifo_Get`** (e.g., the serial port transmit interrupt handler) and a single thread calling **`Fifo_Put`** (e.g., the main program.) On the other hand, if the situation is more general, and multiple threads could call **`Fifo_Put`** or multiple threads could call **`Fifo_Get`**, then the interrupts would have to be temporarily disabled.

### 3.7.3. Two index FIFO implementation

The other method to implement a FIFO is to use indices rather than pointers. This FIFO has the restriction that the size must be a power of 2. In Program 3.8, **FIFOSIZE** is 16 and the logic **PutI&(FIFOSIZE-1)** returns the bottom four bits of the put index. Similarly, the logic **GetI&(FIFOSIZE-1)** returns the bottom four bits of the get index. Using the bottom bits of the index removes the necessary to check for out of bounds and wrapping.

```
// Two-index implementation of the transmit FIFO
// can hold 0 to FIFOSIZE elements
#define FIFOSIZE 16 // must be a power of 2
#define FIFOSUCCESS 1
#define FIFOFAIL    0
typedef char DataType;
unsigned long volatile PutI;// put next
unsigned long volatile GetI;// get next
DataType static Fifo[FIFOSIZE];
// initialize index FIFO
void Fifo_Init(void){
  PutI = GetI = 0;  // Empty
}
// add element to end of index FIFO
int Fifo_Put(DataType data){
  if((PutI-GetI) & ~(FIFOSIZE-1)){
    return(FIFOFAIL); // Failed, fifo full
  }
  Fifo[PutI&(FIFOSIZE-1)] = data; // put
  PutI++;  // Success, update
  return(FIFOSUCCESS);
}
// remove element from front of index FIFO
int Fifo_Get(DataType *datapt){
  if(PutI == GetI ){
    return(FIFOFAIL); // Empty if PutI=GetI
  }
  *datapt = Fifo[GetI&(FIFOSIZE-1)];
  GetI++;  // Success, update
  return(FIFOSUCCESS);
}
```

*Program 3.8. Implementation of a two-index FIFO. The size must be a power of two.*

If the FIFO is full, then **(PutI-GetI)** will equal 16, meaning all elements of the buffer have data. The expression **~(FIFOSIZE-1)** yields the constant 0xFFFFFFF0. For all sizes that are a power of 2, the if statement in put will be nonzero if there are **FIFOSIZE** elements in the FIFO. With this implementation a FIFO with 16 allocated bytes can actually hold 16 data points. The FIFO is empty if **PutI** equals **GetI**. If empty, the **Fifo_Get** function returns with the **FIFOFAIL** condition.

than pointers. This FIFO has the
FIFOSIZE is 16 and the logic
put index. Similarly, the logic
t index. Using the bottom bits of
l wrapping.

r of two.

g all elements of the buffer have
FFFFFFF0. For all sizes that are
: are FIFOSIZE elements in the
bytes can actually hold 16 data
the Fifo_Get function returns

## 3.7.4. FIFO build macros

When we need multiple FIFOs in our system, we could switch over to C++ and define the FIFO as a class, and then instantiate multiple objects to create the FIFOs. A second approach would be to use a text editor, open the source code containing Program 3.7 or 3.8, copy/paste it, and then change names so the functions are unique. A third approach is shown in Programs 3.9 and 3.10, which defines macros allowing us to create as many FIFOs as we need.

```
// macro to create a pointer FIFO
#define AddPointerFifo(NAME,SIZE,TYPE,SUCCESS,FAIL) \
TYPE volatile *NAME ## PutPt;    \
TYPE volatile *NAME ## GetPt;    \
TYPE static NAME ## Fifo [SIZE];        \
void NAME ## Fifo_Init(void){           \
  NAME ## PutPt = NAME ## GetPt = &NAME ## Fifo[0]; \
}                                       \
int NAME ## Fifo_Put (TYPE data){       \
  TYPE volatile *nextPutPt;             \
  nextPutPt = NAME ## PutPt + 1;        \
  if(nextPutPt == &NAME ## Fifo[SIZE]){ \
    nextPutPt = &NAME ## Fifo[0];       \
  }                                     \
  if(nextPutPt == NAME ## GetPt ){      \
    return(FAIL);                       \
  }                                     \
  else{                                 \
    *( NAME ## PutPt ) = data;          \
    NAME ## PutPt = nextPutPt;          \
    return(SUCCESS);                    \
  }                                     \
}                                       \
int NAME ## Fifo_Get (TYPE *datapt){    \
  if( NAME ## PutPt == NAME ## GetPt ){ \
    return(FAIL);                       \
  }                                     \
  *datapt = *( NAME ## GetPt ## ++);    \
  if( NAME ## GetPt == &NAME ## Fifo[SIZE]){ \
    NAME ## GetPt = &NAME ## Fifo[0];   \
  }                                     \
  return(SUCCESS);                      \
}
```

*Program 3.9. Two-pointer macro implementation of a FIFO.*

To create a 20-element FIFO storing unsigned 16-bit numbers that returns 1 on success and 0 on failure we invoke

```
AddPointerFifo(Rx, 20, unsigned short, 1, 0)
```

creating the three functions `RxFifo_Init()`, `RxFifo_Get()`, and `RxFifo_Put()`.

Program 3.10 is a macro allowing us to create two-index FIFOs similar to Program 3.8.

```
// macro to create an index FIFO
#define AddIndexFifo(NAME,SIZE,TYPE,SUCCESS,FAIL) \
unsigned long volatile NAME ## PutI;    \
unsigned long volatile NAME ## GetI;    \
TYPE static NAME ## Fifo [SIZE];        \
void NAME ## Fifo_Init(void){           \
  NAME ## PutI = NAME ## GetI = 0;      \
}                                       \
int NAME ## Fifo_Put (TYPE data){       \
  if(( NAME ## PutI - NAME ## GetI ) & ~(SIZE-1)){  \
    return(FAIL);      \
  }                    \
  NAME ## Fifo[ NAME ## PutI &(SIZE-1)] = data; \
  NAME ## PutI ## ++;  \
  return(SUCCESS);     \
}                      \
int NAME ## Fifo_Get (TYPE *datapt){  \
  if( NAME ## PutI == NAME ## GetI ){ \
    return(FAIL);      \
  }                    \
  *datapt = NAME ## Fifo[ NAME ## GetI &(SIZE-1)];  \
  NAME ## GetI ## ++;  \
  return(SUCCESS);     \
}                      \
unsigned short NAME ## Fifo_Size (void){  \
 return ((unsigned short)( NAME ## PutI - NAME ## GetI ));  \
}
```

*Program 3.10. Macro implementation of a two-index FIFO. The size must be a power of two.*

To create a 32-element FIFO storing signed 32-bit numbers that returns 0 on success and 1 on failure we invoke

```
AddIndexFifo(Tx, 32, long, 0, 1)
```

creating the three functions `TxFifo_Init()`, `TxFifo_Get()`, and `TxFifo_Put()`.

**Checkpoint 3.11:** Show C code to create three FIFOs called CAN1 CAN2 and CAN3. Each FIFO stores 8-bit bytes and must be able to store up to 99 elements.

**Checkpoint 3.12:** Show C code to create two FIFOs called F1 and F2. Each FIFO stores 16-bit halfwords and must be able to store up to 256 elements.