queue.inc

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                              ;
;                                    QUEUE                                     ;
;                        Queue Routine Include Definitions                     ;
;                                   EE/CS 51                                   ;
;                                 Archan Luhar                                 ;
;                                 TA: Joe Greef                                ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Defines queue metadata byte offset constants and test function constants

QUEUE_TEST_LENGTH                   EQU     256

; Number of bytes of metadata before the queue elements data starts
QUEUE_QUEUE_OFFSET          EQU     7

; Maximum number of bytes of queue data in the queue struct.
; The queue struct must me initialized with less bytes than this max number.
; E.g. 256 word sized elements = 512 bytes: good
; E.g. 512 byte sized elements = 512 bytes: good
; E.g. 512 word sized elements = 1024 bytes: good
; E.g. 1024 word sized elements = 2048 bytes: not good
QUEUE_MAX_BYTES             EQU     1024

; Defines the number of bytes in the two possible element sizes byte and word
ELEM_BYTE_SIZE              EQU     1
ELEM_WORD_SIZE              EQU     2


; Queue Structure which defines the metadata and the start of the queue
; elements data.
; elem_size:    1 if each element is byte, 2 if each element is a word
; len:          Max number of elements in queue
; head_index:     Number of elements offset from start of queue elements data
; count:        Current number of elements in the queue.
; queue:        Start of queue elements.
queueSTRUC STRUC
    elem_size   DB  ?
    len         DW  ?
    head_index  DW  ?
    count       DW  ?
    queue       DB  QUEUE_MAX_BYTES DUP (?)
queueSTRUC ENDS
```

```
45
46      NAME    HW3MAIN
47
48  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
49  ;                                                                               ;
50  ;                                   HW3MAIN                                     ;
51  ;                               Homework 3 Main Loop                            ;
52  ;                                    EE/CS 51                                   ;
53  ;                                   Archan Luhar                                ;
54  ;                                   TA: Joe Greef                               ;
55  ;                                                                               ;
56  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
57
58  ; Description:      This program allocates space for myQueue and calls my own
59  ;                   defined MyQueueTest which is used for stepping through
60  ;                   element enqueuing and dequeuing to check correct memory.
61  ;                   This program also calls the QueueTest function provided by
62  ;                   Glen.
63  ;
64  ; Input:            None.
65  ; Output:           None.
66  ;
67  ; User Interface:   None. User can set breakpoint at MyQueueTest to step through
68  ;                   sample queue additions and removals to see changes in
69  ;                   the memory storing the queue.
70  ;                   If QueueTest succeeds, infinite loop occurs at
71  ;                   breakpoint hw3test.QueueGood.
72  ;
73  ; Error Handling:   If QueueTest fails, infinite loop occurs at breakpoint
74  ;                   hw3test.QueueError.
75  ;
76  ; Algorithms:       None.
77  ; Data Structures:  Queue struct is defined in queue.inc. It uses a cyclic array
78  ;
79  ; Known Bugs:       None.
80  ; Limitations:      There must be less than 1024 bytes of elements.
81  ;
82  ; Revision History:
83  ;    11/02/13  Archan Luhar     Created hw3main.asm. Contains main function
84  ;                               that calls test functions. Also allocates
85  ;                               queue struct in DS.
86
87
88  ; Include file defines queue metadata offset constants
89  $INCLUDE(queue.inc)
90
91
92  CGROUP  GROUP   CODE
93  DGROUP  GROUP   DATA, STACK
94
95
96
97  CODE    SEGMENT PUBLIC 'CODE'
98
99
100     ASSUME  CS:CGROUP, DS:DGROUP
101
```

```
102
103
104 ;external function declarations
105
106     EXTRN   QueueInit:NEAR
107     EXTRN   QueueEmpty:NEAR
108     EXTRN   QueueFull:NEAR
109     EXTRN   Dequeue:NEAR
110     EXTRN   Enqueue:NEAR
111     EXTRN   QueueTest:NEAR
112
113
114
115 START:
116
117 MAIN:
118     MOV     AX, DGROUP              ;initialize the stack pointer
119     MOV     SS, AX
120     MOV     SP, OFFSET(DGROUP:TopOfStack)
121
122     MOV     AX, DGROUP              ;initialize the data segment
123     MOV     DS, AX
124
125     MOV     SI, OFFSET(myQueue)     ; Let SI be the pointer to the queue
126     MOV     AX, QUEUE_TEST_LENGTH   ; Set size to that defined in inc
127     MOV     BL, ELEM_BYTE_SIZE      ; Set element size to byte
128     CALL    QueueInit               ; Initialize the queue
129
130     CALL    MyQueueTest             ; Test out the queue briefly
131
132     MOV     CX, QUEUE_TEST_LENGTH   ; Pass the queue and queue length to
133     CALL    QueueTest               ; provided test function.
134
135
136 ; Enqueues a bunch of numbers to see if properly stored. Must use debugger
137 ; to verify queue data in memory at SI.
138 MyQueueTest:
139     MOV AL, 1002H;
140     CALL Enqueue;
141     MOV AX, 3004H;
142     CALL Enqueue;
143     MOV AX, 5006H;
144     CALL Enqueue;
145     MOV AX, 7008H;
146     CALL Enqueue;
147     MOV AX, 9000H;
148     CALL Enqueue;
149     CALL QueueEmpty;
150     CALL QueueFull;
151     CALL Dequeue;
152
153 DATA SEGMENT PUBLIC 'DATA'
154     myQueue   queueSTRUC  <>
155 DATA ENDS
156
157
158 STACK SEGMENT STACK 'STACK'
```

```
159         DB      80 DUP ('Stack ')       ;240 words
160     TopOfStack      LABEL   WORD
161     STACK  ENDS
162
163
164     CODE ENDS
165         END START
```

```

    NAME QUEUE
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                            ;
;                                  QUEUE                                     ;
;                              Queue Routines                                ;
;                                EE/CS 51                                    ;
;                               Archan Luhar                                 ;
;                               TA: Joe Greef                                ;
;                                                                            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; Description:      This file contains several routines to manipulate and read
;                   data from a queues structure in memory:
;                   QueueInit, QueueEmpty, QueueFull, Dequeue, Enqueue
;
; Input:            None.
; Output:           None.
;
; User Interface:   None.
;
; Error Handling:   Enqueing and dequeuing block until queue is valid.
;
; Algorithms:       None.
; Data Structures:  Queue struct is defined in queue.inc. It uses a cyclic array
;
; Known Bugs:       None.
; Limitations:      There must be less than 1024 bytes of elements.
;
; Revision History:
;    10/28/13  Archan Luhar     Initial outline.
;    11/02/13  Archan Luhar     Finished HW2. Passes tests.


; Include file defines queue struct and offset constants
$INCLUDE(queue.inc)


CGROUP  GROUP   CODE
CODE SEGMENT PUBLIC 'CODE'
    ASSUME  CS:CGROUP



; QueueInit
;
; Description:      This function is used to create a queue of a given length
;                   and given element size at a given address.
;
; Operation:        This function writes the meta data of the queue in the first
;                   byte and three words of the queue: the size of each element,
;                   the max number of elements, the index of the head (0), and
;                   the count of elements in the queue also initialized to 0.
;                   The start of the queue elements would be the eigth byte.
;
; Arguments:        AX - the length, max number of elements in the queue.
```

```
223   ;                   SI - the location at which to initialize the the queue.
224   ;                   BL - size of each element (0: bytes, 1: words)
225   ;
226   ; Return Value:     None.
227   ;
228   ; Local Variables:  None.
229   ;
230   ; Shared Variables: None.
231   ; Global Variables: None.
232   ;
233   ; Input:            None.
234   ; Output:           None.
235   ;
236   ; Error Handling:   None.
237   ;
238   ; Algorithms:       None.
239   ;
240   ; Data Structures:  Cyclic array
241   ;
242   ; Registers Used:   AX (return value)
243   ;
244   ; Stack Depth:      0
245   ;
246   ; Author:           Archan Luhar
247   ; Last Modified:    11/02/2013
248   ;
249   ;
250   ; Pseudo Code
251   ; -----------
252   ;   queue.elem_size = size ? 2 : 1  ; queue's size: word if nonzero, byte if 0
253   ;   queue.len = len                 ; set queue's length
254   ;   queue.head_index = 0            ; set queue's head index
255   ;   queue.count = 0                 ; set queue's count of number of elements
256   ;
257   ;   queueSize = len * queue.elem_size
258
259   QueueInit   PROC    NEAR
260               PUBLIC  QueueInit
261
262   InitQueueInit:
263       CMP BL, 0                   ; Check the argument size
264       JE SetQueueSizeByte         ; If zero, then set to byte size element
265
266   SetQueueSizeWord:
267       MOV  [SI].elem_size, ELEM_WORD_SIZE     ; If non-zero, element size is word.
268       JMP SetQueueLength                      ; Jump over setting size to byte.
269
270   SetQueueSizeByte:
271       MOV  [SI].elem_size, ELEM_BYTE_SIZE
272       ; JMP SetQueueLength;
273
274   SetQueueLength:
275       MOV [SI].len, AX            ; Set the number of elements from AX argument
276
277   SetQueueHeadAndCount:
278       MOV [SI].head_index, 0      ; Initialize head index to 0
279       MOV [SI].count, 0           ; Initialize as empty queue having count 0 elems
```

```
280
281   EndQueueInit:
282       RET
283
284   QueueInit   ENDP
285
286
287
288   ; QueueEmpty
289   ;
290   ; Description:      This function is used to see if a given queue is empty.
291   ;
292   ; Operation:        This function simply looks at the word five bytes into
293   ;                   the metadata which stores the count of elements in queue.
294   ;                   Then it returns true if it is zero, else it returns false.
295   ;
296   ; Arguments:        SI - the address of the queue.
297   ;
298   ; Return Value:     ZF - 1 if empty, else 0.
299   ;
300   ; Local Variables:  None.
301   ;
302   ; Shared Variables: None.
303   ; Global Variables: None.
304   ;
305   ; Input:            None.
306   ; Output:           None.
307   ;
308   ; Error Handling:   None.
309   ;
310   ; Algorithms:       None.
311   ;
312   ; Data Structures:  Cyclic array
313   ;
314   ; Registers Used:   ZF
315   ;
316   ; Stack Depth:      0
317   ;
318   ; Author:           Archan Luhar
319   ; Last Modified:    10/28/2013
320   ;
321   ;
322   ; Pseudo Code
323   ; -----------
324   ;   return count == 0
325
326   QueueEmpty  PROC    NEAR
327               PUBLIC  QueueEmpty
328
329       CMP [SI].count, 0           ; If the number of elements (count) is zero
330       RET                         ; the queue is empty. ZF gets set since 0-0 = 0.
331
332   QueueEmpty  ENDP
333
334
335
336   ; QueueFull
```

```
337   ;
338   ; Description:      This function is used to see if a given queue is full.
339   ;
340   ; Operation:        This function simply looks at the word five bytes into
341   ;                   the metadata. This word stores the num of elements in queue.
342   ;                   If it equals the word stored at 1 byte into the metadata,
343   ;                   the length of the queue, then it returns true, else false.
344   ;
345   ; Arguments:        SI - the address of the queue.
346   ;
347   ; Return Value:     ZF - 1 if full, else 0.
348   ;
349   ; Local Variables:  None.
350   ;
351   ; Shared Variables: None.
352   ; Global Variables: None.
353   ;
354   ; Input:            None.
355   ; Output:           None.
356   ;
357   ; Error Handling:   None.
358   ;
359   ; Algorithms:       None.
360   ;
361   ; Data Structures:  Cyclic array
362   ;
363   ; Registers Used:   ZF
364   ;
365   ; Stack Depth:      0
366   ;
367   ; Author:           Archan Luhar
368   ; Last Modified:    11/02/2013
369   ;
370   ;
371   ; Pseudo Code
372   ; -----------
373   ;   return queue.count == queue.length
374
375   QueueFull   PROC    NEAR
376               PUBLIC  QueueFull
377
378       PUSH BX
379       MOV BX, [SI].len            ; BX contains the length of the queue
380       CMP [SI].count, BX          ; If the count == the length, the queue is full.
381       POP BX                      ; ZF gets set if full since count-len = 0.
382       RET
383
384   QueueFull   ENDP
385
386
387   ; Dequeue
388   ;
389   ; Description:      This function returns the value at the head of the queue.
390   ;                   It is a blocking function that waits until there is a value
391   ;                   if initially the queue is empty.
392   ;
393   ; Operation:        This function loops, waiting, until the queue is not empty.
```

```
394   ;                   Then, it stores the head in AL if element size is byte.
395   ;                   Else, element size is word so it stores the head in AX.
396   ;                   It then decrements the count.
397   ;                   And also it sets the head to (head + 1) mod (length - 1).
398   ;                   The location to read the value would be
399   ;
400   ; Arguments:        SI - the address of the queue.
401   ;
402   ; Return Value:     AX if element size is word, else AL - the head of queue.
403   ;
404   ; Local Variables:  None.
405   ;
406   ; Shared Variables: None.
407   ; Global Variables: None.
408   ;
409   ; Input:            None.
410   ; Output:           None.
411   ;
412   ; Error Handling:   None.
413   ;
414   ; Algorithms:       None.
415   ;
416   ; Data Structures:  Cyclic array
417   ;
418   ; Registers Used:   AX if element size is word, else AL.
419   ;
420   ; Stack Depth:      0
421   ;
422   ; Author:           Archan Luhar
423   ; Last Modified:    11/02/2013
424   ;
425   ;
426   ; Pseudo Code
427   ; -----------
428   ;   while (QueueEmpty()):    ; block while queue is empty
429   ;       continue loop
430   ;
431   ;   returnVal = queue.queue[queue.head_index * queue.elem_size]
432   ;   queue.headIndex = (queue.headIndex + 1) mod (queue.len)
433   ;   queue.count--
434   ;   return returnVal
435
436   Dequeue     PROC    NEAR
437               PUBLIC  Dequeue
438
439   BlockingDequeue:                ; Loops until queue is not empty.
440       CALL QueueEmpty             ; See if queue is empty
441       JZ BlockingDequeue          ; If zero flag is set, it is empty, block.
442       ; JNZ QueueNotEmpty
443
444   QueueNotEmpty:
445       PUSH SI                     ; Save queue pointer.
446       PUSH AX                     ; Save AX since we will use it to store the
447                                   ; computed offset for the head element.
448
449       XOR AX, AX                  ; Start with offset AX = 0
450       MOV AL, [SI].elem_size      ; AX = size of each element
```

```
451
452         PUSH DX                     ; Save DX in case MUL overflows
453         MUL [SI].head_index         ; AX = offset from start of queue elems
454                                     ;    = size * head_index
455         POP DX                      ; Restore DX
456         ADD AX, QUEUE_QUEUE_OFFSET  ; AX = size * head_index + start of queue offset
457                                     ;    = offset from start of queue pointer
458
459         CMP  [SI].elem_size, ELEM_BYTE_SIZE      ; If elem size is byte
460         JE GetQueueByte             ; Then dequeu a byte, else dequeue a word.
461
462     GetQueueWord:
463         ADD SI, AX                  ; SI = queue ptr SI + offset
464         POP AX                      ; Restore AX which we were using for offset
465         MOV AX, WORD PTR [SI]       ; Return value AX contains word element at head
466         JMP HeadAhead               ; Move the head forward to next element
467
468     GetQueueByte:
469         ADD SI, AX                  ; SI = queue ptr SI + offset
470         POP AX                      ; Restore AX which we were using for offset
471         MOV AL, BYTE PTR [SI]       ; Return value AL contains byte element at head
472         ; JMP HeadAhead             ; Move the head forward to next element
473
474     HeadAhead:
475         POP SI                      ; SI = queue ptr
476
477         PUSH AX                     ; Save return value.
478         MOV AX, [SI].head_index     ; Computing next head index in AX = head_index
479         INC AX                      ; Increment head index
480
481         PUSH BX                     ; Save BX
482         MOV BX, [SI].len            ; BX = max number of elements in queue
483
484         PUSH DX                     ; Save DX
485         MOV DX, 0                   ; Setup DX for division
486         DIV BX                      ; AX = head index / len. DX = head index mod len
487         MOV AX, DX                  ; If AX > len - 1, wrap around to 0 since
488         POP DX                      ; DX contains remainder. Return DX to original.
489
490         POP BX                      ; Return BX to original..
491
492         MOV [SI].head_index, AX     ; Save the new head index back into queue data
493         POP AX                      ; Return AX back to dequeued elem return value
494
495     EndDequeue:
496         DEC [SI].count              ; Since we've dequeued, decrement count
497         RET
498
499     Dequeue     ENDP
500
501
502
503     ; Enqueue
504     ;
505     ; Description:      This function pushes to the end of a given queue a given
506     ;                   value.
507     ;                   It is a blocking function that waits until the queue is
```

```
508   ;                   not full to enqueue the value.
509   ;
510   ; Operation:        This function loops, waiting, until the queue is not full.
511   ;                   Then it increments the count.
512   ;                   The tail index is just (head index + count) mod (length - 1)
513   ;                   If element size is byte, it stores argument from AL at tail.
514   ;                   Elese element size is word so it stores argument from AX
515   ;                   at tail.
516   ;                   The location to store would be start of queue elements +
517   ;                   tail index * element size.
518   ;
519   ; Arguments:        SI - the address of the queue.
520   ;                   AX if element size is word, else AL - value to enqueue
521   ;
522   ; Return Value:     None.
523   ;
524   ; Local Variables:  None.
525   ;
526   ; Shared Variables: None.
527   ; Global Variables: None.
528   ;
529   ; Input:            None.
530   ; Output:           None.
531   ;
532   ; Error Handling:   None.
533   ;
534   ; Algorithms:       None.
535   ;
536   ; Data Structures:  Cyclic array
537   ;
538   ; Registers Used:   None.
539   ;
540   ; Stack Depth:      0
541   ;
542   ; Author:           Archan Luhar
543   ; Last Modified:    11/02/2013
544   ;
545   ;
546   ; Pseudo Code
547   ; -----------
548   ;   while (QueueFull()):    ; block while queue is full
549   ;       continue loop
550   ;   queue.count++
551   ;   tailIndex = (queue.headIndex + queue.count) mod (queue.length)
552   ;   queue.queue[tailIndex * queue.elem_size] = value
553
554   Enqueue     PROC    NEAR
555               PUBLIC  Enqueue
556
557   BlockingEnqueue:                ; Block until queue is not full.
558       CALL QueueFull              ; Sets zero flag if full
559       JZ BlockingEnqueue          ; If zero flag is set, loop.
560       ; JNZ QueueNotFull
561
562   QueueNotFull:
563       PUSH SI                     ; Save SI queue ptr
564       PUSH AX                     ; Save argument enqueue value
```

```
565
566         MOV AX, [SI].head_index     ; AX = head index
567         ADD AX, [SI].count          ; AX = head index + count
568
569         PUSH BX                     ; Save BX to use for len
570         MOV BX, [SI].len            ; BX = len
571
572         PUSH DX                     ; Save DX
573         MOV DX, 0                   ; Setup DX for division
574         DIV BX                      ; AX = (head index + count) / len
575         MOV AX, DX                  ; AX = DX = (head index + count) mod len
576         POP DX                      ; Restore DX
577                                     ; AX now contains tail index.
578
579         POP BX                      ; Restore BX
580
581         ; multiply index by size
582         PUSH DX                     ; Save DX incase multiplication overflow
583         MUL [SI].elem_size          ; AX = tail offset from start of queue elems
584         POP DX                      ; Restore DX
585
586         ADD AX, QUEUE_QUEUE_OFFSET  ; AX = tail offset from start of queue ptr
587
588         CMP  [SI].elem_size, ELEM_BYTE_SIZE      ; If elem size is byte,
589         JE SetQueueByte             ; Write byte to queue, else write word.
590
591     SetQueueWord:
592         ADD SI, AX                  ; SI = SI queue ptr + tail offset
593         POP AX                      ; Restore enqueue value argument
594         MOV WORD PTR [SI], AX       ; Write enqueue word value argument to tail
595         JMP EndEnqueue              ; Jump over writing a byte to tail
596
597     SetQueueByte:
598         ADD SI, AX                  ; SI = SI queue ptr + tail offset
599         POP AX                      ; Restore enqueue value argument
600         MOV BYTE PTR [SI], AL       ; Write enqueue byte value argument to tail
601         ; JMP EndEnqueue
602
603     EndEnqueue:
604         POP SI                      ; Restore original queue ptr
605         INC [SI].count              ; Increment count of number of elems in queue
606         RET
607
608     Enqueue     ENDP
609
610
611     CODE ENDS
612         END
```

Makefile.mak

```
613
614   # EE/CS 51
615   # HW3 - Queue Routines
616   # Archan Luhar
617   # TA: Joe Greef
618
619   # Makefile.mak
620
621   all: assemble link locate
622
623   check:
624         asm86chk queue.asm
625         asm86chk hw3main.asm
626
627   assemble:
628         asm86 queue.asm m1 ep db
629         asm86 hw3main.asm m1 ep db
630
631   link:
632         link86 hw3main.obj,queue.obj,hw3test.obj
633
634   locate:
635         loc86 hw3main.lnk
```