Real-Time Executive (REX)
User Guide

80-24897-1 Rev. C
September 6, 2001

Restricted Distribution: This document contains critical information about QUALCOMM products and may not be distributed to anyone that is not an employee of QUALCOMM without the approval of Configuration Management.
## Contents

1 Introduction ............................................................................................................. 1-1
   1.1 Purpose .............................................................................................................. 1-1
   1.2 Scope ................................................................................................................. 1-1
   1.3 Organization ..................................................................................................... 1-1
   1.4 Conventions .................................................................................................... 1-1
   1.5 Revision history ............................................................................................... 1-2
   1.6 References ........................................................................................................ 1-2
   1.7 Technical assistance ......................................................................................... 1-2
   1.8 Terms and definition ....................................................................................... 1-3

2 Overview of REX ................................................................................................... 2-1
   2.1 Tasks ................................................................................................................ 2-1
   2.2 Stacks ............................................................................................................... 2-1
   2.3 Priorities and scheduling ................................................................................. 2-2
   2.4 Interrupts ......................................................................................................... 2-2
   2.5 Mutual exclusion ............................................................................................ 2-2
   2.6 Suspending with interrupts disabled .............................................................. 2-2
   2.7 Signals ............................................................................................................. 2-2
   2.8 Timers ............................................................................................................. 2-3
   2.9 Extended interface ........................................................................................ 2-3

3 Tasks ...................................................................................................................... 3-1
   3.1 Task creation ................................................................................................... 3-1
      3.1.1 Scheduler .................................................................................................. 3-2
      3.1.2 Task specifics for ARM platforms ......................................................... 3-3

4 Interrupt Handling .................................................................................................. 4-1

5 Signals and Timers .................................................................................................. 5-1
   5.1 Signals ............................................................................................................. 5-1
   5.2 Timers ............................................................................................................. 5-2
6 Features ............................................................................................................. 6-1

7 Inside REX ........................................................................................................... 7-1
   7.1 Task control block......................................................................................... 7-1
   7.2 Current task .................................................................................................. 7-1
   7.3 Best task ....................................................................................................... 7-1
   7.4 Task list ......................................................................................................... 7-2

8 Programming Reference ..................................................................................... 8-1
   8.1 rex_init( ) .................................................................................................. 8-2
   8.2 rex_def_task( ) ......................................................................................... 8-4
   8.3 rex_set_sigs( ) ......................................................................................... 8-6
   8.4 rex_clr_sigs( ) ......................................................................................... 8-7
   8.5 rex_get_sigs( ) ........................................................................................ 8-8
   8.6 rex_wait( ) ............................................................................................... 8-9
   8.7 rex_def_timer( ) ....................................................................................... 8-10
   8.8 rex_set_timer( ) ....................................................................................... 8-11
   8.9 rex_get_timer( ) ....................................................................................... 8-12
   8.10 rex_clr_timer( ) ..................................................................................... 8-13
   8.11 rex_pause_timer( ) ................................................................................ 8-14
   8.12 rex_resume_timer( ) .............................................................................. 8-15
   8.13 rex_tick( ) ............................................................................................. 8-16
   8.14 rex_timed_wait( ) .................................................................................. 8-17
   8.15 rex_self( ) .............................................................................................. 8-18
   8.16 rex_get_pri( ) ........................................................................................ 8-19
   8.17 rex_set_pri( ) ........................................................................................ 8-20
   8.18 rex_task_pri( ) ...................................................................................... 8-21
   8.19 rex_set_interrupt_vector( ) .................................................................... 8-22
   8.20 rex_enable_interrupt( ) ......................................................................... 8-23
   8.21 rex_disable_interrupt( ) ....................................................................... 8-24
   8.22 rex_init_crit_sect( ) ............................................................................... 8-25
   8.23 rex_enter_crit_sect( ) ............................................................................ 8-26
   8.24 rex_leave_crit_sect( ) ............................................................................ 8-27
Figures

Figure 3-1  Context frame of a suspended task on ARM..................................................3-3

Tables

Table 1-1  Revision history..............................................................................................1-2
Table 1-2  Reference documents and standards...............................................................1-2
1 Introduction

1.1 Purpose

This document describes the features and operation of the Real-Time Executive (REX) operating system for ARM®-processor-based systems.

REX is a simple, efficient, preemptible, multitasking real-time operating system (RTOS) designed for small embedded systems. It was initially designed for use on the Intel® 80186 processor. Subsequently, it was ported to the ARM microprocessor. This document provides a guide to REX. See Chapter 8 for a programming reference.

1.2 Scope

This guide is intended for engineers who need to write applications to run in REX on an ARM processor.

1.3 Organization

This document is organized as follows:

- Chapter 2 – provides an overview of REX
- Chapters 3 through 7 – provide a more detailed look at REX’s features and inner workings
- Chapter 8 – provides a programming reference

1.4 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font. For example: #include

Code variables appear in angle brackets. For example: <number>
1.5 Revision history

The revision history for this document is shown in Table 1-1.

Table 1-1 Revision history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rev.</td>
<td>Jan 1999</td>
<td>Initial release</td>
</tr>
<tr>
<td>A</td>
<td>Mar 1999</td>
<td>Updated content</td>
</tr>
<tr>
<td>B</td>
<td>Jun 2001</td>
<td>Deleted obsolete 80186-related content. Updated REX API. Updated section on interrupt handling.</td>
</tr>
<tr>
<td>C</td>
<td>Sep 2001</td>
<td>Deleted profiling information.</td>
</tr>
</tbody>
</table>

1.6 References

Reference documents, which may include QUALCOMM, standards, and resource documents, are listed in Table 1-2.

Table 1-2 Reference documents and standards

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Document</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>REX Portation User Guide</td>
<td>80-24880-1 X1 Mar 1999</td>
</tr>
<tr>
<td>2</td>
<td>REX++ - REX Extensions Users Guide</td>
<td>80-V3083-1 X1 Apr 2001</td>
</tr>
</tbody>
</table>

1.7 Technical assistance

For assistance or clarification on information in this guide, email QUALCOMM CDMA Technologies atasicapps@qualcomm.com.
1.8 Terms and definition

The following terms are used throughout this document:

APCS     ARM procedure call standard. In order to support flexible mixing of routines generated by different compilers and written in assembly language, ARM Limited has defined this set of rules for procedure entry and exit, and for register usage.

API      Application programming interface

ARM      Advanced RISC Machines Limited (ARM). The manufacturer of the ARM7TDMI processor. In common use the company name, ARM, is also used as the name of the processor.

ARM7TDMI Official name of the ARM processor

CPSR     Current Program Status Register. This register holds the flags, interrupt control bits, and the current operating mode of the processor.

critical section Block of code that accesses a shared resource

FIQ      One of two interrupts used by the ARM processor. FIQ is also used to denote the mode of the processor when the FIQ interrupt is being serviced.

IRQ      One of two interrupts used by the ARM processor. IRQ is also used to denote the mode of the processor when the IRQ interrupt is being serviced.

ISR      Interrupt service routine, also called interrupt trampoline function

Idle Task This is the lowest priority task in the system. It has a priority of zero and runs when there is no other task ready to run. It sits in a tight loop doing nothing and loses control of the CPU only due to an interrupt.

PC       Program counter register

PSR      Program status register

REX      QUALCOMM’s real-time executive operating system

RTOS     Real-time operating system

SPSR     Saved program status register. This is banked for each processor mode. The SPSR for a mode holds the CPSR value at the time the processor changed to that mode.

TCB      Task control block. This is a REX internal data structure that holds task-specific information.

timer block Another name for a timer

trampoline function Function outside REX that services interrupts, also called ISR
2 Overview of REX

REX is a preemptive, multitasking RTOS that provides APIs for task control, task synchronization, mutual exclusion, timers, and interrupt control. All of its functions are executed in the context of the calling task.

2.1 Tasks

REX treats tasks as independently handled entities, each having its own stack and priority that collectively constitute the context of the task. Each task has an associated data structure, called a task control block (TCB), which REX uses to keep track of the task’s context.

REX allows an arbitrary number of tasks to be created dynamically at any time during execution. Realistically, however, REX’s performance degrades slightly with each added task since it must traverse a longer list as more tasks are created. Care must be taken to keep the number of tasks to a minimum.

Ultimately, the task load that REX can support is dependent upon the processor chosen, the processor clock speed, and the interrupt characteristics of the particular system.

2.2 Stacks

As mentioned above, each task has its own stack that is active whenever the task is running. When a task is suspended, its context is saved in a context frame on top of its stack, and the context frame’s stack pointer is saved in the task’s TCB. An example of a suspended task is one that is blocked, waiting for a signal, or waiting for an interrupt service.

When the task is again selected for running, the stack pointer is restored from the task’s TCB, the task’s context frame is popped off its stack, and task execution resumes where it left off. The process of task switching is virtually invisible to tasks.
2.3 Priorities and scheduling

Each task has a priority associated with it that is stored in its TCB. Priority can be any nonzero, positive 32-bit integer with lower values indicating lower priorities. REX reserves priority zero for use with the Idle task. Older versions of REX had the restriction that each task must be assigned a unique priority. This restriction has now been eliminated.

When scheduling, REX always chooses the highest priority ready task—the highest priority task not waiting on some event. If this task is not unique, REX may choose any one of the highest priority tasks. The chosen task will continue to execute until it voluntarily suspends or until an interrupt occurs that reactivates a higher priority task.

When the wait condition for a suspended task is satisfied, that task becomes ready. In cases where all tasks are suspended, the Idle task is executed.

REX also provides a mechanism by which a task can dynamically raise or lower its own or another task’s priority.

2.4 Interrupts

REX implements a preemptible kernel. On returning from an interrupt, control is passed to the highest priority ready task and not necessarily to the task that was interrupted.

2.5 Mutual exclusion

When two tasks share a common resource, a method may be required to make their accesses to that resource mutually exclusive. The block of code that accesses a shared resource is called a critical section.

Locking interrupts is one way that REX facilitates critical sections. REX also provides a less drastic method to lock-shared resources by exporting functions that can be used to lock and free common resources.

2.6 Suspending with interrupts disabled

Disabling interrupts is valid for the current task only. If a task suspends itself with interrupts disabled, the interrupt state is saved with the task, which will be restored when the task becomes active again. Interrupts are then potentially enabled for the next active task. This behavior may change in the future, so the programmer should not depend on its availability.
2.7 Signals

A set of general-purpose signals is associated with each task. These general-purpose signals are kept as part of the task’s context in its TCB, and they are used to notify the task that some kind of event has occurred. A task’s signals may be set or cleared by any task or interrupt handler.

A task can poll the current state of its signals at any time. Moreover, a task can suspend its own execution pending the setting of one or more of its signals. However, it can neither poll nor wait for signals of another task.

Important points to keep in mind about signals are:

- Signals are general-purpose flags whose meanings may be defined in any way by the user.
- A task may optionally suspend execution pending the setting of particular signals, presumably by another task or an interrupt handler. The emphasis here should be optionally, since setting a signal for a task that is not suspended on that signal does not directly affect the execution state of that task.

2.8 Timers

Timers are used to perform software interval timing. When using a timer, a task specifies the interval to be timed and a signal-mask to be set when the timer expires. The task may then poll to determine when the signal mask is set, or it may suspend until the signal mask is set.

Timers may be used in conjunction with other events that set signals, making it possible to perform timeout detection for those events. REX maintains a timer list that contains all active (that is, unexpired) timers. An arbitrary number of timers may be created although, as with tasks, performance deteriorates slightly with each active timer since a longer list must be traversed. There is no overhead cost for timers that have expired.

2.9 Extended interface

REX optionally provides an extended interface that exports functionality such as asynchronous procedure calls, deferred procedure calls, and dynamic memory management. Refer to the REX++ – REX Extensions Users Guide (80-V3083-1) for details.
3 Tasks

This chapter presents a detailed discussion of REX’s scheduling and context-switching mechanism.

3.1 Task creation

Tasks are created by `rex_def_task()`. REX does not allocate space for the task stack or its TCB. These have to be provided by the calling function. But REX does require that its users avoid direct manipulation of the TCB structure.

The task-creating function, `rex_def_task()`, does the following:

1. Handcrafts a context frame into the stack.
2. Sets up the TCB.
3. Places this task in the task list.
4. Calls the scheduler.

If the newly created task has a higher priority than the current task, then it gets control. Otherwise, the scheduler returns without performing a task switch. This task will get its chance to run when it becomes the highest priority ready task.

The initial stack frame is set up such that when a task is first given control, it always starts at `rex_task_preamble`, which in turn calls the task entry point, as specified in `rex_def_task()`.

A task is never supposed to return. So `rex_task_preamble()` has a call to `rex_kill_task()` after the call to the task entry function. This will kill the task if there is an erroneous return.
3.1.1 Scheduler

The scheduler is implemented as a function, `rex_sched()`, which should not be called by applications directly. It is internal to REX and should remain so. It is executed with interrupts disabled. The calling function must set the global variable `rex_best_task` before calling `rex_sched()`. This variable should be set to point to the TCB of the highest priority ready task in the system.

The basic algorithm of the scheduler is:

1. The scheduler checks to see if `rex_best_task` is the same as the current task, `rex_curr_task`.
2. If `rex_best_task` and `rex_curr_task` are the same, the scheduler simply returns.
3. If `rex_best_task` and `rex_curr_task` are different, `rex_curr_task` is assigned `rex_best_task`. Now, the scheduler checks to see if the system is currently servicing interrupts. If it is servicing interrupts, task switch is not performed and the scheduler returns.
4. Once the above tests pass, the scheduler performs the context switch. This involves pushing the current context onto the stack and storing the resultant stack pointer in that task’s TCB. Thereafter, the stack pointer is restored from the new `rex_curr_task` TCB, and the context is restored from that stack.
3.1.2 Task specifics for ARM platforms

ARM is a 32-bit processor and follows the flat memory model. The stack pointer in this case is only one field, `sp`, and is stored in the TCB.

Context frame

The context frame contains registers `r0-r12`, `lr`, `pc`, and `CPSR`. The stack of a suspended task is shown in Figure 3–1.

![Context frame of a suspended task on ARM](image)

Figure 3–1 Context frame of a suspended task on ARM

Remember also that `rex_sched()` runs with interrupts disabled. On the ARM, `rex_sched()` has to do these two actions itself. The Program Status Register stored in the context is the value of CPSR on entry into this function. Thus, the interrupt state is being stored with the task context.

The context is restored by these actions:

1. Stack pointer is loaded from the TCB that is pointed to by `rex_curr_task`.
2. The saved PSR value is copied into the SPSR register.
3. Then an `ldmf d` with a `^` (caret) does the following:
   a. Restores registers r0-r12, the link register, and the program counter.
   b. Switches the mode back to Thumb mode.
   c. Returns to the task.

**NOTE** The caret is required with `ldmf d` because the PC and CPSR need to be updated at the same time.
The actions performed by REX on entering and leaving an interrupt are discussed in this chapter.

The ARM has two interrupts, FIQ and IRQ. The discussion in this chapter pertains to handling the IRQ interrupt on ARM. The handling of the FIQ interrupt is much simpler, since currently context switches are not allowed in a handler that services an FIQ interrupt.

A function programmed into the exception vector table—IRQ_Handler(), written in ARM assembly—gets control as soon as the interrupt is asserted. This function then invokes the interrupt trampoline function. The trampoline function, which is defined outside of REX, typically detects which interrupt source has asserted an interrupt and calls the corresponding service routine. It is installed using rex_set_interrupt_vector() (see Section 8.19 for a detailed description). When an interrupt occurs, the ARM processor changes mode to IRQ and switches to a different bank of registers. The following actions are performed in IRQ_Handler:

1. Some registers will be trashed by the interrupt trampoline function. Save the registers that might be overwritten by the trampoline function on the interrupt stack. Since this function is a C routine, the registers saved are r0-r3, r10, r12, and r14. Save the SPSR as well since this register may be trashed by a subsequent nested interrupt.

2. Increment the interrupt nesting level counter.

3. Switch to System mode and save the System mode link register, which gets overwritten later.

4. Call the registered trampoline function.

5. On return from the trampoline function, restore the System mode link register and switch back to IRQ mode.

6. Decrement the interrupt nesting level counter.

7. Determine whether a task swap is required. A swap is needed only if the nesting level counter is zero and rex_best_task is not the same as rex_curr_task.

8. If a task swap is required, do the following:
   a. Restore the SPSR and saved registers from the IRQ mode stack.
   b. Switch to Supervisor mode.
   c. Save the context of the current task on its stack.
d. Save the Supervisor mode stack pointer in the task’s TCB.

e. Set `rex_curr_task` to `rex_best_task`.

f. Restore the context of `rex_curr_task`.

9. If a task swap is not required, restore the SPSR from the interrupt stack, restore the other registers saved on the interrupt stack, and return, switching back to Supervisor mode in the process.

**NOTE** In some versions of `IRQ_Handler`, there is no switch to System mode before calling the trampoline function. Versions of `IRQ_Handler` that do not support nested interrupts do not keep track of the interrupt nesting level.

This sequence of actions achieves preemption since control is returned to the highest priority ready task rather than the task that was running when the interrupt occurred.
5 Signals and Timers

Signals and timers are mechanisms through which REX tasks can interact and perform interval timing.

5.1 Signals

REX provides signals as an intertask communication mechanism. A set of signals is associated with each task. It is possible for any task to read, set, and clear any other task’s signals using rex_get_sigs(), rex_set_sigs(), and rex_clr_sigs().

A task can wait on any subset of its own signals to be set using rex_wait(). It cannot wait on some other task’s signals. The OR semantics are used, which means that the task will be woken up when any of the flags in its wait pattern is set. If a task requests to wait when a signal from its wait mask is already set, the wait will return immediately without suspending the task.

Signals are implemented as integers. The ARM provides a 32-bit mask, which means that 32 different signals can be set simultaneously for a task.
5.2 Timers

A task can define a new timer using `rex_def_timer()`. Once defined, a timer (also called a timer block) can be used by tasks to perform software interval timing:

- Use `rex_set_timer()` to start a timer.
- Use `rex_get_timer()` to read the time remaining in a timer. A remaining time of zero implies that the timer is not active.
- `rex_clr_timer()` clears a timer. The timer becomes inactive and its count is set to zero.
- `rex_pause_timer()` pauses a timer. The timer becomes inactive, but its current count is retained.
- Use `rex_resume_timer()` to restart a paused timer.

A task can use `rex_timed_wait()` to do timeout detection by waiting for a signal with a timer. Once the timer expires, it sets the signals specified in its block.

**NOTE** It is important to note that the task should be made to wait on a superset of the signals that the timer is going to set. This care is required because the timer block is usually initialized at the beginning of the program, when its signal pattern is set. While using `rex_timed_wait()`, a task specifies the signals it wants to wait on as a separate parameter, which, if the programmer is not careful, may not have anything in common with the signals in the timer block.

Time is specified in milliseconds. The minimum resolution possible is 1 ms. The actual resolution depends upon the platform and is equal to the number of milliseconds per clock-tick interrupt.

Time is advanced at each clock tick by the function `rex_tick()`, which must be called with the number of milliseconds per tick by the clock-tick ISR. The signals specified in the timer block are then set in the specified TCB when the timer expires.

REX maintains a list of unexpired timers. Once a timer expires, however, it is removed from that list and REX has no further knowledge of the structure’s existence. In other words, timer structures are allocated and maintained only by tasks.

**NOTE** `rex_def_timer()` can be called as many times as desired for a timer block. However, users must be very careful not to call this function on an active timer. Doing so will corrupt the timer list as well as a portion of the allocated memory—an action which is fatal to task execution.
6 Features

A brief explanation of REX features will be provided in this chapter in a future revision of this document.
7 Inside REX

Information on some of REX’s internal data structures and variables is provided in this chapter. These should never be manipulated directly by applications, although they are accessible. Knowing about them may help when debugging programs.

7.1 Task control block

Each task has a TCB that keeps track of the task context. The fields of interest in the TCB are:

- **sp** – This field stores the pointer to the top of stack. It is meaningful only for a suspended task.
- **stack_limit** – This field stores the stack limit for this task’s stack. The stack pointer should never go below this value.
- **sigs** – This field holds the signals currently set for this task.
- **wait** – This field holds the signals on which this task is waiting. A nonzero value here implies that the task is suspended, and a zero value implies that the task is ready to run or is running.
- **pri** – This field holds the task’s priority.

7.2 Current task

The global variable `rex_curr_task` contains a pointer to the TCB of the currently running task.

7.3 Best task

The global variable `rex_best_task` contains a pointer to the TCB of the highest priority runnable task.
7.4 Task list

All TCBs are kept in a queue that is sorted by priority. The queue itself is a doubly linked list with `next_ptr` and `prev_ptr` holding the next and previous pointers. The global structure `rex_task_list` is the headnode of this queue. Its `next_ptr` field points to the highest priority task in the system (although that task need not be ready to run). The highest priority ready task is found by traversing this queue from the head to the tail looking for a TCB with a `wait` field equal to zero. This search is guaranteed to succeed because the Idle Task is always at the end of this queue (since it has the lowest priority, zero) and it is always ready to run.

**NOTE** As mentioned earlier, the priority value of zero is reserved and cannot be used by application tasks.
8 Programming Reference

A programming reference for the REX API is provided in this chapter. Programmers should use these functions only according to the interface specifications, since REX does not check any parameter for validity. Functions include:

- `rex_init()`
- `rex_def_task()`
- `rex_set_sigs()`
- `rex_clr_sigs()`
- `rex_get_sigs()`
- `rex_wait()`
- `rex_def_timer()`
- `rex_set_timer()`
- `rex_get_timer()`
- `rex_clr_timer()`
- `rex_wait()`
- `rex_resume_timer()`
- `rex_tick()`
- `rex_timed_wait()`
- `rex_self()`
- `rex_get_pri()`
- `rex_set_pri()`
- `rex_task_pri()`
- `rex_set_interrupt_vector()`
- `rex_enable_interrupt()`
- `rex_disable_interrupt()`
- `rex_init_crit_sect()` (not available with older versions)
- `rex_enter_crit_sect()` (not available with older versions)
- `rex_leave_crit_sect()` (not available with older versions)
8.1 rex_init( )

Synopsis

extern void rex_init (  
    void *                  p_istack,              /* interrupt stack      */
    rex_stack_word_type     p_istksiz,             /* interrupt stack size */
    rex_tcb_type            *p_tcb,                /* task control block   */
    void *                  p_stack,               /* stack                */
    rex_stack_word_type     p_stksiz,              /* stack size           */
    rex_priority_type       p_pri,                 /* task priority        */
    void                    (*p_task)( dword ),    /* task function        */
    dword                   p_param                /* task parameter       */
);  

Description

This function initializes REX. It should be called once and only once during the power-up processing,  
before using any REX API. Interrupts should be enabled in the system only after this initialization.

This function:

1. Creates the first user task, p_task, with:
   - Stack p_stack of size p_stksiz (16-bit words)
   - Priority of p_pri
   - TCB p_tcb

2. Calls the scheduler, which gives control to this newly created task.

The first two parameters of this function, p_istack and p_istksiz, are present for backward  
compatibility and are no longer used.

Side effects

None

Return value

Not applicable
Usage

```c
#include "rex.h"
#define FIRST_STACK_SIZE 512 /* 1024 bytes */
#define FIRST_TASK_PRI 200 /* Some non-zero unique value */
#define FIRST_TASK_PARAM 101 /* Some value understood by the new task */

rex_tcb_type first_task_tcb;
int first_task_stack[FIRST_STACK_SIZE];

void first_task(dword p)
{
  /*
   * This is the first task in the system.
   * When control comes here …
   * - p has been set to FIRST_TASK_PARAM by REX
   * - first_task_stack is in use
   * - this function is not supposed to return.
   */
}

During power-up processing

rex_init (  
    NULL,
    0,
    &first_task_tcb,
    first_task_stack,
    FIRST_STACK_SIZE * sizeof (int) / sizeof (word),
    FIRST_TASK_PRI,
    first_task,
    FIRST_TASK_PARAM
  );
```
8.2 rex_def_task( )

Synopsis

```c
extern void rex_def_task ( 
    rex_tcb_type             *p_tcb,              /* valid TCB for new task */
    void *                   p_stack,             /* stack for new task */
    rex_stack_word_type      p_stksiz,            /* stack size for new task */
    rex_priority_type        p_pri,               /* priority for new task */
    void                     (*p_task)(dword),    /* task startup function */
    dword                    p_param              /* parameter for new task */
);```

Description

This function creates a new REX task. The TCB, pointed to by `p_tcb`, should have been allocated by
the calling function. The new task gets:

- Its own stack, `p_stack`, which is `p_stksiz` words long (16-bit words)
- Priority of `p_pri` and its entry point is `p_task` – This entry point will get the parameter
  `p_param`, when it first runs. This parameter value is not interpreted by REX; it is simply passed
to the task. Applications can attach any meaning to this value.

The scheduler is called as soon as REX defines this new task. However, there is no guarantee that this
task will get control immediately as a result of this API call. That will depend solely upon its priority
relative to other ready tasks in the system at the time it is called. This task will be created ready-to-
run.

Side effects

This function can cause a task switch.

Return value

None
Usage

#include "rex.h"
#define NEW_STACK_SIZE  512 /* 1024 bytes */
#define NEW_TASK_PRI 200 /* Some non-zero unique value */
#define NEW_TASK_PARAM 101 /* Some value understood by the new task */

rex_tcb_type new_task_tcb;
int new_task_stack[NEW_STACK_SIZE];

void new_task(dword p)
{
    /*
     * When control comes here ...
     * - p has been set to NEW_TASK_PARAM by REX
     * - new_task_stack is in use
     * - this function is not supposed to return.
     */
}

void existing_task(dword p)
{
    ...
    rex_def_task ( &new_task_tcb,
                  new_task_stack,
                  NEW_STACK_SIZE * sizeof (int) / sizeof (word) ,
                  NEW_TASK_PRI,
                  new_task,
                  NEW_TASK_PARAM
    );
    ...
}
8.3 rex_set_sigs()

Synopsis

```c
extern rex_sigs_type rex_set_sigs (
    rex_tcb_type   *p_tcb,      /* TCB for which the sigs will be set */
    rex_sigs_type   p_sigs      /* the sigs to set                    */
);```

Description

This function sets a specified mask of signals (p_sigs) for the specified task (p_tcb). The signals, p_sigs, are bitwise-ORed with the signals already set for the task. If that task is waiting on any of the signals set, it will again be eligible for execution.

Side effects

This function can cause a task switch.

Return value

The previous value of the signal mask of the task.

Usage

```c
#include "rex.h"
#define NEW_SIGNALS   0x3   /* Some signals. */

extern rex_tcb_type target_task;

void signal_setting_task( dword p)
{
    rex_sigs_type    old_sigs;
    old_sigs = rex_set_sigs( &target_task, (rex_sigs_type)NEW_SIGNALS);
}
```
8.4  rex_clr_sigs( )

Synopsis

extern rex_sigs_type rex_clr_sigs ( rex_tcb_type *p_tcb, /* TCB for which the signals will be cleared */
                                              rex_sigs_type p_sigs  /* which signals to clear */
                                           );

Description

This function clears the specified signals (p_sigs) for the specified task (p_tcb). The scheduler is not called.

Return value

The previous value of the signal mask of the task.

Usage

#include "rex.h"
#define CLEAR_THOSE 0x30 /* Some signals. */
extern rex_tcb_type target_task;
void signal_clearing_task( dword p)
{
  rex_sigs_type old_sigs;
  old_sigs = rex_clr_sigs(&target_task, (rex_sigs_type)CLEAR_THOSE);
}
8.5 rex_get_sigs()

Synopsis

extern rex_sigs_type rex_get_sigs(
    rex_tcb_type *p_tcb    /* TCB for which sigs will be returned */
);

Description

This function reads the state of the signals for a specified task (p_tcb).

Return value

The present value of the signal mask of the task.

Usage

#include "rex.h"
extern rex_tcb_type target_task;
void signal_reading_task( dword p)
{
    rex_sigs_type present_sigs;
    present_sigs = rex_get_sigs( &target_task );
}
8.6 rex_wait()

Synopsis

extern rex_sigs_type rex_wait (  
    rex_sigs_type p_sigs     /* signals to wait on */  
);  

Description

This function suspends the calling task until at least one signal of a specified signal mask (p_sigs) is  
set for the task. If any signal of the signal mask is already set prior to calling rex_wait(), an  
immediate return is made from the function and the calling task continues to execute.

A task can wait only on its own signals; it may not wait on any other task’s signals.

Side effects

This function can cause a task switch.

Return value

The present value of the signal mask of the task after its waiting is over.

Usage

#include “rex.h”
#define WAIT_FOR_THESE 0x50 /* Some signals. */

void wanting_to_wait_task( dword p)  
{  
    rex_sigs_type got_these_sigs;
    got_these_sigs = rex_wait((rex_sigs_type)WAIT_FOR_THESE);
}
8.7 rex_def_timer()  

Synopsis

```c
extern void rex_def_timer(
   rex_timer_type *p_timer, /* pointer to a valid timer structure */
   rex_tcb_type   *tcb_ptr, /* TCB to associate with the timer */
   rex_sigs_type  sigs    /* sigs to set upon timer expiration */
);
```

Description

This function:
- Initializes a specified timer block for use as an interval timer
- Sets up:
  - Which task (tcb_ptr) to notify when the timer expires
  - Which signal mask (sigs) to set

REX assumes that space for the timer block has already been allocated by the calling function.

This function must be called at least once before the timer block can be used. It must not be called on a timer block if that block is currently an active timer.

Return value

None

Usage

```c
#include "rex.h"
#define SET_THESE  0x12  /* Some signals. */
extern rex_tcb_type target_task;
rex_timer_type   new_timer;
void timer_defining_task( dword p)
{
   rex_def_timer(&new_timer, &target_task, (rex_sigs_type)SET_THESE);
}
```
8.8 rex_set_timer()

Synopsis

```c
extern rex_timer_cnt_type rex_set_timer(
    rex_timer_type       *p_timer, /* pointer to timer to set */
    rex_timer_cnt_type   msecs      /* value in milliseconds   */
);
```

Description

This function starts a timer (p_timer) for a specified number of milliseconds (msecs). The signal mask specified by the timer is cleared for the task associated with the timer and will be set later when the timer expires.

The timer block pointed to by p_timer must have been previously initialized using rex_def_timer(). If this timer is already active and running, then it is restarted with the new count.

Side effects

This function can cause a task switch.

Return value

The timer count before setting it to the new value.

Usage

```c
#include "rex.h"
#define TIMER_VALUE    20 /* 20-millisecond timer */
extern rex_timer_type  existing_timer;
void       timer_setting_task( dword p)
{
    rex_timer_cnt_type   old_timer_count;
    old_timer_count = rex_set_timer(&existing_timer, TIMER_VALUE);
}
```
8.9 rex_get_timer()

Synopsis

```c
extern rex_timer_cnt_type rex_get_timer(
    rex_timer_type *p_timer   /* pointer to the timer to get */
);
```

Description

This function reads out the current counter value for a specified timer (p_timer). A value of zero indicates that this timer is currently inactive.

Return value

The current counter value of the specified timer.

Usage

```c
#include "rex.h"
extern rex_timer_type existing_timer;
void timer_reading_task( dword p)
{
    rex_timer_cnt_type timer_count;
    timer_count = rex_get_timer(&existing_timer);
}
```
8.10 rex_clr_timer( )

Synopsis

extern rex_timer_cnt_type rex_clr_timer(
    rex_timer_type *p_timer    /* pointer to timer to clear */
);

Description

This function sets the counter value for a specified timer to zero. This, in effect, stops the timer and
removes it from the list of active timers.

Return value

The previous counter value of the specified timer.

Usage

#include "rex.h"
extern rex_timer_type existing_timer;
void timer_clearing_task( dword p)
{
    rex_timer_cnt_type old_timer_count;
    old_timer_count = rex_clr_timer(&existing_timer);
}
8.11 rex_pause_timer( )

Synopsis

extern void rex_pause_timer(
    rex_timer_type *p_timer    /* pointer to timer to pause */
);

Description

This function removes a specified timer from the list of active timers, which prevents the timer’s count from being decremented further. This function differs from rex_clr_timer() in that the count is not set to zero before the timer is removed from the active list.

Return value

None

Usage

#include "rex.h"
extern rex_timer_type existing_timer;
void timer_pausing_task( dword p)
{
    rex_pause_timer(&existing_timer);
}
8.12 **rex_resume_timer( )**

**Synopsis**

```c
extern void rex_resume_timer(
    rex_timer_type *p_timer    /* pointer to timer to resume */
);  
```

**Description**

This function resumes a paused timer by restoring it to the active timer list. This function differs from `rex_set_timer()` in that the starting count of the timer is not specified explicitly (the existing value is used instead).

**Return value**

None

**Usage**

```c
#include "rex.h"
extern rex_timer_type  paused_timer;
void    timer_resuming_task( dword p)
{
    rex_resume_timer(&paused_timer);
}
```
8.13 rex_tick()

Synopsis

extern void rex_tick(
    rex_timer_cnt_type msecs    /* number of milliseconds elapsed */
);

Description

This function is not for use by application tasks. It should be called by the clock-tick ISR for each
interrupt. The parameter msecs should be the number of milliseconds per clock tick. This value is
hardware-dependent.

This function runs through the list of unexpired timers and decrements their current count. If any
timer’s count reaches zero, that timer is removed from the list of active timers and the signals
specified in its block are set in the specified task.

Side effects

This function can cause a task switch.

Return value

None

Usage

#include "rex.h"
#define MS_PER_CLK_INTERRUPT 5
void clock_tick_isr()
{
    rex_tick(MS_PER_CLK_INTERRUPT);
}
8.14 rex_timed_wait()

Synopsis

extern rex_timer_cnt_type rex_timed_wait(
  rex_sigs_type p_sigs,  /* sigs to wait on */
  rex_timer_type *p_timer, /* timer to set and wait on */
  rex_timer_cnt_type p_cnt   /* timer to wait */
);

Description

This function is exactly the same as:

  rex_set_timer(p_timer, p_cnt);
  return rex_wait(p_sigs);

The calling task becomes ready to run when one of the signals is set, either by some other task or by
the expired timer.

Users must ensure that the signals (p_sigs) are a superset of the signals held in the timer block
(p_timer->sigs).

Side effects

This function can cause a task switch.

Return value

The present value of the signal mask of the task after its waiting is over.

Usage

#include "rex.h"
#define TIMER_VALUE   20 /* 20-millisecond timer */
#define WAIT_FOR_THESE   0x30  /* Some signals. */
extern rex_timer_type existing_timer;
void timed_waiting_task( dword p)
{
  rex_sigs_type got_these_sigs;
  got_these_sigs = rex_timed_wait ( (rex_sigs_type)WAIT_FOR_THESE),
                   &existing_timer,
                   TIMER_VALUE
                   );
}
8.15 rex_self( )

Synopsis
extern rex_tcb_type* rex_self ( void );

Description
This function returns a pointer to the TCB of the calling task. This allows a task to obtain the address of its own TCB.

Return value
Pointer to the TCB of the calling task.

Usage
#include “rex.h”
void dont_know_my_own_name_task( dword p)
{
    rex_tcb_type * my_tcb;
    my_tcb = rex_self();
}
8.16 rex_get_pri()

Synopsis
extern rex_priority_type rex_get_pri(void);

Description
This function returns the current priority of the calling task.

Return value
Current priority of the calling task.

Usage
#include "rex.h"
void dont_know_my_priority_task(dword p)
{
    rex_priority_type my_priority;
    my_priority = rex_get_pri();
}
8.17 rex_set_pri()

Synopsis

extern rex_priority_type rex_set_pri(
    rex_priority_type p_pri           /* the new priority */
);

Description

This function allows the calling task to change its own priority to that specified in p_pri.

Side effects

This function can cause a task switch.

Return value

Previous priority value.

Usage

#include "rex.h"
void nice_task(dword p)
{
    rex_priority_type old_priority
    /* I am nice. I reduce my priority by 10. */
    old_priority = rex_set_pri( rex_get_pri() - 10);
}


8.18 rex_task_pri()

Synopsis

```c
extern rex_priority_type rex_task_pri(
    rex_tcb_type       *p_tcb,     /* TCB to set priority on */
    rex_priority_type  p_pri       /* the new priority       */
);
```

Description

This function sets the priority of the specified task (`p_tcb`) to the specified value (`p_pri`).

Side effects

This function can cause a task switch.

Return value

Previous priority value of that task.

Usage

```c
#include "rex.h"
extern rex_tcb_type some_other_task;
void extremely_nice_task(dword p)
{
    rex_priority_type old_priority;

    /* I am extremely nice. I set this task’s priority to a 1000
    * above my own priority.
    */
    old_priority = rex_task_pri(
        &some_other_task,
        rex_get_pri() + 1000
    );
}
```
8.19 rex_set_interrupt_vector( )

Synopsis

```c
void rex_set_interrupt_vector(
    rex_vect_type v,
    void (*fnc_ptr)( void )
);
```

Description

REX maintains a software equivalent of an interrupt table. The parameter `v` signifies entries in this table. This function installs handlers for IRQ and FIQ interrupts in this table.

Side effects

None

Return value

None

Usage

```c
#include "arm.h"
#include "rex.h"

void irq_isr ( void );

void irq_handler_setting_function( )
{
    rex_set_interrupt_vector(P_IRQ_VECT, irq_isr);
}
```
8.20 rex_enable_interrupt()

Synopsis

void rex_enable_interrupt (  
    rex_vect_type v  
);  

Description

REX maintains a software equivalent of an interrupt table. The parameter \( v \) signifies entries in this table. This function enables IRQ and FIQ interrupts by setting the appropriate bits in CPSR.

Side effects

None

Return value

None

Usage

#include "arm.h"
#include "rex.h"

void irq_enabling_function( )
{
    rex_enable_interrupt (P_IRQ_VECT);
}
8.21 rex_disable_interrupt( )

Synopsis

void rex_disable_interrupt (  
  rex_vect_type v
)

Description

REX maintains a software equivalent of an interrupt table. The parameter v signifies entries in this table. This function disables IRQ and FIQ interrupts by setting the appropriate bits in CPSR.

Side effects

None

Return value

None

Usage

#include “arm.h”
#include “rex.h”

void fiq_disabling_function( )
{
  rex_disable_interrupt (P_FIQ_VECT);
}


8.22 rex_init_crit_sect( )

Synopsis
extern void rex_init_crit_sect (  
    rex_crit_sect_type *crit_sect /* Pointer to critical section variable */  
);  

Description
Initializes a critical section. All critical section variables must be initialized prior to use.

Side effects
None

Return value
None

Usage
#include "rex.h"

rex_crit_sect_type sample_cs;

void cs_initializing_function( )
{
    rex_init_crit_sect(&sample_cs);
}

NOTE This function is not available with some versions of REX.
8.23 rex_enter_crit_sect( )

Synopsis

extern void rex_enter_crit_sect ( 
    rex_crit_sect_type *crit_sect /* Pointer to critical section variable */
);

Description

Acquires a critical section and proceeds. If the critical section is locked by another task, puts the
calling task onto a queue to be woken up later when the owning task releases it. If the critical section
cannot be acquired immediately, the calling task blocks.

Side effects

The calling task will be suspended if the critical section is not available.

Return value

None

Usage

#include "rex.h"

extern rex_crit_sect_type initialized_cs;

void cs_entering_function( )
{
    rex_enter_crit_sect(&initialized_cs);
}

NOTE This function is not available with some versions of REX.
8.24 rex_leave_crit_sect( )

Synopsis

extern void rex_leave_crit_sect (  
       rex_crit_sect_type *crit_sect /* Pointer to critical section variable */  
);

Description

Allows a task to release a critical section that it is holding. The critical section then becomes available for acquisition by another task. If a higher priority task is waiting on the critical section, then when this task releases the critical section, the higher priority task will be swapped in.

Side effects

A context switch might occur.

Return value

None

Usage

#include "rex.h"

extern rex_crit_sect_type acquired_cs;

void cs_leaving_function( )  
{}  
   rex_leave_crit_sect(acquired_cs);

NOTE This function is not available with some versions of REX.