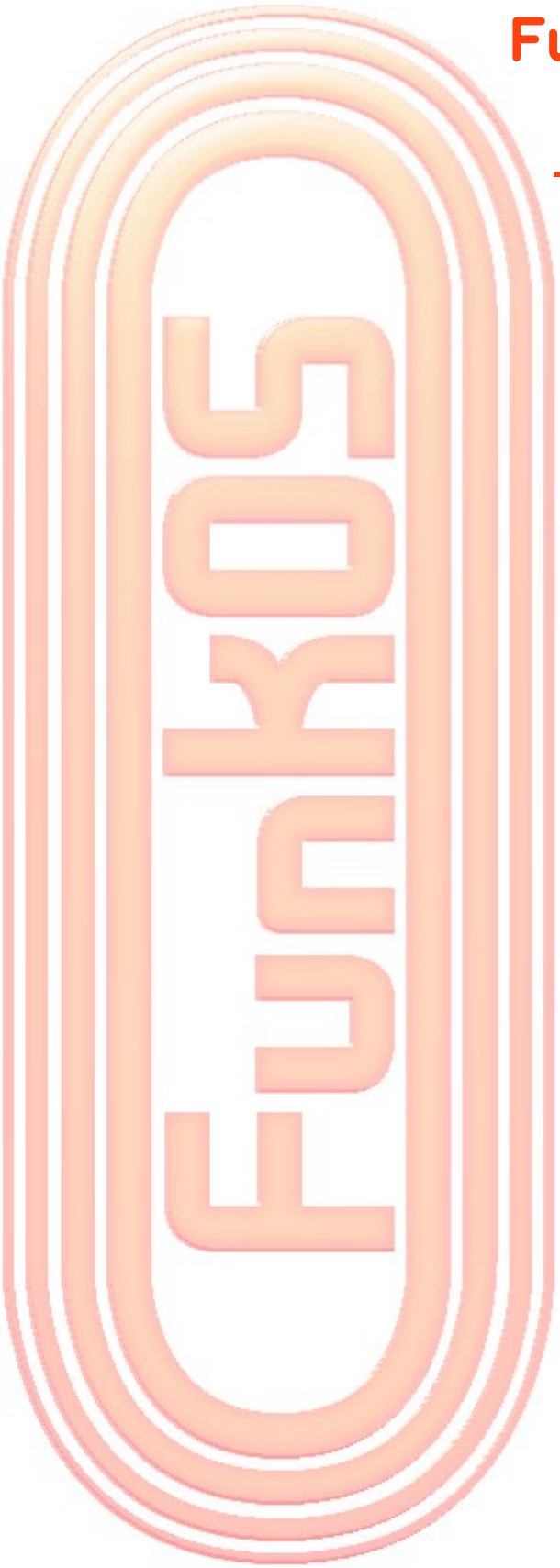

FunkOS++ RTOS Kernel Reference Guide

FunkOS Version R3

March 20, 2010

Mark Slevinsky
Funkenstein
Software Consulting

Table of Contents

Document History:....4
Introduction:....5
Features:....5
  Flexible Scheduler....5
  Resource Protection....5
  Synchronization Objects....5
  Timer Callbacks....6
  Driver API....6
  Robust Interprocess Communications....6
  Dynamic Memory....6
Multiprogramming in FunkOS++:....6
  Implementing Tasks....6
  The FunkOS++ Scheduler....7
  Creating a Task....7
  Adding a task to the scheduler....7
  Starting the scheduler....8
  Suspending a task....8
  The Idle Task....9
  Invoke a task switch....10
Lightweight Timer Threads....10
  The Lightweight Timer Context....10
  Creating a Timer Object....10
  Starting a Timer Object....10
  The Timer Callback....11
Task Synchronization:....11
  Initialization....11
  Waiting....11
  Posting....12
Resource Protection:....12
  Initializing....12
  Claiming a Mutex....12
  Releasing a Mutex....12
  Resource protection example....13
System Watchdog Interface....13
  Initializing the Watchdog Module....13
  Starting a Watchdog-Enabled Code Block....14
  Ending a Watchdog-Enabled Code Block....14
  Removing a Watchdog-Enabled Code Block....14
  Watchdog-Enabled Code Block....14

Event Queues.....15
Creating event queues.....15
Using event queues.....16
Dynamic Memory:.....16
Heap Implementation.....16
Customizing The Heap.....17
Initializing the Heap.....17
Allocating Memory.....18
Deallocating Memory.....18
Device Drivers:.....18
The Device Driver Type.....19
Implementing a Driver.....20
Starting/Stopping drivers.....20
Driver I/O operations.....21
Contact.....21
Contribute!.....22

Document History:

Rev	Date	Description	Author
A	Feb, 2010	First release	moslevin

Introduction:

FunkOS++ is a variant of the FunkOS RTOS kernel, which was ported from the original C implementation to C++. Since the original kernel used a number of OO constructs, the porting process was quite trivial, taking only a matter of hours to translate the kernel into its various classes. The end result is an easy to read, easy to use real-time-kernel with comparable performance to the original C version, allowing application programmers to take full advantages of a high-quality RTOS without having to sacrifice objected oriented concepts. Each feature in FunkOS++ is based off of an equivalent from the FunkOS kernel, as a result, it is easy to port programs written for FunkOS to FunkOS++.

Currently, FunkOS++ is only supported by GCC (WinAVR), but it should be simple to provide ports for other targets based on the existing C code.

This document is meant to serve as a companion to the FunkOS manual – you should read and understand the object model and concepts in that document before using the FunkOS++ kernel. For more documentation on the FunkOS and FunkOS++ APIs, please refer to the doxygen documentation for each.

Features:

FunkOS++ is a fully-featured real-time kernel, and is feature-competitive with other open-source and commercial RTOS's in the embedded arena. The key features of this kernel are:

Flexible Scheduler

- Unlimited number of tasks with 255 priority levels
- Unlimited tasks per priority level
- Round-robin scheduling for tasks at each priority level
- Time quantum scheduling for each task in a given priority level
- Optional watchdog timer objects to ensure all tasks meet their deadlines
- Configurable stack size for each task

Resource Protection

- Integrated mutual-exclusion semaphores (mutex)
- Priority-inheritance on mutex objects to prevent priority inversion

Synchronization Objects

- Binary/counting semaphores to coordinate task execution

Timer Callbacks

- Any number of timer callbacks can execute at configurable intervals
- Timer phasing can be specified to provide even processor load

Driver API

- A hardware abstraction layer is provided to simplify driver development

Robust Interprocess Communications

- Threadsafe event queues configurable for each task

Dynamic Memory

- Simple fixed block-size heap implementation

Multiprogramming in FunkOS++:

Implementing Tasks

Multithreading in FunkOS++ is accomplished by declaring a number of thread objects (tasks) that are managed by the FunkOS Scheduler. A task is implemented by declaring an instance of a class that inherits from the base class `FunkOS_Task`. The inheriting class is responsible for both declaring and assigning the task's **stack, stack size** and **entry function** from the constructor. An example of such a class is presented below:

```
//---------------------------------------------------------------------
class Task1 : public FunkOS_Task
{
public:
      Task1();                    // Required constructor
private:
      WORD m_awStack[128];   // Declare the thread stack
      static void RunMe(void*);    // Declare the thread's entry function
};

//---------------------------------------------------------------------
Task1::Task1()
{
      m_pfFunc = (&Task1::RunMe); // Assign the entry function
      m_usStackSize = 128;              // Assign the stack size
      m_pwStack = m_awStack;       // Assign the stack
}

//---------------------------------------------------------------------
void Task1::RunMe(void *pvThis_)
```

```
{
	// Because static functions don't get access to the "this pointer"
	// we pass it in as a parameter to the task's entry function instead.
	Task1 *pstThis = (Task1*)pvThis_;

	while(1)
	{
		// Do stuff...
	}
}
```

The FunkOS++ Scheduler

The scheduler is functionally equivalent to the "normal" FunkOS scheduler, implemented as a class containing only static methods. As a result, its usage and syntax is very similar, and does not require a user to declare any additional objects.

Before tasks can be added to the kernel's scheduler, the scheduler must first be initialized. This is accomplished by calling the Scheduler's initialization function as follows:

```
FunkOS_Scheduler::Init();
```

Creating a Task

After instantiating a task object, it must then have its stack initialized through a call to the Task object's `CreateTask()` method as demonstrated below:

```
MyTask.CreateTask("Hello World!", 3); // Task Name & priority
```

Adding a task to the scheduler

After a task has been created and initialized, it can be added to the scheduler by calling the `Add()` method as follows.

```
FunkOS_Scheduler::Add(&MyTask);
```

To enable the task to run after it has been added to the scheduler's task list, call the task's `Start()` method.

```
MyTask.Start();
```

Starting the scheduler

The scheduler should be invoked from main in an interrupt-disabled context. Once the scheduler is started, program flow will be transferred from main to the application tasks, and never again return to main.

The scheduler's `StartTasks()` method performs the following operations in sequenece, transferring control to the highest priority thread. Note that this function never returns.

1) Start the kernel timer driver
   Before starting the kernel, the timer must be initialized, this is done by calling `KernelTimer::Config()` followed by `KernelTimer::Start()`.

2) Start the kernel software interrupt driver
   Since task switching is dependent on software interrupts, the appropriate software interrupts used on the specified platform must first be initialized as well. This is done with a call to `KernelSWI::Config()` and `KernelSWI::Start()`.

3) Enable interrupts
   Interrupts must be enabled for both the timer and the SWI modules, so a call to globally enable interrupts must be made prior to starting the scheduler. This is done by invoking the `ENABLE_INTERRUPTS()` macro, which is defined for each platform. Care must be taken to ensure that the kernel timer will not cause a context switch before StartTasks() is finished below.

4) Context Switch
   Program control will be switched to the highest priority task.

Suspending a task

A task can be suspended at any point by invoking the Tasks's sleep function:

```
MyTask.Sleep(usTime);
```

Where usTime is the minimum time in system ticks to sleep. If the value `TIME_FOREVER` is set, the task will be permanently suspended.

The Idle Task

Every FunkOS system requires an idle task – a default thread that is executed when none of the tasks are in the ready state. The idle task must have a priority level of 0, and must not perform any operation that could cause a task switch (no drivers, mutex, semaphore, or IPC access). Typically this function is used to set the low-power mode of the CPU until the next timer interrupt. This task is not created automatically by the system, so care must be taken to ensure that this task is implemented. In the event that an idle task is not present, the system may execute another task at random, which may corrupt synchronization objects, etc. The following example demonstrates how the idle task should be implemented:

```
//---------------------------------------------------------------------------
class Idle: public FunkOS_Task
{
public:
      Idle();                       // Required constructor
private:
      WORD m_awStack[128];   // Declare the thread stack
      static void RunMe(void*);    // Declare the thread's entry function
};

//---------------------------------------------------------------------------
Idle::Idle()
{
      m_pfFunc = (&Idle::RunMe);  // Assign the entry function
      m_usStackSize = 128;              // Assign the stack size
      m_pwStack = m_awStack;       // Assign the stack
}

//---------------------------------------------------------------------------
void Idle::RunMe(void *pvThis_)
{
      Idle *pstThis = (Idle*)pvThis_;

      while(1)
      {
            // Do stuff...
      }
}

<Task initialization:>

      IdleTask.CreateTask("Idle", 0);
```

Invoke a task switch

At any time, a task switch can be invoked by calling the static method `FunkOS::Yield()`. This function causes a software interrupt to trigger which suspends the currently operating task and switches to another task. When called from an interrupt, the function will trigger the SWI immediately after exiting the ISR (or immediately if interrupt nesting is enabled).

Lightweight Timer Threads

In addition to regular application threads, another class of threads exist in the system to allow small time-critical functions to run at a predictable, regular frequency. These items are implemented with `FunkOS_Timer` objects, and managed through the global `FunkOS_TimerList` object.

The Lightweight Timer Context

Lightweight threads are simply callback functions that are executed from within the timer interrupt ISR at a regular frequency specified by the user. Depending on the kernel configuration, these tasks may be from within a nested interrupt context, or an interrupt-disabled context.

Creating a Timer Object

A timer object can be created by calling `FunkOS_TimerList::Add` as follows:

```
FunkOS_TimerList::Add(usTicks_, // Ticks between
                      usOffset_, // phase offset in ticks
                      Callback); // callback function pointer
```

If successful, the handle to the timer object will be returned to the user (pointer to `FunkOS_Timer` object). If the operation fails, it is likely due to the size of the timer object table being too small. If this is the case, simply change the number of timer threads in the timer.h file to a value appropriate to your project.

Starting a Timer Object

Once a timer handle has been obtained for the lightweight thread, it can be started by calling the timer object's `Start()` method. The callback function will thus be called every `usTicks_` intervals, offset by a value of `usOffset_` ticks as specified.

The Timer Callback

Once the timer object has been started, the timer callback function will be executed at a regular interval, as specified when the thread was created. Since all timer callback functions are called from the interrupt context, it is wise to keep them as short/small as possible, while using as little stack as possible. If any of these constraints cannot be met from the timer callback, consider enabling interrupt nesting, or re-factoring the offending timer thread as a true task.

Task Synchronization:

Task synchronization is implemented using objects of the semaphore class (`FunkOS_Semaphore`). These objects can be declared in any context, but should be scoped appropriately for the application so that all of the necessary tasks and interrupts have the access they need. In a future revision of the RTOS, tasks may be accessed by name, allowing for further decoupling of tasks from each other.

Initialization

The semaphore is initialized upon instantiation – and can be initialized as a binary semaphore by default, or as a counting semaphore by specifying a maximum count value as a parameter.

```
FunkOS_Semaphore mySem;             // binary sempahore
FunkOS_Semaphore mySem2(10);        // counting semaphore
```

Waiting

A task can wait for a semaphore by calling:

```
MySemaphore.Pend(Time);
```

If the semaphore is unavailable, the task will sleep and wait until the semaphore is available. The time value is the number of system ticks to wait before timing out and continuing execution. When a value of `TIME_FOREVER` is specified, the task will pend on the semaphore forever, until the semaphore becomes available. Multiple tasks can also wait on a single semaphore; in this case the highest-priority waiting task will claim the semaphore and execute. This operation must be called from a task – never from an ISR.

Posting

To post a semaphore call:

```
MySemaphore.Post();
```

This function releases the semaphore, allowing it to be claimed by another task. If other tasks are waiting for the semaphore, the highest-priority waiting task is activated and a task-switch is executed. In the event the call is being made from an ISR, the ISR will complete before the task switch executed.

Resource Protection:

Resource locks are implemented using mutual exclusion semaphore class `(FunkOS_Mutex)`. Protected blocks can be placed around any resource that may only be accessed by one task at a time. If additional tasks attempt to access the protected resource, they will be placed in a wait queue until the resource becomes available. When the resource becomes available, the task with the highest original priority claims the resource and is activated. Priority inheritance is included in the implementation to prevent priority inversion.

Initializing

Initializing a mutex object is performed upon instantiation, and does not require user interaction.

Claiming a Mutex

A Mutex is claimed by calling the mutex's `Claim()` method. If the time value specified is `TIME_FOREVER`, the calling thread will wait forever until the resource is available, returning TRUE when the resource is claimed. If other values are given and the mutex timeout occurs, it will return a FALSE return value, indicating that the mutex was not claimed within the time allowed.

```
MyMutex.Claim(TIME_FOREVER);
```

Releasing a Mutex

Releasing a mutex is performed by calling the object's `Release()` method. If other threads are waiting for the mutex, the highest-priority waiting thread will be given the mutex (with rules for priority inheritence to prevent priority inversions).

```
MyMutex.Release();
```

Resource protection example

```
MyMutex.Claim(Time);
...
<resource protected block>
...
MyMutex.Release();
```

System Watchdog Interface

One very desirable feature of a multi-threaded embedded system design is the ability to verify that every system task is meeting its deadlines, and is operating in a generally timely manner.

Many systems implement a watchdog timer to force a system reset or perform some other action if the timer is not serviced on a regular interval. FunkOS provides a method to connect the watchdog kick event to all of the tasks, requiring that each task meets its deadline before servicing the timer. For development, this can be an especially useful tool, as the watchdog callback can be modified to interface with logging code. The watchdog module can be easily modified to interface with custom error handling code, providing information about all tasks that fail to meet their deadlines back to the user.

Initializing the Watchdog Module

Software watchdog timers can be created as instances of the `FunkOS_Watchdog` class.

The watchdog module maintains a list of watchdog-enabled code sections, and before use, this list must be initialized by calling the static method `FunkOS_Watchdog::Init()`. Care should be taken to ensure that the software watchdogs list is large enough to handle the maximum number of simultaneous watches.

Starting a Watchdog-Enabled Code Block

A task can place a code block in the watchdog monitor list by first calling the `AddTask()` method.

```
BOOL AddTask(USHORT usTime_);
```

where:

`usTime_` is the code block time deadline (in WDT ticks)

Following this, a call to `Start()` must be made to enable monitoring of this section. This operation is described in the following example.

Calls to Add and Remove the watchdog must occur in pairs within the lifespan of the object.

Ending a Watchdog-Enabled Code Block

There are two ways to terminate the watchdog-enabled block. If the code block is something that must be checked on a regular basis, then it can be idled to allow the block to be ignored by the watchdog module until it is reset.

```
void Idle(void);
```

Removing a Watchdog-Enabled Code Block

If the code block is unlikely to be used frequently, it can be removed from the watchdog-enabled task list if desired. This is useful to keep the number of used slots in the watchdog list to a minimum.

```
void RemoveTask(void);
```

Watchdog-Enabled Code Block

A complete example of using watchdog-enabled code blocks is demonstrated below:

```
void MyTask(TASK_STRUCT *pstTask_)
{
    FunkOS_Semaphore clSem;
```

```
        FunkOS_Watchdog clWDTask;

        // Initialize the watchdog list - done ONCE!
        FunkOS_Watchdog::Init();

        // Add block to the watch list - timeout at 1000 ticks
        clWDTask.AddTask(1000);

        while(1)
        {
             // Wait for semaphore
             clSem.Pend(TIME_FOREVER);

             // Start the watchdog-enabled code block
             clWDTask.Start();

             <...monitored code block...>

             // Time-critical block ended, reset/stop timeout
             clWDTask.Idle();
        }
}
```

Event Queues

Embedded systems guru Jack Ganssle once said that without a robust form of interprocess communications (IPC), an RTOS is just a toy. FunkOS implements a couple of different forms of IPC to provide safe and flexible messaging between threads.

Using kernel-managed IPC offers significant benefits over other forms of data sharing (i.e. Global variables) in that it avoids synchronization issues and race conditions common to the practice. Using IPC also enforces a more disciplined coding style that keeps tasks decoupled from one another and minimizes global data – preventing careless and hard-to-debug errors.

Creating event queues

The simple form of IPC provided in FunkOS is the event queue. Any task can have an event queue by declaring an object of type FunkOS_Message, and passing the handle to the object to associated tasks. Event queues are implemented as thread safe circular buffers of a user-defined size. These buffers

are for general event signaling where data can be stored in a simple 16-bit ID + 16-bit Data format.

Using event queues

Before being used, event queues must be initialized by calling `Init()` as follows:

```
Init(astMessages, usSize);
```

Where astMessages is the array of events messages (`EVENT_STRUCT`), and usSize is the size of the circular buffer.

After initialization, an event queue can be written to by calling the `Send()` method:

```
Send(usEventID, usEventData);
```

Where the event ID and event data are 16-bit values that signal an action to the appropriate task. `Send()` also includes a call to Post the event queue's semaphore which can be used to trigger a corresponding event handler in another thread pending on the associated semaphore, as demonstrated below.

```
// Sleep until the message queue has been written
clQueue.m_clSem.Pend(TIME_FOREVER);

// Read through the event queue
while (clQueue.Read(&usEventID, &usEventData))
{
     <Do something>
}
```

Dynamic Memory:

Heap Implementation

The FunkOS heap is defined as a large byte array representing all of the dynamic memory availalble to the system. An array of control structures is maintained in concert with the heap array to define the location, size, and status of the individual blocks within the heap. The size and number of different block types are defined statically, and can be customized as necessary by modifying the

heap header file. All dynamic memory operations in FunkOS are thread-safe.

Customizing The Heap

The size of the heap array, as well as the number of blocks is completely by a group of `#defines` in heap.h. These values can be modified by the user at design-time to optimize the memory allocation in the system, as described below.

From heap.h:

Define the number of block size groups:

`NUM_BLOCK_SIZES`	The number of different block sizes supported

For each block size:

`ELEMENT_SIZE_X`	The size in bytes of each block of group X
`NUM_BLOCK_X`	The number of blocks in group X
`SIZE_BLOCK_X`	(ELEMENT_SIZE_X * NUM_BLOCK_X)

Blocks are ordered smallest to largest by element size

Define the size of the heap and the number of control blocks:

`HEAP_NUM_ELEMENTS`	(NUM_BLOCK_1 + ... NUM_BLOCK_N)
`HEAP_SIZE_BYTES`	(SIZE_BLOCK_1 + ... SIZE_BLOCK_N)

From heap.c:

The `HEAP_INIT_STRUCT` contains an ordered pair of `{NUM_BLOCK_X, ELEMENT_SIZE_X}` for each size group as shown below:

```
HEAP_INIT_STRUCT astHeapInit[] =
{
     {NUM_BLOCK_1, ELEMENT_SIZE_1},
     {NUM_BLOCK_2, ELEMENT_SIZE_2},
     ...
     {NUM_BLOCK_N, ELEMENT_SIZE_N}
};
```

Initializing the Heap

Before using dynamic memory, the user must call the static `Heap::Init()` `method`. This function pre-allocates the memory pool and sets the block size, array location, and status flags for each block of memory.

Allocating Memory

Allocating a block of memory is similar to the standard-C `malloc()` paradigm, implemented as a class with static methods. To allocate a block of memory in FunkOS, the user calls `Heap::Alloc()` as demonstrated below:

```
MY_STRUCT *pstMyStruct;

pstMyStruct = (MY_STRUCT*)Heap::Alloc(sizeof(MY_STRUCT));
```

Assuming a large enough vacant block of memory is available, the pointer to the block is returned, and assigned to the appropriate pointer. When no suitable blocks remain in the pool, `Heap::Alloc()` returns NULL (this condition should be checked on every allocation).

Memory is allocated as single blocks only, and while this limits the size of allocatable objects to the maximum block size, this successfully prevents memory fragmentation which is critical to the stability of real-time systems.

Deallocating Memory

Deallocating memory is similar to the standard-C free() paradigm. A previously allocated block of memory is released by calling `Heap::Free()` as shown:

```
Heap::Free(pstMyStruct);
```

Device Drivers:

The FunkOS drivers API provides a consistent way of implementing and accessing device drivers in a system. This provides a benefit in that it limits the visibility of the drivers to the rest of the application, and vice-versa. As the API is very small (only 4 function calls to learn), driver usage from applications is simple and predictable. Through the API, device I/O access is also treated as Mutually-exclusive, effectivley preventing multiple tasks from accessing a single device at any given time. While using this API does add a small amount of overhead, the benefits to stability and ease of use are highly desirable.

The Device Driver Type

Device drivers are implementations of an abstract driver class type. Driver interfaces are simplified so that only 6 functions are required to be implemented by an inheriting class to create a complete driver implementation. These functions include a constructor, start/stop functions, and a control function. The generic driver module ensures that all I/O operations take place in mutex protected blocks, ensuring that only one task can access a resource at a time, without the risk of priority inversion.

The device driver class is defined below:

```
class FunkOS_Driver
{
public:
     FunkOS_Driver();

     BOOL Start(void);   //!< These are the public interfaces
     BOOL Stop(void);

     USHORT Control(USHORT usID_, void* pvData_);
     USHORT Read(UCHAR *pucData_, USHORT usLen_);
     USHORT Write(UCHAR *pucData_, USHORT usLen_);

private:
     //! methods overridden by inheriting classes, accesses through
     //! the public interface
     virtual BOOL DeviceInit(void)
          {return FALSE;}
     virtual BOOL DeviceStart(void)
          {return FALSE;}
     virtual BOOL DeviceStop(void)
          {return FALSE;}

     virtual USHORT DeviceControl(USHORT usID_, void* pvData_)
          {return 0;}
     virtual USHORT DeviceRead(UCHAR *pucData_, USHORT usLen_)
          {return 0;}
     virtual USHORT DeviceWrite(UCHAR *pucData_, USHORT usLen_
          {return 0;}

     UCHAR *m_szName;
     DRIVER_STATE        m_eState;
     DRIVER_TYPE         m_eType;
```

```
    //--[Resource protection]--------------------------------
    FunkOS_Mutex        m_clMutex;
};
```

Note that the mutex is only provided when support for Mutexes is included in the kernel.

Implementing a Driver

To create a new device driver, first, create a new class inheriting from the base FunkOS_Driver class. For each driver class, you must specify the overridden virtual driver functions. This is demonstrated in the following example.

```
class MyDriver : public FunkOS_Driver
{
public:
    BOOL DeviceInit(void);
    BOOL DeviceStart(void);
    BOOL DeviceStop(void);
     USHORT DeviceControl(USHORT usID_, void *pvData_);
    // Driver Read/Write are unused - using the base class stubs

private:
    // Data internal to this driver
    SOME_DATA_TYPE m_stReport;

    // Function used internally by this driver
    void Read(void);
};
```

Starting/Stopping drivers

To start a driver, you must call the driver's Device`Start()` method.

```
DeviceStart(void);
```

`DeviceStart()` ensures that the driver is valid and initialized before attempting to perform any driver I/O operations..

To stop the driver, call `DeviceStop()` in the same way:

```
DeviceStop();
```

Driver I/O operations

Driver I/O is performed by using the drivers control function, which is accessed through the DeviceControl() function:

```
DeviceControl(USHORT usID_, void *pvData_);
```

Where:

`usID_` is the control event ID

`pvData_` is the input/output data parameter for the control function.

Extreme care must be taken if performing direct driver I/O from outside of the task context. This ability has been put in place to allow driver access from within interrupts, but as a result, care must be taken to ensure that driver operations are not corrupted by access from interrupts.

I/O functions can also be performed using the dedicated Read/Write functions as follows.

For a device write operation, call:

```
USHORT DeviceWrite(UCHAR *pucData_, USHORT usLen_);
```

Where pucData_ is a pointer to a block of data to write, and usLen_ is the size of the data block to write (in bytes). The value returned by this function is the number of bytes that were successfully written to the device as a result of calling the function.

Similarly, for read operations, one would call:

```
USHORT DeviceRead(UCHAR *pucData_, USHORT usLen_);
```

Where pucData_ is a pointer to a block of data to be read, and usLen_ is the size of the data block to read (in bytes). The value returned by this function is the number of bytes that were successfully read from the device as a result of calling the function.

Contact

For any questions or comments, feel free to contact Funkenstein Software using the following:

Email:
funk_dev@hotmail.com

Project homepage:
http://funkos.sourceforge.net/

Contribute!

Funkenstein Software is an looking for developers to help port the kernel to different architectures, contribute demos, write kernel services, or help with documentation. If you like what you see, or if you think you could write better code blindfolded with one-arm-uphill-both-ways, we want to hear from you!