# Digital Force White Paper

Date: 31st December 2009
(Last Build: December 31, 2009)

## An Ultra Small Real Time Multi Tasking Kernel for Embedded Applications

Mario Trams
Mario.Trams@digital-force.net


Digital Force / Mario Trams
http://www.digital-force.net

# An Ultra Small Real Time Multi Tasking Kernel for Embedded Applications

Mario Trams

Mario.Trams@digital-force.net

**Abstract**

Scope of this paper is the discussion of the basic principles as well as an actual implementation of a tiny real time multi tasking kernel for small embedded applications. Although it does not provide the same functionality as known from "regular" operating system kernels, the presented kernel can be considered as some kind of a micro kernel. Despite the fact that this work does not present fundamentally new concepts, it has been found worthwhile publishing this design in a comprehensive form, hence making it available for the world.

## 1 Introduction

When speaking about small embedded systems I'm referring to midrange–class micro controllers such as PIC–, ATmega–, or 8051–based systems.

Typically, implementing (pseudo–) concurrency in a very simple way is by making use of timer interrupts, or interrupts in general.

When there are some things to be done periodically, you could make use of a simple timer interrupt. For instance, when there are two routines A and B to be executed regularly, you could call A in every even occurrence of the timer interrupt and B in every odd occurrence. This limits the run time of routines A and B to at most one timer interval, which might be difficult to guarantee. Furthermore this approach is not really useful for implementing multiple persistent tasks.

Yet another way is to implement this pseudo–concurrency within a main software loop. But here it is also quite difficult to achieve a reasonable task or function scheduling. Just imagine again two functionalities A and B that are to be embedded into a single endless loop. Sometimes A is blocking because it is waiting for something. So does B. It might be possible to implement such a system, but it will usually result in a complete mess of nasty and mostly unreadable code. Much of the code will take care about the scheduling and the actual functional flow of A and B won't be visible at all. In case of a strong interaction between the involved tasks some kind of coroutine–model might be ok. But this definitively does not work well for multiple and formally independent tasks.

As I will show throughout this paper, it is possible to implement a simple multi tasking mechanism with little effort even for small embedded systems. The presented system is targeted for 8051–based micro controllers and requires a stunningly low amount of less than 600 bytes of code. Furthermore this system has been extensively analyzed regarding its timing behavior and therefore is a good base for fine-grained real time applications.

First of all, this paper will provide a short overview for the basic requirements of the kernel. Next some basic principles and features are discussed, before I describe the actual implementation and various details of it. A discussion of timing aspects is included as well, which is important for real time applications. The paper rounds up with the kernel API description and a small exemplary firmware design making use of the kernel.

Apart from the technical aspects this paper has been written to be somewhat self–containing. That is, the reader does not need to have a larger background in operating system theory — although this would be a plus, of course.

## 2 Basic Requirements

### 2.1 Reduce to the Essentials!

When it is coming to rather small systems having just a few kilobytes of code and data memory it is really crucial to keep unnecessary expenses at an absolute minimum. From a formal point of view, any kind operation system is unnecessary. Nonetheless an operating system is important because it is helping to structure complex systems, hence making the overall system more understandable and less error-prone by reusing well-understood and approved basic function blocks. Usually, operating systems provide a vast amount of functionality. But for a multi tasking operating system there is just one essential function to be provided: Transparently achieve a pseudo-parallel execution of multiple threads or tasks.

Exactly this target is aimed by the work presented here. Nothing more. All other things one might expect from something called "operating system" such as input/output, communication between tasks etc. is left for the actual application. So what is getting outsourced from the system/firmware designers point of view is purely the multi tasking functional-

ity.

### 2.2 Task Model

The task model is to be fairly simple and as usual: Each task has its root function, essentially representing the task. There is no difference when compared to processes or threads on regular operating systems.

What is surely missing is a protection/isolation of different tasks from each other and the kernel itself. That is, a single task can easily disturb or crash other tasks or the whole firmware. This is no artificial restriction. It is coming from the fact that tiny micro controllers do not provide mechanisms such as virtual memory or privilege levels. Even though it would be nice having these features, this lack of protection does not pose a disadvantage when compared to traditional firmware coding.

### 2.3 Preemptive Scheduling Model

Actually, this is a trivial conclusion coming from the requirement for a transparent multi tasking. I.e. the single tasks do not necessarily need to be aware of the fact that there are running other tasks besides them. That is, sooner or later a running task will be temporarily forced to sleep while another task will resume its operation.

As I will show later, a commonly used time slice mechanism has been employed here.

### 2.4 Target Architecture

The kernel has been primarily targeted for the 8051 processor architecture, and in particular the FX2LP from Cypress Semiconductor [1]. Although a few FX2LP-specific functionalities have been used, it should be easily portable to other 8051-based architectures.

Due to efficiency, much of the code has been written in (inline) assembly language. This makes it more difficult to port the kernel to other processor architectures. However, as this kernel is rather small and just consists of a hand full of rather short functions, it should be no big deal to port it to almost arbitrary architectures. The only prerequisite is that these other target architectures provide similar or better stack manipulation mechanisms as the 8051 does.

### 2.5 Programming Language

The preferred language is C. As compiler for this specific FX2LP implementation the free *Small Devices C Compiler* (SDCC [2]) has been chosen. This one is doing a fairly good job. With little modifications it should also be compatible with other compilers such as KEIL.

### 2.6 All–in–one Compilation

Usually, for small embedded applications designed to serve a specific function there is no need for an on-demand program loading. There is just a single persistent firmware that is consisting of the kernel, a couple of application tasks and possibly some additional application-specific stuff such as interrupt routines.

Furthermore it is assumed that there is a fixed number of tasks that is known at compile time. That is, neither a task can terminate in that sense, nor new tasks can be spawned dynamically. Perhaps this is worth to be considered for a future enhancement, as this is a quite useful feature.

This allows to consider the whole firmware as a single project that becomes compiled in one piece. So there is no need for relocatable code.

Another advantage of such a design is that various settings and adjustments can be done at compile time, hence resulting in optimal code size.

## 3 Operational Principle

Roughly spoken, the operation of the system as a whole can be summarized as following:

- Initially, all tasks have to be made known to the kernel by means of registering their root functions.
- The kernel is to be fired up, the tick timer becomes started, and the first task is starting to execute.
- Once a timer interrupt occurs, the currently running task becomes frozen and another one becomes activated.

Basically that's everything. The details are somewhat more tricky, of course.

### 3.1 Context Saving/Restoring

The basic question coming up is: How we can freeze and later on resume a task? For simple designs with interrupt routines, an interrupt routine needs to save all processor registers that it is using throughout its execution, and later on restores these registers immediately before returning back to the regular code.

In that sense, the "regular code" is the one and only task.

When we want to switch between multiple tasks it is not sufficient to just keep the CPU registers of an interrupted task in mind. Instead, the whole state needs to be considered.

The state of a task consists of:

- all relevant CPU registers.

- the stack carrying vital function flow information (return addresses etc.) as well as possibly local variables
- the heap that is used for more or less static variable space allocation

Switching the processor from one task state to another task state is generally referred as *context switch*.

For processor architectures providing virtual memory support such a context switch with regard to the stack and heap can be mostly reduced to an exchange of the page tables. Some processors do even support multiple register banks that eliminate the need to safe most of the CPU registers to a certain extend.

Many of these features are usually not (yet) available in tiny micro controllers. So all of this work needs to be done manually.

What is playing a little bit into our hand is the fact that the whole firmware is to be compiled as a single project. In the particular case of the SDCC all local variables of all functions will be located either on one global heap or on the stack. Effectively, this eliminates the need to exchange the heap when switching between tasks.

For this 8051–based implementation there has been made the restriction that the stack is at most 128 bytes in size and is located in internal RAM from 0x80 – 0xFF. All relevant registers that belong to a task's state will be also put onto the stack. In order to freeze the state of a task the whole stack resp. the used area of the stack needs to saved somewhere. So there are needed 128 bytes of context memory per task.

## 3.2 Scheduling Strategy

The scheduling strategy, i.e. the mechanism to answer the question "Which one of the tasks is to get the CPU next?" has been laid out as a very simple round–robin strategy without any priorities. So the available processing time usually becomes evenly distributed across all tasks. "Usually", because every task has the capability to suspend itself for a certain time. This will be discussed later on.

## 3.3 Tick Timer / Time Slice Length

Another aspect of a multi tasking kernel that is specially important with regard to real time requirements is the length of a time slice. The shorter the time slice, the more often the scheduler is called and the more often different tasks will get some CPU time. A logical drawback from this is that the smaller the time slice the more CPU time will be spent for switching between tasks.

Because the requirements for the time slice length might be different for different applications, its selection has been made adjustable. In the particular case of the initial FX2LP reference implementation the time slice can be adjusted in steps of one millisecond from 1ms to 16ms, 32ms, or 65ms (depending on the selected CPU clock frequency).

## 3.4 Additional Gimmicks

Although the explained goal was an absolute reduced function set of the kernel, there have been added a few useful add–ons.

### 3.4.1 CPU Clock Selection

The Cypress FX2LP can run at 12MHz, 24MHz, or 48MHz. The frequency to be used as regular operating frequency can be selected via `#define` and hence becomes fixed at compile time.

### 3.4.2 Dynamic Frequency Reduction

Energy can be saved when the clock frequency becomes reduced. When all tasks have nothing to do and are sleeping, the clock frequency can be reduced without any impact on overall performance.

For this Cypress FX2LP implementation the clock can be reduced from 48MHz or 24MHz down to 12MHz.

Unfortunately, timers inside the FX2LP (including the timer used for the kernel tick timer) are fed by the CPU clock. As a result, switching between frequencies is associated with minor inaccuracies in the timing. In cases where these inaccuracies are problematic or clock switching is not desired at all, the dynamic clock reduction can be disabled at compile time. This also saves a few bytes of code.

In case the regular operating frequency is already 12MHz, the switching code won't be included automatically.

### 3.4.3 Sleep Functionality

Sometimes it is useful to put a task to sleep for a certain time. So there is a kernel function available that can be called in order to put the calling task to sleep for a given number of kernel ticks or time slices.

### 3.4.4 Suspend Functionality

There is also the capability for a task to suspend itself without a dedicated suspend time. Instead, the task will become rescheduled at the next occasion.

Such a capability is useful for communication among application tasks, or in general when a task needs to wait for a specific event (polling). In such a case it is usually better for the task to release the CPU, hence giving other tasks a chance to get some CPU

time. This is important because one of the other tasks might be responsible for generating the awaited event. Without the capability for a task suspend, the task would poll for the event until the end of the time slice and many CPU time would have been just wasted.

#### 3.4.5 Kernel Tick Counter Retrieval

The kernel is keeping an internal 16 bit tick counter that is incremented with every kernel tick timer interrupt. An application task can fetch the current counter value and evaluate it. This feature can be used for simple time measurements. A typical application is the detection of timeout conditions.

#### 3.4.6 Optional Kernel Semaphore Support

The Kernel is providing support for semaphores as a primitive for inter process communication (IPC). Because this support increases the kernel code size quite a lot (about 90 bytes have been observed), kernel semaphore support can be optionally enabled with a simple `#define`.

#### 3.4.7 Critical Section Support

More than one task accessing dedicated resources is a common scenario. Imagine two tasks that write logging messages into a serial console or access data in one EEPROM.

Dealing with these issues has been addressed by the support of so–called *Mutexes*, an abbreviation for *mutual exclusion*. Mutexes are application–defined. A single mutex is associated with a certain unique resource. Retrieving (request for entering a critical section) or releasing (leave the critical section) a mutex is handled by small macros.

These macros make use of kernel semaphores in case their support has been enabled. When no kernel semaphores are available, other code will be automatically used. Then, semaphores will be handled completely at the application level. While this is functioning identically from a semantic point of view, it can cause some side effects such as formally unnecessary context switches.

#### 3.4.8 Task Synchronization Support

Another common situation is the need for synchronization between two or more tasks. I.e. one task needs to wait until another task reached a certain point of its execution.

There is provided a simple mechanism where a task is blocking its further execution once it reached a certain point. This point is called *Barrier*.

Other tasks or some independent interrupt service routine might clear that barrier so that the waiting task can continue its operation. While passing this cleared barrier, it will become set automatically so that the task will be blocked again when reaching that point in future.

Similarly to mutexes, barriers are supported by macros that make either use of the kernel semaphores or are handled completely at application level.

# 4 Implementation Details

## 4.1 General Structure

A broad schematic view of the kernel is given in figure 1.

There are shown the vital functions (by means of C functions) of the kernel. Actually, these are almost all functions of the kernel.

The figure also shows execution times of various functions. These times apply for the Cypress FX2LP with the following assumptions:

- The CPU clock is running fixed at 48MHz. For 24MHz and 12MHz it is ok to multiply the times by 2 resp. 4.
- There is no clock frequency switching enabled.
- The so–called *stretch–value* for accesses to external memory is set to zero, meaning that no extra instruction cycles are inserted when accessing external memory. Notice that external memory also refers to the 16kB of memory included in the FX2LP!
- Kernel semaphore support has been disabled.

More about the timing will be discussed later in section 5.

As it can be seen, there are two ways how the kernel can be entered. One is a more or less direct way by calling either `sleep()` or `suspend()`. Because of the requirement for a preemptive multi tasking the kernel can be also entered through the tick timer interrupt.

The individual steps when a tick timer interrupt occurs can be roughly summarized as following:

1. Immediately after entering `timer2_interrupt()` vital CPU register contents are put onto the stack.
2. The kernel tick counter becomes incremented.
3. `schedule()` is entered.
4. The next task to be scheduled will be determined using a simple round–robin mechanism.
5. There are three cases:
   (a) It turns out that no task is runnable. In this case the state of the current task will be

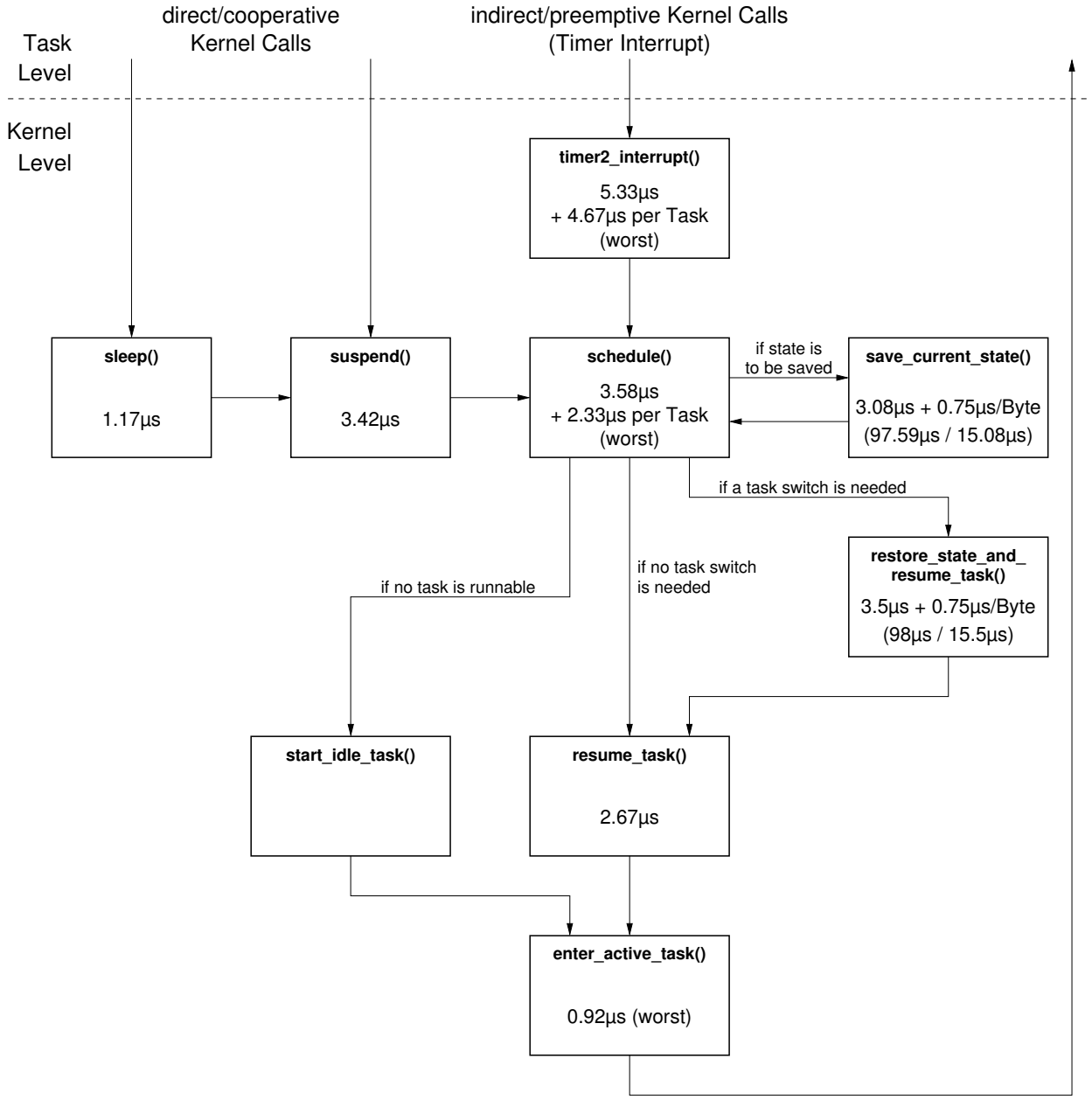


Fig. 1: Basic Structure of the Kernel System
(Times relate to FX2LP at fixed 48MHz and zero stretch for external memory)

saved into its specific context memory by calling `save_current_state()` and `schedule()` is left towards `start_idle_task()`, where a dedicated idle task will be entered.

(b) Another task than the currently active one is to receive some CPU time next. Here the current state will be saved as well and the kernel proceeds with `restore_state_and_resume_task()`. There, the stack will be restored with the information stored in the context memory of the according (next) task before it will be resumed.

(c) The same task that has been interrupted is due to keep the CPU because no other task is runnable. So there are no task context operations necessary and the flow can continue with `resume_task()`.

6. In `resume_task()` the registers will be restored with the values that have been previously saved on the stack. Then the CPU will be effectively handed over to the task. This effectively happens in `enter_active_task()`.

The flow in case of the direct kernel calls via `sleep()` or `suspend()` is more or less self–explaining. As it can be seen, `sleep()` is effectively making use of `suspend()`, while additionally preparing a sleep tick timer for the calling task. Essentially, `suspend()` is putting the current values of all relevant CPU registers onto the stack in the same way as it is done in `timer2_interrupt()`. So from point of view of

`schedule()` and all remaining functional blocks in the flow it does not matter whether the kernel has been cooperatively entered or preemptively via tick timer interrupt.

It should be noted that from a technical point of view all functional blocks that are visible in figure 1 have been implemented as C functions. However, it is obviously mostly pointless to consider them as functions but merely as routines. For instance it is obvious that `schedule()` is not calling `resume_task()`. When `schedule()` is left towards `resume_task()`, the flow will never return back to `schedule()`. An exception is `save_current_state()`.

Most of these functional blocks are treated as C functions with the intention of having a structured code. But they are entered not by calling them but by directly jumping into them. Of course, `sleep()` and `suspend()` as well as all other kernel functions that are externally available have to be called normally.

## 4.2 Task State and Context Memory

It has been already described that the state of a task consists of its stack contents and some CPU registers. What are the registers of interest here?

- The general accumulator `ACC`.
- The program status word `PSW`.
- The so–called data pointer `DPTR` consisting of the two 8 bit halfs `DPL` and `DPH`.
- The special register `B` that is used for some arithmetic operations.
- The eight general purpose registers `R0` — `R7`.
- The (soft) base pointer register `BP`.
- The stack pointer `SP`.
- The current program address of the task represented by `PCL` and `PCH` (not directly accessible as registers).

For a few of these registers there need to be added some more words.

- In the particular case of the Cypress FX2LP there is support for two data pointers `DPTR0` and `DPTR1`. The SDCC is not aware of the second data pointer and does not make implicit use of it. Therefore it is not considered as part of the task state. In case an application is making use of both data pointers, the second data pointer as well as the selection register (`DPS`) need to be handled too.
- The 8051 architecture is supporting four register banks for `R0` — `R7`. The currently active set can be chosen via control bits within `PSW`. The SDCC as of version 2.9.0 does not make implicit use of this feature and does not alter the according bits within `PSW`. The kernel assumes that only the very first register bank is used and accordingly safes only these registers. In case an application is making use of multiple register banks, the kernel needs to be extended by also handling these register banks. Although this is not a big deal, it costs additional time and space.
- In case the SDCC has been advised to generate reentrant code for functions, especially when the compiler option `--stack-auto` is used, the parameter passing for function calls as well as local variables of functions will be allocated on the stack. This is a common mechanism that is not to be discussed here. But for this mechanism there is a so–called *base pointer*, or `BP` required. Because the usage of `BP` within an application is very likely, this register (RAM location, actually) will be saved as well.
- **Important:** Depending on certain 8051 derivatives and application requirements there might be also other registers that need to be saved (for instance auto pointer registers etc.). **That should be checked in detail then!** If additional registers are used, their save/restore handling needs to be added to the according kernel routines.

Due to reasons of simplicity, all registers are saved onto the stack. So after saving them, the stack can be considered as a complete representation of the according task state.

Because the maximum stack size is assumed to be 128 bytes, there need to be reserved 128 bytes per task. Figure 2 illustrates the structure of such a task state data set.

Figure 2 should be mostly self–explaining. Just before the task is calling the kernel or a tick timer interrupt is to occur, the stack pointer is at a position `T_SP`[1]. Then the current program counter will be automatically put onto the stack (`PCL` and `PCH`). Finally all remaining registers are saved by the kernel onto the stack.

When the kernel stores the current stack contents into the context memory, it just copies the relevant (used) data — i.e. from the beginning of the stack up to including offset `T_SP + 16`. Copying only relevant data saves plenty of time, as the stack is usually used just fractionally and there's a lot of unused space.

The very last byte of the task state structure is used to store the stack pointer itself. Of course, due to the dynamic nature of the stack the stack pointer itself needs to be stored at a fixed location. Although it could be also stored at a separate location, the last task state structure offset is a good place because in this implementation of the kernel the last two stack bytes cannot be used due to technical reasons.

So overall the maximum stack size is 126 bytes. For the application tasks it can be stated that they can fill the stack with up to $126 - 16 = 110$ bytes.

[1] Actually `T_SP + 0x80`, because the stack is starting at address `0x80`.

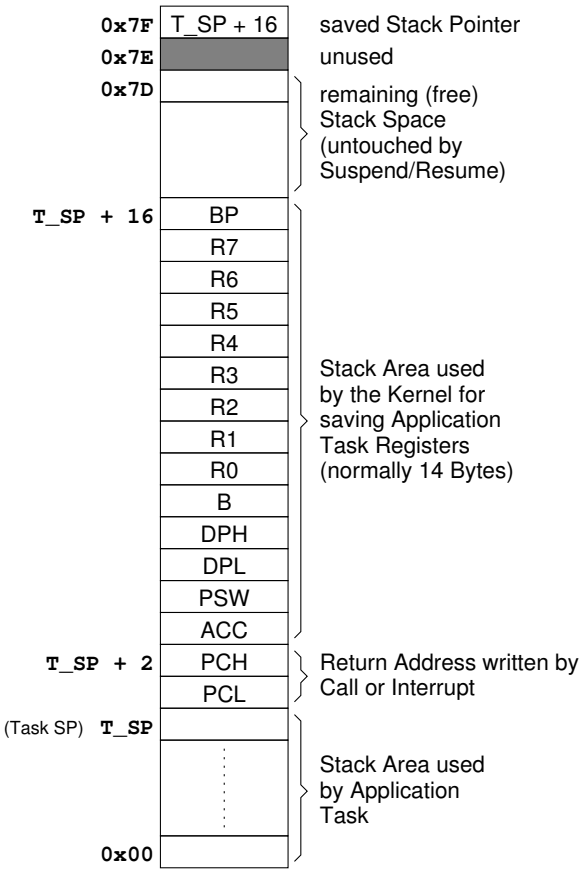


Fig. 2: Structure of a frozen Task State

**Note:** In case the application firmware is making use of other interrupts apart from the tick timer interrupt of the kernel, there is less stack space available for the task — depending on the needs of the interrupt service routine(s).

**Note:** The SDCC has the capability of dealing with a second "soft" stack allocated in external memory that is used for function parameter passing and local variables of functions. This feature can be enabled via the `--xstack` compiler option (refer to according SDCC documentation [2]). The kernel in its current version **does not support** this external stack. Consequently, making use of this feature will result in a crashing or unpredictable firmware behavior!

The context memory itself is located in external memory. Per default in case of the Cypress FX2LP it has been placed at the very end of the integrated external memory. The definition of the context memory address has been made in a dynamic way so that it becomes adjusted automatically depending on the number of tasks. As there are needed 128 bytes per task, for an exemplary task count of 4 the total context memory is 512 bytes in size and extends from `0x3D00` — `0x3FFF`. This needs to be taken into account when the firmware code is making explicit use of external memory or is having the compiler to allocate memory in external memory. Especially with respect to the latter one, the `--xram-loc` and `--xram-size` compiler options need to be set accordingly so that the compiler does not place other data across the context memory.

## 4.3 Scheduling Details

As it has been mentioned earlier, a plain and simple round–robin scheduling scheme is used. This is achieving a fair and easily predictable distribution of the CPU time across all tasks. Technically this has been implemented by a simple method of walking through the task indices. The next task that is runnable will be scheduled then. I.e. when currently task with index $N$ has been interrupted or suspended, the scheduler is checking for task $N+1$, then $N+2$, and so on, before starting over with index 0 again. So it might be well possible that the same task $N$ will be scheduled in case there is no other task runnable. This method is just fine for a small number of tasks this kernel will be very likely faced with.

In case there is no task runnable at all the idle task will become activated. This is the case when it turns out that even the task that just has been suspended is not runnable.

Although this mechanism is very simple, there are some technical pitfalls requiring additional and not so obvious precautions to preserve a flawless and fair scheduling.

Because the kernel is not reentrant, the timer interrupt needs to be disabled when the kernel is entered via either `sleep()` or `suspend()`. I.e. interrupting the kernel code by itself must be avoided.

Such a temporary tick timer interrupt disabling will also induce a small jitter into the regular scheduling interval, but that's not the point.

What can happen in such a scenario is best explained by having a look at a small example. This example is illustrated by figure 3.

The following things happen in this example:

- At a certain point in time task 1 suspends itself.
- While the kernel is active with processing the suspend request the next tick timer interrupt occurs.
- Because the tick timer interrupt has been disabled, it does not become effective yet.
- The kernel normally proceeds with its activities and is about to activate task 2 as next.
- Just before task 2 is about to continue its operation, the tick timer interrupt becomes re–enabled.
- Because there is still a tick timer interrupt pending, the kernel is re–entered immediately through the interrupt without having executed a single instruction of task 2.
- The kernel (in wake of having activated task 2 last time by) proceeds normally, preemptively sus-

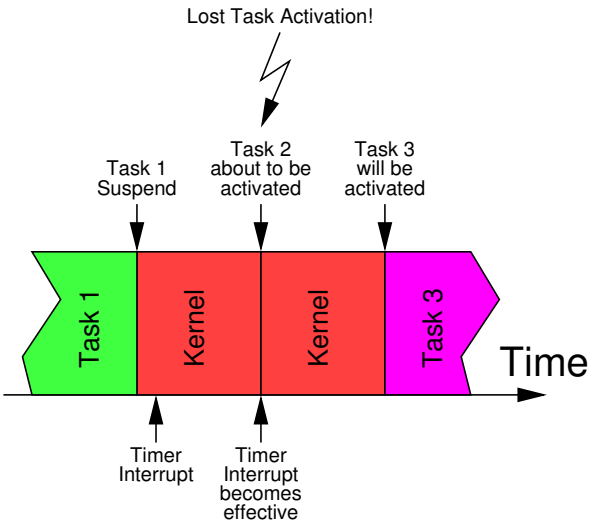


Fig. 3: An example of a lost task activation scenario

pends task 2 again, and activates task 3.

So the activation of task 2 has been effectively lost in that case.

Although this might appear uncritical because task 2 very likely receives the CPU next time by, it poses a violation for real time requirements. And even more worse, it can lead to a complete starvation of a single task so that it never becomes activated. Such a case can be constructed easily. Imagine two tasks, where one task "accidently" suspends itself always just a moment before the tick timer interrupt occurs. Here the other task will never receive a single instruction cycle of CPU time. Such a scenario must be avoided by design.

A simple method has been integrated to cancel these situations. That is, whenever the kernel is entered via `suspend()` (and hence also via `sleep()`) the scheduler becomes disabled for the next forced kernel activation through a tick timer interrupt.

The drawback of this solution is that a task might be active for almost up to two consecutive time slices. Though, this does not affect worst case reaction times for the overall design. But most importantly, the scenario that a task can starve without having a chance to do anything against it has been eliminated.

## 4.4 Semaphore Support

Semaphores represent an important primitive that is to be used for implementing higher–level synchronization mechanisms such as mutexes and barriers (see below).

The kernel in its initial revision presented herein is providing support for simple binary semaphores. I'm not going to describe all the formal details of a semaphore here. If interested, refer to general operating systems literature.

The semantics behind a binary semaphore is quite simple: At any given time, only one process can "own" this semaphore. Or in other words: A semaphore can be "acquired" exactly once at a time.

When it's coming down to the processor instruction level, acquiring a semaphore needs to be carried out as an atomic operation. Otherwise it might occur that more than one task can have one and the same semaphore at a time. This would result in troubles, of course.

There are two supported semaphores variants: Application level or kernel supported.

For the application level variant, the mechanism employed for acquiring a semaphore has been implemented as a simple spin–lock. This is implemented as a macro as shown in listing 1.

```
1 #define ACQUIRE_SEMAPHORE(SEMAPHORE) \
2   DISABLE_TICK_TIMER_INTERRUPT; \
3   while (SEMAPHORE == SEMAPHORE_OCCUPIED) { \
4     suspend(); \
5     DISABLE_TICK_TIMER_INTERRUPT; \
6   }; \
7   SEMAPHORE = SEMAPHORE_OCCUPIED; \
8   ENABLE_TICK_TIMER_INTERRUPT;
```

Listing 1: Semaphore Acquire Macro (Application Level)

As it can be seen, there is made a check whether the semaphore to be acquired is already occupied by somebody else or not. When it is in use currently, the task will suspend itself. Doing so is very sensible, as the semaphore needs to be released by another task. When the task that wants to acquire the semaphore does not suspend itself, it is not giving another task the chance to release the semaphore for the remainder of the current time slice. Hence CPU time is just wasted.

Once the semaphore appears to be not occupied any more, the task is marking it "occupied", and hence has got it now.

To make this process atomic, the kernel tick timer interrupt becomes disabled temporarily. Disabling and re–enabling the tick timer interrupt is done with internal macros here. It should be noted that it is needed to disable the tick timer interrupt after `suspend()` has been called. This is important, because the tick timer interrupt will be inherently enabled by the kernel after a task resume operation.

The disadvantage of such an application level semaphore handling is that it essentially implies an active polling by the task itself. That is, eventually the task needs to be activated over and over again until the semaphore is free. This involves lots of context switches which are quite time consuming.

To overcome this situation there has been integrated an optional mechanism for semaphore support through the kernel. In this case there will be called a kernel function `acquire_semaphore()` when a semaphore is to be acquired. In case the specified semaphore is not occupied, the calling task won't be suspended and the function is returning immediately. Otherwise the kernel internally marks that the calling task is waiting for the given semaphore to be free. Within the upcoming subsequent scheduling decisions (inside `schedule()`) there is made an additional check whether a task is waiting for a semaphore and whether this semaphore is free. Only when the semaphore is free it will be marked "occupied" again and the according task will resume. Else the task will be kept in suspend.

When a semaphore is to be released, this can be easily done by changing the value of the according variable appropriately. There are no special precautions needed. So this is just a simple assignment.

The kernel semaphore support has been made optional because it is increasing the kernel code size by around 90 bytes (checked with SDCC 2.9.0). However, it should be noted that each time a macro is instantiated for acquiring a semaphore within the application level this costs 19 bytes of code, whereas calling the kernel costs just 10 bytes (including parameter passing and register saving). So when there are more than 10 occasions in the application where a semaphore is to be acquired, it is generally better to make use of the kernel semaphores from this point of view.

Application level semaphores have been based on single bits by making use of the bit–addressable memory range feature of the 8051. Kernel semaphores, however, are based on whole bytes located within the internal RAM of the 8051. Although a single bit would be just fine, the 8051 unfortunately does not support indirect addressing of the bit–addressable memory range. Furthermore, kernel level semaphore support requires additional space of internal RAM (one byte per task) that is needed to store the information for which semaphore a task is waiting for.

**Note:** There is a potential risk that tasks starve under certain conditions. I.e. they will never ultimately get the semaphore. Although this is a very unlikely scenario, this might be addressed in a future release of the kernel that is implementing a "first come, first serve" policy for semaphores.

## 4.5 Synchronization Support Macros

Building upon the semaphores (either application or kernel level) there are provided a few macros that are intended for some basic support for what is known from regular operating systems as *inter process communication*, or *IPC*. These macros do not provide any real IPC mechanism, but they provide the basic synchronization/handshake primitives needed to implement them.

These are two things: Mutexes and barriers. They are shortly described in the following subsections.

### 4.5.1 Critical Section (Mutex)

Whenever multiple tasks make use of some common resource, their accesses need to be interlocked from each other. Simple examples for such a resource are memory locations used to exchange data between tasks, or an EEPROM controller that is used by multiple tasks for storing some data into a common external EEPROM.

Figure 4 illustrates a simple example for two tasks sharing a critical section.

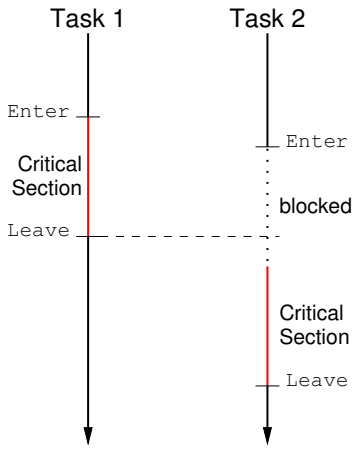


Fig. 4: An example for a Critical Section

When task 1 wants to enter the critical section this request can be permitted immediately. When task 2 wants to enter the critical section this cannot be permitted instantly because task 1 is still within the critical section. Hence task 2 needs to take care about this and cannot enter the critical section. Instead it needs to wait until task 1 is leaving the critical section (at earliest).

The implementation of the macros to be used for entering and leaving a critical section are shown in listing 2.

```
1 #define ENTER_CRITICAL_SECTION(MUTEX) \
2   ACQUIRE_SEMAPHORE(MUTEX)
3 #define LEAVE_CRITICAL_SECTION(MUTEX) \
4   RELEASE_SEMAPHORE(MUTEX)
```

Listing 2: Critical Section Macros

As it can be seen, the mutex handling macros just make proper use of a semaphore. It is required that

the mutex resp. semaphore is marked to be free initially. This is automatically achieved by using a provided macro for the definition of the mutex.

**Important:** Care must be taken to not make use of critical sections from within an interrupt service routine!

### 4.5.2 Barrier

Another useful synchronization primitive is a barrier. A barrier can be used to instruct a task to not pass a certain point of execution until there is not given a "go" from another task. When the task passed the cleared barrier, the barrier becomes set automatically.

Notice that a barrier in such explicit sense is not common for regular operating systems. It is merely indirectly present there.

A simple example illustrating the semantics is given by figure 5.

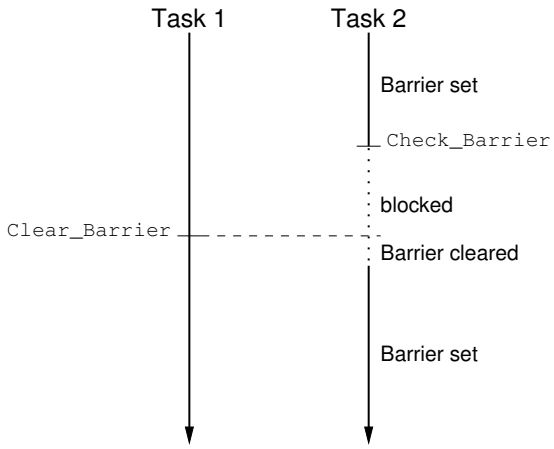


Fig. 5: Barrier Illustration

A semaphore as basic element suits such a barrier very well. That is, the function for acquiring a semaphore serves as the function for checking whether a barrier can be passed or not. The function for releasing a semaphore serves as function for clearing the barrier. Consequently, the according macro definitions are looking as shown in listing 3.

```
1 #define WAIT_FOR_BARRIER(BARRIER) \
2   ACQUIRE_SEMAPHORE(BARRIER)
3 #define CLEAR_BARRIER(BARRIER) \
4   RELEASE_SEMAPHORE(BARRIER)
```

Listing 3: Barrier Macros

In contrast to a mutex, the semaphore for a barrier needs to be marked "occupied" initially. This will be done automatically during the definition of a barrier.

Notice that `CLEAR_BARRIER()` is the only synchronization macro that is allowed to be used from within an interrupt service routine. This makes it possible to keep a regular application thread sleeping until it is getting awakened indirectly through an interrupt service routine.

## 4.6 Foreign Interrupts

The kernel itself is making use of timer 2 and its associated interrupt as tick timer. Of course, most firmware designs will need their own interrupt service routines for special purposes.

Formally, such "foreign" interrupt service routines can coexist besides the kernel. There is just one important requirement: An interrupt service routine must never get interrupted by the kernel resp. its tick timer interrupt service routine. Vice versa there is no problem.

Bad things happen whenever the kernel interrupts another interrupt service routine. This is, because an interrupt routine will always run within the context of the task that has been interrupted by it. When this routine becomes interrupted by the kernel and the kernel carries out a context switch, this partly executed interrupt will be suspended as well. Besides the fact that this delays the completion of this interrupt routine, which is usually not wanted, this causes some other confusions as well.

A simple solution for this issue is to give the kernel tick timer interrupt the lowest priority among all available interrupts.

The FX2LP implements a default (natural) prioritization of all interrupt sources, where the priority of the kernel tick timer interrupt is 6 (0 is the highest and 12 the lowest priority). So there are 6 more interrupt sources (interrupts 7–12) that still have an higher priority. The FX2LP does also implement a high and a low priority group for interrupts. Within these groups, the natural prioritization applies. Per default all interrupts are assigned to the low priority group. So all that needs to be done is to put the 6 interrupts with a naturally lower priority into the high priority group. Then the kernel tick timer interrupt has the lowest priority and it is impossible that it can interrupt another interrupt service routine.

Exactly this method is employed by the kernel.

**Note:** Placing interrupts 7–12 into the high priority group makes them automatically higher prioritized than interrupts 0–4. In case this is not desired, the application might also place interrupts 0–4 into the high priority group. This would match the default (natural) overall prioritization except that the kernel tick timer interrupt has the lowest priority.

**Note:** Applications that have other (incompatible) requirements for the interrupt prioritization cannot

be directly used. Other mechanisms need to be implemented here. I.e. context switches need to be avoided somehow in the critical cases.

# 5 Timing Aspects

Going back to figure 1 on page 5, we can derive some information about the impact of the kernel with regard to the processing overhead caused by it as well as the real time capabilities of a firmware system.

Again, notice that the times shown in figure 1 are based on a Cypress FX2LP running at 48MHz. In case of a 24MHz or 12MHz clock the times can be simply multiplied by two resp. four.

In the specific case of the FX2LP, accesses to external data memory ("external" by means of 8051 terminology) can be stretched by up to 7 additional instruction cycles. As the context memory is due to reside in external memory, this can have a major impact especially on the `save_current_state()` and `restore_state_and_resume_task()` functional blocks. Here we assume a stretch value of zero. For other stretch values the timings need to be reconsidered, which is not very difficult.

Yet another minor impact on the timing is the fact whether the dynamic clock frequency reduction feature is used or not. Using this feature is adding some additional time for the needed code as well as the fact that a few instructions in `timer2_interrupt()` will be executed at the low frequency before the clock is set to its regular value. For the following considerations we assume that the clock frequency remains constant at 48MHz.

## 5.1 Kernel Overhead

Under "normal" conditions we can assume that there are a couple of tasks running pseudo–concurrently. Every time a tick timer interrupt occurs a complete context switch is to be carried out. The question is, how much of the expensive CPU time is consumed by the kernel and hence is not available for the actual application.

It is rather easy to calculate the time that will be spent within the kernel in worst case. In some cases the time depends on the number of tasks. For an exemplary typical system we can assume a value of 4 here, for instance. The resulting worst case time in micro seconds is $5.33 + 4 \cdot 4.67 + 3.58 + 4 \cdot 2.33 + 97.59 + 98 + 2.67 + 0.92 = 236.09$. Or in general:

$$kerneltime_{max} = 208.09\mu s + \text{TASK\_COUNT} \cdot 7\mu s$$

So under worst conditions one has to assume that approx. $236\mu s$ or 0.236ms will be spent inside the kernel for each tick timer interrupt in this example ($kerneltime_{max} = 0.236$ms).

The two major contributors to this time are the routines for saving and restoring the task states, each having a worst case run time of almost $100\mu s$. The run times of these routines heavily depends on the number of stack bytes that are to be handled. The minimum number of stack bytes is induced by the kernel mechanism and equals 16 (2 for the return address into the task and 14 for saved register values; refer also back to figure 2 on page 7).

In a typical system the stack won't be fully utilized. Though this depends on the application, of course. A stack usage of say 50 bytes seems to be quite a good value for a small task consisting of some levels of function calls and perhaps a few stack–based local variables. So it should be safe to estimate the typical kernel overhead at a value of say $150\mu s$ or 0.15ms ($kerneltime_{typ} = 0.15$ms) per tick timer interrupt.

Based on these values the kernel overhead expressed in percent can be easily expressed as a function of the time slice that has been chosen:

$$kerneloverhead[\%] = \frac{kerneltime \cdot 100[\%]}{timeslice}$$

So with a time slice of 1ms (1kHz tick timer rate) the kernel overhead is with 23.6% (worst case) resp. 15% (typical) rather high. For a time slice of 10ms (100Hz tick timer rate) it is dropping to a rather irrelevant 2.36% resp. 1.5%.

## 5.2 Real Time Considerations

Answering the question "When a certain task will be scheduled next in worst case?" is rather easy.

In the most simple behavior the task scheduling will be on a solely preemptive basis. That is, no task will suspend itself and it is just the tick timer interrupt that dictates when a task will become suspended and another one becomes resumed. Such an exemplary scenario is illustrated in figure 6 assuming a number of four tasks.

So it can be stated that any task is receiving CPU time every $timeslice \cdot N$ units, where $N$ is the number of tasks. Furthermore, the CPU time every task receives within a period of $timeslice \cdot N$ equals $timeslice - kerneltime$. Assuming an exemplary system with 4 tasks and a time slice of 10ms, the following characteristics can be derived:

- Every task receives CPU time every 40ms.
- Every task will be scheduled 25 times per second.
- For each awake–period every task can get the CPU for at least 9.764ms, or for 9.85ms typically.
- Within one second, every task receives a total CPU time of at least approx. 244ms, or approx. 246ms typically.

Under conditions when tasks make use of the suspend

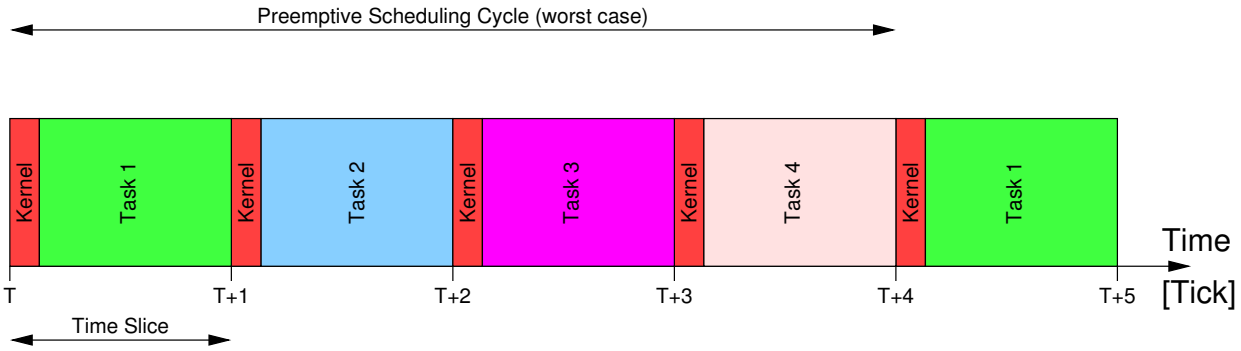


Fig. 6: A trivial task scheduling example with 4 tasks

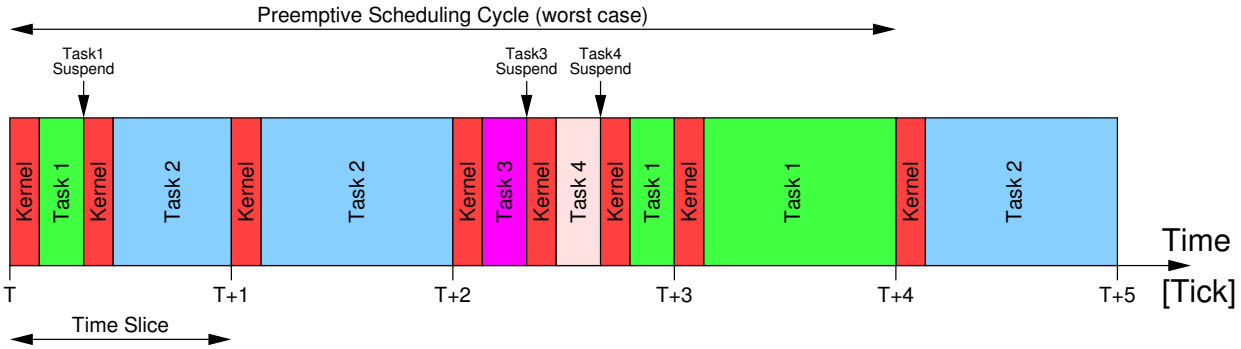


Fig. 7: A task scheduling example with 4 tasks including suspends

functionality, things are looking a little bit more complicated as illustrated exemplary in figure 7.

What is optically visible from figure 7 is that the CPU time each task receives over a given time span might be not equal from task to task and can significantly differ from each other. So, as shown in the example, across a period of four time slices, tasks 1 and 2 receive much more CPU time than tasks 3 and 4.

Whenever a tasks is suspending itself it does agree to "give away" the remaining time of the current time slice. Taking the worst case conditions into account, this task cannot expect to get back the CPU prior to a time of $timeslice \cdot N$ units (where $N$ is again the number of tasks). In the particular example shown in figure 7 task 1 is rescheduled earlier within less than $timeslice \cdot 3$ units. However, this is just because tasks 3 and 4 were so "glad" and have suspended theirself.

Notice that in the example shown in figure 7 the kernel does not switch to task 3 at tick $T + 1$ and keeps task 2 active. Similarly task 1 is kept active at tick $T + 3$ instead of switching to task 2. The reason for this behavior has been already explained back in section 4.3.

# 6 Kernel API and Sources

## 6.1 Kernel API Description

The following text gives a short description for all the relevant kernel functions that are to be (or can be) used by the application firmware.

These are just 5 resp. 6 functions in total.

### 6.1.1 `initialize_task_context()`

Synopsis:

```
void initialize_task_context(
  unsigned char task_id,
  unsigned char initial_sleep,
  void *task_pointer
)
```

`initialize_task_context()` is to be called for each task that is to be registered within the kernel. This function needs to be called for each task one after each other. There are three parameters:

- `task_id` specifies the unique identifier of the task starting at 1. The largest allowable value is specified by the global constant `TASK_COUNT` (see `kernel_setup.h` later). Notice that for simplicity there is not made any boundary check for this parameter! Specifying values larger than

TASK_COUNT will silently fail with unpredictable system crashes.

- initial_sleep specifies the number of time slices that will be (at least) waited before the task becomes started. If this feature is not needed a value of 0 is just fine.
- task_pointer is a simple pointer to the root function of the according task.

Overall, initialize_task_context() is preparing the context memory of the given task properly. When the task becomes activated for the first time the function that has been specified through the task_pointer argument will start executing.

**Note:** All tasks defined by TASK_COUNT need to be registered! I.e. when TASK_COUNT is 4, there need to be registered tasks with ids 1, 2, 3, and 4. Not less, not more, and no other ids. In case one task id is missing, this task will start up with an uninitialized context memory entry which results in an unpredictable behavior. Although it would not be difficult to add according error handling into the kernel, it has been not done for the sake of a small code footprint.

#### 6.1.2 start_kernel()

Synopsis:

```
void start_kernel(void)
```

This function first initializes some data structures required by the kernel and is setting up the CPU clock frequency to the value that has been defined in the header file containing various fixed kernel settings (also refer to the example application that is shown later in this document).

After the tick timer interrupt has been set up it proceeds directly with the scheduler and the kernel essentially comes to life.

start_kernel() has no arguments and it will never return. Therefore calling start_kernel() is the last operation to be done within the application firmware startup code. Code that is following start_kernel() won't ever become executed!

**Note:** Prior to calling start_kernel() the firmware needs to enable interrupts globally. The kernel itself will only enable its own timer interrupt, which is timer 2 in that case.

**Note:** Before calling start_kernel() all tasks need to be registered via initialize_task_context().

#### 6.1.3 suspend()

Synopsis:

```
void suspend(void)
```

By calling suspend() a task can put down its activities before this will be done preemptively at the end of the current time slice. Hence a task can give other tasks a chance to do some work instead of just burning CPU time. Depending on the situation, the task might not actually become suspended. This is the case when no other task is runnable momentarily.

In a worst case scenario and when all other tasks are busy, the time to wakeup can be specified as:

$$suspendtime_{max} = timeslice \cdot \texttt{TASK\_COUNT}$$

#### 6.1.4 sleep()

Synopsis:

```
void sleep(unsigned char ticks)
```

sleep() can be used by tasks to self–suspend them for the given number of kernel ticks. So the absolute sleep time depends on the selected kernel tick time resp. time slice.

A value of zero for ticks has no effect resp. has the same effect as a call of suspend().

As the 8 bit argument restricts the maximum number of sleep ticks to 255, longer periods can be achieved by calling sleep() multiple times within a loop or so.

Notice that the sleep time is associated with a certain jitter. Under worst case conditions the actual minimum and maximum sleep times are as follows:

$$sleeptime_{min} = timeslice \cdot (ticks - 1)$$
$$sleeptime_{max} = timeslice \cdot (ticks + \texttt{TASK\_COUNT} - 1)$$

#### 6.1.5 get_kernel_tick()

Synopsis:

```
int get_kernel_tick(void)
```

This is just a helper function that needs to be used for retrieving the so–called kernel tick counter. Calling this function does not enter the kernel itself.

The kernel tick counter is a simple 16 bit integer that increments with every tick timer interrupt. When passing the value of 65535 it reverts back to 0.

Evaluating this counter is useful for implementing timeout detection mechanisms in software.

#### 6.1.6 acquire_semaphore()

Synopsis:

```
void acquire_semaphore(
  idata unsigned char *semaphore
)
```

`acquire_semaphore()` is only present in case the kernel is compiled with semaphore support. As argument it receives a pointer to a semaphore variable that needs to be located within the internal RAM of the 8051. In case the specified semaphore is free, `acquire_semaphore()` will return immediately while marking the semaphore to be occupied. Else it will cause a suspend of the calling task and will return when the semaphore is getting free some time in the future.

Notice that `acquire_semaphore()` is usually not directly used by application code. Instead, it is indirectly used by higher–level macros that provide mutex and barrier support.

### 6.1.7 Mutex Macros

Before making use of a mutex, it needs to be globally defined via the following macro:

```
DEFINE_MUTEX(MUTEX)
```

The parameter `MUTEX` is an identifier for the according mutex. Defining a mutex automatically does declare it to be "free" or "not occupied".

The mutex can be used by the application by instantiating the following macros:

```
ENTER_CRITICAL_SECTION(MUTEX)
LEAVE_CRITICAL_SECTION(MUTEX)
```

Notice that these macros must not be used within interrupt service routines!

### 6.1.8 Barrier Macros

The barrier handling is much like the mutex handling. A barrier needs to be defined with

```
DEFINE_BARRIER(BARRIER)
```

The parameter `BARRIER` is an identifier for the according barrier. Defining a barrier automatically does declare it to be "set". I.e. a task reaching it won't pass it.

The barrier can be used by instantiating the following macros:

```
WAIT_FOR_BARRIER(BARRIER)
CLEAR_BARRIER(BARRIER)
```

Notice that per barrier there has to be exactly one task and no more that is making use of `WAIT_FOR_BARRIER()`, while multiple tasks including application interrupt service routines can apply `CLEAR_BARRIER()` for one and the same barrier.

## 6.2 Kernel Source Structure

The kernel consists of the following files:

- `kernel.c`
  This is the main implementation file.
- `kernel.h`
  `kernel.h` is the main header file that is to be included by the application firmware. It contains prototype declarations of "public" kernel functions.
- `kernel_definitions.h`
  This header file contains various definitions and macros.
- `kernel_setup.h`
  In `kernel_setup.h` several settings regarding the kernel operation are to be made. Normally this is the only file that is to be modified according to the application firmware needs.
  These things will be explained along with the example firmware description shown in the next section.

# 7 Example Application

In this section I want to present a small exemplary firmware that demonstrates the ease of use of the kernel and most of its features. It is part of the kernel distribution and so to speak the default application.

This application firmware is not of practical use, but merely academic. The things realized by the firmware:

- There are two tasks 1 and 2.
- Task 1 inverts bit 4 of port C every approx. 0.5s.
- Task 2 inverts bit 5 of port C every time it is instructed from task 1 to do so.
- Task 1 is telling task 2 to do its job every 10 inversions of bit 4 of port C.
- There is an independent timer interrupt that inverts bit 0 of port E regularly by making use of timer 0.
- The system clock is to be set to 48MHz.

As both tasks 1 and 2 access port C, this access needs to be protected by a critical section. Because critical sections cannot be used from within an interrupt service routine, the interrupt routine cannot access port C as well. For this example application this is no problem as the interrupt service routine is accessing the independent port E.

## 7.1 Setting up the Kernel

First of all, some of the vital kernel settings need to be made. These things are to be done within the header file `kernel_setup.h`. Its contents for this exemplary firmware are shown in listing 4.

The CPU frequency is set to 48MHz (line 37). The

```
28 #ifndef __KERNEL_SETUP_H__
29 #define __KERNEL_SETUP_H__
30
31 // Definition of the maximal CPU clock frequency to be used. For the FX2LP this
32 // can be either 12MHz, 24MHz or 48MHz. Notice that the frequency might get
33 // forced down to 12MHz in case the system is idle and this feature has been
34 // enabled (see MINIMIZE_CLOCK_WHEN_IDLE below).
35 //#define CPU_FREQ_MHZ 12
36 //#define CPU_FREQ_MHZ 24
37 #define CPU_FREQ_MHZ 48
38
39 // The KERNEL_TICK_TIME_MILLISECONDS constant specifies the so-called
40 // tick time of the operating system in milliseconds.
41 // Notice that there are need to be paid attention to maximum values
42 // depending on the selected frequency:
43 //  - 12MHz: maximum permitted kernel tick time is 65ms
44 //  - 24MHz: maximum permitted kernel tick time is 32ms
45 //  - 48MHz: maximum permitted kernel tick time is 16ms
46 #define KERNEL_TICK_TIME_MILLISECONDS 10
47
48 // Definition whether the CPU frequency is to be set to the lowest
49 // possible frequency (12MHz) in case the system is idle.
50 // Remove/comment this definition in case changing the frequency is
51 // to be omitted.
52 //#define MINIMIZE_CLOCK_WHEN_IDLE 1
53
54 // Definition whether kernel semaphores are to be used or not.
55 // Remove/comment this definition to disable kernel semaphore support.
56 #define USE_KERNEL_SEMAPHORES 1
57
58 // Number of tasks excluding the idle task.
59 #define TASK_COUNT 2
60
61 // Definition of the base address for the context memory area.
62 // ATTENTION: Due to code optimization task_context_memory MUST start
63 //            at a 128 byte boundary in external memory
64 //            (i.e. 0xXX00 or 0xXX80)!!!!
65 // We set the base in a flexible manner occupying the upper range of
66 // the external memory included in the FX2LP. So the exact base depends
67 // on the number of tasks.
68 // Notice that we do not have to include the idle task here, because
69 // this one has got no context memory.
70 #define CONTEXT_MEMORY_BASE 0x4000 - 128 * TASK_COUNT
71
72 #endif // __KERNEL_SETUP_H__
```

Listing 4: `kernel_setup.h`

kernel tick time is set to a value of 10ms here (line 46).

The frequency reduction feature has been turned off (line 52). This is, because we want to have the application–specific timer interrupt rate at a constant level. When we allow a frequency switching, this would also affect the timer rate which is not desired.

Kernel semaphore support is turned on (line 56).

The number of tasks is set to 2 (line 59).

Finally, the address of the context memory location is defined in line 70. A flexible macro is used here that is placing the context memory at the very end of the 16kB of external memory offered by the FX2LP. So as there are two tasks, the memory from `0x3E00` — `0x3FFF` will be utilized. As already described in section 4.2, the compiler needs to be advised to not use this range for other data storage.

## 7.2 Firmware Main Code

The main firmware code is mostly self–explaining through its extensive comments. Nonetheless it's worth some extra words.

The `main()` function as well as some global definitions are shown in listing 5.

Definitions for the needed mutex as well as for the barrier by making use of the according macros can be seen in lines 34 and 38.

In `main()` first of all the stretch–setting becomes set to zero (line 54). This is an FX2LP–related feature that allows to insert wait states during accesses to external memory.

Next timer 0 is set up (lines 64–78). It will be used as time base to generate a square wave output at bit 0 of port E.

Then port E is configured properly (lines 82–84).

```
31  // Definition of a (global) mutex that is needed by this application
32  // in order to ensure avoiding "lost updates" for accesses to some
33  // io port registers.
34  DEFINE_MUTEX(port_access_mutex);
35
36  // Definition of a (global) barrier that is needed for the control of
37  // task 2 through task 1.
38  DEFINE_BARRIER(task2_barrier);
39
40  // Declaration of some function prototypes.
41  void timer0_interrupt(void) __interrupt 1;
42  void task1(void);
43  void task2(void);
44
45  ////////////////////////////////////////////////////////////////////////////
46  // main() is the initial function that will be entered after a reset.
47  void main(void) {
48
49    // Clear bits 2:0 of CKCON in order to select a stretch of 0.
50    // Note: Per default stretch=1 is selected which is slowing down
51    //       the access of external memory by one instruction cycle.
52    //       Notice that in case of the FX2LP the on-chip 16kB memory is
53    //       also treated as external memory!
54    CKCON &= ~0x07;
55
56    // ------------------------------------------------------------------------
57    // Initialize timer 0 which will be used for generating periodic interrupts.
58    // This is just for explanatory purposes.
59    // Timer 0 will be configured as simple 16 bit counter that will generate
60    // an interrupt every 65536*12 CPU clock cycles (approx. 61Hz for 48MHz).
61
62    // TMOD.2 = 0 (Timer 0 is a timer)
63    // TMOD.1:0 = 01 (Timer 0 in mode 1 - 16 bit timer)
64    TMOD &= 0xF8; // clear bits 2:0
65    TMOD |= 0x01; // bits 1:0 = 01 => Timer 0 mode 1
66
67    // Clear bit 3 (T0M) of CKCON (Timer 0 uses CLK24/12)
68    CKCON &= 0xF7;
69
70    // Load the initial Timer 0 value.
71    TL0 = 0;
72    TH0 = 0;
73
74    // Enable Timer 0 interrupt.
75    IE |= 0x02;
76
77    // Enable Timer 0.
78    TCON |= 0x10;
79
80    // ------------------------------------------------------------------------
81    // Initialize port E and configure bit 0 of port E as output.
82    PORTECFG = 0x00;
83    IOE = 0x00;
84    OEE = 0x01;
85
86    // ------------------------------------------------------------------------
87    // Initialize context data for task 1.
88    initialize_task_context(1, 0, (void*) &task1);
89
90    // Initialize context data for task 2.
91    initialize_task_context(2, 0, (void*) &task2);
92
93    // Enable global interrupts.
94    IE |= 0x80;
95
96    // Start up the kernel including the registered tasks.
97    // We will not return from there and this main() function basically
98    // terminates here.
99    start_kernel();
100
101 }
```

Listing 5: `firmare.c` (Main Code)

```
103 ////////////////////////////////////////////////////////////////////////////
104 // timer0_interrupt() is the interrupt service routine for timer 0. It just
105 // inverts bit 0 of port E.
106 //
107 // Notes:
108 //  - Kernel functions (i.e. sleep() or suspend()) cannot be called from
109 //    interrupt routines!
110 //  - No synchronization macros except CLEAR_BARRIER() must be used within an
111 //    interrupt service routine!
112 void timer0_interrupt(void) __interrupt 1 {
113
114   // Just invert bit 0 of port E.
115   IOE ^= 0x01;
116
117 }
```

Listing 6: `firmare.c` (Application–specific Interrupt Service Routine)

```
119 ////////////////////////////////////////////////////////////////////////////
120 // task1() is a simple task that is preparing port C / bit 4 and toggles its
121 // state regularly. Every 10 toggles there is cleared a barrier for task 2,
122 // so that task 2 can proceed its work.
123 void task1(void) {
124
125   unsigned char counter = 0;
126
127   // First we initialize port C accordingly.
128   // Although this is not really necessary, because the register change
129   // operations are carried out atomically by the 8051, we protect
130   // the access by an according mutex. So this does also illustrate the
131   // mutex operation for demonstrational purposes.
132   // Notice, that for simplicity we put all three initialization steps
133   // within a single critical section.
134   ENTER_CRITICAL_SECTION(port_access_mutex);
135   PORTCCFG &= 0xEF; // clear bit 4 (make port C / bit 4 normal IO pin)
136   IOC      &= 0xEF; // clear bit 4 (set port C / bit 4 zero)
137   OEC      |= 0x10; // set bit 4 (make port C / bit 4 output)
138   LEAVE_CRITICAL_SECTION(port_access_mutex);
139
140   // The actual task consists of an endless loop.
141   do {
142
143     // Put the task to sleep for a dedicated (minimal) time. The time is to be
144     // specified in kernel ticks. The time a kernel tick takes has to be specified
145     // in kernel_setup.h. With a sleep time of 50 and a kernel tick time of 10ms,
146     // this equals a total time of 0.5s.
147     sleep(50);
148
149     // Invert port C / bit 4. Again, we protect this operation by the
150     // according mutex, although this is not really necessary, because
151     // the inversion will be carried out by the 8051 as an atomic operation.
152     ENTER_CRITICAL_SECTION(port_access_mutex);
153     IOC ^= 0x10;
154     LEAVE_CRITICAL_SECTION(port_access_mutex);
155
156     // Increment a counter.
157     // Every time it reaches a value of 10, it is set back to zero and a
158     // barrier evaluated by task2 becomes cleared.
159     counter++;
160     if (counter == 10) {
161       counter = 0;
162       CLEAR_BARRIER(task2_barrier);
163     }
164
165   } while (1);
166
167 }
```

Listing 7: `firmare.c` (Task 1)

```
169 ////////////////////////////////////////////////////////////////////////////
170 // task2() just inverts bit 5 of port C whenever it receives a "go" from
171 // task 1. This illustrates the barrier functionality.
172 void task2(void) {
173
174   // Similar to task 1, initialize bit 5 of port C.
175   ENTER_CRITICAL_SECTION(port_access_mutex);
176   PORTCCFG &= 0xDF; // clear bit 5 (make port C / bit 5 normal IO pin)
177   IOC      &= 0xDF; // clear bit 5 (set port C / bit 5 zero)
178   OEC      |= 0x20; // set bit 5 (make port C / bit 5 output)
179   LEAVE_CRITICAL_SECTION(port_access_mutex);
180
181
182   // The actual task consists of an endless loop.
183   do {
184
185     // We go to sleep here and wait for the barrier task2_barrier to be cleared
186     // by task 1.
187     WAIT_FOR_BARRIER(task2_barrier);
188
189     // When the barrier has been cleared, we invert bit 5 of port C.
190     ENTER_CRITICAL_SECTION(port_access_mutex);
191     IOC ^= 0x20;
192     LEAVE_CRITICAL_SECTION(port_access_mutex);
193
194     // Then we will repeat the loop. I.e. we wait until the barrier has been
195     // cleared again.
196
197   } while (1);
198
199 }
```

Listing 8: `firmare.c` (Task 2)

Now the actual kernel–related work is carried out. In lines 88 and 91 the contexts of the two involved tasks become initialized. The global interrupt enable flag becomes set as well (line 94). Enabling interrupts is important as this is not done within the kernel.

Finally, in line 99 `start_kernel()` is called which rounds up the `main()` function. Code that follows calling `start_kernel()` would never get executed.

The interrupt service routine for timer 0 is shown by listing 6. It just inverts bit 0 of port E. That's all.

Listing 7 shows the code of the function that is representing task 1.

Task 1 has a local counter variable as it needs to know when to signal task 2 to proceed.

In lines 134–138 the according fraction of port C becomes initialized. Although not really necessary, the accesses of the port C registers are wrapped within a critical section. Notice that one would actually put the port C initialization into the `main()` function. However, this has not been done here for purposes of illustration.

The actual work of task 1 is carried out within an endless loop that spans across lines 141–165.

Within the loop, first of all task 1 puts itself to sleep for 50 kernel ticks. As the kernel tick time has been set to 10ms (see listing 4), the absolute sleep time is 500ms or 0.5s. Notice that this time is subject to a small jitter as described earlier.

After task 1 is waking up again, it inverts bit 4 of port C (lines 152–154). This inversion is also wrapped within a critical section, because formally task 2 can simultaneously access port C as well. Actually, this will never be a problem, as the toggle–operation will be translated into an atomic read–modify–write. Nonetheless, the critical section is used in order to be formally correct and to illustrate the use of it.

Once the primary work is done the local counter becomes incremented and checked whether it reached 10 already. If so, it becomes reset to zero and the barrier that is used for the control of task 2 becomes cleared.

The initial code of task 2 (listing 8) is similar to that of task 1. There is just made the proper configuration of bit 5 of port C.

Likewise to task 1, the main part of task 2 is done within an endless loop (lines 183–197).

In the beginning, task 2 is waiting for the barrier to be cleared. Once the barrier has been cleared by task 1, task 2 will wake up at the next occasion. Then it inverts bit 5 of port C and the loop is starting over again.

### 7.3 Checklist

As a reminder, here are a few points that should be checked in order to avoid troubles by means of saving days to weeks of endless debugging. Not taking care of these points can result in a very unpredictable behavior and in worst case very sporadic failures that are very difficult to locate.

- Do more than one of my tasks make use of other CPU registers than those ones described in section 4.2? If so, the kernel needs to be extended accordingly.
- Is my firmware **not** compiled with the `--xstack` compiler option? Using an external stack is not yet supported by the kernel!
- Does the position of the context memory in external memory (`CONTEXT_MEMORY_BASE`) as defined in `kernel_setup.h` not overlap with the space for `xdata` variables? The latter space is defined through the compiler options `--xram-loc` and `--xram-size`.
- Do I have declared all functions and their sub functions that can be called by different tasks as `reentrant`? Is the same true for third–party libraries that I'm using?
- Do I make use of critical sections for accesses to resources used by more than one task?

## 8 Availability

The source code for the multi tasking kernel is freely available through www.digital-force.net. It is provided together with the exemplary firmware described herein (including a small make file), which is an easy starting point for developing own applications.

There are no specific prerequisites to compile the example except for the availability of the common make system and an installed SDCC. As of this writing, the example has been verified to work correct when compiled with SDCC version 2.9.0.

## 9 Licensing/Legal Issues

This piece of software (including the source code) is provided as it is without being bound to any specific licensing model. It can be used and/or modified without any notice for any applications including commercial ones. There is no warranty that it is free of errors. You are using it on your own risk.

Furthermore notice that I'm not providing official support for this project. In case you encountered a problem with it or you discovered a bug, you are welcome to report this. But I can't promise that I've got the time to have a look at it.

## 10 Summary and Future Work

There has been presented a small but powerful multi tasking kernel. It does by far not replace larger operating system kernels, but it is also not intended to do so. Instead, it is removing much of the burden from the firmware designer to deal with complex issues of pseudo–concurrency within his application, while the extra overhead is kept at a comparably low level.

One of the most important differences when compared to regular operating systems is its source–based nature. That is, it cannot be compiled as a stand–alone system and then plugged together with the actual application code by means of linking object files. This is because all of its parameterization is done on a source–level. Although it is well possible to change that situation, this would yield a further increase of the kernel overhead. Considering the fact that the kernel is supposed to be open–source anyway, this extra overhead can be saved.

For the near future there is no dedicated work planned. Although there are many things one can imagine that could be improved or extended, the kernel will more and more loose one of its advantages: Its simplicity and its small size.

Anyway, possibly there will be some future enhancements and ports to other processor architectures. This will depend on pending projects and their requirements.

## References

[1] Cypress Semiconductor: EZ-USB FX2LP USB Microcontroller, CY7C68013A, CY7C68014A, CY7C68015A, CY7C68016A.
http://www.cypress.com (see USB High-Speed Peripherals)

[2] Sandeep Dutta et al.: Small Devices C Compiler.
http://sdcc.sourceforge.net