

Appendix A – Porting Notes

TargetFFS was developed using TargetOS, Blunk Microsystems' real-time kernel and has been hosted on pSOS⁺, VxWorks, and other kernels. The code is 100% ANSI C and is easy to port. This appendix discusses the issues involved in porting TargetFFS.

Kernel Objects

TargetFFS uses two semaphores to protect critical sections and support operation in a multitasking environment. The semaphore calls used are those that create a semaphore, get a semaphore token, and give up a semaphore token. The easiest approach for porting is to write wrapper functions that implement the TargetOS `semCreate()`, `semPost()`, and `semPend()` functions using the RTOS being ported to, so that it “looks” like TargetOS to the TargetFFS code. The TargetOS manual pages for these calls are included at the end of this appendix.

Start-up Sequence

TargetFFS is integrated with TargetOS's start-up sequence to ensure TargetFFS is initialized before any task could call the file system. Again, this should be a trivial change. You can call the TargetFFS initialization routine from your initial task, ensuring it is called before any TargetFFS calls are made.

Name Collision

With the TargetOS runtime library, TargetFFS supports the file-related functions from both the Standard C and POSIX standards. To prevent linker collision with other Standard C libraries, the Standard C interface is normally omitted when porting to another RTOS. Except for `rename()`, the Standard C API is a subset of the POSIX API in functionality.

If name collision occurs with the TargetFFS POSIX routines, the names can be “mangled” so that they do not conflict with other libraries (Ex. use `FfsRead()` instead of `read()`).

Memory Allocation

TargetFFS uses `malloc()` and `free()` to allocate memory for control structures and cache buffers during operation. The RTOS being ported to must support re-entrant `malloc()` and `free()` functions.

OsSecCount

TargetFFS uses a global variable, `OsSecCount`, to set file access and modification times. This variable has type “long” and is updated by TargetOS once a second. If the system ported to maintains the time-of-day, `OsSecCount` should be initialized at startup using `mktime()` and thereafter incremented once a second. If this is not done, the access and modification times returned by `stat()` should be ignored.

semCreate()

PROTOTYPE

```
#include <kernel.h>
```

```
SEM semCreate(char name[8], int init_count, int mode);
```

DESCRIPTION

semCreate() is used to allocate and initialize a semaphore. If successful, **semCreate()** returns a pointer to the semaphore control block. This identifier is used in subsequent semaphore-related service calls. If a nonfatal error occurs, *errno* is set to the appropriate error code and NULL is returned. It is a fatal error to call **semCreate()** from an interrupt service routine.

name points to a string used to assign the semaphore name. Characters are copied until either NULL or a total of eight characters are reached. The name is useful for debugging. The kernel does not use it. No check is made to ensure semaphore names are unique. A semaphore name can be converted to an identifier using **semGetId()**. If names are duplicated, **semGetId()** returns the identifier of the semaphore created first.

Semaphores maintain an internal counter. If positive or zero, the count indicates the number of tokens available. If negative, it indicates the number of tasks blocked on the semaphore, waiting for tokens to be posted by **semPost()**. *init_count* is the initial semaphore count. It must be either zero or a positive number.

mode must be set to either OS_FIFO or OS_PRIORITY. *mode* determines how a task is selected when **semPost()** is called on a semaphore with waiting tasks. If the mode is OS_FIFO, the task that has been waiting longest is made ready. If the mode is OS_PRIORITY, the task with the highest priority is made ready.

Semaphore memory is dynamically allocated from the kernel memory pool when a semaphore is created. If FIFO task queuing is selected, creating a semaphore requires a total of 32 bytes for the semaphore control block. If priority task queuing is used, a semaphore requires a total of 320 bytes: 64 bytes for the control block and 256 bytes for the priority queue.

ERROR CODES

OS_NO_MEMORY	semCreate() was unable to allocate the memory needed.
OS_PARM_ERR	<i>init_count</i> is negative or <i>mode</i> is neither OS_FIFO nor OS_PRIORITY.

EXAMPLE

```
/*-----*/
/* Initializaton for TCP socket list. */
/*-----*/
TcpSockList.head = NULL;
TcpListSem = semCreate("tcp list", 1, OS_FIFO);
```

semPend()

PROTOTYPE

```
#include <kernel.h>

int semPend(SEM sem, ui32 wait_opt);
```

DESCRIPTION

semPend() is used to request a token from a semaphore. *sem* is a semaphore identifier previously returned by **semCreate()**. If successful, **semPend()** returns 0. Otherwise, *errno* is set to the appropriate error code and -1 is returned. If a token is available, **semPend()** always returns immediately.

If a token is not available and the caller is an interrupt service routine, **semPend()** immediately returns -1 with *errno* set to OS_WOULD_BLOCK, regardless of the value of *wait_opt*. Table 3.2 shows the effect of *wait_opt* if a token is not available and the caller is a task. If a timed wait is selected and a token is not posted within *wait_opt* kernel ticks, **semPend()** returns -1 with *errno* set to OS_TIMED_OUT.

<i>wait_opt</i> value	Action if no token available
NO_WAIT (0)	Immediately returns -1 with <i>errno</i> set to OS_WOULD_BLOCK
WAIT_FOREVER (-1)	Waits indefinitely for token to be posted
Other	Waits up to <i>wait_opt</i> kernel ticks for token to be posted.

Table 3.2 Effect of *wait_opt* when task calls **semPend()**

Determining an appropriate tick count wait requires knowing the kernel tick frequency. At startup, the board support package tick interrupt driver starts the tick interrupt and initializes the global variable *OsTicksPerSec* to the kernel tick frequency. A typical value is 100 ticks per second.

When a token is posted to a semaphore with multiple tasks waiting, the semaphore's mode determines which task is made ready. If the semaphore was created in OS_FIFO mode, the task that has been waiting longest is made ready. If the mode is OS_PRIORITY, the highest priority task receives the token and is made ready.

Semaphores are general-purpose tools, useful for both resource allocation and task synchronization. They have a small memory footprint and low processing overhead. Unless the priority inheritance of a mutex, the broadcasting or transitory behavior of a nexus, or the data passing ability of a queue is needed, a semaphore should be used.

ERROR CODES

OS_INVALID_ID	<i>sem</i> is not a valid semaphore identifier.
OS_SEM_DELETED	The semaphore was deleted while the calling task was waiting for a token.
OS_TIMED_OUT	The calling task timed out while waiting for a token.
OS_WOULD_BLOCK	Semaphore has no tokens and either <i>wait_opt</i> is NO_WAIT or caller is an ISR.

EXAMPLE

```
/*-----*/
/* Try to get a token. Maximum wait is two seconds. */
/*-----*/
if (semPend(BufSem, 2 * OsTicksPerSec)) return NULL;
```

semPost()

PROTOTYPE

```
#include <kernel.h>

int semPost(SEM sem);
```

DESCRIPTION

semPost() is used to post a semaphore token. If successful, **semPost()** returns 0. Otherwise, *errno* is set to the appropriate error code and -1 is returned.

sem is the identifier of a semaphore previously created by **semCreate()**. If a token is posted to a semaphore with multiple tasks waiting, the semaphore's mode determines which task is made ready. If the semaphore was created in OS_FIFO mode, the task that has been waiting longest is made ready. If the mode is OS_PRIORITY, the highest priority task receives the token and is made ready.

If the kernel is configured to be preemptive and the task that receives the token has a higher priority than the running task, then the running task is preempted. The kernel places the running task on the kernel's ready list and switches to the task made ready by the semaphore token post.

Semaphores are general-purpose tools, useful for both resource allocation and task synchronization. They have a small memory footprint and low processing overhead. Unless the priority inheritance of a mutex, the broadcasting or transitory behavior of a nexus, or the data passing ability of a queue is needed, a semaphore should be used.

ERROR CODES

OS_INVALID_ID *sem* is not a valid semaphore identifier.

EXAMPLE

```
/* ***** */
/* RetBuffer: Returns one buffer to free buffer pool */
/* ***** */
/* Input: buf_ptr = pointer to buffer being returned */
/* ***** */
void RetBuffer(BufType *buf_ptr)
{
    isrMask();
    buf_ptr->next = FreeHead;
    FreeHead = buf_ptr;
    isrUnmask();
    semPost(BufSem);
}
```