

portos

The RTOS Revolution

User and Reference Guide

Softwave Wireless

<http://www.portos.org>

© 2003-2009 by Softwave Wireless

All rights reserved. No part of this manual may be reproduced in any form or by any means without prior agreement and written consent from Softwave Wireless.

Trademarks

Portos is a registered trademark of Softwave Wireless.

All other product and company names are trademarks or registered trademarks of their respective holders.

Contents

1	INTRODUCTION	4
2	PRIORITY FUNCTIONS	6
3	TIMERS	10
4	SIGNALS	13
5	QUEUES	15
6	DYNAMIC MEMORY MANAGEMENT	17
7	MISCELLANEOUS	19
8	TASKS VS. PRIORITY FUNCTIONS	24
9	BENCHMARK	28
10	REFERENCE	30

1 Introduction

Portos is a new paradigm for real-time embedded programming whose goal is to make things simpler, smaller and faster. When programming with Portos, you don't need tasks, threads, messages, mailboxes, semaphores, events, etc. You focus on your main problem rather than on the details of the RTOS. Portos is so simple that it feels like the "one-click shopping" of Real Time Operating Systems.

At the heart of Portos is the priority function that allows simple, efficient and modular programming. If time to market is important to you, you may want to consider Portos. If code reuse and upgrade is important to you, take a look at Portos. Priority functions are like plain functions and can be called from anywhere. They are independent of the program's architecture, and hence modularity is preserved and code upgrade is effortless. Contrast this with traditional tasks or threads where there are numerous kinds of interdependencies, including inter-task communications. In a traditional RTOS, you cannot call a function directly from a different task: you have to send a message. Modularity is lost and code upgrade is difficult.

In addition, priority functions calls in Portos are automatically optimized by the compiler and preprocessor, and function arguments are automatically handled (when temporary storage is needed).

Along with priority functions, Portos provides especially simple to use *timers* and *signals*, as well as highly efficient and flexible *dynamic memory allocation* routines.

In the current version of Portos, all priority functions run in a single *virtual task*, with a dedicated stack. In future versions of Portos, it will be possible to group priority functions into distinct virtual tasks that we call *supertasks* or *superthreads*. They are more powerful than traditional tasks; we will briefly mention them in this document.

Portos can be used either as a stand-alone RTOS or as an add-on library that complements an existing RTOS. The advantages of using Portos as a stand-alone RTOS are mainly smaller memory size and higher execution speed. The advantages of using Portos together with a second RTOS are:

- the programmer has access to the features of both RTOS,
- Portos is quasi-automatically ported to any DSP or CPU platform that is supported by the second RTOS, and
- existing projects using the second RTOS can be upgraded with Portos functions.

Today Portos is only available as an add-on library for the Texas Instruments BIOS (TI BIOS) and runs on all DSPs supported by the TI BIOS. After reading this document, you can refer to the document *Portos Interface to TI BIOS* for more information on topics specific to the TI BIOS. Note that if you have an existing project that uses TI BIOS tasks, you can simply start adding Portos instructions to that same project. Portos is fully compatible with the TI BIOS.

For the time being, the Portos preprocessor supports the C programming language and the Windows Operating System only. Linux will be supported later.

2 Priority Functions

A priority function is a C function that has been assigned a priority level. When the priority function is called, it will either

- execute right away, if it has higher priority than the current priority level, or
- wait until all higher priority functions are done.

In the second case, the scheduler automatically stores the priority function in a database, along with its arguments. It will wait there until the more urgent functions are done.

An example priority function named `myfunc`, with priority level 5, is declared as follows,

```
void myfunc(po_priority(5), int a, double b, char *c)
{
    ...
}
```

The *first argument* of a priority function must be the directive `po_priority(P)`. The prefix `po_` denotes a Portos directive or function. The Portos preprocessor recognizes the directive `po_priority` and automatically transforms `myfunc` into a priority function with priority level `P`, where `P` can be any C expression (e.g. macro or variable).

The priority function can be called from anywhere as follows,

```
myfunc(po_priority, x, y, "hello");
```

When calling the priority function, we do not supply a priority level. The priority level is only supplied in the function declaration and function prototype.

2.1 Dynamic Priority Level

The priority level can be dynamic, i.e., a variable rather than a constant. For example,

```
void myfunc(po_priority(a), int a, double b, char *c)
{
    ...
}
```

In the above declaration, the second argument, `a`, specifies the priority level in a dynamic fashion. If we call the function twice as in

```
myfunc(po_priority, 7, y, "hello");
myfunc(po_priority, 8, y, "world");
```

then the priority function will be called a first time at priority level 7 and a second time at priority level 8. In this example, the second parameter determines the priority level.

But the priority level is not limited to a constant or a variable. It can be any C expression. Some examples,

```
void myfunc(po_priority(2*b + some_global),
           int a, double b, char *c)
{
    ...
}
```

or

```
void send_to_printer(po_priority(job->priority),
                   Printer *job, char *data)
{
    ...
}
```

In the last example, a printer job structure contains the priority level at which the data is sent to the printer. We can easily instantiate different printer jobs at different priority levels. We can denote the printer job structure by *priority object*, and future versions of Portos will automatically handle the C++ language.

Important note: do not declare too many functions as priority functions. There is an overhead incurred by each priority function. For each module, or each group of modules, you should clearly specify the entry functions into the module or the group. These functions will be declared as priority functions. They are the functions that change the priority level (as if entering a new task). Other functions internal to the modules are called as normal functions.

The current version of the Portos preprocessor optimizes for speed. A next version will provide the ability to optimize for either speed or code size.

2.2 Priority Functions Cannot Return a Value

A priority function cannot return a value, and hence the return type must be *void*. It cannot return a value since there is no guarantee it will be immediately executed. No recipient can take the return value when the priority function is delayed rather than executed immediately.

The priority function cannot be an *inline* function but it can be *static*, i.e., local to the C file.

2.3 Shared Data or Resources

In real-time programming, due to the presence of priority levels, a high priority function can preempt a low priority function, and then it can attempt to access shared data or

resources that the low priority function was in the midst of modifying. The result is a corruption of the data or resource.

Traditional RTOSes deal with this issue by using *semaphores* to lock access to the shared data, but this technique is not usually recommended because of various drawbacks.

A better technique in traditional RTOSes is to use the *priority ceiling* mechanism: before accessing the shared data, the priority level is raised to a high value such that no other task can preempt the current task. The current task accesses the shared data, and when it is done, it restores the priority level to its original value. At that point it can be preempted but the shared data has been protected. Any access to the shared data must happen at the high priority level in order to prevent preemption during the access.

Portos offers the *priority ceiling* mechanism as well as a powerful new one, the *fixed priority* mechanism.

2.3.1 Priority Ceiling

The *priority ceiling*, like in traditional RTOSes, works by temporarily raising the priority level. It allows safe and immediate write or read of the data. The priority is raised and restored via the two functions

```
int po_priority_raise(int new_priority); //returns old priority
void po_priority_restore(int old_priority);
```

Function `po_priority_raise` returns the old priority level, which should be supplied to function `po_priority_restore` in order to restore that priority level.

2.3.2 Fixed Priority

The *fixed priority* mechanism consists of restricting the access to the data to a small set of dedicated priority functions (known as methods in C++). No other function is allowed direct access to the data, unless the type of access is harmless such as a simple read (but even a simple read can be tricky). All the dedicated priority functions or methods must run at a *fixed priority* level. Hence, the consistency of the data is always guaranteed.

For example, in order to increment a shared integer `counter`, we call a dedicated priority function `increment_counter()`. Unlike *priority ceiling*, in the *fixed priority* mechanism the dedicated priority function can be called from either lower or higher priority levels. Since it always executes at a fixed priority level, the various calls are automatically queued and executed sequentially (no preemption), in a first-in-first-out fashion.

The advantage of *fixed priority* over *priority ceiling* is that the priority level does not have to be high. It can be low. This advantage is significant if we wish to access the shared data from hardware interrupt. Hardware interrupts cannot use the *priority ceiling* mechanism because their priority level is already too high, above any task's priority

level. Hardware interrupts cannot use *semaphores* either since they are not allowed to block. Hence, in a traditional RTOS, the programmer has to work hard to create signals or messages that the hardware interrupt sends to request a modification of the shared data. With the *fixed priority* mechanism, the hardware interrupt can directly call the dedicated priority function. Portos takes care of the rest.

The disadvantage of *fixed priority* is that an immediate read instruction does not work. After submitting a request to read the data, the read instruction is queued (if the priority level is low), and one must wait for the read instruction to finish before obtaining the result. A signal or a synchronization queue can be used to notify when the read instruction terminates and the result is available (cf. sections 4 and 5).

The *fixed priority* technique may not be too suitable for simple data accesses such as incrementing an integer and reading its value. But it is very useful for parsing larger amounts of data, or for accessing the slow ports of peripheral devices, etc.

The *fixed priority* technique does not exist in traditional RTOSes because it is prohibitively expensive to implement. It means dedicating to the shared data an entire task with its own stack, mailbox for messaging, mailbox management routines, costly context switches, etc.

3 Timers

We can schedule a predefined priority function to be called at a given time in the future, for example, at time 123,

```
myfunc(po_time(123,0), x, y, "hello");
```

We simply replace the directive `po_priority`, which means call now, by the directive `po_time(T,C)`, which means call when the specified time `T` is reached. The time `T` is an integer number. The second argument `C` is the clock number. In the above, `myfunc` will be called when clock number 0 displays the time 123.

When time `T` is reached, `myfunc` is called at its predefined priority level `P` that can be a constant or a variable as explained in section 2.

3.1 Clocks

Several clocks can be defined for different purposes. In time-slotted communication systems, for example, clock number 0 can be incremented at the start of each frame (frame clock) while clock number 1 can be incremented at the start of each timeslot (slot clock). The timeslot clock can wrap to zero at the end of each frame.

The start of a frame or timeslot is often signaled by a hardware interrupt. Hence, one way to increment the clock is to call, from the hardware interrupt, the following function

```
po_time_tick(C);
```

In order to wrap the clock value back to zero, or to set it to any time value `T`, we call the function

```
po_time_set(T,C);
```

The current time is obtained via

```
T = po_time_get(C);
```

3.2 Canceling a Timer

The above scheduled call, `myfunc(po_time(123,0), x, y, "hello")`, cannot be canceled before time 123 is reached. The reason being that there is no way to tell Portos what needs to be canceled. In order to have the flexibility to cancel the timer hidden behind this call, we need to take an extra step of defining a handle to the timer. This is done as follows

```
po_time_Handle handle = po_time_INIT;
myfunc(po_time(123,0,&handle), x, y, "hello");
```

The `po_time` directive is overloaded by adding a third argument, `handle`. Portos fills out the structure `handle` with the necessary timer information so that later we can cancel this timer via the function

```
po_time_cancel(&handle);
```

Notice that we initialized `handle` to the value `po_time_INIT`. This is a macro that sets the initial content of `handle` to something that lets Portos detect errors such as trying to reuse a handle that is currently active. If `handle` is allocated in dynamic memory, then it can be initialized via the call

```
po_time_init(&handle);
```

We can tell if the handle is active, i.e., the timer is still running, via the function

```
int po_time_isactive(&handle);
```

This function returns non-zero if the handle is active.

3.3 Creating Clocks

Portos creates the default clock number 0 that you can modify if desired. And you can create more clocks if needed.

To create a clock, call

```
po_time_Clock *po_time_createclock(
    int clockIndex,      // clock number
    int clockPriority,   // clock priority
    int hashSize,       // hash table size
    int memRegion       // dynamic memory region
);
```

`clockIndex` is the clock number.

`clockPriority` is the level at which the clock dedicated priority functions will execute. In other words, the clock functions run at *fixed priority* level as discussed in section 2.3.2. This *fixed priority* level is normally above the priority level of all priority functions that will use this clock.

`hashSize` is the clock's database size. It indicates the number of entries in a hash table. For higher performance, use a large `hashSize`. But this means more memory space.

If `hashSize` is chosen to be a power of 2, then the clock can take any integer value. On the other hand, if `hashSize` is not a power of 2, then the clock can only take the values from 0 to `hashSize-1`. An error is reported if the clock is incremented beyond the allowed segment. The reason for this limitation is that the modulo operation used by the hash key is inefficient when `hashSize` is not a power of 2.

When the clock spans a large segment of integers, or the entire set of integers, choose a **hashSize** power of 2. On the other hand, if you know the clock will only take the values 0 to 9, for example, then you could choose a **hashSize** of 10.

memRegion is the dynamic memory region that will be automatically used by the timers of this clock (cf. section 6.1).

3.4 Portability

The clock time used by Portos is of integer type. On 32 bit machines, this value can range from 0 to 0xFFFFFFFF (and wraps around). On 16 bit machines, this value can range from 0 to 0xFFFF. For portability across different machines, you could limit the clock range to something between 0 and 0xFFFF. For example, each time the clock reaches the maximum value of 10000 it is manually reset to 0. In a future version, Portos may provide a 32 bit timer module for 16 bit machines.

4 Signals

Very similar to timers, signals can be used to schedule the execution of priority functions when certain events occur (in fact, timers are signals in Portos). For instance, a hardware interrupt can post the signal “keyboard button depressed” and several functions waiting for this event are automatically called.

We can attach a predefined priority function to signal 25, for example, as follows,

```
myfunc(po_signal(25,0), x, y, "hello");
```

We simply replace `po_priority` by `po_signal(S,G)`, which means don't call now but wait until the specified signal `s` is posted by someone. `s` is an integer. `G` is the signal group number. Different groups of signals can be defined if needed, although this is seldom useful (unlike the case for different clocks).

4.1 Posting a Signal

When signal 25 occurs, `myfunc` is called at its predefined priority level `P` that can be a constant or a variable as explained in section 2. How does signal 25 occur? Simply call the following function when the desired event occurs (often from a hardware interrupt),

```
po_signal_post(S,G);
```

4.2 Detaching a Priority Function from a Signal

The above scheduled call cannot be canceled before signal 25 occurs since we have no way to tell Portos what is to be canceled. In order to have the flexibility to detach a function from a signal before it is called, we need to take an extra step of defining a handle,

```
po_signal_Handle handle = po_signal_INIT;
myfunc(po_signal(25,0,&handle), x, y, "hello");
```

The `po_signal` directive is overloaded by adding a third argument, `handle`. Now Portos fills out `handle` with the necessary information so that later we can detach the handle via the function

```
po_signal_detach(&handle);
```

Notice that we initialized `handle` to the value `po_signal_INIT`. This is a macro that sets the initial content of `handle` to something that lets Portos detect errors such as trying to reuse a handle that is currently active. If `handle` is allocated in dynamic memory, then it can be initialized via the call

```
po_signal_init(&handle);
```

We can tell if the handle is active, i.e., still attached, via the function

```
int po_signal_isactive(&handle);
```

This function returns non-zero if the handle is active.

4.3 Using Signals

One important use of signals is to maintain various libraries independent and therefore cleaner and reusable. The library, whether you purchased it from a third party or whether you developed it yourself, does not need to know about new functions that you write. In other words, the library does not directly call your new functions. Rather, it posts a signal to which you can attach your new priority functions. Then the priority functions are automatically called, at the corresponding priority level and with the supplied arguments. The signal is represented by a unique integer that everyone agrees to set aside and use for a particular purpose.

5 Queues

In order to help in synchronizing priority functions, Portos provides a *queue of priority functions*. It is similar to a queue of messages but it holds a function and its arguments. And it is automatically handled by the Portos preprocessor.

The purpose of the queue is to synchronize priority functions of same or different priority levels. *One function at a time executes in a first-in-first-out order, no matter what the priority level is.* When a function terminates, it releases a token (i.e. semaphore) that allows the next function in the queue to execute.

A priority function can be easily appended to a queue:

```
myfunc(po_queue(&Q), x, y, "hello");
```

If the queue is empty, the priority function is serviced right away. If the queue is not empty, the priority function has to wait for all priority functions ahead of it in the queue to terminate (or rather to release a token) even if they have lower priority levels.

When a priority function at the head of the queue is started, it is automatically assigned a token. When the priority function terminates, it releases the token. At that point, the next priority function in the queue is started. Note: the token can be released at any point in time, not necessarily when the priority function is about to terminate. As soon as the token is released, the queue is serviced and the currently running priority function could be preempted by a higher priority function.

The queuing behavior can be manually implemented by the programmer via signals and some additional logic but `po_queue` simplifies the job.

As an example, if we add the priority function `myfunc1` to the queue `Q`, and then we add the priority function `myfunc2` to `Q`, the second priority function will not execute until the first one releases the token, independently of priority levels. `myfunc1` could call many other low priority functions: they will execute before `myfunc2` as long as the token is not released.

Essentially, a queue of priority functions acts like a thread of functions. Even though certain priority functions can have a higher priority level, they must wait for other lower priority functions in the thread to terminate before they can start.

A queue is declared and initialized statically or dynamically via one of

```
po_queue_Queue Q = po_queue_INIT(&Q, numServers, memRegion);  
po_queue_init(&Q, numServers, memRegion);
```

`memRegion` is the dynamic memory region (section 6) used by this queue for internal memory allocation. `numServers` is the number of simultaneously available tokens. If this number is greater than 1, for example, if it is 3, then 3 priority functions can be active

simultaneously (the highest priority one runs first). Each priority function will be given one token and should release the token at some point. A token is released by calling

```
po_queue_next (&Q) ;
```

It is currently not possible to cancel a function that is waiting in the queue. But this option may be added in a future release of Portos.

5.1 More on Synchronizing Priority Functions

One drawback of priority functions is that they cannot suspend and wait for some event or some result. Unlike tasks or threads that own a dedicated stack, priority functions share a common stack and therefore they cannot suspend and wait. But this drawback is, in many cases, an advantage. It forces the programmer to write the code in a way to avoid context switches between tasks, and often to avoid deadlocks. Note: once supertasks or coroutines are implemented in Portos, the suspend and wait mechanism will become available.

As an example, say a priority function **F1** calls a *lower* priority function **F2**. And say that **F1** needs to wait for the result of **F2** before it proceeds. Since there is no wait mechanism for priority functions, then the solution is for **F1** to call **F2** and terminate. After it terminates, **F2** starts. When **F2** is done, it posts some signal **s**. The signal **s** starts a new priority function **F3** that should proceed from the point where **F1** stopped (**F1** could have attached **F3** to signal **s** before terminating).

F1 and **F3** can be implemented like a continuation or coroutine, i.e. they are the same function with a state. When called the second time, the function knows it has to proceed from where it stopped earlier. You can find more information on coroutines at this location: <http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html>.

Like signals, queues can also be used to achieve a similar behavior. **F1** inserts **F2** and **F3** in priority queue **Q** and terminates. **F2** is called first. When it terminates it releases the token via `po_queue_next (&Q)`. Then **F3** is automatically called and it should proceed from the point where **F1** stopped.

6 Dynamic Memory Management

In order for Portos to automatically manage the priority functions and their arguments, it uses its own fast dynamic memory management routines. These routines can be used by the programmer. They have two main advantages and one disadvantage.

First the disadvantage: the Portos memory allocation routines use up more memory space than traditional memory allocation algorithms. On average they use 25% more space (a less wasteful version will be provided in the future). However, this is not a real disadvantage. Many modern RTOS provide “pools of constant size blocks” for fast dynamic allocation. Such pools also waste memory because blocks cannot be recycled ubiquitously (possibly more than 25% waste on average).

Second, the important advantages:

- Portos memory allocation routines are very fast
- They prevent memory fragmentation on the long run
- They are ubiquitous and not cumbersome, unlike the pools of constant size blocks provided by many modern RTOS

For optimum performance, a programmer can use:

- The Portos memory allocation routines for short term, small to medium size blocks
- The system memory allocation routines for long term memory, large size blocks

The memory management routines are

```
void *po_malloc(int size);
void po_free(block);
```

The function `po_malloc` is quite efficient but there is an *even faster* version that can be used when `size` is a constant known at compile-time, for example 50 or `sizeof(MyStruct)`,

```
block = po_malloc_const(50);
```

WARNING: Do not use `po_malloc_const` if size is unknown (i.e. variable) at compile-time because it expands a significantly large inline function.

6.1 Dynamic Memory Regions

Many embedded systems incorporate two or more types of Random Access Memories (RAM). For example, a small but fast RAM and a large but slow RAM. Or an internal RAM where sensitive information is stored for security reasons. Hence, it is often

desirable to define a different memory heap in each RAM. Urgent code such as hardware interrupts and high priority functions can use a fast RAM's heap. Less urgent code can use a much larger heap in a slow RAM.

With Portos you can create heaps in different memory regions. The default memory region, number 0, is accessed via the default functions

```
void *po_malloc(int size);
void *po_malloc_const(int size);
void po_free(void *block);
```

To access the non-default memory region, use

```
void *po_rmalloc(int size, int region);
void *po_rmalloc_const(int size, int region);
void po_rfree(void *block, int region);
```

How do we create a heap? Call the function

```
void po_memory_create(
    int region,           // memory region number
    void *heap,          // user supplied segment of memory
    long length,         // memory heap length in char
    int maxBlockSize     // largest block size in char
);
```

heap is a pointer to a user-supplied segment of memory. The size of this segment is **length** in bytes, or more precisely in *char* on machines where *char* is 16 bits. And the maximum size of a block that can be requested (via `po_malloc`) from this memory region is **maxBlockSize**.

Before starting Portos, you must create at least the default memory region 0.

7 Miscellaneous

7.1 Entering and Exiting Interrupts

Priority functions execute at a priority level *below* hardware interrupts and below some software interrupts. Hence, if a priority function is called from interrupt level, it cannot be executed immediately. It must be delayed. But Portos does not currently auto-detect the interrupt level for two reasons: portability across platforms and efficiency. And for similar reasons, Portos does not currently auto-detect when the interrupt terminates.

Hence, the programmer must inform Portos when we are entering an interrupt before calling any priority functions. The programmer must also inform Portos when we are exiting an interrupt. Essentially, any calls to priority functions from within interrupts must be wrapped by calls to the functions:

```
po_interrupt_enter();
po_interrupt_exit();
```

WARNING 1: Interrupts that do not call priority functions do not need to call the above enter and exit functions. However, you have to be careful that certain Portos functions or certain of your own functions may themselves call priority functions, perhaps in a hidden manner. Portos cannot detect such calls and the system will eventually crash.

In general, it does not hurt to call the enter/exit wrapper somewhere toward the beginning and end of your interrupts. Unless you have a very short and urgent interrupt.

Note that the dynamic memory allocation routines of Portos don't call any priority functions and hence they can be used from interrupt level without the enter/exit wrapper.

WARNING 2: The enter/exit wrapper must only be called in hardware interrupts and high priority software interrupts that run above the priority functions level. If you have certain software interrupts that run below the priority functions level, do not call the enter/exit wrapper in such low priority interrupts.

7.1.1 Interrupt Latency

The current version of Portos is extremely flexible and allows calling any Portos function or any priority function from interrupt level, including dynamic memory allocation routines. Interrupt locking is minimally used; however, certain embedded systems may require more stringent interrupt latency. A future version of Portos could be provided for these systems where interrupt latency is decreased but the interrupts have more limitations in terms of what functions they can call.

7.2 Starting Portos

Portos is started by calling the function `po_init()`. But before you can call `po_init()` you must create at least memory region 0 as described in section 6.1.

After starting Portos, you can start calling Portos functions and priority functions.

7.3 Header Files

The header files are under `portos/include` directory. Add the path to this directory to your headers' path list, usually via a `-I` compiler option.

The only header that you need to include is `portos.h`. Add the line

```
#include <portos.h>
```

to any C file that uses Portos functions or directives. This header file automatically includes all the remaining Portos headers.

7.4 Configuring Portos

The configuration header files cannot be modified unless you have the C source files and you recompile the Portos library. However, the default configuration is suitable in most cases.

The header files under `portos/include/cfg` are the default configuration files. They are not automatically included unless you add the path to `portos/include/cfg` to your headers' path list, usually via a `-I` compiler option.

The general configuration file is:

```
po_cfg.h
```

And the target specific configuration files are:

```
po_cfg_tibios.h          Texas Instruments BIOS interface
```

For a given project, you only need to consider two configuration headers: the general `po_cfg.h` as well as one target specific header.

If you wish to modify the configuration in one of these header files, copy it to your own project directory. Modify it. And make sure that your `-I` directives point first at your own project area and then at `portos/include/cfg`. Portos tries to include the first version of these configuration headers that it finds. This is why you have to place the path to `portos/include/cfg` after the path to your own configuration file.

To be on the safe side (so that you don't end up using the wrong configuration file), copy all necessary configuration headers to your own project area, and don't add a `-I` directive to the path `portos/include/cfg`.

7.5 Compiling Portos

The Portos library files are under `portos/lib`. Add the appropriate library file to your linker's list.

Alternatively, the source files (if you have them) can be found under `portos/src` directory. You need to recompile them if you modify the default configuration. Add the following files to your project and compile them:

<code>po_memory.c</code>	Dynamic memory management
<code>po_function.c</code>	Priority functions scheduler
<code>po_signal.c</code>	Signal manager
<code>po_time.c</code>	Time manager
<code>po_queue.c</code>	Queue manager
<code>po_lib.c</code>	Library functions

Also add one interface file to the third-party specific RTOS. The list of interface files is:

<code>po_target_tibios.c</code>	Texas Instruments BIOS interface
---------------------------------	----------------------------------

7.6 Portos Preprocessor

The Portos preprocessor automatically handles priority functions and their arguments. It greatly simplifies the job of a software programmer. For Microsoft Windows, the Portos preprocessor is the executable `portos/bin/po_preprocess.exe`. A version for Linux will be available in the future.

Every file that includes the header `portos.h` must be preprocessed before it can be compiled. *The current version of the Portos preprocessor works after the C preprocessor but before the C compiler.*

In the make file (or equivalent) of your project you need to call the C preprocessor first to obtain a C preprocessed file. Using `gcc`, for example, invoke the command

```
gcc -E myfile.c -o myfile.pp.c
```

The `-E` option instructs `gcc` to output a C preprocessed file, `myfile.pp.c`, without performing the compilation. Then call the Portos preprocessor as follows

```
po_preprocess myfile.pp.c myfile.po.c
```

`myfile.po.c` is a C preprocessed and Portos preprocessed output file. You can finally run the compiler on this file to produce the object file,

```
gcc -c myfile.po.c -o myfile.o
```

For simplicity, you could C preprocess and Portos preprocess all files before invoking the compiler. There is no harm in running the Portos preprocessor on a C file that does not include `portos.h`.

The Portos preprocessor is very similar to the C preprocessor in behavior. It can fail, just like the C preprocessor, if some code section is not properly bounded, for example. You can usually detect the error after the compilation phase fails. You can also refer to the Portos preprocessor output file (e.g. `myfile.po.c`) to check if there is any issue there.

7.7 Debugging the Real-Time Program

Portos is full of features to help you debug your program and to report error messages. In the first phase of your product, you should enable debugging. This can be done by defining the macro `DEBUG` or `_DEBUG` in the command line of your compiler, usually via a `-D` compiler option.

Most of the debugging features and error reports are turned off in the second phase, i.e., the release version. It is therefore harder to debug problems in the release version.

7.7.1 Error Messages

The errors reported by Portos are listed in the following table. In order to obtain more error reporting, set the value of the macro `DEBUG` or `_DEBUG` to 2. But it may slow down your program. Set the value to 1 for less error reporting, and 0 for minimal error reporting (Release mode).

Portos Error Code	Description
100	Heap memory is full
101	Heap memory is corrupt
102	Invalid memory region index
103	Freeing a heap memory block more than once
104	Calling <code>po_free</code> with a <code>NULL</code> pointer
105	Block too large, above maximum block size for this memory region
106	Cannot free a block allocated by <code>po_malloc_forever</code>
200	Linked list is corrupt (<code>po_list</code>)
400	Priority level of priority function is out of range
401	<code>po_priority_raise</code> called with a lower priority level

500	Posting signal out of range (<code>hashSize</code> is not a power of 2)
501	Attaching to signal out of range (<code>hashSize</code> is not a power of 2)
502	Invalid signal group index
503	Signal handle is corrupt or uninitialized
504	Attaching a signal handle that is already in-use
600	Invalid clock index
1100	Failed to create Software Interrupt

Note that timers are a type of signals and that errors reported for signals can actually be related to timers.

In the default configuration, when Portos catches an error, it prints the error number and then aborts the program. Currently, Portos does not print the file name and line number where the error may have originated. This feature maybe added to Portos in the future. But the best way to debug errors is to place a breakpoint where Portos catches the error. In the header file `po_error.h`, place a breakpoint inside the inline function `po_abort`. When Portos catches an error the program stops at this breakpoint. By checking the stack content, you find the list of functions where the error may have originated.

7.7.2 Debugging Dynamic Memory

Portos' dynamic memory allocation routines keep track of the source file and line number of each memory allocation request. Therefore, you can quickly find the origin of any memory leak. This option is also available in non-debug mode and can be turned off in the final product.

7.7.3 Debugging Priority Functions

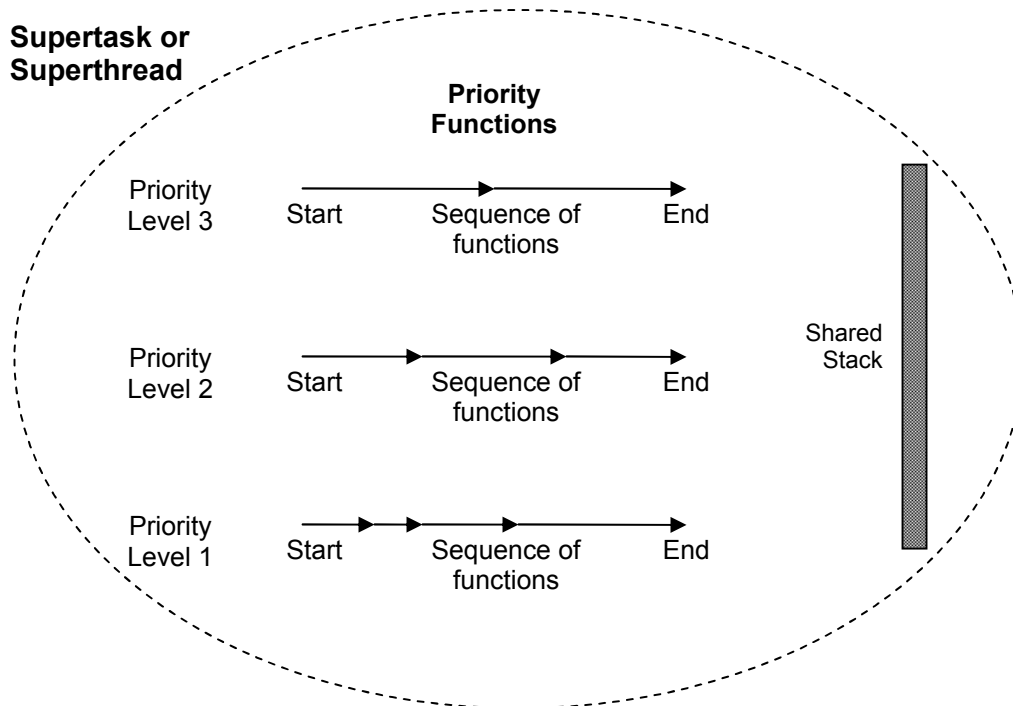
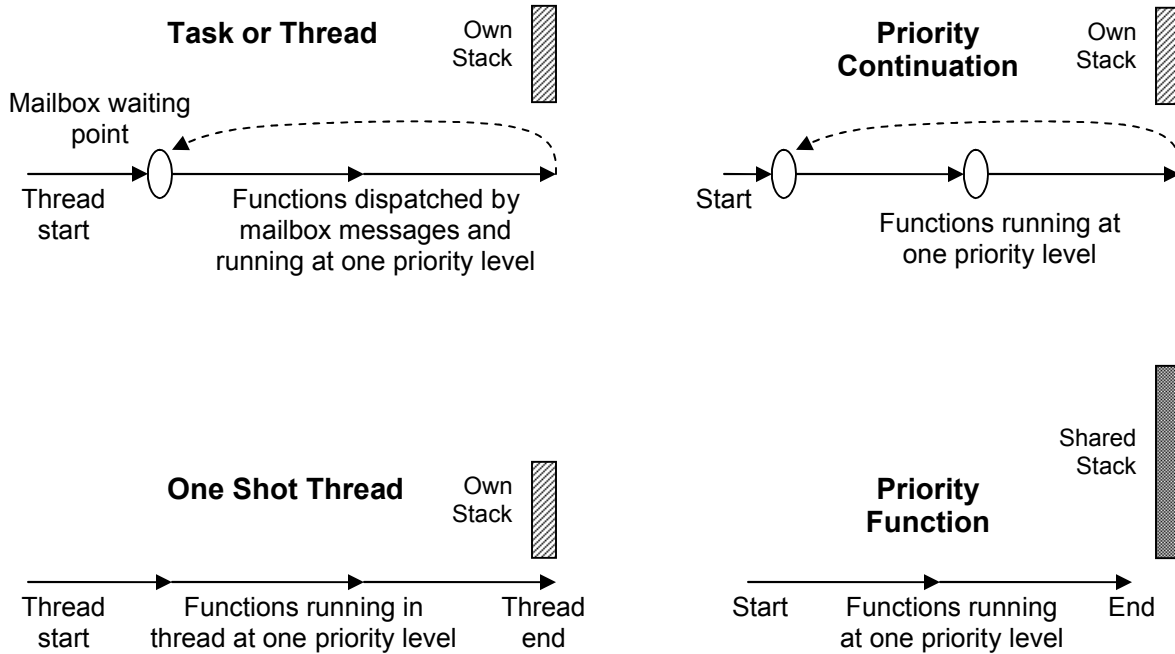
In debug mode, Portos keeps track of every priority function that has started or that is waiting to start. You can analyze if your program's functions are being properly called.

7.7.4 Debugging Timers and Signals

In debug mode, Portos keeps track of every priority function attached to a timer or a signal.

8 Tasks vs. Priority Functions

For the curious reader, in this section we explain the philosophy behind Portos.



A *task* or *thread* is a *sequence of function calls* that run at a given priority level and that use a dedicated stack. In one version, the task never ends: it loops waiting for new requests arriving via messages in a mailbox, and it services them. In the *one shot thread* version, on the other hand, a sequence of functions is started and ended. In both cases, there is a dedicated stack that allows the task to be suspended and resumed at any point in time. The state of the task is preserved on its own stack.

A priority function is somewhat similar to a one shot thread. It is a sequence of functions running at one priority level, except that it does not have its own stack. It shares a stack with other priority functions. Hence, a priority function cannot suspend and resume. If it were to suspend, other priority functions using the same stack would be affected.

A priority continuation is a variation of a priority function that owns a dedicated stack (or equivalent). Hence, a continuation is able to suspend and resume. It can be viewed as a lightweight task, depending on how it is implemented.

By having a dedicated stack, a task can be paused and resumed anytime. It can be paused, for example, to wait for an event or to let other tasks run (time-slicing). While this appears to be an advantage, in reality it complicates matters a lot, as we shall see.

8.1 Priority Queues of Requests

What is a real-time program? Basically, it is a program that receives requests via hardware interrupts and should respond to them as soon as possible. If the system has infinite processing speed, the requests are promptly serviced and there is little need for an RTOS scheduler: no need for tasks or priority functions (except perhaps to isolate and protect the memory of different pieces of code but that can be achieved separately).

But processing speed is never infinite and is often the bottleneck. Hence, jobs are organized into urgent and less urgent categories. We can view the scheduler as a set of *priority queues of requests*: urgent requests are inserted in the high priority queues. Less urgent requests are inserted in the low priority queues.

A task or a priority function is essentially a priority queue of requests. It processes requests at a given priority level. The main problem with tasks is that the simple concept of priority queue is implemented with great complexity. A list of things that a task needs:

- A dedicated stack.
- A mailbox for inter-task communication: it is not possible to directly call a function that should run in another task. In order to make such a call, a programmer must create a message, fill it out with the necessary information (such as function to be called and arguments), and send the message to the desired task's mailbox. When the task receives a message, it should read the content, unpack the arguments and call the appropriate function.
- Since tasks pause and resume, therefore CPU registers and other state information have to be saved and restored each time. A *context switch* occurs when we switch between tasks. Too many context switches can slow down the program.

Priority functions, on the other hand, are a very straightforward and efficient way of implementing the priority queues of requests. Priority functions can be directly called from anywhere, hence no need for a mailbox. The shared stack reduces memory requirements. No context switches take place.

Since priority functions can be *directly called from anywhere*, without the need for inter-task messages, the software tends to remain more modular and more upgradeable. And the programmer has more time to spend on the real application rather than on RTOS considerations.

8.2 Supertasks

Portos resolves the complexity of the task via the following steps:

- By using the one shot thread method, we *get rid of the mailbox and context switch*: every job or request initiates a one shot thread. All arguments are supplied at the start of the thread. The thread terminates when the job is done.
- By sharing the stack we allow for a *large number of one shot threads*. Hence, rather than dedicating a stack to each thread, we do the opposite: we combine several one shot threads and assign them a common stack.
- By using priority functions in place of the usual threads, we *allow the compiler to optimize the priority function start* (thread start).
- By grouping priority functions into a *supertask* or *superthread*, we achieve the power of a task or thread, and more.
- By using a special preprocessor, we *greatly simplify the handling of messages, and the packing and unpacking of arguments*.

We denote by supertask, or superthread, a collection of one shot threads using a common stack. This is a new concept that will be soon implemented in Portos. A supertask provides the capabilities of a task, and more. The programmer will be able to define a priority function that executes in some supertask as follows,

```
void myfunc(po_priority(P, T), int x, double y, char *z)
{
    ...
}
```

where P is the priority level and T is the supertask inside which the function should execute. T can be a dynamic variable or any C expression. T can be a macro, for example, that you can set to any supertask handle. This way, you can instantly modify how priority functions are grouped inside supertasks, and you can instantly merge or split supertasks. After debugging is done, you could merge several supertasks and save stack space and context switching.

Like a task, a supertask can suspend and hence time-slicing can be achieved. In addition, a supertask can suspend either entirely or partially! For instance, we can

choose to suspend all priority functions or threads within the supertask that have a priority level below \mathfrak{P} . Priority functions or threads with priority levels above \mathfrak{P} can continue to run. With the shared stack of the supertask, we can avoid creating too many small threads.

9 Benchmark

A comprehensive and fair benchmark is beyond the scope of this document. Here we give a vague idea of the performance of Portos.

You can find among the demo programs a simplistic benchmarking code. Three versions are written, one using TI BIOS tasks, one using TI BIOS software interrupts (SWI) and one using the priority functions of Portos.

The benchmarking code consists of a continuous ping pong between 2 tasks, or 2 software interrupts (SWI), or 2 priority functions. The first task calls second task and goes to wait state. Then the second task calls the first task and goes to wait state. And so on. During this ping pong match, we measure how many times the tasks are called. And the same is done for SWI and priority functions. The higher the number of times the tasks, SWI or the priority functions are called, the faster the context switching is.

The program was simulated on a TI C62xx DSP, build tools version 6.1.7. A timer interrupt triggered the start and stop of the simulation. A counter counted the number of times one of the tasks, or SWI or priority function was called. The “Release” configuration was used. The table below shows the counter value. The higher this counter is, the higher the performance.

	Counter value reached between 2 timer interrupts
Tasks	228
SWI	2227
Priority functions	980 ¹

Switching between priority functions is faster than switching between tasks but slower than switching between SWI. However, priority functions may outperform SWI if the scheduler is written in machine language (it is currently written in C language and it is not yet fully optimized). Note that priority functions offer far more flexibility than SWI, and hence the scheduler is more complex.

¹ The priority levels of the priority functions are chosen to be different. If we choose equal priority levels then the performance decreases to 450. The reason is that the scheduler manages to optimize the switching when we “raise” the priority level, but not when we lower it or keep it constant.

9.1 Code Size

In the current implementation, the Portos kernel adds a little more memory on top of the TI BIOS, since Portos is a layer that uses the TI BIOS itself. But certain functions of the TI BIOS are unused and not linked in, and some savings are achieved in this way.

Moreover, many code reductions are still possible in Portos (some create functions can be made static rather than dynamic) and the increase in code size may become negligible. We will improve the code size soon.

Note that a version of Portos that does not use the TI BIOS can be quite tiny, but currently unavailable.

10 Reference

10.1 Dynamic Memory Management

```
void *po_malloc(int size)
void *po_rmalloc(int size, int region)
```

size	block size in bytes (or <code>char</code> more precisely).
region	memory region (defaults to region 0 in <code>po_malloc</code>).
RETURNS	pointer to allocated block (<code>NULL</code> if it failed, or system abort).

Fast dynamic memory allocation using Portos memory manager.

All functions can be called from any context, including interrupt context. But for fastest results, and whenever possible, you may want to use `po_malloc_const` where `size` is a constant known at compile time.

Portos allows defining many dynamic memory regions, for example, one associated with the fast internal RAM, another one with the slow external RAM, etc. If the region index is not specified, then it defaults to region 0.

The functions return a pointer to the allocated block when they succeed. If they fail, they either return a `NULL` pointer or they can be configured to abort the program (cf. `portos/include/cfg/po_cfg.h`) so that you can catch the problem without your additional test code, i.e. saving code space.

In order to be fast, these routines use more memory than typical `malloc`, or equivalent routines. But the advantages are many: in addition to being much faster, they prevent memory fragmentation on the long run. Memory fragmentation can result in a system crash. For best performance, you could use these functions for small size blocks and use the system `malloc` for large size blocks.

NOTE: the dynamic memory allocation API may be slightly modified in a future version. In particular, the integer region argument may be replaced by a pointer to a structure for faster access.

```
void *po_malloc_const(int size)
void *po_rmalloc_const(int size, int region)
```

size block size in bytes (or `char` more precisely). It must be constant.

region memory region (defaults to region 0 in `po_malloc_const`).

RETURNS pointer to allocated block (`NULL` if it failed, or system abort).

Even faster memory allocation using Portos dynamic manager. However, in order to call these routines, `size` must be a constant known at compile time. You have to be careful when using these functions. Unfortunately, current compilers do not provide a directive to tell if the input argument to an inline function or a macro is a known constant.

Other than the constraint on `size`, the behavior of these functions is similar to the slower version `po_malloc`. The idea of a constant `size` emulates what many modern RTOS denote by “pool of constant size blocks”. In Portos, such pools are automatically handled to simplify the programmer’s job.

```
void po_free(void *block)
```

block block to be freed.

Frees a block allocated with the memory allocation routines of Portos.

```
void po_memory_create(int region, void *heap, long length,
                     int maxBlockSize)
```

region index of memory region.

heap pointer to user supplied heap area.

length length of heap area.

maxBlockSize maximum block size that can be allocated in this region (in bytes).

Creates a memory region. The programmer must create at least memory region 0. It is internally used by the Portos library.

NOTE: the dynamic memory allocation API may be slightly modified in a future version. In particular, the region may be split into a distinct control structure (e.g. in fast RAM) and data memory (e.g. in slow RAM).

10.2 Priority Functions

```
po_priority(int P)
```

P priority level, and can be any C expression.

Directive for declaring a priority function. E.g.

```
void myfunc(po_priority(5), complex x, int y)
{
  ...
}
```

The range of allowed priority levels is 0 to **po_priority_MAX**. The higher the priority number means the higher the priority level (contrary to certain OS where the lower the priority number means the higher the priority level).

Since priority functions currently run at software interrupt level, a range of software interrupts is reserved for priority functions. This range is mapped onto the priority levels 0 to **po_priority_MAX** (e.g. cf. **portos/include/cfg/po_cfg_tibios.h**).

```
po_priority
```

Directive for calling a priority function. E.g.

```
myfunc(po_priority, a, b);
```

```
int po_priority_get(void)
```

RETURNS current priority level.

```
int po_priority_raise(int P)
```

P new priority level.

RETURNS old priority level (used by **po_priority_restore**).

Raises the priority level. Usually used to protect a critical region (priority ceiling). Old priority level is restored via function **po_priority_restore**.

```
void po_priority_restore(int oldPriority)
```

`oldPriority` previous priority level.

Restores the priority level following a call to function `po_priority_raise`. The argument passed here is the one returned by `po_priority_raise`.

10.3 Interrupts

```
void po_interrupt_enter(void)
```

This function should be called at the start of a hardware interrupt before calling any priority functions. It informs Portos that we are at interrupt level.

This function should also be called at the start of a software interrupt if it has a higher priority level than priority functions.

Basically, a range of software interrupts is reserved for priority functions (for an example, cf. `portos/include/cfg/po_cfg_tibios.h`). These software interrupts are automatically used by Portos. But any other user-defined software interrupt that has a priority level above this range must call `po_interrupt_enter` before it calls any priority function.

```
void po_interrupt_exit(void)
```

This function should be called before exiting a hardware or software interrupt that has higher priority level than priority functions.

The pair `po_interrupt_enter` and `po_interrupt_exit` should wrap any priority function calls from within interrupt level.

10.4 Signals

```
po_signal(int S, int group)
po_signal(int S, int group, po_signal_Handle *handle)
```

s signal value.

group signal group index (typically 0).

handle optional: signal handle.

Directive for attaching a priority function to a signal. When the signal **s** is posted, the function is called. The optional signal handle gives the possibility to detach (cancel) the call before the signal is posted. E.g.

```
myfunc(po_signal(10,0,&myhandle), a, b);
```

```
void po_signal_init(po_signal_Handle *handle)
```

handle signal handle.

Function to initialize a signal handle. A statically allocated handle can also be initialized via `po_signal_INIT` macro.

```
po_signal_Handle handle = po_signal_INIT;
```

handle signal handle.

Macro to initialize a handle that is statically allocated (i.e. not allocated in dynamic heap).

```
void po_signal_post(int S, int group)
```

s signal value.

group signal group index (typically 0).

Posts a signal. All priority functions attached to the signal are readied to be called. They will be called in the proper priority order. However, until they are effectively called, they can still be canceled at this stage.

```
void po_signal_detach(po_signal_Handle *handle)
```

handle signal handle.

Detaches a signal handle, and the corresponding priority function is not called (unless it is too late).

```
int po_signal_isactive(po_signal_Handle *handle)
```

handle signal handle.

RETURNS non-zero if handle is active.

Returns a non-zero value if the signal handle is active, i.e. in-use, i.e. the corresponding function has not yet been called.

```
void po_signal_creategroup(int group, int groupPriority,
                           int hashSize, int memRegion)
```

group signal group index.

groupPriority priority level of signal group (for internal priority functions).

hashSize size of hash table (internal database).

memRegion dynamic memory region (for internal memory allocation).

Creates a signal group. In most cases, you only need a unique signal group (index 0). But you can create more groups when needed.

Each group can handle a range of signals. The range of signals is either

- 0 to **hashSize** - 1 if **hashSize** is not a power of 2
- the entire range of integers if **hashSize** is a power of 2

The reason for this distinction is that the operation **x modulo m** is slow unless **m** is a power of 2. Hence, we restricted the signals to the range **[0, hashSize-1]** when **hashSize** is not a power of 2.

The larger **hashSize** is, the more efficient the internal database will be, and the faster the signal routines will execute. But more memory is used.

If you know your signal range is only 0 to 9, then there is no point in using a **hashSize** greater than 10. On the other hand, if your signals can span the entire range of integers (i.e. they can take any integer value) then you must choose **hashSize** as a power of 2.

groupPriority is the priority level of internal priority functions. For proper behavior, choose this priority level above the priority level of any priority function that can be attached to signals in this group. Usually, you can set **groupPriority** to the highest priority level available (e.g. **po_priority_MAX**).

memRegion is the index of the dynamic memory region that will be used by internal memory allocation.

10.5 Timers

```
po_time(int T, int clock)
po_time(int T, int clock, po_time_Handle *handle)
```

T	time value.
clock	clock index.
handle	optional: timer handle.

Directive for calling a priority function at a time in the future. When the time is reached by the selected clock, the function is called. The optional time handle gives the possibility to cancel the timer (cancel the call) before the time is reached. E.g.

```
myfunc(po_time(123,0,&myhandle), a, b);
```

```
void po_time_init(po_time_Handle *handle)
```

handle	timer handle.
---------------	---------------

Function to initialize a timer handle. A statically allocated handle can also be initialized via `po_time_INIT` macro.

```
po_time_Handle handle = po_time_INIT;
```

handle	time handle.
---------------	--------------

Macro to initialize a handle that is statically allocated (i.e. not allocated in dynamic heap).

```
void po_time_tick(int clock)
```

clock	clock index.
--------------	--------------

Increments the clock time. All priority functions that are scheduled at the new time are readied to be called. They will be called in the proper priority order. However, until they are effectively called, they can still be canceled at this stage.

```
void po_time_cancel(po_time_Handle *handle)
```

handle time handle.

Detaches a timer handle, and the corresponding priority function is not called (unless it is too late).

```
int po_time_isactive(po_time_Handle *handle)
```

handle timer handle.

RETURNS non-zero if handle is active.

Returns a non-zero value if the timer handle is active, i.e. in-use, i.e. the corresponding function has not yet been called.

```
int po_time_get(int clock)
```

clock clock index.

RETURNS current clock time.

Returns the current clock time.

```
void po_time_set(int time, int clock)
```

time time value.

clock clock index.

Sets the clock to the desired time. Typically used to wrap the clock time for clocks that do not span the entire integer range. For example, if the clock should vary between 0 and 5, then you can set its value back to 0 before it increments to 6.

```
void po_time_createclock(int clock, int clockPriority,
                        int hashSize, int memRegion)
```

clock clock index.

clockPriority priority level of clock (for internal priority functions).

hashSize size of hash table (internal database).

memRegion dynamic memory region (for internal memory allocation).

Creates a clock.

Each clock can handle a range of times. The range of times is either

- 0 to **hashSize** - 1 if **hashSize** is not a power of 2
- the entire range of integers if **hashSize** is a power of 2

The reason for this distinction is that the operation $x \bmod m$ is slow unless m is a power of 2. Hence, we restricted the times to the range $[0, \text{hashSize}-1]$ when **hashSize** is not a power of 2.

The larger **hashSize** is, the more efficient the internal database will be, and the faster the time routines will execute. But more memory is used.

If you know your time range is only 0 to 9, then there is no point in using a **hashSize** greater than 10. On the other hand, if your time range can span the entire range of integers (i.e. they can take any integer value) then you must choose **hashSize** as a power of 2.

clockPriority is the priority level of internal priority functions. For proper behavior, choose this priority level above the priority level of any priority function that can be scheduled using this clock. Usually, you can set **clockPriority** to the highest priority level available (e.g. **po_priority_MAX**).

memRegion is the index of the dynamic memory region that will be used by internal memory allocation.

10.6 Queues

```
po_queue(po_queue_Queue *Q)
```

Q pointer to queue.

Directive for inserting a priority function in a queue of priority functions. The function is immediately called if there are available tokens in this queue. Otherwise it waits until a token becomes available. E.g.

```
myfunc(po_queue(&myqueue), a, b);
```

When the function `myfunc` is called, it is implicitly assigned a token. It should release the token when it no longer needs it (via `po_queue_next`). As soon as the token is released, the next priority function waiting in the queue can be called.

```
void po_queue_init(po_queue_Queue *Q, int nServers,
                  int memRegion)
```

Q pointer to queue.

nServers number of servers that service this queue (often 1).

memRegion dynamic memory region (for internal memory allocation).

Function to initialize a queue. A statically allocated queue can also be initialized via `po_queue_INIT` macro. After initialization, the number of tokens available for this queue is equal to `nServers`.

```
po_queue_Queue Q = po_queue_INIT(&Q, int nServers,
                                 int memRegion);
```

Q pointer to queue.

nServers number of servers that service this queue (often 1).

memRegion dynamic memory region (for internal memory allocation).

Macro to initialize a statically allocated queue. After initialization, the number of tokens available for this queue is equal to `nServers`.

```
void po_queue_next(po_queue_Queue *Q)
```

Q pointer to queue.

Releases a token (i.e. semaphore) that can be used by the next priority function waiting in the queue. For every priority function we insert in the queue, we must release the implicitly assigned token at some point.