

• 一.队列的引入

- 假设我们有两个任务，分别是**任务A和任务B**，我们定义一个全局变量a，任务A和B的程序均是a++，如果在FreeRTOS多任务操作系统中，**任务A和B的优先级相同的情况下**，我们来分析一下经过一次系统任务调度之后，a的值最终是多少

RTOS

int a;

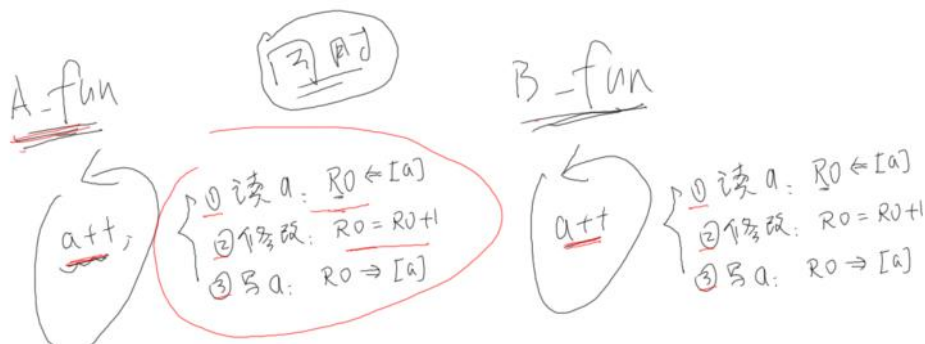
main()

```

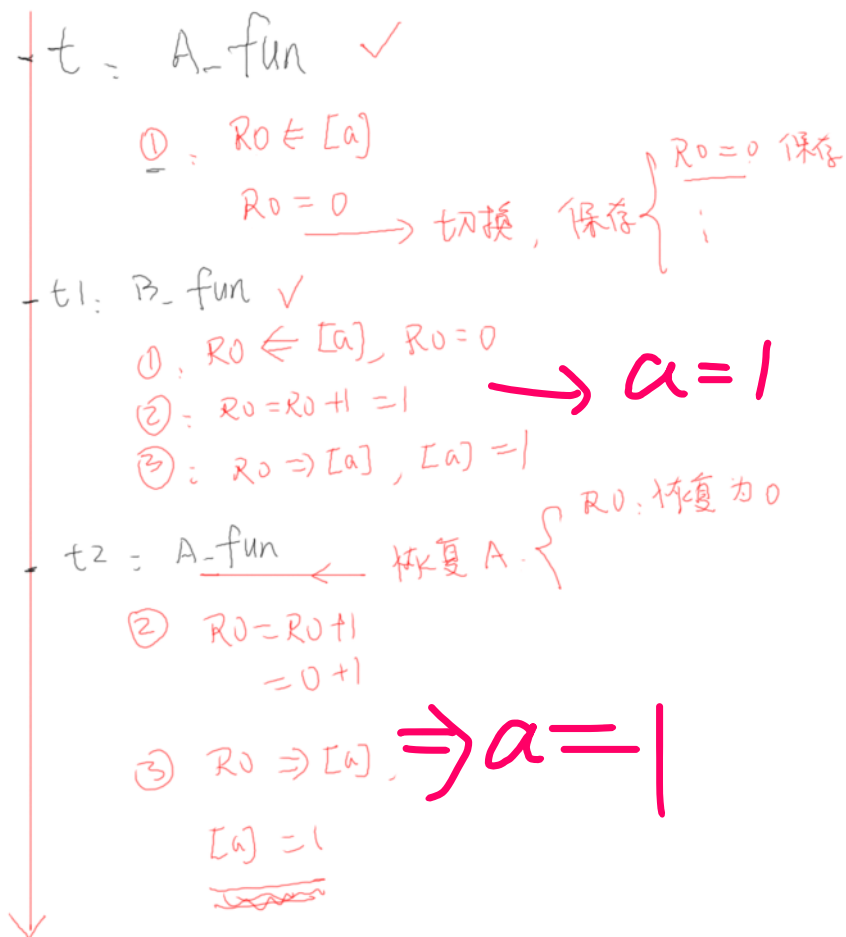
{
    creat_Task(A_fun);
    creat_Task(B_fun);
}

```

- 经过上节课的学习我们可以知道，任务中的一个操作在ARM架构中需要细分为很多步骤，就拿a++这行代码来说，首先CPU中的寄存器需要读取a的值，并将其保存到某个寄存器中，接着需要从Flash中读取a++的指令，完成对a变量值的++，最后寄存器需要将a的值重新写入到a的值中。



- 就如上图所示，任务A和B都会执行这样一个过程，那么上节课中我们说过，相同任务的优先级是采取时间片轮转的方式，当A任务执行时，假设任务A只执行到① 读取a的值时，此时一个时间片结束，或者此时任务B因为某些事件或着改变任务的优先级过来抢占任务A，此时任务让出CPU的资源，将此时的任务现场（任务上下文，寄存器的值，局部变量等）保存到对应的任务栈中，那么此时全局变量的值还未被修改，被寄存器读取的值为0并保存到寄存器中。任务B抢占任务A之后，任务B开始执行，任务B按照1, 2, 3的顺序依次执行完之后，此时变量a的值被修改成了1。任务B执行完之后，又轮到任务A执行，此时任务A从任务栈中恢复任务现场，继续执行2和3，由于之前保存到CPU寄存器的值为0，所以此时a的值是保存之前的0而不是1，任务A执行完成之后，此时a的值变为了1。明明任务A和B都执行了，也执行完成了两次a++，可是到最后变量a的值却是1。



- 所以这种情况下**无法保证数据的完整性**，我们需要的是在任务A执行时，**不要进行任务切换，不要被抢占**，即任务A执行时**免受外界的打扰**。达到一种**互斥**的效果！

○ 这里我们引入一个新的概念叫做**队列**。合理的运用队列不仅可以**保护任务在执行的过程中不会被打断**，保证了**数据的完整性**。而且还可以进行**任务间的通信和同步**等功能。

• 二.队列的核心与创建

○ 2.1队列的创建

- 队列的创建需要调用API函数**xQueueCreate(uxQueueLength, uxItemSize)**来创建一个队列。

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength,
                                   const UBaseType_t uxItemSize,
                                   const uint8_t ucQueueType )
```

- ```
Queue_t * pxNewQueue = NULL;
size_t xQueueSizeInBytes;
uint8_t * pucQueueStorage;
```

```
traceENTER_xQueueGenericCreate(uxQueueLength, uxItemSize, ucQueueType);
```

- **xQueueCreate**的函数原型就是上面这个函数，我们来分析一下
  - 首先这个函数的类型是**QueueHandle**类型，类似于任务创建时的TaskHandle类型，是创建队列成功后**返回**的一个队列句柄，用于后续对**队列的操作与管理**，如果后续需要操作队列则需要接收用对应类型的变量接收返回值，否则不需要接收。
  - 第一个参数时**UBaseType\_t**类型的变量**uxQueueLength**：该参数是表示创建**队列的深度**，即队列**最多可以储存多少个数据项**。

- 第二个参数依然是**UBaseType\_t**类型的**uxItemSize**：表示队列中**每个数据项的大小**（以**字节**为单位）。
- 第三个是**uint8\_t**类型的**ucQueueType**：用于指定队列的类型，不同的类型可能对应不同的队列行为或特性，但是在我们真正创建队列是，**这个参数并不需要给出**，由系统确定。
- 经过上面的学习，相信你已经了解了队列的创建方式，那么队列的核心是什么，为什么队列会有这么大的作用呢，接下来我们就来深入了解一下队列的核心。

## ○ 2.2队列的核心

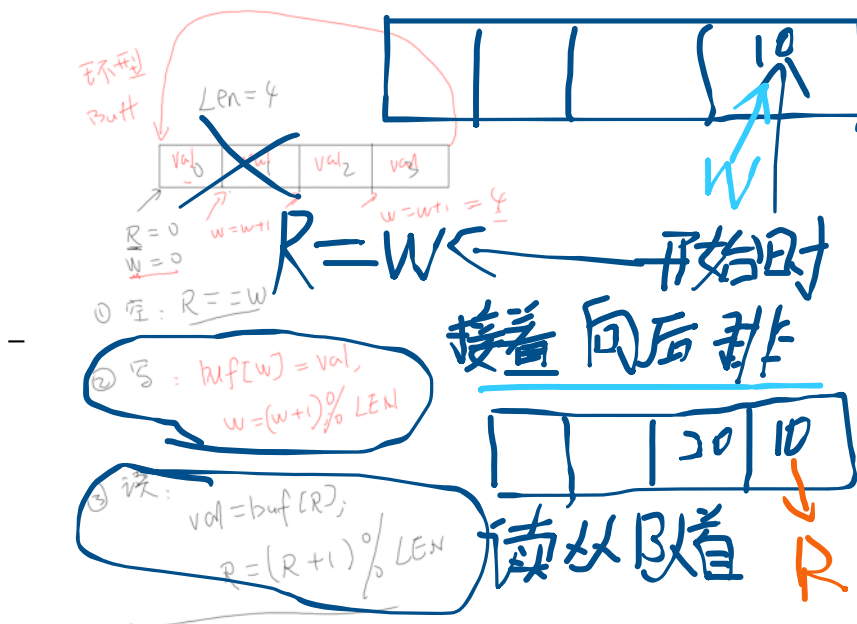
- 队列的核心包括三个部分，分别是**关中断**，**环形缓冲区**，**链表**，接下来我们逐一分析
- **2.2.1 关中断**
  - FreeRTOS为了保证任务并发访问导致的数据不一致，会暂时通过关中断来阻止任务的并发访问，确保在同一时间只有一个任务对数据进行访问，保证了数据的完整性。
  - **那么队列是怎么实现对中断的控制的呢？**
    - ◆ 在任务对队列进行读写操作时，在Free RTOS的底层都会调用API函数来开关系统的中断
    - ◆ **taskENTER\_CRITICAL();**这是一个宏定义的函数，该函数是用来关闭系统中断的API函数。这个函数的原名叫做进入临界区。就是通过调用这么一个函数，阻止了任务在执行过程中，被别的数据打断或者抢占。
    - ◆ 函数的作用
      - ◇ 1.关中断（进入临界区，屏蔽当前处理器所有可屏蔽的中断）
      - ◇ 2.禁止任务调度器转换
    - ◆ **taskEXIT\_CRITICAL();**这个函数和上面那个函数一样，也是一个由宏定义的函数。这个函数和上面那个函数是对应的，这个函数叫退出任务临界区。通过调用这个函数，恢复了任务调度器和系统中断。
    - ◆ 函数的作用：
      - ◇ 1.递减临界区嵌套计数器，当计数值为零时，恢复中断使能（退出临界区，恢复进入临界区前的中断屏蔽状态）
      - ◇ 2.允许任务调度器重新工作
    - ◆ 注意：这两个**宏必须严格配对**，即两者**必须成对使用**，即使用了多少次**taskENTER\_CRITICAL();**就要使用多少次**taskEXIT\_CRITICAL();**，否则会导致系统中断被长期屏蔽。每调用一次**taskENTER\_CRITICAL();**，临界区嵌套计数器就会+1，每调用一次**taskEXIT\_CRITICAL();**，临界区嵌套计数器就会-1，**只有当临界区嵌套计数器为0时系统的中断才会正常运行。**
- **2.2.2 环形缓冲区**
  - 关于环形缓冲区他就是一个**由用户分配的一个类似于数组的**，用于**保存数据的一个空间**，我们既可以向缓冲区中写数据，也可以从缓冲区中读数据。
  - **一.向缓冲区中写数据**
    - ◆ 环形缓冲区刚创建时里面是空的，此时可以向缓冲区中写入数据，注意开始时是从队尾开始写入数据，此时队首就是队尾（无数据时可以写，如果读的话会进入等待（阻塞态）），每写一次，**缓冲区的写位置pcWriteTo就+1**即向后移动一个，**当写到最后一个数据的是时候，如果此时再向缓冲区中写数据的话，此时pcWriteTo指向的**

位置就从指向最后一个数据位置变为指向第一个数据位置，此时队列就会进入等待（阻塞态）。所以对于写队列是pcWriteTo指向的位置我们可以总结  $W = (W+1) \% \text{Length}$ 。其中W表示写位置，Length表示队列的深度。

## 二.从缓冲区中读数据

- 当缓冲区中有数据时，我们可以从缓冲区中读取数据，与写队列类似，都队列也有一个读取位置叫pcReadFrom。开始时，pcReadFrom为0，指向队首位置，当有数据别写入队列中时，此时就可以读取写入的数据，每读取一次，pcReadFrom+1，当读到缓冲区最后一个数据时，此时如果再读取数据，则会和写队列一样，此时pcReadFrom由最后一个位置变为指向缓冲区第一个位置，此时读取队列也会进入阻塞状态，直到缓冲区中又有数据。同理，我们可以将都队列时的位置总结为： $R = (R+1) \% \text{Length}$ 。R表示读位置，Length表示队列的深度。

- 为了更好的展现环形缓冲区的读写，我们用下图来辅助理解



- 在了解完什么时数据缓冲区之后，接下来我们要结合数据缓冲区来理解队列的核心
- 由队列创建的API函数可以知道，在创建队列的时候我们需要传入两个参数，一个参数是uxQueueLength，表示创建队列的深度，即创建的队列可以保存多少个数据。
- 另外一个参数是uxItemSize，表示每个数据的大小。
- 在创建队列的时候系统不仅会创建一个环形缓冲区，而是会创建一个队列头和一个用户定义的环形缓冲区（大小），环形缓冲区经过上面的学习我们知道了，那么队列头是什么呢。我们深入队列代码的底层看一下。

```

90: typedef struct QueueDefinition /* The old naming convention
91: {
92: int8_t * pcHead; /* Points to the beginning of the
93: int8_t * pcWriteTo; /* Points to the free next place
94: union
95: {
96: QueuePointers_t xQueue; /* Data required exclusive
97: SemaphoreData_t xSemaphore; /* Data required exclusive
98: } u;
99: List_t xTasksWaitingToReceive; /* List of tasks that
100: List_t xTasksWaitingToSend; /* List of tasks that
101: volatile UBaseType_t uxMessagesWaiting; /* The number of
102: UBaseType_t uxLength; /* The length of the
103: UBaseType_t uxItemSize; /* The size of each
104: volatile int8_t * cRxLock; /* Stores the number
105: volatile int8_t * cTxLock; /* Stores the number
106: #if ((configSUPPORT_STATIC_ALLOCATION == 1) && (config
107: uint8_t ucStaticallyAllocated; /* Set to pdTRUE if the
108: #endif
109: #if (configQUEUE_SEM_SETS == 1)
110: struct QueueDefinition * pxQueueSetContainer;
111: #endif
112: #if (configQUEUE_TRACE_FACILITY == 1)
113: UBaseType_t uxQueueNumber;
114: uint8_t ucQueueType;
115: #endif
116: } QueueDefinition_t;
117: } QueueDefinition_t;
118: #endif
119: #endif
120: #endif
121: #endif
122: #endif
123: #endif
124: #endif
125: #endif
126: #endif
127: #endif
128: #endif
129: #endif
130: #endif
131: #endif
132: #endif
133: #endif
134: #endif
135: #endif
136: #endif
137: #endif
138: #endif
139: #endif
140: #endif
141: #endif
142: #endif
143: #endif
144: #endif
145: #endif
146: #endif
147: #endif
148: #endif
149: #endif
150: #endif
151: #endif
152: #endif
153: #endif
154: #endif
155: #endif
156: #endif
157: #endif
158: #endif
159: #endif
160: #endif
161: #endif
162: #endif
163: #endif
164: #endif
165: #endif
166: #endif
167: #endif
168: #endif
169: #endif
170: #endif
171: #endif
172: #endif
173: #endif
174: #endif
175: #endif
176: #endif
177: #endif
178: #endif
179: #endif
180: #endif
181: #endif
182: #endif
183: #endif
184: #endif
185: #endif
186: #endif
187: #endif
188: #endif
189: #endif
190: #endif
191: #endif
192: #endif
193: #endif
194: #endif
195: #endif
196: #endif
197: #endif
198: #endif
199: #endif
200: #endif
201: #endif
202: #endif
203: #endif
204: #endif
205: #endif
206: #endif
207: #endif
208: #endif
209: #endif
210: #endif
211: #endif
212: #endif
213: #endif
214: #endif
215: #endif
216: #endif
217: #endif
218: #endif
219: #endif
220: #endif
221: #endif
222: #endif
223: #endif
224: #endif
225: #endif
226: #endif
227: #endif
228: #endif
229: #endif
230: #endif
231: #endif
232: #endif
233: #endif
234: #endif
235: #endif
236: #endif
237: #endif
238: #endif
239: #endif
240: #endif
241: #endif
242: #endif
243: #endif
244: #endif
245: #endif
246: #endif
247: #endif
248: #endif
249: #endif
250: #endif
251: #endif
252: #endif
253: #endif
254: #endif
255: #endif
256: #endif
257: #endif
258: #endif
259: #endif
260: #endif
261: #endif
262: #endif
263: #endif
264: #endif
265: #endif
266: #endif
267: #endif
268: #endif
269: #endif
270: #endif
271: #endif
272: #endif
273: #endif
274: #endif
275: #endif
276: #endif
277: #endif
278: #endif
279: #endif
280: #endif
281: #endif
282: #endif
283: #endif
284: #endif
285: #endif
286: #endif
287: #endif
288: #endif
289: #endif
290: #endif
291: #endif
292: #endif
293: #endif
294: #endif
295: #endif
296: #endif
297: #endif
298: #endif
299: #endif
300: #endif
301: #endif
302: #endif
303: #endif
304: #endif
305: #endif
306: #endif
307: #endif
308: #endif
309: #endif
310: #endif
311: #endif
312: #endif
313: #endif
314: #endif
315: #endif
316: #endif
317: #endif
318: #endif
319: #endif
320: #endif
321: #endif
322: #endif
323: #endif
324: #endif
325: #endif
326: #endif
327: #endif
328: #endif
329: #endif
330: #endif
331: #endif
332: #endif
333: #endif
334: #endif
335: #endif
336: #endif
337: #endif
338: #endif
339: #endif
340: #endif
341: #endif
342: #endif
343: #endif
344: #endif
345: #endif
346: #endif
347: #endif
348: #endif
349: #endif
350: #endif
351: #endif
352: #endif
353: #endif
354: #endif
355: #endif
356: #endif
357: #endif
358: #endif
359: #endif
360: #endif
361: #endif
362: #endif
363: #endif
364: #endif
365: #endif
366: #endif
367: #endif
368: #endif
369: #endif
370: #endif
371: #endif
372: #endif
373: #endif
374: #endif
375: #endif
376: #endif
377: #endif
378: #endif
379: #endif
380: #endif
381: #endif
382: #endif
383: #endif
384: #endif
385: #endif
386: #endif
387: #endif
388: #endif
389: #endif
390: #endif
391: #endif
392: #endif
393: #endif
394: #endif
395: #endif
396: #endif
397: #endif
398: #endif
399: #endif
400: #endif
401: #endif
402: #endif
403: #endif
404: #endif
405: #endif
406: #endif
407: #endif
408: #endif
409: #endif
410: #endif
411: #endif
412: #endif
413: #endif
414: #endif
415: #endif
416: #endif
417: #endif
418: #endif
419: #endif
420: #endif
421: #endif
422: #endif
423: #endif
424: #endif
425: #endif
426: #endif
427: #endif
428: #endif
429: #endif
430: #endif
431: #endif
432: #endif
433: #endif
434: #endif
435: #endif
436: #endif
437: #endif
438: #endif
439: #endif
440: #endif
441: #endif
442: #endif
443: #endif
444: #endif
445: #endif
446: #endif
447: #endif
448: #endif
449: #endif
450: #endif
451: #endif
452: #endif
453: #endif
454: #endif
455: #endif
456: #endif
457: #endif
458: #endif
459: #endif
460: #endif
461: #endif
462: #endif
463: #endif
464: #endif
465: #endif
466: #endif
467: #endif
468: #endif
469: #endif
470: #endif
471: #endif
472: #endif
473: #endif
474: #endif
475: #endif
476: #endif
477: #endif
478: #endif
479: #endif
480: #endif
481: #endif
482: #endif
483: #endif
484: #endif
485: #endif
486: #endif
487: #endif
488: #endif
489: #endif
490: #endif
491: #endif
492: #endif
493: #endif
494: #endif
495: #endif
496: #endif
497: #endif
498: #endif
499: #endif
500: #endif
501: #endif
502: #endif
503: #endif
504: #endif
505: #endif
506: #endif
507: #endif
508: #endif
509: #endif
510: #endif
511: #endif
512: #endif
513: #endif
514: #endif
515: #endif
516: #endif
517: #endif
518: #endif
519: #endif
520: #endif
521: #endif
522: #endif
523: #endif
524: #endif
525: #endif
526: #endif
527: #endif
528: #endif
529: #endif
530: #endif
531: #endif
532: #endif
533: #endif
534: #endif
535: #endif
536: #endif
537: #endif
538: #endif
539: #endif
540: #endif
541: #endif
542: #endif
543: #endif
544: #endif
545: #endif
546: #endif
547: #endif
548: #endif
549: #endif
550: #endif
551: #endif
552: #endif
553: #endif
554: #endif
555: #endif
556: #endif
557: #endif
558: #endif
559: #endif
560: #endif
561: #endif
562: #endif
563: #endif
564: #endif
565: #endif
566: #endif
567: #endif
568: #endif
569: #endif
570: #endif
571: #endif
572: #endif
573: #endif
574: #endif
575: #endif
576: #endif
577: #endif
578: #endif
579: #endif
580: #endif
581: #endif
582: #endif
583: #endif
584: #endif
585: #endif
586: #endif
587: #endif
588: #endif
589: #endif
590: #endif
591: #endif
592: #endif
593: #endif
594: #endif
595: #endif
596: #endif
597: #endif
598: #endif
599: #endif
600: #endif
601: #endif
602: #endif
603: #endif
604: #endif
605: #endif
606: #endif
607: #endif
608: #endif
609: #endif
610: #endif
611: #endif
612: #endif
613: #endif
614: #endif
615: #endif
616: #endif
617: #endif
618: #endif
619: #endif
620: #endif
621: #endif
622: #endif
623: #endif
624: #endif
625: #endif
626: #endif
627: #endif
628: #endif
629: #endif
630: #endif
631: #endif
632: #endif
633: #endif
634: #endif
635: #endif
636: #endif
637: #endif
638: #endif
639: #endif
640: #endif
641: #endif
642: #endif
643: #endif
644: #endif
645: #endif
646: #endif
647: #endif
648: #endif
649: #endif
650: #endif
651: #endif
652: #endif
653: #endif
654: #endif
655: #endif
656: #endif
657: #endif
658: #endif
659: #endif
660: #endif
661: #endif
662: #endif
663: #endif
664: #endif
665: #endif
666: #endif
667: #endif
668: #endif
669: #endif
670: #endif
671: #endif
672: #endif
673: #endif
674: #endif
675: #endif
676: #endif
677: #endif
678: #endif
679: #endif
680: #endif
681: #endif
682: #endif
683: #endif
684: #endif
685: #endif
686: #endif
687: #endif
688: #endif
689: #endif
690: #endif
691: #endif
692: #endif
693: #endif
694: #endif
695: #endif
696: #endif
697: #endif
698: #endif
699: #endif
700: #endif
701: #endif
702: #endif
703: #endif
704: #endif
705: #endif
706: #endif
707: #endif
708: #endif
709: #endif
710: #endif
711: #endif
712: #endif
713: #endif
714: #endif
715: #endif
716: #endif
717: #endif
718: #endif
719: #endif
720: #endif
721: #endif
722: #endif
723: #endif
724: #endif
725: #endif
726: #endif
727: #endif
728: #endif
729: #endif
730: #endif
731: #endif
732: #endif
733: #endif
734: #endif
735: #endif
736: #endif
737: #endif
738: #endif
739: #endif
740: #endif
741: #endif
742: #endif
743: #endif
744: #endif
745: #endif
746: #endif
747: #endif
748: #endif
749: #endif
750: #endif
751: #endif
752: #endif
753: #endif
754: #endif
755: #endif
756: #endif
757: #endif
758: #endif
759: #endif
760: #endif
761: #endif
762: #endif
763: #endif
764: #endif
765: #endif
766: #endif
767: #endif
768: #endif
769: #endif
770: #endif
771: #endif
772: #endif
773: #endif
774: #endif
775: #endif
776: #endif
777: #endif
778: #endif
779: #endif
780: #endif
781: #endif
782: #endif
783: #endif
784: #endif
785: #endif
786: #endif
787: #endif
788: #endif
789: #endif
790: #endif
791: #endif
792: #endif
793: #endif
794: #endif
795: #endif
796: #endif
797: #endif
798: #endif
799: #endif
800: #endif
801: #endif
802: #endif
803: #endif
804: #endif
805: #endif
806: #endif
807: #endif
808: #endif
809: #endif
810: #endif
811: #endif
812: #endif
813: #endif
814: #endif
815: #endif
816: #endif
817: #endif
818: #endif
819: #endif
820: #endif
821: #endif
822: #endif
823: #endif
824: #endif
825: #endif
826: #endif
827: #endif
828: #endif
829: #endif
830: #endif
831: #endif
832: #endif
833: #endif
834: #endif
835: #endif
836: #endif
837: #endif
838: #endif
839: #endif
840: #endif
841: #endif
842: #endif
843: #endif
844: #endif
845: #endif
846: #endif
847: #endif
848: #endif
849: #endif
850: #endif
851: #endif
852: #endif
853: #endif
854: #endif
855: #endif
856: #endif
857: #endif
858: #endif
859: #endif
860: #endif
861: #endif
862: #endif
863: #endif
864: #endif
865: #endif
866: #endif
867: #endif
868: #endif
869: #endif
870: #endif
871: #endif
872: #endif
873: #endif
874: #endif
875: #endif
876: #endif
877: #endif
878: #endif
879: #endif
880: #endif
881: #endif
882: #endif
883: #endif
884: #endif
885: #endif
886: #endif
887: #endif
888: #endif
889: #endif
890: #endif
891: #endif
892: #endif
893: #endif
894: #endif
895: #endif
896: #endif
897: #endif
898: #endif
899: #endif
900: #endif
901: #endif
902: #endif
903: #endif
904: #endif
905: #endif
906: #endif
907: #endif
908: #endif
909: #endif
910: #endif
911: #endif
912: #endif
913: #endif
914: #endif
915: #endif
916: #endif
917: #endif
918: #endif
919: #endif
920: #endif
921: #endif
922: #endif
923: #endif
924: #endif
925: #endif
926: #endif
927: #endif
928: #endif
929: #endif
930: #endif
931: #endif
932: #endif
933: #endif
934: #endif
935: #endif
936: #endif
937: #endif
938: #endif
939: #endif
940: #endif
941: #endif
942: #endif
943: #endif
944: #endif
945: #endif
946: #endif
947: #endif
948: #endif
949: #endif
950: #endif
951: #endif
952: #endif
953: #endif
954: #endif
955: #endif
956: #endif
957: #endif
958: #endif
959: #endif
960: #endif
961: #endif
962: #endif
963: #endif
964: #endif
965: #endif
966: #endif
967: #endif
968: #endif
969: #endif
970: #endif
971: #endif
972: #endif
973: #endif
974: #endif
975: #endif
976: #endif
977: #endif
978: #endif
979: #endif
980: #endif
981: #endif
982: #endif
983: #endif
984: #endif
985: #endif
986: #endif
987: #endif
988: #endif
989: #endif
990: #endif
991: #endif
992: #endif
993: #endif
994: #endif
995: #endif
996: #endif
997: #endif
998: #endif
999: #endif
1000: #endif
1001: #endif
1002: #endif
1003: #endif
1004: #endif
1005: #endif
1006: #endif
1007: #endif
1008: #endif
1009: #endif
1010: #endif
1011: #endif
1012: #endif
1013: #endif
1014: #endif
1015: #endif
1016: #endif
1017: #endif
1018: #endif
1019: #endif
1020: #endif
1021: #endif
1022: #endif
1023: #endif
1024: #endif
1025: #endif
1026: #endif
1027: #endif
1028: #endif
1029: #endif
1030: #endif
1031: #endif
1032: #endif
1033: #endif
1034: #endif
1035: #endif
1036: #endif
1037: #endif
1038: #endif
1039: #endif
1040: #endif
1041: #endif
1042: #endif
1043: #endif
1044: #endif
1045: #endif
1046: #endif
1047: #endif
1048: #endif
1049: #endif
1050: #endif
1051: #endif
1052: #endif
1053: #endif
1054: #endif
1055: #endif
1056: #endif
1057: #endif
1058: #endif
1059: #endif
1060: #endif
1061: #endif
1062: #endif
1063: #endif
1064: #endif
1065: #endif
1066: #endif
1067: #endif
1068: #endif
1069: #endif
1070: #endif
1071: #endif
1072: #endif
1073: #endif
1074: #endif
1075: #endif
1076: #endif
1077: #endif
1078: #endif
1079: #endif
1080: #endif
1081: #endif
1082: #endif
1083: #endif
1084: #endif
1085: #endif
1086: #endif
1087: #endif
1088: #endif
1089: #endif
1090: #endif
1091: #endif
1092: #endif
1093: #endif
1094: #endif
1095: #endif
1096: #endif
1097: #endif
1098: #endif
1099: #endif
1100: #endif
1101: #endif
1102: #endif
1103: #endif
1104: #endif
1105: #endif
1106: #endif
1107: #endif
1108: #endif
1109: #endif
1110: #endif
1111: #endif
1112: #endif
1113: #endif
1114: #endif
1115: #endif
1116: #endif
1117: #endif
1118: #endif
1119: #endif
1120: #endif
1121: #endif
1122: #endif
1123: #endif
1124: #endif
1125: #endif
1126: #endif
1127: #endif
1128: #endif
1129: #endif
1130: #endif
1131: #endif
1132: #endif
1133: #endif
1134: #endif
1135: #endif
1136: #endif
1137: #endif
1138: #endif
1139: #endif
1140: #endif
1141: #endif
1142: #endif
1143: #endif
1144: #endif
1145: #endif
1146: #endif
1147: #endif
1148: #endif
1149: #endif
1150: #endif
1151: #endif
1152: #endif
1153: #endif
1154: #endif
1155: #endif
1156: #endif
1157: #endif
1158: #endif
1159: #endif
1160: #endif
1161: #endif
1162: #endif
1163: #endif
1164: #endif
1165: #endif
1166: #endif
1167: #endif
1168: #endif
1169: #endif
1170: #endif
1171: #endif
1172: #endif
1173: #endif
1174: #endif
1175: #endif
1176: #endif
1177: #endif
1178: #endif
1179: #endif
1180: #endif
1181: #endif
1182: #endif
1183: #endif
1184: #endif
1185: #endif
1186: #endif
1187: #endif
1188: #endif
1189: #endif
1190: #endif
1191: #endif
1192: #endif
1193: #endif
1194: #endif
1195: #endif
1196: #endif
1197: #endif
1198: #endif
1199: #endif
1200: #endif
1201: #endif
1202: #endif
1203: #endif
1204: #endif
1205: #endif
1206: #endif
1207: #endif
1208: #endif
1209: #endif
1210: #endif
1211: #endif
1212: #endif
1213: #endif
1214: #endif
1215: #endif
1216: #endif
1217: #endif
1218: #endif
1219: #endif
1220: #endif
1221: #endif
1222: #endif
1223: #endif
1224: #endif
1225: #endif
1226: #endif
1227: #endif
1228: #endif
1229: #endif
1230: #endif
1231: #endif
1232: #endif
1233: #endif
1234: #endif
1235: #endif
1236: #endif
1237: #endif
1238: #endif
1239: #endif
1240: #endif
1241: #endif
1242: #endif
1243: #endif
1244: #endif
1245: #endif
1246: #endif
1247: #endif
1248: #endif
1249: #endif
1250: #endif
1251: #endif
1252: #endif
1253: #endif
1254: #endif
1255: #endif
1256: #endif
1257: #endif
1258: #endif
1259: #endif
1260: #endif
1261: #endif
1262: #endif
1263: #endif
1264: #endif
1265: #endif
1266: #endif
1267: #endif
1268: #endif
1269: #endif
1270: #endif
1271: #endif
1272: #endif
1273: #endif
1274: #endif
1275: #endif
1276: #endif
1277: #endif
1278: #endif
1279: #endif
1280: #endif
1281: #endif
1282: #endif
1283: #endif
1284: #endif
1285: #endif
1286: #endif
1287: #endif
1288: #endif
1289: #endif
1290: #endif
1291: #endif
1292: #endif
1293: #endif
1294: #endif
1295: #endif
1296: #endif
1297: #endif
1298: #endif
1299: #endif
1300: #endif
1301: #endif
1302: #endif
1303: #endif
1304: #endif
1305: #endif
1306: #endif
1307: #endif
1308: #endif
1309: #endif
1310: #endif
1311: #endif
1312: #endif
1313: #endif
1314: #endif
1315: #endif
1316: #endif
1317: #endif
1318: #endif
1319: #endif
1320: #endif
1321: #endif
1322: #endif
1323: #endif
1324: #endif
1325: #endif
1326: #endif
1327: #endif
1328: #endif
1329: #endif
1330: #endif
1331: #endif
1332: #endif
1333: #endif
1334: #endif
1335: #endif
1336: #endif
1337: #endif
1338: #endif
1339: #endif
1340: #endif
1341: #endif
1342: #endif
1343: #endif
1344: #endif
1345: #endif
1346: #endif
1347: #endif
1348: #endif
1349: #endif
1350: #endif
1351: #endif
1352: #endif
1353: #endif
1354: #endif
1355: #endif
1356: #endif
1357: #endif
1358: #endif
1359: #endif
1360: #endif
1361: #endif
1362: #endif
1363: #endif
1364: #endif
1365: #endif
1366: #endif
1367: #endif
1368: #endif
1369: #endif
1370: #endif
1371: #endif
1372: #endif
1373: #endif
1374: #endif
1375: #endif
1376: #endif
1377: #endif
1378: #endif
1379: #endif
1380: #endif
1381: #endif
1382: #endif
1383: #endif
1384: #endif
1385: #endif
1386: #endif
1387: #endif
1388: #endif
1389: #endif
1390: #endif
1391: #endif
1392: #endif
1393: #endif
1394: #endif
1395: #endif
1396: #endif
1397: #endif
1398: #endif
1399: #endif
1400: #endif
1401: #endif
1402: #endif
1403: #endif
1404: #endif
1405: #endif
1406: #endif
1407: #endif
1408: #endif
1409: #endif
1410: #endif
1411: #endif
1412: #endif
1413: #endif
1414: #endif
1415: #endif
1416: #endif
1417: #endif
1418: #endif
1419: #endif
1420: #endif
1421: #endif
1422: #endif
1423: #endif
1424: #endif
1425: #endif
1426: #endif
1427: #endif
1428: #endif
1429: #endif
1430: #endif
1431: #endif
1432: #endif
1433: #endif
1434: #endif
1435: #endif
1436: #endif
1437: #endif
1438: #endif
1439: #endif
1440: #endif
1441: #endif
1442: #endif
1443: #endif
1444: #endif
1445: #endif
1446: #endif
1447: #endif
1448: #endif
1449: #endif
```



- 观察了队列底层的代码之后，我们可以发现，队列头是一种类似于任务结构体的数据，我们就可以把它叫做**队列结构体**，接下来我们来深入分析一下队列结构体中的内容，观察一下队列结构体和环形缓冲区结合是怎么操控队列的
- **三.队列结构体**
  - ◆ 首先在结构体的最上面我们可以看到，队列结构体在**一开始就定义了环形缓冲区的写入位置pcWriteTo和读取位置pcReadFrom**，说明了队列结构体和队列的环形缓冲区有着直接的关系，事实确实如此，**在队列结构体之后紧接着的就是队列环形缓冲区**，即在创建队列的时候不仅仅只是创建**队列结构体**或者是**环形缓冲区**，而是在创建时，**二者共同构成了队列**，在创建时分配内存时，**pcNewQueue=Queue\_t + Buffer**。
  - ◆ 接着向下看，我们可以看到**两个List\_t类型的链表**，分别是**xTasksWaitingToSend（任务等待发送链表）和xTasksWaitingToReceive（任务等待接收链表）**。那么这两个链表有什么用呢，前面我们说过在读取队列和写入队列时，当缓冲区满时（写队列进入等待）或者缓冲区空时（读队列进入等待），**任务会被记录在这个两个链表中的对应链表**，但**注意只是把他们任务记录在链表中，任务并不在链表中**，因为任务处于等待状态，所以任务会从就绪态任务列表中移到阻塞态任务列表中，所以**任务实际被存储到阻塞链表中**，并不在队列链表中。
  - ◆ 对于剩余参数我们来看一下
  - ◆ 在队列结构体的开始定义了一个**pcHead**变量，此变量指向**队列存储区域的起始地址**，队列的所有数据都**存放在以该指针为起点的连续内存中**。
  - ◆ 在队列链表下面定义了一个**volatile UBaseType\_t uxMessagesWaiting**：该变量表示**待接收消息的数量**。
  - ◆ 紧接着是队列的两个参数**uxQueueLength**即队列深度，**uxQueueItem**即队列数据大小。
  - ◆ 下面的时队列锁和条件编译部分（根据配置项启用不同功能），这里就不再详述。

### ▪ 2.2.3 队列的链表

- 上面在队列结构体中，我们发现了两个队列等待链表，在上面我们已经详述过了，**这两个链表使用与记录那个任务处于等待状态**，以便于当任务满足条件时，**系统从队列等待链表中找到任务，将任务从链表中移除**，与此同时，**任务从阻塞态任务列表中被恢复到之前就绪态任务链表中**。

### ▪ 将核心结合起来看队列（综合）

- 假设有一个队列，有两个任务A（高优先级）和任务B（低优先级）写队列，有两个任务C（高优先级）和任务D（低优先级）去读队列，起始队列中并没有数据。
- **一.任务A和任务B同时向队列中写数据**
  - ◆ 因为A的优先级高，所以开始时，任务A向队列中写入数据，此时任务A的运行过程为
    - ◇ 1.**关闭FreeRTOS系统的屏蔽中断**，调用API函数让任务进入临界区
    - ◇ 2.**任务A判断队列的数据是否已满，缓冲区中是否还有空间**
      - ▶ 2.1当缓冲区中**还有空间时**，任务A**直接向队列中写入数据**，数据被保存到环



形缓冲区中。

- ▶ 2.2当缓冲区中**没有空间**时，任务A想要向队列中写入数据，发现数据已满，此时任务会有以下几个选择
  - 一.当任务不想等待时，此时**直接返回ERR**
  - 二.任务会**进入休眠**，此时会进行两步操作
    - ◆ 1.任务A会被记录到队列写等待链表 (**xTasksWaitingToSend**) 中
    - ◆ 2.任务A会从原来的就绪态列表被转移到阻塞态任务列表
- ▶ 2.3任务A阻塞等待着，当缓冲区有空间了，此时任务A会进行如下操作
  - 首先，任务A被**唤醒**
    - ◆ 任务A会从队列写等待链表 (**xTasksWaitingToSend**) 中移除
    - ◆ 任务A会由阻塞态任务列表恢复到原来的就绪态任务列表
  - 然后，任务A会向缓冲区中写数据（注意：因为任务A向缓冲区中写队列的时候数据已满，所以此时对应的写位置pcWriteTo指向的是第一个数据位）



## □ 二.任务C和任务D同时从队列中读取

- ◆ 因为C的优先级高，所以开始时，任务C先队列中读取数据，此时任务C的运行过程为
  - ◇ 1.关闭FreeRTOS系统的屏蔽中断，调用API函数让任务进入临界区
  - ◇ 2.任务A判断队列的数据是否为空，缓冲区中是否有数据
    - ▶ 2.1当缓冲区中有数据时，此时任务C**直接从队列中读取数据 (Copy)**（注意：任务从队列缓冲区中读取数据是**将数据复制**，保存到自己的缓冲区中，然后队列会将数据移除，使其不在队列当中了，看似是直接拿走的过程，但其实本质是值的传递，而不是指针传递）
    - ▶ 2.2当缓冲区中**没有数据**时，此时任务C再向队列中读取数据，此时和写队列一样会有两个选择
      - 一.任务**不等待**，**直接返回ERR**
      - 二.任务**等待**，此时任务会进行以下操作
        - ◆ 1.任务会被记录到接收数据等待链表 (**xTasksWaitingToReceive**) 中
        - ◆ 2.任务会从原来的就绪态任务链表被移动到阻塞态任务链表中

▶ 2.3当任务的条件满足时，即队列中有数据了，此时任务会进行如下操作

– 首先任务C被唤醒

- ♦ 任务C会**从队列读等待链表 (xTasksWaitingToReceive) 中移除**
- ♦ 任务C会**从阻塞态任务列表回到原来的就绪态任务列表**
- ♦ 注意，和写队列一样，当读取数据到最后一位时，此时读位置 pcReadFrom 会来到第一位



□ 三.如果等待任务超时了

- ♦ 此时任务会**直接从阻塞态任务列表被移动到原来的就绪态任务列表**

- 从上面几个方面我们已经了解了队列的创建，已经队列在底层是怎么操作的，那么学习完了队列的内部机制之后，接下来我们要学习如何使用队列。

• 三.队列的特性与使用

○ 3.1.队列的特性

▪ 1.数据的存储

往队列中写入数据是**通过字节拷贝把数据复制存储到队列中**，从队列读取数据是得把**队列中的数据拷贝删除**。

▪ 2.可被多任务存取

- 队列是具有对立权限的内核对象，并不属于或赋予任何任务，所有任务都可以向同一队列写入或读出。

▪ 3.读队列是阻塞

- 当某个任务试图读一个队列时，其可以指定一个阻塞超时时间。在这段时间中，如果队列为空，该任务将保持阻塞状态以等待队列数据有效。当其它任务或中断服务例程往其等待的队列中写入了数据，该任务将自动由阻塞态转移为就绪态。当等待的时间超过了
- 由于队列可以被多个任务读取，所以对于单个队列而言，也**可能多个任务处于阻塞状态以等待队列数据有效**。这种情况下，一旦数据有效，只能有一个任务解除阻塞，这个任务就是**所有等待任务中优先级最高的**。而**如果优先级相同，那么则是等待最久的任务**。

▪ 4.写队列时阻塞

- 同读队列一样，任务也可以写队列时指定一个阻塞超时时间。这个时间是当被写队列已满时，任务进入阻塞态以等待队列空间有效的最长时间。
- 由于队列也可以被多个任务写入，所以对于单个队列而言，也**可能多个任务处于阻塞**

**状态以等待有效的空间。**这种情况下，一旦空间有效，只能由一个任务解除阻塞，最高任务就是**所有等待任务中优先级最高的**。而**如果优先级相同，则时等待最久的任务**。

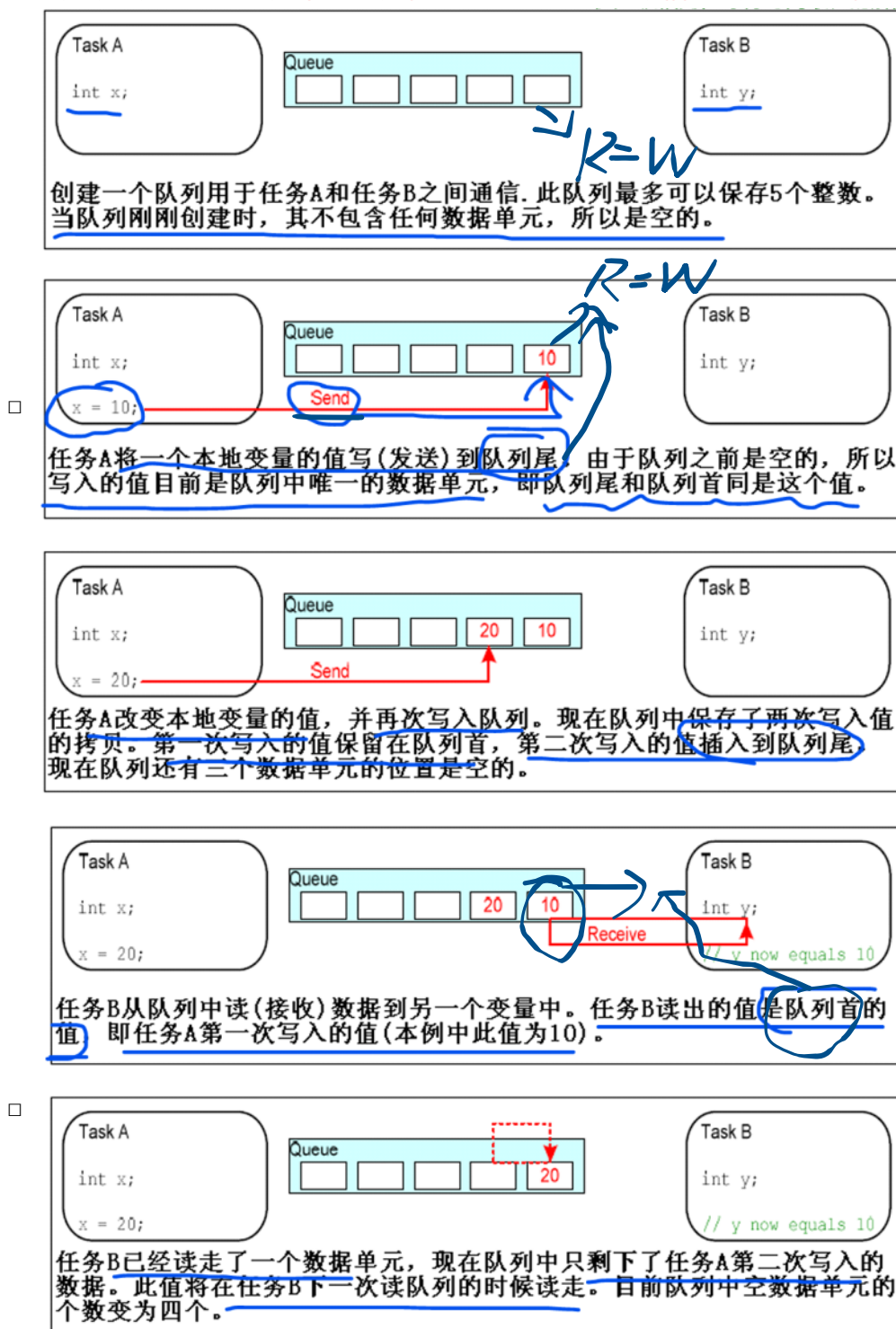


图 19 队列读写过程示例

- 以上就是队列读和写的全部过程

### ○ 3.2队列的使用

- 1.创建队列，调用API函数 `xQueueCreate( uxQueueLength,uxItemSize )`

- `xQueueCreate( uxQueueLength,uxItemSize )`API函数

- ◆ 队列在**使用前必须创建**
    - ◆ `xQueueCreate()`用于创建一个队列，并**返回一个xQueueHandle类型的队列句柄**，用于对其创建的队列进行引用
    - ◆ 在创建队列是，FreeRTOS会**从堆空间中分配内存空间**。分配的内存空间**用于存储队**



## 列数据结构本身（队列结构体）以及队列中包含的数据单元。

表 7 xQueueCreate() 参数与返回值

| 参数名           | 描述                                                                       |
|---------------|--------------------------------------------------------------------------|
| uxQueueLength | 队列能够存储的最大单元数目，即队列深度。                                                     |
| uxItemSize    | 队列中数据单元的长度，以字节为单位。                                                       |
| 返回值           | NULL 表示没有足够的堆空间分配给队列而导致创建失败。<br>非 NULL 值表示队列创建成功。此返回值应当保存下来，以作为操作此队列的句柄。 |

- 2.向队列发送数据，调用API函数xQueueSend()和xQueueSendToBack() 与以及 xQueueSendToFront() API 函数
  - xQueueSend()完全等于xQueueSendToBack()。如同函数名一样，这两个函数都是将数据发送到队列的尾部。而xQueueSendToFront() 是将数据发送到队列首
  - 注意上面这几个发送函数不能在中断服务程序中调用，如果需要在中断服务程序中调用队列发送函数则调用安全版本的 xQueueSendToFrontFromISR()与 xQueueSendToBackFromISR()，这是专门在中断服务程序中调用的API函数

表 8 xQueueSendToFront()与 xQueueSendToBack()函数参数及返回值

| 参数名           | 描述                                                                                  |
|---------------|-------------------------------------------------------------------------------------|
| xQueue        | 目标队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。                                          |
| pvItemToQueue | 发送数据的指针。其指向将要复制到目标队列中的数据单元。<br>由于在创建队列时设置了队列中数据单元的长度，所以会从该指针指向的空间复制对应长度的数据到队列的存储区域。 |
| xTicksToWait  | 阻塞超时时间。如果在发送时队列已满，这个时间即是任务处于阻塞态等待队列空间有效的最长等待时间。                                     |

如果 `xTicksToWait` 设为 0，并且队列已满，则 `xQueueSendToFront()`与 `xQueueSendToBack()`均会立即返回。

阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 `portTICK_RATE_MS` 可以用来把心跳时间单位转换为毫秒时间单位。

如果把 `xTicksToWait` 设置为 `portMAX_DELAY`，并且在 `FreeRTOSConfig.h` 中设定 `INCLUDE_vTaskSuspend` 为 1，那么阻塞等待将没有超时限制。

返回值

有两个可能的返回值:

#### 1. `pdPASS`

□

返回 `pdPASS` 只会有一种情况，那就是数据被成功发送到队列中。

如果设定了阻塞超时时间(`xTicksToWait` 非 0)，在函数返回之前任务将被转移到阻塞态以等待队列空间有效—在超时到来前能够将数据成功写入到队列，函数则会返回 `pdPASS`。

#### 2. `errQUEUE_FULL`

如果由于队列已满而无法将数据写入，则将返回 `errQUEUE_FULL`。

如果设定了阻塞超时时间 (`xTicksToWait` 非 0)，在函数返回之前任务将被转移到阻塞态以等待队列空间有效。但直到超时也没有其它任务或是中断服务例程读取队列而腾出空间，函数则会返回 `errQUEUE_FULL`。

- `xQueueSend()`和发送到队列尾部的函数一样，这两者调用哪一个函数都可以。
- 注意上面函数的第二个参数`pvItemToQueue`，它是发送数据的指针，记得在传参数的时候需要传入数据的地址。
- 3.从队列中读取数据，调用`xQueueReceive()`与 `xQueuePeek()` API 函数
  - 上面两个读队列函数，我们常用第一种`xQueueReceive()`，它用于从队列中接收(读取数据单元。接收到的单元同时会从队列中删除。(效果上类似于电脑的剪切，但是注意，是对数据拷贝完之后再删除的，而不是直接从队列中拿过来的)
  - `xQueuePeek()`也是从队列中接收数据单元，不同的是并不从队列中删出接收到的单元。(效果上类似于电脑中的复制操作) `xQueuePeek()`从队列首接收到数据后，不会修改队列中的数据，也不会改变数据在队列中的存储顺序。
  - 和上面向队列中写数据一样，不可以再中断服务程序中调用这两个API函数，需要调用加 `FromISR`后缀的函数

表 9 xQueueReceive()与 xQueuePeek()函数参数与返回值

| 参数名          | 描述                                                                                                                                                                                                                                                                                                                                                                 |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| xQueue       | 被读队列的句柄。这个句柄即是调用 xQueueCreate()创建该队列时的返回值。                                                                                                                                                                                                                                                                                                                         |
| pvBuffer     | 接收缓存指针。其指向一段内存区域，用于接收从队列中拷贝来的数据。<br><br>数据单元的长度在创建队列时就已经被设定，所以该指针指向的内存区域大小应当足够保存一个数据单元。                                                                                                                                                                                                                                                                            |
| xTicksToWait | 阻塞超时时间。如果在接收时队列为空，则这个时间是任务处于阻塞状态以等待队列数据有效的最长等待时间。<br><br>如果 xTicksToWait 设为 0，并且队列为空，则 xQueueReceive()与 xQueuePeek()均会立即返回。<br><br>阻塞时间是以系统心跳周期为单位的，所以绝对时间取决于系统心跳频率。常量 portTICK_RATE_MS 可以用来把心跳时间单位转换为毫秒时间单位。<br><br>如果把 xTicksToWait 设置为 portMAX_DELAY，并且在 FreeRTOSConfig.h 中设定 INCLUDE_vTaskSuspend 为 1，那么阻塞等待将没有超时限制。                                           |
| 返回值          | 有两个可能的返回值：<br><br>1. pdPASS<br><br>只有一种情况会返回 pdPASS，那就是成功地从队列中读到数据。<br><br>如果设定了阻塞超时时间(xTicksToWait 非 0)，在函数返回之前任务将被转移到阻塞态以等待队列数据有效——在超时到来前能够从队列中成功读取数据，函数则会返回 pdPASS。<br><br>2. errQUEUE_FULL<br><br>如果在读取时由于队列已空而没有读到任何数据，则将返回 errQUEUE_FULL。<br><br>如果设定了阻塞超时时间（xTicksToWait 非 0），在函数返回之前任务将被转移到阻塞态以等待队列数据有效。但直到超时也没有其它任务或是中断服务例程往队列中写入数据，函数则会返回 errQUEUE_FULL。 |

- 这里我们也需要关注第二个参数pvBuffer，它是接收数据的缓存指针，在传参的时候也需要传入接收变量的地址

- 要查询队列中环形缓冲区中有效单元的个数，我们需要调用 `uxQueueMessagesWaiting()` API 函数

```
unsigned portBASE_TYPE uxQueueMessagesWaiting(xQueueHandle xQueue);
```

程序清单 33 `uxQueueMessagesWaiting()` API 函数原型

表 10 `uxQueueMessagesWaiting()` 函数参数及返回值

| 参数名                 | 描述                                                        |
|---------------------|-----------------------------------------------------------|
| <code>xQueue</code> | 被查询队列的句柄。这个句柄即是调用 <code>xQueueCreate()</code> 创建该队列时的返回值。 |
| 返回值                 | 当前队列中保存的数据单元个数。返回 0 表明队列为空。                               |

- 只需要传入队列句柄，就可以查询到队列中数据的有效个数
- 到这里队列常用的API函数我们就都了解了，我们上面都是队列用于接收小数据单元，如果数据工作于大数据单元会怎样呢
- 3.3队列工作于大数据单元
  - 如果队列存储的数据单元尺寸较大，那**最好是利用队列来传递数据的指针**，而不是对数据本身在队列上一个个拷贝进或者拷贝出。传递指针无论是在**处理速度上还是在内存空间利用上都更有效**。但是利用队列传递指针需要注意以下几点
    - 1. **指针指向的内存空间所有权必须明确**
      - ◆ 原则上，共享内存在其指针发送到队列之前，只允许发送任务访问。共享内存指针从队列中被读出后，只允许接收任务访问
    - 2. **指针指向的内存空间必须有效**
      - ◆ 如果指针指向的内存空间是动态分配的，只应该有一个任务负责对其内存进行释放。当这段内存被释放之后，就不应该有任何一个任务再访问这段空间。切忌用指针访问任务栈上分配的空间。因为当栈帧改变，栈上的数据将不在有效。