zilog
Embedded in Life
An IXYS Company

eZ80® CPU

Zilog Real-Time Kernel

Reference Manual

RM000619-1211

www.zilog.com

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

Revision History

Each instance in the following revision history table reflects a change to this document from its previous version. For more details, refer to the corresponding pages or appropriate links provided in the table.

Date	Revision Level	Description	Page
Dec 2011	19	Globally updated for the RZK v2.4.0 release.	All
Aug 2010	18	Globally updated for the RZK v2.3.0 release.	All
Nov 2008	17	Updated for the RZK v2.2.0 release; added the Board Support Package APIs, Wireless Local Area Network Driver, Wireless Local Area Network APIs, Universal Serial Bus Device APIs and Universal Serial Bus APIs sections; updated the Application Development and EMAC Data Structure sections.	xiv, 30, 42, 44, 295, 2, 333
Jul 2007	16	Updated for proper branding.	All
Jul 2007	15	Globally updated for the RZK v2.1.0 release.	All
Apr 2007	14	Updated the RZKSuspendInterruptThread, RZKResumeInterruptThread, RZKCreateTimer, RZKDevRead, RZKDevWrite and UARTControl sections; removed section titled Application Initialization in the IAR Environment.	91, 92, 146, 213, 215, 254
Jul 2006	13	Globally updated for the RZK v2.0.0 release.	All

Table of Contents

Introduction

This Reference Manual describes the APIs associated with Zilog Real-Time Kernel (RZK) software for Zilog's eZ80® CPU-based microprocessors and microcontrollers. The current RZK release supports the eZ80L92 microprocessor and the eZ80Acclaim!® family of devices, which includes the eZ80F91, eZ80F92 and eZ80F93 microcontrollers.

About This Manual

Zilog® recommends that you read and understand the chapters in this manual before using the product. This manual is a reference guide for RZK APIs.

Intended Audience

This document is written for Zilog customers who are familiar with real-time operating systems and are experienced at working with microprocessors, or in writing assembly code or in higher level languages such as C.

Manual Organization

This Reference Manual contains the following chapters and appendices.

Zilog Real-Time Kernel

This chapter provides an overview of RZK and why it is used for the eZ80® family of devices.

RZK Architecture

This chapter provides functional description of the RZK objects.

RZK APIs

This chapter provides summary of the RZK APIs.

Board Support Package APIs

This chapter summarizes and describes each of the BSP APIs, which cover the EMAC, UART, RTC, SPI, I^2C, WDT, ADC, USB, WLAN and Flash device driver.

RZK APIs and Context Switching

This chapter provides a list of each RZK API in terms of its context-switching capability.

API Definitions

This chapter provides detailed descriptions for the RZK APIs in terms of syntax, argument descriptions, return values and example code.

Appendix A. RZK Data Structures

This appendix provides description for RZK data structures.

Appendix B. RZK Error Conditions

This appendix provides description for RZK error conditions.

Appendix C. Interrupt Handling

This appendix provides a description of the RZK-specific prologues and epilogues used to manage interrupt service routines.

Related Documents

Table 1 lists the related documents that you must be familiar with, to use RZK efficiently.

Table 1. Related Documents

Zilog Real-Time Kernel Quick Start Guide	QS0048
Zilog File System Quick Start Guide	QS0050
Zilog File System Reference Manual	RM0039
Zilog TCP/IP Software Suite Quick Start Guide	QS0049
Zilog TCP/IP Software Suite Reference Manual	RM0041

Manual Conventions

This manual has adopted the following conventions to provide clarity and ease of use.

Use of X.Y.Z

Throughout this document, `X.Y.Z` refers to the RZK version number in `Major.Minor.Revision` format.

Use of <tool>

Throughout this documents, `<tool>` refers to ZDSII.

Courier New Typeface

Code lines and fragments, functions and various executable items are distinguished from general text by appearing in the Courier New typeface.

For example: `#include "ZSysgen.h"`.

Safeguards

When you use RZK with one of Zilog's development platforms, follow the precautions listed on this page to avoid permanent damage to the development platform.

Power-Up Precautions

When powering up, observe the following sequence.

1. Apply power to the PC and ensure that it is running properly.
2. Start the terminal emulator program on the PC.
3. Apply power through connector P3 on the eZ80 Development Platform.

Power-Down Precautions

When powering down, observe the following sequence.

1. Exit the monitor program.
2. Remove power from the eZ80 Development Platform.

Note: Always use a grounding strap to prevent damage resulting from electrostatic discharge (ESD).

Zilog Real-Time Kernel

The Zilog Real-Time Kernel (RZK) is a real-time, preemptive multitasking kernel designed for time-critical embedded applications. It is currently available for the eZ80® family of microprocessors and the eZ80Acclaim!® family of microcontrollers.

A major portion of the RZK source code is written in the ANSI C language; assembly language code is used only for target-related information.

You can link real-time applications with the RZK library. The resulting object can be downloaded to the Flash or RAM target, or placed in ROM.

Real-Time Response

Real-time response describes the software that produces the correct response to external and internal events at the proper time. *Real-time*, itself, is categorized as *hard* and *soft* real-time. In *soft* real-time, failure to produce a response at the correct time is acceptable. However, a similar failure occurring in *hard* real-time has the potential to be catastrophic.

Why RZK?

The advantages of using RZK are briefly described below:

- RZK's modular design concept allows you to custom-tailor RZK to meet your product requirements
- RZK is a reliable kernel that is tested extensively. It is backed by a seasoned software development and support team at Zilog with the intention of helping Zilog customers succeed in their respective ven-

tures by offering quality software components and technical knowledge

- RZK provides all of the standard benefits of a true kernel with very low memory requirements
- RZK is affordable in terms of product and development costs and resource overhead for most present-day implementations

Application Development

Embedded real-time applications are developed on a host computer system such as an IBM PC or a UNIX workstation, and the resulting application is cross-compiled to a target environment. The application software runs on another system referred to as a target system (for example, an eZ80® development module). The resulting binary image is either downloaded to target system RAM or programmed into ROM, EEPROM, Flash or some other nonvolatile device on the target system.

Debugging software on a target system usually involves the use of an extended set of debug tools such as Zilog Developer Studio (ZDSII) and ZPAKII. When developing with RZK via the ZDSII tool, RZK startup is executed through the ZDSII interface.

Application Initialization in the ZDSII Environment

Figure 1 displays the control flow of the Zilog RZK, from initialization in the ZDSII environment to the first thread's entry point function.

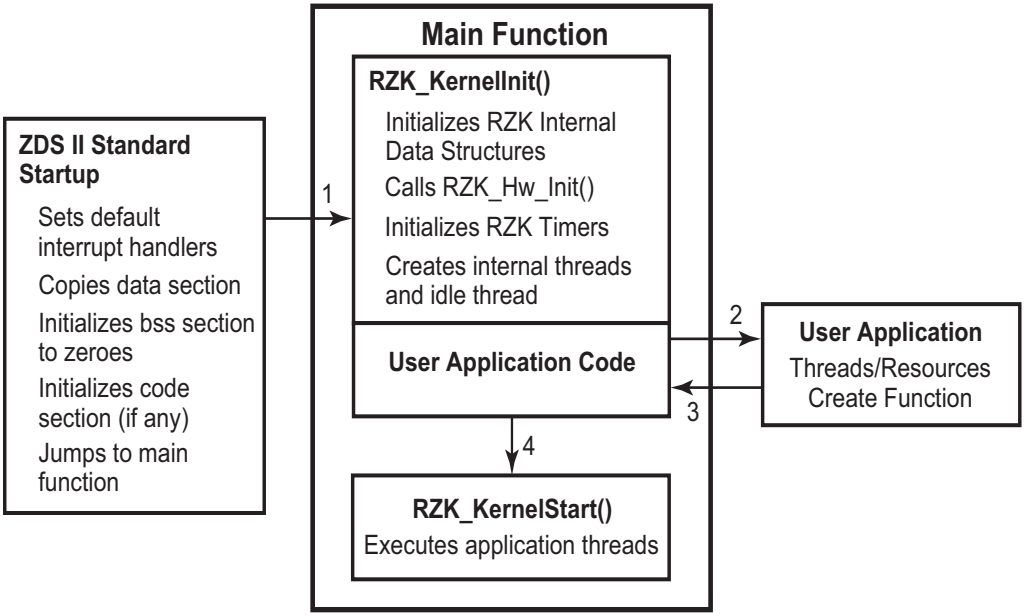

Figure 1. Control Flow of RZK Application Initialization in the ZDSII Environment

The ZDSII standard start-up function invokes the `main()` routine after execution of the basic system bootup. The `main()` routine contains the required RZK function calls. Initially, this routine invokes the `RZK_KernelInit()` routine that initializes the RZK internal data structures by creating internal threads. The `main()` routine then invokes the application routine that creates threads or resources.

Note: The user code is contained in the application routine and not in the `main()` routine.

The following section applies to ZDSII.

At the end of the `main()` routine, the `RZK_KernelStart()` function must be called. Calling this function starts the execution of application threads. After the `RZK_KernelStart()` function is invoked, the control never returns to the next statement in the `main()` routine.

The following code snippet is a sample implementation of the `main()` function.

```
int main( int argc, void *argv[] )
{
RZK_KernelInit() ; // This first statement of the main
// function must be present. It is a user application
// statement that creates threads or resources.

RZK_KernelStart() ; // This function is the final
// function that the main function executes and
// must be present.
}
```

Notes:

1. The `RZK_KernelInit()` routine creates and resumes the lowest-priority kernel thread that runs when no user-created threads are in a ready-to-run state. The entry point function name of this kernel thread is `idlethread`. This entry point function is exposed and contains an infinite while loop. To handle idle scenario, you can write a routine in this kernel thread.
2. Avoid performing blocking operations inside the `main()` routine. This function runs until the `RZK_KernelStart()` routine has completed executing, then passes control to the scheduler.
3. An idle thread executes when no other threads are executing. When entering a value for the maximum number of threads in the `RZK_Conf.c` file, this idle thread must be added to the number required by the user application in addition to an RZK internal thread. Therefore, the total number of threads is equal to the number of application threads plus two.

4. You can work without creating any other thread, in which case the application executes sequentially. That is, if no threads are created in the `main()` function, the application is executed sequentially and run to completion until `RZK_KernelStart()` is called.
5. Interrupts are disabled throughout the execution of the `main()` function. The interrupts are enabled only after the first thread starts running, that is, after the execution of the `RZK_KernelStart()` routine.
6. RZK creates another internal thread to run the RZK timer. Whenever a timer interrupt occurs, this interrupt thread resumes and processes various threads that are in finite suspended/blocked states. This interrupt thread also processes software timers and other bookkeeping routines.

Caution: Do not enable interrupts from the `main()` function. RZK execution is unpredictable under such circumstances.

RZK Architecture

This chapter discusses the RZK architecture. This architecture is configurable, scalable and modular in design, and provides a rich set of features and easy-to-use APIs. RZK features are tuned to the stringent memory and performance requirements of the 8-bit domain.

The RZK kernel consists of a preemptive scheduler, an algorithm to manage the Resource queue, the Dispatch queue (contains ready-to-run threads at different priority levels) and the Time queue (contains inter-task communications mechanism objects such as semaphores, message queues and events that are finitely blocked) and hardware-dependent routines (see Figure 2 on page 8). All RZK objects use the kernel services for resource management and provide a set of APIs as an interface to the application. As an option, the application can use the inter-task communication mechanism objects described above, as appropriate.

In contrast to a messaging approach, RZK is designed to provide standalone APIs that perform a number of required operations that increase operational efficiency and minimize time of execution.

Resource Queue Manager

The Resource Queue Manager manages RZK inter-task communication and synchronization mechanism objects such as semaphores, message queues and event groups, on which different threads are pending and/or blocking. Whenever a resource object (for example, a semaphore) is being released, has received a message (via the message queue) or has received an event (via an event group), the pertinent thread that is blocked on the resource object is awakened, depending on the attribute of the resource (Priority or FIFO). In essence, if a resource is created to wake up threads in a Priority order, the highest-priority thread that is blocked is

awakened. If in FIFO order, the thread that is blocked on the resource first is awakened.

Scheduler

The Scheduler, which depends upon different thread states, manages processor execution by highest-priority thread. The scheduler implements scheduling algorithms to manage priority-driven preemptive scheduling and round-robin scheduling.

Time Queue Manager

The Time Queue Manager manages the timing of different threads that must be executed and manages a queue that stores time-outs for threads in a round-robin fashion; these threads are finitely blocked on resources such as semaphores, message queues and events. For every system timer tick, the Time queue is updated and an appropriate thread is executed.

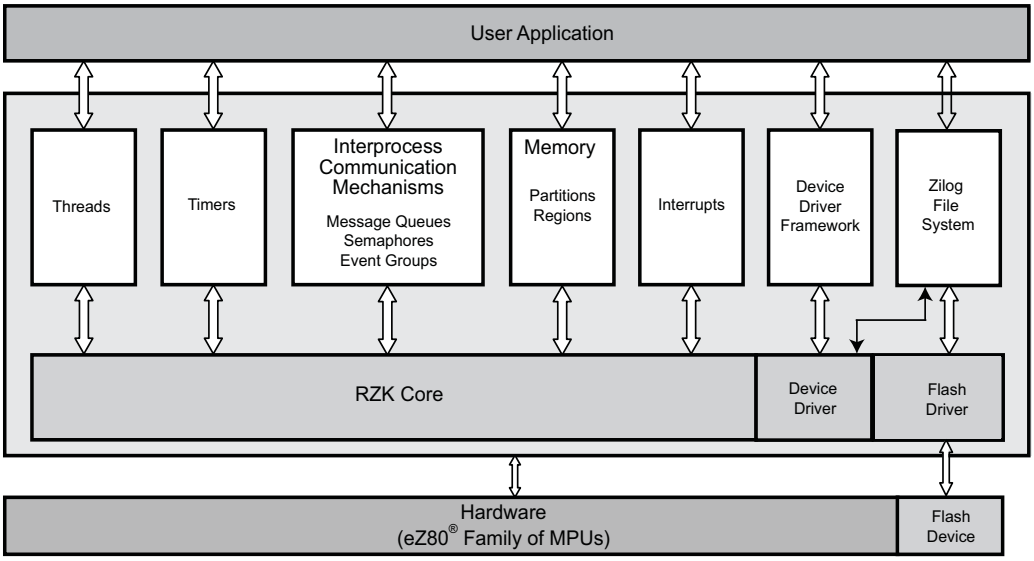

Figure 2. Zilog Real-Time Kernel Architecture

Threads

A thread is an RZK object. It is synonymous to the *task* definition in VRTX32™ context, and represents a self-sufficient execution entity that runs concurrently with other threads. Each thread competes with other threads for system resources such as CPU cycles, messages, semaphores and so forth. RZK enables a particular thread to execute among all other threads based on its priority and readiness to execute. RZK cannot be configured without a thread object. A unique handle identifies each thread. This thread handle is a pointer to the static thread control block (referred to as *TCB*, an RZK internal structure containing numerous details about a

thread). Each thread can also be identified by a name; this name is a fixed-length ASCII string.

Note: To keep the size of the TCB compact, the *name* field is omitted for nondebug mode.

Each thread contains a number of priority values. These values are described in Table 2.

Table 2. Thread Priority Values

Inherited	This value is determined by RZK based on the priority inheritance protocol. For details, see the inheritance and inversion concepts in the Semaphores section on page 15.
Dispatch	This value is used for queuing and selecting a thread for execution.

In addition, a part of the thread context (which refers to all CPU registers, program counter and stack pointer values when a thread is preempted), is stored in the TCB and updated when that thread is required to be preempted by another thread or an interrupt service routine (the routine called when an interrupt signal is encountered during the execution of an RZK application). To restart a thread, the initial values of the thread context, such as the stack pointer, the machine status word (refers to all CPU registers/stack pointers and the program counter), the round-robin time slice, pointers to an argument list and entry points, are stored separately.

The unit of time slice for round-robin scheduling is *system ticks*. The duration for time slicing is stored on thread basis.

To maintain statistics about thread information, such as the total and actual run-time of threads, the number of times a thread is blocked is also stored.

The state of the thread, such as `RUN`, `BLOCKED`, `TIMED BLOCK` and `SUSPENDED`, are also stored in the state field in the TCB. A thread waiting for an input to a message queue is in a `BLOCKED` state. If a message queue message is not received, the thread remains in the `BLOCKED` state. The `TIMED BLOCK` feature allows the application to specify how long a thread stays `BLOCKED` prior to discontinue waiting for the arrival of an input. A blocking time-out, in ticks, is stored by RZK.

For events, the respective masks are provided. A NULL terminated list of function addresses is managed to support clean-up functions at the time of thread deletion.

A number of thread states are defined in Table 3.

Table 3. Thread States

Thread State	Description
Running	Thread is running.
Ready	Thread is ready but another thread is running.
Suspended	Thread is suspended for a finite or infinite time (usually developer-specified) so that a lower-priority thread runs while this thread waits for an elapsed time.
Deleted	Thread has been deleted. This state occurs either because its operation is complete and a self delete occurs automatically, or because another thread has deleted it.
Blocked/Timed Block	Thread is blocked while waiting for a resource (resource is not available).

The following subsections cover the attributes of a thread, such as static creation, thread-switching time, and preemption.

Static Creation

You must decide the maximum number of thread objects to create. This number is referred to as a *static creation value*. Although RZK objects are created and deleted dynamically, they cannot exceed this static creation value.

Thread-Switching Time

The switching time for suspending one thread and resuming another is a constant.

Preemption

Preemption is the act of revoking CPU resources from a lower-priority thread when a higher-priority thread is ready for execution. Preemption can be enabled or disabled on an individual thread basis using the RZKCreateThread API (see the RZKCreateThread API definition on page 60). When preemption is disabled, no other thread can execute until the current thread suspends, blocks or enables preemption, or until the thread deletes itself.

Yield

Yield is a mechanism provided to a thread to voluntarily relinquish a processor resource to other ready threads at the same priority level. The thread that yields executes or resumes after all such ready threads at its priority level utilize the CPU resource per the allowed quantum scheduling policies, or until they are no longer in a running state.

Time Slicing

Time slicing (round-robin scheduling) is a CPU resource sharing mechanism, provided by RZK, whereby a thread executes up to a specific time interval prior to relinquishing the CPU resource to the next ready thread. A *time slice* is the maximum number of time ticks that occur before control is ceded to the next ready-to-run thread at the same priority level. A thread using time slicing can be preempted – before its time slice is completed – by a higher-priority thread.

Autostart

The *autostart* option, if active at the time of thread creation, allows the thread to start running as soon as it is created – provided that no other higher-priority thread is executing.

Timers

A *timer* is an optional RZK object. Timer objects invoke user-supplied functions that are to be processed at periodic intervals. The Timer object scheduling takes place through the hardware timer tick. Because the timer object runs certain functions periodically, it runs from the timer interrupt routine.

These periodically-run functions cannot make a blocking object call, and they use the RZK timer interrupt thread stack to run. The timer functions can be enabled or disabled dynamically.

Ticks

A *tick* is the basic unit of time for all RZK timer facilities. For more details about the tick rate, see Appendix C. Interrupt Handling on page 359.

Margin of Error

A timer request can be satisfied by as much as one tick early in actual time. A tick can occur immediately after the timer request. The first tick of a timer request represents an actual duration of time, ranging from zero to the rate of the hardware timer interrupt. For example, the actual time elapsed for a request of *n* ticks falls between the actual time *n* and *n–1* ticks present.

Hardware Requirement

RZK timer services require a periodic timer interrupt from the hardware. Without this interrupt, timer facilities cannot function. Other RZK facilities are not affected by the absence of timer facilities. For more details, see Appendix C. Interrupt Handling on page 359.

Interprocess Communication Mechanisms

RZK provides several objects for the purpose of interprocess communication and synchronization. These objects are message queues, event groups and semaphores. Synchronization of threads is required when two or more threads share a common area of memory or resources.

Message Queues

The message queue object provides a mechanism to transfer multiple bytes of information or messages. Two or more threads can use a message queue to communicate with each other asynchronously. The length of each message is variable but cannot exceed the specified maximum size of the message when the message queue is created.

Message contents are user-defined. Messages are added to the end of the queue by default; however, there is a facility to insert a message at the head of the queue. Messages are retrieved only from the head of the queue.

The message queue is an optional RZK object. Because the message queue is a multiple message-based interprocess communication (IPC) mechanism, it is not directed toward a specific thread. The message queue handler contains a unique value and points to the static message queue *control block*.

You can also identify each message queue by name. This name must be a fixed-length ASCII string. To maintain the message queue control block's compact size, the name field is omitted for nondebug mode. RZK provides a fixed-length message queue storage area.

You must allocate necessary memory for the message queue object created. In addition, all post operations result in a copy operation; i.e., the contents of the user-supplied message input parameter in a send operation is copied to the message queue object, and vice versa, in a pend operation. Therefore, a sender can reuse the buffer upon returning from the post API.

If a message pointer is passed, ensure that the message buffer is globally accessible and shareable between the sender and the recipient. In addition, you must also ensure the safety of the buffer from untimely or accidental modification or deletion. You must also design an independent mechanism for allocation and deallocation of these message buffers.

Static Creation

As with all RZK objects, you must decide the maximum number of message queue objects that can be created. This static value is referred to as *static creation*. Therefore, although RZK objects are created and deleted dynamically, they cannot exceed this static creation value.

Blocking

Threads can be blocked on a message queue for several reasons. A thread attempting to receive a message from an empty message queue can be blocked. A thread attempting to send a message to a full queue can be blocked. A blocked thread is resumed when the message queue is able to satisfy that thread's request. Threads are blocked in either FIFO or PRI-

ORITY order, depending on the option with which the message queue is created. If the queue is blocked with the FIFO option, the threads are resumed in the order in which they were blocked. If the PRIORITY option is supported, threads are resumed in priority order, from highest to lowest priority.

You can choose not to block on message queues to receive or send message queue messages. When this option is chosen, a TIMEOUT error occurs if the message queue is empty while receiving, or full while sending messages.

Processing Time

The processing time for sending and receiving a message in a message queue is constant, but the processing time required to copy a message is relative to the message size.

However, the processing time for sending and receiving a message increases if there are number of threads trying to access the same resource (message queue). The increase in processing time depends on whether the executing thread is created with the PRIORITY option or the FIFO option. With the FIFO option, the processing time depends on the number of threads already blocked on the message queue. With the PRIORITY option, the processing time depends on the thread's priority level.

Semaphores

RZK provides semaphores and event groups for thread synchronization purposes. Thread synchronization is used when two or more threads share a common area of memory or resources.

A semaphore is an optional RZK object. It is the only object requiring a priority inheritance protocol and is tightly coupled with scheduling methods.

Priority inheritance is a method of solving priority inversion problems. Priority inversion occurs when a high-priority thread is starved of a par-

ticular CPU resource because a low-priority thread that owns the resource (in this case, semaphore) and retains exclusive use of that resource, is blocked for various reasons. One of these reasons is that a medium-priority thread preempts it and begins executing.

With priority inheritance, this low-priority thread that owns the semaphore temporarily inherits the highest priority, allowing it to execute completely, release the resource and return to its original priority level. The higher-priority thread that requires the same resource (in this case, the semaphore) acquires it as soon as it is released from the low-priority thread. Priority inheritance is supported for the binary semaphore. For more information about how to enable the priority inheritance protocol in RZK, refer to the Zilog Real-Time Kernel User Manual (UM0075).

As a semaphore uses a mutual-exclusion mechanism (such as IPC), it is not directed to a specific thread. The semaphore handle is a unique value that points to a static semaphore control block. Each semaphore is also identified by a name. This name is a fixed-length ASCII string. To maintain the compact size of the semaphore control block, the name field is omitted for nondebug mode.

RZK implements semaphore behavior in the following manner:

1. A semaphore created by Thread A can be acquired or released by Thread B if the semaphore is any of the following types:
   - A counting semaphore with a receiving order of priority or FIFO
   - A binary semaphore with a receiving order of FIFO
   - A binary semaphore with a receiving order of PRIORITY but does not use a priority inheritance algorithm
2. Thread B can delete a semaphore waiting to be acquired by Thread A. In this case, an error called `RZKERR_OBJECT_DELETED` is returned from the `RZKAcquireSemaphore()` API in Thread A.
3. When a task is deleted, no other task is notified about this task deletion. The `RZKThreadLockForDelete()` and the `RZKThreadUn-`

`LockForDelete()` routines address problems that originate from an unexpected deletion of the thread. The `RZKThreadLockForDelete()` routine protects the thread from deletion by other threads. This protection is often required when a thread executes in a critical region or engages a critical resource.

For example, a thread may take a semaphore for exclusive access to data. When executing inside the critical region, the thread can be deleted by another thread. Because the thread is not able to complete execution in the critical region, the data may be corrupt or inconsistent. As the semaphore can never be released by the thread, the critical resource is now unavailable for use by any other thread and is therefore frozen.

The `RZKThreadLockForDelete()` API protects the thread that acquires the semaphore and prevents an outcome in which the thread is frozen. Any thread that tries to delete a thread protected by the `RZKThreadLockForDelete()` API is returned with an error.

When a critical resource is exhausted, the protected thread can become available for deletion by calling the `RZKThreadUnLockForDelete()` API. To support nested deletion-safe regions, a count of the number of times the APIs `RZKThreadLockForDelete()` and `RZKThreadUnLockForDelete()` are called is maintained. Deletion is allowed only when this count is zero; i.e., there are as many unlocks as there are locks. Protection operates only on the calling thread. A thread cannot prepare another thread to be safe or unsafe from deletion.

The following code snippet illustrates the usage of the `RZKThreadLockForDelete()` and `RZKThreadUnLockForDelete()` APIs:

```
ThreadBody()
{
RZKThreadLockForDelete();
RZKAcquireSemaphore(sem_handle);
// Critical region
RZKReleaseSemaphore(sem_handle);
```

```
RZKThreadUnLockForDelete();
}
```

Note: A binary semaphore cannot be acquired or released from an ISR because the ISRs do not have a fixed task context. ISRs run in the context of the currently-running thread. When called within an ISR, the acquire/release APIs return an error.

Static Creation

You must decide the maximum number of semaphore objects that can be created. This number is referred to as a *static creation value*. Therefore, although RZK objects are created and deleted dynamically, they cannot exceed this static creation value.

Blocking

A thread trying to acquire a semaphore can become blocked when the number of threads already with semaphore(s) equals the initial count. Threads can get blocked in either FIFO or PRIORITY order, depending on the option with which the semaphore is created. If a semaphore is blocked with the FIFO option, threads are resumed in the order in which they were blocked. If the PRIORITY option is supported, threads are resumed from highest to lowest priority.

You can also specify threads not to block on a required semaphore. When this option is chosen, a TIMEOUT error occurs.

Deadlock

A deadlock refers to a situation in which two or more threads are infinitely blocked on the semaphore. For example, a system contains two threads and two semaphores. The first thread occupies the second semaphore and the second thread occupies the first semaphore. If the second

thread tries to occupy the second semaphore and the first thread tries to occupy the first semaphore, both block for an indefinite duration on the requested semaphore. Prevention is the best way to deal with such situations: specifying that threads must not possess more than one semaphore at a time prevents a deadlock from occurring. Deadlocks can also be prevented if the order of acquiring multiple semaphores is the same for all threads.

Processing Time

The processing time for acquiring and releasing a semaphore is constant. However, the processing time increases if there are a number of threads are trying to access the same resource (semaphore). The increase in processing time depends on whether the executing thread is created with the PRIORITY option or the FIFO option. With the FIFO option, the processing time depends on the number of threads already blocked on the message queue. With the PRIORITY option, the processing time depends on the thread's priority level.

Event Groups

Event groups are optional RZK objects. Events within an event group can be masked but cannot be counted. They provide control synchronization and do not carry any information. Events are directed to specified threads and can be operated logically. Event groups provide an efficient mechanism to communicate that a certain system event occurred, such as the availability of data at the drive.

An event group accommodates a maximum of 24 events. Event groups are not thread-specific, and can be directed to the selected thread's group. The event group handle is a unique value that points to a static *event group control block*. Each event group is also identified with a name that is a fixed-length ASCII string. To keep the event group control block size compact, the name field is omitted for nondebug mode. The event group control block contains the following elements:

- Pending event
- Mask
- List of blocked threads

Posting to an event group is equivalent to selectively broadcasting control messages.

Static Creation

You must decide the maximum number of event group objects that can be created. This number is referred to as a *static creation value*. Therefore, although RZK objects are created and deleted dynamically, they cannot exceed this static creation value.

Blocking

Blocking occurs when a thread attempts to receive a combination of event flags that are yet to occur. The thread resumes only after all or any of the expected events occur depending upon the logical operation specified.

You can specify threads not to block on events or event groups. When this option is chosen, a TIMEOUT error occurs if an event group fails to receive expected events.

Processing Time

The processing time for pending an event group is constant. The number of threads blocked on the event group affect the processing time required for posting to an event group.

Memory Management

Memory objects are optional RZK objects. However, they are required for dynamic memory allocation by other objects (for example, message

queues). Memory objects that are allocated by a thread using a global variable are not exclusive to that thread; this memory block can also be utilized by other threads. The handle to the memory object is unique, and points to the static memory control block.

Unused system memory is organized into two categories based on their size: *fixed* and *variable*.

Partitions

Fixed-size memory blocks are known as *partitions*, which are helpful for deterministic memory allocation time. However, the disadvantage of partitions is the potential waste of memory when the required memory is less than the partition segment size.

Regions

Variable-size memory blocks are known as *regions*. Regions are helpful for allocating variable-size memory dynamically. The memory allocation time using regions is not deterministic because the regions must keep track of the already-allocated and unallocated memory. It is advantageous to use variable-size memory because it makes efficient use of available memory.

Static Creation

You must decide the maximum number of memory objects that can be created. This number is referred to as a *static creation value*. Therefore, although RZK objects are created and deleted dynamically, they cannot exceed this static creation value.

Dynamic Creation

RZK memory objects are created and deleted dynamically within static creation values. Memory can be dynamically allocated and deallocated within a created memory object.

Interrupts

Interrupt objects are the basic entities for a multithreading kernel and provide concurrent and independent execution of application threads. Interrupts are generated by synchronous devices, such as a timer, as well as by asynchronous devices such as a keyboard or a mouse.

After a system reset, system interrupt tables display undetermined contents. Processor interrupt handling is normally disabled after a processor reset. Processor interrupts must be disabled to protect the RZK data structures. The time taken between an interrupt disable and an interrupt enable can affect interrupt latency and context switching times.

RZK normally operates with interrupts enabled, and provides APIs to disable or enable interrupts to protect critical sections in the code. You can also call RZK APIs from within an interrupt service routine (ISR). The ISR runs to completion by itself and, at the end of execution, it passes the control to the Scheduler API, if required.

Application Interrupt Lockout

RZK provides users the ability to disable and enable interrupts. When a user application disables an interrupt, an application interrupt lockout occurs. An interrupt locked out by an application remains locked until the application unlocks it.

Device Driver Framework

RZK features a basic device driver framework (DDF) that facilitates communication with I/O devices. This DDF provides a standard driver interface in addition to an abstraction layer that can be used for communication with the device. The RZK DDF is generic in nature and supports different devices that exhibit different characteristics.

As Figure 3 illustrates, the DDF functions as an abstraction layer between the application and the hardware device. The RZK DDF provides a standard set of APIs for I/O operations.

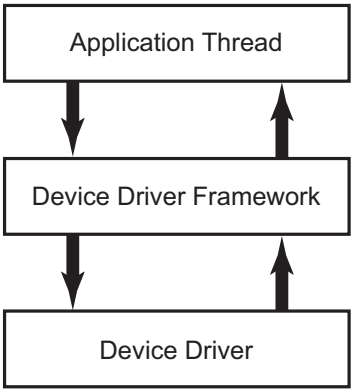

Figure 3. I/O System Organization

Observe the following procedure to add a new device to the DDF and to begin using it.

1. Attach the required new device to the device driver table. To do so, the user application must call the `RZKDevAttach()` function. A call to this function adds the control block for the specific device to the device driver table and calls the device-specific initialization function.
2. After the device is attached to the system's device driver table, open the device by invoking the `RZKDevOpen()` function. A call to this function opens the device for communication by performing the necessary initialization operations and by reserving the resources.
3. After opening the device, I/O operations can be performed by invoking the `RZKDevRead()` or the `RZKDevWrite()` APIs.

4. The device control parameters can be changed using the `RZKDevIOCTL()` API.
5. After I/O operations are complete, the device can be closed by invoking the `RZKDevClose()` API. A call to this function releases the resources used by the device driver.
6. If required to clear the device control block to accommodate other devices, a device can be detached from the DDF by invoking the `RZKDevDetach()` API.

Note: The steps listed above are only guidelines for using a device driver that conforms to DDF.

Device Driver Table

The RZK Driver Framework calls device drivers by using the I/O device driver table; the user supplies the driver table within the confines of the device driver table structure. The driver table contains pointers to device driver entry points. The `RZK_DEVICE_CB_t` structure describes the device driver table.

The user typically enters driver routines in this structure so that they are stored in the DDF and so that any references to this driver are for appropriate operations; the corresponding driver routines are called. For more details, see the Device Driver Framework section on page 23.

A typical example for a device driver block is provided in the following routine.

```
RZK_DEVICE_CB_t Serial0Dev =
{
  RZK_FALSE,
  "SERIAL0",
  UARTInit,
```

```
  (FNPTR_RZKDEV_STOP)IOERR,
  UARTOpen,
  UARTClose,
  UARTRead,
  UARTWrite,
  (FNPTR_RZKDEV_SEEK)IOERR,
  UARTGetc,
  UARTPutc,
  (FNPTR_RZKDEV_IOCTL)UARTControl,
  (RZK_PTR_t)uart0isr,
  0000,
  (UINT8*)&Uart0_Blk,
  0,
  0
};
```

How to Write a Device Driver Using DDF

The DDF provides a simple wrapper over the actual device driver calls. Observe the following procedure to write a device driver using DDF:

1. Write an init routine with a prototype of: `DDF_STATUS_t My_Driver_Init(struct RZK_DEVICE_CB_t *pdev)` that initializes the device.
2. The operation of opening the device can be performed by writing the following routine:

   ```
   DDF_STATUS_t My_Driver_Open ( struct
   RZK_DEVICE_CB_t *pdev,RZK_DEV_NAME_t
   *devName,RZK_DEV_MODE_t *devMode);
   ```

3. Close the device by writing the following string:

   ```
   DDF_STATUS_t My_Driver_Close(struct RZK_DEVICE_CB_t
   *pdev)
   ```

Note: This driver read/write routine is valid only after the driver is opened.

4. Write the read and write operations with the function prototypes to read/write a single byte, or multiple bytes, to or from the driver using the function prototypes `FNPTR_RZKDEV_READ`, `FNPTR_RZKDEV_WRITE`, `FNPTR_RZKDEV_GETC`, or `FNPTR_RZKDEV_PUTC`.
5. Write the `IOCTL` routine that sets the custom information for the device with a prototype of `FNPTR_RZKDEV_IOCTL`.
6. Write a device driver control block that contains information about the different driver routines that must be called.
7. Call the `RZKDevAttach()`, `RZKDevOpen()`, `RZKDevRead()`, `RZKDevWrite()`, `RZKDevIOCTL()`, `RZKDevClose()`, `RZKDevDetach()` routines sequentially, with appropriate parameters to cause a complete life-cycle of a driver initialization, from open through read/write to close and to detach the device driver from RZK.

Notes: Review the sample drivers implemented using the DDF for more information. For more information about the DDF APIs, see the Device Driver Framework section on page 39.

Zilog recommends using DDF APIs such as `RZKDevOpen()`, `RZKDevClose()`, `RZKDevRead()`, `RZKDevWrite()` and `RZKDevIOCTL()` instead of driver routines such as `UARTOpen`, `UARTClose`, `UARTRead`, `UARTWrite` and `UARTControl` for the UART device. This method of calling the DDF APIs instead of calling device-specific APIs facilitates the application to be device independent. The application can also function with different versions of the product and conceal the changes made to device-specific APIs (if any).

Sample Device Drivers That Use DDF

The following device drivers, provided with RZK, are written with respect to the DDF guidelines discussed in the previous section.

These drivers are provided in the board support package (BSP) and are described in the following sections.

Serial Peripheral Interface Device Driver

The serial peripheral interface (SPI) device driver is developed using the DDF implementation. The SPI device driver performs the following tasks:

- Supports automatic poll/interrupt mode operation for data transfer/reception
- Supports full duplex data transfer
- Supports byte transfers of data between SPI-compatible master and slave devices
- Supports a facility for connecting two slave devices

The SPI device driver only transfers data of the binary type, and the mode of transfer is poll/interrupt mode. This transfer mode is determined dynamically by the driver based on the baud rate.

The SPI device driver that is packaged in this release has been tested – at up to 2 KB of data at a data rate of 3 Mbps – for the following two master/slave combinations:

1. To test the eZ80® CPU as master: eZ80® CPU (master)/25C160 EEPROM (slave)
2. To test the eZ80® CPU as a slave: eZ80® CPU (master)/eZ80® CPU (slave)

Caution: If, in the second Master/Slave combination above, any other processor is used as the master, precautions must be taken. The master device in this scenario must pull the eZ80® CPU's Slave Select (SS) pin High for approximately 60 microseconds before it can start sending data. At the data rate outlined above, the eZ80® CPU slave can then respond to the master. Additionally, before sending data bytes, approximately 10 microseconds' delay time must be provided.

Universal Asynchronous Receiver/Transmitter Driver

The UART driver enables you to communicate over the serial interface. The UART driver provided with the RZK BSP is based on the RZK DDF. Therefore, the use model for the UART driver is same as that for the RZK DDF. For more details about its configuration, refer to the Zilog Real-Time Kernel User Manual (UM0075). For more information about UART APIs and details about their operation, see the Universal Asynchronous Receiver/Transmitter APIs section on page 243.

The UART driver supports the following features:

Reentrant APIs. These APIs are designed for multithreaded environments. The critical sections of the code are protected so that they are accessed exclusively.

Fully Configurable. The driver is fully configurable, and users can change the default values of the configurable parameters to suit application and memory requirements.

In addition, the following two features are supported by this UART driver:

- Synchronous and asynchronous data communication
- Hardware flow control using RTS/CTS

Ethernet Media Access Control Driver

The EMAC driver is based on RZK DDF. It manages the Data Link layer and communicates with the upper layers through a well-defined interface. For more details about its configuration, refer to the Zilog Real-Time Kernel User Manual (UM0075). For more information about EMAC APIs and details about their operation, see the Ethernet Media Access Control APIs section on page 223.

The EMAC driver includes the following features:

Configurability. The EMAC driver is fully configurable, and users can change the default values of the configurable parameters to suit application and memory requirements.

Memory Management. The EMAC driver provides a very basic form of memory management for received packets.

The EMAC driver provides the following additional support features:

- Functions on 10/100 BT networks supporting half/full duplex communication
- Currently supports the eZ80F91 MCU's Realtek and CS8900 EMAC controllers

- On the eZ80F91 MCU, The EMAC driver supports the AMD and Micrel PHYs
- Supports multicast addressing

Wireless Local Area Network Driver

The Wireless Local Area Network Driver (WLAN) is developed as per the RZK DDF. The WLAN driver is written for the RTL8711 chipset.

The WLAN driver can be configured either to transfer unencrypted data or to use a security mechanism that will encrypt all wireless data. The WLAN driver supports the following security mechanisms:

- 64-bit Wired Equivalent Privacy (WEP)
- 128-bit WEP
- Wi-Fi Protected Access (WPA)
- Wi-Fi Protected Access II (WPA2)

Inter-Integrated Circuit Driver

The I^2C driver provides a generic framework that can be used to develop more specialized applications that can communicate over the I^2C bus. The I^2C driver, which is based on the DDF model, initializes the I^2C bus for communication and also handles the protocols for Transmit and Receive operations within the APIs. Users must configure the I^2C for parameters such as slave address, bus speed and subnet address, then send or receive data over the bus. For more details about I^2C configuration, refer to the Zilog Real-Time Kernel User Manual (UM0075). For more information about I^2C APIs and details about their operation, see the Inter-Integrated Circuit APIs section on page 282.

The I^2C driver features the following elements:

- Supports master and slave mode
- I^2C protocol handling within the API

- Fully configurable

Real-Time Clock Driver

The Real-Time Clock (RTC) driver can be used to keep time in a system. Input to the RTC can be via crystal oscillator or power line frequency. When an alarm is enabled, the RTC driver generates an interrupt at a specified time. For more details about its configuration, refer to the Zilog Real-Time Kernel User Manual (UM0075). For more information about RTC APIs and details about their operation, see the Real-Time Clock APIs section on page 263.

The Real-Time Clock can be programmed to get and set time through the RZK DDF common interface. The Real-Time Clock includes the following features:

- Reentrant APIs
- Configurability
- Alarm support

Watchdog Timer Driver

RZK features Watchdog Timer (WDT) driver support. WDT is used in situations wherein system response is not observed for durations of minutes/hours/days, and is used as an option to maintain integrity in the system. The WDT resets the system whenever a time-out occurs; such time-outs are specified in the WDT registers.

A value written into the WDT Control Register explicitly resets the WDT. Implicitly, WDT is reset after the `RZKIsrProlog` and `RZKIsrEpilog` functions are invoked. For information about WDT APIs, see the Watchdog Timer APIs section on page 299.

RZK APIs

This chapter describes the APIs that are provided by RZK, including DDF APIs.

RZK API Summary

The RZK APIs described in this chapter are grouped based on their usage, as shown by the list below.

Kernel Startup

Kernel start-up APIs are used to initialize RZK and start the kernel for the execution of application threads. Table 4 lists the kernel start-up APIs that are described in this subsection.

Table 4. Kernel Start-Up API

RZK_KernelInit	RZK_KernelStart

Thread Control

Thread APIs are used to create threads and perform the following operations on a created thread:

- Suspend a thread finitely or infinitely
- Resume a thread
- Delete a thread
- Change the priority of a thread
- Yield control to other threads
- Get thread parameters
- Get scheduler parameters

Table 5 lists the thread control APIs.

Table 5. Thread Control APIs

RZKCreateThread	RZKDeleteThread
RZKCreateThreadEnhanced	RZKDeleteThreadEnhanced
RZKDeleteThreadEnhanced	RZKSetThreadPriority
RZKYieldThread	RZKGetSchedulerParameters
RZKGetThreadParameters	RZKResumeThread

Thread Communication

The following operations are possible on a message queue using the thread communication/message queue APIs.

- Posting messages to a message queue – the post time-out can range from immediate to infinite

- Receiving a message from a message queue – the pending time-out can range from immediate to infinite
- Posting/inserting a message to the front of a message queue
- Deleting a message queue
- Getting message queue parameters
- Peeking into a message queue and copying a message, if present
- Sending a unique message to the queue

The message queue APIs are listed in Table 6.

Table 6. Thread Communication (Message Queue) APIs

RZKCreateQueue	RZKDeleteThread
RZKSendToQueueFront	RZKSendToQueue
RZKPeekMessageQueue	RZKReceiveFromQueue
RZKGetQueueParameters	RZKSendToQueueUnique

Thread Synchronization

Thread synchronization APIs provide access to RZK semaphores and event flags.

Semaphore APIs

Semaphore APIs are used to synchronize two threads, provide mutual exclusion during access to shared resources, and prevent interference from other threads while modifying critical sections of a shared resource such as a database. They can also be used to protect global variables that are shared across threads. A semaphore can be created as either a *binary* semaphore or a *counting* semaphore. The following operations can be performed on a created semaphore:

- Deleting an unused semaphore
- Acquiring the semaphore
- Releasing the semaphore
- Getting the semaphore parameters

Event Group APIs

Event group objects provide a mechanism to convey that a certain system activity has occurred. Events can be grouped and can be logically operated upon. The following operations can be performed on the event groups:

- Deleting an unused event group
- Posting the event to an event group
- Pending on an event group with timed pend and infinite pend
- Getting the event group parameters

Table 7 lists the semaphore and event group APIs.

Table 7. Semaphore and Event Group APIs

RZKCreateSemaphore	RZKDeleteSemaphore
RZKReleaseSemaphore	RZKAcquireSemaphore
RZKCreateEventGroup	RZKGetSemaphoreParameters
RZKPostToEventGroup	RZKDeleteEventGroup
RZKGetEventGroupParameters	RZKPendOnEventGroup

Software Timer

Timer APIs provide access to RZK timer facilities. Timer APIs are helpful when a user-supplied function must be executed at periodic intervals. The following operations can be performed on the software timers:

- Creating timers with a user-supplied timer handler function
- Deleting the timer
- Enabling the timer
- Disabling the timer
- Getting the parameters of a timer object
- Get timer resolution
- Getting system clock value
- Setting system clock value

Table 8 lists the Timer APIs.

Table 8. Timer APIs

RZKCreateTimer	RZKDeleteTimer
RZKEnableTimer	RZKDisableTimer
RZKGetTimerParameters	RZKGetClock
RZKGetTimerResolution	RZKSetClock

Memory Management

The memory APIs consist of partition and region APIs and provide access to RZK memory management facilities.

Partition APIs

Partitions are useful for storing fixed-size data and are also helpful for determining memory allocation time. The following operations can be performed on partitions:

- Creating a partition with maximum number of blocks by specifying the size of each block
- Allocating a memory block in partition
- Freeing the allocated memory block in partition
- Deleting the unused partition
- Getting the partition parameters

Region APIs

These objects are similar to partitions but the memory that can be allocated from a region is of variable size. The following operations can be performed on regions:

- Allocating a variable length of memory block
- Freeing the allocated block
- Deleting the unused region
- Getting the region parameters
- Initialize malloc
- Allocate memory using malloc
- Free allocated memory

- Query for available memory

Table 9 lists the partitions and region APIs.

Table 9. Partition and Region APIs

RZKCreatePartition	RZKDeletePartition
RZKAllocFixedSizeMemory	RZKFreeFixedSizeMemory
RZKGetPartitionParameters	RZKDeleteRegion
RZKCreateRegion	RZKAllocSegment
RZKFreeSegment	RZKGetRegionParameters
malloc	free
RZKQueryMem	

Interrupt Management

Interrupt APIs provide access to RZK interrupt management facilities and are the main source for external and internal events. To protect shared data in a multithread kernel, interrupts are disabled. After an operation upon shared data is complete, interrupts are brought to their previous state. The following operations are performed on interrupts:

- Installing the interrupt handler for the interrupt
- Disabling the interrupt before accessing/modifying the critical data
- Enabling the interrupts after the critical data is modified/accessed
- Calls to be included while designing the RZK for ISR (making RZK calls from inside an ISR)
- Resume interrupt thread
- Suspend interrupt thread

Table 10 lists the interrupt APIs.

Table 10. Interrupt APIs

RZKInstallInterruptHandler	RZKEnableInterrupts
RZKDisableInterrupts	RZKISRProlog
RZKISREpilog	RZKResumeInterruptThread
RZKSuspendInterruptThread	

Device Driver Framework

DDF APIs are used to call hardware-specific driver routines which are present in another global table. The following operations can be performed using DDF APIs.

- Opening a specified device
- Closing a specified device
- Reading a specified number of bytes from the device
- Writing specified number of bytes to the device
- Controlling the device-specific hardware using an API

Table 11 lists the DDF APIs.

Table 11. Device Driver Framework APIs

RZKDevOpen	RZKDevClose	RZKDevIOCTL
RZKDevWrite	RZKDevRead	RZKDevPutc
RZKDevGetc	RZKDevAttach	RZKDevDetach

Miscellaneous APIs

A number of miscellaneous APIs are used to obtain information about the threads and the statistics of threads and timers. The following operations can be performed using these APIs:

- Getting the current executing thread handle
- Getting the error number stored in the TCB, which is set by an RZK API execution
- Getting the arguments list that is passed to a thread
- Formatting an error to print in an error string
- Getting thread execution and timer execution statistics
- Resetting the complete system
- Disable/enable/restore preemption calls
- Get current system time in ticks
- Lock a thread for delete operation
- Unlock a thread from delete operation

Table 12 lists the miscellaneous APIs.

Table 12. Miscellaneous APIs

RZKFormatError	RZKGetCurrentThread
RZKGetErrorNum	RZKGetThreadStatistics
RZKGetTimerStatistics	RZK_Reboot
GetDataPersistence	SetDataPersistence
RZKSystemTime	RZKThreadLockForDelete
RZKThreadUnLockForDelete	RZKDisablePreemption

Table 12. Miscellaneous APIs (Continued)

RZKEnablePreemption	RZKRestorePreemption
FreePktBuff	

Board Support Package APIs

This section summarizes and describes each of the BSP APIs.

Ethernet Media Access Control APIs

Ethernet Media Access Control (EMAC) APIs are used to call specific driver routines that are used to communicate with Ethernet media. The EMAC APIs in Table 13 are called through the DDF interface.

Table 13. Ethernet Media Access Control APIs

EmacOpen	EmacClose	EmacWrite
EmacRead	EmacControl	AddEmac

Wireless Local Area Network APIs

Wireless Local Area Network (WLAN) APIs are used to call specific driver routines that are used to communicate through wireless media. The WLAN APIs in Table 14 are called through the DDF interface. Only eZ80F91 supports WLAN driver.

Table 14. Wireless Local Area Network APIs

AddWlan	wlanWrite	wlanClose
wlanOpen	wlanRead	

Universal Asynchronous Receiver/Transmitter APIs

Universal Asynchronous Receiver/Transmitter (UART) APIs are used to perform I/O operations on a UART device. The UART APIs in Table 15 are called through the DDF interface.

Table 15. Universal Asynchronous Receiver/Transmitter APIs

UARTOpen	UARTClose	UARTWrite
UARTRead	UARTControl	AddUart0
AddUart1		

Real-Time Clock APIs

Real-Time Clock (RTC) APIs are used to set and get time. They also support an alarm function. The RTC APIs in Table 16 are called through the DDF interface.

Table 16. Real-Time Clock APIs

RTCRead	RTCControl	AddRtc

Serial Peripheral Interface APIs

Serial Peripheral Interface (SPI) APIs are used to call SPI specific driver routines which are placed in a global table. The APIs in Table 17 are called through the DDF interface.

Table 17. Serial Peripheral Interface APIs

SPI_Open	SPI_Close	SPI_Write
SPI_Read	SPI_IOCTL	AddSpi

Inter-Integrated Circuit APIs

RZK provides a generic set of inter-integrated circuit (I^2C) APIs to enable you to develop drivers for different types of I^2C slave devices. The APIs in Table 18 are called through the DDF interface.

Table 18. Inter-Integrated Circuit APIs

I2COpen	I2CClose	I2CControl
I2CWrite	I2CRead	AddI2c

Universal Serial Bus Device APIs

USB device APIs are used to call specific driver routines to communicate with a USB host. These USB device APIs are called directly without the DDF interface. See Table 43 on page 295 for a reference to the USB APIs for RZK.

Watchdog Timer APIs

RZK provides a generic set of Watchdog Timer APIs to enable you to reset the system according to user application requirements. The WDT APIs are listed in Table 19.

Table 19. Watchdog APIs

wdt_init	wdt_reset

Flash Device Driver APIs

RZK provides a generic set of Flash Device Driver APIs to enable you to read/write the data from/to the Flash device, or to erase the Flash device. These driver APIs can be directly interfaced to the Zilog File System to store files in the corresponding Flash device. These driver APIs do not

comply with the RZK Device Driver framework, and are implemented as stand-alone APIs that can be invoked in the program.

The Flash Device Driver APIs are listed in Table 20.

Table 20. Flash Device Driver APIs

FLASHDEV_Init	FLASHDEV_Read
FLASHDEV_Erase	FLASHDEV_Close
FLASHDEV_Write	

In this table, FLASHDEV represents one of the following devices:

- MT28F008
- AT49BV162
- AM29LV160
- IntFlash

RZK APIs and Context Switching

Table 21 contains the key for the RZK API context-switching capability listings shown in Table 22.

Table 21. RZK Context Switching Key

+	Context switching is possible.
–	Context switching is not possible.
±	Context switching is possible under certain circumstances.

Table 22. RZK APIs and Context Switching

RZK API	Context Switch
RZKCreateThread[1]	±
RZKDeleteThread[2]	±
RZKCreateThreadEnhanced	±
RZKDeleteThreadEnhanced	±
RZKSuspendThread	+
RZKResumeThread	+

Notes:

1. Context switching occurs only if the created thread holds a higher priority than the thread that created it and the AUTOSTART attribute is present. One exception to this rule is when main() runs at the highest priority.
2. Context switching is conditional. If any RZK API, which can cause a context switch, is called within the `RZKDeleteThread()`'s clean-up function, context switching is delayed until the delete operation is completed.

Table 22. RZK APIs and Context Switching (Continued)

RZK API	Context Switch
RZKGetThreadParameters	–
RZKYieldThread	+
RZKGetSchedulerParameters	–
RZKCreateQueue	–
RZKDeleteQueue	+
RZKSendToQueue	+
RZKSendToQueueFront	+
RZKReceiveFromQueue	+
RZKGetQueueParameters	–
RZKPeekMessageQueue	–
RZKCreateSemaphore	–
RZKDeleteSemaphore	+
RZKAcquireSemaphore	+
RZKReleaseSemaphore	+
RZKGetSemaphoreParameters	–
RZKCreateEventGroup	–
RZKDeleteEventGroup	+

Notes:

1. Context switching occurs only if the created thread holds a higher priority than the thread that created it and the AUTOSTART attribute is present. One exception to this rule is when main() runs at the highest priority.
2. Context switching is conditional. If any RZK API, which can cause a context switch, is called within the `RZKDeleteThread()`'s clean-up function, context switching is delayed until the delete operation is completed.

Table 22. RZK APIs and Context Switching (Continued)

RZK API	Context Switch
RZKPostToEventGroup	+
RZKPendOnEventGroup	+
RZKGetEventGroupParameters	–
RZKCreateTimer	–
RZKDeleteTimer	–
RZKEnableTimer	–
RZKDisableTimer	–
RZKGetTimerParameters	–
RZKGetTimerResolution	–
RZKSetClock	–
RZKGetClock	–
RZKCreatePartition	–
RZKDeletePartition	–
RZKAllocFixedSizeMemory	–
RZKFreeFixedSizeMemory	–
RZKGetPartitionParameters	–
RZKCreateRegion	–

Notes:

1. Context switching occurs only if the created thread holds a higher priority than the thread that created it and the AUTOSTART attribute is present. One exception to this rule is when main() runs at the highest priority.
2. Context switching is conditional. If any RZK API, which can cause a context switch, is called within the `RZKDeleteThread()`'s clean-up function, context switching is delayed until the delete operation is completed.

Table 22. RZK APIs and Context Switching (Continued)

RZK API	Context Switch
RZKDeleteRegion	+
RZKAllocSegment	+
RZKFreeSegment	+
RZKGetRegionParameters	–
RZKInstallInterruptHandler	–
RZKEnableInterrupts	–
RZKDisableInterrupts	–
RZKFormatError	–
RZKGetTimerStatistics	–
RZKGetThreadStatistics	–
RZKGetCurrentThread	–
RZKGetErrorNum	–
RZKDevOpen	–
RZKDevClose	–
RZKDevIOCTL	–
RZKDevWrite	±
RZKDevRead	±

Notes:

1. Context switching occurs only if the created thread holds a higher priority than the thread that created it and the AUTOSTART attribute is present. One exception to this rule is when main() runs at the highest priority.
2. Context switching is conditional. If any RZK API, which can cause a context switch, is called within the `RZKDeleteThread()`'s clean-up function, context switching is delayed until the delete operation is completed.

Table 22. RZK APIs and Context Switching (Continued)

RZK API	Context Switch
RZKDevPutc	±
RZKDevGetc	±
RZK_Reboot	–
RZKDisablePreemption	–
RZKRestorePreemption	±
RZKEnablePreemption	±
RZKSuspendInterruptThread	±
RZKResumeInterruptThread	±
RZKSystemTime	–
wdt_init	–
wdt_reset	–
RZK_KernelInit	–
RZK_KernelStart	+
malloc	±
free	±
RZKSetThreadPriority	±
RZKThreadLockForDelete	–

Notes:

1. Context switching occurs only if the created thread holds a higher priority than the thread that created it and the AUTOSTART attribute is present. One exception to this rule is when main() runs at the highest priority.
2. Context switching is conditional. If any RZK API, which can cause a context switch, is called within the `RZKDeleteThread()`'s clean-up function, context switching is delayed until the delete operation is completed.

Table 22. RZK APIs and Context Switching (Continued)

RZK API	Context Switch
RZKThreadUnLockForDelete	–
RZKSendtoQueueUnique	±
RZKQueryMem	–

Notes:

1. Context switching occurs only if the created thread holds a higher priority than the thread that created it and the AUTOSTART attribute is present. One exception to this rule is when main() runs at the highest priority.
2. Context switching is conditional. If any RZK API, which can cause a context switch, is called within the `RZKDeleteThread()`'s clean-up function, context switching is delayed until the delete operation is completed.

API Definitions

This section provides detailed descriptions of the APIs available in the Zilog Real-Time Kernel. To use RZK APIs, the `ZSysgen.h` and `ZTypes.h` header files must be included in the application program. Other header files are included as and when necessary.

Note: To maintain solid RZK performance, avoid defining `RZK_DBG`, which performs error-checking logic.

Standard Data Types

Table 23 describes the standard data types that RZK uses.

Table 23. Standard Data Types

Data Type	Description
`unsigned int`	An integer corresponding to the natural word size of the machine. In the eZ80® core, the natural word size is 3 bytes.
`unsigned char`	An 8-bit unsigned character.
`void`	Equivalent to the target compiler's void type.

Include Files

Table 24 lists the header files included in the RZK APIs.

Table 24. Header Files

Header File Name	Description
`ZSysgen.h`	Defines the configurable system parameters.
`ZTypes.h`	Defines the typedefs, macros and enums used by RZK.
`ZThread.h`	Provides the declaration of RZK Thread structures.
`ZTimer.h`	Provides the declaration of RZK Timer structures and APIs.
`ZScheduler.h`	Provides the declaration of RZK Scheduler structures and APIs.
`ZMessageQ.h`	Provides the declaration of RZK message queue structures and APIs.
`ZSemaphore.h`	Provides the declaration of RZK semaphore structures and APIs.
`ZEventgroup.h`	Provides the declaration of RZK event group structures and APIs.
`ZClock.h`	Provides the declaring of RZK clock parameters and APIs.
`ZMemory.h`	Provides the declaration of RZK memory partition structures and APIs.
`ZRegion.h`	Provides the declaration of RZK region structures and APIs.
`ZInterrupt.h`	Provides the declaration of RZK interrupt handling APIs.
`ZDevice.h`	Provides the declaration of RZK device driver framework.
`EtherMgr.h`	Provides the declaring of macros and structure declarations related to EMAC driver.
`Serial.h`	Provides the serial driver macros and structure definitions.
`Dataperstruct.h`	Provides data structures required for data persistence of difference values.

Table 24. Header Files (Continued)

`rtc.h`	Provides the macros and structure declarations for RTC driver.
`spi.h`	Provides the macros and structure declarations for SPI driver.
`i2c.h`	Provides the macros and structure declarations for I^2C driver.
`wdt.h`	Provides the macros and structure declarations for watchdog timer.
`ZThreadstatistics.h`	Provides the declaration of RZK thread statistics structure and APIs.
`ZTimerstatistics.h`	Provides the declaration of RZK timer statistics structure and APIs.

API Definition Format

Descriptions for each RZK API follow a standard format. In this document, header file names are shown just below the function prototype and are followed by the function syntax, its parameters, return values and an example, which is in turn followed by a list of APIs for referencing. A brief discussion of the API description format is provided below.

Include

This section provides the name of the header files included in the API.

Prototype

This section contains the exact declaration of the API call.

Description

This section contains a paragraph describing the API.

Argument

This section describes the arguments (if any) to the API.

Return Value

This section describes the return value of the API, if any.

Example(s)

This section(s) contain examples of how the API function is called.

See Also

This section lists related API calls.

RZK API Quick Reference

Table 25 provides a quick reference to the RZK APIs that are described in this section.

Table 25. RZK API Quick Reference

Kernel Start-Up APIs	Clock APIs
Thread Control APIs	Partition APIs
Scheduler APIs	Region APIs
Message Queue APIs	Interrupt APIs
Semaphore APIs	Watchdog Timer APIs
Event Group APIs	Flash Device Driver APIs
Software Timer APIs	Miscellaneous APIs

Kernel Start-Up APIs

Table 26 provides a quick reference to two kernel start-up RZK APIs that are described in this subsection.

Table 26. Kernel Start-Up API Quick Reference

RZK_KernelInit
RZK_KernelStart

RZK_KERNELINIT

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZK_KernelInit( void ) ;
```

Description

The `RZK_KernelInit()` API initializes the kernel. This API is the first API that must be called in the `main()` function.

Argument(s)

None.

Return Value(s)

None.

Example

```
int main( int argc, void *argv[] )
{
  RZK_KernelInit() ;

  // Application code

  RZK_KernelStart();
}
```

See Also

RZK_KernelStart

RZK_KERNELSTART

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZK_KernelStart( void ) ;
```

Description

The `RZK_KernelStart()` API starts the RZK kernel and executes the application threads. This API is the last API that must be called in the `main()` function. After calling this API, the control never returns to the caller. The control is passed to the scheduler, which schedules the threads for execution.

Argument(s)

None.

Return Value(s)

None.

Example

```
int main( int argc, void *argv[] )
{
  RZK_KernelInit() ;
  // Application code
  RZK_KernelStart();
}
```

See Also

RZK_KernelInit

Thread Control APIs

Table 27 provides a quick reference to a number of thread control APIs that are described in this subsection.

Table 27. Thread Control API Quick Reference

RZKCreateThread	RZKYieldThread
RZKDeleteThread	RZKGetThreadParameters
RZKCreateThreadEnhanced	RZKDisablePreemption
RZKDeleteThreadEnhanced	RZKEnablePreemption
RZKSuspendThread	RZKRestorePreemption
RZKSetThreadPriority	RZKSuspendInterruptThread
RZKResumeThread	RZKResumeInterruptThread

RZKCREATETHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_THREADHANDLE_t RZKCreateThread (
 RZK_NAME_t             szName [MAX_OBJECT_NAME_LEN],
 RZK_PTR_t              pEntryFunction,
 FNP_THREAD_ENTRY       *pCleanupFunction,
 CADDR_t                pInitialStack,
 RZK_THREAD_PRIORITY_t  etPriority,
 TICK_t                 tQuantum,
 RZK_OPERATIONMODE_t    uOperationMode,
 UINT8                  nArgs, .....
)
```

Description

The `RZKCreateThread()` API call is used to create a thread with an entry point function, initial stack, operation mode, time slice for round-robin scheduling, and parameters to be passed to the thread entry function.

Argument(s)

`szName`	Specifies the name of the thread (ASCII string). The length of this thread name must be less than 12 characters.
`pEntryFunction`	Pointer to a entry point function from where the thread starts running. The thread entry function prototype depends on the arguments passed and the types of arguments. Irrespective of any number of parameters passed to the entry point function, the return value must be void.
`pCleanupFunction`	Pointer to a null terminated list of clean up function(s) called when the thread is deleted (global data). The clean-up function prototype must be void MyThreadCleanupFn(void);
`pInitial Stack`	Pointer to the top of the stack of the thread.
`etPriority`	The priority of the thread can range from 1 to 31, where 1 represents the highest priority and 31 represents the lowest priority.
`tQuantum`	The time slice in system ticks for which the thread runs, for round robin mode. If this slice is 0, user-provided default time slice (RZK_TIME_SLICEH) is considered for round-robin time slice.
`uOperationMode`	Specifies the mode of operation of the thread and can be one or a combination of the following values: • RZK_THREAD_ROUNDROBIN: two or more threads with the same priority • RZK_THREAD_AUTOSTART: starts immediately after creation • RZK_THREAD_PREEMPTION: thread can be preempted by higher-priority thread

`nArgs`	Specifies the number of arguments that can be passed to the thread as parameters. After the this, you can pass the parameters that are equal to the nArgs value. Only pointer, (unsigned) char and (unsigned) integer variables type can be passed. RZK does not support any other data types.
`...`	Specifies the arguments that can be passed to the thread's entry point function depending on the value of nArgs.

Return Value(s)

When the thread is created successfully, this API returns a handle to the thread and `RZKERR_SUCCESS` is set in the thread's control block.

If the thread is not created, a `NULL` is returned and the API sets one of the following error values in the current thread's control block. `RZKGetErrorNum()` API can be called to retrieve the error number stored in the thread control block.

`RZKERR_INVALID_STACK`	Indicates that the initial stack pointer is invalid.
`RZKERR_CB_UNAVAILABLE`	Indicates that the control block is unavailable for the allocation. Number of threads exceeds the (MAX_THREADSH−2) value*.
`RZKERR_INVALID_ARGUMENTS`	Indicates that one or more arguments are invalid.
`RZKERR_INVALID_PRIORITY`	Indicates that the specified Priority value is invalid. Valid Priority Range is between 1 and 31.

RZKERR_INVALID_SIZE	Indicates that the length of the thread name is invalid, if it exceeds 12 characters.

Note: *There are two kernel threads launched by RZK (idle thread of lowest priority and timer thread of highest priority). The minimum number for MAX_THREADSH must be 2.

Example 1

A thread is created with the name *Sample*, and the thread entry function is named *ThreadEntry*. No arguments are passed to the thread entry function by the thread, but it passes an array that contains the address of clean-up functions. The thread priority is 15; round-robin ticks = 10. The thread is created in PREEMPTION mode; i.e., this thread can be preempted by other threads in the system. The thread entry function does not accept any parameters.

```
#define THREAD_STACK_SIZE 1024
#define THREAD_PRIORITY 15
#define THREAD_RR_TICKS 10
extern void ThreadCleanup(void);
extern void ThreadEntry( void ) ;
CADDR_t g_threadStack[ THREAD_STACK_SIZE ];

FNP_THREAD_ENTRY g_SampleCleanupfns[] =
{ThreadCleanup, NULL };
RZK_THREADHANDLE_t g_hThreadHandle;

g_hThreadHandle = RZKCreateThread(
  ( RZK_NAME_t[] ) "Sample",
  ThreadEntry,
  g_SampleCleanupfns,
  ( CADDR_t ) (g_threadStack + THREAD_STACK_SIZE),
  THREAD_PRIORITY,
  THREAD_RR_TICKS,
  RZK_THREAD_PREEMPTION,
  0 ) ;
```

Example 2

A thread is created with the name Sample and thread entry function as *ThreadEntry*. Two arguments of type `char` and `int` and values `B` and `4721` are passed to the thread entry function. The thread priority is 15 with round-robin ticks equal to 10. The thread is created in ROUND ROBIN mode with an AUTOSTART option. This thread cannot be pre-empted by other threads during execution.

```
#define THREAD_STACK_SIZE 1024
#define THREAD_PRIORITY   15
#define THREAD_RR_TICKS   10
extern void ThreadCleanup(void);
extern void ThreadEntry( char ch_type, int n_value ) ;
CADDR_t g_threadStack[ THREAD_STACK_SIZE ];

FNP_THREAD_ENTRY g_SampleCleanupfns[] =
  { ThreadCleanup, NULL };
RZK_THREADHANDLE_t g_hThreadHandle;

g_hThreadHandle = RZKCreateThread(
  ( RZK_NAME_t[] ) "Sample",
  ThreadEntry,
  g_SampleCleanupfns,
  ( CADDR_t ) (g_threadStack + THREAD_STACK_SIZE),
  THREAD_PRIORITY,
  THREAD_RR_TICKS,
  RZK_THREAD_ROUNDROBIN | RZK_THREAD_AUTOSTART,
  2,                // Number of parameters
  ( char ) 'B',
  ( int ) 4721 );
```

See Also

RZKDeleteThread
RZKSuspendThread
RZKSetThreadPriority
RZKYieldThread
RZKGetThreadParameters
RZKGetErrorNum
RZK_OPERATIONMODE_t
RZKDisablePreemption

RZKEnablePreemption RZKRestorePreemption
RZKSuspendInterruptThread RZKResumeInterruptThread

RZKCREATETHREADENHANCED

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_THREADHANDLE_t RZKCreateThreadEnhanced (
  RZK_NAME_t            szName [MAX_OBJECT_NAME_LEN],
  RZK_PTR_t             pEntryFunction,
  FNP_THREAD_ENTRY      *pCleanupFunction,
  COUNT_t               uStackSize,
  RZK_THREAD_PRIORITY_t etPriority,
  TICK_t                tQuantum,
  RZK_OPERATIONMODE_t   uOperationMode,
  UINT8                 nArgs, .....
)
```

Description

The `RZKCreateThreadEnhanced()` API call is used to create a thread with an entry point function, stack size, operation mode, time slice for round-robin scheduling and parameters to be passed to the thread entry function. The `RZKCreateThreadEnhanced()` API allocates memory for the thread's stack.

Note: If a thread is created using the `RZKCreateThreadEnhanced()` API, that thread must be deleted only using the `RZKDeleteThreadEnhanced()` API, not using the `RZKDeleteThread()` API.

Argument(s)

`szName`	Specifies the name of the thread (ASCII string). This thread name must be less than 12 characters.
`pEntryFunction`	Pointer to a entry point function from where the thread starts running. The thread entry function prototype depends on the arguments passed and the types of arguments. Irrespective of any number of parameters passed to the entry point function, the return value must be void.
`pCleanupFunction`	Pointer to a null terminated list of clean up function(s) called when the thread is deleted (global data). The clean-up function prototype must be void MyThreadCleanupFn(void);
`uStackSize`	Stack size in number of bytes.
`etPriority`	The priority of the thread can range from 1 to 31, where 1 represents the highest priority and 31 represents the lowest priority.
`tQuantum`	The time slice in system ticks for which the thread runs, for round robin mode. If this slice is 0, user-provided default time slice (RZK_TIME_SLICEH) is considered for round-robin time slice.

`uOperationMode`	Specifies mode of operation of the thread and can be one or a combination of the following values: • RZK_THREAD_ROUNDROBIN: two or more threads with the same priority • RZK_THREAD_AUTOSTART: starts immediately after creation • RZK_THREAD_PREEMPTION: thread can be preempted by higher-priority thread
`nArgs`	Specifies the number of arguments that can be passed to the thread as parameters. After the this, you can pass the parameters that are equal to the nArgs value. Only pointer, (unsigned) char and (unsigned) integer variables type can be passed. RZK does not support any other data types.
`...`	Specifies the arguments that can be passed to the thread's entry point function depending on the value of nArgs.

Return Value(s)

When the thread is created successfully, this API returns a handle to the thread and `RZKERR_SUCCESS` is set in the thread's control block.

If the thread is not created, a `NULL` is returned and the API sets one of the following error values in the current thread's control block. `RZKGetErrorNum()` API can be called to retrieve the error number stored in the thread control block.

RZKERR_INVALID_STACK	Indicates that the initial stack pointer is invalid.
RZKERR_CB_UNAVAILABLE	Indicates that the control block is unavailable for the allocation. Number of threads exceeds the (MAX_THREADSH−2) value.*
RZKERR_INVALID_ARGUMENTS	Indicates that one or more arguments are invalid.
RZKERR_INVALID_PRIORITY	Indicates that the specified Priority value is invalid. Valid Priority Range is between 1 and 31.
RZKERR_INVALID_SIZE	Indicates that the length of the thread name is invalid, if it exceeds 12 characters.

Note: *There are two kernel threads launched by RZK (idle thread of lowest priority and timer thread of highest priority). The minimum number for MAX_THREADSH must be 2.

Example

A thread is created with the name *Sample* and the thread entry function is named *ThreadEntry*. No arguments are passed to the thread entry function by the thread, but it passes an array that contains the address of clean-up functions. The thread priority is 15 with round-robin ticks = 10. The thread is created in PREEMPTION mode, that is, this thread can be preempted by other threads in the system. The thread entry function does not accept any parameters.

```
#define THREAD_STACK_SIZE 1024
#define THREAD_PRIORITY 15
#define THREAD_RR_TICKS 10
extern void ThreadCleanup(void);
extern void ThreadEntry(void);
CADDR_t g_threadStack[ THREAD_STACK_SIZE ];
```

```
FNP_THREAD_ENTRY g_SampleCleanupfns[] =
{ThreadCleanup, NULL};
RZK_THREADHANDLE_t g_hThreadHandle;

g_hThreadHandle = RZKCreateThreadEnhanced(
  ( RZK_NAME_t[] ) "Sample",
  ThreadEntry,
  g_SampleCleanupfns,
  THREAD_STACK_SIZE,
  THREAD_PRIORITY,
  THREAD_RR_TICKS,
  RZK_THREAD_PREEMPTION,
  0 ) ;
```

See Also

RZKDeleteThread	RZKSuspendThread
RZKSetThreadPriority	RZKYieldThread
RZKGetThreadParameters	RZKGetErrorNum
RZK_OPERATIONMODE_t	RZKDisablePreemption
RZKEnablePreemption	RZKRestorePreemption
RZKSuspendInterruptThread	RZKResumeInterruptThread

RZKDELETETHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKDeleteThread(
RZK_THREADHANDLE_t hThread);
```

Description

The `RZKDeleteThread()` API deletes a thread and invalidates the thread control block (TCB). This invalidated control block is allocated to other threads when they are created. The clean-up functions are called first through a clean-up function pointer provided during a `RZKCreateThread()` call and then the thread handle is invalidated.

A thread can delete another thread. If an API is called inside a clean-up function, note that the clean-up function executes from the context of the thread calling `RZKDeleteThread()`. The clean-up functions are used to release the resources that were used by the thread being deleted.

When a thread deletes itself, the delete operation is automatic, implying that any context switching, which occurs as a result of making an RZK API call within the thread's clean-up function, is delayed until the delete operation is complete.

Notes:
1. If `RZK_PRIORITYINHERITANCE` is enabled, the binary semaphore held by the thread is released when the thread is deleted.
2. Any allocated partition /region/acquired semaphore is not freed automatically in the `RZKDeleteThread()` call. You must free these allocated partitions/regions/acquired semaphores in the thread's clean-up function.

3. RZKERR_CB_UNAVAILABLE and RZKERR_CB_BUSY are returned only if RZK_PERFORMANCE is undefined.

Refer to the Zilog Real-Time Kernel User Manual (UM0075) for more details.

Argument(s)

hThread	Handle to the thread to be deleted.

Return Value(s)

RZKERR_SUCCESS	Indicates that the function returned successfully.
RZKERR_INVALID_HANDLE	Indicates that the thread handle to be deleted is invalid or already deleted.
RZKERR_INVALID_OPERATION	Indicates that you tried to delete a thread that is not in DELETED state or contains a locked thread from DELETED state.

Example

A previously-created thread, hThreadHandle is deleted. The thread deletion status is stored in the status variable.

```
/*contains thread handle to be deleted.*/
RZK_THREADHANDLE_t hThreadHandle;
RZK_STATUS_t status;

status = RZKDeleteThread (hThreadHandle);
```

See Also

RZKCreateThread
RZKDeleteThread
RZKSetThreadPriority
RZKYieldThread
RZKGetThreadParameters
RZKDisablePreemption
RZKEnablePreemption
RZKRestorePreemption
RZKSuspendInterruptThread
RZKResumeInterruptThread

RZKDELETETHREADENHANCED

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKDeleteThreadEnhanced(
RZK_THREADHANDLE_t hThread);
```

Description

The `RZKDeleteThreadEnhanced()` API deletes a thread and invalidates the thread control block (TCB). This invalidated control block is allocated to other threads when they are created. The clean-up functions are called first through a clean-up function pointer provided during a `RZKCreateThread()` call and then the thread handle is invalidated.

A thread can delete another thread. If an API is called inside a clean-up function, be aware that the clean-up function executes from the context of the thread calling `RZKDeleteThread()`. The clean-up functions are used to release the resources that were used by the thread being deleted.

When a thread deletes itself, the delete operation is automatic, implying that any context switching, which can occur as a result of making an RZK API call within the thread's clean-up function, is delayed until the delete operation is complete.

The `RZKDeleteThreadEnhanced()` API deallocates memory of the thread stack, which is allocated by the `RZKCreateThreadEnhanced()` API.

Notes: 1. If `RZK_PRIORITYINHERITANCE` is enabled, the binary semaphore held by the thread is released when the thread is deleted.

2. Any allocated partition /region/acquired semaphore is not freed automatically in the `RZKDeleteThreadEnhanced()` call. You must free these allocated partitions/regions/acquired semaphores in the thread's clean-up function.
3. `RZKERR_CB_UNAVAILABLE` and `RZKERR_CB_BUSY` are returned only if `RZK_PERFORMANCE` is undefined.

Argument(s)

`hThread`	Handle to the thread to be deleted.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the function returned successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the thread handle to be deleted is invalid or already deleted.
`RZKERR_INVALID_OPERATION`	Indicates that you tried to delete a thread that is not in DELETED state or contains a locked thread from DELETED state.

Example

A previously-created thread, `hThreadHandle` is deleted. The thread deletion status is stored in the `status` variable.

```
/*contains thread handle to be deleted.*/
RZK_THREADHANDLE_t hThreadHandle;
RZK_STATUS_t status;

status = RZKDeleteThreadEnhanced (hThreadHandle);
```

See Also

RZKCreateThread
RZKDeleteThread
RZKSetThreadPriority
RZKYieldThread
RZKGetThreadParameters
RZKDisablePreemption
RZKEnablePreemption
RZKRestorePreemption
RZKSuspendInterruptThread
RZKResumeInterruptThread

RZKSUSPENDTHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKSuspendThread(
RZK_THREADHANDLE_t  hThread,
TICK_t tTicks);
```

Description

The `RZKSuspendThread()` API suspends a thread finitely or infinitely. If suspended infinitely, it can be started again only by a `RZKResumeThread()` call. For suspending the thread infinitely, pass the `MAX_INFINITE_SUSPEND` to `tTicks`.

A thread can call the `RZKSuspendThread()` API to suspend itself finitely, but when it tries to suspend another thread finitely, an `RZKERR_INVALID_OPERATION` is returned.

Note: This API returns immediately when called within the clean-up function of a thread that has deleted itself because the preemption for the deleted thread is disabled. See the RZKDeleteThread API definition on page 71.

Argument(s)

`hThread`	Handle to the thread to be suspended.
`tTicks`	The time-out period to wait before returning. If time-out period is 0, the API returns immediately. If the time-out period is `MAX_INFINITE_SUSPEND`, the API waits infinitely.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the function returned successfully.
`RZKERR_INVALID_HANDLE`	Indicates that thread handle to be suspended is invalid.
`RZKERR_INVALID_OPERATION`	Indicates that you tried to perform a timed suspend on another thread OR tried to suspend a thread in the process of being deleted.
`RZKERR_CB_BUSY`	Indicates that the thread control block is busy.
`RZKERR_TIMEOUT`	Indicates that the system timer ISR was unable to resume the thread within the time.

Example 1

A thread is suspended from execution for a period of 10 ticks. The status of execution is stored in the `status` variable.

```
#define THREAD_SUSPEND_TICKS          10

RZK_THREADHANDLE_t hThreadHandle;
RZK_STATUS_t status;
```

```
status = RZKSuspendThread(hThreadHandle,
THREAD_SUSPEND_TICKS);
```

Example 2

A thread is suspended from execution infinitely. The status of execution is stored in the `status` variable.

```
RZK_THREADHANDLE_t hCurr_Thread;
RZK_STATUS_t status;
status = RZKSuspendThread(hCurr_Thread,
MAX_INFINITE_SUSPEND);
```

See Also

RZKCreateThread	RZKDeleteThread
RZKSetThreadPriority	RZKYieldThread
RZKGetThreadParameters	RZKDisablePreemption
RZKEnablePreemption	RZKRestorePreemption
RZKSuspendInterruptThread	RZKResumeInterruptThread

RZKSETTHREADPRIORITY

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKSetThreadPriority(
RZK_THREADHANDLE_t      hThread,
RZK_THREAD_PRIORITY_t   threadPriority);
```

Description

The `RZKSetThreadPriority()` API sets a new priority value for a thread.

Argument(s)

`hThread`	Handle to the thread to be resumed.
`threadPriority`	The new priority for the thread. The range of values is from 1 to 31.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the function returned successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the thread handle to be resumed is invalid.
`RZKERR_INVALID_ARGUMENTS`	Indicates that the thread priority value is invalid.

RZKERR_INVALID_OPERATION	Indicates that you tried to set a new priority value to a thread that is in the process of being deleted.
RZKERR_CB_BUSY	Indicates that the thread control block is in exclusive use.

Example

The priority of thread `hThreadHandle` is changed to a value 20.

```
#define NEW_PRIORITY 20

RZK_THREADHANDLE_t hThreadHandle;
RZK_STATUS_t status;

status = RZKSetThreadPriority(hThreadHandle,
NEW_PRIORITY);
```

See Also

RZKCreateThread	RZKDeleteThread
RZKSuspendThread	RZKYieldThread
RZKGetThreadParameters	RZKDisablePreemption
RZKEnablePreemption	RZKRestorePreemption
RZKSuspendInterruptThread	RZKResumeInterruptThread

RZKRESUMETHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKResumeThread(
RZK_THREADHANDLE_t    hThread);
```

Description

The `RZKResumeThread()` API resumes an infinitely-suspended thread.

Note: This API returns immediately when called within the clean-up function of a thread that has deleted itself because the preemption for the deleted thread is disabled. See the RZKDeleteThread API definition on page 71.

Argument(s)

`hThread`	Handle to the thread to be resumed.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the function returned successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the thread handle to be resumed is invalid.

`RZKERR_INVALID_OPERATION`	Indicates operation is invalid and you tried to resume a finitely suspended thread OR tried to resume a thread in the process of being deleted.
`RZKERR_CB_BUSY`	Indicates that the thread control block is in exclusive use.

Example

An infinitely suspended thread `hThreadHandle` resumes and stores the call status in `status` variable.

```
/** thread handle to be resumed */
RZK_THREADHANDLE_t hThreadHandle;
RZK_STATUS_t status;
status = RZKResumeThread(hThreadHandle);
```

See Also

RZKCreateThread
RZKDeleteThread
RZKSuspendThread
RZKYieldThread
RZKGetThreadParameters
RZKSetThreadPriority
RZKDisablePreemption
RZKEnablePreemption
RZKRestorePreemption
RZKSuspendInterruptThread
RZKResumeInterruptThread

RZKYIELDTHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKYieldThread();
```

Description

The `RZKYieldThread()` API yields control and use of the CPU to a next ready-to-run thread in the dispatch queue that is at an equal priority level.

Note: This API returns immediately when called within the clean-up function of a thread that has deleted itself because the preemption for the deleted thread is disabled. See the RZKDeleteThread API definition on page 71.

Argument(s)

None.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the function returned successfully.

Example

Control of a processor from `hThreadHandle` is yielded to other threads that are created with the same priority. The call status is stored in the `status` variable.

```
/** thread handle from which the control to be yielded
*/
RZK_THREADHANDLE_t hThreadHandle;
RZK_STATUS_t status;
status = RZKYieldThread();
```

See Also

RZKCreateThread	RZKDeleteThread
RZKSuspendThread	RZKSetThreadPriority
RZKGetThreadParameters	RZKDisablePreemption
RZKEnablePreemption	RZKRestorePreemption
RZKSuspendInterruptThread	RZKResumeInterruptThread

RZKGETTHREADPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKGetThreadParameters(
RZK_THREADHANDLE_t      hThread,
RZK_THREADPARAMS_t      *pThreadParams);
```

Description

The `RZKGetThreadParameters()` API obtains the current parameters of the thread from the thread control block and returns them in the `RZK_THREADPARAMS_t` structure. See Table 57 on page 346 for the members of the `RZK_THREADPARAMS_t` structure.

Argument(s)

`hThread`	Handle to the thread for which the parameters are required.
`pThreadParams`	A pointer to the structure that receives the appropriate values.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the function returned successfully.
`RZKERR_INVALID_HANDLE`	Indicates that thread handle is invalid.

RZKERR_INVALID_ARGUMENTS	Indicates that pThreadParams is invalid.
RZKERR_CB_BUSY	Indicates that thread control block is in exclusive use, that is, it is busy.

Example

The parameters of a thread, pointed to by hThreadHandle, are stored in the RZK_THREADPARAMS_t structure. The call status is returned in the status variable.

```
RZK_THREADHANDLE_t hThreadHandle;
RZK_THREADPARAMS_t ThreadParams;
RZK_STATUS_t status;
status = RZKGetThreadParameters( hThreadHandle,
  &ThreadParams);
```

See Also

RZKCreateThread	RZKDeleteThread
RZKSuspendThread	RZKSetThreadPriority
RZKYieldThread	RZK_THREADPARAMS_t
RZKDisablePreemption	RZKRestorePreemption
RZKSuspendInterruptThread	RZKResumeInterruptThread

RZKDISABLEPREEMPTION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATE_t RZKDisablePreemption();
```

Description

The `RZKDisablePreemption()` API disables the preemption of the current task.

Argument(s)

None.

Return Value(s)

This API returns a value 1 if preemption is enabled, or returns a value of 0 if preemption is disabled.

See Also

RZKCreateThread
RZKDeleteThread
RZKSuspendThread
RZKSetThreadPriority
RZKYieldThread
RZK_THREADPARAMS_t
RZKEnablePreemption
RZKRestorePreemption
RZKSuspendInterruptThread
RZKSuspendInterruptThread

RZKENABLEPREEMPTION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZKEnablePreemption ();
```

Description

The `RZKEnablePreemption()` API enables the preemption of the current task irrespective of the previous status of preemption.

Argument(s)

None.

Return Value(s)

None.

See Also

RZKCreateThread
RZKDeleteThread
RZKSuspendThread
RZKSetThreadPriority
RZKYieldThread
RZK_THREADPARAMS_t
RZKDisablePreemption
RZKRestorePreemption
RZKSuspendInterruptThread
RZKResumeInterruptThread

RZKRESTOREPREEMPTION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZKRestorePreemption (RZK_STATE_t uState);
```

Description

The `RZKRestorePreemption()` API restores the preemption of the current task. If the preemption was disabled twice previously, restore does not enable a preemption. If preemption disabling was conducted only only one time previously, this call enables preemption.

Argument(s)

Previous status of preemption (return value of the previous `RZKDisablePreemption()` call).

Return Value(s)

None.

See Also

RZKCreateThread	RZKDeleteThread
RZKSuspendThread	RZKSetThreadPriority
RZKYieldThread	RZK_THREADPARAMS_t
RZKDisablePreemption	RZKEnablePreemption
RZKSuspendInterruptThread	RZKResumeInterruptThread

RZKSUSPENDINTERRUPTTHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZKSuspendInterruptThread ();
```

Description

The `RZKSuspendInterruptThread()` API suspends the currently-executing interrupt thread. This API must only be called from within an interrupt thread. You must exercise caution while using this API, because it does not return error values. This API must be called under disable interrupts and to disable the interrupt `RZKDisableInterrupt` API is called.

Argument(s)

None.

Return Value(s)

None.

See Also

RZKCreateThread
RZKDeleteThread
RZKSuspendThread
RZKSetThreadPriority
RZKYieldThread
RZK_THREADPARAMS_t
RZKDisablePreemption
RZKEnablePreemption
RZKRestorePreemption
RZKSuspendInterruptThread
RZKResumeInterruptThread

RZKRESUMEINTERRUPTTHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZKResumeInterruptThread (RZK_THREADHANDLE_t
  hThread);
```

Description

The `RZKResumeInterruptThread()` API resumes an interrupt thread and must be called only from within an interrupt's prologue. You must exercise caution while using this API, because it does not return error values. This API must be called under disable interrupts and to disable the interrupt `RZKDisableInterrupt` API is called.

Argument(s)

The interrupt thread's handle.

Return Value(s)

None.

See Also

RZKCreateThread
RZKDeleteThread
RZKSuspendThread
RZKSetThreadPriority
RZKYieldThread
RZK_THREADPARAMS_t
RZKDisablePreemption
RZKEnablePreemption
RZKRestorePreemption
RZKSuspendInterruptThread
RZKResumeInterruptThread

Scheduler APIs

RZK includes one scheduler API, which is described below.

RZKGETSCHEDULERPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZScheduler.h"
```

Description

The `RZKGetSchedulerParameters()` API is defined as a macro to the `uDefaultTimeSlice` variable, which indicates the current default time slice. The value of this variable is used for round-robin threads if you specify zero quantum during thread creation.

Argument(s)

None.

Return Value(s)

The value of the `uDefaultTimeSlice` variable is returned whenever the `RZKGetSchedulerParameters()` API is invoked.

Example

The default round-robin time slice value is assigned to the `rrTICS` variable.

```
TICKS_t rrTICS;
rrTICS=RZKGetSchedulerParameters();
```

Message Queue APIs

Table 28 provides a quick reference to a number of message queue APIs that are described in this subsection.

Table 28. Message Queue API Quick Reference

RZKCreateQueue	RZKReceiveFromQueue
RZKDeleteQueue	RZKPeekMessageQueue
RZKSendToQueue	RZKGetQueueParameters
RZKSendToQueueFront	RZKSendToQueueUnique

RZKCREATEQUEUE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMessageQ.h"
```

Prototype

```
RZK_MESSAGEQHANDLE_t RZKCreateQueue(
RZK_NAME_t          szName[MAX_OBJECT_NAME_LEN],
COUNT_t             uQueueLength,
RZK_PTR_t           pMessage,
COUNT_t             uMaxSizeOfMessage,
RZK_RECV_ATTRIB_et  etAttrib);
```

Description

The `RZKCreateQueue()` API creates a message queue with the specified parameters and sets various items in the message queue control block. The messages are stored in the memory area that is allocated by the program that calls this function. The length of the memory area allocated is: `(uQueueLength * sizeof(COUNT_t)) + ( uQueueLength * uMaxSizeOfMessage );`

Argument(s)

`szName`	Specifies the name of the message queue–ASCII string.
`uQueueLength`	Specifies the queue length, which is the number of messages that can be stored in the message queue.

`pMessage`	Pointer to the memory area where messages are to be stored.
`uMaxSizeOfMessage`	The maximum size of individual messages.
`etAttrib`	Specifies the receiving order attribute and contains one of the following.*
	RECV_ORDER_FIFO — Receiving order is FIFO.
	RECV_ORDER_PRIORITY — Receiving order is PRIORITY.

Note: *If receiving order is not `RECV_ORDER_FIFO`, by default it is taken as `RECV_ORDER_PRIORITY`.

Return Value(s)

The function returns a handle to the message queue if it is created successfully or returns `NULL`. If `NULL` is returned it sets one of the following error values in the thread control block of the current thread. `RZKGetErrorNum()` API can be called to retrieve the error number stored in the thread control block.

`RZKERR_CB_UNAVAILABLE`	The control block is not available to create a new queue.
`RZKERR_INVALID_ARGUMENTS`	The parameters are invalid.

Example 1

A message queue is created with the name *Zilog*, a queue length of 10, a maximum message size of 200 and a receiving order of messages of FIFO. The return value is stored in the message queue handle.

```
#define MSGQ_LENGTH                    10
#define MESSAGE_SIZE                   200
```

```
unsigned char msgBuffer[ (MSGQ_LENGTH * MESSAGE_SIZE)
+ (MSGQ_LENGTH * sizeof(COUNT_t))];
RZK_MESSAGEQHANDLE_t hMessageQueue;
hMessageQueue = RZKCreateQueue((RZK_NAME_t [
])"Zilog",
  MSGQ_LENGTH,
  msgBuffer,
  MESSAGE_SIZE,
  RECV_ORDER_FIFO);
```

Example 2

A message queue is created with the name *Zilog*, a queue length of 10, a maximum message size of 200 and a receiving order of messages of *priority-based*. The return value is stored in the message queue handle.

```
#define MSGQ_LENGTH                10
#define MESSAGE_SIZE               200
unsigned char msgBuffer[ (MSGQ_LENGTH * MESSAGE_SIZE)
+
  (MSGQ_LENGTH * sizeof(COUNT_t)) ];
RZK_MESSAGEQHANDLE_t hMessageQueue;
hMessageQueue = RZKCreateQueue((RZK_NAME_t [
])"Zilog",
  MSGQ_LENGTH,
  msgBuffer,
  MESSAGE_SIZE,
  RECV_ORDER_PRIORITY);
```

The memory to be allocated in bytes to hold the messages in the message queue can be calculated. Assuming that `MESSAGE_SIZE` is the maximum message size that can be present in the message queue and `MSGQ_LENGTH` is the message queue length, then the following equation can be constructed.

```
Memory to be allocated in bytes = (MESSAGE_SIZE *
MSGQ_LENGTH) + (MSGQ_LENGTH * sizeof(COUNT_t))
```

See Also

RZKDeleteQueue
RZKSendToQueue
RZKReceiveFromQueue
RZKSendToQueueUnique
RZKSendToQueueFront
RZKGetQueueParameters
RZKPeekMessageQueue

RZKDELETEQUEUE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMessageQ.h"
```

Prototype

```
RZK_STATUS_t RZKDeleteQueue(
RZK_MESSAGEQHANDLE_t  hMessageQueue);
```

Description

The `RZKDeleteQueue()` API deletes a queue and the threads that are waiting on the queue to pend/post are placed into the READY state from the `BLOCKED/TIME_WAIT` state.

Argument(s)

`hMessageQueue`	Specifies the message queue handle to be deleted.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	The specified message queue handle is invalid.
`RZKERR_CB_BUSY`	The message queue control block is used for an exclusive purpose; for example, it is busy.

Example

A previously-created message queue represented by `hMessageQueue`, is deleted. The return value is stored in the `status` variable.

```
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
RZK_STATUS_t status;
status = RZKDeleteQueue(hMessageQueue);
```

See Also

RZKCreateQueue
RZKSendToQueue
RZKReceiveFromQueue
RZKSendToQueueUnique
RZKSendToQueueFront
RZKGetQueueParameters
RZKPeekMessageQueue

RZKSENDTOQUEUE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMessageQ.h"
```

Prototype

```
RZK_STATUS_t RZKSendToQueue(
RZK_MESSAGEQHANDLE_t  hMessageQueue,
RZK_PTR_t             pMessage,
COUNT_t               uSize,
TICK_t                tBlockTime);
```

Description

This function sends a message to a specified queue. The message is appended to the end of the queue. Thread blocking occurs when the queue cannot store any more messages because the maximum queue length specified during queue creation is reached.

If the message size sent to the queue is greater than the maximum message size set during queue creation, then the previously set maximum message size is used and rest of the message is truncated.

Argument(s)

`hMessageQueue`	Specifies the handle to the message queue to which messages are required to be sent.
`pMessage`	Pointer to the message.

uSize	Message size.
tBlockTime	Period to wait if the message queue is full. For infinite blocking, use MAX_INFINITE_SUSPEND.

Return Value(s)

RZKERR_SUCCESS	The operation completed successfully.
RZKERR_INVALID_HANDLE	The specified message queue handle is invalid.
RZKERR_INVALID_ARGUMENTS	The parameters are invalid.
RZKERR_OBJECT_DELETED	The message queue on which a thread is pending to send a message is deleted.
RZKERR_CB_BUSY	The message queue control block is used for an exclusive purpose, that is, busy.
RZKERR_TIMEOUT	A time-out occurred and the pend operation could not be completed within the specified time.

Example 1

A message is sent that is pointed to by the pMessage to the message queue with a handle that is stored in hMessageQueue. If the queue is not empty, the thread waits for a period of 200 ticks (worst case). Message size is 500 bytes and the return value is the call status.

```
#define MESSAGE_SIZE 500
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
unsigned char pMessage[ MESSAGE_SIZE ] = "Hello
World";
```

```
RZK_STATUS_t status;
status =
RZKSendToQueue(hMessageQueue,pMessage,MESSAGE_SIZE,200
);
```

Example 2

A message is sent that is pointed to by the `pMessage` to the message queue with a handle of `hMessageQueue`. This example tries to post message with time-out value as zero. If the message queue is full, the `status` variable contains an error code `RZKERR_TIMEOUT`.

```
#define MESSAGE_SIZE 500
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
unsigned char pMessage[ MESSAGE_SIZE ] = "Hello
World";
RZK_STATUS_t status;
status = RZKSendToQueue(hMessageQueue,
  pMessage,
  MESSAGE_SIZE,
  0);
```

See Also

RZKCreateQueue
RZKDeleteQueue
RZKReceiveFromQueue
RZKSendToQueueFront
RZKPeekMessageQueue
RZKGetQueueParameters
RZKSendToQueueUnique

RZKSENDTOQUEUEFRONT

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMessageQ.h"
```

Prototype

```
RZK_STATUS_t RZKSendToQueueFront(
RZK_MESSAGEQHANDLE_t  hMessageQueue,
RZK_PTR_t             pMessage,
COUNT_t               uSize);
```

Description

This function sends the message to the front position in the queue. If the queue is full, it returns an error. If there are a number of threads waiting to receive the message, this function directly copies the message to the first thread waiting to receive the message. The selection of the thread is based on FIFO or PRIORITY mode of operation.

If the message size sent to the queue is greater than the maximum message size set during queue creation, then the previously set maximum message size is used.

Argument(s)

`hMessageQueue`	Specifies the handle to the message queue to whose front the message is to be posted.
`pMessage`	Pointer to the message.
`uSize`	Size of the message.

Return Value(s)

RZKERR_SUCCESS	The operation completed successfully.
RZKERR_INVALID_HANDLE	Specified message queue handle invalid.
RZKERR_INVALID_ARGUMENTS	The parameters are invalid.
RZKERR_CB_BUSY	message queue control block is used for an exclusive purpose; for example, it is busy.
RZKERR_QUEUE_FULL	Indicates that the message queue is full.

Example

A message is sent that is pointed by `pMessage` to the front of the message queue with a handle of `hMessageQueue`. The size of the message is 500 bytes. The return value is stored in the `status` variable.

```
#define MESSAGE_SIZE 500
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
unsigned char pMessage[ MESSAGE_SIZE ] = "Hello
World";
RZK_STATUS_t status;
status = RZKSendToQueueFront(hMessageQueue,
  pMessage,
  MESSAGE_SIZE);
```

See Also

RZKCreateQueue
RZKDeleteQueue
RZKReceiveFromQueue
RZKSendToQueue
RZKPeekMessageQueue
RZKGetQueueParameters
RZKSendToQueueUnique

RZKRECEIVEFROMQUEUE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMessageQ.h"
```

Prototype

```
RZK_STATUS_t RZKReceiveFromQueue(
RZK_MESSAGEQHANDLE_t hMessageQueue,
RZK_PTR_t pMessage,
COUNT_t *uSize,
TICK_t tBlockTime);
```

Description

The `RZKReceiveFromQueue` API receives a message from the front of the message queue. If the message is not available, the thread performs a timed/infinite wait until it receives a message. The message is received depending on the receiving attribute (`RZK_RECV_FIFO` or `RZK_RECV_PRIORITY`) of the message queue. The message size received from the queue is either the size represented by the `uSize` parameter or the actual maximum message size for the message queue, whichever is minimum.

Argument(s)

`hMessageQueue`	Specifies the handle to the message queue to receive the message from.
`pMessage`	Pointer to the memory where the received message is required to be stored.

`uSize*`	This parameter is an input and an output to/from the API. Input: uSize defines the buffer size meaning a message of this size or less to be received (expected maximum size). Output: When this API executes successfully and receives message.
`tBlockTime`	The period to wait if message is not available. Use `MAX_INFINITE_SUSPEND` for infinite blocking.

Note: `*uSize` contains the actual size of the message received, depending on current queue status.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	Handle of specified message queue is invalid.
`RZKERR_INVALID_ARGUMENTS`	The parameters are invalid.
`RZKERR_OBJECT_DELETED`	The message queue from which the message is to be received is deleted by another thread.
`RZKERR_CB_BUSY`	The message queue control block is used for an exclusive purpose; for example, it is busy.
`RZKERR_TIMEOUT`	A time-out occurred and the receive operation could not be completed within the specified time.

Example 1

A message is received from the message queue with a handle of `hMessageQueue`; this message is copied into the buffer `pMessage`. If the queue is empty, the thread waits in the time queue for 200 ticks. The maximum message size that `pMessage` can receive is 500 bytes (stored in `uSize`). The API execution status is stored in the `status` variable. When this API executes successfully, the `uSize` variable contains the actual size of the received message.

```
#define MESSAGE_SIZE 500
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
unsigned char pMessage[ MESSAGE_SIZE ];
RZK_STATUS_t status;
COUNT_t uSize = MESSAGE_SIZE;
status = RZKReceiveFromQueue(hMessageQueue,
  pMessage,
  &uSize,
  200);
```

Example 2

A message is received from the message queue with a handle of `hMessageQueue`; it is copied into the buffer `pMessage`. If the queue is empty, the thread waits infinitely. The maximum message size that `pMessage` can receive is 500 bytes (stored in `uSize`). The API execution status is stored in the `status` variable. When this API executes successfully, the `uSize` variable contains the actual size of the received message.

```
#define MESSAGE_SIZE 500
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
unsigned char pMessage[ MESSAGE_SIZE ];
RZK_STATUS_t status;
COUNT_t uSize = MESSAGE_SIZE;
status = RZKReceiveFromQueue(hMessageQueue,
  pMessage,
  &uSize,
  MAX_INFINITE_SUSPEND);
```

See Also

RZKCreateQueue
RZKDeleteQueue
RZKSendToQueueFront
RZKSendToQueue
RZKPeekMessageQueue
RZKGetQueueParameters
RZKSendToQueueUnique

RZKPEEKMESSAGEQUEUE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMessageQ.h"
```

Prototype

```
RZK_STATUS_t RZKPeekMessageQueue(
RZK_MESSAGEQHANDLE_t hMessageQueue,
RZK_PTR_t pMessage
COUNT_t *uSize);
```

Description

The `RZKPeekMessageQueue()` API call peeks into the specified message queue and copies the message from the queue to a specified location. Being a nonblocking call, the `RZKPeekMessageQueue()` returns immediately. The message queue status remains intact. Before the call is executed, `uSize` takes a value for the buffer that holds the message. If the call is successful, `uSize` reflects the actual message size.

Argument(s)

`hMessageQueue`	Specifies the handle to the message queue for peeking into.
`pMessage`	Pointer to the memory area where the message is required to be stored.

`uSize*`	This parameter is an input and an output to/from the API. Input: uSize defines the buffer size meaning a message of this size or less to be received (Expected MAX size) Output: when this API executes successfully and receives message.

Note: `*uSize` contains the actual size of the message received depending on current queue status.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	The handle of the message queue specified is invalid.
`RZKERR_CB_BUSY`	The message queue control block is used for an exclusive purpose; for example, it is busy.
`RZKERR_QUEUE_EMPTY`	The message queue is empty.
`RZKERR_INVALID_ARGUMENTS`	The parameters are invalid.

Example

A message queue with a handle of `hMessageQueue` peeks into and determines if any message is present. If a message is present, it is copied to the buffer `pMessage` with a maximum size of 500 bytes (stored in `uSize`). The API execution status is stored in the `status` variable. When this API executes and if the call is successful, `uSize` contains the actual size of the message being copied to `pMessage`.

```
#define MESSAGE_SIZE 500
```

```
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
unsigned char pMessage[ MESSAGE_SIZE ];
RZK_STATUS_t status;
COUNT_t uSize = MESSAGE_SIZE;
status = RZKPeekMessageQueue(hMessageQueue,
  pMessage,
  &uSize);
```

See Also

RZKCreateQueue
RZKDeleteQueue
RZKSendToQueueFront
RZKSendToQueue
RZKGetQueueParameters
RZKReceiveFromQueue
RZKSendToQueueUnique

RZKGETQUEUEPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMessageQ.h"
```

Prototype

```
RZK_STATUS_t RZKGetQueueParameters(
RZK_MESSAGEQHANDLE_t hMessageQueue,
RZK_MESSAGEQPARAMS_t *pQueueParams);
```

Description

The `RZKGetQueueParameters()` API obtains the parameters of the specified message queue and stores it in the `RZK_MESSAGEQPARAMS_t` structure. See Table 59 on page 348 for members of the `RZK_MESSAGEQPARAMS_t` structure.

Argument(s)

`hMessageQueue`	Specifies the handle to the message queue whose parameters must be obtained.
`pQueueParams`	Pointer to the structure to receive the parameters.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	Specified message queue handle invalid.

RZKERR_INVALID_ARGUMENTS	The parameters are invalid.
RZKERR_CB_BUSY	The message queue control block is used for exclusive purpose; for example, it is busy.

Example

The parameters of the message queue with a handle of hMessageQueue are stored into the RZK_MESSAGEQPARAMS_t structure. The RZKGetQueueParameters() API execution status is stored in the status variable.

```
extern RZK_MESSAGEQHANDLE_t hMessageQueue;
RZK_MESSAGEQPARAMS_t *pParams;
RZK_STATUS_t status;
status = RZKGetQueueParameters(hMessageQueue,
  pParams);
```

See Also

RZKCreateQueue	RZKDeleteQueue
RZKSendToQueueFront	RZKSendToQueue
RZK_MESSAGEQPARAMS_t	RZKReceiveFromQueue
RZKPeekMessageQueue	RZKSendToQueueUnique

RZKSENDTOQUEUEUNIQUE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZMessageQ.h"
```

Prototype

```
COUNT_t RZKSendToQueueUnique (
  RZK_MESSAGEQHANDLE_t hMessageQueue,
  RZK_PTR_t pMessage,
  COUNT_t uSize,
  TICK_t tBlockTime
  );
```

Description

The `RZKSendToQueueUnique()` API sends only unique messages to the message queue. If the queue already contains messages that you must resend, this API returns the error `RZKERR_MSG_PRESENT`.

Argument(s)

`hMessageQueue`	Specifies the handle to the message queue to which messages must be sent.
`pMessage`	Pointer to the message.
`uSize`	Message size.
`tBlockTime`	Period to wait if the message queue is full. For infinite blocking, use `MAX_INFINITE_SUSPEND`.

Return Value(s)

RZKERR_SUCCESS	Indicates that the operation completed successfully.
RZKERR_INVALID_HANDLE	Indicates that the message queue handle is invalid.
RZKERR_INVALID_ARGUMENTS	Indicates that the parameters are invalid.
RZKERR_OBJECT_DELETED	Indicates the message queue on which a thread is pending to send a message is deleted.
RZKERR_CB_BUSY	Indicates that the message queue control block is used for an exclusive purpose, that is, busy.
RZKERR_TIMEOUT	Indicates that a time-out occurred and the pend operation could not be completed within the specified time.
RZKERR_MSG_PRESENT	Indicates that the message is already present.

See Also

RZKCreateQueue
RZKDeleteQueue
RZKSendToQueueFront
RZKSendToQueue
RZK_MESSAGEQPARAMS_t
RZKReceiveFromQueue
RZKPeekMessageQueue

Semaphore APIs

Table 29 provides a quick reference to a number of semaphore APIs that are described in this subsection.

Table 29. Semaphore API Quick Reference

RZKCreateSemaphore	RZKReleaseSemaphore
RZKDeleteSemaphore	RZKGetSemaphoreParameters
RZKAcquireSemaphore	

RZKCREATESEMAPHORE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZSemaphore.h"
```

Prototype

```
RZK_SEMAPHOREHANDLE_t RZKCreateSemaphore(
RZK_NAME_t szName[MAX_OBJECT_NAME_LEN],
COUNT_t uInitialCount,
RZK_RECV_ATTRIB_et etAttrib);
```

Description

The `RZKCreateSemaphore()` API call creates a semaphore specifying the initial count and the mode of waiting for the threads on this semaphore. The initial count (`uInitialCount`) decides the number of threads that can hold this shared resource simultaneously. The notion of exclusive ownership of a resource is present only for a binary semaphore.

Argument(s)

`szName`	Specifies the name of the semaphore/ASCII string.
`uInitialCount`	The maximum number of objects that can acquire the semaphore. If this value is 1, the semaphore is a binary semaphore. If this value is 0, the semaphore is a binary semaphore and it acquires immediately. Otherwise, it is a counting semaphore.

`etAttrib`	Specifies the receiving order attribute and contains one of the following.	
	`RECV_ORDER_FIFO`	Receiving order is FIFO.
	`RECV_ORDER_PRIORITY`	Receiving order is Priority.

Note: *If the receiving order is not `RECV_ORDER_PRIORITY`, by default it is taken as `RECV_ORDER_FIFO`.

Return Value(s)

The function returns a handle to the semaphore if it is created successfully or else returns NULL. If NULL is returned it sets one of the following error values in the current thread's thread control block.The `RZKGetErrorNum()` API is called to retrieve the error number stored in the thread control block.

`RZKERR_INVALID_ARGUMENTS`	The parameters are invalid.
`RZKERR_CB_UNAVAILABLE`	The control block is not available to create the semaphore.

Example 1

A counting semaphore is created with the name *Zilog*, an initial count of 5 and a receive order of FIFO. Upon successful creation, the semaphore handle is returned to the `hSemaphore` variable.

```
#define MAX_SEM_COUNT 5
RZK_SEMAPHOREHANDLE_t hSemaphore;
hSemaphore = RZKCreateSemaphore((RZK_NAME_t [
])"Zilog",
  MAX_SEM_COUNT,
  RECV_ORDER_FIFO);
```

Example 2

A counting semaphore is created with the name *Zilog*, an initial count of 0 (binary semaphore) and a receive order of PRIORITY. Upon successful creation, the semaphore handle is returned into the `hSemaphore` variable.

```
#define MAX_SEM_COUNT 1
RZK_SEMAPHOREHANDLE_t hSemaphore;
hSemaphore = RZKCreateSemaphore((RZK_NAME_t [
])"Zilog",
  MAX_SEM_COUNT,
  RECV_ORDER_PRIORITY);
```

Note: For priority inheritance to work, `uInitialCount` must be 1, `etAttrib` must be `RECV_ORDER_PRIORITY` and `RZK_PRIORITYINHERITANCE` must be defined in the `ZSysgen.h` file.

See Also

RZKDeleteSemaphore
RZKAcquireSemaphore
RZKGetSemaphoreParameters
RZKReleaseSemaphore
RZKGetErrorNum
RZK_RECV_ATTRIB_et

RZKDELETESEMAPHORE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZSemaphore.h"
```

Prototype

```
RZK_STATUS_t RZKDeleteSemaphore(
RZK_SEMAPHOREHANDLE_t hSemaphore);
```

Description

The `RZKDeleteSemaphore()` API call invalidates an existing semaphore control block and resumes all waiting threads on this semaphore in the order specified during the semaphore creation. It sets the appropriate status in the thread control blocks and makes the semaphore control block available for `RZKCreateSemaphore()` calls.

Argument(s)

`hSemaphore`	Specifies the handle of the semaphore to delete.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	The specified Semaphore handle is invalid.
`RZKERR_CB_BUSY`	If the semaphore control block is used for an exclusive purpose; for example, it is busy.

Example

A previously-created semaphore with a handle of `hSemaphore` is deleted, making the handle invalid. The API execution status is stored in the `status` variable.

```
extern RZK_SEMAPHOREHANDLE_t hSemaphore;
RZK_STATUS_t status;
status = RZKDeleteSemaphore(hSemaphore);
```

See Also

RZKCreateSemaphore
RZKAcquireSemaphore
RZKReleaseSemaphore
RZKGetSemaphoreParameters

RZKACQUIRESEMAPHORE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZSemaphore.h"
```

Prototype

```
RZK_STATUS_t RZKAcquireSemaphore(
RZK_SEMAPHOREHANDLE_t hSemaphore,
TICK_t tBlockTime);
```

Description

The `RZKAcquireSemaphore()` API is used to acquire a semaphore. The thread waits for the duration specified in `tBlockTime` if the semaphore is not immediately available (that is, the maximum count is already reached). If this semaphore supports priority inheritance, the priority of the thread that acquired this semaphore is raised, if necessary.

Argument(s)

`hSemaphore`	Specifies the handle to the Semaphore that is acquired.
`tBlockTime`	Specifies the time to wait if semaphore not available. For infinite time waiting, use `MAX_INFINITE_SUSPEND`.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the operation completed successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the handle to the specified semaphore to acquire is invalid.
`RZKERR_TIMEOUT`	Indicates that the requested semaphore could not be acquired in the specified time.
`RZKERR_OBJECT_DELETED`	Indicates that the semaphore on which the acquire is requested is deleted by another thread.
`RZKERR_CB_BUSY`	Indicates that the semaphore control block is used for an exclusive purpose; for example, it is busy.
`RZKERR_INVALID_OPERATION`	Indicates that the API is called from an ISR and the semaphore is a binary semaphore.

Example 1

The acquisition of a previously-created counting semaphore with a handle of `hSemaphore` is attempted. If the semaphore is already acquired or the semaphore is not free then the thread waits 200 ticks to acquire the semaphore. The execution status of the API is stored in the `status` variable.

```
extern RZK_SEMAPHOREHANDLE_t hSemaphore;
RZK_STATUS_t status;
status = RZKAcquireSemaphore(hSemaphore,
  200);
```

Example 2

The acquisition of a previously-created binary semaphore with a handle of `hSemaphore` is attempted. If the semaphore is already acquired or the

semaphore is not free then the thread waits infinitely to acquire the semaphore. The execution status of the API is stored in the `status` variable.

```
extern RZK_SEMAPHOREHANDLE_t hSemaphore;
RZK_STATUS_t status;
status = RZKAcquireSemaphore(hSemaphore,
  MAX_INFINITE_SUSPEND);
```

See Also

RZKCreateSemaphore
RZKDeleteSemaphore
RZKGetSemaphoreParameters
RZKReleaseSemaphore

RZKRELEASESEMAPHORE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZSemaphore.h"
```

Prototype

```
RZK_STATUS_t RZKReleaseSemaphore(
RZK_SEMAPHOREHANDLE_t hSemaphore);
```

Description

The `RZKReleaseSemaphore()` API releases a previously acquired semaphore and makes it available for further acquiring. If this semaphore supports priority inheritance, the owner thread's priority and the priority of the other threads in sequence of the priority inheritance, are restored.

Note: If `RZK_PRIORITYINHERITANCE` is defined, the binary semaphore held by the thread is released when the thread terminates.

Argument(s)

`hSemaphore`	Specifies the handle to the semaphore to be released.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the operation is completed successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the handle to the semaphore specified for release is invalid.

RZKERR_INVALID_OPERATION	Indicates that the API is called from an ISR and the semaphore is a binary semaphore.
RZKERR_SEM_NOTOWNED	Generated if the semaphore is not owned by the thread which calls the release API. This error is returned only if RZK_PRIORITYINHERITANCE is defined.
RZKERR_CB_BUSY	Indicates that the semaphore control block is used for an exclusive purpose; for example, it is busy.

Example

An acquired semaphore is released (freed) after data manipulation in a critical section. The `RZKReleaseSemaphore()` API execution call status is stored in the `status` variable.

```
extern RZK_SEMAPHOREHANDLE_t hSemaphore;
RZK_STATUS_t status;
status = RZKReleaseSemaphore(hSemaphore);
```

See Also

RZKCreateSemaphore
RZKDeleteSemaphore
RZKGetSemaphoreParameters
RZKAcquireSemaphore

RZKGETSEMAPHOREPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZSemaphore.h"
```

Prototype

```
RZK_STATUS_t RZKGetSemaphoreParameters(
RZK_SEMAPHOREHANDLE_t   hSemaphore,
RZK_SEMAPHOREPARAMS_t   *pSemaphoreParams);
```

Description

The `RZKGetSemaphoreParameters()` API gets the semaphore parameters and places it into the `RZK_SEMAPHOREPARAMS_t` structure. See Table 60 on page 349 for the members of the `RZK_SEMAPHOREPARAMS_t` structure.

Argument(s)

`hSemaphore`	Specifies the handle to the semaphore, the parameters of which are to be obtained.
`pSemaphoreParams`	Pointer to the structure type `RZK_SEMAPHOREPARAMS_t` receiving the semaphore parameters.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the operation is completed successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the handle to the semaphore whose parameters are requested is invalid.

RZKERR_INVALID_ARGUMENTS	Indicates that the function arguments are invalid.
RZKERR_CB_BUSY	Indicates that the semaphore parameters could not be obtained at this time.

Example

This example stores the parameters of a semaphore with a handle of hSemaphore into RZK_SEMAPHOREPARAMS_t structure. The RZKGetSemaphoreParameters() API execution status is stored in the status variable.

```
extern RZK_SEMAPHOREHANDLE_t hSemaphore;
RZK_SEMAPHOREPARAMS_t SemParams;
RZK_STATUS_t status;
status = RZKGetSemaphoreParameters(hSemaphore,
&SemParams);
```

See Also

RZKCreateSemaphore
RZKDeleteSemaphore
RZKReleaseSemaphore
RZKAcquireSemaphore
RZK_SEMAPHOREPARAMS_t

Event Group APIs

Table 30 provides a quick reference to a number of event group APIs that are described in this subsection.

Table 30. Event Groups and Events API Quick Reference

RZKCreateEventGroup	RZKPendOnEventGroup
RZKDeleteEventGroup	RZKGetEventGroupParameters
RZKPostToEventGroup	

RZKCREATEEVENTGROUP

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZEventgroup.h"
```

Prototype

```
RZK_EVENTHANDLE_t RZKCreateEventgroup(
RZK_NAME_t szName [MAX_OBJECT_NAME_LEN],
RZK_MASK_t mEventMask);
```

Description

The `RZKCreateEventGroup()` API creates the event group with the specified mask (`mEventMask`) and returns a handle.

Argument(s)

`szName`	An ASCII string that specifies the name of the event group being created.
`mEventMask`	The events to mask.

Return Value(s)

The function returns a handle to the event group if it is created successfully or returns NULL. If NULL is returned, it sets the following error value in the current thread's thread control block. The `RZKGetErrorNum()` API is called to retrieve the error number stored in the thread control block.

`RZKERR_CB_UNAVAILABLE`	The control block is not available for the event group to be created.

Example

This example creates an event group with a name of *Zilog* and a group mask of `0x000b`. That is, threads can pend and post on this event group for three events only (`0x000b`). The handle of the created event group is stored in `hEventGroup`.

```
RZK_EVENTHANDLE_t hEventGroup;
hEventGroup = RZKCreateEventGroup((RZK_NAME_t [
])"Zilog",
  0x000B);
```

See Also

RZKDeleteEventGroup
RZKPostToEventGroup
RZKPendOnEventGroup
RZKGetEventGroupParameters
RZKGetErrorNum

RZKDELETEEVENTGROUP

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZEventgroup.h"
```

Prototype

```
RZK_STATUS_t RZKDeleteEventgroup(
RZK_EVENTHANDLE_t hEventGroup);
```

Description

The `RZKDeleteEventGroup()` API invalidates the event group handle and releases all of the threads waiting on this handle with appropriate status.

Argument(s)

`hEventGroup`	Specifies the handle to the event group to delete.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the operation is completed successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the handle to event group to be deleted is invalid.
`RZKERR_CB_BUSY`	Indicates that event group control block is used for exclusive purpose; for example, it is busy.

Example

This example deletes a previously-created event group, `hEventGroup` and makes the handle invalid. The `RZKDeleteEventGroup()` API execution status is stored in the `status` variable.

```
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_STATUS_t status;
status = RZKDeleteEventGroup(hEventGroup);
```

See Also

RZKCreateEventGroup
RZKPostToEventGroup
RZKPendOnEventGroup
RZKGetEventGroupParameters

RZKPOSTTOEVENTGROUP

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZEventgroup.h"
```

Prototype

```
RZK_STATUS_t RZKPostToEventgroup(
RZK_EVENTHANDLE_t hEventGroup,
RZK_EVENT_t eEvent,
RZK_EVENT_OPERATION_et etOperation);
```

Description

The `RZKPostToEventGroup()` API call sends the specified events to particular event groups. While posting the specified events it also performs operations such as AND, OR and XOR with the current events.

Argument(s)

`hEventGroup`	Specifies the handle to the EventGroup to which the event is posted.
`eEvent`	The event to be posted.

`etOperation`	Specifies the operation of the event with the existing value. The following operations can be performed with the events:	
	`EVENT_AND`	AND operation with the current events.
	`EVENT_OR`	OR operation with the current events.
	`EVENT_XOR`	XOR operation with the current events.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the operation is completed successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the handle of the event group is invalid.
`RZKERR_INVALID_OPERATION`	This error is returned if wrong etOperation is passed. `EVENT_CONSUME` must not be used in this API.
`RZKERR_CB_BUSY`	Indicates if the event group control block is used for an exclusive purpose; for example, it is busy.

Example 1

An event group is created with the name *Zilog* and an event mask of `0x0Fh`. The event handle is stored in `hEventGroup`. The following operation could be performed on the event group created.

```
#define EVENT_MASK  0x0F
RZK_EVENTHANDLE_t hEventGroup;
```

```
hEventGroup = RZKCreateEventGroup((RZK_NAME_t [
])"Zilog",
  EVENT_MASK);
```

Example 2

An event `0x05h` is posted to an event group, `hEventGroup`. The operation to be performed on the event is AND (`EVENT_AND`). The return value is stored in the `status` variable. The AND operation is performed with the existing events that are present in the event group and the result is used for any pending threads.

```
#define EVENT_TOBE_POSTED 0x05
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_STATUS_t status;
status = RZKPostToEventGroup(hEventGroup,
  EVENT_TOBE_POSTED,
  EVENT_AND);
```

Example 3

An event `0x05h` is posted to an event group, `hEventGroup`. The operation to be performed on the event is OR (`EVENT_OR`). The return value is stored in the `status` variable. The OR operation is performed with the existing events that are present in the event group and the result is used for any pending threads.

```
#define EVENT_TOBE_POSTED    0x05
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_STATUS_t status;
status = RZKPostToEventGroup(hEventGroup,
  EVENT_TOBE_POSTED,
  EVENT_OR);
```

Example 4

An event `0x05h` is posted to an event group, `hEventGroup`. The operation to be performed on the event is XOR (`EVENT_XOR`). The return value is stored in the `status` variable. The XOR operation is performed with

the existing events that are present in the event group and the result is used for any pending threads.

```
#define EVENT_TOBE_POSTED  0x05
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_STATUS_t status;
status = RZKPostToEventGroup(hEventGroup,
EVENT_TOBE_POSTED,
EVENT_XOR);
```

See Also

RZKCreateEventGroup
RZKDeleteEventGroup
RZKPendOnEventGroup
RZKGetEventGroupParameters
RZK_EVENT_OPERATION_et

RZKPENDONEVENTGROUP

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZEventgroup.h"
```

Prototype

```
RZK_STATUS_t RZKPendOnEventgroup(
RZK_EVENTHANDLE_t  hEventGroup,
RZK_EVENT_t        eEvent,
TICK_t             tBlockTime
RZK_EVENT_OPERATION_et etOperation);
```

Description

The calling thread uses the `RZKPendOnEventGroup()` API to either retrieve a logic operation (based on a sufficient combination of successful events from the event flag group), or perform a timed blocking pend until the requested events are set.

Argument(s)

`hEventGroup`	Specifies the handle to the EventGroup to pend on.
`eEvent`	The events to pend on.
`tBlockTime`	The pend time-out if events fail to occur. To pend infinitely, use `MAX_INFINITE_SUSPEND`.

`etOperation`	Specifies the operation of the event with the existing value.The following operations can be performed with the events:	
	`EVENT_AND`	AND operation performed with the received events.
	`EVENT_OR`	OR operation performed with the received events.
	`EVENT_CONSUME`	The event received is consumed. This operation can be combined with `EVENT_OR` or `EVENT_AND`. `EVENT_CONSUME` nullifies received events.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the operation is completed successfully.
`RZKERR_OBJECT_DELETED`	Indicates that the EventGroup is deleted.
`RZKERR_TIMEOUT`	The time-out occurred before the EventGroup could be retrieved.
`RZKERR_CB_BUSY`	If the EventGroup control block is used for an exclusive purpose; for example, it is busy.
`RZKERR_INVALID_HANDLE`	Handle passed to the routine is invalid.

Example 1

An event group is created with the name *Zilog* and an event mask of `0x07h`. The event handle is stored in `hEventGroup`. The following operation is performed on the event group created.

```
#define EVENT_MASK  0x07
RZK_EVENTHANDLE_t hEventGroup;
hEventGroup = RZKCreateEventGroup((RZK_NAME_t [
])"Zilog",
  EVENT_MASK);
```

Example 2

An event group is pended with a handle of `hEventGroup` for event `0x06h` and an operation of `EVENT_OR` for a finite period of 200 ticks. The thread calling this API first checks for the event to which it is pending, `0x06h`. If any of the bits are set in the value `0x06h`, then this thread unblocks and successfully returns a value. If the event(s) are not set, then the API blocks until the time-out period or when required events are set. The status of API execution is stored in the `status` variable.

```
#define EVENT_TOBE_PENDON 0x06
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_STATUS_t status;
status = RZKPendOnEventGroup(hEventGroup,
  EVENT_TOBE_PENDON,
  200,
  EVENT_OR);
```

Example 3

An event group is pended with a handle of `hEventGroup` for event `0x06h` and an operation of `EVENT_AND` for an infinite period. The thread calling this API first checks for the event to which it is pending, `0x06h`. If all of the bits are set in the value `0x06h`, then this thread unblocks and successfully returns a value. If the event(s) are not set, then the API blocks until the time-out period or when required events are set. The status of API execution is stored in the `status` variable.

```
#define EVENT_TO_PENDON    0x06
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_STATUS_t status;
```

```
status = RZKPendOnEventGroup(hEventGroup,
  EVENT_TO_PENDON,
  MAX_INFINITE_SUSPEND,
  EVENT_AND);
```

Example 4

An event group is pended with a handle of `hEventGroup` for event `0x06h` and an operation of `EVENT_AND` for infinite time. The thread calling this API first checks for the event to which it is pending, `0x06h`. If all of the bits are set in the value `0x06h`, then the event for the event group is reset and the calling thread unblocks and successfully returns a value. If the event(s) are not set, then the API blocks until the time-out period or when required events are set. The status of API execution is stored in the `status` variable.

```
#define EVENT_TOBE_PENDON    0x06
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_STATUS_t status;
status = RZKPendOnEventGroup(hEventGroup,
  EVENT_TOBE_PENDON,
  MAX_INFINITE_SUSPEND,
  EVENT_AND | EVENT_CONSUME);
```

See Also

RZKCreateEventGroup
RZKDeleteEventGroup
RZKPostToEventGroup
RZKGetEventGroupParameters
RZK_EVENT_OPERATION_et

RZKGETEVENTGROUPPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZEventgroup.h"
```

Prototype

```
RZK_STATUS_t RZKGetEventgroupParameters(
RZK_EVENTHANDLE_t        hEventGroup,
RZK_EVENTGROUPPARAMS_t   *pEventGroupParams);
```

Description

The `RZKGetEventGroupParameters()` API gets the event group parameters and stores them in `RZK_EVENTGROUPPARAMS_t` structure. See Table 61 on page 350 for members of the `RZK_EVENTGROUPPARAMS_t` structure.

Argument(s)

`hEventGroup`	Specifies the handle to the event group from which to get parameters.
`pEventGroupParams`	Pointer to a structure to receive the requested parameters.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the operation is completed successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the handle to the event group is invalid.

RZKERR_INVALID_ARGUMENTS	Indicates that the pEventGroupParams is invalid.
RZKERR_CB_BUSY	If the event group control block is used for an exclusive purpose; for example, it is busy.

Example

Parameters or information about a previously-created event group with an hEventGroup handle is stored onto the RZK_EVENTGROUPPARAMS_t structure. The API execution status is stored in the status variable.

```
extern RZK_EVENTHANDLE_t hEventGroup;
RZK_EVENTGROUPPARAMS_t EventGroupParams;
RZK_STATUS_t status;
status = RZKGetEventGroupParameters(hEventGroup,
                    &EventGroupParams);
```

See Also

RZKCreateEventGroup
RZKDeleteEventGroup
RZKPostToEventGroup
RZKPendOnEventGroup
RZK_EVENTGROUPPARAMS_t

Software Timer APIs

Table 31 provides a quick reference to a number of Software Timer APIs that are described in this subsection.

Table 31. Software Timer API Quick Reference

RZKCreateTimer	RZKDisableTimer
RZKDeleteTimer	RZKGetTimerParameters
RZKEnableTimer	RZKGetTimerResolution

RZKCREATETIMER

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZTimer.h"
```

Prototype

```
RZK_TIMERHANDLE_t RZKCreateTimer(
RZK_NAME_t szName [MAX_OBJECT_NAME_LENs],
FNP_TIMER_FUNCTION    pTimerFunction,
TICK_t tInitialDelay,
TICK_t tPeriod);
```

Description

The `RZKCreateTimer()` API creates a software timer with specified time-out values and an initial delay. It returns a unique handle. A created timer is activated only after a `RZKEnableTimer()` function is called.

Argument(s)

`szName`	This parameter specifies the name to be used for identifying the timer.
`pTimerFunction`	This parameter specifies a pointer to the entry point function for the timer. The timer entry function must feature the following prototype: void `MyTimerEntryFunction(void);`
`tInitialDelay`	This parameter specifies the initial delay (in ticks) for the timer.
`tPeriod`	This parameter specifies a cyclic period (in ticks) for the timer after an initial delay.

Return Value(s)

The function returns a handle to the Timer if it is created successfully; else it returns NULL. If NULL is returned, it sets one of the following error values in the thread control block of the current thread. The `RZKGetErrorNum()` API is called to retrieve the error number stored in the thread control block.

`RZKERR_INVALID_ARGUMENTS`	Indicates that some of the parameters were incorrectly passed.
`RZKERR_CB_UNAVAILABLE`	Indicates that the system is unable to allocate the required control block.

Example

This example creates a software timer with handler function as `MyTimerHandler`, with a name as *Zilog*, initial delay as 5 ticks and time-out period of 10 ticks. The timer handle is stored in `hTimer`.

```
#define INITIAL_DELAY    5
#define TIMEOUT_PERIOD   10
extern void MyTimerHandler(void);
RZK_TIMERHANDLE_t hTimer;
hTimer = RZKCreateTimer((RZK_NAME_t [ ])"Zilog",
  MyTimerHandler,
  INITIAL_DELAY,
  TIMEOUT_PERIOD);
```

Note: A number of limitations apply to the `pTimerFunction` pointer passed for this function, as it is directly invoked as a subroutine from the timer ISR. There can be several timers created by the collection of all running threads and applications. Ensure that all these functions run one after the other (an extreme case), the timer ISR can complete in a reasonable amount of time and not interfere with system functionality.

No arguments are passed to the `pTimerFunction` function. The `RZKCreateTimer()` function must not consume more than a specified thread's stack space (including any function/procedure calls inside) and must be a compact, efficient routine that is designed to be reentrant and nonblocking. The timer callback function works on timer thread context.

Any assembly routine called from this routine is expected to save and restore registers on entry and exit.

The `RZKGetTimerResolution()` API can be used to get the resolution of timer used by RZK in terms of ticks per second. The ticks per second can be used to calculate the time-out period in seconds or milliseconds.

See Also

RZKDeleteTimer
RZKDisableTimer
RZKGetErrorNum
RZKEnableTimer
RZKGetTimerParameters
RZKGetTimerResolution

RZKDELETETIMER

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZTimer.h"
```

Prototype

```
RZK_STATUS_t RZKDeleteTimer(
RZK_TIMERHANDLE_t hTimer);
```

Description

The `RZKDeleteTimer()` API deletes a software timer specified by the handle of the timer object.

Argument(s)

`hTimer`	Specifies the handle of the timer for deletion.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the timer object is deleted successfully.
`RZKERR_INVALID_HANDLE`	Indicates that the `hTimer` parameter is invalid.
`RZKERR_CB_BUSY`	Indicates that the timer block is in exclusive use; for example, it is busy.
`RZKERR_OBJECT_IN_USE`	Indicates that the timer block is in use; for example, not yet disabled. The timer object must be disabled before it is deleted.

Example

A previously-created timer, `hTimer` is deleted. The API execution status is stored in the `status` variable.

```
extern RZK_TIMERHANDLE_t hTimer;
RZK_STATUS_t status;
status = RZKDeleteTimer(hTimer);
```

See Also

RZKCreateTimer
RZKDisableTimer
RZKGetTimerResolution
RZKEnableTimer
RZKGetTimerParameters

RZKENABLETIMER

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZTimer.h"
```

Prototype

```
RZK_STATUS_t RZKEnableTimer(
RZK_TIMERHANDLE_t hTimer);
```

Description

The `RZKEnableTimer()` API enables the software timer associated with the input timer handle. The timer pointed to by the timer handle `hTimer` is restarted, if it is already disabled, by using `RZKDisableTimer()` when this API is called.

Argument(s)

`hTimer`	This parameter specifies the handle of the timer.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the timer object is successfully enabled.
`RZKERR_INVALID_HANDLE`	This error occurs when the hTimer parameter is invalid.
`RZKERR_CB_BUSY`	The timer control block is used for an exclusive purpose; for example, it is busy.
`RZKERR_OBJECT_IN_USE`	If the timer is not in disabled state.

Example

A previously-created disabled timer, `hTimer`, is enabled. The API execution status is stored in the `status` variable.

```
extern RZK_TIMERHANDLE_t hTimer;
RZK_STATUS_t status;
status = RZKEnableTimer(hTimer);
```

See Also

RZKCreateTimer
RZKDeleteTimer
RZKDisableTimer
RZKGetTimerParameters
RZKGetTimerResolution

RZKDISABLETIMER

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZTimer.h"
```

Prototype

```
RZK_STATUS_t RZKDisableTimer(
RZK_TIMERHANDLE_t hTimer);
```

Description

The `RZKDisableTimer()` API disables a software timer when it is provided with the timer object handle.

Argument(s)

`hTimer`	This parameter specifies the handle of the timer.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the timer is successfully disabled.
`RZKERR_INVALID_HANDLE`	This error occurs when the `hTimer` parameter is invalid.
`RZKERR_CB_BUSY`	If the timer control block is used for an exclusive purpose; for example, it is busy.
`RZKERR_INVALID_OPERATION`	If the timer is already disabled.

Example

A previously enabled timer, `hTimer`, is disabled. The API execution status is stored in the `status` variable.

```
extern RZK_TIMERHANDLE_t hTimer;
RZK_STATUS_t status;
status = RZKDisableTimer(hTimer);
```

See Also

RZKCreateTimer
RZKDeleteTimer
RZKEnableTimer
RZKGetTimerParameters
RZKGetTimerResolution

RZKGETTIMERPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZTimer.h"
```

Prototype

```
RZK_STATUS_t RZKGetTimerParameters(
RZK_TIMERHANDLE_t hTimer,
RZK_TIMERPARAMS_t *pTimerParams);
```

Description

The `RZKGetTimerParameters()` API obtains the parameters of the software timer and stores them in `RZK_TIMERPARAMS_t` structure. See Table 62 on page 351 for the members of `RZK_TIMERPARAMS_t` structure.

Argument(s)

`hTimer`	This parameter specifies the handle of the timer.
`pTimerParams`	This parameter specifies a pointer to the structure for receiving the Timer parameters.

Return Value(s)

This function returns the status after retrieving the Timer parameters. In case of an error, one of the following values is returned.

RZKERR_SUCCESS	Indicates that the timer parameters are successfully stored.
RZKERR_INVALID_HANDLE	This error occurs when the hTimer parameter is invalid.
RZKERR_INVALID_ARGUMENTS	Indicates that a parameter is incorrectly passed.
RZKERR_CB_BUSY	Indicates that the timer control block is used for an exclusive purpose; for example, it is busy.

Example

A previously-created software timer with a handle of hTimer is stored into the TimerParams structure. The API execution status is also stored in the status variable.

```
extern RZK_TIMERHANDLE_t hTimer;
RZK_TIMERPARAMS_t TimerParams;
RZK_STATUS_t status;
status = RZKGetTimerParameters(hTimer,
  &TimerParams);
```

See Also

RZKCreateTimer
RZKDeleteTimer
RZKEnableTimer
RZKDisableTimer
RZK_TIMERPARAMS_t
RZKGetTimerResolution

RZKGETTIMERRESOLUTION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZTimer.h"
```

Prototype

```
TICK_t RZKGetTimerResolution()
```

Description

This macro provides the calling thread with the resolution of the timer in ticks per second.

Argument(s)

None.

Return Value(s)

The macro returns the number of ticks per second.

Example

The timer resolution of the system is stored into the `timres` variable.

```
TICK_t timres;
timres = RZKGetTimerResolution();
```

See Also

RZKCreateTimer
RZKDeleteTimer
RZKEnableTimer
RZKDisableTimer
RZKGetTimerParameters

Clock APIs

Table 32 provides reference to the different Clock APIs. The following sections provide description for each Clock API.

Table 32. Clock API Quick Reference

RZKGetClock
RZKSetClock

RZKGETCLOCK

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZClock.h"
```

Prototype

```
void RZKGetClock(
RZK_CLOCKPARAMS_t    *pClockParams);
```

Description

The `RZKGetClock()` API gets the current system time-elapsed value from the most recent board reset/most recent `RZKSetClock()` call invocation. This time-elapsed value is obtained in a clock parameter structure that contains year/month/date/hour/minute/seconds. This API is useful to compute the elapsed time between two events. See Table 67 on page 356 for members of `RZK_CLOCKPARAMS_t` structure.

Argument(s)

`pClockParams`	This parameter is a pointer to structure that holds the time elapsed values.

Return Value(s)

This function returns the current clock time.

Example

The clock parameters are stored into the `ClockParams` structure.

```
RZK_CLOCKPARAMS_t Clockparams;
RZKGetClock(&Clockparams);
```

See Also

RZKSetClock

RZK_CLOCKPARAMS_t

RZKSETCLOCK

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZClock.h"
```

Prototype

```
void RZKSetClock (
RZK_CLOCKPARAMS_t *pClockParam);
```

Description

The `RZKSetClock()` API sets the system time to the input value from a clock parameter structure. The input structure contains year/month/date/hour/minute/second information. See Table 67 on page 356, for members of the `RZK_CLOCKPARAMS_t` structure.

Argument(s)

`pClockParams`	This parameter is a pointer to structure that holds the time to set.

Return Value(s)

None.

Example

The system clock parameters that are present in the `RZK_CLOCKPARAMS_t` structure are set.

```
RZK_CLOCKPARAMS_t clkParams = {
  1990, /** year */
```

```
1, /** month */
1, /** day */
10, /** hour */
50, /** minutes */
27}; /** seconds */
RZKSetClock(&clkParams);
```

Note: The reference value for the RZK Clock is the year 1970.

See Also

RZKGetClock

RZK_CLOCKPARAMS_t

Partition APIs

Table 33 provides a quick reference to the memory/partition APIs. The following sections provide description for each memory/partition API.

Table 33. Memory Partition API Quick Reference

RZKCreatePartition	RZKFreeFixedSizeMemory
RZKDeletePartition	RZKGetPartitionParameters
RZKAllocFixedSizeMemory	

RZKCREATEPARTITION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMemory.h"
```

Prototype

```
RZK_PARTITIONHANDLE_t RZKCreatePartition(
RZK_NAME_t szName [MAX_OBJECT_NAME_LEN],
RZK_PTR_t pMemory,
UINT uMemoryBlocks,
UINT uBlockSize);
```

Description

The `RZKCreatePartition()` API call creates a memory partition at the specified pointer. RZK uses the memory pointer to allocate fixed-length memory blocks. The size of the memory is calculated as follows:

```
((uMemoryBlocks * uBlockSize) + (uMemoryBlocks * size
of pointer)).
```

Argument(s)

`szName`	Specifies the name used to identifying the partition area.
`pMemory`	Specifies a pointer to the memory area, where the partition must be created.
`uMemoryBlocks`	Specifies the number of memory blocks used as a part of this partition.
`uBlockSize`	Specifies the size of each memory block.

Return Value(s)

This function returns a handle to the partition if it is created successfully or returns `NULL`. If `NULL` is returned it sets one of the following error values in the thread control block of the current thread. `RZKGetErrorNum()` API is called to retrieve the error number stored in the thread control block.

`RZKERR_INVALID_ARGUMENTS`	This error indicates that some of the parameters were incorrectly passed.
`RZKERR_CB_UNAVAILABLE`	This error indicates that the system is unable to allocate the required control block.

Example

A partition is created with the name *Zilog* containing `MEMORY_BLOCKS` blocks of `BLOCK_SIZE` length. Memory is allocated with required bytes. The partition handle is stored in `hPartition`.

```
#define MEMORY_BLOCKS   10
#define BLOCK_SIZE    100

unsigned char memBuffer[ (MEMORY_BLOCKS * BLOCK_SIZE)
+ (MEMORY_BLOCKS * sizeof(void *)) ];
RZK_PARTITIONHANDLE_t hPartition;
hPartition = RZKCreatePartition((RZK_NAME_t [
])"Zilog",
  memBuffer,
  MEMORY_BLOCKS,
  BLOCK_SIZE);
```

See Also

RZKDeletePartition	RZKAllocFixedSizeMemory
RZKFreeFixedSizeMemory	RZKGetPartitionParameters
RZKGetErrorNum	RZKQueryMem

RZKDELETEPARTITION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMemory.h"
```

Prototype

```
RZK_STATUS_t RZKDeletePartition(
RZK_PARTITIONHANDLE_t hPartition);
```

Description

The `RZKDeletePartition` API call deletes the memory partition specified by the handle if (and only if) all blocks in it are currently unallocated/freed.

Argument(s)

`hPartition`	This parameter specifies the handle of the partition area for deletion,

Return Value(s)

`RZKERR_SUCCESS`	The partition is deleted and API executed successfully.
`RZKERR_INVALID_HANDLE`	This error occurs when the hPartition parameter is invalid.
`RZKERR_CB_BUSY`	If the partition control block is used for an exclusive purpose; for example, it is busy.
`RZKERR_OBJECT_IN_USE`	The segments allocated from this partition are not freed.

Example

A previously-created partition, `hPartition`, is deleted, making the handle invalid. The API execution status is stored in the `status` variable.

```
extern RZK_PARTITIONHANDLE_t hPartition;
RZK_STATUS_t status;
status = RZKDeletePartition(hPartition);
```

See Also

RZKCreatePartition
RZKAllocFixedSizeMemory
RZKFreeFixedSizeMemory
RZKGetPartitionParameters

RZKALLOCFIXEDSIZEMEMORY

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMemory.h"
```

Prototype

```
RZK_PTR_t RZKAllocFixedSizeMemory(
RZK_PARTITIONHANDLE_t hPartition);
```

Description

The `RZKAllocFixedSizeMemory()` API call allocates a fixed memory size from the specified partition and returns a pointer to the memory block.

Argument(s)

`hPartition`	This parameter specifies the handle of the partition from which memory block must be allocated.

Return Value(s)

This function returns a pointer to the newly created memory block. In case of an error, one of the following values is set in the thread control block of the thread making this call. `RZKGetErrorNum()` API is called to retrieve the error number stored in the thread control block.

RZKERR_INVALID_HANDLE	This error occurs when the hPartition parameter is invalid.
RZKERR_OUT_OF_MEMORY	This error occurs when there is no available space for creating another memory block. Number of memory blocks allocated is greater than the memory blocks at the time of partition creation.
RZKERR_CB_BUSY	The timer control block is used for an exclusive purpose; for example, it is busy.

Example

A memory block is allocated from a partition with a handle that is stored in hPartition. The allocated memory address is stored into the pMemBuf variable.

```
extern RZK_PARTITIONHANDLE_t hPartition;
void *pMemBuf;
pMemBuf = RZKAllocFixedSizeMemory(hPartition);
```

See Also

RZKCreatePartition
RZKDeletePartition
RZKFreeFixedSizeMemory
RZKGetPartitionParameters
RZKGetErrorNum

RZKFREEFIXEDSIZEMEMORY

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMemory.h"
```

Prototype

```
RZK_STATUS_t RZKFreeFixedSizeMemory(
RZK_PARTITIONHANDLE_t   hPartition,
RZK_PTR_t pBlock);
```

Description

The `RZKFreeFixedSizeMemory()` API call frees fixed size of memory and returns the call status.

Argument(s)

`hPartition`	This parameter specifies the handle of the partition area required for freeing a memory block.
`pBlock`	This parameter specifies the pointer to the allocated block of memory.

Return Value(s)

`RZKERR_SUCCESS`	If the block is successfully returned to the partition.
`RZKERR_INVALID_HANDLE`	This error occurs when the `hPartition` parameter is invalid.

RZKERR_INVALID_ARGUMENTS	This error indicates that the pBlock parameter is not a valid block address.
RZK_INVALID_OPERATION	This error occurs when a thread tries to free a block when all of the blocks in the specified partition are already deleted.
RZKERR_CB_BUSY	The partition control block is used for exclusive purpose; for example, it is busy.

Example

A previously-allocated memory block pMemBuf is released (freed) from its partition with a handle that is stored in hPartition. The API execution status is stored in the status variable.

```
extern RZK_PARTITIONHANDLE_t hPartition;
extern void *pMemBuf;
RZK_STATUS_t status;
status = RZKFreeFixedSizeMemory(hPartition,
  pMemBuf);
```

See Also

RZKCreatePartition
RZKDeletePartition
RZKAllocFixedSizeMemory
RZKGetPartitionParameters

RZKGETPARTITIONPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMemory.h"
```

Prototype

```
RZK_STATUS_t RZKGetPartitionParameters(
RZK_PARTITIONHANDLE_t    hPartition,
RZK_PARTITIONPARAMS_t    *pPartitionParams);
```

Description

The `RZKGetPartitionParameters()` API gets the parameters of the specified partition and stores them in the `RZK_PARTITIONPARAMS_t` structure. See Table 63 on page 352 for members of the `RZK_PARTITIONPARAMS_t` structure.

Argument(s)

`hPartition`	This parameter specifies the partition area handle required for retrieving the partition parameters.
`pPartitionParams`	This parameter specifies a pointer to the structure for receiving the partition parameters.

Return Value(s)

RZKERR_SUCCESS	If the partition parameters were successfully obtained.
RZKERR_INVALID_HANDLE	This error occurs when the hPartition parameter is invalid.
RZKERR_INVALID_ARGUMENTS	This error indicates that a parameter passed is incorrect.
RZKERR_CB_BUSY	If the partition control block is used for an exclusive purpose; for example, it is busy.

Example

Information related to a partition's memory blocks and block size, with a handle that is stored in hPartition, is retrieved into the RZK_PARTITIONPARAMS_t structure.

```
extern RZK_PARTITIONHANDLE_t hPartition;
RZK_PARTITIONPARAMS_t PartitionParams;
RZK_STATUS_t status;
status = RZKGetPartitionParameters(hPartition,
  &PartitionParams);
```

See Also

RZKCreatePartition
RZKDeletePartition
RZKAllocFixedSizeMemory
RZKFreeFixedSizeMemory
RZK_PARTITIONPARAMS_t

Region APIs

Table 34 provides a quick reference to a number of region APIs that are described in this subsection.

Table 34. Region API Quick Reference

RZKCreateRegion	RZKGetRegionParameters
RZKDeleteRegion	malloc
RZKAllocSegment	free
RZKFreeSegment	

RZKCREATEREGION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZMemory.h"
```

Prototype

```
RZK_REGIONHANDLE_t RZKCreateRegion(
RZK_NAME_t szName[MAX_OBJECT_NAME_LEN],
void     *RnAddr,
COUNT_t  uLength,
COUNT_t  uUnit_Size,
UINT     uRnDelete,
RZK_RECV_ATTRIB_et etAttrib);
```

Description

The `RZKCreateRegion()` API creates a region and returns a handle to the region control block.

Argument(s)

`szName`	Specifies the name of the thread/ASCII string.
`RnAddr`	Specifies the memory address for the region.
`uLength`	Specifies the total memory size for the region.
`uUnit_Size`	Specifies the minimum allocatable memory size from the region. This minimum unit size is 16 bytes.

`uRnDelete`	Specifies whether the region releases the memory allocated when it is deleted. If the value is 1, RZK frees memory when `RZKDeleteRegion()` is called. If this value is 0, you must release all allocated memory before calling `RZKDeleteRegion()`.
`etAttrib`	Specifies the receiving attributes for the threads waiting on the region and can be one of the following.*

`RECV_ORDER_FIFO`	Receive order is FIFO
`RECV_ORDER_PRIORITY`	Receive order is priority.

Note: *The default receive order is `RECV_ORDER_PRIORITY`.

Return Value(s)

This API returns a handle to the region if it is created successfully or returns NULL. If NULL is returned, one of the following error values is set in the current thread control block. `RZKGetErrorNum()` API is called to retrieve the error number stored in the thread control block.

`RZKERR_INVALID_ARGUMENTS`	Indicates specified region parameter is invalid. This error occurs when `RnAddr` is NULL, `uLength` is zero, or when a unit size is less than 16.
`RZKERR_CB_UNAVAILABLE`	Indicates that a control block could not be allocated for the region. The number of regions created is more than the value of `MAX_REGIONSH`.

Example

The `RZKCreateRegion()` API call creates:

- A region with the name `Zilog`

- A pointer to the memory area where the region must be created, with
  - A total memory size set to 100
  - A minimum allocatable size of 20
  - A zero specifying that it is not necessary for the region to release the allocatable memory while deleting
  - A receive order of `RECV_ORDER_FIFO`

The created region handle is stored in `hRegion`.

```
#define TOTAL_MEM_FOR_REGION_ALLOC100
#define MINIMUM_ALLOC_SIZE20
#define AUTO_RELEASE_MEM_ENABLE1
#define AUTO_RELEASE_MEM_DISABLE0

RZK_REGIONHANDLE_t hRegion;
unsigned char memBuf[ TOTAL_MEM_FOR_REGION_ALLOC ];
hRegion = RZKCreateRegion((RZK_NAME_t [ ])"Zilog",
  memBuf,
  TOTAL_MEM_FOR_REGION_ALLOC,
  MINIMUM_ALLOC_SIZE,
  AUTO_RELEASE_MEM_DISABLE,
  RECV_ORDER_FIFO);
```

See Also

RZKDeleteRegion
RZKAllocSegment
RZKGetRegionParameters
RZKFreeSegment
RZKGetErrorNum
RZK_RECV_ATTRIB_et

RZKDELETEREGION

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZRegion.h"
```

Prototype

```
RZK_STATUS_t RZKDeleteRegion(
RZK_REGIONHANDLE_t  hRegion);
```

Description

The `RZKDeleteRegion()` API deletes the region specified by the handle.

Argument(s)

`hRegion`	Specifies the handle of the region to delete.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	The handle of the region is invalid.
`RZKERR_CB_BUSY`	Indicates that another thread uses the region exclusively.
`RZKERR_INVALID_OPERATION`	Indicates operation is invalid and you tried to delete a region when threads were blocked.

Example

A previously-created region with a handle of `hRegion` is deleted, making the handle invalid. API execution status is stored in the `status` variable.

```
extern RZK_REGIONHANDLE_t hRegion;
RZK_STATUS_t status;
status = RZKDeleteRegion(hRegion);
```

See Also

RZKCreateRegion
RZKFreeSegment
RZKAllocSegment
RZKGetRegionParameters

RZKALLOCSEGMENT

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZRegion.h"
```

Prototype

```
RZK_PTR_t RZKAllocSegment(
RZK_REGIONHANDLE_t hRegion,
COUNT_t     uSize,
TICK_t      tBlockTime);
```

Description

This function allocates a variable size of memory from the specified region (if available) and returns a pointer to the memory block.

Argument(s)

`hRegion`	Specifies the handle of the region area required for allocating a memory block.
`uSize`	Specifies the memory size to be allocated.
`tBlockTime`	Specifies the time for which the thread must block on the region, if the memory is not available. Use `INFINITE_SUSPEND` to block infinitely.

Return Value(s)

The `RZKAllocSegment()` API returns the pointer to the memory location that is allocated, on success, or returns `NULL`. If `NULL` is returned, one of the following error values is set in the current thread control block.

`RZKGetErrorNum()` API is called to retrieve the error number stored in the thread control block.

`RZKERR_INVALID_HANDLE`	This error occurs when the region handle is invalid.
`RZKERR_INVALID_ARGUMENTS`	Arguments to the function are invalid.
`RZKERR_SCB_UNAVAILABLE`	There are not enough segment control blocks available. Maximum number of allocations for regions (system wide) are used.
`RZKERR_CB_BUSY`	A thread is using the region exclusively.
`RZKERR_TIMEOUT`	This error occurs when there is no memory to allocate and the time of blocking expires.
`RZKERR_OBJECT_DELETED`	This error occurs when the region is deleted while the thread is blocked on it.

Example

A variable size of memory (5 bytes each) is allocated from the specified region with a handle that is stored in `hRegion`. If memory is not available, the thread waits for 10 ticks.

```
#define MEM_SIZE_TOBE_ALLOCATED     5
extern RZK_REGIONHANDLE_t hRegion;
RZK_PTR_t hSegment;
hSegment = RZKAllocSegment(hRegion,
  MEM_SIZE_TOBE_ALLOCATED,
  10);
```

See Also

RZKCreateRegion
RZKDeleteRegion
RZKGetRegionParameters
RZKFreeSegment
RZKGetErrorNum

RZKFREESEGMENT

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZRegion.h"
```

Prototype

```
RZK_STATUS_t RZKFreeSegment(
RZK_REGIONHANDLE_t hRegion,
RZK_PTR_t pSegment);
```

Description

The `RZKFreeSegment()` API frees the allocated memory from the specified region. It frees the variable memory size from the specified segment of the specified region handle.

Argument(s)

`hRegion`	Specifies the handle of the region area required for allocating a memory block.
`pSegment`	Pointer to the memory to be freed.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	This error occurs when the region handle is invalid.

RZKERR_INVALID_ARGUMENTS	The arguments to the function are invalid.
RZKERR_CB_BUSY	When a thread is using the region exclusively.

Example

A memory segment of `hSegment` that is allocated previously from a region with a handle that is stored in `hRegion` is released (freed). The API execution status is stored in `status`.

```
extern RZK_REGIONHANDLE_t hRegion;
extern RZK_PTR_t hSegment;
RZK_STATUS_t status;
status = RZKFreeSegment(hRegion,
  hSegment);
```

See Also

RZKCreateRegion
RZKDeleteRegion
RZKGetRegionParameters
RZKAllocSegment

RZKGETREGIONPARAMETERS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZRegion.h"
```

Prototype

```
RZK_PTR_t RZKGetRegionParameters(
RZK_REGIONHANDLE_t    hRegion,
RZK_REGIONPARAMS_t    *pRegionParams);
```

Description

The `RZKGetRegionParameters()` API returns the region parameters into `RZK_REGIONPARAMS_t` structure which you provided. See Table 64 on page 353 for members of structure `RZK_REGIONPARAMS_t`.

Argument(s)

`hRegion`	Specifies the handle of the region area required for allocating a memory block.
`pRegionParams`	Pointer to the structure type `RZK_REGIONPARAMS_t`, which receives memory region parameters.

Return Value(s)

`RZKERR_SUCCESS`	The operation completed successfully.
`RZKERR_INVALID_HANDLE`	This error occurs when the region handle is invalid.

RZKERR_INVALID_ARGUMENTS	The arguments to the function are invalid.
RZKERR_CB_BUSY	When a thread is using the region exclusively.

Example

The parameters of a region with a handle that is stored in hRegion are retrieved and stored into the RZK_REGIONPARAMS_t structure. The API execution status is stored status variable.

```
extern RZK_REGIONHANDLE_t hRegion;
RZK_REGIONPARAMS_t regionParams;
RZK_STATUS_t status
status= RZKGetRegionParams(hRegion,
  &regionParams);
```

See Also

RZKCreateRegion
RZKDeleteRegion
RZKFreeSegment
RZKAllocSegment
RZK_PARTITIONPARAMS_t

Note: A macro named MAX_REGION_TABH is provided in the RZK_Conf.c header file. This macro indicates the maximum number of allocations that can be made using all region handles defined in the application. For example, an application creates 10 regions with varying maximum memory capacity. If five allocations are made in each region, there are 50 allocations in total. Minimum value of MAX_REGION_TABH must be 50. During debug phase, this value must be quite high. After the application is totally developed, this value can be modified depending on the number of allocations in the application.

RZKQUERYMEM

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZRegion.h"
```

Prototype

```
COUNT_t RZKQueryMem(RZK_REGIONHANDLE_t hRegion)
```

Description

The `RZKQueryMem()` API returns the size of the largest portion of free memory in the memory blocks of the passed region handles.

Argument(s)

`hRegion`	Region handle.

Return Value(s)

The `RZKQueryMem()` API returns the number of bytes available as free memory in the region.

Example

This example gets the largest portion of free memory available.

```
extern RZK_REGIONHANDLE_t hRegion;
COUNT_t nfree_bytes;

nfree_bytes = RZKQueryMem( hRegion ) ;

printf("\n free memory available is : %d", nfree_bytes
) ;
```

See Also

RZKCreateRegion
RZKDeleteRegion
RZKFreeSegment
RZKAllocSegment
RZKGetErrorNum
RZKGetRegionParameters
RZK_RECV_ATTRIB_et

MALLOC

Include

```
#include <stdio.h>
#include <stdlib.h>
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZRegion.h"
```

Prototype

```
void * malloc( size_t size ) ;
```

Description

The `malloc()` API allocates the size of memory, in bytes, from the heap through RZK regions. If this required memory is not available, this API returns an error.

Note: This API is a different implementation than the ZDSII tool implementation of `malloc()`.

Argument(s)

`size` The number of bytes of memory to allocate.

Return Value(s)

The `malloc` API returns the address of the starting memory location if allocated, otherwise it returns NULL, indicating an error in memory allocation or that memory has been exhausted.

Note: Before using this API, the `RZK_KernelInit()` API must first be called from the `main()` function. Only then the `malloc()` API can be called.

Example

This example allocates 50 bytes of memory from the heap.

```
void *ptr;
ptr = malloc( 50 );
if( ptr == NULL )
  printf("\nMemory is not allocated");
else
  printf("\nMemory is allocated and the address is
:%d",ptr ) ;
```

See Also

free

FREE

Include

```
#include <stdio.h>
#include <stdlib.h>
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZRegion.h"
```

Prototype

```
void free( void *ptr ) ;
```

Description

The `free()` API frees the previously-allocated memory, the starting address location of which is present in the argument. If the pointer is invalid, this API does not return an error.

Note: Before using this API, the `RZK_KernelInit()` API must first be called from the `main()` function. Only then the `free()` API can be called. Note that this API is a different implementation than the ZDSII tool implementation of `malloc()`.

Argument(s)

`ptr`	The starting address of the memory location that was previously allocated.

Return Value(s)

None.

Example

This example frees the memory allocated, the starting address of which is present in the `ptr` variable.

```
extern void *ptr ; // Assuming the memory was
//allocated earlier
free( ptr ) ;
```

See Also

malloc

Interrupt APIs

Table 35 provides a quick reference for the Interrupt APIs. The following sections provide description for each Interrupt API.

Table 35. Interrupt API Quick Reference

RZKInstallInterruptHandler	RZKISRProlog
RZKEnableInterrupts	RZKISREpilog
RZKDisableInterrupts	

RZKINSTALLINTERRUPTHANDLER

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZInterrupt.h"
```

Prototype

```
RZK_FNP_ISR RZKInstallInterruptHandler(
RZK_FNP_ISR                 pHandlerFunc,
RZK_INTERRUPT_NUM_t         nInterruptNum);
```

Description

The `RZKInstallInterruptHandler()` API call installs an interrupt handler in the Interrupt Vector Table.

Argument(s)

`pHandlerFunc`	Specifies the address of the interrupt handler function. The interrupt handler function prototype must be shown as void MyIntrHandler(void);
`nInterruptNum`	Offset in the interrupt vector table for the interrupt for which the interrupt handler must installed. For specific interrupt offset, refer to the documents about target specific product specification listed in the Related Documents section on page xv.

Return Value(s)

This function returns the interrupt handler that is installed. During initialization, the default interrupt handlers are installed. The default interrupt

handlers print *uninitialized interrupt* and bring the processor to the HALT mode.

Example

An interrupt handler function `MyIntHandler` for the `timer()` interrupt (offset 0x54) of the eZ80F91 MCU is installed. The `prevHandler` stores the current interrupt handler stored in the address.

```
extern void MyIntHandler(void);
RZK_FNP_ISR prevHandler;
prevHandler =
RZKInstallInterruptHandler(MyIntHandler,0x54);
```

See Also

RZKEnableInterrupts

RZKDisableInterrupts

RZKENABLEINTERRUPTS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZInterrupt.h"
```

Prototype

```
void RZKEnableInterrupts(
UINTRMASK     mInterruptMask);
```

Description

The `RZKEnableInterrupts()` API enables the interrupt(s) that were disabled using the `RZKDisableInterrupts()` API, as per the `mInterruptMask` parameter. In effect this API is used to restore the interrupt status previous to the `RZKDisableInterrupts()` call.

Argument(s)

`mInterruptMask`	Specifies the mask for enabling the interrupts.

Return Value(s)

None.

Example

The interrupt status that is stored in the `intrStatus` variable is enabled.

```
extern UINTRMASK intrStatus;
RZKEnableInterrupts(intrStatus);
```

See Also

RZKInstallInterruptHandler
RZKDisableInterrupts

RZKDISABLEINTERRUPTS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZInterrupt.h"
```

Prototype

```
UINTRMASK RZKDisableInterrupts();
```

Description

The `RZKDisableInterrupts` API disables interrupts and returns the interrupt status.

Caution: Take care while using this function as it can affect the interrupt latency time.

Argument(s)

None.

Return Value(s)

The function returns the current Interrupt Mask.

Example

The interrupt status is stored in the `intrStatus` variable after the interrupts are disabled.

```
UINTRMASK intrStatus;
intrStatus = RZKDisableInterrupts();
```

See Also

RZKEnableInterrupts RZKInstallInterruptHandler

RZKISRPROLOG

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZInterrupt.h"
```

Prototype

```
void RZKISRProlog(void);
```

Description

RZK is typically unaware of a user-defined interrupt event. When such an interrupt occurs, the processor directly calls a user defined ISR installed against the relevant vector. If an RZK API is invoked directly from within an ISR, it can result in RZK performing a context switch or some other internal processing that can, in turn result in unpredictable performance of the RZK.

`RZKISRProlog()` and `RZKISREpilog()` are two assembly routines designed to facilitate the invocation of RZK services such as sending a message, using semaphores and so forth from within an ISR. `RZKISR-Prolog()` call ensures that the RZK places itself in a well defined state and all subsequent RZK API calls are prevented from invoking any extraneous RZK functioning. For example, a send message operation simply queues the message and returns to the ISR immediately.

`RZKISRProlog()` must be called exactly one time in an ISR prior to any other RZK API call from within that ISR, typically at the beginning of that ISR. This routine can be called from an assembly routine by prefixing the name with an underscore as `_RZKISRProlog` or can be called from a C routine by just calling the function name. This routine must be called if you need nested interrupts. See Appendix C. Interrupt Handling on page 359 for more details.

If an `RZKISRProlog()` call is made from within the ISR, the final call before termination of the ISR must be a call to `RZKISREpilog()`.

Argument(s)

None

Return Value(s)

None

Note: Registers used within this routine are saved and restored.

Example

See Appendix C. Interrupt Handling on page 359.

See Also

RZKISREpilog

RZKISREPILOG

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZInterrupt.h"
```

Prototype

```
void RZKISREpilog(void);
```

Description

The `RZKISREpilog()` call ensures that the RZK resumes normal functioning after the `RZKISRProlog()` API is called. This API is called only if the `RZKISRProlog()` API is called previously. `RZKISREpilog()` is called within an ISR after all other RZK API calls are made, typically at the end of the ISR.

`RZKISREpilog()` can be called from an assembly routine by prefixing the name with an underscore `_RZKISRProlog` or can be called from a C routine by calling the function name.

Calling this API causes RZK to resume normal operation and the control flow gets passed from ISR back to the RZK scheduler. If the interrupted thread continues to be the highest priority thread after servicing the interrupt (that is, if a higher-priority thread becomes ready as a result of an action within the ISR), control passes to the higher-priority thread and the previously interrupted thread is returned to the ready queue.

The user code must execute activities such as restoring any saved registers, reenabling the interrupts and other such activities, before this function is called.

Argument(s)

None.

Return Value(s)

None.

Note: Registers used within this routine are saved and restored.

Example

See Appendix C. Interrupt Handling on page 359.

See Also

RZKISRProlog

Device Driver Framework APIs

Table 36 provides a quick reference for the Device Driver Framework RZK APIs. The following sections provide description for each of these APIs.

Note: The APIs listed in Table 36 inturn call the device-specific functions that perform basic operations such as read/write/init/ioctl/close/uninit. It is therefore advised to use these APIs instead of using device driver functions such as `UARTOpen`, `UARTRead` etc., for the UART devices.

Table 36. Device Driver Framework API Quick Reference

RZKDevAttach	RZKDevWrite
RZKDevDetach	RZKDevIOCTL
RZKDevOpen	RZKDevGetc
RZKDevClose	RZKDevPutc
RZKDevRead	

RZKDEVATTACH

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
DDF_STATUS_t RZKDevAttach(RZK_DEVICE_CB_t *pdev)
```

Description

The `RZKDevAttach()` API attaches a device to the DDF by initializing the device driver table with the information provided by the `RZK_DEVICE_CB_t *pdev` structure pointer. Any device used with the RZK DDF must be attached to the DDF before any other operations are performed using that device. Any device that is not attached to the DDF, is not recognized by the system. The `RZKDevAttach()` function also invokes the device-specific initialization function after initializing the device driver table.

Argument(s)

`pdev`	Pointer to a structure of `RZK_DEVICE_CB_t` type. The structure must be initialized appropriate to the device.

Return Value(s)

Upon successful execution of the API, an `RZKERR_SUCCESS` value is returned. Negative values are returned for the following error conditions:

`DDFERR_INVALID_ARGUMENTS`	Specifies that the API execution is not successful because the arguments are not valid.

DDFERR_DCB_UNAVAILABLE	When no device control blocks are free.

Other error values are returned by the driver routine in case of any other error conditions.

Example

This example adds a UART device block to the system. The example assumes that necessary functions are declared externally.

```
RZK_DEVICE_CB_t Serial0Dev =
{
  RZK_FALSE, "SERIAL0",
  UARTInit, (FNPTR_RZKDEV_STOP)IOERR, UARTOpen,
UARTClose, UARTRead, UARTWrite,
(FNPTR_RZKDEV_SEEK)IOERR, UARTGetc, UARTPutc,
(FNPTR_RZKDEV_IOCTL)UARTControl,
  (RZK_PTR_t)uart0isr, 0, (UINT8*)&Uart0_Blk,\
   0,0
}
DDF_STATUS_t status ;
status = RZKDevAttach( &Serial0Dev ) ;
```

See Also

RZKDevRead	RZKDevWrite
RZKDevClose	RZKDevIOCTL
RZKDevGetc	RZKDevPutc
RZKDevOpen	RZKDevDetach

RZKDEVDETACH

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
DDF_STATUS_t RZKDevDetach(RZK_DEVICE_CB_t *pdev
```

Description

The `RZKDevDetach()` API detaches the device entry from the device driver table and releases the control block.

Argument(s)

`pdev`	Pointer to the device control block located in the device driver table. The pointer returned by `RZKDevOpen` must be passed here.

Return Value(s)

Upon successful execution of the API, an `RZKERR_SUCCESS` value is returned. The following error value is returned if an error occurs:

`DDFERR_INVALID_ARGUMENTS`	Specifies that the API execution is not successful as the arguments are not valid.

See Also

RZKDevAttach
RZKDevClose
RZKDevGetc
RZKDevOpen
RZKDevWrite
RZKDevIOCTL
RZKDevPutc
RZKDevRead

RZKDEVOPEN

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
RZK_DEVICE_CB_t * RZKDevOpen(
  RZK_DEV_NAME_t *devName,
  RZK_DEV_MODE_t *devMode)
```

Description

The `RZKDevOpen()` API opens an I/O device and returns a `HANDLE` to the opened device if successful.

Argument(s)

`*devName`	Specifies the name of the device (ASCII characters) as initialized in the `usrDevBlk` structure.
`*devMode`	Specifies the mode in which the device needs to be opened.

Return Value(s)

If the device is opened successfully, a `HANDLE` is returned to the opened device. The `HANDLE` is of the type `RZK_DEVICE_CB_t *` which is a pointer to the device control block of the device present in the device driver table. Returns NULL if the device to be opened is invalid.

See Also

RZKDevRead RZKDevWrite
RZKDevClose RZKDevIOCTL

RZKDevRead RZKDevWrite
RZKDevGetc RZKDevPutc
RZKDevAttach RZKDevDetach

RZKDEVCLOSE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
RZK_STATUS_t RZKDevClose (
RZK_DEVICE_CB_t *pdev        // device handle
);
```

Description

The `RZKDevClose()` API closes the device specified by the handle and calls a device-specific close handler as registered in the `usrDevBlk` structure.

Argument(s)

`pdev`	The handle of the device to be closed.

Return Value(s)

Upon successful execution of the API, an `RZKERR_SUCCESS` value is returned. Negative values are returned for the following error conditions:

`DDFERR_INVALID_ARGUMENTS`	Specifies that the API execution is not successful.
`DDFERR_INVALID_INITIALIZER`	When the function pointer for `close` is initialized to NULL in the Device Driver Table.

Other error values are returned by the driver routine in case of any other error conditions.

See Also

RZKDevRead
RZKDevWrite
RZKDevOpen
RZKDevIOCTL
RZKDevGetc
RZKDevPutc
RZKDevAttach
RZKDevDetach

RZKDEVREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
DDF_STATUS_t RZKDevRead(
  RZK_DEVICE_CB_t *pdev,
  RZK_DEV_BUFFER_t *buf,
  RZK_DEV_BYTES_t nBytes)
```

Description

The `RZKDevRead()` API reads a specified number of bytes from the device into a given buffer.

Argument(s)

`pdev`	Specifies the device handle to be read from.
`buf`	Specifies the pointer to the memory location where the data must be read into.
`nBytes`	The number of bytes to read from the device.

Return Value(s)

This API returns the value that is returned by the actual device read function (for example, `UARTRead`, `SPIRead`, `I2CRead` etc).

See Also

RZKDevRead RZKDevWrite
RZKDevClose RZKDevIOCTL

RZKDevGetc RZKDevPutc
RZKDevAttach RZKDevDetach

RZKDEVWRITE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
DDF_STATUS_t RZKDevWrite(
  RZK_DEVICE_CB_t *pdev,
  RZK_DEV_BUFFER_t *buf,
  RZK_DEV_BYTES_t nBytes
)
```

Description

The `RZKDevWrite()` API writes a specified number of bytes from the device into a given buffer.

Argument(s)

`pdev`	Specifies the device handle to write to.
`buf`	Specifies the pointer to the memory location where the data needs to be written into.
`nBytes`	No of bytes to write to the device.

Return Value(s)

This API returns the value that is returned by the actual device write function (for example, `UARTWrite`, `SPIWrite`, `I2CWrite` etc).

See Also

RZKDevOpen RZKDevRead
RZKDevClose RZKDevIOCTL

RZKDevGetc RZKDevPutc
RZKDevAttach RZKDevDetach

RZKDEVIOCTL

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
DDF_STATUS_t RZKDevIOCTL(
  RZK_DEVICE_CB_t *pdev,
  RZK_DEV_BYTES_t uOperation,
  void *addr1,
  void *addr2
)
```

Description

The `RZKDevIOCTL()` API is used to set or get the I/O control parameters from the device. Because the I/O control parameters are entirely device-specific, the framework calls the device-specific I/O control function as specified in the usrDevBlk structure.

Argument(s)

`pdev`	Specifies the device handle for which I/O control parameters must be set.
`optype`	Specifies the type of control operation that must be performed on the device.
`*addr1`	The first character pointer through which a device-specific parameter can be passed to the driver routine.
`*addr2`	The second character pointer through which a device-specific parameter can be passed to the driver routine.

Return Value(s)

Upon successful execution of the API, an `RZKERR_SUCCESS` value is returned. Negative values are returned for the following error conditions:

`DDFERR_INVALID_ARGUMENTS`	Indicates that the API execution is not successful.
`DDFERR_INVALID_INITIALIZER`	When the function pointer for `Ioctl` is initialized to NULL in the Device Driver Table.

Other error values are returned by the driver routine in case of any other error conditions.

See Also

RZKDevOpen	RZKDevRead
RZKDevClose	RZKDevGetc
RZKDevWrite	RZKDevPutc
RZKDevAttach	RZKDevDetach

RZKDEVGETC

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
DDF_STATUS_t RZKDevGetc(
  RZK_DEVICE_CB_t *pdev);
```

Description

The `RZKDevGetc()` API reads one byte from the device specified by the device `HANDLE`. This API returns the read byte.

Argument(s)

`pdev`	Specifies the device handle to be read from.

Return Value(s)

Upon successful execution of the API, the byte read from the device is returned. Negative values are returned for the following error conditions:

`DDFERR_INVALID_ARGUMENTS`	Indicates that the API execution is not successful.
`DDFERR_INVALID_INITIALIZER`	When the function pointer for `getc` is initialized to NULL in the Device Driver Table.

Other error values are returned by the driver routine in case of any other error conditions.

See Also

RZKDevOpen
RZKDevClose
RZKDevGetc
RZKDevAttach
RZKDevWrite
RZKDevIOCTL
RZKDevRead
RZKDevDetach

RZKDEVPUTC

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZDevice.h"
```

Prototype

```
DDF_STATUS_t RZKDevPutc(
  RZK_DEVICE_CB_t *pdev,
  RZK_DEV_BUFFER_t buf
)
```

Description

The `RZKDevPutc()` API writes one byte to a device from the a buffer.

Argument(s)

`pdev`	Specifies the device handle to write to.
`buf`	Specifies the pointer to the memory location where the data needs to be written into.

Return Value(s)

Upon successful execution of the API, an `RZKERR_SUCCESS` value is returned. Negative values are returned for the following error conditions.

`RZKERR_SUCCESS`	Indicates that the API completed the operation successfully.
`DDFERR_INVALID_ARGUMENTS`	Indicates that the API execution is not successful.

`DDFERR_INVALID_INITIALIZER`	When the function pointer for `putc` is initialized to NULL in the Device Driver Table.

Other error values are returned by the driver routine in case of any other error conditions.

See Also

RZKDevOpen	RZKDevWrite
RZKDevClose	RZKDevIOCTL
RZKDevGetc	RZKDevRead
RZKDevAttach	RZKDevDetach

Ethernet Media Access Control APIs

Table 37 provides a quick reference for Ethernet Media Access Control (EMAC) RZK APIs. The following sections provide description for each of these APIs.

Table 37. EMAC API Quick Reference

AddEmac	EmacWrite
EmacOpen	EmacRead
EmacClose	EmacControl

ADDEMAC

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "ZEmacMgr.h"
```

Prototype

```
DDF_STATUS_t AddEmac( void ) ;
```

Description

The `AddEmac()` API adds the default EMAC device control block to the RZK device control table and initializes the EMAC device, depending on the platform.

Argument(s)

None.

Return Value(s)

`RZKERR_SUCCESS`	The EMAC device block is added to the RZK device table.

Errors values returned by the RZKDevAttach API are returned in the event of any error conditions.

See Also

EmacOpen EmacClose
EmacWrite EmacRead

EMACOPEN

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
```

Prototype

```
DDF_STATUS_t EmacOpen (RZK_DEVICE_CB_t
*pDev,RZK_DEV_NAME_t *devName, RZK_DEV_MODE_t * mode);
```

Description

The `EmacOpen()` API opens the EMAC device for communication and initializes the EMAC controller and the PHY device. This API also creates the kernel resources required for communication and must be called prior to any Transmit/Receive operation involving the EMAC device.

Argument(s)

`pDev`	EMAC device handle
`devName`	Device name
`mode`	Not Applicable

Return Value(s)

One of the following error values is returned:

`EMACDEV_ERR_SUCCESS`	EMAC device successfully opened.
`EMACDEV_ERR_RESOURCE_NOT_CREATED`	If the open call fails to create an RZK resource (Thread/Message Queue).

See Also

EmacClose EmacWrite
EmacRead EmacControl

EMACCLOSE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
```

Prototype

```
DDF_STATUS_t EmacClose (RZK_DEVICE_CB_t *pDev);
```

Description

The `EmacClose()` API closes the EMAC device. Receive/transmit operations are not possible after the device is closed. This API frees up any resources created during the opening of the device.

Argument(s)

`pDev`	EMAC device handle.

Return Value(s)

The following error value is returned:

`EMACDEV_ERR_SUCCESS`	EMAC device successfully closed.

See Also

EmacOpen EmacWrite
EmacRead EmacControl

EMACWRITE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
```

Prototype

```
DDF_STATUS_t EmacWrite (RZK_DEVICE_CB_t *pDev,
ETH_PKT_t *pep, RZK_DEV_BYTES_t len);
```

Description

The `EmacWrite()` API transmits a packet to a network. The packet to be transmitted must be in the `ETH_PKT_t` structure format. See Appendix A. RZK Data Structures on page 332 for a listing of the packet structure.

`EmacWrite` is a nonblocking call. If the EMAC device is busy, the packet waits in an internal queue and is transmitted as soon as the EMAC transmitter is free.

Argument(s)

`pDev`	EMAC device handle.
`*pep`	Pointer to the packet to be transmitted (`ETH_PKT_t` structure pointer).
`len`	Total length of `ETH_PKT_t` structure.

Return Value(s)

One of the following error values is returned:

`EMACDEV_ERR_SUCCESS`	Packet successfully transmitted.

EMACDEV_ERR_INVALID_OPERATION	If this API is called without opening the device
EMACDEV_ERR_INVALID_ARGS	If the len specified is greater than ETHPKT_MAXLEN (that is, Total size of ETH_PKT_t structure).
EMACDEV_ERR_TX_WAITING	EMAC device is busy and the packet is queued up

See Also

EmacOpen EmacClose
EmacRead EmacControl

EMACREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
```

Prototype

```
DDF_STATUS_t EmacRead (RZK_DEVICE_CB_t *pDev,
ETH_PKT_t **pep, RZK_DEV_BYTES_t len);
```

Description

The `EmacRead()` API reads a single packet from the EMAC device. Memory for the packet is allocated by the API. It is a blocking call and the calling thread is suspended until the packet arrives. A packet pointer is returned through `pDev`. After the packet is processed, the memory allocated for the packet must be freed by calling `FreePktBuff()` function.

Argument(s)

`pDev`	EMAC device handle.
`*pep`	Pointer to the packet to be transmitted (`ETH_PKT_t` structure pointer).
`len`	Not applicable.

Return Value(s)

If the call is successful, this API returns the number of bytes read. Upon an error, it returns one of the following error values.

EMACDEV_ERR_INVALID_OPERATION	If this API is called without opening the device.
EMACDEV_ERR_KERNEL_ERROR	There was an error in receiving the packet.

See Also

EmacOpen EmacClose
EmacWrite EmacControl

EMACCONTROL

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
```

Prototype

```
DDF_STATUS_t EmacControl(RZK_DEVICE_CB_t *pDev,
RZK_DEV_BYTES_t func, CHAR *arg1, CHAR *arg2);
```

Description

The `EmacControl()` API allows you to perform some control operations on the EMAC device.

Argument(s)

`pDev`	EMAC device handle.	
`func`	A type of control function; the following values can be passed through this parameter.	
	`EPC_MADD`	Add a MAC address to the multicast group.
	`EPC_MDEL`	Delete a MAC address from the multicast group.
	`EPV_IRQ_ENABLE`	Enable EMAC interrupts.
	`EPV_IRQ_DISABLE`	Disable EMAC interrupts.
	`EPV_RESET`	Reset EMAC device.
`*arg1`	Pointer to the value to be passed for a particular control function.	

For the control functions listed above, the following values are expected:

`EPC_MADD`	arg1 ⟶ pointer to the Ethernet multicast address to be added to the group.
`EPC_MDEL`	arg1 ⟶ pointer to the Ethernet multicast address to be deleted from the group.
`EPV_IRQ_ENABLE`	No arguments required.
`EPV_IRQ_DISABLE`	No arguments required.
`EPV_RESET`	No arguments required.

`*arg2` Pointer to the value to be passed to a control function. For the current set of functions, a NULL value must be passed.

Return Value(s)

`EMACDEV_ERR_SUCCESS`	Packet successfully transmitted.
`EMACDEV_ERR_INVALID_OPERATION`	If this API is called without opening the device.

See Also

EmacOpen EmacClose
EmacWrite EmacRead

Wireless Local Area Network APIs

Table 38 provides a quick reference for Wireless Local Area Network (WLAN) RZK APIs. The following sections provide description for each of these APIs.

Table 38. Wireless Local Area Network APIs Quick Reference

AddWlan	wlanRead
wlanOpen	wlanClose
wlanWrite	

ADDWLAN

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "ZWlan.h"
```

Prototype

```
DDF_STATUS_t AddWlan( void ) ;
```

Description

The `AddWlan()` API adds the default WLAN device control block to the RZK device control and initializes the device. The WLAN initialization includes SDIO initialization, reading the configuration from EEPROM and downloading FIRMWARE to the WLAN chipset.

Argument(s)

None

Return Value(s)

`RZKERR_SUCCESS`	The WLAN device block is added to the RZK device table.

Error values returned by `RZKDevAttach` API are returned in case of any error conditions.

See Also

wlanOpen wlanWrite
wlanClose wlanRead

WLANOPEN

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
#include "ZWlan.h"
```

Prototype

```
DDF_STATUS_t wlanOpen(RZK_DEVICE_CB_t *pDev,
RZK_DEV_NAME_t *devName, RZK_DEV_MODE_t * mode);
```

Description

The `wlanOpen()` API opens the WLAN device for communication and does the site survey. This API also creates the kernel resources required for communication and must be called prior to any Transmit/Receive operation involving the WLAN device.

Argument(s)

`pDev`	WLAN device handle
`devName`	Device name
`mode`	Not Applicable

Return Value(s)

One of the following values is returned.

`WLANDEV_ERR_SUCCESS`	WLAN device successfully opened.
`WLANDEV_ERR_AP_NOT_FOUND`	Unable to connect to the specified access point (AP).

See Also

wlanOpen wlanWrite
wlanClose wlanRead

WLANWRITE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
#include "ZWlan.h"
```

Prototype

```
DDF_STATUS_t wlanWrite(RZK_DEVICE_CB_t *pDev,
ETH_PKT_t *pep, RZK_DEV_BYTES_t len);
```

Description

The `wlanWrite()` API transmits a packet to a wireless network. The packet to be transmitted must be in the `ETH_PKT_t` structure format. See Appendix A. RZK Data Structures on page 332 for a listing of the packet structure. wlanWrite is a nonblocking call. The packet is written in to the WLAN chipset FIFO and thereafter the WLAN chipset firmware takes care of the transmission. If FIFO is full, then the packet is dropped without returning any error to the caller.

Argument(s)

`pDev`	WLAN device handle
`*pep`	Pointer to the packet to be transmitted (ETH_PKT_t structure pointer)
`len`	Total length of ETH_PKT_t structure

Return Value(s)

One of the following error values is returned.

`WLANDEV_ERR_SUCCESS`	Packet successfully transmitted.
`WLANDEV_ERR_INVALID_OPERATION`	If this API is called without opening the device.
`WLANDEV_ERR_INVALID_ARGS`	If the len specified is greater than `ETHPKT_MAXLEN` (that is, total size of `ETH_PKT_t` structure).

See Also

wlanOpen	wlanWrite
wlanClose	wlanRead

WLANREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
#include "ZWlan.h"
```

Prototype

```
DDF_STATUS_t wlanRead(RZK_DEVICE_CB_t *pDev, ETH_PKT_t
**pep, RZK_DEV_BYTES_t len);
```

Description

The `wlanRead()` API reads a single packet from the WLAN device. It is a blocking call and the calling thread is suspended until the packet arrives or times out. A packet pointer is returned through `pDev`. After the packet is processed, the memory allocated for the packet must be freed by calling the `FreePktBuff()` function.

Argument(s)

`pDev`	WLAN device handle.
`*pep`	Pointer to the packet to be transmitted (ETH_PKT_t structure pointer).
`len`	Not applicable.

Return Value(s)

If the call is successful this API returns the number of bytes read. Upon an error, it returns one of the following error values.

`WLAN_ERR_INVALID_OPERATION`	If this API is called without opening the device.
`WLANDEV_ERR_KERNEL_ERROR`	There is an error in receiving the packet.

See Also

wlanOpen	wlanWrite
wlanClose	wlanRead

WLANCLOSE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
#include "ZWlan.h"
```

Prototype

```
DDF_STATUS_t wlanClose(RZK_DEVICE_CB_t *pdev)
```

Description

The `wlanClose()` API closes the WLAN device. Receive/Transmit operations are not possible after the device is closed. This API frees up any resources created during the opening of the device.

Argument(s)

`pDev`	WLAN device handle.

Return Value(s)

The following error value is returned.

`WLANDEV_ERR_SUCCESS`	WLAN device successfully closed.

See Also

wlanOpen	wlanWrite
wlanClose	wlanRead

Universal Asynchronous Receiver/Transmitter APIs

Table 39 provides a quick reference for Universal Asynchronous Receiver/Transmitter (UART) RZK APIs. The following sections provide description for each of these APIs.

Table 39. Universal Asynchronous Receiver/Transmitter API Quick Reference

AddUart0	UARTRead
AddUart1	UARTControl
UARTOpen	UARTPeek
UARTClose	UARTGetc
UARTWrite	UARTPutc

ADDUART0

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t AddUart0( void ) ;
```

Description

The `AddUart0()` API adds the default UART0 device control block to the RZK device control table and initializes the UART0 device. This API must be called before calling the UART0 device through DDF.

Argument(s)

None.

Return Value(s)

UARTDEV_ERR_SUCCESS	The UART0 device block is added to the RZK device table.

Errors values returned by the RZKDevAttach API are returned in the event of any error conditions.

See Also

UARTOpen UARTClose
UARTWrite UARTRead
UARTPeek UARTGetc
RTCControl

ADDUART1

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t AddUart1( void ) ;
```

Description

The `AddUart1()` API adds the default UART1 device control block to the RZK device control table and initializes the UART1 device. This API must be called before calling the UART1 device through DDF.

Argument(s)

None.

Return Value(s)

`UARTDEV_ERR_SUCCESS`	The UART1 device block is added to the RZK device table.

Errors values returned by the RZKDevAttach API are returned in the event of any error conditions.

See Also

UARTOpen UARTClose
UARTWrite UARTRead
UARTPeek UARTGetc
RTCControl

UARTOPEN

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t UARTOpen
(
  RZK_DEVICE_CB_t* pDev,
  RZK_DEV_NAME_t* pName,
  RZK_DEV_MODE_t* pMode
)
```

Description

The `UARTOpen()` API opens the UART device for communication. This API sets the baud rate and initializes the control register with values provided in the `serparams` structure located in the `UART_Conf.c` file. In addition, this API also creates the required kernel resources and installs the interrupt vector.

Argument(s)

`pDev`	UART device handle. When called through the `RZKDevOpen()` DDF API, the handle is generated within the `RZKDevOpen` API and passed to the UARTOpen function call.
`pName`	Name of the device.
`pMode`	Ignore.

Return Value(s)

One of the following values is returned:

`UARTDEV_ERR_SUCCESS`	Device successfully opened.
`UARTDEV_ERR_KERNEL_ERROR`	Kernel resources not created.
`UARTDEV_ERR_INVALID_OPERATION`	The device is not initialized.

See Also

UARTClose
UARTRead
UARTPeek
UARTPutc
UARTWrite
UARTControl
UARTGetc

UARTCLOSE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t
UARTClose
(
  RZK_DEVICE_CB_t* pDev
)
```

Description

The `UART_Close()` API closes the UART device for communication. No further Transmit/Receive operation can be performed after closing the UART device. All of the resources created during the `UART_Open()` function call are released.

Argument(s)

`pDev`	UART device handle.

Return Value(s)

One of the following values is returned:

`UARTDEV_ERR_SUCCESS`	Device successfully closed.
`UARTDEV_ERR_INVALID_OPERATION`	Device is not open.

See Also

UARTOpen
UARTRead
UARTPeek
UARTPutc
UARTWrite
UARTControl
UARTGetc

UARTWRITE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t
UARTWrite
(
  RZK_DEVICE_CB_t * pDev,
  CHAR * pBuf,
  RZK_DEV_BYTES_t Len
)
```

Description

The `UARTWrite()` API writes `Len` number of bytes into the buffer pointed to by the `pBuf` pointer. For the UART device, if the `SERSET_SYNC` bit in the `serparams` structure is set, the data is transmitted in the poll mode. If this bit is not set, then the data transmission is interrupt driven and the thread is blocked for some time if the `Transmit Hold` register is not empty. If the `SERSET_RTSCTS` bit is set, the calling thread is blocked until the CTS line is asserted. The thread calling this API cannot be deleted until this call is completed.

If a write operation is already in progress and another thread tries to write to the same device, the second thread is blocked until the first read operation is complete.

Argument(s)

`pDev`	UART device handle.
`*pBuf`	Transmit buffer pointer.
`Len`	Number of bytes to be transmitted.

Return Value(s)

If the `UARTWrite()` API is executed successfully, it returns the number of bytes transmitted. If an error occurs, the following error value is returned:

UARTDEV_ERR_INVALID_OPERATIO N	Device is not open.

See Also

UARTOpen	UARTClose
UARTRead	UARTControl
UARTPeek	UARTGetc
UARTPutc	

UARTREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t
UARTRead
(
  RZK_DEVICE_CB_t * pDev,
  CHAR * pBuf,
  RZK_DEV_BYTES_t Len
)
```

Description

The `UARTRead()` API reads the number of bytes specified by the `Len` argument into the buffer pointed to by the `pbuf` argument. If the specified number of bytes are not yet read, then this API blocks the calling thread. If a read operation is in progress and another thread tries to read from the same device, the second thread is blocked until the first read operation is complete. The thread calling this API cannot be deleted until this call is completed.

Argument(s)

`pDev`	UART device handle.
`*pbuf`	Receive buffer pointer.
`Len`	Number of bytes to be received.

Return Value(s)

If the `UARTRead()` API is executed successfully, it returns the number of bytes received. If an error occurs, the following error value is returned:

`UARTDEV_ERR_INVALID_OPERATION`	Device is not open.

See Also

UARTOpen
UARTClose
UARTWrite
UARTControl
UARTPeek
UARTGetc
UARTPutc

UARTCONTROL

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t UARTControl(
  RZK_DEVICE_CB_t *pDCB,
  RZK_DEV_BYTES_t uOperation,
  void *addr1,
  void *addr2
)
```

Description

The `UARTControl()` API allows you to set the UART I/O control parameters during runtime. The control parameters that can be set through the `UARTControl()` API are baud rate, data bits, stop bits and parity. To set any of these control parameters, set the appropriate values in the `SERIAL_PARAMS` structure that is defined in the `Uart_Conf.c` file. To realize these changes, invoke the `UARTControl` API with the correct operation. This API allows you to set either the individual control parameters such as baud rate or all of the parameters in one function call. If a read/write operation is currently in progress and involves the same device, then the `UARTControl` API blocks the calling thread. The thread calling this API cannot be deleted until this call is completed.

Argument(s)

`pDCB`	UART device handle.

`uOperation`	Type of control function. The following values can be passed through this parameter:	
	`SET_BAUD`	Set the baud rate.
	`SET_DATABIT`	Set data bits.
	`SET_PARITY`	Set parity.
	`SET_STOPBITS`	Set stop bits.
	`SET_ALL`	Set all of the above parameters.
*addr1	Ignore.	
*addr2	Ignore.	
	`SET_READ_DELAY`	Set the read delay.
*addr1	Timer value in ticks for `SET_READ_DELAY` operation. Ignore for remaining operations.	
*addr2	Ignore.	

Return Value(s)

One of the following values is returned:

`UARTDEV_ERR_SUCCESS`	If the `UARTControl()` API is executed successfully.
`UARTDEV_ERR_INVALID_ARGS`	If incorrect arguments are passed to the API.

See Also

UARTOpen UARTClose
UARTWrite UARTRead

UARTPeek UARTGetc
UARTPutc

UARTPEEK

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t
UARTPeek
(
  RZK_DEVICE_CB_t* pDev
)
```

Description

The `UARTPeek()` API returns the number of bytes available in the receive buffer. This API provides information about the number of bytes that are received after the `UARTReceive` API is invoked. Therefore, the probability of a `UARTRead()` call being blocked due to the nonavailability of data is eliminated.

Argument(s)

`pDev`	UART device handle.

Return Value(s)

If the device is open, the UARTPeek() API returns the number of bytes available in the receive buffer. If the device is not open, then this API returns the `UARTDEV_ERR_INVALID_OPERATION` error.

See Also

UARTOpen UARTClose
UARTWrite UARTRead
UARTControl UARTGetc
UARTPutc

UARTGETC

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t
UARTGetc
(
  RZK_DEVICE_CB_t*  pDev
)
```

Description

The `UARTGetc()` API reads one byte from the UART device and returns this byte to the function caller. This operation is equivalent to calling the `UARTRead()` API with the `Len` argument equal to one byte.

Argument(s)

`pDev`	UART device handle.

Return Value(s)

If the`UARTGetc()` API is executed successfully, then this API returns `the byte read from the UART device`. If unsuccessful, this API returns the following error:

`UARTDEV_ERR_INVALID_OPERATION`	Device is not open.

See Also

UARTOpen
UARTClose
UARTWrite
UARTRead
UARTPeek
UARTControl
UARTPutc

UARTPUTC

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "serial.h"
```

Prototype

```
DDF_STATUS_t
UARTPutc
(
  RZK_DEVICE_CB_t * pDev,
  CHAR Data
)
```

Description

The `UARTPutc()` API writes one byte to the UART device. This operation is equivalent to calling the `UARTWrite()` API with the `Len` argument equal to one byte.

Argument(s)

`pDev`	UART device handle.
`Data`	One byte data to be transmitted

Return Value(s)

If the `UARTGetc()` API is executed successfully, then this API returns the number of bytes that is transmitted, that is, one. If unsuccessful, this API returns the following error:

`UARTDEV_ERR_INVALID_OPERATION`	Device is not open.

See Also

UARTOpen
UARTClose
UARTWrite
UARTRead
UARTPeek
UARTGetc
RTCControl

Real-Time Clock APIs

Table 40 provides a quick reference to the real-time clock (RTC) RZK APIs that are described in this subsection.

Table 40. Real-Time Clock API Quick Reference

AddRtc
RTCRead
RTCControl

ADDRTC

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "rtc.h"
```

Prototype

```
DDF_STATUS_t AddRtc ( void ) ;
```

Description

The `AddRtc()` API adds the default RTC device control block to the RZK device control table and initializes the RTC device. This API must be called before opening the RTC device through DDF.

Argument(s)

None.

Return Value(s)

`RZKERR_SUCCESS`	The RTC device block is added to the RZK device table.

Errors values returned by the RZKDevAttach API are returned in the event of any error conditions.

See Also

RTCControl

RTCREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "rtc.h"
```

Prototype

```
DDF_STATUS_t RTCRead (RZK_DEVICE_CB_t *pdev,
RZK_DEV_BUFFER_t *buf, RZK_DEV_BYTES_t nBytes);
```

Description

The `RTCRead()` API reads the current date and time into the TIME data structure. The TIME structure must be passed to the API through the `*buf` argument. See Appendix A. RZK Data Structures on page 332 for a listing of the TIME data structure.

Argument(s)

`pDev`	RTC device handle.
`*buf`	Pointer to the TIME structure.
`nBytes`	Not applicable.

Return Value(s)

`RTC_ERR_SUCCESS`	The current date and time has been read successfully.

See Also

RTCControl

RTCCONTROL

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "rtc.h"
```

Prototype

```
DDF_STATUS_t RTCControl(RZK_DEVICE_CB_t *pdev,
RZK_DEV_BYTES_t uOperation, CHAR *addr1, CHAR *addr2);
```

Description

The `RTCControl()` API allows multiple control operations to be performed on the RTC device, including the setting of date and time. It also allows enabling and disabling of the alarm.

Argument(s)

`pDev`	RTC device handle.
`uOperationMode`	A type of control function. The following values can be passed through this parameter

`RTC_SET_SEC`	Set seconds value.
`RTC_SET_MIN`	Set minutes value.
`RTC_SET_HRS`	Set hours value.
`RTC_SET_MON`	Set month value.
`RTC_SET_DOW`	Set day of the week value.
`RTC_SET_DOM`	Set day of the month value.
`RTC_SET_YEAR`	Set year value..

RTC_SET_CENT	Set century value
RTC_SET_ALL	Set all the values mentioned above.
RTC_ENABLE_BCD	The values in the register are in Binary Coded Decimal format.
RTC_DISABLE_BCD	The values in the register are in binary format.
RTC_RESET_CONTROL	Initializes RTC control register to zero.
RTC_ENABLE_ALARM	Enables RTC alarm to the given date and time. The driver configures the RTC to generate an interrupt at the specified date and time and schedules an interrupt task. The thread body is exposed to you and the required operation can be programmed in the thread body.
RTC_DISABLE_ALARM	Disable RTC alarm.
RTC_CLK_SEL_CRYSTAL_OSCILL	The crystal oscillator output is the RTC clock source.
RTC_CLK_SEL_POWER_LINE_FREQ	The power-line frequency input is the RTC clock source.
RTC_POWER_LINE_FREQUENCY50HZ	Power-line frequency is 50 Hz
RTC_POWER_LINE_FREQUENCY60HZ	Power-line frequency is 60 Hz.

*addr1 The following values must be passed through this argument for the control functions listed above.

RTC_SET_SEC	Seconds value.
RTC_SET_MIN	Minute value.
RTC_SET_HRS	Hours value.
RTC_SET_MON	Month value.
RTC_SET_DOW	Day of the week value.
RTC_SET_DOM	Day of the month value.
RTC_SET_YEAR	Year value.
RTC_SET_CENT	Century value.
RTC_SET_ALL	Pointer to the TIME structure with all the fields initialized to appropriate values.
RTC_CLK_SEL_CRYSTAL_OSCILL	None.
RTC_CLK_SEL_POWER_LINE_FREQ	None..
RTC_POWER_LINE_FREQUENCY50HZ	None
RTC_POWER_LINE_FREQUENCY60HZ	None.

Note: The values must be passed with consideration for whether BCD is enabled or disabled.

RTC_ENABLE_BCD	None.
RTC_DISABLE_BCD	None.
RTC_RESET_CONTROL	None.

RTC_ENABLE_ALARM	Pointer to the TIME structure with the date and time field initialized to the value at which an alarm must occur.
RTC_DISABLE_ALARM	None.

Return Value(s)

RTC_ERR_SUCCESS	Operation successfully completed.
RTC_ERR_RESOURCE_NOT_CREATED	Unable to create resource for the operation specified.
RTC_ERR_RESOURCE_NOT_DELETED	Unable to delete resource.

See Also

RTCRead

Serial Peripheral Interface APIs

Table 41 provides a quick reference to a number of Serial Peripheral Interface (SPI) RZK APIs that are described in this subsection.

Table 41. Serial Peripheral Interface API Quick Reference

AddSpi	SPI_Write
SPI_Open	SPI_Read
SPI_Close	SPI_IOCTL

ADDSPI

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "spi.h"
```

Prototype

```
DDF_STATUS_t AddSpi ( void ) ;
```

Description

The `AddSpi()` API adds the default SPI device control block to the RZK device control block and initializes the SPI device. This API must be called before calling the SPI device through DDF.

Argument(s)

None.

Return Value(s)

One of the following error values is returned.

`RZKERR_SUCCESS`	The SPI device block is added to the RZK device table.

See Also

SPI_Close	SPI_Write
SPI_Read	SPI_IOCTL

SPI_OPEN

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "spi.h"
```

Prototype

```
DDF_STATUS_t SPI_Open (RZK_DEVICE_CB_t *pDCB,
RZK_DEV_NAME_t *devName, RZK_DEV_MODE_t *devMode);
```

Description

The `SPI_Open()` API opens the SPI device for communication. A device must be closed with the `SPI_Close` call before it can be opened again. `SPI_Open` installs the interrupt vector table and creates an SPI interrupt thread for transferring data. By default, this device operates in interrupt mode. The driver automatically shifts between poll and interrupt mode based upon baud rate. For low baud rates, it operates in interrupt mode; for higher baud rates, it automatically operates in poll mode.

Argument(s)

`pDCB`	SPI device handle.
`devName`	Name of the device.
`devMode`	Mode in which device must be opened.

Return Value(s)

One of the following values is returned.

`RZKERR_SUCCESS`	Indicates that the device is successfully opened.
`SPIDEV_ERR_INVALID_OPERATION`	Indicates if the device is opened again.
`SPIDEV_ERR_KERNEL_ERROR`	Indicates if interrupt thread creation is unsuccessful.

See Also

SPI_Close
SPI_Read
SPI_Write
SPI_IOCTL

SPI_CLOSE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "spi.h"
```

Prototype

```
DDF_STATUS_t SPI_Close (RZK_DEVICE_CB_t *pDCB);
```

Description

The `SPI_Close()` API closes the SPI device for communication. No further Receive/Transmit operation is possible after closing the device. All resources created during the `SPI_Open` call are freed or deleted.

Argument(s)

`pDCB`	SPI device handle.

Return Value(s)

One of the following values is returned.

`RZKERR_SUCCESS`	Device successfully opened.
`SPIDEV_ERR_INVALID_OPERATION`	If the device is not yet opened.

See Also

SPI_Open SPI_Write
SPI_Read SPI_IOCTL

SPI_WRITE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "spi.h"
```

Prototype

```
DDF_STATUS_t SPI_Write (RZK_DEVICE_CB_t *pDCB,
RZK_DEV_BUFFER_t *buf, RZK_DEV_BYTES_t size);
```

Description

The `SPI_Write()` API writes the number of bytes specified in the `size` argument to the SPI slave device. It suspends the calling thread if it is unable to transmit. It resumes transmission after the transmitter is free. This API must be placed into the global device table if the eZ80® CPU is configured as a master device.

Argument(s)

`pDCB`	SPI device handle.
`*buf`	Transmit buffer pointer.
`size`	Number of bytes to be transmitted.

Return Value(s)

If the transmission occurs successfully, the API returns the transmitted number of bytes. On error, the following error values are returned:

`SPIDEV_ERR_BUSY`	If slave device is busy.
`SPIDEV_ERR_WCOL`	If there is a write collision.
`SPIDEV_ERR_MULTIMASTER`	If there is a multimaster conflict.

See Also

SPI_Open SPI_Close
SPI_Read SPI_IOCTL

SPI_READ

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "spi.h"
```

Prototype

```
DDF_STATUS_t SPI_Read (RZK_DEVICE_CB_t *pDCB,
RZK_DEV_BUFFER_t *buf, RZK_DEV_BYTES_t size);
```

Description

The `SPI_Read()` API reads the number of bytes specified in the `size` argument from the SPI master device. It suspends the calling thread if it is unable to receive. It resumes transmission after the master sends the data. This API must be placed into the global device table if the eZ80® CPU is configured as a slave device.

Argument(s)

`pDCB`	SPI device handle.
`*buf`	Receive buffer pointer.
`size`	Number of bytes to be received.

Return Value(s)

If the transmission occurs successfully, the API returns the transmitted number of bytes. On error, the following error values are returned:

`SPIDEV_ERR_BUSY`	Indicates if slave device is busy.

SPIDEV_ERR_WCOL	Indicates if there is a write collision.
SPIDEV_ERR_MULTIMASTER	Indicates if there is a multimaster conflict.

See Also

SPI_Open
SPI_Close
SPI_Write
SPI_IOCTL

SPI_IOCTL

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "spi.h"
```

Prototype

```
DDF_STATUS_t SPI_IOCTL (RZK_DEVICE_CB_t *pDCB,
RZK_DEV_BYTES_t uOperation, CHAR *addr1, CHAR *addr2);
```

Description

The `SPI_IOCTL()` API allows you to perform control operations on the SPI device.

Argument(s)

`pDev`	SPI device handle.
`uOperation`	Type of control function. Following values can be passed through this parameter.

`SET_BAUD`	Set the SPI baud rate.
`LOAD_CONTROL`	Load the SPI control register.
`SLAVE_SELECT`	Select Slave.
`SLAVE_DESELECT`	Deselect Slave.
`READ_STATUS`	Read SPI status register.

`*arg1` Pointer to the value to be passed for a particular control function. For the control functions listed above the following values are expected:

`SET_BAUD`	`BAUD_9600`
	`BAUD_19200`
	`BAUD_38400`
	`BAUD_57600`
	`BAUD_115200`
`LOAD_CONTROL`	Value to be loaded into SPI control register.
`SLAVE_SELECT`	Any of the mentioned four characters (A/B/C/D).
`READ_STATUS`	Read value is placed here.

`*arg2` Pointer to the value to be passed to a control function. For the current set of functions, a NULL value must be passed, except in the case of SLAVE_SELECT and SLAVE_DESELECT. A value from 1 to 8 can be passed with this argument to indicate whether a slave is present on A1/B3/C5/D8. The first character is taken from `argument1`.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the control operation was performed successfully.

See Also

SPI_Open SPI_Close
SPI_Write SPI_Read

Inter-Integrated Circuit APIs

Table 42 provides a quick reference to a number of inter-integrated circuit (I^2C) RZK APIs that are described in this subsection.

Table 42. Inter-Integrated Circuit API Quick Reference

AddI2c	I2CWrite
I2COpen	I2CRead
I2CClose	I2CPeek
I2CControl	

ADDI2C

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "i2c.h"
```

Prototype

```
DDF_STATUS_t AddI2c( void ) ;
```

Description

The `AddI2c()` API adds the default I^2C device control block to the RZK device control table and initializes the I^2C device. This API must be called before opening the I^2C device through DDF.

Argument(s)

None.

Return Value(s)

`RZKERR_SUCCESS`	The I^2C device block is added to RZK.

Errors values returned by the RZKDevAttach API are returned in the event of any error conditions.

See Also

I2CClose I2CControl
I2CWrite I2CRead

I2COPEN

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "i2c.h"
```

Prototype

```
DDF_STATUS_t I2COpen(RZK_DEVICE_CB_t *pDCB, CHAR
*SlaveName, CHAR * devMode);
```

Description

The `I2COpen()` API initializes the I^2C bus and prepares it for communication. It creates the kernel resources required for driver operations. This API can be called with either `I2C_MASTER`/`I2C_SLAVE`, depending on whether the eZ80® CPU is a slave or a master.

Argument(s)

pDCB	I^2C device handle.
devName	Name string as given in the I^2C device control block.
devMode	Whether I^2C is a master or slave (`I2C_MASTER`/ `I2C_SLAVE`).

Return Value(s)

`I2CERR_SUCCESS`	Indicates that the I^2C device opened successfully.
`I2CERR_KERNELERROR`	Indicates that the open call failed to create an interrupt thread.

See Also

I2CClose I2CControl
I2CWrite I2CRead

I2CCLOSE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "i2c.h"
```

Prototype

```
DDF_STATUS_t I2CClose(RZK_DEVICE_CB_t *pDCB);
```

Description

The `I2CClose()` API closes the I^2C device. All of the kernel resources acquired by the driver are released. For any further communication with the device, the device must be reopened with the `I2COpen()` API.

Argument(s)

`pDCB`	I^2C device handle.

Return Value(s)

`I2CERR_SUCCESS`	Indicates that the I^2C device closed successfully.
`I2CERR_INVALID_OPERATION`	Indicates that the slave has not been opened.

See Also

I2COpen I2CControl
I2CWrite I2CRead

I2CCONTROL

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "i2c.h"
```

Prototype

```
DDF_STATUS_t I2CControl( RZK_DEVICE_CB_t *pDCB,
RZK_DEV_BYTES_t uOperation, CHAR *addr1, CHAR *addr2 )
```

Description

The `I2CControl()` API allows you to perform control operations on the I^2C device.

Argument(s)

`pDCB`	The I^2C device handle.

`uOperation` The operation to be performed. The following operations are supported:

`I2C_SET_SLAVE_ADDR`	Set the slave address to which all further `I2CRead`/`I2CWrite` operations are directed.
`I2C_SUBADDR_USED`	The subaddress is being used for the slave device. If this option is set in MASTER mode, the slave device subaddress are sent after sending the control byte. The length of the subaddress can be set using the `I2C_SUBADDR_LEN` option. The subaddress value must be passed through the `addr1` argument.
`I2C_SUBADDR_NOT_USED`	No subaddress are sent to the slave device after sending the control byte.

Return Value(s)

`I2CERR_SUCCESS`	Indicates that the specified operation has been successfully performed.
`I2CERR_BUSBUSY`	If the I^2C bus is in use.
`I2CERR_INVALID_OPERATION`	If the I^2C bus is not open.

See Also

I2COpen I2CClose
I2CWrite I2CRead

I2CWRITE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "i2c.h"
```

Prototype

```
DDF_STATUS_t I2CWrite(RZK_DEVICE_CB_t *pDCB, CHAR
*buf, RZK_DEV_BYTES_t size);
```

Description

Transmits a specified number of bytes of data on the I^2C bus. The nature of the write call depends on the mode in which the device is operating.

If the eZ80® CPU is operating in MASTER mode, the number of bytes specified by `size` are transmitted to the slave device. The slave device address is taken from the `currSlaveAddr` field of the `I2C_CONFIG_t` structure. The start condition, control byte and subaddress transmission are performed within this function. `I2CWrite` is a nonblocking call and returns immediately if an error is encountered.

In SLAVE mode, the `I2CWrite()` call adds the data provided in `buf` to a Tx circular queue. These data bytes are transmitted from the interrupt thread context when such a request arrives from a master. Therefore, calling `I2CWrite` when the eZ80® CPU is in SLAVE mode does not necessarily mean that the bytes have been transmitted to the slave. The circular queue does not generate any warning if the queue is full. It simply keeps overwriting the buffer if the data in the circular buffer is not being fetched and transmitted. The size of this circular buffer is configurable.

Argument(s)

`pDCB`	I^2C device handle.
`*buf`	Pointer to the buffer which contains data to be transmitted.
`size`	Number of bytes to be transmitted.

Return Value(s)

The following values are returned from the `I2CWrite()` API. Other errors are returned based on the values contained in the I^2C Status Register.

`I2CERR_SUCCESS`	Indicates that the write operation has been performed successfully.
`I2CERR_BUSBUSY`	Indicates if the I^2C bus is in use.
`I2CERR_INVALID_OPERATION`	Indicates if the I^2C bus is not open.
`I2CERR_FAILURE`	Indicates if the status is not handled by the API.
`I2CERR_TIMEOUT`	Indicates that the data byte transfer is not completed within the polling time.

See Also

I2COpen I2CClose
I2CControl I2CRead

I2CREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "i2c.h"
```

Prototype

```
DDF_STATUS_t I2CRead(RZK_DEVICE_CB_t *pDCB,
CHAR *buf, RZK_DEV_BYTES_t size) ;
```

Description

Reads a specified number of bytes of data from the slave/master on the I^2C bus. Again the nature of the call depends upon whether eZ80 is acting as a master or slave.

When the eZ80® CPU is operating in MASTER mode, a specified number of bytes are read from the slave device into the `buf` application buffer. The slave device address is taken from the `currSlaveAddr` field of the `I2C_CONFIG_t` structure. The start condition, control byte and subaddress transmission are performed within this function. `I2CRead()` is a nonblocking call and returns immediately if an error is encountered.

When the eZ80® CPU is operating in SLAVE mode, the `I2CRead` call simply reads a specified number of bytes from an Rx circular queue into the application buffer. The data received from the master is appended to the Rx circular queue from the interrupt thread context. The circular queue does not generate any warning if the queue is full. It simply keeps overwriting the buffer if the data in the circular buffer is not being read. The size of this circular buffer is configurable.

Argument(s)

`pDCB`	I^2C device handle.
`*buf`	Pointer to the receive buffer.
`size`	Number of bytes to be received.

Return Value(s)

The following values are returned from the `I2CRead()` API. Other errors are returned based on the values contained in the I^2C Status register.

`I2CERR_SUCCESS`	Indicates that the read operation is performed successfully.
`I2CERR_BUSBUSY`	Indicates if the I^2C bus is in use.
`I2CERR_INVALID_OPERATION`	Indicates if the I^2C bus is not open.
`I2CERR_FAILURE`	Indicates if the status is not handled by the API.
`I2CERR_TIMEOUT`	Indicates if the data byte transfer is not completed within the polling time.

See Also

I2COpen I2CClose
I2CControl I2CWrite

I2CPEEK

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "i2c.h"
```

Prototype

```
DDF_STATUS_t I2CPeek()
```

Description

Returns the number of bytes present in the Receive circular buffer.

Argument(s)

None.

Return Value(s)

Returns the number of bytes present in the circular buffer.

See Also

I2COpen I2CClose
I2CControl I2CWrite

Universal Serial Bus APIs

The Philips PDIUSBD12 external USB device is interfaced with the eZ80F91 MCU to function as a USB device. Table 43 provides a quick reference to the Universal Serial Bus (USB) device RZK APIs, each of which is described in this section.

Table 43. Universal Serial Bus API Quick Reference

EZ80D12_init
EZ80D12_Connect ()
EZ80D12_Disconnect ()

EZ80D12_INIT

Include

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "EZ80D12_Usb.h"
#include "EZ80F91.h"
#include "zinterrupt.h"
```

Prototype

```
INT16 EZ80D12_init (void)
```

Description

The `EZ80D12_init ()` API creates the interrupt thread and initializes the PDIUSBD12 USB device.

Argument(s)

None.

Return Value(s)

`1`	If the USB device is initialized successfully.
`EZ80D12DEV_ERR_THRD_NOT_CREATED`	If there is an error in creating the interrupt thread.

See Also

EZ80D12_Disconnect ()

EZ80D12_Connect ()

EZ80D12_CONNECT ()

Include

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "EZ80D12_Usb.h"
#include "EZ80F91.h"
#include "zinterrupt.h"
```

Prototype

```
void EZ80D12_Connect (void)
```

Description

The `EZ80D12_Connect ()` API initializes the PDIUSBD12 USB device.

Argument(s)

None.

Return Value(s)

None.

See Also

EZ80D12_Disconnect ()

EZ80D12_DISCONNECT ()

Include

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "EZ80D12_Usb.h"
#include "EZ80F91.h"
#include "zinterrupt.h"
```

Prototype

```
void EZ80D12_Disconnect (void)
```

Description

The `EZ80D12_Disconnect ()` API disconnects the PDIUSBD12 USB device from host and resets initialization done in the `EZ80D12_Connect ()` API.

Argument(s)

None.

Return Value(s)

None.

See Also

EZ80D12_Connect ()

Watchdog Timer APIs

Table 44 provides a quick reference to the two Watchdog Timer RZK APIs that are described in this subsection.

Table 44. Watchdog Timer API Quick Reference

wdt_init
wdt_reset

WDT_INIT

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "wdt.h"
```

Prototype

```
void wdt_init( UCHAR bWdtCtrl);
```

Description

The `wdt_init()` function can be used to control the watchdog timer peripheral of the eZ80® device. This function configures the watchdog timer according to the value passed by you. The time-out clock cycle period is user-configurable. For information about watchdog timer functionality, refer to the relevant product specification for the specific device belonging to the eZ80Acclaim!® family of microcontrollers.

The operations performed when the watchdog timer times out are controlled by the value of the `bWdtCtrl` parameter that is passed onto the `wdt_init()` function.

Argument(s)

bWdtCtrl	Contains the value to be written into WDT control register. For more details about #defined values, refer to the `WDT.h` file.

Return Value(s)

None.

Example

```
void SetupRoutine(void)
{
  Note:
  /*
  * Direct ZTP to manipulate the WDT.
  * Force reset after specified cycles.
  */
  wdt_init (0xC0);
}
```

See Also

wdt_reset

WDT_RESET

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "wdt.h"
```

Prototype

```
void wdt_reset();
```

Description

The `wdt_reset()` function can be used to reset the WDT timer even before it times out.

Argument(s)

None

Return Value(s)

None

Example

```
void SetupRoutine(void)
{
  Note:
  /*
  * Direct ZTP to manipulate the WDT.
  * Force reset after specified cycles.
  */
  wdt_reset ();
}
```

See Also

wdt_init

Flash Device Driver APIs

The APIs for different Flash devices feature a similar prototype for similar operation. The name of these APIs can be differentiated by replacing FLASHDEV with the device name as provided in Table 45. The Flash Device Driver APIs consider that only the equally sized blocks are accessed and erased throughout the system operation.

Table 45 lists the Flash Device Driver RZK APIs that are described in this subsection.

Table 45. Flash Device Driver APIs

API	MT28F008B	AM29LV160B	AT49BV162A	Internal Flash (F91, F92, F93 only)
FLASHDEV_Init	Initializes the Flash device	Initializes the Flash device	Initializes the Flash device	Initializes the Flash device
FLASHDEV_Read	Reads the number of bytes specified regardless of the erasable block boundary	Reads the number of bytes specified regardless of the erasable block boundary	Reads the number of bytes specified regardless of the erasable block boundary	Reads the number of bytes specified regardless of the erasable block boundary
FLASHDEV_Write	Writes the number of bytes specified regardless of the erasable block boundary	Writes the number of bytes specified regardless of the erasable block boundary	Writes the number of bytes specified regardless of the erasable block boundary	Writes the number of bytes specified regardless of the erasable block boundary

Table 45. Flash Device Driver APIs (Continued)

API	MT28F008B	AM29LV160B	AT49BV162A	Internal Flash (F91, F92, F93 only)
FLASHDEV_Erase	Erases the equally sized blocks (Normally 128K bytes)	Erases the equally sized blocks (Normally 64K bytes).	Erases the equally sized blocks (Normally 64K bytes).	Erases the equally sized blocks (For F91: 32K bytes, 8 blocks, F92: 16K bytes, 8 blocks F93: 16K bytes 4 blocks)
FLASHDEV_Close	Closes the device (Does not disable)	Closes the device (Does not disable)	Closes the device (Does not disable)	Closes the device (Does not disable)

Note: Due to the inherent characteristics of Flash, access to the Flash device is sequential and occurs through a synchronization object. Therefore, wrapper APIs are provided to perform a semaphore acquire, call the driver API, and resume from a semaphore for all read/write/erase operations.

FLASHDEV_INIT

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
```

Prototype

```
INT FLASHDEV_Init( VOID *paddr, UINT32 num_bytes ) ;
```

Description

The `FLASHDEV_Init()` API initializes the Flash device and configures the device in the read mode.

Argument(s)

`paddr`	Starting address of the Flash device.
`num_bytes`	Number of bytes that the Flash device addresses.

Return Value(s)

This API returns 0 when successful and a nonzero value when unsuccessful.

See Also

FLASHDEV_Read FLASHDEV_Write
FLASHDEV_Erase FLASHDEV_Close

FLASHDEV_READ

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
```

Prototype

```
INT32 FLASHDEV_Read(VOID *paddr, VOID *pbuf, UINT
num_bytes);
```

Description

This API reads the data from Flash and stores it in the specified buffer.

Argument(s)

`paddr`	Address in the Flash device from where the num_bytes must be read.
`pbuf`	Address in RAM where data read from the Flash device must be stored.
`num_bytes`	Number of bytes to read.

Return Value(s)

This API returns the number of bytes read when successful and a value-less than or equal to 0 when unsuccessful.

See Also

FLASHDEV_Init FLASHDEV_Write
FLASHDEV_Erase FLASHDEV_Close

FLASHDEV_WRITE

Include

```
#include “ZSysgen.h”
#include “ZTypes.h”
```

Prototype

```
INT32 FLASHDEV_Write(VOID *paddr, VOID *pbuf, UINT
num_bytes) ;
```

Description

This API writes the data from the RAM to the Flash device at the specified address.

Argument(s)

`paddr`	Address in Flash device to which data must be written.
`pbuf`	Address in RAM from which the data is read and written onto the Flash device.
`num_bytes`	Number of bytes to be written.

Return Value(s)

This API returns the number of bytes read when successful and a value less than or equal to 0 when unsuccessful.

See Also

FLASHDEV_Init
FLASHDEV_Read
FLASHDEV_Erase
FLASHDEV_Close

FLASHDEV_ERASE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
```

Prototype

```
INT FLASHDEV_Erase(VOID *paddr, UINT32 num_bytes);
```

Description

The `FLASHDEV_Erase()` API erases the physical block of a device assigned with a specific address. For the logical block size of each device, see Table 45 on page 303.

Argument(s)

`paddr`	Address location of the Flash device that must be erased.
`num_bytes`	Number of bytes to be written.

Return Value(s)

This API returns 0 when successful and a nonzero value when unsuccessful.

See Also

FLASHDEV_Init	FLASHDEV_Write
FLASHDEV_Read	FLASHDEV_Close

FLASHDEV_CLOSE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
```

Prototype

```
INT FLASHDEV_Close( void );
```

Description

The `FLASHDEV_Close()` API closes the device for access (It does not disable the Flash device for accessing).

Argument(s)

None

Return Value(s)

This API returns 0 when successful and a nonzero value when unsuccessful.

See Also

FLASHDEV_Init FLASHDEV_Write
FLASHDEV_Read FLASHDEV_Erase

Miscellaneous APIs

Table 46 provides a quick reference to the miscellaneous RZK APIs that are described In this subsection.

Table 46. Miscellaneous API Quick Reference

RZKFormatError	RZKSetCwd
RZKGetCurrentThread	RZKGetHandleByIndex
RZKGetErrorNum	RZKSystemTime
RZKGetThreadStatistics	GetDataPersistence
RZKGetTimerStatistics	SetDataPersistence
RZK_Reboot	FreePktBuff
RZKGetCwd	

RZKFORMATERROR

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZKFormatError(UINT num);
```

Description

The `RZKFormatError()` API prints the error corresponding to the error number, only if the DEBUG configuration is chosen. Refer to the RZK Configuration chapter of the Zilog Real-Time Kernel User Manual (UM0075) for more details.

Argument(s)

`num`	Specifies the error number

Return Value(s)

None.

Example

An error string for a specified error number is displayed on the console.

```
extern UINT errNum;
RZKFormatError(errNum);
```

See Also

RZKGetCurrentThread RZKGetErrorNum
RZKGetThreadStatistics RZKGetTimerStatistics
RZK_Reboot

RZKGETCURRENTTHREAD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_THREADHANDLE_t RZKGetCurrentThread();
```

Description

The `RZKGetCurrentThread()` API call returns the currently-running thread handle.

Argument(s)

None.

Return Value(s)

Returns the current running thread handle.

Example

The handle of the current running thread is stored in `hCurrHandle`.

```
RZK_THREADHANDLE_t hCurrHandle;
hCurrHandle = RZKGetCurrentThread();
```

See Also

RZKFormatError
RZKGetErrorNum
RZKGetThreadStatistics
RZKGetTimerStatistics
RZK_Reboot

RZKGETERRORNUM

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKGetErrorNum();
```

Description

The `RZKGetErrorNum()` API call returns the error number for any recently-called API function of the current thread, provided that errors are not directly returned by that API. For example, the `RZKCreateThread()` API returns a handle; therefore, if an error occurs while creating the thread, the error can be obtained by calling `RZKGetErrorNum()`. The same logic applies for all *create* calls. For *noncreate* calls, if a handle is not returned, errors are logged into the thread control block of the current thread, which can be retrieved by the `RZKGetErrorNum()` API.

Argument(s)

None.

Return Value(s)

Returns the error number for the recent API call by current thread.

Example

The error number that is set in the current TCB is retrieved after creation of an object or allocation of memory using region or partition APIs.

```
RZK_STATUS_t nErrNum;
nErrNum = RZKGetErrorNum();
```

See Also

RZKFormatError
RZKGetCurrentThread
RZKGetThreadStatistics
RZKGetTimerStatistics
RZK_Reboot

RZKGETTHREADSTATISTICS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThreadStatistics.h"
```

Prototype

```
RZK_STATUS_t RZKGetThreadStatistics(
RZK_THREADHANDLE_t          hThread,
RZK_THREADSTATISTICS_t      *pThreadStatistics
```

Description

The `RZKGetThreadStatistics()` API returns the statistics of the specified thread through `RZK_THREADSTATISTICS_t` structure. Statistics such as the total time run, actual time run and number of times the thread is blocked. This structure is filled only if `RZK_STATISTIC` (with DEBUG option enabled) is defined. See Table 65 on page 354 for members of the `RZK_THREADSTATISTICS_t` structure.

Argument(s)

`hThread`	The thread handle whose statistics is to be obtained.
`pThreadStatistics`	Pointer to the `RZK_THREADSTATISTICS_t` structure where the thread statistics are stored.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the API is executed successfully.
`RZKERR_INVALID_HANDLE`	Indicates if the thread handle passed is invalid (NULL/deleted handle/ wrong handle).
`RZKERR_INVALID_ARGUMENTS`	Indicates if pointer to the statistics structure is invalid.

Example

The statistics of a thread with a handle that is stored in `hThread` are retrieved. The API execution status is stored in the `status` variable.

```
extern RZK_THREADHANDLE_t hThread;
RZK_THREADSTATISTICS_t threadStats;
RZK_STATUS_t status;
status = RZKGetThreadStatistics(hThread,
&threadStats);
```

See Also

RZKFormatError	RZKGetCurrentThread
RZKGetErrorNum	RZKGetTimerStatistics
RZK_Reboot	RZK_THREADSTATISTICS_t

RZKGETTIMERSTATISTICS

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZTimerStatistics.h"
```

Prototype

```
RZK_STATUS_t RZKGetTimerStatistics(
RZK_TIMERHANDLE_t       *hTimer,
RZK_TIMERSTATISTICS_t   *pTimerStatistics);
```

Description

The `RZKGetTimerStatistics()` API returns the statistics of the specified software timer through the `RZK_TIMERSTATISTICS_t` structure. This function works only if `RZK_STATISTIC` (with the DEBUG option enabled) is defined. See Table 66 on page 355 for members of the `RZK_TIMERSTATISTICS_t` structure.

Argument(s)

`hTimer`	The timer handle for which the statistics must be retrieved.
`pTimerStatistics`	The pointer to the `RZK_TIMERSTATISTICS_t` structure where the timer statistics is required to be stored.

Return Value(s)

`RZKERR_SUCCESS`	Indicates that the API is executed successfully.
`RZKERR_INVALID_HANDLE`	Indicates if the thread handle passed is invalid (NULL/deleted handle/ wrong handle).
`RZKERR_INVALID_ARGUMENTS`	Indicates if pointer to the statistics structure is invalid.

Example

The statistics of a previously-created software timer with a handle that is stored in `hTimer` are retrieved. The API execution status is stored in the `status` variable.

```
extern RZK_TIMERHANDLE_t htimer;
RZK_TIMERSTATISTICS_t timerStats;
RZK_STATUS_t status;
status = RZKGetTimerStatistics(htimer,
                    &timerStats);
```

See Also

RZKFormatError	RZKGetCurrentThread
RZKGetErrorNum	RZKGetThreadStatistics
RZK_Reboot	RZK_TIMERSTATISTICS_t

RZK_REBOOT

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZK_Reboot ();
```

Description

The `RZK_Reboot()` API stores the current program counter (PC) onto the stack and completely reboots the system by placing `000000` at the PC.

Argument(s)

None.

Return Value(s)

None.

See Also

RZKFormatError
RZKGetCurrentThread
RZKGetErrorNum
RZKGetThreadStatistics
RZKGetTimerStatistics

RZKGETCWD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKGetCwd
(
  RZK_THREADHANDLE_t hThread,
 CHAR *pFsDir
)
```

Description

The `RZKGetCwd()` API returns the working directory of the thread handle passed as a parameter.

Argument(s)

`hThread`	The thread handle from which a working directory must be obtained.
`pFsDir`	A pointer to the address of the working directory.

Return Value(s)

On successful execution, the API returns `RZKERR_SUCCESS`; otherwise, it returns an error. Through `pFsDir`, it returns the address of the working directory string.

See Also

RZKFormatError RZKGetCurrentThread

RZKSETCWD

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKSetCwd
(
 CHAR *pFsDir
)
```

Description

The `RZKSetCwd()` API sets the working directory of the current thread.

Argument(s)

`pFsDir`	A pointer to the working directory.

Return Value(s)

On successful execution, the API returns `RZKERR_SUCCESS`; otherwise, it returns an error. Through `pFsDir`, it sets a working directory in the handle of the current thread.

See Also

RZKFormatError	RZKGetCurrentThread
RZKGetErrorNum	RZKGetThreadStatistics
RZKGetTimerStatistics	RZK_Reboot

RZKGETHANDLEBYINDEX

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKGetHandleByIndex
(
 UINT nIndex,
 RZK_THREADHANDLE_t *hThread
)
```

Description

The `RZKGetHandleByIndex()` API gets the thread handle of the specified index.

Argument(s)

`nIndex`	The index passed.
`hThread`	The thread handle that is returned.

Return Value(s)

On successful execution, the API returns `RZKERR_SUCCESS`; otherwise, it returns an error. Through `hThread`, it returns the working directory in the thread's handle that corresponds to the index value.

See Also

RZKFormatError	RZKGetCurrentThread
RZKGetErrorNum	RZKGetThreadStatistics
RZKGetTimerStatistics	RZK_Reboot

RZKSYSTEMTIME

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
TIME_t RZKSystemTime ();
```

Description

This function returns the time elapsed from the boot time in seconds.

Argument(s)

None.

Return Value(s)

Time in terms of number of seconds elapsed after bootup.

Example

```
void SetupRoutine(void)
{
  Note:
  /*
  * Direct ZTP to manipulate the WDT.
  * Force reset after specified cycles.
  */
  RZKSystemTime ();
}
```

See Also

RZKGetCurrentThread RZKGetErrorNum
RZKGetThreadStatistics RZKGetTimerStatistics

GETDATAPERSISTENCE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "DataPerStruct.h"
```

Prototype

```
INT8 GetDataPersistence( PDATA_PER_t p_data_per ) ;
```

Description

This function returns the data present in the appropriate nonvolatile device that is used to store values that are intended to remain persistent despite reboots of the calling device.

Argument(s)

`p_data_per`	Pointer to the `DATA_PER_t` structure where the data that is read from the nonvolatile memory device must be stored.

Return Value(s)

Returns `RZKERR_SUCCESS` when data is read successfully; any other values indicate an error.

See Also

SetDataPersistence

SETDATAPERSISTENCE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "DataPerStruct.h"
```

Prototype

```
INT8 SetDataPersistence( PDATA_PER_t p_data_per ) ;
```

Description

This function writes data that is intended to remain persistent to a nonvolatile memory device, despite reboots of the calling device.

Argument(s)

`p_data_per`	Pointer to the `DATA_PER_t` structure where the data that is read from the nonvolatile memory device must be stored.

Return Value(s)

Returns `RZKERR_SUCCESS` when data is read successfully; any other values indicate an error.

See Also

GetDataPersistence

RZKSETFSDATA

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
RZK_STATUS_t RZKSetFSData(UINT32 uldata);
```

Description

The `RZKSetFSData()` API sets the file system's error in the current thread's thread control block (TCB).

Argument(s)

The `uldata` value to be written to the current thread's thread control block (TCB).

Return Value(s)

`RZKERR_SUCCESS`	API returned successfully.

See Also

RZKGetFSData

RZKGETFSDATA

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
UINT32 RZKGetFSData();
```

Description

The `RZKGetFSData()` API gets the file system's error from the current thread's thread control block (TCB).

Argument(s)

None.

Return Value(s)

`UINT32` value is returned by the current thread's TCB.

See Also

RZKSetFSData

RZKTHREADLOCKFORDELETE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZKThreadLockForDelete();
```

Description

The `RZKThreadLockForDelete()` API locks a thread from deletion.

Argument(s)

None.

Return Value(s)

None.

See Also

RZKThreadUnLockForDelete

RZKTHREADUNLOCKFORDELETE

Include

```
#include "ZSysgen.h"
#include "ZTypes.h"
#include "ZThread.h"
```

Prototype

```
void RZKThreadUnLockForDelete();
```

Description

The `RZKThreadUnLockForDelete()` unlocks a thread for deletion.

Argument(s)

None.

Return Value(s)

None.

See Also

RZKThreadLockForDelete

FREEPKTBUFF

Include

```
#include "Zsysgen.h"
#include "Ztypes.h"
#include "ZThread.h"
#include "ZDevice.h"
#include "EtherMgr.h"
```

Prototype

```
void  FreePktBuff (void *ptr);
```

Description

The `FreePktBuff()` API releases the memory for the packet, which is allocated by the `EmacRead()` API.

Argument(s)

`*ptr`	Pointer to the packet to be released.

Return Value(s)

None.

See Also

EmacOpen EmacClose
EmacWrite EmacRead

Appendix A. RZK Data Structures

This section defines the RZK data types, structures, enumerators, constants and macros and lists the RZK data types in the EMAC and RTC data structures.

RZK Data Types

Table 47 defines the RZK data types.

Table 47. RZK Data Types

Data Type	Definition
UINT32	Unsigned int 32-bit
COUNT_t	Unsigned int
TICK_t	Unsigned int
TIME_t	Unsigned long
RZK_STATE_t	Unsigned int
RZK_STATUS_t	Unsigned int
RZK_NAME_t	Unsigned char
UCHAR	Unsigned char
RZK_THREAD_PRIORITY_t	Unsigned char
RZK_OPERATIONMODE_t	Unsigned int
RZK_PREEMPTION_t	Unsigned int
RZK_EVENT_t	Unsigned int
RZK_MASK_t	Unsigned int
RZK_MESSAGE_PTR_t	Unsigned char pointer
CADDR_t	Unsigned char pointer
RZK_INTERRUPT_NUM_t	Unsigned int

Table 47. RZK Data Types (Continued)

Data Type	Definition
RZK_HANDLE_t	void pointer
RZK_THREADHANDLE_t	void pointer
RZK_MESSAGEQHANDLE_t	void pointer
RZK_SEMAPHOREHANDLE_t	void pointer
RZK_EVENTHANDLE_t	void pointer
RZK_TIMERHANDLE_t	void pointer
RZK_PARTITIONHANDLE_t	void pointer
RZK_REGIONHANDLE_t	void pointer
RZK_PTR_t	void pointer
UINTRMASK	unsigned int
RZK_LENGTH_t	unsigned int
FNP_THREAD_ENTRY	void (*ThreadCleanupFn)(void);
FNP_TIMER_FUNCTION	void (*TimerFn)(void);
RZK_FNP_ISR	void (*IsrHandler)(void);

EMAC Data Structure

The following code defines how Ethernet packets are structured.

```
/* Ethernet Packet Structure */
typedef struct ETH_PKT {/* complete structure of
 Ethernet packet*/
 QUEUE_NODE_t link;   /* pointers to link the packet
     /* to the queue */
UINT32 ethPktNextHop ; /* input() uses this */
UINT8 ifNum ;   /* Interface Number */
UINT16 PPPoEField;
UINT16 ethPktOrder ;  /* byte order mask (for
    /* debugging) */
```

```
UINT16 ethPktLen ;   /* length of the packet */
UINT8 wlanHdr[22];    /* WLAN HEADER */
ETH_HEADER_t ethPktHeader ; /* the ethernet header */
UINT8 ethPktData[ETHPKT_DLEN]; /*data in the packet */
} ETH_PKT_t;

/* ethernet header*/
typedef struct ETH_HEADER {
 ETH_ADDR ethDst;     /* destination host address*/
 ETH_ADDR ethSrc;     /* source host address */
 UINT16 ethType;      /* Ethernet packet type */
} ETH
```

UART Data Structure

The following code defines the UART data structure.

```
typedef enum { PAREVEN, PARODD, PARNONE } serparity;

typedef struct serialparam {
 UINT32 baud;// The Baud rate value. Eg 57600,
  // 9600
 UINT16 databits;// Number of data bits. Eg 7, 8
 UINT16 stopbits;// Number of stop bits. Eg 1, 2
 serparity parity;// Parity setting. One of the
  // values in serparity enumeration
 UINT16 settings;// Other Control settings, e.g.,
  // SERSET_RTSCTS, SERSET_SYNC
}SERIAL_PARAMS;
```

RTC Data Structure

The following code defines the TIME data structure.

```
typedef struct TIME{
 UCHAR sec ;/* Seconds */
```

```
 UCHAR minutes ;/* Minutes */
 UCHAR hrs ;/* Hours */
 UCHAR dayOfMonth ;/* Day of the month */
 UCHAR dayOfWeek ;/* Day of the week */
 UCHAR mon ;/* Month */
 UCHAR year ;/* Year */
 UCHAR cent ;/* Century */
}TIME ;
```

Data Persistence Data Structure

The data persistence structure provides values for predefined variables and provides an option for storing customer-defined values into nonvolatile memory, where values are required to be stored despite reboots. The `DATA_PER_t.c_userdata` structure member is used to store user data. You can store data in this structure member.

```
typedef struct {
  UINT8 c_EmacAddr[6] ;
  UINT8 c_IsDhcpEnabled ;
  UINT32 ul_IPAddr ;
  UINT32 ul_NetMask ;
  UINT32 ul_GateWay ;
  UINT8 c_userdata[ 128 ] ; // User data
} DATA_PER_t, *PDATA_PER_t ;
```

RZK Enumerators

Table 48 lists the RZK enumerators.

Table 48. RZK Enumerators

RZK_EVENT_OPERATION_et
RZK_RECV_ATTRIB_et
RZK_ERROR_et

RZK_EVENT_OPERATION_ET

This enumerator governs event operations, as shown in the code sample below.

```
typedef enum
{
  EVENT_AND,        /* event operation is ANDed with
                    /* the existing value*/
  EVENT_OR          /* event operation is ORed with
                    /* the existing value*/
} RZK_EVENT_OPERATION_et;
```

RZK_RECV_ATTRIB_ET

This enumerator governs receiving order attributes, as shown in the code sample below.

```
typedef enum
{
  RECV_ORDER_FIFO,  /* receiving attribute is FIFO */
  RECV_ORDER_PRIORITY, /* receiving attribute is
                       /* PRIORITY */
} RZK_RECV_ATTRIB_et;
```

RZK_ERROR_ET

This enumerator governs RZK errors. RZK error codes are stored in `RZK_ERROR_et` enumerators or in RZK_STATUS_t data type variables. See Table 68 on page 338 for a list of error conditions.

RZK Constants

Table 49 provides a quick reference to the RZK constants that are described in this section.

Table 49. RZK Constants

RZK_OPERATIONMODE_t
RZK_STATE_t
RZK_EVENT_OPERATION_et

RZK_OPERATIONMODE_T

This data type is used for thread operation mode variables. Table 50 lists a combination of the values stored in variables of type `RZK_OPERATIONMODE_t`.

Table 50. RZK_OPERATIONMODE_t Values

Stored Variable	Hex Values	Definition
`#define RZK_THREAD_ROUNDROBIN`	`0x01`	Round robin time slice enabled to perform round-robin scheduling.
`#define RZK_THREAD_AUTOSTART`	`0x02`	Starts thread at creation time.
`#define RZK_THREAD_PREEMPTION`	`0x04`	Current thread can be preempted.
`#define RZK_THREAD_REGISTER`	`0x10`	Runs the thread as a register thread.

RZK_STATE_T

This data type is used for object state variables. Table 51 lists a combination of values stored in variables with data type `RZK_STATE_t` for thread objects.

Table 51. RZK_STATE_t Values: Thread Objects

Stored Variable	Hex Value	Definition
`#define THREAD_CREATED`	`0x01`	Thread is created.
`#define THREAD_BUSY`	`0x02`	Thread control block is busy updating with data.
`#define THREAD_RUNNING`	`0x08`	Thread is running (ready to run).
`#define THREAD_BLOCKEDSUSPEND`	`0x10`	Thread is suspended because of blocking on a resource or timed suspend.
`#define THREAD_TIMEDBLOCK`	`0x40`	Thread is in timed block because of blocking on a resource.
`#define THREAD_INFINITESUSPEND`	`0x80`	Thread is suspended infinitely.

Table 52 lists the combination values stored in variables of type `RZK_STATE_t` for objects other than thread objects.

Table 52. RZK_STATE_t Values: Nonthread Objects

Stored Variable	Hex Value	Definition
`#define OBJECT_CREATED`	`0x01`	Object is created; common for all objects.
`#define OBJECT_BUSY`	`0x02`	Object control block is busy updating with data; common for all objects.
`#define OBJECT_RECV_PRIORITY`	`0x04`	Object's receiving method is of PRIORITY type; not for events and event groups.
`#define OBJECT_FULL`	`0x08`	Object is full.

RZK_EVENT_OPERATION_ET

Table 53 lists the macros defined for use with Event object APIs with the enumerator type `RZK_EVENT_OPERATION_et`.

Table 53. RZK_EVENT_OPERATION_et Macros

Macro	Hex Value
EVENT_AND	0x00
EVENT_OR	0x01
EVENT_XOR	0x02
EVENT_CONSUME	0x04

Additional RZK Macros

Table 54 lists the additional macros used by RZK.

Table 54. Additional RZK Macros

Stored Variable	Hex Value	Definition
`#define MAX_INFINITE_SUSPEND`	((unsigned int)(–1))	Maximum time for the Infinite Suspend.
`#define MAX_OBJECT_NAME_LEN`	8	Maximum length of an object's name.
`#define RZK_TRUE`	1	Macro definition for common TRUE.
`#define RZK_FALSE`	0	Macro definition for common FALSE.

Semaphore Macro

Table 55 lists the macro defined for semaphores.

Table 55. The RZK_PRIORITY_INHERITANCE Macro

Stored Macro	Hex Value	Definition
`#define RZK_PRIORITY_INHERITANCE`	0x80	Priority Inheritance is supported.

RZK Objects

Table 56 offers a quick reference to the RZK parameter structures that are described in this section.

Table 56. RZK Objects Quick Reference

RZK_THREADPARAMS_t
RZK_SCHEDPARAMS_t
RZK_MESSAGEQPARAMS_t
RZK_SEMAPHOREPARAMS_t
RZK_EVENTGROUPPARAMS_t
RZK_TIMERPARAMS_t
RZK_PARTITIONPARAMS_t
RZK_REGIONPARAMS_t
RZK_THREADSTATISTICS_t
RZK_TIMERSTATISTICS_t
RZK_CLOCKPARAMS_t

RZK_THREADPARAMS_T

Table 57 lists the members of the thread parameter structure RZK_THREADPARAMS_t.

Table 57. RZK_THREADPARAMS_t Structure Members

Data Type	Member Name	Description
RZK_NAME_t	szName[MAX_OBJECT_NAME_LEN]	The name of the thread; available only if DEBUG configuration is chosen. In NON_DEBUG configuration, the contents of this member have no meaning.
RZK_STATE_t	uState	Specifies the state the thread is in. See RZK_STATE_t for contents required to be stored in the RZK_STATE_t data type.
UCHAR	uBankSelector	Specifies the count of the register bank. Ignore for the eZ80® family of processors.
RZK_OPERATIONMODE_t	uOperationMode	Specifies the mode of operation, such as round-robin or priority. See RZK_OPERATIONMODE_t for contents required to be stored in the RZK_OPERATIONMODE_t data type.
TICK_t	tQuantum	Specifies the (round-robin) Time slice for the thread.
RZK_THREAD_PRIORITY_t	cPriority	Specifies the thread's priority.
RZK_EVENT_t	eEventsReceived	Specifies the events received by the thread.

RZK_SCHEDPARAMS_T

Table 58 lists the members of the scheduler parameter's structure, `RZK_SCHEDPARAMS_t`.

Table 58. RZK_SCHEDPARAMS_t Structure Members

Data Type	Member Name	Description
`TICK_t`	`tTimeSlice`	Specifies the default (round-robin) time slice for round-robin scheduler.

RZK_MESSAGEQPARAMS_T

Table 59 lists the members of the message queue parameter's structure, RZK_MESSAGEQPARAMS_t.

Table 59. RZK_MESSAGEQPARAMS_t Structure Members

Data Type	Member Name	Description
COUNT_t	uQueueLength	Length of queue
COUNT_t	uMaxMessageSize	Maximum size of each message
COUNT_t	uNumThreads	Number of threads waiting to post and receive
COUNT_t	uMessageSpaceLeft	Specifies the number of free message slots.
RZK_STATE_t	uState	Specifies the state of the message queue.
RZK_MESSAGE_PTR_t	pStart	Pointer to Start position of Message Queue Area

RZK_SEMAPHOREPARAMS_T

Table 60 lists the members of the semaphore parameter's structure, RZK_SEMAPHOREPARAMS_t.

Table 60. RZK_SEMAPHOREPARAMS_t Structure Members

Data Type	Member Name	Description
COUNT_t	uInitialCount	Specifies the initial count of the semaphore.
COUNT_t	uDynamicCount	Specifies the dynamic count of the counting semaphore.
COUNT_t	nNumThreads	Specifies the number of threads waiting on the semaphore.
RZK_STATE_t	uState	Specifies the state of the semaphore.

RZK_EVENTGROUPPARAMS_T

Table 61 lists the members of the event group parameter's structure, `RZK_EVENTGROUPPARAMS_t`.

Table 61. RZK_EVENTGROUPPARAMS_t Structure Members

Data Type	Member Name	Description
`RZK_EVENT_t`	`eEventsReceived`	Specifies the events received on the EventGroup object.

RZK_TIMERPARAMS_T

Table 62 lists the members of the timer parameter's structure, `RZK_TIMERPARAMS_t`.

Table 62. RZK_TIMERPARAMS_t Structure Members

Data Type	Member Name	Description
`TICK_t`	`tInitialDelay`	Specifies the initial delay.
`TICK_t`	`tPeriod`	Specifies the period.

RZK_PARTITIONPARAMS_T

Table 63 lists the members of the partition parameter's structure, RZK_PARTITIONPARAMS_t.

Table 63. RZK_PARTITIONPARAMS_t Structure Members

Data Type	Member Name	Description
COUNT_t	uNumOfBlocks	Specifies number of blocks in a partition.
COUNT_t	uBlockSize	Specifies size of each memory block.
COUNT_t	nBlocksUsed	Specifies number of memory blocks used.
RZK_STATE_t	uState	Specifies the state of the partition.
RZK_PTR_t	pMemory	Pointer to memory area for each block.

RZK_REGIONPARAMS_T

Table 64 lists the members of the region parameter's structure, `RZK_REGIONPARAMS_t`.

Table 64. RZK_REGIONPARAMS_t Structure Members

Data Type	Member Name	Description
`COUNT_t`	`uUnitSize`	Specifies the initial count of the region.
`UINT_t`	`uRnDelete`	Specifies the number of threads waiting on the region.
`RZK_STATE_t`	`uState`	Specifies the state of the region.

RZK_THREADSTATISTICS_T

Table 65 lists the members of the thread statistics parameter's structure, `RZK_THREADSTATISTICS_t`.

Table 65. RZK_THREADSTATISTICS_t Structure Members

Data Type	Member Name	Description
`TICK_t`	`tTotalTimeRun`	Specifies the total time the thread executed.
`TICK_t`	`tActualTimeRun`	Specifies the time the thread executed.
`COUNT_t`	`nNumTimesBlocked`	Specifies number of times the thread is blocked.

RZK_TIMERSTATISTICS_T

Table 66 lists the timer statistics parameter's structure, `RZK_TIMERSTATISTICS_t`.

Table 66. RZK_TIMERSTATISTICS_t Structure Members

Data Type	Member Name	Description
`TICK_t`	`tJitter`	Specifies the Jitter value.
`TICK_t`	`tDrift`	Specifies the Drift value.

RZK_CLOCKPARAMS_T

Table 67 lists the members of the clock parameter's structure, `RZK_CLOCKPARAMS_t`.

Table 67. RZK_CLOCKPARAMS_t Structure Members

Data Type	Member Name	Description
`UINT`	`uCurr_Year`	Year
`UINT`	`uCurr_Month`	Month
`UINT`	`uCurr_Date`	Date
`UINT`	`uCurr_Hour`	Hour
`UINT`	`uCurr_Minute`	Minute
`UINT`	`uCurr_Seconds`	Seconds

Appendix B. RZK Error Conditions

This appendix contains all RZK error constants. When an error occurs, you must pass one of these constants to the error handling function, `RZK-FormatError()`, which prints the error. The error values can be stored either in `RZK_STATUS_t` data type or in `RZK_ERROR_et` data type.

Table 68 lists the values for error conditions that can occur while using RZK.

Table 68. Values for Error Conditions

Error	Value
RZKERR_SUCCESS	0
RZKERR_INVALID_HANDLE*	1
RZKERR_INVALID_ARGUMENTS*	2
RZKERR_INVALID_OPERATION*	3
RZKERR_CB_UNAVAILABLE*	4
RZKERR_QUEUE_EMPTY	5
RZKERR_OBJECT_DELETED	6
RZKERR_TIMEOUT	7
RZKERR_INVALID_SIZE	8
RZKERR_OBJECT_IN_USE	10
RZKERR_INVALID_STACK	11
RZKERR_INVALID_PRIORITY	12
RZKERR_QUEUE_FULL	13
RZKERR_SCB_UNAVAILABLE	14
RZKERR_OUT_OF_MEMORY	15

Note: *These errors are returned only when the DebugPI or DebugNPI libraries are used.

Table 68. Values for Error Conditions (Continued)

Error	Value
RZKERR_CB_BUSY	16
RZKERR_FATAL	17
RZKERR_PREEMPTION_DISABLED	18
RZKERR_INVALID_ERROR_NUMBER	19
RESERVED	20–30
RZKERR_SEM_NOTOWNED	31
RESERVED	32–39
RZKERR_USER_ERROR	251
Note: *These errors are returned only when the DebugPI or DebugNPI libraries are used.	

Appendix C. Interrupt Handling

Each peripheral device that generates an interrupt contains an entry in the Interrupt Vector Table. RZK provides an API to install the handler for these interrupts; see the RZKInstallInterruptHandler API definition on page 194.

The `RZKInstallInterruptHandler()` API employs the following syntax:

```
RZKInstallInterruptHandler(ISR Routine, interrupt
offset in vector table)
```

To improve RZK's interrupt latency, RZK v1.2.1 and later contains an interrupt model that supersedes the model employed by RZK v1.0.0. The application developer must follow this new model so that performance improves and interrupt handling behaves as expected.

Note: RZK v1.2.1 and later versions do not work with the RZK v1.0.0 interrupt model. If you are using RZK v1.0.0 for your applications, you must change your code so that RZK operates correctly. The instructions that follow alert the reader to the newer interrupt model in greater detail. To install the interrupt handler in versions of RZK that are subsequent to v1.1.0, see the `RZKInstallInterruptHandler()` API.

RZK contains specific prologues and epilogues to manage interrupts. All interrupt service routines must use these prologues and epilogues, as stipulated in the follow instructions.

1. Create a thread for handling each of the interrupts, and pass `RZK_THREAD_PREEMPTION | RZK_THREAD_INTERRUPT` as `uOperationMode`. A thread used for interrupt handling cannot be an autostart thread.

2. Call the `RZKInstallInterruptHandler` function to install user-specific prologue code into a user-specified interrupt location.
3. As soon as the required interrupt fires, the prologue function you installed is called.
4. The format of the prologue function must adhere to the following sequence:
   a. PUSH the relevant registers.
   b. Disable the device interrupt for which this current prologue is executing. This device interrupt remains disabled until the ISR completes its task.
   c. Call the `RZKIsrProlog` routine.
   d. Call the RZKResumeInterruptThread API to resume the infinitely-suspended interrupt thread that you created previously.
   e. If the interrupt thread's priority is higher than the current thread's priority, call the `RZKIsrEpilog` function to change the context to that of an interrupt thread . Otherwise, control is returned to the prologue.
   f. POP the relevant registers and return from the prologue routine.
5. The interrupt thread's entry point function that you created must adhere to the following format:

```
Interrupt Thread entry function ()
 {
  while (1)
  {
  ISR ();         // The function that gets executed
                  // for every interrupt
  // Call RZKDisableInterrupts API of RZK.
  // Enable device-specific interrupt here. This
  // was disabled when the device-specific
  // interrupt was fired.
  // Call RZKSuspendInterruptThread () API here.
  // Details about this API are described in a
```

```
  // separate function under Thread APIs.
  Call RZKEnableInterrupts API of RZK.
  }
 }
```

6. Application program users can utilize the `RZKDisablePreemption` and `RZKRestorePreemption` APIs to minimize system latency and to protect certain blocks of code. Usage of these two APIs is preferred over using the `RZKDisableInterrupts` and `RZKEnableInterrupts` APIs due to the reduced latency they offer.
7. The stack size of RZK's timer interrupt can be configured using the `RZK_Conf.c` file using the macro `RZK_SYSTIMERSTACK_SIZEH`.
8. A RETI instruction is executed when a call to the `RZKIsrEpilog` API is made. It is not necessary to insert the RETI instruction into the ISR.
9. The `RZKIsrEpilog` API must be the final API called in an ISR, just prior to restoring the registers.
10. Ensure that interrupts remain disabled when the `RZKIsrProlog` or `RZKIsrEpilog` calls are made. Additionally, ensure that interrupts remain disabled while saving and restoring registers.
11. The stack size of the idle thread is 256 bytes by default. The idle thread stack can become corrupted when the idle thread is running, and a large number of variables are created in the ISRs of the interrupts that occur – or even when many interrupts occur together (nested interrupts). Therefore, sufficient idle thread stack space must be allocated. Idle thread stack size can be configured in RZK by changing the value of the `RZK_STACK_SIZEH` macro in the `RZK_Conf.c` file.
12. Use the APIs provided by RZK, namely `RZKEnableInterrupts()` and `RZKDisableInterrupts()`, to enable and disable interrupts in the C routines. EI and DI instructions can be used directly in assembly routines; however, ensure that every DI is provided with a matching EI.

13. Ensure that the timer register value does not exceed 16 bits.
14. Interrupts must be enabled (using EI) before calling the interrupt handler to enable nested interrupts.
15. By default, TIMER0 is used by RZK to handle system interrupts. This timer can be changed by the application developer by setting either 1 or 2 or 3 in the `HWTIMER_TO_USE` macro located in the `RZK_Conf.c` file.

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.