Project Group 'Racing Car IT'

caRTOS - AVR Operating System

Concepts and usage

Peer Adelt
Benjamin Koch

University of Paderborn
Summer Semester 2012

Advisers:
Prof. Dr. Marco Platzner, Tobias Beisel, Sebastian Meisner, Lars Schäfers

Contents

1 Introduction

This document introduces the fundamental concepts of the *caRTOS* (car Real Time Operating System). This operating system was developed to allow a better predictabiliy of task runtimes. If all program parts run in a single while-loop, runtime estimation gets quite difficult. Furthermore, the *response times* of the different program parts need to be recalculated, if the while-loop changes.

caRTOS is a static operating system. Resource usage as well as the set of tasks is known a priori. This also means, that no new tasks or resources can be added at runtime. The next sections will explain the task switching mechanism and the resource management provided by caRTOS.

2 Process management

Application specific code runs in *Tasks*. Each task has an associated *Task State*. Defined states are: *Suspended*, *Ready*, *Waiting* and *Running*.

Figure 1 shows all four task states and transitions between these states. They are discussed in detail in the next two sections. For API descriptions, see section 4.

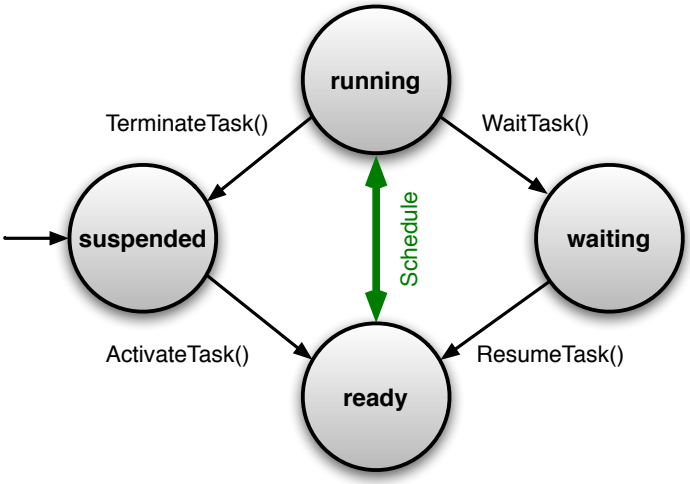

Figure 1: Task states and transitions

2.1 Task States

If a task was not yet run or has already finished its work, it resides in the *Suspended* state. Suspended tasks are not considered at the time of schedule.

All *Ready* tasks are considered at the point of schedule. It depends on the *scheduling policy*, which of the Ready tasks is assigned to the CPU.

A task may wait for a shared resource. If a resource is not immediately available, the requesting task can put itself into *Waiting* state. Waiting tasks are also not considered for scheduling.

The *Running* task is currently executed on the AT90CAN microprocessor. Since it is a single core processor, exactly one task may be in *Running* state at any point in time.

2.2 Task State Transitions

A suspended task can be made Ready with the `ActivateTask(id)` function, where `id` is the identification number of the task being activated.

According to the *Scheduling Policy* the operating system scheduler decides, which of the Ready tasks to run next. For now, there are no functions for manually transferring Running tasks to Ready state and vice versa.

Each task must call `TerminateTask()` after completion to reset the the task's stack space.

If a task wants to access a shared resource which is occupied, it is required to call `WaitTask()`. The operating system will then switch the current task to state *Waiting* and choose another task for execution. The task will stay in this state, until it is unblocked by `ResumeTask(id)`, where `id` is the identification number of the Waiting task.

2.3 Task Control Blocks

The operating system manages tasks in *Task Control Blocks (TCBs)*. A TCB stores the task's unique identification number, current state, a pointer to its stack top address and its the stack pointer. Furthermore, it stores a pointer to the task function. The relationship between TCB, stack and task program is shown in Figure 2.

2.4 Scheduling

This section shows the the scenarios, in which scheduling may occur. Furthermore, the scheduling algorithm is described.

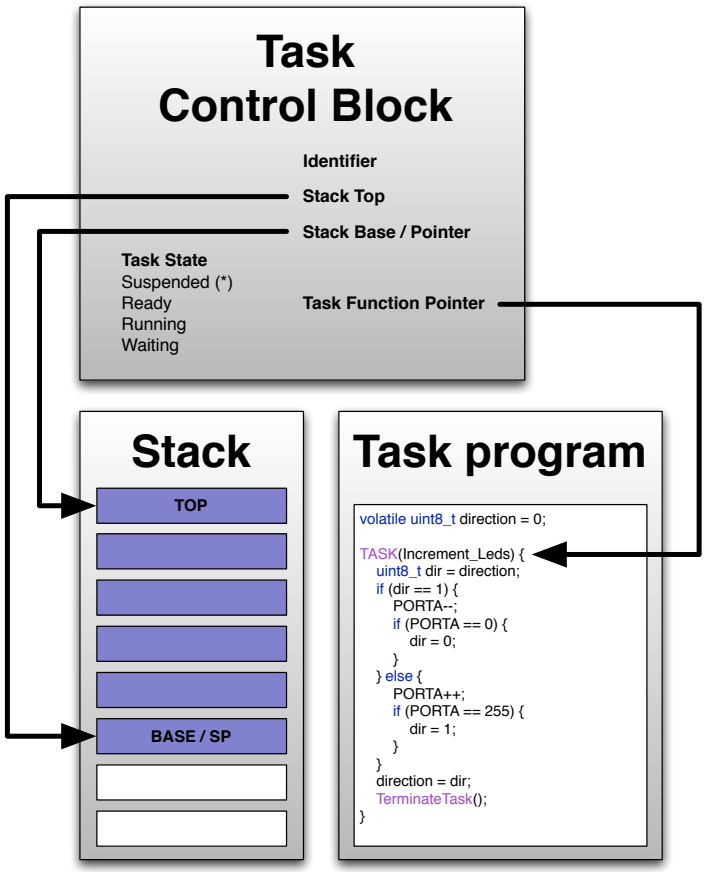

Figure 2: Task structure

2.4.1 Points of schedule

Like in most operating systems, scheduling takes place at a regular time interval. caRTOS requires one of the four timers of the AT90CAN microprocessor for this purpose.

If a task gets blocked (i.e. is switched to state Waiting), scheduling will be performed instantaneously. The API function `WaitTask()` handles this internally.

2.4.2 Algorithm

Tasks are scheduled by the *round robin algorithm*. On each system timer interrupt, the current task is preempted and the next ready task is executed. The round robin approach was chosen in order to prevent *starvation*, as it might occur in priority based scheduling policies.

Algorithm 1 describes the behavior of the operating system scheduler. The variable `os_currentTcb` points to the currently active TCB.

```
1:  function Os_Schedule
2:      if os_currentTcb.state = RUNNING then
3:          os_currentTcb.state ← READY                    ▷ Switch task READY
4:      end if
5:      os_nextId ← wrap_decrement(os_currentTcb.Id)
6:      for i = 0 → OS_NUMBER_OF_TCBS do
7:          if os_Tcbs[os_nextId].state = READY then
8:              os_currentTcb = os_Tcbs[os_nextId]         ▷ Found READY task
9:              break
10:         else
11:             if i = OS_NUMBER_OF_TCBS − 1 then
12:                                  ▷ No READY task found: switch to IDLE
13:                 os_currentTcb ← IDLE_TASK_TCB
14:             end if
15:             os_nextId ← wrap_decrement(os_nextId)      ▷ Continue search
16:         end if
17:         i ← i + 1
18:     end for
19:     os_currentTcb.state ← RUNNING                      ▷ Execute next task
20: end function
```

Algorithm 1: Scheduling algorithm (Pseudocode)

First, the current task's state is switched to ready, if it was not terminated or is blocked before (lines 2-4). The local variable `os_nextId` is initialized with the index of the preceding TCB (line 5). The function `wrap_decrement` to determine this index is described in algorithm 2. `wrap_decrement` skips the Idle task (id 0) automatically.

The following *for* loop (lines 6-18) searches for the next ready task. The search takes up to `OS_NUMBER_OF_TCBS - 1` steps. If the TCB with index `os_nextId` is ready, the search is complete (lines 8-9). If the task is not ready, `os_nextId` is decremented with `wrap_decrement(os_nextId)` (line 15). If no ready task can be found, the Idle task is executed (line 13). The last statement (line 19) proceeds with the execution of the chosen task.

3 Resource management

This section describes the resource management of caRTOS. We support semaphores for task synchronization and queues for inter task communication.

```
1: function WRAP_DECREMENT(i)
2:     if i = 1 then
3:         return OS_NUMBER_OF_TCBS − 1                    ▷ wrap i
4:     else
5:         i ← i − 1                                       ▷ decrement i
6:         return i
7:     end if
8: end function
```

Algorithm 2: Wrap and decrement (Pseudocode)

3.1 Semaphore extensions

A semaphore consists of a counter with the amount of free resources and a waiting queue which contains the waiting tasks. A task may use `sem_wait` to get access to the semaphore (e.g. before a critical section). If its counter is non-negative, access is granted immediately. Otherwise, the task will be put into the waiting queue. In both cases, the counter is decremented. If a task releases a resource guarded by the semaphore, it should call `sem_signal` to increment the counter and activate a waiting task (if there is one).

Semaphores are a well-known technique of process synchronization. They can be used to provide mutual exclusion (critical sections), but they can also handle more complex scenarios like producer-consumer. Details are elaborated in [TW06].

In our implementation, memory for the semaphores is assigned at compile-time. This means that the maximum queue length must be known in advance, but we save the overhead of dynamic memory management by avoiding queues with dynamic length.

We have implemented two extensions of the standard semaphore interface. We describe those extensions in subsection 3.1. In the subsequent sections, we present the API and some details of our implementation of extended and standard semaphores.

We propose two extensions of the standard semaphore interface:

'many' semaphores The 'many' kind of semaphores supports operations that act like several calls to `sem_wait` or `sem_signal`, but it is more efficient and reduces the potential for deadlocks.

asynchronous semaphore access The `sem_wait` call may block the calling process. We provide an asynchronous counterpart that can be used to implement micro-threading.

The extensions will be outlined in more detail in the following paragraphs. In the last paragraph of this section, we show that we need an additional counter for each semaphore, if we implement the extensions.

3.1.1 'many' semaphores

Semaphores with the 'many' extension can reserve and release several bits of the guarded resource with only one call to the semaphore API functions.

Motivation Semaphores are often used in a producer-consumer scenario. The 'many' extensions are especially useful in that context. Producer-consumer requires three semaphores:

- ... a reader semaphore that grants access to bits of data in the shared storage,
- ... a writer semaphore that grants access to free space in that storage
- ... and a binary semaphore to protect shared data structures (critical section).

Readers and writers will block at the reader / writer semaphore, if until there is some data / free space. Only if this semaphore lets them pass, they enter the critical section for a short time to move data from / to the buffer.

In our system, we use a queue with producer-consumer semaphores to synchronize access to the serial line. Several readers can write to a shared queue (the writers) and a driver process (the reader) will transfer the data to the hardware one byte at a time. Processes usually write one line of text. For this example, we assume two writer processes: Process A writes 20 bytes each time and process B writes 30 bytes. The shared queue can hold up to 40 bytes of data.

There are two obvious ways of implementing `send_line` for those processes:

1. Repeat for each byte that we want to send: Wait until we have free space for one byte (1 call to `sem_wait`) and put one byte into the buffer.
2. Wait until we have free space for all the bytes (n calls to `sem_wait`) and put all of them into the buffer.

Both implementations have a problem. With the first implementation, the lines will be intermixed in the output. This means that the output is incomprehensible in most cases. The second implementation can get stuck in a deadlock: If process A passes 15 calls to `sem_wait` and process B passes 25, all of the free space (40 bytes) is reserved by them, but neither process can continue.

Extension We propose the 'many' extension: We add the parameter `amount` to `sem_wait` and `sem_signal`. `sem_wait` records the amount of resources that the process needs to continue. We store this information for each process in the waiting queue. `sem_signal` increments the semaphore counter by `amount` and unblocks one or more processes. Only processes at the head of the waiting queue may be unblocked.

We compare the extended semaphores to semaphores without this extension:

processor time Normal semaphores need one call to both semaphore functions for each byte. Each blocking call to `sem_wait` causes two context switches. The extended semaphores need only two calls and two context switches regardless of the number of bytes.

memory The extended semaphores need an additional field for each entry of the waiting queue. This roughly doubles the memory usage for the queue.

deadlock potential The available resources are pooled up, until the first process can run. Processes get all of there reservation or none at all. Therefore, the case of deadlock that is outlined above cannot occur.[1]

3.1.2 Asynchronous access

Motivation The wait operation of a semaphore is inherently a blocking (synchronous) operation. Semaphores don't need polling because they block the process and unblock it at the right time. However, asynchronous operations are necessary for micro-threads that cooperate with code that uses synchronous semaphores.

Threads or tasks have a lot of overhead because the operating system has to store their stack and during a context switch the complete processor context must be saved. In some cases, the running program has some knowledge about the running threads that can be used to speed up the switch. For example, a

[1] Similar deadlock prevention can be achieved with standard semaphores, but you need two more semaphores.

stack based virtual machine only needs to replace the instruction pointer and stack pointer when switching to another thread. This is called micro-threading and usually it yields a better performance than full-blown threads or tasks.

To the operating system, a set of micro-threads looks like one thread. It is paramount that the micro-threads don't use blocking operations because they would block the OS thread including all the micro-threads. Therefore, micro-threaded architectures use only asynchronous operations.[2]

We generate code for statemachines which has useful property: Its main function runs for a short time and after it returns it doesn't need any state except some global data. This makes statemachines an ideal candidate for cooperative micro-threading. The statemachines need semaphores to communicate with each other and with other processes, so we need asynchronous counterparts for blocking semaphore operations.

Requirements The asynchronous statemachine operations should have a few properties:

compatible with synchronous access We will use asynchronous and synchronous operations on the same semaphore and at the same time.

fair Asynchronous operations must be recorded in the waiting queue, so they get access before a process that is asking at a later time.

compatible with 'many' semaphores We will use them with shared queues, so we want to get the benefits of the 'many' extension.

trust the caller We assume that a waiting micro-thread regularly checks the status of its token(s).

Implementation A trivial (and incorrect) implementation would provide the `sem_is_free` functions which returns true, iff the semaphore counter is positive. The caller would then use `sem_wait` to gain access to the semaphore. This implementation has at least two problems: First, it is prone to a race condition. The call to `sem_wait` might block, if another thread calls `sem_wait` between the two calls. Second, it lacks fairness. `sem_signal` wakes up waiting processes while the asynchronous clients might never gets a chance, if the processes keep filling the waiting queue.

[2]Some micro-threading implementations rewrite synchronous operations under the hood.

To achieve fairness, the asynchronous clients must be represented in the waiting queue. Usually, the task id is used to identify objects in the waiting queue. We cannot use the task id because micro-threads might live in the same task, but we still have to tell them apart in the queue. Therefore, we give a unique token to each micro-thread that starts waiting at the semaphore. This token is put into the queue. The caller must save this token and provide it when asking for the status of its asynchronous operation.

Tokens use values from the task id space which are not assigned to a task. They must be larger than the maximum task id.[3] As we know this value, we can easily distinguish task ids from asynchronous tokens.

The token has to be removed from the queue at some point. For synchronous access, `sem_signal` wakes up the task and it removes the task id which is no longer waiting for the semaphore. For an asynchronous token, this is not so easy. It must be kept until the micro-thread asks for its status. Actually, we wait for the micro-thread to acknowledge that it will use the resource. That way, the caller can pass the token to a library function which checks the state again as part of its pre-conditions.

Asynchronous clients might opt out early which a blocked synchronous task cannot do. For example, we may want to drop a debug message to the serial line, if we cannot post it within half a second. In this case, we must undo the effects that the asynchronous waiting has had on the semaphore: We remove the token from the queue and increment the semaphore counter.

3.1.3 The ready counter

With ordinary semaphores, the implementation of `sem_signal` is straightforward because each call can wake up exactly one process (if a process is waiting). Both extensions might case `sem_signal` to not wake up a process, although some processes are waiting for the semaphore. For 'many' semaphores this will happen, if the amount of released resources is smaller than the amount the first waiting process needs. `sem_signal` also cannot wake up an asynchronous token and it mustn't give the resource to another process, if the token is at the head of the waiting queue.

If `sem_signal` cannot wake up some process, it must remember the amount of free resources that it hasn't used to wake up processes. We have added an additional

[3] This must hold true at all time which can be difficult, if new tasks can be created dynamically. This is not possible for our system.

counter to each semaphore which will be incremented by that amount.[4] It contains the amount of resource units that are ready to be given to a process, so we name it 'ready counter'. In contrast to the semaphore counter, it may have a positive value, although the waiting queue is not empty.[5]

`sem_signal(semaphore, amount)` can use `amount` + `ready_count` resource units to unblock waiting processes. It looks at the first processes and tokens in the waiting queue, which together don't need more resource units than it can provide[6]. All processes in this set are unblocked and removed from the queue; the resource units for tokens and the remaining resource units are stored in `ready_count`.

`ready_count` is also used to determine whether a token is ready: A token is ready, iff all processes and tokens up to and including the token itself need at most `ready_count` resource units.

3.2 Queues

caRTOS provides *Queue* data structures for *Inter Process Communication (IPC)*. Queue accesses are internally handled by semaphores (see preceeding section). The *Producer Consumer Paradigm* ensures, that no two queue accesses can collide. Enqueue and dequeue operations can be done synchronously and asynchronously. Furthermore, these operations can be intermixed.

3.2.1 Synchronous Access

To enable safe inter-process communication via queues, 'many' semaphores were used. Following the producer-consumer concept, an enqueue operation of n elements will firstly aquire n producer semaphores. To avoid collisions, a mutex must be requested afterwards. After the actual enqueue operation, the mutex is released and n consumer semaphores are released (allowing consumers to read n more items from the queue).

The dequeue operation works the other way around: before the actual operation, n consumer semaphores are aquired. After the mutex was aquired, the operation was finished and the mutex was freed again, n producer semaphores are freed

[4] For 'many' semaphores, we could store this information in the waiting queue, but using the 'ready counter' is more robust.

[5] We believe that we can remove the usual semaphore counter, if we have the 'ready counter' and the waiting queue can tell whether it is empty. This requires extensive changes to the standard semaphore logic, so we sacrifice a bit of memory in favor of more robust semaphores.

[6] $\sum_{i=1}^{n}$ requested-amount(waiting-process(i)) $\leq$ amount + ready_count

(allowing n new elements to be placed into the queue). Please see section 4 for details.

3.2.2 Asynchronous Access

When blocking needs to be avoided, queue accesses can be requested asynchronously. The operation needs to initiated by requesting a token (see subsubsection 3.1.2). Since no task blocking occurs, the task is responsible for checking the token on a regular basis. As soon as the token becomed valid, the operation can be finished. Please see section 4 for details.

4 API specification

This section describes the *Application Programming Interface (API)* provided by the caRTOS operating system.

4.1 Process management

The implementations of these functions are located in files *os/internal/semaphore/Os_ Task.c* and *os/internal/Os_ Task.h* of the OS distribution.

Name:	void **ActivateTask**(TaskType id)
Description:	Switches the task with identification number id to state READY.
Parameters:	TaskType id: Identification number of task to activate
Return value:	none
Remarks:	This function only activates tasks, which are in state SUSPENDED. RUNNING, WAITING and READY tasks are ignored. (Note: use ResumeTask() to unblock a WAITING task.)

Name:	void **TerminateTask**(void)
Description:	Terminates the current task and switches its state to SUSPENDED.
Parameters:	none
Return value:	none
Remarks:	none

Name:	void **WaitTask**(void)
Description:	Blocks the current task and switches its state to WAITING.
Parameters:	none
Return value:	none
Remarks:	none

Name:	void **ResumeTask**(TaskType id)
Description:	Unblocks the task with identification number id and switches its state to READY.
Parameters:	TaskType id: Identification number of task to unblock
Return value:	none
Remarks:	This function only unblocks tasks, which are in state WAITING. If this function is called for a task with a different state, this is considered an error.

4.2 Resource management

The implementations of these functions are located in *os/semaphore/* and *os/IPC/* subdirectories of the OS distribution.

4.2.1 Basic semaphores

Name:	**SEMAPHORE**(s, i, c)
Description:	This preprocessor macro defines all necessary datastructures for the basic semaphore s. It is initialized with i and can take up to c entries.
Parameters:	s: Name of the semaphore i: Initial value for semaphore c: Capacity of semaphore queue
Return value:	none
Remarks:	This macro is intended to be used in global scope of a C file.

Name:	void **sem_wait**(s)
Description:	This macro synchronously aquires the semaphore s. If the ressource is not available, the calling task is autmatically blocked.
Parameters:	s: Name of the semaphore
Return value:	none
Remarks:	none

Name:	void **sem_signal**(s)
Description:	This macro releases the semaphore s. If there is another task waiting for the semaphore, it gets unblocked. If the next entry in the queue is a token, the token will become valid.
Parameters:	s: Name of the semaphore
Return value:	none
Remarks:	none

Name:	sem_token_t **sem_start_wait**(s)
Description:	It returns a token that you can use with `sem_continue_wait` (see below). You must eventually finish or abort waiting.
Parameters:	s: Name of the semaphore
Return value:	A new token for this semaphore.
Remarks:	none

Name:	BOOL **sem_continue_wait**(s, t)
Description:	Query the state of an asynchronous waiting token. If the semaphore has been granted to the owner of the token, a non-zero value is returned. If false (or zero) is returned, you must try again later (or abort waiting).
Parameters:	s: Name of the semaphore t: Token for this semaphore
Return value:	True (not 0), if semaphore granted. False (0), if semaphore not granted.
Remarks:	False (0) is also returned on any kind of error.

Name:	void **sem_finish_wait**(s, t)
Description:	Signalize, that the task waiting for the ressource will actually use it later.
Parameters:	s: Name of the semaphore t: Token for this semaphore
Return value:	none
Remarks:	The function `sem_continue_wait` has to return true for this token. This function must be called, when the ressource will be used as intended. The token may not be used again after calling this function.

Name:	void **sem_abort_wait**(s, t)
Description:	Abort waiting on a semaphore when the associated ressource is not needed anymore.
Parameters:	s: Name of the semaphore t: Token for this semaphore
Return value:	none
Remarks:	This function must be called, when the ressource will not be used. The token may not be used again after calling this function. You mustn't use the associated resource or call `sem_signal`.

4.2.2 'many' semaphores

Name:	**SEMAPHORE_N**(s, i, c)
Description:	This preprocessor macro defines all necessary datastructures for the 'many' semaphore s. It is initialized with i and can take up to c entries.
Parameters:	s: Name of the semaphore i: Initial value for semaphore c: Capacity of semaphore queue
Return value:	none
Remarks:	This macro is intended to be used in global scope of a C file.

All functions for basic semaphores are available for 'many' semaphores as well. The functions are named analogously with `_n` appended to their identifiers. The only difference in usage is, that they additionally expect the amount of ressource being requested and work (only) on semaphores declared with the `SEMAPHORE_N` macro. You cannot use the basic semaphore functions (without `_n` suffix) with 'many' semaphores!

4.3 Queues

Each queue is defined with a preprocessor macro which ensures, that mutex, producer and consumer semaphores are created together with the queue as well.

Name:	**QUEUE**(q, c, rc, wc)
Description:	This preprocessor macro defines all necessary datastructures for queues. Mutex, producer and consumer semaphores are instantiated as well.
Parameters:	q: Name of the queue c: Capacity of queue rc: Number of readers (size of consumer semaphore queue) wc: Number of writers (size of producer semaphore queue)
Return value:	none
Remarks:	This macro is intended to be used in global scope of a C file.

Name:	void **queue_enqueue**(q, uint8_t d)
Description:	Puts a byte into the queue. If the queue is not accessible at the moment of the function call, the current task will get blocked before the queue access.
Parameters:	q: Name of the queue d: New data to be put into the queue
Return value:	none
Remarks:	none

Name:	void **queue_enqueue_many**(q, uint8_t c, const uint8_t * d)
Description:	Puts a sequence of bytes into the queue. If the queue is not accessible at the moment of the function call, the current task will get blocked before the queue access.
Parameters:	q: Name of the queue c: Number of bytes to store in the queue d: Pointer to array of data to be inserted
Return value:	none
Remarks:	The queue must be long enough to hold all the bytes at the same times. Otherwise, the call will block indefinetly and by that it will also block other writers.

Name:	uint8_t **queue_dequeue**(q)
Description:	Reads the first byte from the queue. If the queue is empty at the moment of the function call, the current task will get blocked until at least one byte is available. The first byte is returned and removed from the queue.
Parameters:	q: Name of the queue
Return value:	Byte that was read from the queue
Remarks:	none

Name:	void **queue_dequeue_many**(q, uint8_t c, const uint8_t * d)
Description:	Reads a sequence of bytes from the queue. Like `queue_dequeue` it will block the current task until enough bytes are available and it will remove the read bytes from the queue.
Parameters:	q: Name of the queue c: Number of bytes to read from the queue d: Data read from the queue
Return value:	none
Remarks:	The function reads consecutive bytes from the queue. If you request more bytes than the queue can hold, it will block forever and by that block all readers on that queue.

You can also use the queues in an asynchronous fashion. You can use the functions `queue_start_enqueue`, `queue_continue_enqueue`, `queue_finish_enqueue` and `queue_abort_enqueue` (similar for dequeue). They work like the asynchronous semaphore functions. The `finish` function has an argument for the number of bytes, although you have already passed that value to the `start` function. We don't want you to pass a

pointer to data and the data size at different times because we think that would lead to errors. Therefore, you have to pass the same value twice. However, we have taken the opportunity to make the system a bit more flexible. Actually, you can pass a smaller amount to the `finish` function and it will only use part of your reservation. The remaining bytes (of free space or available data) will be released, so they will be available for other readers/writers.

References

[TW06] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice Hall, January 2006.