EE443 - Embedded Systems

Lecture 6
Real-Time Programming Tools

Contents:

6.1 Stacks

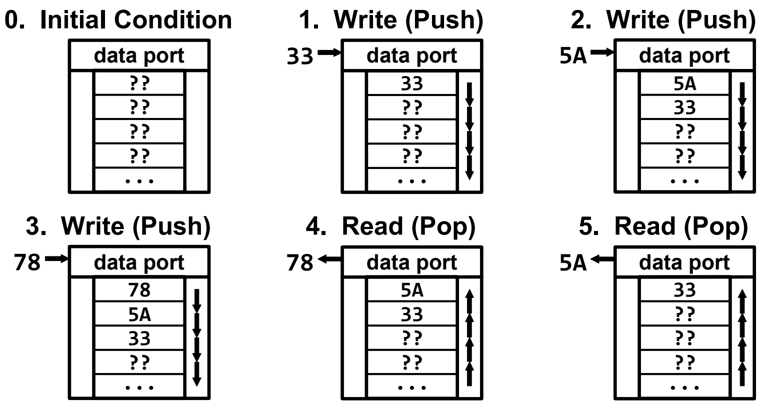
Stack is a sequential-access data storage where the data stored last is retrieved first. This access sequence is called "**Last In First Out**" or "**LIFO**" in short. The two operations applicable to all stacks are:

- **Push operation:** A new data item is written on the top of the stack. Previously written items (if any) are pushed deeper into the stack storage.
- **Pop** (or pull) **operation:** The last stored data item is read from the top of the stack. Previously written items (if any) are pulled towards the top of the stack.

The typical usage of stack is storing local data and call information for procedure calls.

6.1.1 Hardware Stack

Hardware implementation of stack is a bunch of shift registers operating in parallel. Eight shift registers put together make a stack that stores one byte at a time.



The hardware implementation of stack has a single bidirectional data port that is used for write and read operations. There is no address port. Location of data elements in the stack depends on the history of the write (PUSH) and read (POP) operations.

6.1.2 Stack Implementation In Processors

Most processors do not have a sequential-access memory that works as the hardware stack. They utilize a dedicated section of the memory to implement the stack operations. The "Last In First Out" access sequence is obtained by using an address register called "**Stack Pointer**" or **SP**. The stack pointer is automatically incremented or decremented when the processor executes **PUSH** or **POP** instructions. The same LIFO access sequence is obtained simply by changing the address stored in the Stack Pointer, which is much easier and faster than shifting the stack contents

- **PUSH operation:** A new data item is placed at the location pointed to by the stack pointer, and the address in the stack pointer is decremented by the size of the data item.
- **POP** (pull) **operation:** The last stored data item at the current location pointed to by the stack pointer is read back, and the stack pointer is incremented by the size of the data item

0. Initial Condition:

`A=33, B=55`

	addr	data
SP →	FFFF	??
	FFFE	??
	FFFD	??
	FFFC	??
	....	..

1. Store register-A:

`PUSH A`

	addr	data
↓	FFFF	33
SP →	FFFE	??
	FFFD	??
	FFFC	??
	....	..

2. Store register-B:

`PUSH B`

	addr	data
	FFFF	33
↓	FFFE	55
SP →	FFFD	??
	FFFC	??
	....	..

3. Retrieve reg-B:

`POP B`

	addr	data
	FFFF	33
SP →	FFFE	55
↑	FFFD	??
	FFFC	??
	....	..

4. Retrieve reg-A:

`POP A`

	addr	data
SP →	FFFF	33
↑	FFFE	55
	FFFD	??
	FFFC	??
	....	..

What happens after these instructions?

```
1. PUSH  A;
2. PUSH  B;
3. POP   A;
4. POP   B;
```

The example given above assumes that the memory space reserved for stack operations is located at the highest memory addresses. Different parts of the memory can be reserved for stack usage in other memory organizations. Following example summarizes the stack usage during a procedure call.

In the main program:

```
    LOAD  A,#0   ; main program uses register-A
    CALL  Sub1   ; 1. store addr. of the next instruction (incremented PC)
                 ;    at memory location pointed to by SP
                 ; 2. decrement SP 2 times, (PC has 2 bytes)
                 ; 3. set PC to address of Sub1
    ADD   A,#1   ; next instruction of the main program
                 ; all register contents are the same as before the call
    ...
```

In the procedure:

```
Sub1:
   PUSH    A      ; 1) save register-A on stack,
                  ; 2) decrement SP
   PUSH    B      ; 1) save register-B on stack,
                  ; 2) decrement SP
   ...
   perform procedure operations using registers A and B
   ...
   POP     B      ; 1) increment SP,
                  ; 2) restore register-B contents
```

```
    POP     A      ; 1) increment SP,
                   ; 2) restore register-A contents
    RETURN         ; 1) increment SP twice (one for each address byte),
                   ; 2) read return address into PC
```

It is crucial to **pop** exactly the same amount of data that was **pushed** in a procedure before executing the **Return** instruction. Otherwise, the return address loaded into PC will not be the address of the next instruction saved during the call operation.

Utilizing the stack in procedure calls has two advantages compared to having dedicated memory locations to store the return address and register contents.

1. Using stack is faster since SP provides an address already available in the processor. **Push** and **Pop** instructions require two or three memory access cycles, one cycle for fetching opcode and one or two cycles for storing or retrieving register contents. **Store** and **Load** instructions on the other hand, require 2 more memory access cycles to get the storage address in direct addressing (assuming a memory interface with 8-bit data and 16-bit address).

2. Stack memory is reusable. We can use the stack memory over and over again for all procedure calls. In the other case, every procedure call requires its own storage memory to save registers. So, memory is utilized more efficiently when the stack is used.

Example: Trace the stack operations during procedure calls:

Code Addr.	Instruction	Trace Order	Program Counter	Stack Pointer	--------- Stack Memory --------- FFFF	FFFE	FFFD	FFFC	FFFB	FFFA	FFF9
	MainProg:	--	----	----	--	--	--	--	--	--	--
----	----------	0	0A10	FFFF	??	??	??	??	??	??	??
0A10	Load A, #0xAA;	1	0A12	FFFF	??	??	??	??	??	??	??
0A12	Call Sub1;	2	1B20	FFFD	0A	15	??	??	??	??	??
0A15	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
0B20	Load A, #0xBB;	6	0B22	FFFF	0A	15	AA	??	??	??	??
0B22	Call Sub2	7	2C30	FFFD	0B	25	AA	??	??	??	??
0B25	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
	Sub1:	--	----	----	--	--	--	--	--	--	--
1B20	Push A;	3	1B21	FFFC	0A	15	AA	??	??	??	??
1B21	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
1B40	Pop A;	4	1B41	FFFD	0A	15	AA	??	??	??	??
1B41	Return;	5	0A15	FFFF	0A	15	AA	??	??	??	??
----	----------	--	----	----	--	--	--	--	--	--	--
	Sub2:	--	----	----	--	--	--	--	--	--	--
2C30	Push A;	8	2C31	FFFC	0B	25	BB	??	??	??	??
2C31	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
2C50	Pop A;	9	2C51	FFFD	0B	25	BB	??	??	??	??
2C51	Return;	10	0B25	FFFF	0B	25	BB	??	??	??	??
----	----------	--	----	----	--	--	--	--	--	--	--

Example: Trace the stack operations during nested procedure calls:

Code Addr.	Instruction	Trace Order	Program Counter	Stack Pointer	--------- Stack Memory ---------						
					FFFF	FFFE	FFFD	FFFC	FFFB	FFFA	FFF9
	MainProg:	--	----	----	--	--	--	--	--	--	--
----	----------	0	0A10	FFFF	??	??	??	??	??	??	??
0A10	Load A, #0xAA;	1	0A10	FFFF	??	??	??	??	??	??	??
0A12	Call Sub1;	2	1B20	FFFD	0A	15	??	??	??	??	??
0A15	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
	Sub1:	--	----	----	--	--	--	--	--	--	--
1B20	Push A;	3	1B21	FFFC	0A	15	AA	??	??	??	??
1B21	Load A, #0xBB;	4	1B23	FFFC	0A	15	AA	??	??	??	??
1B23	Call Sub2;	5	2C30	FFFA	0A	15	AA	1B	26	??	??
1B26	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
1B40	Pop A;	9	1B41	FFFD	0A	15	AA	1B	26	BB	??
1B41	Return;	10	0A15	FFFF	0A	15	AA	1B	26	BB	??
----	----------	--	----	----	--	--	--	--	--	--	--
----	----------	--	----	----	--	--	--	--	--	--	--
	Sub2:	--	----	----	--	--	--	--	--	--	--
2C30	Push A;	6	2C31	FFF9	0A	15	AA	1B	26	BB	??
----	----------	--	----	----	--	--	--	--	--	--	--
2C50	Pop A;	7	2C51	FFFA	0A	15	AA	1B	26	BB	??
2C51	Return;	8	1B26	FFFC	0A	15	AA	1B	26	BB	??
----	----------	--	----	----	--	--	--	--	--	--	--

6.2 Queues

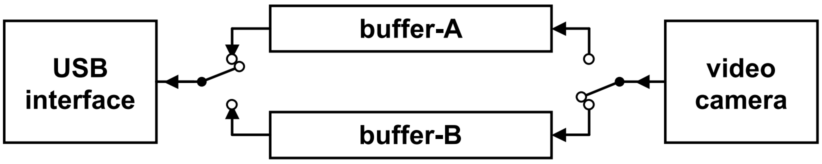
Queue is another type of sequential-access data storage where the first element added to the queue will be the first one to be removed. This access sequence is called "**First In First Out**" or "**FIFO**", in short. The two operations applicable to all queues are:

- **Enqueue operation:** A new item is written through the rear data port of the queue. Previously written items (if any) are shifted towards the front data port.
- **Dequeue operation:** The first stored item is read from the front data port of the queue. The remaining items (if any) are shifted towards the front data port.

Queues function as buffers where data or events are stored and held to be processed later.

Fixed-length queues serve as I/O buffers for data transfers in predetermined packet sizes. For example, data transfers in hard disks are grouped in sectors. 512 bytes of data are stored in each sector. Storing data in smaller packet sizes is inefficient because the necessary synchronization and error correction information are added to every sector of data written on the hard disk. Data transmission through USB, firewire, ethernet, and similar connections or wireless channels occur as packet transfers. The data to be transferred are first stored in a buffer until the buffer is filled, and then the entire buffer contents are transmitted as a single data packet.

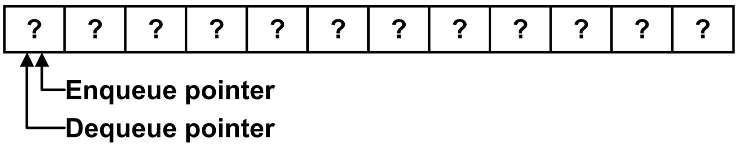
In some applications, several I/O buffers are required to manage continuous inflow of data or to maintain a higher data transfer rate. The following figure shows arrangement of two buffers for transferring video data through a USB connection. The video camera produces a continuous stream of bytes at a rate lower than the USB transfer rate. When buffer-A is filled, the incoming data is directed to buffer-B without interrupting the stream of bytes. The USB interface transmits the buffer-A contents while video data is being written into buffer-B. Buffer-A is ready to store data again when buffer-B is full.



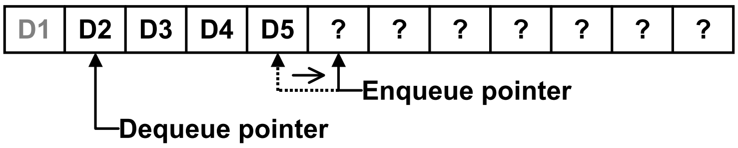
Hardware implementation of queues is possible by combining several shift registers operating in parallel similar to the stack implementation. A queue structure has two data ports for storing and retrieving data unlike the stack structure that uses a single bidirectional port. Long buffers cannot be constructed as shift registers since the power consumption increases proportionally with the buffer size. RAM storage is used for buffering long data packets. Sequential addresses can be generated using simple counters while accessing the buffer contents.

Variable-length queues are used to store data or event information when it is necessary to respond to the events at a later time. The system cannot process incoming data immediately after an event, because dedicating processor time to a single task for long periods may hinder response to other events. The following figure shows the enqueue and dequeue operations on a buffer utilizing two pointers.

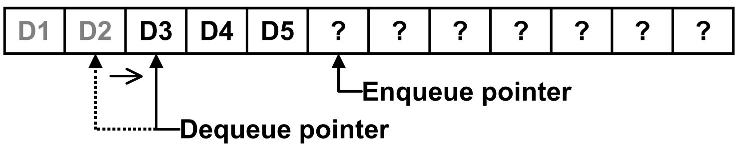
Initial condition - there is no data stored in the queue:



Enqueue operation (D5 is stored):



Dequeue operation (D2 is retrieved):



Initially, enqueue and dequeue pointers have the same address indicating that there is no data stored in the queue. The new incoming item, **D5**, is stored at the location addressed by the enqueue pointer, and the enqueue pointer advances to the next location available in the buffer. When the processor time is available, the previously stored item, **D2**, is retrieved and processed, and the dequeue pointer advances to the next stored item. In the figure, **D3**, **D4**, and **D5**, are the items that are waiting to be processed after the dequeue operation. If the dequeue pointer catches up with the enqueue pointer (when both pointers have the same address again), then this means that the queue is empty.

6.2.1 Circular Queues

It is obvious in the figure given above that the enqueue operation will fail when the empty locations available in the buffer are all used up. Circular queues reuse dequeued item locations moving the pointers back to the first location when the end of the buffer is reached. If the processor is too slow to retrieve items faster then the storage rate, then more and more unprocessed items remain in the queue. If the enqueue pointer catches up with the dequeue pointer, then this results in a **buffer overrun error** since there are no more available locations to store new items.

The following C functions implement the circular queue structure for 32-bit signed integer numbers.

Circular queue example 1: Circular buffer for 32-bit integers.

```
// Global declarations:
signed long int   Qbuffer[QbufferSize];  // queue buffer storage
signed long int   *EnQptr;  // enqueue pointer
signed long int   *DeQptr;  // dequeue pointer
signed long int   *EOBptr;  // pointer to end of the buffer

// Initialization function does not return any value.
void InitQueue(void)
// Initializes enqueue and dequeue pointers.
{ EnQptr = Qbuffer;
  DeQptr = Qbuffer;
  EOBptr = Qbuffer;
  EOBptr += QbufferSize;
}

unsigned char EnQueue(signed long int  *pDataIn)
/* Stores an item into the queue buffer.
   pDataIn points to the data to be queued.
   Returned values are:
   0 => enqueue operation is successful
   1 => buffer overrun - no space available in the buffer
*/
{
signed long int   *TempQptr;  // temporary pointer storage

  TempQptr = EnQptr;  // save current enqueue pointer
  EnQptr ++;  // increment enqueue pointer
  if (EnQptr == EOBptr)  // check if reached end of buffer
    EnQptr = Qbuffer;  // go back to the first location
  if (EnQptr == DeQptr)  // check for buffer overrun
  { EnQptr = TempQptr;  // restore the original EnQptr address
    return (unsigned char)1;  // buffer overrun
  }
  else
  { *TempQptr = *pDataIn;  // copy input data to buffer
    return (unsigned char)0;  // enqueue operation is successful
  }
}  // end of function EnQueue
```

```
unsigned char DeQueue(signed long int  *pDataOut)
/* Retrieves an item from the queue buffer.
   Retrieved item is stored at the location pointed to by pDataOut.
   Returned values are:
   0 => dequeue operation is successful
   1 => there are no queued items
*/
{
  if (DeQptr == EnQptr)
    return (unsigned char)1;  // no items available in the queue
  else
  { *pDataOut = *DeQptr;  // copy data to output location
    DeQptr ++;  // increment dequeue pointer
    if (DeQptr == EOBptr)  // check if reached end of buffer
      DeQptr = Qbuffer;  // go back to the first location
    return (unsigned char)0;  // dequeue operation is successful
  }
}  // end of function DeQueue
```

The functions given above can be adapted for any data type by replacing the existing "**signed long int**" with the new data type in the declarations. These functions require all data items to have the same fixed size. A more flexible circular queue structure can be formed allowing data items to have variable size.

Circular queue example 2: Circular buffer for variable data size.

```
// First member of all data structures involved in the queue operations
// specify the data length in bytes.
typedef struct
{ unsigned char       DTsize;   // data length in bytes (max. 255)
  unsigned short int  Member1;  // first member of data item
  signed long int     Member2;  // second member of data item
  . . .               . . .
} MyDataStruct;
```

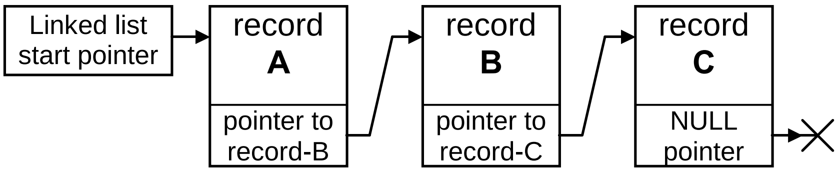
Following are the modified queue functions that can handle variable data lengths.

```
// Global declarations:
unsigned char        Qbuffer[QbufferSize];  // queue buffer storage
unsigned char       *EnQptr;    // enqueue pointer
unsigned char       *DeQptr;    // dequeue pointer
unsigned char       *EOBptr;    // pointer to end of the buffer
unsigned short int  NBavail;  // number of bytes available

void InitQueue(void)
// Initializes enqueue and dequeue pointers.
{ EnQptr = Qbuffer;
  DeQptr = Qbuffer;
  EOBptr = Qbuffer;
  EOBptr += QbufferSize;
  NBavail = QbufferSize;
}

// Pointer to void is compatible with pointer to any data type.
unsigned char EnQueue(void  *pDataIn)
/* Stores an item into the queue buffer.
   pDataIn points to the data to be queued.
   Returned values are:
   0 => enqueue operation is successful
   1 => buffer overrun - no space available in the buffer
*/
```

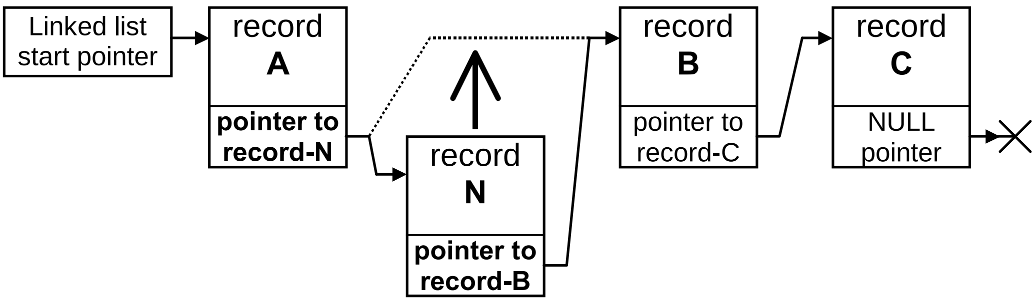
```
{
unsigned char  NByte;  // number of bytes in data item
unsigned char  i;      // byte index
unsigned char  *Bptr;  // byte pointer

  NByte = *(unsigned char *)pDataIn;  // get the number of bytes
  if (NByte > NBavail)  // check for buffer overrun
    return (unsigned char)1;  // buffer overrun error
  else
  { Bptr = (unsigned char *)pDataIn;  // copy input pointer
    for (i = Nbyte; i > 0; i--)  // copy data from input location
    { *EnQptr = *Bptr;
      EnQptr ++;  // increment enqueue pointer
      if (EnQptr == EOBptr)  // check if reached end of buffer
        EnQptr = Qbuffer;  // go back to the first location
      Bptr ++;  // increment input pointer
    }
    NBavail -= Nbyte;  // less bytes are available now
    return (unsigned char)0;  // enqueue operation is successful
  }
}  // end of function EnQueue

unsigned char DeQueue(void  *pDataOut)
/* Retrieves an item from the queue buffer.
   Retrieved item is stored at the location pointed to by pDataOut.
   Storage available at pDataOut must be big enough to handle all
   possible data types.
   Returned values are:
   0 => dequeue operation is successful
   1 => there are no queued items
*/
{
unsigned char  NByte;  // number of bytes in data item
unsigned char  i;      // byte index
unsigned char  *Bptr;  // byte pointer

  if (DeQptr == EnQptr)
    return (unsigned char)1;  // no items available in the queue
  else
  { NByte = *DeQptr;  // get the number of bytes
    Bptr = (unsigned char *)pDataOut;  // copy output pointer
    for (i = Nbyte; i > 0; i--)  // copy data to output location
    { *Bptr = *DeQptr;
      DeQptr ++;  // increment dequeue pointer
      if (DeQptr == EOBptr)  // check if reached end of buffer
        DeQptr = Qbuffer;  // go back to the first location
      Bptr ++;  // increment output pointer
    }
    NBavail += Nbyte;  // more bytes are available now
    return (unsigned char)0;  // dequeue operation is successful
  }
}  // end of function DeQueue
```

6.3 Linked Lists

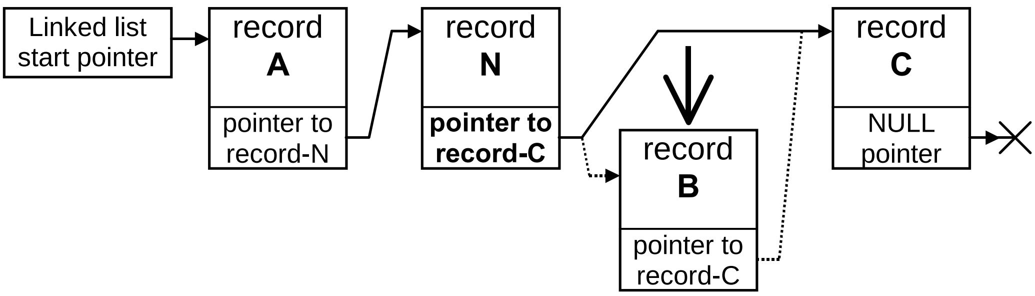
A linked list contains a sequence of data records that are flexible organizers of data storage. A linked list can be broken into pieces and attached to other linked lists easily. It is possible to insert a new data record into a linked list, or to remove an existing data record without shifting the actual memory contents. Elements of a linked list are connected to each other through pointers. Following figure shows a linked list with three data records.



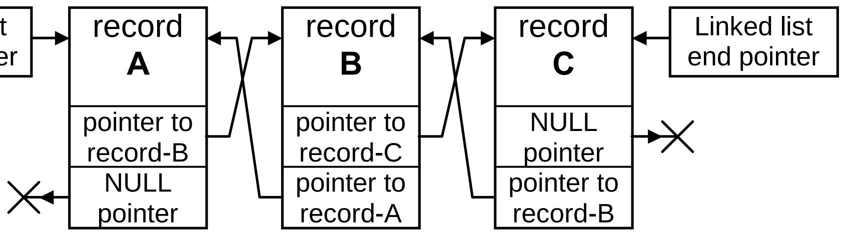
The start pointer has the address of the first record. Every record in the list contains a pointer telling us where to find the next record, except for the last record that has a NULL pointer. The NULL pointer is a predefined address constant (i.e. 0x0000) which is an invalid address for all practical purposes. All data records can be located anywhere in the available memory space. A new data record can be inserted between the records **A** and **B** simply by assigning new addresses to the pointers as shown below.



The updated pointers are identified with boldface letters in the figure. Similarly, an existing record can be deleted by rearranging the chain of links through the pointers.



A circular linked list can be obtained by writing the address of the first record into the pointer field of the last record. Circular link lists are typically used for implementing circular queues or buffers. A **double linked list** allows tracing of data records in both directions as shown below:



Many other linked structures are possible as long as the access through pointers to all data records is provided in a consistent way. As an example, **trees** are another type of data structures that are organized in a hierarchical shape, where each data record contains two or more children.

6.3.1 Linked List Implementation

The following data structure is an example of data records that can be used in making a link list.

Linked list example 1: Single linked list using dynamic memory allocation functions.

```
// Define structure data type for the linked list record:
typedef struct
{ unsigned char   DataType;  // sample record data
  . . .           . . .      // other structure members
  . . .           . . .      // other structure members
  void           *pNext;     // pointer to next record
} LLrecord;
```

The two functions given below insert a new data record, and delete an existing record in the linked list.

```
void InsertAfter(LLrecord  *pPrevRec)
/* Inserts a new record into the linked list after the element
   pointed to by pPrevRec.
*/
{
LLrecord  *pNewRec;

// sizeof() function returns the number of bytes in a datatype.
  pNewRec = malloc(sizeof(LLrecord));  // allocate memory
// New record will point to the record that follows it.
  pNewRec->pNext = pPrevRec->pNext;  // update new record's pointer
// Previous record will point to the new record.
  pPrevRec->pNext = pNewRec;  // update previous record's pointer
}  // end of function InsertAfter()

void DeleteAfter(LLrecord  *pPrevRec)
/* Deletes the record after the element pointed to by pPrevRec. */
{
LLrecord  *pDelete;

  pDelete = pPrevRec->pNext;  // pointer to the record to be deleted
  if (pDelete != NULL)  // cannot delete if reached the end
  { pPrevRec->pNext = pDelete->pNext;  // update the previous
                                       // record's pointer
    free(pDelete);  // deallocate memory
  }
}  // end of function DeleteAfter()
```

These functions make use of the **dynamic memory allocation** functions, `malloc()` and `free()` available in standard C. The `malloc()` function returns a pointer to the available memory location in the **heap** which is the general purpose system memory. Dynamic memory allocation is not applicable in most embedded systems, because the system resources are limited and memory is not large enough to support these operations. Specifically:

1. Dynamic memory allocation requires **garbage collection**. The parts of heap memory deallocated through the `free()` function are not immediately available for reallocation. Reallocation is possible after the garbage collection procedure performs its periodic maintenance work on the heap. Typical embedded programs run forever and the heap maintenance task can be too complex for an embedded processor.

2. The heap memory becomes **fragmented** after several allocation and deallocation operations. The free parts of the heap memory may be too small for any practical purpose.

In a typical embedded system, dynamic memory allocation may not be affordable, but the memory should be large enough for buffering functions required for consistent system operation. It is better to allocate static memory for a linked list when the maximum memory required for all data records is known in an embedded system. The following data structure and the related functions make use of a global array of records to implement a double linked list.

Linked list example 2: Double (bidirectional) linked list using a static array of records as the reserved memory.

```
// Define structure data type for the double linked list record:
typedef struct
{ unsigned char       DataType;  // sample record data
  . . .               . . .      // other structure members
  . . .               . . .      // other structure members
  void               *pNext;     // pointer to next record
  void               *pPrev;     // pointer to previous record
} LLrecord;
LLrecord   LLreserve[MaxNumberOfRecord];  // reserve storage for records
LLrecord  *pFreeRec;   // pointer to list of unused records
LLrecord  *pFirstRec;  // pointer to first record
LLrecord  *pLastRec;   // pointer to last record

void InitLL(void)
/* Initializes the linked list of unused records, and creates an empty
   linked list. */
{
LLrecord  *pRecord;
unsigned short int  i;

  pRecord = pFreeRec = LLreserve;  // pointer to first unused record
// Link all unused records in the LLreserve array.
  for (i = MaxNumberOfRecord; i > 0; i--)
  { pRecord->pNext = pRecord + 1;
    pRecord ++;
  }
  pRecord --;  // go back to the last record
  pRecord->pNext = NULL;  // terminate the list of unused records
// Initially there are no records in the linked list.
  pFirstRec = pLastRec = NULL;
}  // end of function InitLL()

unsigned char InsertAfter(LLrecord  *pPrevRec)
/* Inserts a new record into the double linked list after the element
   pointed to by pPrevRec.  Returned values are:
   0 => insert operation is successful
   1 => no unused records available in the list of free records
*/
{
LLrecord  *pNewRec;
LLrecord  *pNextRec;

// Check if there are any records left in the list of free records.
  if (pFreeRec == NULL)
```

```
    return 1;
  else
  { pNewRec = pFreeRec;  // get an unused record
    pFreeRec = pNewRec->pNext;  // update the free list pointer
// Check if the linked list is currently empty.
    if (pFirstRec == NULL)
    { pFirstRec = pLastRec = pNewRec;  // this will be the only record
      pNewRec->pNext = NULL;  // no other records
      pNewRec->pPrev = NULL;
    }
    else
// Establish links with the previous and next records.
    { pNextRec = pPrevRec->pNext;
      pPrevRec->pNext = pNewRec;  // previous record to new record
      pNewRec->pPrev = pPrevRec;  // new record to previous record
      pNewRec->pNext = pNextRec;  // new record to next record
// Check if inserting at the end of the linked list.
      if (pNextRec == NULL)
        pLastRec = pNewRec;  // this will be the last record
      else
        pNextRec->pPrev = pNewRec;  // link next record to new record
    }
    return 0;
  }
}  // end of function InsertAfter()

unsigned char DeleteRecord(LLrecord  *pRecord)
/* Deletes the linked list record pointed to by pRecord.
   Returned values are:
   0 => delete operation is successful
   1 => linked list is empty
*/
{
LLrecord  *pNextRec;
LLrecord  *pPrevRec;

// Check if the linked list is currently empty.
  if (pFirstRec == NULL)
    return 1;  // there are no records to delete
  else
  { pNextRec = pRecord->pNext;
    pPrevRec = pRecord->pPrev;
// Check if this is the LAST record of the linked list.
    if (pNextRec == NULL)
      pLastRec = pPrevRec;  // update the last record pointer
    else
      pNextRec->pPrev = pPrevRec;  // update next record's link
// Check if this is the FIRST record of the linked list.
    if (pPrevRec == NULL)
      pFirstRec = pNextRec;  // update the first record pointer
    else
      pPrevRec->pNext = pNextRec;  // update previous record's link
// Return the deleted record back to the list of free records.
    pRecord->pNext = pFreeRec;
    pFreeRec = pRecord;
    return 0;
  }
}  // end of function DeleteRecord()
```