

Introduction To The FunkOS Real-Time Operating System

FunkOS Version R3

March 20, 2010

Mark Slevinsky
Funkenstein
Software Consulting

Table of Contents

Document History:	5
Introduction:	6
About:	6
What's new:	6
Features:	7
Flexible Scheduler:	7
Resource Protection:	7
Synchronization Objects:	7
Timer Callbacks:	7
Driver API:	7
Robust Interprocess Communications:	8
Dynamic Memory:	8
Cooperative Mode Scheduler:	8
Pipsqueak Real-Time Kernel:	8
Software real-time clock:	8
2D Graphics API:	8
Rich GUI framework :	8
Design Goals:	8
Lightweight:	8
Modular:	9
Easily Portable:	9
Easy To Use:	9
Simple to Understand:	9
System Architecture:	9
Source Layout:	9
Conventions:	10
Configuration:	11
Building the kernel:	12
Multiprogramming in FunkOS:	13
Tasks:	13
The Scheduler:	13
Task Struct:	13
Stack:	14
Entry Function:	14
Creating a Task:	14
Adding a task to the scheduler:	15
Starting the scheduler:	15
Suspending a task:	16
The Idle Task:	16

Invoke a task switch.....	17
Lightweight Timer Threads.....	17
The Lightweight Timer Context.....	17
Creating a Timer Object.....	17
Starting a Timer Object.....	17
The Timer Callback.....	17
Task Synchronization:.....	18
Initialization.....	18
Waiting.....	18
Posting.....	18
Resource Protection:.....	19
Initializing.....	19
Resource protection example.....	19
System Watchdog Interface.....	19
Initializing the Watchdog Module.....	20
Starting a Watchdog-Enabled Code Block.....	20
Ending a Watchdog-Enabled Code Block.....	20
Removing a Watchdog-Enabled Code Block.....	20
Watchdog-Enabled Code Block.....	21
Event Queues.....	21
Creating event queues.....	22
Using event queues.....	22
The plumber.....	23
Plumber ports.....	23
Plumber pipes.....	23
Initializing the plumber task.....	23
Adding entries to the plumbing table.....	23
Opening ports.....	24
Waiting for data on ports.....	24
Sending data to tasks.....	24
Dynamic Memory:.....	25
Heap Implementation.....	25
Customizing The Heap.....	25
Initializing the Heap.....	26
Allocating Memory.....	26
Deallocating Memory.....	27
Device Drivers:.....	27
The Device Driver Type.....	27
Implementing a Driver.....	28
Starting/Stopping drivers.....	29
Driver I/O operations.....	29
The Pipsqueak nano-kernel.....	30

The Pipsqueak TASK_STRUCT	31
Initializing the kernel.....	31
Initializing Tasks.....	31
Adding tasks to the scheduler.....	31
Starting the RTOS.....	32
Switching Tasks.....	32
Invoking the Idle Task.....	32
Example.....	32
The Cooperative Scheduler.....	35
Pseudotasks.....	35
Initializing a PTask.....	36
Initialize a Ptask event queue.....	37
Initialize a Ptask list	37
Add a Ptask to the Ptask list.....	37
Run the cooperative scheduler.....	38
Send an event to a task.....	38
Read an event from an event queue.....	38
Task execution in the cooperative scheduler	39
Porting requirements.....	39
Example	39
Porting Overview.....	41
Context Switching.....	41
Task_SaveContext().....	42
Task_RestoreContext().....	42
Critical Sections.....	42
Stack Initialization.....	42
Kernel Triggers.....	43
Software interrupt module – kernelswi.c/.h.....	43
Timer interrupt module – kerneltimer.c/.h.....	43
Watchdog timer module – kernelwdt.d/.h.....	43
Contact.....	44
Contribute!.....	44
Acknowledgments.....	44
License.....	45

Document History:

Rev	Date	Description	Author
A		Initial Release	moslevin
B	11.2009	R2 Updates	moslevin
C	01.2009	R3 Updates	moslevin

Introduction:

About

FunkOS is a Real-Time Operating Systems (RTOS) for small-to-medium scale embedded systems. It is designed to be lightweight, powerful, and flexible, allowing the programmer to easily customize the kernel for a specific platform or application. The system is implemented with a microkernel architecture, where features external to the core are separated in a modular and logically divided way.

This document intends to provide a concise companion guide to the FunkOS kernel and services- allowing a user to get up to speed quickly on writing applications using the RTOS, and spending less time looking through source. It does, however, rely on the reader having a familiarity with using an RTOS, and a thorough understanding of using C in embedded systems. For a more in-depth guide to the API, a complete Doxygen reference of the source code is also included.

What's new:

R3:

Ports:

- AVR port for Imagecraft (ICCV7)
- AVR port for IAR

Kernel:

- New cooperative scheduler capable of running on any platform (with demo)
- Added a fully object-oriented kernel in C++, the **FunkOS++ kernel**
- Added a minimal preemptive kernel – the **Pipsqueak Nano-Kernel**
- Cleaned up the Task API calls to use proper function pointer types
- Semaphores can now be configured as counting semaphores
- Resource protection added to mutex objects

Features:

- Added the Fooey GUI library with a basic widget set

R2: Bacchus

Ports:

- TI MSP430 port is now stable (IAR and CCS)
- AVR XMEGA port added (WinAVR)
- Preliminary ARM Cortex M3 port (Keil, GCC, both untested)

Kernel:

- Simplified the Semaphore, Mutex, and Sleep API's.
- Task delete function now provided.
- Added 4D-Systems uOLED display driver to AVR port
- Added PSX joystick, and simple arcade joystick input drivers to AVR port
- Added generic 2D display driver framework, with support for hardware accelerated or software rendering for common graphics primitives
- Added bitmap font rendering engine, font rendering tool coming soon

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 RTOS 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 and 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
- Write-one/read-many pipes+ports interface (The Plumber)

Dynamic Memory

- Simple fixed block-size heap implementation

In addition to the "main" FunkOS kernel, there are other runtime environments provided for extremely resource constrained devices:

Cooperative Mode Scheduler

- Non-preemptive mode for applications not requiring true multi-tasking
- Will run on any architecture.

Pipsqueak Real-Time Kernel

Small, fast RTOS kernel where size is more important than features

A rich variety of complementary application middleware is also provided along with the kernel to simplify application development and provide

Software real-time clock

- Tick-based software clock with event timestamp functionality

2D Graphics API

- Software-rendering for basic point/line/shape APIs
- Support for bitmapped images and rendering text
- Hardware acceleration interface for device drivers

Rich GUI framework

- Full suite of basic widgets
- Support for keyboard/mouse events
- Fully customizable and expandable

Design Goals:

Lightweight

FunkOS can be configured to have an extremely low static memory footprint. Each task is defined with its own stack, and each task structure can be configured to take as little as 18 bytes of RAM. A typical FunkOS application can

be implemented in under 6K of code space and 1K of RAM, depending on the architecture.

Modular

Each system feature can be enabled or disabled by modifying the kernel configuration header file. Include what you want, and ignore the rest to save code space and RAM.

Easily Portable

FunkOS should be portable to a variety of 8, 16 and 32 bit architectures without MMUs. Porting the OS to a new architecture is relatively straightforward, requiring only device-specific implementations for the lowest-level operations such as context switching and timer setup.

Easy To Use

FunkOS is small by design – which gives it the advantage that it's also easy to develop for. The included demo applications, this manual, and the Doxygen guide provide ample documentation to get you up to speed quickly.

Simple to Understand

Not only is the FunkOS API rigorously documented, but the architecture and naming conventions are intuitive - it's easy to figure out where code lives, and how it works. Individual modules are small due to the “one feature per file” rule used in development. This makes FunkOS an ideal platform for learning about aspects of RTOS design.

System Architecture:

Source Layout

One key aspect of FunkOS is that system features are organized into their own separate source modules. These modules are further grouped together into folders based on the type of features represented:

Root	
config	<- kernel configuration headers
cooperative	<- cooperative scheduler code
demos	<- demo application code
docs	<- documentation (what you're reading now)
doxygen	<- kernel + middleware API reference

doxygen++	<- kernel API reference (FunkOS++)
drivers	<- device drivers
graphics	<- graphics API and GUI framework
kernel	<- RTOS kernel implementation
kernel++	<- RTOS kernel implementation in C++
pipsqueak	<- The FunkOS Pipsqueak RTOS kernel
port	<- architecture-specific code
services	<- non-internal features

Conventions

The following is a list of coding conventions used in FunkOS.

- Each function name is prefixed by the module name and an underscore. In this way, there is no ambiguity between a function and its owner module.
- Each function argument has an underscore suffix to prevent naming ambiguity between variables declared in the function, and parameters passed into the function.
- Macros are named in all caps
- The types used in the kernel are defined as follows:
 - LONG long
 - ULONG unsigned long
 - SHORT short
 - USHORT unsigned short
 - BYTE char
 - UCHAR unsigned char
 - INT int
 - WORD long
 - UWORD unsigned long
 - BOOL unsigned char
- The following prefixes are used in variable naming to identify the type
 - c CHAR/BYTE
 - b BOOL
 - s SHORT
 - l LONG
 - w WORD
 - st any struct type
 - e any enumerated type

- Signedness is further specified in naming with the following prefix:
 - u unsigned
 - Pointer variables are named with the following prefix:
 - p pointer
 - Arrays are declared with the following prefix:
 - a array
 - When multiple variable prefixes are used, the following order is used:
 - [a][p][u][type]
- Variables represent objects, and shall be named as NOUNS
 - Functions are actions, and shall be named as VERBS
 - Struct types are named in all-caps, and must end with `_STRUCT`
 - All module data and functions not required to be directly accessed externally are declared static.
 - Function documentation is placed in the implementation, not the header
 - Tab stops are set at 4 spaces
 - Every function in every module must be documented in doxygen, and must generate no warnings when the documentation is generated

Configuration

Kernel features are enabled by setting a series of `#defines` in **kernelcfg.h**. The current configuration options are shown below:

KERNEL_STACK_DEBUG

- When this feature is enabled, a stack depth monitor is compiled in to the kernel, allowing each task's stack margin to be evaluated at runtime.

KERNEL_USE_SEMAPHORE

- Includes binary semaphore synchronization objects and associated functionality.

KERNEL_USE_MUTEX

- Includes resource-protection locks with built-in priority inheritance.

KERNEL_USE_QUANTUM

- Enables the use of per-task time quantum. This is useful for round-robin

scheduling, allowing each task in a priority level to claim a fixed ratio of the total CPU time.

KERNEL_USE_WDT

- Enables the inclusion of the watchdog timer module. This causes the watchdog timer to trigger based on each task in the system meeting its preconfigured time constraints.

KERNEL_USE_TIMERS

- Includes the timer-triggered lightweight thread manager.

KERNEL_USE_HEAP

- Adds support for simple block-based dynamic memory allocation.

KERNEL_NESTED_TICK

- Allows interrupt nesting from within the system tick interrupt. This creates an extra priority group above regular threads but below other ISRs.

Platform-specific configuration settings are found in the platform's taskport.h file.

Building the kernel

Unlike many RTOS kernels which require the kernel be built into a library, FunkOS is designed to be built directly into your application.

The easiest way to build the kernel into your project is to copy the appropriate source files from the **kernel**, **drivers**, **config**, and the appropriate **port** folder directly into your project source folder. Optional modules such as graphics and middleware services can also be included as required.

Source files containing features that are not required can be removed as necessary to clean up your source directory and eliminate unused code.

Also, an architecture #define macro must be present in your project, or added to types.h. This is used to define types for a specific port

```
#define ARCH_AVR for AVR targets  
#define ARCH_MSP430 for TI MSP430  
#define ARCH_ARM_CM3 for ARM Cortex M3 targets
```

Multiprogramming in FunkOS:

Tasks

In FunkOS, programs are defined by a set of executing threads (Tasks). These tasks operate independently of each other to create the illusion that each task owns its own CPU. Each task is defined by an instance of the `TASK_STRUCT` struct, which is the key data structure used by the kernel to identify and schedule tasks.

Tasks can exist in any of a number of different states, as defined below:

<code>TASK_UNINIT</code>	<- the task has not been initialized properly
<code>TASK_READY</code>	<- available to run (or is currently running)
<code>TASK_BLOCKED</code>	<- blocked, waiting on a semaphore or mutex
<code>TASK_SLEEP</code>	<- task is sleeping for a specified amount of time
<code>TASK_STOPPED</code>	<- task currently disabled
<code>TASK_TIMEOUT</code>	<- task IO operation has timed out

The Scheduler

The scheduler keeps track of all of the tasks in the system. Whenever the scheduler is called (either by timer interrupt or by the application), the system determines the highest priority ready task in the system and switches context to run that task. In the event that there are multiple tasks at the same priority level, the priority group executes those tasks in a pure round-robin fashion, where each task gets an equal share of the CPU time (unless specified by the task quantum).

The scheduler is initialized by calling `Task_Init()`. This must be done prior to any other operation involving the kernel.

Task Struct

A task struct can be declared on its own or as the first element of another struct, allowing for other data to be encapsulated with the task. Casting a pointer of the new type to `TASK_STRUCT*` thus allows the new struct to be used with the kernel directly, as demonstrated below.

```
typedef struct
{
    TASK_STRUCT stTask; // Must be first
    USHORT usArray[10];
} MY_TASK;

MY_TASK stMyTask;
TASK_STRUCT *pstTask = (TASK_STRUCT*) (&stMyTask);
```

Tasks can also be specified as global variables, local to a creating function or task, or privately by declaring them inside of `main()`.

Stack

A stack must also be declared for each task. By default, a task stack is simply defined as an array of 8-bit values aligned on a 16-bit boundary. For other platforms this can be configured as necessary.

Entry Function

Each task also requires an entry function; the function that will be run when the task is executed. Entry functions have the following form:

```
void Entry_Function(TASK_STRUCT *pstTask_);
```

The pointer to the appropriate task struct is passed in as the first (and only) argument to the function. It is important to note that entry functions must never run to completion.

Creating a Task

After declaring a `TASK_STRUCT` object for the particular task in question, it is initialized through a call to `Task_CreateTask()` as demonstrated below.

```
Task_CreateTask(&Task1, // pointer to the task object
               "Hello World!\n", // Name of the task
               (UCHAR*)Stack1, // stack pointer
               TASK_STACK_SIZE, // Size of the stack (in bytes)
               3, // Task priority (1-255)
               (void*)Func1); // Entry function for the task
```

Adding a task to the scheduler

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

```
Task_Add(&Task1);
```

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

```
Task_Start(&Task1);
```

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.

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 `Task_StartTasks()` is finished below.

4) Start the kernel

After the above steps, the kernel can be started by calling `Task_StartTasks()`. Once called, program control will be switched to

the highest priority task.

Suspending a task

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

```
Task_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:

Task creation:

```
Task_CreateTask(&tskIdle,           // pointer to the task struct
                "Idle\n",           // task name
                (UCHAR*)StackIdle,  // pointer to the task stack
                TASK_STACK_SIZE,    // stack size
                0,                   // priority level 0 for idle
                (void*)Idle);        // Idle Function
```

Task entry function:

```
void Idle(TASK_STRUCT *pstTask_)
{
    while(1)
    {
        <System sleep function>
    }
}
```

Invoke a task switch

At any time, a task switch can be invoked by calling `Task_SwitchSWI()`. 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.

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 `Timer_Add()` as follows:

```
TIMER_STRUCT *Timer_Add(usTicks_, // Ticks between calls
                        usOffset_, // phase offset in ticks
                        Callback); // callback function pointer
```

If successful, the handle to the timer object will be returned to the user. 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 `Timer_Start(pstTimer_)` with the handle to the timer object as the argument. 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 binary semaphore objects (`SEMAPHORE_STRUCT`). 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

Before use, the semaphore must be initialized by calling:

```
Semaphore_Init(&Semaphore).
```

Waiting

A task can wait for a semaphore by calling:

```
Semaphore_Pend(&Semaphore, 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:

```
Semaphore_Post(&Semaphore);
```

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 semaphores (MUTEX_STRUCT). 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 by calling:

```
Mutex_Init(&Mutex);
```

Resource protection example

```
Mutex_Claim(&Mutex, Time);  
...  
<resource protected block>  
...  
Mutex_Release(&Mutex);
```

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

The watchdog module maintains a list of watchdog-enabled code sections. Before use, this list must first be initialized by calling `Watchdog_Init()`.

Starting a Watchdog-Enabled Code Block

A task can place a code block in the watchdog monitor list by first calling `Watchdog_AddToList()`:

```
BOOL Watchdog_AddTask(WATCHDOG_TASK *pstTask_, USHORT usTime_);
```

where:

`pstTask_` is the pointer to the code block descriptor object
`usTime_` is the code block time deadline (in WDT ticks)

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

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 Watchdog_IdleTask(WATCHDOG_TASK *pstTask_);
```

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 Watchdog_RemoveTask(WATCHDOG_TASK *pstTask_);
```

Watchdog-Enabled Code Block

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

```
void MyTask(TASK_STRUCT *pstTask_)
{
    SEMAPHORE_STRUCT stSem;
    WATCHDOG_TASK stWDTTask;

    Semaphore_Init(&stSem);

    // Add block to the watch list
    Watchdog_AddTask(&stSem, 1000);

    while(1)
    {
        // Wait for semaphore
        Semaphore_Pend(&stSem);

        // Start the watchdog-enabled code block
        Watchdog_StartTask(&stWDTTask);

        <...monitored code block...>

        // Time-critical block ended, stop monitoring
        Watchdog_IdleTask(&stWDTTask);
    }
}
```

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 `EVENT_QUEUE`, 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 `Event_InitQueue()` as follows:

```
Event_InitQueue(&stQueue, astMessages, usSize);
```

Where `stQueue` is the event queue, `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 `Event_Send()`:

```
Event_Send(&stQueue, usEventID, usEventData);
```

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

```
// Sleep until the message queue has been written
Semaphore_Pend(&queue_semaphore, TIME_FOREVER);
// Read through the event queue
while (Event_Read(&stQueue, &usEventID, &usEventData))
{
    <Do something>
}
```

The plumber

A second, powerful, more flexible (but slower) IPC method is also included to allow larger messages to be sent between tasks. This method uses a combination of buffered **pipes** for data storage along with **ports** to determine the message routing. The plumber itself is implemented as a high-priority task, and uses synchronization and mutex regions to ensure that sending messages is as reliable as possible.

Plumber ports

A plumber port is a 16-bit value representing where a message should be routed. Application tasks wait for data to arrive on a specific port, after which it can process the data appropriately. If multiple tasks pend on the same port, each will receive its own copy of the message data.

Plumber pipes

A plumber pipe is a byte array containing the message data to be sent to the waiting task(s). These pipes are fixed-size byte arrays, which are buffered, and thread-safe.

Initializing the plumber task

The plumber runs as a system task, so before it can be used it must be initialized for use with the RTOS. This is accomplished by calling `Plumber_CreateTask()` prior to starting the scheduler.

Adding entries to the plumbing table

Packet routing information is stored in a data structure called the plumbing table. Each entry in the table contains information about the tasks, associated ports, and the list of pipes currently allocated to the entry. Tasks may add entries to this table by calling `Plumber_AddToList`:

```
PLUMB_RETURN Plumber_AddToList(TASK_STRUCT *pstTask_,
                                USHORT usPortNum_,
                                SEMAPHORE_STRUCT *pstSem_)
```

where:

- `pstTask_` - the pointer to the owner task of the entry
- `usPortnum_` - the plumber port

pstSem_ - semaphore to post once data is available

Opening ports

Ports are closed by default – they must be manually opened by calling `Plumber_OpenPort(usPortNum_)` with the appropriate port number. Entries added to the plumbing table after previously opening a port will require another call to `Plumber_OpenPort()` before they will take effect.

Waiting for data on ports

For a task to wait on a port for data, it must pend on the semaphore specified when configuring the entry in the plumber table. An example of this is shown below.

```
void Task(TASK_STRUCT *pstTask_)
{
    SEMAPHORE_STRUCT stSem;
    PLUMBER_STRUCT *pstPlumber;
    Semaphore_Init(&stSem); // initialize the semaphore
    // Add a listener to port 10 for this task, open the port
    Plumber_AddToList(pstTask_, 10, &stSem);
    Plumber_OpenPort(10);
    while(1)
    {
        Semaphore_Pend(&stSem, TIME_FOREVER); // wait for data
        // Read all of the data available on the port.
        pstPlumber = Plumber_ClaimPipe(pstTask_, 10);
        while (pstPlumber != NULL)
        {
            <... Do Something ...>

            // Release the pipe back to the pool
            Plumber_FreePipe(pstPlumber_);

            // Check to see if there's more data waiting
            pstPlumber = Plumber_ClaimPipe(pstTask_, 10);
        }
    }
}
```

Sending data to tasks

Data can be sent from one task to another (or many others) by writing data to

the plumber using `Plumber_WriteToPort()`:

```
Plumber_WriteToPort(UCHAR *pucData_, UCHAR ucLen_, USHORT usPort_)
```

Where `pucData_` is a pointer to the array of data, `ucLen_` is the length of the message, and `usPort_` is the destination port. All tasks listening on the port will be given a copy of the source data, allocated from the pipe pool.

Dynamic Memory:

Heap Implementation

The FunkOS heap is defined as a large byte array representing all of the dynamic memory available 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:

<code>NUM_BLOCK_SIZES</code>	The number of different block sizes supported
------------------------------	---

For each block size:

<code>ELEMENT_SIZE_X</code>	The size in bytes of each block of group X
<code>NUM_BLOCK_X</code>	The number of blocks in group X
<code>SIZE_BLOCK_X</code>	$(\text{ELEMENT_SIZE_X} * \text{NUM_BLOCK_X})$

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 `Heap_Init()`. 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. 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 6 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, effectively 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 struct type. Driver interfaces are simplified so that only 6 functions are required 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 structure is defined below:

```
typedef struct
{
    UCHAR *szName;           // Driver Name
    DRIVER_STATE eState;     // Driver state
    DRIVER_TYPE eType;       // Driver type enumeration
    DRIVER_CONSTRUCTOR pfConstructor; // Pointer to constructor fn
    DRIVER_START pfDriverStart; // Pointer to driver start fn
    DRIVER_STOP pfDriverStop; // Pointer to driver stop fn
    DRIVER_CONTROL pfControl; // Pointer to driver event fn
    DRIVER_READ pfDriverRead; // I/O Read
    DRIVER_WRITE pfDriverWrite; // I/O Write
    MUTEX_STRUCT stMutex;    // Resource protection mutex
}
```

```
} DRIVER_STRUCT;
```

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 struct with an element of **DRIVER_STRUCT** as the first element. For each object of the new type, you must manually specify the driver functions, driver type, and the driver name in the declaration. This is demonstrated in the following example.

```
typedef struct
{
    DRIVER_STRUCT stDriver;
    USHORT usValue;
} MY_DRIVER;    // New driver type declaration

MY_DRIVER stDriver =
{
    {
        "dev\\MyDevice",
        DRIVER_UNINIT,
        DRIVER_TYPE_MY_DEVICE,
        My_Init,    // These are the unique functions for this driver
        My_Start,
        My_Stop,
        My_Control,
        My_Read,
        My_Write
        // Do not initialize the mutex here...
    },
    0xF00F    // Data unique to this driver type
};

BOOL My_Init(DRIVER_STRUCT *pstDriver_)
{
    <... Driver initialization code goes here...>
}

BOOL My_Start(DRIVER_STRUCT *pstDriver_)
{
    <... Code to Start Driver ...>
}
```

```
BOOL My_Stop(DRIVER_STRUCT *pstDriver_)
{
    <... Code to Stop Driver ...>
}

BOOL My_Control(void *pstThis_, USHORT usID_, void *pvData_)
{
    <... Driver control code goes here...>
}
```

In this way, multiple drivers of the same type can be implemented with details like ports and I/O configuration specified from within the individual object declarations. This also creates a benefit from a code space perspective, where multiple, separately-addressable drivers can share the same function implementations.

Starting/Stopping drivers

To start a driver call `Driver_Start`, casting the custom driver type back to the `DRIVER_STRUCT` type as the first parameter as follows:

```
Driver_Start((DRIVER_STRUCT*)&stDriver);
```

`Driver_Start` ensures that the driver is valid and initialized before attempting to execute the driver start code.

To stop the driver, call `Driver_Stop` in the same way:

```
Driver_Stop((DRIVER_STRUCT*)&stDriver);
```

Driver I/O operations

Driver I/O is performed by using the drivers control function, which is accessed through the `Driver_Control` function:

```
Driver_Control(DRIVER_STRUCT *pstDriver_,
               TASK_STRUCT *pstTask_,
               USHORT usID_,
               void *pvData_);
```

Where:

`pstDriver_` is the pointer to the driver object

`pstTask_` is the pointer to the calling task, or NULL if from an ISR
`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 Driver_Write(DRIVER_STRUCT *pstDriver_, UCHAR *pucData_, USHORT usLen_);
```

Where `pstDriver_` is a pointer to the driver object, `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 Driver_Read(DRIVER_STRUCT *pstDriver_, UCHAR *pucData_, USHORT usLen_);
```

Where `pstDriver_` is a pointer to the driver object, `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.

The Pipsqueak nano-kernel

Even at its smallest, there will be some applications that do not require many of the more advanced features provided by FunkOS, but could still benefit from a basic, time-sharing kernel. In response, we have developed an alternate real-time kernel with a smaller footprint- providing true multi-threading in an extremely small package. This alternate kernel is called the Pipsqueak nano-kernel.

A timer executes at a regular interval, and each task is executed in turn for a pre-defined interval specified in the task's structure. Once the task's time quantum has expired, the next task is executed in a continuous round-robin

fashion. Tasks can also trigger an Idle thread to perform low-priority actions in any spare cycles that a task might have available. A task can also invoke a context switch automatically if desired.

Pipsqueak has advantages in that it has a very small API, yet still includes all of the basic elements of a multithreading, round-robin RTOS. The Pipsqueak API can be broken down as follows:

The Pipsqueak TASK_STRUCT

Each task in the system contains the following elements:

<code>pwStack</code>	-> Pointer to the task's stack
<code>usStackSize</code>	-> Size of the stack (in minimum address units)
<code>usTicks</code>	-> Maximum number of ticks to execute before switching to the next task
<code>pfHandler</code>	-> Task's handler function pointer
<code>pstNext</code>	-> Pointer to the next task in the list (maintained by the kernel itself)

Initializing the kernel

The Pipsqueak kernel is initialized by calling:

```
void Task_Init(void);
```

This function must be called before any other kernel function.

Initializing Tasks

The stack of each task must be initialized prior to being added to the scheduler. This is done by calling:

```
void Task_InitTaskStack(TASK_STRUCT *pstTask_);
```

Note that the task must have all of its stack parameters (size and pointer) assigned prior to calling this function, or the kernel will not run correctly.

Adding tasks to the scheduler

Each task must be added to the scheduler before the kernel is started. To do this, call:

```
void Task_AddTask(TASK_STRUCT *pstTask_)
```

For each task to be added to the scheduler.

Starting the RTOS

To start the scheduler and start executing the defined tasks in round-robin order, call:

```
void Task_StartTasks(void);
```

This function invokes the kernel, and thus program control never returns after calling this function.

Switching Tasks

Tasks can be switched manually by calling the `KernelSWI_Trigger()` function, which causes a software interrupt that in turn executes a context switch. The next task in the scheduler is executed automatically.

Task switching also takes place automatically when the number of kernel timer ticks specified in the `usTicks` member of the `TASK_STRUCT` has expired.

Invoking the Idle Task

A task can also invoke the idle task, giving up the rest of its time quantum, by calling:

```
void Task_GoIdle(void);
```

This is useful for processing low-priority events, and/or invoking a CPU's low-power state when there are free cycles available.

Example

The following example implements a basic system demonstrating the use of each of the Pipsqueak APIs. It is also a convenient template for building applications using this minimal kernel.


```
#include "types.h"
#include "pipsqueak.h"
#include "taskport.h"

//-----
//!!! Define task handler functions for 2 round-robin tasks and an idle task
//-----
void Task1(void);
void Task2(void);
void Idle(void);

//-----
//!!! Define task structures for 2 RR tasks and an idle task
//-----
TASK_STRUCT stTask1;
TASK_STRUCT stTask2;
TASK_STRUCT stIdleTask;

//-----
//!!! Define stack space for 2 round-robin tasks and an idle task
//-----
WORD awStack1[128];
WORD awStack2[128];
WORD awIdleStack[128];

//-----
int main(void)
{
    //-----
    // Initialize the kernel
    Task_Init();

    //-----
    // Initialize the task objects
    stTask1.pfHandler = (TASK_FUNC)Task1;    // Task's entry function
    stTask1.pwStack = awStack1;              // Assign the stack
    stTask1.usTicks = 3;                     // 3 ticks per turn
    stTask1.usStackSize = 128;               // Size of the stack in bytes

    stTask2.pfHandler = (TASK_FUNC)Task2;
    stTask2.pwStack = awStack2;
    stTask2.usTicks = 3;
    stTask2.usStackSize = 128;

    stIdleTask.pfHandler = (TASK_FUNC)Idle;
    stIdleTask.pwStack = awIdleStack;
    stIdleTask.usTicks = 3;
    stIdleTask.usStackSize = 128;

    //-----
    // Initialize the stack for each task
    Task_InitStack(&stTask1);
    Task_InitStack(&stTask2);
    Task_InitStack(&stIdleTask);
}
```

```
//-----
// Add the two round-robin tasks to the scheduler
Task_AddTask(&stTask1);
Task_AddTask(&stTask2);

// Add the idle task to the scheduler
Task_SetIdleTask(&stIdleTask);

// Start the scheduler
Task_StartTasks();

// Program will never get this far...
while(1){};
return 1;
}

//-----
// Task 1: Do something for a while, then go to the next task immediately
//-----
void Task1(void)
{
    int i = 0;
    while(1)
    {
        while (i < 10)
        {
            i++;
        }
        i = 0;
        KernelSWI_Trigger();    // Go to the next task immediately (don't wait)
    }
}

//-----
// Task 2: Do something for a while, then go to sleep for the remaining ticks
//-----
void Task2(void)
{
    int i = 0;
    while(1)
    {
        while (i < 10)
        {
            i++;
        }
        i = 0;

        Task_GoIdle();          // Go to the idle thread
    }
}

//-----
// Idle task: Process any low priority events, put CPU in sleep mode, etc.
```

```
//-----  
void Idle(void)  
{  
    volatile int i = 0;  
    while(1)  
    {  
        i++;  
    }  
}
```

The Cooperative Scheduler

We at Funkenstein Software realize that a fair amount of microcontroller applications either don't need an RTOS, or are running on limited hardware where even a small kernel like FunkOS (or even Pipsqueak) isn't practical.

In response to this large class of real-time applications, we have developed a "cooperative mode" scheduler, which allows a large portion of the features of FunkOS to be used along with non-preemptive tasks. Available FunkOS features in cooperative mode include:

- Device drivers (including graphics drivers)
- Fixed-block-size dynamic memory allocation
- Watchdog handler
- Drawing routines
- Lightweight timer threads (however, a suitable timer)
- Software RTC service (also, requires a suitable timer)

Cooperative mode tasks benefit from reduced memory consumption due to a single-stack execution. This scheduler also does not use any hardware-specific code; as a result, it can be easily compiled for almost any platform.

Pseudotasks

The "threads" in a system using the cooperative scheduler are called Ptasks (pseudotasks), and are implemented as run-to-completion functions. Ptasks are handled primarily by API functions, and the following three datatypes are used to define the structure of a Ptask-based system.

Individual tasks are elements of type `PTASK_STRUCT`, and contains a priority, a pointer to an event queue, and a pointer to a function implementing the task logic.

PTASK_STRUCT elements:

ucPriority	-> The task priority (1 = lowest, 255 = highest)
pstEvents	-> A pointer to the task's event queue
pfHandler	-> A pointer to the task's entry function

Where the function handler type is of the form:

```
void Handler(void *this_, USHORT usEventID_, USHORT usEventData_)
```

Where:

this_	- is the pointer to the PTASK_STRUCT object being referenced
usEventID_	- is the current event ID
usEventData_	- is the current event Data

Since this is a non-preemptive, cooperative scheduler, handler functions must run to completion.

The task event queues (PTASK_EVENT_STRUCT) are implemented as configurable circular buffers that can contain anywhere from 1-255 elements as required.

PTASK_EVENT_STRUCT elements:

pusEventID	-> Pointer to the event queue's event ID buffer
pusEventData	-> Pointer to the event queue's event Data buffer
ucNumEvents	-> Size of the event queue
ucHead	-> Head index of the circular buffer
ucTail	-> Tail index of the circular buffer

The tasks are grouped together into a list of tasks, as defined by a PTASK_LIST_STRUCT object.

PTASK_LIST_STRUCT parameters

ucSize	-> Number of Ptasks managed in the list
ucCurrentTask	-> The index of the current Ptask being executed
pstTasks	-> Pointer to an array of PTasks

Initializing a PTask

```
void Ptask_InitTask(PTASK_STRUCT *pstTask_,  
                    PTASK_EVENT_STRUCT *pstEvent_,  
                    PTASK_FUNCTION pfTask_,
```

```
    UCHAR ucPriority_);
```

Where:

pstTask_ -> pointer to the Ptask object to initialize
pstEvent_ -> pointer to the event queue for the Ptask object
pfTask_ -> pointer to the Ptask handler function
ucPriority_-> priority of the Ptask object

Initialize a Ptask event queue

```
void Ptask_InitEventQueue(PTASK_EVENT_STRUCT *pstEvent_,  
                          USHORT *pusEventID_,  
                          USHORT *pusEventData_,  
                          UCHAR ucSize_);
```

Where:

pstEvent_ -> pointer to the event queue object to initialize
pusEventID_ -> pointer to the event ID array
pusEventData_ -> pointer to the event Data array
ucSize_ -> number of objects in the event queue

Initialize a Ptask list

```
void Ptask_InitTaskList( PTASK_LIST_STRUCT *pstList_,  
                        PTASK_STRUCT **pstTask_,  
                        UCHAR ucSize_);
```

Where:

pstList_ -> pointer to the event list to initialize
pstTask_ -> pointer to the Ptask array associated with this list
ucSize_ -> size of the Ptask array

Add a Ptask to the Ptask list

```
BOOL Ptask_AddTaskToList(PTASK_LIST_STRUCT *pstList_,  
                         PTASK_STRUCT *pstTask_);
```

Where:

pstList_ -> pointer to the Ptask list
pstTask_ -> pointer to the Ptask to add

Run the cooperative scheduler

```
void PTask_Loop(PTASK_LIST_STRUCT *pstList_);
```

Where:

pstList_ -> pointer to the Ptask list containing all the tasks to run

This is the "superloop" which executes the Ptasks in the list. This function returns when all events have been processed.

Send an event to a task

```
BOOL Ptask_QueueEvent( PTASK_STRUCT *pstTask_,  
                       USHORT usEventID_,  
                       USHORT usEventData_);
```

Where:

pstTask_ -> pointer to the task to send the data to
usEventID_ -> event ID
usEventData_ -> event data

Pushes a message into the task's circular buffer. Returns TRUE on success, and if there is no room for the event in the queue, the function returns FALSE.

Read an event from an event queue

```
BOOL Ptask_ReadEvent( PTASK_STRUCT *pstTask_,  
                      USHORT *pusEventID_,  
                      USHORT *pusEventData_);
```

Where:

pstTask_ -> pointer to the task to send the data to
pusEventID_ -> pointer where event ID
usEventData_ -> event data

This function removes the oldest element in the task's event queue, if an event is found. Returns TRUE on success -if no events are currently queued for the task, the function returns FALSE. This function is used primarily by the cooperative scheduler itself, and should otherwise be used with caution.

Task execution in the cooperative scheduler

Like tasks running in the preemptive kernel, PTasks running in cooperative mode are each assigned a priority. The priority determines the order in which tasks will execute when there are multiple Ptasks are capable of running.

Tasks are executed in priority order from high to low, with the highest-priority task with pending events executed first.

Each pending event is removed from the Ptask's queue, and passed as parameters into the task's handler function. After all events for each priority level have been processed, the highest priority level is recalculated, and execution continues. As a result, only tasks with pending events are executed, and any task can have multiple events in its queue.

When there are no more pending events for any task in the system, the task loop exits. This allows for the manager function to place the hardware into a low-power sleep mode until event processing is required again.

Care must be taken to ensure that all Ptasks can meet their deadlines, otherwise low priority tasks may suffer from starvation.

Porting requirements

While the cooperative scheduler will run on virtually any hardware, it is necessary to define two macros to provide resource protection for events. This ensures that events can be queued from interrupts and other tasks without causing race conditions and other artifacts.

`CS_ENTER()` is used to enter a critical section

`CS_EXIT()` is used to exit a critical section

Place these macros in a header file called `protect.h` to ensure proper compilation.

Example

The following is an example of how a system with 3 tasks could be implemented using the cooperative task library:

```
//-----  
// Define three tasks for our system  
static PTASK_STRUCT stTask1;  
static PTASK_STRUCT stTask2;  
static PTASK_STRUCT stTask3;  
  
//-----
```

```
// Define the event queue structures for each of the three tasks
static PTASK_EVENT_STRUCT    stTask1Events;
static PTASK_EVENT_STRUCT    stTask2Events;
static PTASK_EVENT_STRUCT    stTask3Events;

//-----
// Event queues
// Task 1 has a single element buffer
static USHORT usEvent1ID;
static USHORT usEvent1Data;

// Task 2 has a 10 element buffer
static USHORT ausEvent2ID[10];
static USHORT ausEvent2Data[10];

// Task 2 has a 25 element buffer
static USHORT ausEvent3ID[25];
static USHORT ausEvent3Data[25];

//-----
// Define the task list and task list array for our 3 tasks
static PTASK_LIST_STRUCT stTaskList;
static PTASK_STRUCT *apstTasks[3];

//-----
// Task handlers for each PTask
void Task1_Run(PTASK_STRUCT *pstTask_, USHORT usEventID_, USHORT usEventData_);
void Task2_Run(PTASK_STRUCT *pstTask_, USHORT usEventID_, USHORT usEventData_);
void Task3_Run(PTASK_STRUCT *pstTask_, USHORT usEventID_, USHORT usEventData_);

//-----
<... task initialization code ...>
    // initialize our 3 ptasks (Task 1 priority 1, 2 @ 2, 3 @ 3)
    PTask_InitTask(&stTask1, &stTask1Events, (PTASK_FUNCTION)Task1_Run, 1);
    PTask_InitTask(&stTask2, &stTask2Events, (PTASK_FUNCTION)Task2_Run, 2);
    PTask_InitTask(&stTask3, &stTask3Events, (PTASK_FUNCTION)Task3_Run, 3);
```



```
// Initialize each PTask's event queue specifying sizes
PTask_InitEventQueue(&stTask1Events, &usEvent1ID, &usEvent1Data, 1);
PTask_InitEventQueue(&stTask2Events, ausEvent2ID, ausEvent2Data, 10);
PTask_InitEventQueue(&stTask3Events, ausEvent3ID, ausEvent3Data, 25);

// Initialize the global task list
PTask_InitTaskList(&stTaskList, apstTasks, 3);

// Add each task to the list
PTask_AddTaskToList(&stTaskList, &stTask1);
PTask_AddTaskToList(&stTaskList, &stTask2);
PTask_AddTaskToList(&stTaskList, &stTask3);
```

```
<... @ end of main ...>
```

```
while(1)
{
    // Run through all pending events
    Ptask_Loop(&stTaskList);
    // ToDo: Add low power sleep code below.
}
```

Porting Overview

Porting an RTOS to a different architecture can be somewhat of a daunting task the first time around. FunkOS attempts to simplify the process by minimizing the amount of code that is not completely portable, and by organizing code into smaller modules. This section provides an overview of the non-portable modules, and how to go about re-implementing them for a new processor/compiler target.

Context Switching

This is the key to any RTOS – being able to save the state of one task and load the state of another. In simple terms, the context switching function is composed of three parts: The context save, the task switch logic, and the context restore. Therefore, a task switch function looks something like this:

```
{  
    Task_SaveContext();  
    Task_Switch();  
    Task_RestoreContext();  
}
```

Since the task switch is itself portable, only the context save and restore need to be ported to the new architecture.

Task_SaveContext()

Saving the task context involves saving the register state of the currently running task to the task's own stack. This typically involves all of the general-purpose registers, any system status flags, and the stack pointer. Depending on the platform and tools available, this may be written as a macro using inline-assembly, or as an assembly-language function.

Task_RestoreContext()

Restoring a task context is the opposite of saving the context. The newly-selected task's context is pop'd off of the stack task. At the end of the context restore operation, execution of the selected task resumes where it had left off the last time. This is also something that is typically written using inline-assembly or as an assembly-language function.

Critical Sections

Macros also need to be written to implement a critical section. A critical section for FunkOS is an interrupt-disabled context where it can be guaranteed that no task switch or interrupt can possibly occur within the block. In FunkOS, these functions are named `CS_ENTER()` and `CS_EXIT()`.

Stack Initialization

In order to initialize the task properly, it is necessary to initialize each task's stack so that the task's entry function is run when the context is switched to the task for the first time. This is implemented in the `Task_InitStack()` function.

This function consists of the following operations:

- 1) Clearing the task's stack space
- 2) Pushing the initial task state to the stack
 1. Pointer to the entry function
 2. Entry function arguments (pointer to the `TASK_STRUCT` object)
 3. "Fake" status registers and system registers
- 3) Set the new stack pointer

Note that the order of these operations may vary from platform to platform.

Kernel Triggers

A non-portable function is required to start the kernel and scheduler:

```
Task_StartTasks()
```

This function enables the scheduler and restores the context of the last task added to the system, causing the first context-switch of the program's execution.

Once called, execution from main ceases, and program control is given to the tasks.

To trigger a context-switch during program execution a non-portable software interrupt interface must be implemented:

```
Task_YieldSWI()
```

This function is implemented to trigger the software interrupt that in turn runs the RTOS kernel context switch.

Software interrupt module – `kernelswi.c/.h`

This module contains platform-specific functions to configure, enable, and disable the software interrupt used by FunkOS.

Timer interrupt module – `kerneltimer.c/.h`

This module contains platform-specific functions to configure, enable, and disable the timer used by FunkOS to generate the system tick interrupt.

Watchdog timer module – `kernelwdt.d/.h`

This module contains platform-specific functions to configure, enable and disable the watchdog timer used by the FunkOS Watchdog timer module.

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!

Acknowledgments

The author would like to acknowledge the following:

- **Richard Barry** of the FreeRTOS project (freertos.org) for his wonderful document on task switching for the AVR, which served as the initial inspiration for the RTOS and provided a starting point for the AVR port.
- **Matthew Welch** for the White Rabbit console font used in this documentation, available at <http://www.squaregear.net/fonts/>
- **Brian Kent** for the Bandwidth font used for the  logo

License

Copyright (c) 2009

Funkenstein Software Consulting. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the end-user license agreement provided with the distribution.
- Modifications/additions to the source to "port" the software to processor/compiler targets other than those included in the source distribution must be submitted to Funkenstein Software Consulting for inclusion in future revisions of the source.
- Distribution in binary form as firmware included on a hardware product (whether in a modified form or as provided), must include the following attribution statement in at least one official product manual (if provided), whether in printed or digital form:

"Includes the FunkOS Realtime Operating System by Funkenstein Software Consulting, available at <http://funkos.sourceforge.net>"

THIS SOFTWARE IS PROVIDED BY FUNKENSTEIN SOFTWARE CONSULTING ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED. IN NO EVENT SHALL FUNKENSTEIN SOFTWARE CONSULTING BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Revised - November 2009