Quadros®
Systems Inc.

RTXC Kernel Services Reference, Volume 1

Levels, Threads, Exceptions, Pipes, Event Sources, Counters, and Alarms

Disclaimer

Quadros Systems, Inc. makes no representations or warranties with respect to the contents or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Quadros Systems, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Quadros Systems, Inc. makes no representations or warranties with respect to any Quadros software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Quadros Systems, Inc. reserves the right to make changes to any and all parts of Quadros software, at any time, without any obligation to notify any person or entity of such changes.

Trademarks

Quadros is a registered trademark of Quadros Systems, Inc. **RTXC**, **RTXC Quadros**, and **RTXC DSP** are trademarks of Quadros Systems, Inc.

Other product and company names mentioned in this document may be the trademarks or registered trademarks of their respective owners.

Quadros Systems, Inc.
10450 Stancliff, Suite 110
Houston, TX 77099-4336
USA

RTXC Kernel Services Reference, Volume 1
Part Number: RTXC-KSRV1-0602
June 2002
RTXC Kernel, Version 1.0

Contents

List of Tables

List of Examples

CHAPTER 1 Introduction To RTXC/ss Kernel Services

In This Chapter

We discuss the contents of this manual, then list the **RTXC/ss** kernel services by class and briefly describe each service.

Using This Manual

Note: The *RTXC Kernel Services Reference, Volume 1* contains information needed by users of both the Single Stack and the Dual Mode configurations of the **RTXC** Kernel. If you purchase the Single Stack configuration (**RTXC/ss** only) of the **RTXC** Kernel, you receive only Volume 1 of this book.

The *RTXC Kernel Services Reference, Volume 2* contains information needed by users of the Dual Mode configuration of the **RTXC** Kernel. If you purchase the Dual Mode configuration (both **RTXC/ss** and **RTXC/ms**), you receive both Volume 1 and Volume 2.

Kernel services are the functions performed by a real time kernel. This manual describes the complete set of kernel services available in the **RTXC** Kernel. This section describes the types of information and the organization of that information in this manual.

Kernel Service Description Format

The remaining chapters of this manual describe each kernel service in detail. The chapters separate the services into classes or subclasses, and the descriptions are in alphabetical order of the service name minus the service prefix within each class or subclass. Each description includes a complete explanation of the kernel service function, according to the topics listed in Table 1-1 on page 3.

Prototypes

The Synopsis section of each service description shows the formal ANSI C declaration and argument prototype for that service. These prototypes come from the **`rtxcapi.h`** file, which is included with each **RTXC** RTOS Software Development Kit (SDK). Because the **RTXC** Kernel is designed with portability in mind, the API defined in the **`rtxcapi.h`** file is essentially identical for all **RTXC** RTOS SDKs. However, there are differences between some of the processors on which the **RTXC** Kernel operates that lead to variations in the size of certain parameters used by the kernel services.

Similarly, there may be syntactical differences between C compilers from different manufacturers. For example, one C compiler may use the key words `near` and `far` to permit different memory models due to the processor's architecture, whereas a compiler targeted to a different processor may not require the `near` and `far` keywords.

Table 1-1. Kernel Service Description Format

Name	Brief Functional Description
Zones	The zonal prefixes supported by the service (`IS_`, `TS_`, `KS_`), if more than one.[a]
Synopsis	The formal ANSI C declaration including argument prototyping.
Inputs	A brief description of each input argument.
Description	A complete description of what the service does, the data types it uses, and so on.
Outputs	A description of each argument modified by the service and each possible return value from the service.
Example	One or more typical uses of the service. The examples assume the syntax of ANSI Standard C.[b] *SELFTASK* is used in many of the examples to denote the Current Task. It is defined in **`rtxcapi.h`** as `(TASK)0`. *SELFTHREAD* is used in many of the examples to denote the Current Thread. It is defined in **`rtxcapi.h`** as `(THREAD)0`. The *putline* function moves the content of a character buffer to an assumed console device.

Table 1-1. Kernel Service Description Format *(continued)*

Name	Brief Functional Description
See Also	A list of related kernel services, if any, that could be examined in conjunction with the current function.
Special Notes	Additional notes and technical comments, if any.

a. Services that support more than one zone are listed with an *XX_* prefix. The *XX_* prefix is not a valid prefix, only a placeholder.

b. The code examples in this manual often refer to functions or entities outside the given code fragment used in the example. The functions or entities so referenced may be real or assumed within the actual context of the code example but are not shown. The purpose of such references is to add coherence to the example rather than to imply a particular methodology or usage.

General Form of Kernel Service Call

The general form of an **RTXC** Kernel service function call is:

```
XX_name ([arg1][, arg2]...[, argn])
```

where the service prefix character string *XX_* is one of the following:

`IS_`	Identifies a service callable from an exception handler in Zone 1.
`TS_`	Identifies a thread-based service callable from Zone 2.
`KS_`	Identifies a service callable from Zone 3.

Some services are callable from all three zones, others from zones 2 and 3, and still others from Zone 2 or Zone 3 only. The detailed descriptions of the services in this book include the zones from which the service can be called if it can be called from more than one.

Following the service prefix is the name of the **RTXC** Kernel service. The service prefix should prevent the name from being misidentified by a linker with some similarly-named function in the runtime library of the compiler. In general, *name* is composed as follows:

```
<Verb><Class>[noun|property][suffix]
```

where the strings within the angle brackets (<>) are mandatory and those within the brackets ([]) are optional. The vertical bar (|) indicates an OR. Therefore, the general composition of name is a verb, followed by the object class, followed by an optional noun or object property, followed by an optional suffix.

The optional suffix is one or more upper-case characters and is used as a qualifier for the service:

`W` Indicates an unconditional wait version of the service. For example, the `KS_AllocBlkW` service is the unconditional wait version of the `KS_AllocBlk` service.

`T` Indicates a tick-limited wait version of the service. For example, the `KS_AllocBlkT` service is the tick-limited wait version of the `KS_AllocBlk` service.

`M` Indicates a service to be performed on multiple semaphore objects. For example, `KS_SignalSemaM` signals multiple semaphores.

Arguments to Kernel Services

The **RTXC** Kernel service descriptions show the function prototypes with generalized **RTXC** arguments. Similarly, the descriptions show the values returned from kernel service functions symbolically as described in Table 1-2 on page 6.

Kernel Service Return Codes

Many of the **RTXC** Kernel services return a value that conveys information about the service's operation. This value is the *kernel service return code* (KSRC) value. The *Outputs* section of each service description lists and describes the KSRC values for the service.

Diagnostic Mode and Fatal Errors

The **RTXC** Kernel provides a diagnostic mode of operation to speed up the development process. When the application is generated in diagnostic mode, the **RTXC** Kernel performs numerous validity tests on the arguments being passed in kernel service calls. When an argument fails its validity test, the kernel passes a fatal error code to the system error function. The *Errors* section of each service description lists and describes the fatal errors that may be generated by the service. For a complete list of the error codes and the services that generate those codes, see Appendix A, "Fatal Error Codes."

Table 1-2. Kernel Service Return Value Types

Symbol	Description
TASK	Task handle
THREAD	Thread handle
PRIORITY	Priority of a task or a message
TSLICE	Number of TICKS in the time quantum for a time-sliced task
SEMA	Semaphore handle
SEMACOUNT	Number of signals that a semaphore has received
MBOX	Mailbox handle
MSGENV	Message envelope
QUEUE	Queue handle
PART	Memory partition handle
BLKSIZE	Size of a block of memory in a partition
MUTX	Mutex handle
EVNTSRC	Event Source handle
COUNTER	Counter handle
ALARM	Alarm handle
TICKS	Units of time maintained by the system time base
EXCPTN	Exception handle

Table 1-2. Kernel Service Return Value Types *(continued)*

Symbol	Description
KSRC	Kernel Service Return Code

Kernel Service Classes

The **RTXC/ss** component kernel services are divided into the following basic classes and subclasses:

- Thread Management
- Exception Management
- Pipe Management
- Event Source Management
- Counter Management
- Alarm Management

The **RTXC/ms** component kernel services are divided into the following basic classes and subclasses:

- Task Management
- Intertask Communication and Synchronization
  - Semaphores
  - Queues
  - Mailboxes
  - Messages
- Memory Partition Management
- Mutex Management

The **RTXC** Kernel also includes a number of kernel services that are independent of the object classes and are available for use in either component. These services are called Special Services.

The remaining sections describe each class and subclass. Each section includes a table listing all of the services within that class or subclass. The table contains a brief description of each service and a cross-reference to the detailed description of the service in the reference chapters of this book.

RTXC/ss Component Services

The **RTXC/ss** component of the **RTXC** Kernel features a single stack model with a low-latency thread scheduler and a small footprint, making it ideally suited for applications requiring high frequency interrupt processing, such as in digital signal processing. The following sections describe the object classes supported in the **RTXC/ss** component and their related kernel services.

Thread Management Services

The Thread Management services, listed in Table 1-3, allow for complete control of threads and their respective interactions, including scheduling threads and maintaining information about thread scheduling requests. For detailed descriptions, see Chapter 2, "Thread Services."

Table 1-3. Thread Services Summary

Service	Description	Zones	Ref.
`XX_ClearThreadGateBits`	Clear bits in a thread gate.	1 2 3	23
`XX_DecrThreadGate`	Decrement the thread gate.	1 2 3	26
`XX_DefThreadArg`	Define a new argument pointer for the thread.	1 2 3	28
`XX_DefThreadEntry`	Define or redefine a thread's entry point.	1 2 3	30
`XX_DefThreadEnvArg`	Define the thread's environment arguments.	2 3	32
`KS_DefThreadName`	Define the name of a previously opened dynamic thread.	3	34
`XX_DefThreadProp`	Define the thread's properties.	2 3	36

Table 1-3. Thread Services Summary *(continued)*

Service	Description	Zones	Ref.
`TS_DisableThreadScheduler`	Disable thread scheduling.	2	38
`TS_EnableThreadScheduler`	Enable thread scheduling.	2	39
`TS_GetThreadArg`	Get the argument pointer for a thread.	2	40
`TS_GetThreadBaseLevel`	Get a thread's base execution priority level.	2	42
`KS_GetThreadClassProp`	Get the Thread object class properties.	3	44
`TS_GetThreadCurrentLevel`	Get the Current Thread's execution priority level.	2	47
`XX_GetThreadEnvArg`	Get the pointer to the thread's environment arguments.	2 3	48
`XX_GetThreadGate`	Get the value of the thread's thread gate.	2 3	50
`TS_GetThreadGateLoadPreset`	Get the value of the Current Thread's thread gate and then load the thread gate with the value of the thread gate preset.	2	52
`XX_GetThreadGatePreset`	Read the content of the thread gate preset.	2 3	54
`TS_GetThreadID`	Read the Current Thread's ID.	2	55
`KS_GetThreadName`	Get the thread's name.	3	56
`XX_GetThreadProp`	Get the properties of the specified thread.	2 3	58
`XX_IncrThreadGate`	Increment a thread gate.	1 2 3	60
`KS_LookupThread`	Look up a thread by its name to get its handle.	3	62
`TS_LowerThreadLevel`	Lower the Current Thread's execution priority level.	2	64
`XX_ORThreadGateBits`	Set the bits in a thread gate using logical OR.	1 2 3	66

Table 1-3. Thread Services Summary *(continued)*

Service	Description	Zones	Ref.
`XX_PresetThreadGate`	Set the new thread gate value to the current thread gate preset value.	2 3	68
`TS_RaiseThreadLevel`	Raise the Current Thread's execution priority level.	2	70
`XX_ScheduleThread`	Schedule execution of a thread.	1 2 3	72
`XX_ScheduleThreadArg`	Schedule execution of a thread and define a new argument.	1 2 3	75
`XX_SetThreadGate`	Set new thread gate and thread gate preset values.	2 3	78
`XX_SetThreadGatePreset`	Set a new thread gate preset value.	2 3	80
`INIT_ThreadClassProp`	Initialize the Thread object class properties.	3	82
`XX_UnscheduleThread`	Unschedule execution of a thread.	1 2	84

Exception Services

The Exception services, listed in Table 1-4, provide a method of performing certain operations to facilitate the design and use of exception handlers. For detailed descriptions, see Chapter 3, “Exception Services.”

Table 1-4. Exception Services Summary

Service	Description	Zones	Ref.
`KS_CloseException`	End the use of a dynamic exception.	3	88
`KS_DefExceptionName`	Define the name of a previously opened exception.	3	90
`XX_DefExceptionProp`	Define the properties of an exception.	2 3	92
`INIT_ExceptionClassProp`	Initialize the Exception object class properties.	3	94
`KS_GetExceptionClassProp`	Get the Exception object class properties.	3	96
`KS_GetExceptionName`	Get the name of an exception.	3	98
`XX_GetExceptionProp`	Get the properties of an exception.	2 3	100
`KS_LookupException`	Look up an exception’s name to get its handle.	3	102
`KS_OpenException`	Allocate and name a dynamic exception.	3	104
`KS_UseException`	Look up a dynamic exception by name and mark it for use.	3	107

Pipe Services

The Pipe services, listed in Table 1-5, move data between a single producer and a single consumer and maintain information about pipe states. For detailed descriptions, see Chapter 4, “Pipe Services.”

Table 1-5. Pipe Services Summary

Service	Description	Zones	Ref.
`KS_ClosePipe`	End the use of a dynamic pipe.	3	110
`XX_DefPipeAction`	Define action to perform following `XX_PutFullPipeBuf` or `XX_PutEmptyPipeBuf` services.	2 3	112
`XX_DefPipeProp`	Define the properties of a pipe.	2 3	115
`KS_DefPipeName`	Define the name of a previously opened dynamic pipe.	3	118
`XX_GetEmptyPipeBuf`	Get an empty buffer from a specified pipe.	1 2 3	120
`XX_GetFullPipeBuf`	Get a full buffer from a specified pipe.	1 2 3	122
`XX_GetPipeBufSize`	Get the maximum usable size of buffers in the specified pipe.	1 2 3	124
`KS_GetPipeClassProp`	Get the Pipe class properties.	3	126
`KS_GetPipeName`	Get the pipe’s name.	3	128
`XX_GetPipeProp`	Get the pipe’s properties.	2 3	130
`XX_JamFullGetEmptyPipeBuf`	Put a full buffer at the front of a pipe and then get an empty buffer from the same pipe.	1 2 3	132
`XX_JamFullPipeBuf`	Put a full buffer at the front of a pipe.	1 2 3	136
`KS_LookupPipe`	Look up a pipe by name to get its handle.	3	138
`KS_OpenPipe`	Allocate and name a dynamic pipe.	3	140

Table 1-5. Pipe Services Summary *(continued)*

Service	Description	Zones	Ref.
`INIT_PipeClassProp`	Initialize the Pipe object class properties.	3	142
`XX_PutEmptyGetFullPipeBuf`	Put an empty buffer into a pipe and then get a full buffer from the same pipe.	1 2 3	144
`XX_PutEmptyPipeBuf`	Return an empty buffer to a pipe.	1 2 3	147
`XX_PutFullGetEmptyPipeBuf`	Put a full buffer into a pipe and then get an empty buffer from the same pipe.	1 2 3	149
`XX_PutFullPipeBuf`	Put a full buffer into a pipe.	1 2 3	152
`KS_UsePipe`	Look up a dynamic pipe by name and mark it for use.	3	154

Event Source Management Services

The Event Source Management directives, listed in Table 1-6, when used with the Counter services listed in Table 1-7 on page 16, provide a way of maintaining accumulators of the number of events occurring on various event sources in the system. For detailed descriptions, see Chapter 5, "Event Source Services."

Table 1-6. Event Source Services Summary

Service	Description	Zones	Ref.
`XX_ClearEventSourceAttr`	Clear one or more event source attributes.	2 3	158
`KS_CloseEventSource`	End the use of a dynamic event source.	3	160
`KS_DefEventSourceName`	Define the name of a previously opened event source.	3	162
`XX_DefEventSourceProp`	Define the event source's properties.	2 3	164
`INIT_EventSourceClassProp`	Initialize the Event Source object class properties.	3	167
`XX_GetEventSourceAcc`	Get the event sources's accumulator.	1 2 3	169
`KS_GetEventSourceClassProp`	Get the Event Source object class properties.	3	171
`KS_GetEventSourceName`	Get the event source's name.	3	173
`XX_GetEventSourceProp`	Get the event source's properties.	2 3	175
`KS_LookupEventSource`	Look up an event source by its name to get its handle.	3	177
`KS_OpenEventSource`	Allocate and name a dynamic event source.	3	179
`XX_ProcessEventSourceTick`	Process a tick on an event source.	1 2 3	181

Table 1-6. Event Source Services Summary *(continued)*

Service	Description	Zones	Ref.
`XX_SetEventSourceAcc`	Set the event source's accumulator to a specified value.	2 3	183
`XX_SetEventSourceAttr`	Set one or more event source attributes.	2 3	185
`KS_UseEventSource`	Look up a dynamic event source by name and mark it for use.	3	187

Counter Management Services

The Counter Management directives, listed in Table 1-7, when used with the Event Source services listed in Table 1-6 on page 14, provide a way of maintaining and accumulating tick counts based on the number of events occurring on various event sources in the system so that tasks and threads may perform operations with respect to those counters. For detailed descriptions, see Chapter 6, "Counter Services."

Table 1-7. Counter Services Summary

Service	Description	Zones	Ref.
`XX_ClearCounterAttr`	Clear one or more attributes for a counter.	2 3	190
`KS_CloseCounter`	End the use of a dynamic counter.	3	192
`INIT_CounterClassProp`	Initialize the Counter object class properties.	3	194
`KS_DefCounterName`	Define the name of a previously opened dynamic counter.	3	196
`XX_DefCounterProp`	Define the counter's properties.	2 3	198
`XX_GetCounterAcc`	Get the counter's tick accumulator.	1 2 3	202
`KS_GetCounterClassProp`	Get the Counter object class properties.	3	204
`KS_GetCounterName`	Get the counter's name.	3	206
`XX_GetCounterProp`	Get the counter's properties.	2 3	208
`XX_GetElapsedCounterTicks`	Compute the number of counter ticks that have elapsed between two events.	2 3	210
`KS_LookupCounter`	Look up a counter by name to get its handle.	3	214
`KS_OpenCounter`	Allocate and name a dynamic counter.	3	216

Table 1-7. Counter Services Summary *(continued)*

Service	**Description**	**Zones**	**Ref.**
`XX_SetCounterAcc`	Set the accumulator of a counter to a specified value.	2 3	218
`XX_SetCounterAttr`	Set one or more attributes for a counter.	2 3	220
`KS_UseCounter`	Look up a dynamic counter by name and mark it for use.	3	222

Alarm Management Services

The alarm-based directives, listed in Table 1-8, provide for the synchronization of tasks with events. They provide a generalized method of handling events relative to ticks that accumulate on an associated counter, allowing for time-based alarms as well as alarms based on other kinds of real-world events. For detailed descriptions, see Chapter 7, “Alarm Services.”

Table 1-8. Alarm Services Summary

Service	Description	Zones	Ref.
`XX_AbortAlarm`	Abort an active alarm.	2 3	226
`INIT_AlarmClassProp`	Initialize the Alarm object class properties.	3	228
`XX_ArmAlarm`	Arm and start an alarm.	2 3	230
`XX_CancelAlarm`	Make an active alarm inactive.	2 3	232
`KS_CloseAlarm`	End the use of a dynamic alarm.	3	234
`XX_DefAlarmAction`	End the use of a dynamic alarm.	2 3	236
`XX_DefAlarmActionArm`	Define the action to perform when an alarm expires and then arm and start the alarm.	2 3	238
`KS_DefAlarmName`	Define the name of a previously opened alarm.	3	240
`XX_DefAlarmProp`	Define the properties of a alarm.	2 3	242
`KS_DefAlarmSema`	Associate a semaphore with a alarm event.	3	244
`KS_GetAlarmClassProp`	Get the Alarm object class properties.	3	246
`KS_GetAlarmName`	Get the name of a alarm.	3	248
`XX_GetAlarmProp`	Get the properties of a alarm.	2 3	250
`KS_GetAlarmSema`	Get the handle of the semaphore associated with a alarm event.	3	252

Table 1-8. Alarm Services Summary *(continued)*

Service	Description	Zones	Ref.
`XX_GetAlarmTicks`	Get the number of counter ticks remaining until the expiration of an active alarm.	2 3	254
`KS_LookupAlarm`	Look up a alarm's name to get its handle.	3	256
`KS_OpenAlarm`	Allocate and name a dynamic alarm.	3	258
`XX_RearmAlarm`	Rearm and restart an active alarm.	2 3	260
`KS_TestAlarm`	Get the time, in ticks, remaining on an active alarm.	3	262
`KS_TestAlarmT`	Wait a specified number of ticks for an alarm to expire.	3	265
`KS_TestAlarmW`	Wait for a alarm to expire.	3	268
`KS_UseAlarm`	Look up a dynamic alarm by name and mark it for use.	3	270

Special Services

The Special services, listed in Table 1-9, perform special functions not based on the object classes. For detailed descriptions, see Chapter 8, “Special Services.”

Table 1-9. Special Services Summary

Service	Description	Zones	Ref.
XX_AllocSysRAM	Allocate a block of system RAM.	2 3	274
XX_DefFatalErrorHandler	Establish the system error function.	2 3	276
XX_GetFatalErrorHandler	Get the system error function.	2 3	278
XX_GetFreeSysRAMSize	Get the size of free system RAM.	2 3	279
KS_GetSysProp	Get the system properties.	3	280
KS_GetVersion	Get the version number of the **RTXC** Kernel.	3	282
INIT_SysProp	Initialize the RTXC system properties.	3	284

CHAPTER 2

Thread Services

In This Chapter

We describe the Thread kernel services in detail. The Thread kernel services schedule threads and maintain information about thread states.

XX_ClearThreadGateBits

Clear bits in a thread gate.

Zones

1 IS_ClearThreadGateBits
2 TS_ClearThreadGateBits
3 KS_ClearThreadGateBits

Synopsis

```
KSRC XX_ClearThreadGateBits (THREAD thread,
   GATEKEY gatekey)
```

Inputs

thread	The handle of the thread containing the thread gate whose bits are to be cleared. The thread handle can be that of the Current Thread or it can be zero (0), representing the Current Thread.
gatekey	A mask value containing the bits to clear in *thread*'s thread gate.

Description

The `XX_ClearThreadGateBits` kernel service clears bits in the thread gate of the specified *thread* according to the bits in *gatekey*. If the content of the thread gate is zero (0) before the service call, there is no change to the thread gate and control returns to the Current Thread without scheduling *thread*. If the resulting content of the thread gate is zero (0), the service schedules *thread*. At the same time the service schedules *thread*, it also loads the value of *thread*'s thread gate preset into the thread gate.

If an interrupt service routine (ISR) calls this service and the result requires scheduling *thread*, execution of *thread* cannot occur until the current ISR and all other ISRs are completed.

A preemption of the Current Thread may occur if *thread* is of a higher priority level than the Current Thread. In such a case, execution of *thread* is immediate. If *thread* is of the same or a lower priority level than that of the Current Thread, its execution does not occur until the termination of the Current Thread or at an even later time depending on the scheduling protocol in use for the given priority level.

A gatekey value of zero (0) causes no change to *thread*'s thread gate value and results in a normal return.

Output

This service returns a KSRC value as follows:

- RC_GOOD if the service was successful.
- RC_GATE_ALREADY_ZERO if the gate contained a value of zero (0) before clearing.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_THREAD if the specified thread ID is not valid.
- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.

Example

In Example 2-1, the Current Thread sets the gate for the thread specified in THREADA to 0xC00 and then clears the gate bits with two separate service calls. THREADA is scheduled only when both bits have been cleared.

Example 2-1. Clear Thread Gate Bits

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */


/* set THREADA's gate */
TS_SetThreadGate (THREADA, (GATEKEY)0xC00);

... perform some operations.

/* clear single bit in Thread's Gate */
TS_ClearThreadGateBits (THREADA, (GATEKEY)0x800);

... perform some operations.

/* clear single bit to schedule THREADA */
TS_ClearThreadGateBits (THREADA, (GATEKEY)0x400);

... THREADA was scheduled, continue
```

See Also

XX_ORThreadGateBits, page 66

XX_DecrThreadGate

Decrement the thread gate.

Zones

1 `IS_DecrThreadGate`
2 `TS_DecrThreadGate`
3 `KS_DecrThreadGate`

Synopsis

```
KSRC XX_DecrThreadGate (THREAD thread)
```

Input

thread	The handle of the thread containing the thread gate to decrement. The thread handle can be that of the Current Thread or it can be zero (0), representing the Current Thread.

Description

The `XX_DecrThreadGate` kernel service decrements by one the thread gate of the specified *thread*. If the resulting content of the thread gate is zero (0), the service schedules *thread*.

If an ISR calls this service and the result requires scheduling the thread, execution of the thread cannot occur until the current ISR and all other ISRs are completed.

A preemption of the Current Thread may occur if the *thread* whose gate becomes zero (0) after being decremented is of a higher priority level than the Current Thread. In such a case, execution of *thread* is immediate. If *thread* is of the same or a lower priority level than that of the Current Thread, its execution does not occur until the termination of the Current Thread or at an even later time depending on the scheduling protocol in use for the given priority level.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_GATE_UNDERFLOW` if gate contained a value less than or equal to zero (0) before decrement.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.

- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.

Example

In Example 2-2, the Current Thread sets the gate of the thread specified in THREADA to 2, and then decrements the gate value to zero with two separate Kernel service calls. THREADA is scheduled only when the gate value is zero.

Example 2-2. Decrement Thread Gate

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

/* set THREADA's gate */
TS_SetThreadGate (THREADA, (GATEKEY)2);

/* decrement Thread's Gate */
TS_DecrThreadGate (THREADA);

... perform some operations.

/* decrement Thread's Gate, which will schedule Thread */
TS_DecrThreadGate (THREADA);

... THREADA was scheduled, continue
```

See Also

XX_IncrThreadGate, page 60

XX_DefThreadArg

Define a new argument pointer for the thread.

Zones

1 `IS_DefThreadArg`
2 `TS_DefThreadArg`
3 `KS_DefThreadArg`

Synopsis

```
void XX_DefThreadArg (THREAD thread, void *parg)
```

Inputs

thread	The handle of the thread receiving the new argument definition.
parg	A pointer to the argument for the specified *thread*.

Description

The `XX_DefThreadArg` kernel service establishes a pointer, *parg*, to an argument containing one or more parameters to be used by the specified *thread*. Each time *thread* executes, it automatically receives the pointer to its arguments. The *parg* pointer may point to a scalar datum or a structure. The **RTXC** Kernel places no restrictions on the size or content of the argument structure.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-3 on page 29, the Current Thread defines the argument for the thread specified in `THREADA` and then schedules `THREADA`.

Example 2-3. Define Thread Argument Pointer

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

char buffer[80];         /* argument for THREADA */

/* define arguments for THREADA */
TS_DefThreadArg (THREADA, (void *)&buffer);

/* schedule THREADA */
TS_ScheduleThead (THREADA);

... continue
```

See Also

`XX_ScheduleThreadArg`, page 75

XX_DefThreadEntry

Define or redefine a thread's entry point.

Zones

1 `IS_DefThreadEntry`
2 `TS_DefThreadEntry`
3 `KS_DefThreadEntry`

Synopsis

```
void XX_DefThreadEntry (THREAD thread,
   void (*pentry) (void *, void *))
```

Inputs

thread	The handle of the thread being defined.
pentry	Address of thread's new entry point.

Description

The `XX_DefThreadEntry` kernel service establishes a pointer, *pentry*, to the entry point of the specified *thread*. The next time *thread* gets control of the CPU, it begins execution at the address defined by *pentry*.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.
- `FE_NULL_THREADENTRY` if the specified Thread entry address is null.

Example

In Example 2-4 on page 31, the Current Thread changes the entry point of the thread specified in `THREADA` to `newentry`, and then schedules `THREADA`.

Example 2-4. Define Thread Entry Point

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

extern void newentry (void *, void *);

/* define new entry point for THREADA */
TS_DefThreadEntry (THREADA, newentry);

/* schedule THREADA with its new entry */
TS_ScheduleThread (THREADA);

... continue
```

XX_DefThreadEnvArg

Define the thread's environment arguments.

Zones

2 `TS_DefThreadEnvArg`
3 `KS_DefThreadEnvArg`

Synopsis

```
void XX_DefThreadEnvArg (THREAD thread, void *parg)
```

Inputs

thread	The handle of the thread being defined.
parg	A pointer to a Thread environment arguments structure.

Description

The `XX_DefThreadEnvArg` kernel service establishes a pointer, *parg*, to a structure containing parameters that define the environment of the specified *thread*. Because threads inherently have no context saved or restored by **RTXC/ss** or **RTXC/ms** between execution cycles, the environment arguments structure serves as a place to save those parameters that are specific to a thread's operation. The **RTXC** Kernel places no restrictions on the size or content of the environment arguments structure.

Note: To use this service, you must enable the *Environment Arguments* attribute of the Thread class during system generation.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-5 on page 33, the Current Thread defines the environment arguments for the thread specified in `THREADA` and then schedules `THREADA`.

Example 2-5. Define Thread Environment Arguments Pointer

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

struct {
   int count;
   char buffer[80];
} envargA;                /* environment argument for THREADA */

/* define environment arguments for THREADA */
TS_DefThreadEnvArg (THREADA, (void *)&envargA);

/* schedule THREADA */
TS_ScheduleThead (THREADA);

... continuue
```

See Also

`KS_GetThreadClassProp`, page 44
`XX_GetThreadEnvArg`, page 48

KS_DefThreadName

Define the name of a previously opened dynamic thread.

Synopsis

```
KSRC KS_DefThreadName (THREAD thread,
   const char *pname)
```

Inputs

thread	The handle of the thread being defined.
pname	A pointer to a null-terminated name string.

Description

The `KS_DefThreadName` kernel service names or renames the specified dynamic *thread*. The service uses the null-terminated string pointed to by *pname* for the new name. The kernel only stores *pname* internally, which means that the same array cannot be used to build multiple dynamic thread names. Static threads cannot be named or renamed under program control.

Note: To use this service, you must enable the *Dynamics* attribute of the Thread class during system generation.

This service does not check for duplicate thread names.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_STATIC_OBJECT` if the thread being named is static.
- `RC_OBJECT_NOT_FOUND` if the *Dynamics* attribute of the Thread class is not enabled.
- `RC_OBJECT_NOT_INUSE` if the dynamic thread being named is still in the free pool of dynamic threads.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_THREAD` if the specified thread ID is not valid.

Example

Example 2-6 assigns the name `NewThread` to the thread specified in `dynthread` variable so other users may reference it by name.

Example 2-6. Define Dynamic Thread Name

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

THREAD dynthread;

if (KS_DefThreadName (dynthread, "NewTask") != RC_GOOD)
{
   ... Probably is a static thread. Deal with it here.
}

... else the naming operation was successful. Continue
```

See Also

`KS_GetThreadName`, page 56

XX_DefThreadProp

Define the thread's properties.

Zones

2 TS_DefThreadProp
3 KS_DefThreadProp

Synopsis

```
void XX_DefThreadProp (THREAD thread,
   THREADPROP *pthreadprop)
```

Inputs

thread	The handle of the thread being defined.
pthreadprop	A pointer to a Thread properties structure.

Description

The XX_DefThreadProp kernel service defines the properties of the specified *thread* by using the values contained in the THREADPROP structure pointed to by *pthreadprop*. You may use this service on static or dynamically allocated threads. It is typically used to define a static thread during system startup.

Example 2-7 shows the organization of the THREADPROP structure.

Example 2-7. Thread Properties Structure

```
typedef struct _threadprop
{
   KATTR attributes;        /* thread attributes */
   TLEVEL level;            /* thread base level */
   TORDER order;            /* thread order      */
   void (*threadentry)(void *, void *);
                            /* current entry point address */
} THREADPROP;
```

The entry point of the thread is specified in *threadentry* in the THREADPROP structure. At the initial definition of *thread*'s properties, *threadentry* should contain *thread*'s initial entry point. Afterwards, the content of *threadentry* is subject to change through the use of this kernel service as well as the more direct XX_DefThreadEntry kernel service.

> **Warning:** The values for *level* and *order* are provided for information only and must never be changed. Altering these values after their initial definition may cause errors or undesirable thread behavior.

Output

This service does not return a value.

Error

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_NULL_THREADENTRY` if the specified Thread entry address is null.

Example

During system initialization, the startup routine must create and initialize the Thread object class and define the properties of all the static Threads before the start of Thread scheduling, as illustrated in Example 2-8.

Example 2-8. Define Thread Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

extern const KCLASSPROP threadclassprop;
extern const THREADPROP threadprop[];

int objnum;
KSRC ksrc;

   /* initialize the THREAD class/object data */
   if ((ksrc = INIT_ThreadClassProp (&threadclassprop))
       != RC_GOOD)
      return ksrc;

   for (objnum = 1; objnum <= threadclassprop.n_statics; objnum++)
   {
      TS_DefThreadProp (objnum, &threadprop[objnum]);
   }
```

See Also

`XX_GetThreadProp`, page 58

TS_DisableThreadScheduler

Disable thread scheduling.

Synopsis

`void TS_DisableThreadScheduler (void)`

Inputs

This service has no inputs.

Description

The `TS_DisableThreadScheduler` kernel service disables further scheduling of threads by the RTXC**/ss** Scheduler until such time as the Current Thread re-enables thread scheduling.

Output

This service does not return a value.

Example

In Example 2-9, the Current Thread disables Thread scheduling during some critical code section, then re-enables Thread scheduling when the critical section is complete.

Example 2-9. Disable Thread Scheduling

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

/* disable Thread scheduling */
TS_DisableThreadScheduler ();

... execute critical code

/* enable Thread scheduling */
TS_EnableThreadScheduler ();

... continue
```

See Also

`TS_EnableThreadScheduler`, page 39

TS_EnableThreadScheduler

Enable thread scheduling.

Synopsis

`void TS_EnableThreadScheduler (void)`

Inputs

This service has no inputs.

Description

The `TS_EnableThreadScheduler` kernel service enables scheduling of threads by the RTXC**/ss** Scheduler after being previously disabled. The service returns the priority level of the Scheduler to the Current Thread's base execution level.

Output

This service does not return a value.

Example

In Example 2-10, after performing some critical function with Thread scheduling disabled, the Current Thread re-enables Thread scheduling.

Example 2-10. Enable Thread Scheduling

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

/* disable Thread scheduling */
TS_DisableThreadScheduler ();

... execute critical code

/* enable Thread scheduling */
TS_EnableThreadScheduler ();

... continue
```

See Also

`TS_DisableThreadScheduler`, page 38

TS_GetThreadArg

Get the argument pointer for a thread.

Synopsis

```
void * TS_GetThreadArg(THREAD thread)
```

Inputs

thread	The handle of the thread containing the argument definition.

Description

The `TS_GetThreadArg` kernel service locates and returns the current value of the thread argument for the specified *thread*. This service would not typically be used by the Current Thread because each time a thread executes, it automatically receives the pointer to its argument. Therefore, the specified *thread* typically different than the current thread.

Output

This service returns the value of the thread argument. The returned pointer may be a scalar datum or a structure. If it is a structure, the **RTXC** Kernel places no restrictions on the size or content of the argument structure.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-11 on page 41, the Current Thread retrieves the argument for the thread specified in `THREADA,` verifies it as being non-zero, and if so, schedules `THREADA`. If the thread argument is zero, it takes a different path.

Example 2-11. Get Thread Argument

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kproject.h"     /* */
#include "kthread.h"      /* THREADA */

/* get argument for THREADA */
if (TS_GetThreadArg(THREADA) <> (void *)0);
{
   /* schedule THREADA */
   TS_ScheduleThead(THREADA);

... continue
}
else
{
   do something else…
}
```

See Also

`XX_DefThreadArg`, page 28

TS_GetThreadBaseLevel

Get a thread's base execution priority level.

Synopsis

```
TLEVEL TS_GetThreadBaseLevel (THREAD thread)
```

Input

thread	The handle of the thread whose base execution level is being read. The value of *thread* may be zero (0), representing the Current Thread.

Description

The `TS_GetThreadBaseLevel` kernel service reads the base execution priority level of the specified *thread* and returns it to the caller.

Output

This service returns a `TLEVEL` type value containing *thread*'s base execution priority level.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

Example 2-12 on page 43, the Current Thread reads the base execution level of the thread specified in `THREADA` and raises the Current Thread's level if it is less than `THREADA`'s base execution level. Remember that higher priority levels are numerically smaller than lower priority levels.

Example 2-12. Read Thread Base Execution Priority Level

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

TLEVEL tlevel;

/* get THREADA's base execution level */
tlevel = TS_GetThreadBaseLevel (THREADA);

/* check current priority level against THREADA's */
if (TS_GetThreadCurrentLevel () > tlevel)
{
   /* THREADA priority is higher than Current Thread, */
   /* so raise Current Thread's level.                */
   TS_RaiseThreadLevel (tlevel);
}

... continue
```

See Also

`TS_GetThreadCurrentLevel`, page 47

KS_GetThreadClassProp

Get the Thread object class properties.

Synopsis

```
const KCLASSPROP * KS_GetThreadClassProp (int *pint)
```

Input

pint	A pointer to an integer variable in which to store the current number of unused dynamic threads.

Description

The `KS_GetThreadClassProp` kernel service obtains a pointer to the `KCLASSPROP` structure that was used during system initialization by the `INIT_ThreadClassProp` service to initialize the Thread object class properties.

If the *pint* pointer contains a non-zero address, the current number of unused dynamic threads is stored in the indicated address. If *pint* contains a null pointer (`(int *)0`), the service ignores the parameter. If the Thread object class properties do not include the *Dynamics* attribute, the service stores a value of zero (0) at the address contained in *pint*.

Example 2-13 shows the organization of the `KCLASSPROP` structure.

Example 2-13. Object Class Properties Structure

```
typedef struct
{
   KATTR attributes;
   KOBJECT n_statics;        /* number of static objects */
   KOBJECT n_dynamics;       /* number of dynamic objects */
   short objsize;            /* used for calculating offsets */
   short totalsize;          /* used to alloc object array RAM */
   ksize_t namelen;          /* length of the name string */
   const char *pstaticnames;
} KCLASSPROP;
```

The *attributes* element of the Thread property structure supports the class property attributes and corresponding masks listed in Table 2-1 on page 45.

Table 2-1. Thread Class Attributes and Masks

Attribute	Mask
Static Names	ATTR_STATIC_NAMES
Dynamics	ATTR_DYNAMICS
Thread Gates	ATTR_THREAD_GATES
Environment Arguments	ATTR_THREAD_ENV
Thread Arguments	ATTR_THREAD_ARG

Output

If successful, this service returns a pointer to a `KCLASSPROP` structure.

If the Thread class is not initialized, the service returns a null pointer (`(KCLASSPROP *)0`).

If *pint* is not null (`(int *)0`), the service returns the number of available dynamic threads in the variable pointed to by *pint*.

Example

In Example 2-14 on page 46, the Current Thread needs access to the information contained in the `KCLASSPROP` structure for the Thread object class.

Example 2-14. Read Thread Object Class Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KCLASSPROP *pthreadclassprop;
int free_dyn;

/* Get the thread kernel object class properties */
if ((pthreadclassprop = KS_GetThreadClassProp (&free_dyn))
   == (KCLASSPROP *)0)
{
   putline ("Thread Class not initialized");
}
else
{
   ...thread object class properties are available for use
      "free_dyn" contains the number of available dynamic threads
}
```

See Also

`INIT_ThreadClassProp`, page 82

TS_GetThreadCurrentLevel

Get the Current Thread's execution priority level.

Synopsis

`TLEVEL TS_GetThreadCurrentLevel (void)`

Inputs

This service has no inputs.

Description

The `TS_GetThreadCurrentLevel` kernel service reads the Current Thread's execution priority level.

Output

This service returns a `TLEVEL` type value containing the Current Thread's execution priority level.

Example

Example 2-15, the Current Thread compares its current level with the base execution level of the thread specified in `THREADA` and, if its level is less than `THREADA`, raises its level to that of `THREADA`. Remember that higher priority levels are numerically smaller than lower priority levels.

Example 2-15. Read Thread Execution Priority Level

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

TLEVEL tlevel;

/* get THREADA's base execution level */
tlevel = TS_GetThreadBaseLevel (THREADA);

/* see if current priority level is lower than THREADA's */
if ( TS_GetThreadCurrentLevel () > tlevel )
{
   /* Yes, it is, so raise current Thread's level. */
   TS_RaiseThreadLevel (tlevel);
}

... continue
```

See Also

`TS_GetThreadBaseLevel`, page 42

XX_GetThreadEnvArg

Get the pointer to the thread's environment arguments.

Zones

2 `TS_GetThreadEnvArg`
3 `KS_GetThreadEnvArg`

Synopsis

```
void * XX_GetThreadEnvArg (THREAD thread)
```

Input

thread	The handle of the thread whose environment arguments pointer is being read.

Note: The Current Thread already has the pointer to its environment arguments (if defined), having received it as one of two parameters passed to it by the **RTXC/ss** Scheduler. It would be unnecessary for the Current Thread to use this service when referring to itself. Instead, the value of *thread* would more likely be the handle of a thread other than that of the Current Thread.

Description

The `XX_GetThreadEnvArg` kernel service reads the pointer to the environment arguments structure for the specified *thread* and returns that pointer to the caller.

Note: To use this service, you must enable the *Environment Arguments* attribute of the Thread class during system generation.

Output

This service returns a pointer to *thread*'s environment structure as follows:

- a valid non-null pointer if the service was successful
- a null (0) pointer if the thread's environment arguments have not been defined.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

Example 2-16, the Current Thread reads the environment arguments for the thread specified in `THREADA` and performs some operation if *count* is non-zero.

Example 2-16. Read Thread Environment Arguments Pointer

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

typedef struct {
   int count;
   char buffer[80];
} envargA;                /* environment argument for THREADA */

envargA *envarg;

/* get THREADA's environment arguments */
envarg = (envargA *)TS_GetThreadEnvArg (THREADA);

/* test the count */
if (envarg->count != 0)
{
   ... perform some operation
}

... continue
```

See Also

`XX_DefThreadEnvArg`, page 32

XX_GetThreadGate

Get the value of the thread's thread gate.

Zones

2 `TS_GetThreadGate`
3 `KS_GetThreadGate`

Synopsis

```
GATEKEY XX_GetThreadGate (THREAD thread)
```

Input

thread	The handle of the thread containing the thread gate to read. The thread handle can be that of the Current Thread, which is assumed if the thread handle is zero (0).

Description

The `XX_GetThreadGate` kernel service reads the thread gate content of the specified *thread* and returns it to the caller. No change occurs to the value of the thread gate.

Output

This service returns the thread gate's content as a `GATEKEY` type value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-17 on page 51, the Current Thread reads its own thread gate value and performs some operation `gatevalue` times.

Example 2-17. Read Thread Gate

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

GATEKEY gatevalue;
int i;

gatevalue = TS_GetThreadGate (SELFTHREAD);

for ( i = 1; i <= gatevalue; i++ )
{
   ... perform some operation
}

... continue
```

See Also

`XX_SetThreadGate`, page 78

TS_GetThreadGateLoadPreset

Get the value of the Current Thread's thread gate and then load the thread gate with the value of the thread gate preset.

Synopsis

```
GATEKEY TS_GetThreadGateLoadPreset (void)
```

Inputs

This service has no inputs.

Description

The `TS_GetThreadGateLoadPreset` kernel service reads the value of the Current Thread's thread gate and returns it to the Current Thread. At the same time, the service also gets the value of the Current Thread's thread gate preset and moves it into the associated thread gate.

If the Current Thread has been rescheduled at the time of its request for this service, the service removes the scheduling request, allowing the thread to continue to operate if it chooses.

Output

This service returns the value of the Current Thread's thread gate.

Example

In Example 2-18 on page 53, the Current Thread reads its thread gate value and simultaneously presets its thread gate in preparation for the next execution cycle. It uses the thread gate value it reads as the counter for the number of times to execute an internal loop before returning control of the CPU.

Example 2-18. Read and Preset Thread Gate

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

void threada (void *env, void *arg)
{
   GATEKEY loopct;
   int i;

   /* preset thread gate (preset = 2) */
   loopct = TS_GetThreadGateLoadPreset ();

   for (i=0; i<=loopct; i++)
   {
      …do some thing
   }
   return
}
```

XX_GetThreadGatePreset

Read the content of the thread gate preset.

Zones

2 `TS_GetThreadGatePreset`
3 `KS_GetThreadGatePreset`

Synopsis

`GATEKEY XX_GetThreadGatePreset (THREAD thread)`

Inputs

thread	The handle of the thread containing the thread gate to read. The thread handle can be that of the Current Thread, which is assumed if the thread handle is zero (0).

Description

The `XX_GetThreadGatePreset` kernel service reads the content of the thread gate preset of the specified *thread* and returns the content to the caller. No change occurs to the value of the thread gate or the thread gate preset.

Output

This service returns the content of the thread gate preset as a `GATEKEY` type value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-19, the Current Thread reads its own thread gate preset value.

Example 2-19. Read Thread Gate Preset

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

GATEKEY presetvalue;

presetvalue = TS_GetThreadGatePreset (SELFTHREAD);

... continue
```

TS_GetThreadID

Read the Current Thread's ID.

Synopsis

`THREAD TS_GetThreadID (void)`

Inputs

This service has no inputs.

Description

The `TS_GetThreadID` kernel service reads the Current Thread's ID and returns it to the Current Thread.

Output

This service returns the Current Thread's ID as a `THREAD` type value.

Example

Example 2-20, the Current Thread reads its own thread ID.

Example 2-20. Read Current Thread ID

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

THREAD thread;

thread = TS_GetThreadID ();

... continue
```

KS_GetThreadName

Get the thread's name.

Synopsis

```
char * KS_GetThreadName (THREAD thread)
```

Input

thread	The handle of the thread being queried.

Description

The `KS_GetThreadName` kernel service obtains a pointer to the null-terminated string containing the name of the specified *thread*. The thread may be static or dynamic.

Note: To use this service on static threads, you must enable the *Static Names* attribute of the Thread class during system generation.

Output

If *thread* has a name, this service returns a pointer to its null-terminated name string.

If *thread* has no name, the service returns a null pointer (`(char *)0`).

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_THREAD` if the specified thread ID is not valid.

Example

In Example 2-21 on page 57, the Current Task reports the name of the dynamic thread specified in `dynthread`.

Example 2-21. Read Thread Name

```
#include <stdio.h>          /* standard i/o */
#include "rtxcapi.h"        /* RTXC Kernel Service prototypes */

static char buf[128];
THREAD dynthread;
char *pname;

if ((pname = KS_GetThreadName (dynthread)) == (char *)0)
   sprintf (buf, "Thread %d has no name", dynthread);
else
   sprintf (buf, "Thread %d name is %s", dynthread, pname);

putline (buf);
```

See Also

`KS_DefThreadName`, page 34

XX_GetThreadProp

Get the properties of the specified thread.

Zones

2 TS_GetThreadProp
3 KS_GetThreadProp

Synopsis

```
void XX_GetThreadProp (THREAD thread,
   THREADPROP *pthreadprop)
```

Inputs

thread	The handle of the thread being queried. The thread handle can be that of the Current Thread, which is assumed if the thread handle is zero (0).
pthreadprop	A pointer to a Thread properties structure in which to store the thread's properties.

Description

The XX_GetThreadProp kernel service obtains all of the property values of the specified *thread* in a single call. The *thread* input argument may specify a static or a dynamic thread. The service stores the property values in the THREADPROP structure pointed to by *pthreadprop* and returns to the caller.

The THREADPROP structure has the following organization:

```
typedef struct _threadprop
{
   KATTR attributes;        /* thread attributes */
   TLEVEL level;            /* thread base level */
   TORDER order;            /* thread order      */
   void (*threadentry)(void *, void *);
                            /* current entry point address */
} THREADPROP;
```

The entry point of the thread is specified by *threadentry* in the THREADPROP structure. At the initialization of the thread, *threadentry* should contain *thread*'s initial entry point. The content of *threadentry* is subject to change through the use of the XX_DefThreadProp kernel service as well as the more direct XX_DefThreadEntry kernel service.

Output

This service returns *thread*'s properties in the property structure pointed to by *pthreadprop*.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_THREAD if the specified thread ID is not valid.
- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.

Example

In Example 2-22, the Current Thread changes the entry point for the thread specified in THREADA. The Current Thread first obtains the current properties of THREADA, then modifies the entry point in the THREADPROP structure. It then uses the XX_DefThreadProp service to redefine the properties for THREADA. The same results can be obtain using the XX_DefThreadEntry service.

Example 2-22. Read Thread Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

extern void newentry (void *, void *);

THREADPROP threadprop;

/* get current Thread Properties */
TS_GetThreadProp (THREADA, &threadprop);

/* modify just the entry element */
threadprop.threadentry = newentry;

/* define the new Thread properties */
TS_DefThreadProp (THREADA, &threadprop);

... continue
```

See Also

XX_DefThreadProp, page 36

XX_IncrThreadGate

Increment a thread gate.

Zones

1 `IS_IncrThreadGate`
2 `TS_IncrThreadGate`
3 `KS_IncrThreadGate`

Synopsis

```
KSRC XX_IncrThreadGate (THREAD thread)
```

Input

thread	The handle of the thread containing the thread gate to increment. The thread handle can be that of the Current Thread or it can be zero (0), representing the Current Thread.

Description

The `XX_IncrThreadGate` kernel service adds one (1) to the contents of the thread gate of the specified *thread*. Following the addition, the service schedules *thread*. The value of the thread gate remains as incremented until another request to increment the thread gate occurs or until *thread* executes and reads the thread gate and simultaneously resets it using the `TS_GetThreadGateLoadPreset` kernel service.

Incrementing the thread gate does not cause a rollover of the thread gate should *thread* fail to run or to read and reset the content of the thread gate. The value of the thread gate contents is limited to the maximum unsigned integer value capable of being stored in the thread gate as a value of the `GATEKEY` type.

If an ISR calls this service, *thread* is scheduled for execution. However, execution of *thread* cannot occur until the current ISR and all other ISRs are completed.

A preemption of the Current Thread occurs if *thread* is of a higher priority level than the Current Thread. In such a case, execution of *thread* is immediate. If *thread* is of the same or lower priority level, its execution does not occur until the termination of the Current Thread or at an even later time depending on the order number of the thread and the scheduling protocol in use for the given priority level.

A task incrementing a thread gate is always preempted because *thread* is scheduled at Zone 2, which is of higher priority than the task operation at Zone 3.

Output

This service returns a KSRC value as follows:

- RC_GOOD if the service was successful.
- RC_GATE_OVERFLOW if the gate content is clipped at its maximum unsigned value.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_THREAD if the specified thread ID is not valid.
- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.

Example

In Example 2-23, the Current Thread increments the thread gate of the thread specified in THREADA.

Example 2-23. Increment Thread Gate

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

/* increment THREADA's Gate */
if (TS_IncrThreadGate (THREADA) != RC_GOOD)
{
   ...must have had a gate overflow. Something may be wrong.
}
else
{
   ...thread gate incremented, THREADA scheduled, continue
}
```

See Also

XX_DecrThreadGate, page 26
XX_GetThreadGate, page 50
XX_SetThreadGate, page 78

KS_LookupThread

Look up a thread by its name to get its handle.

Synopsis

```
KSRC KS_LookupThread (const char *pname,
   THREAD *pthread)
```

Inputs

pname	A pointer to the null-terminated name string for the thread.
pthread	A pointer to a variable in which to store the matching thread's handle, if found.

Description

The `KS_LookupThread` kernel service obtains the handle of a static or dynamic thread whose name matches the null-terminated string pointed to by *pname*. The lookup process terminates when it finds a match between the specified string and a static or dynamic thread name or when it finds no match. The service searches dynamic names, if any, first. If a match is found, the service stores the thread handle in the variable pointed to by *pthread*.

Note: To use this service on dynamic threads, you must enable the *Dynamics* attribute of the Thread class during system generation.

To use this service on static threads, you must enable the *Static Names* attribute of the Thread class during system generation.

This service has no effect on the registration of the specified thread by the Current Task.

The time required to perform this operation varies with the number of thread names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search succeeds. The service stores the matching thread's handle in the variable pointed to by *pthread*.

- RC_OBJECT_NOT_FOUND if the service finds no matching thread name.

Example

In Example 2-24, the Current Task needs to use the DynThread2 dynamic thread. If the thread name is found, the example outputs the thread handle to the console in a brief message.

Example 2-24. Look Up Thread by Name

```
#include <stdio.h>         /* standard i/o */
#include "rtxcapi.h"       /* RTXC Kernel Service prototypes */

THREAD dynthread;
static char buf[128];

/* lookup the thread name to see if it exists */
if (KS_LookupThread ("DynThread2", &dynthread) != RC_GOOD)
{
   putline ("Thread DynThread2 name not found");
}
else  /* thread exists */
{
   sprintf (buf, "DynThread2 is thread %d", dynthread);
   putline (buf);
}
```

See Also

KS_DefThreadName, page 34
KS_GetThreadName, page 56

TS_LowerThreadLevel

Lower the Current Thread's execution priority level.

Synopsis

```
KSRC TS_LowerThreadLevel (TLEVEL newlevel)
```

Input

newlevel	The new temporary execution priority level for the Current Thread.

Description

The `TS_LowerThreadLevel` kernel service lowers the Current Thread's execution priority level to the value specified in *newlevel*. If *newlevel* is zero (0), the service returns the Current Thread to its base execution priority level. If *newlevel* specifies an execution priority level less than the Current Thread's base execution priority level, the thread's base execution priority level is substituted for the value of *newlevel* and the operation proceeds but with a notification of the condition.

Note: The priority of a level decreases as its numerical value increases.

This service may cause a preemption of the Current Thread if *newlevel* or the base execution priority level of the Current Thread is a lower execution priority than a thread scheduled by the Current Thread during the time when it is at a priority level higher than its base execution priority level.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_REQUESTED_LEVEL_TOO_LOW` if the new execution priority level is lower than the base execution priority level of the thread.

Error

This service may generate the following fatal error code:

- `FE_ILLEGAL_LEVEL` if the specified level is not valid.

Example

In Example 2-25, the Current Thread raises its current execution level to the maximum level, executes some critical function, and then lowers its level back to its previously defined value.

Example 2-25. Lower Current Thread Execution Priority Level

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

TLEVEL tlevel, higherlevel;

/* get Current Thread's execution level */
tlevel = TS_GetThreadCurrentLevel ();

/* raise execution level to a higher level */
TS_RaiseThreadLevel (higherlevel);

... perform some critical operation

/* restore execution level to previously defined value */
TS_LowerThreadLevel (tlevel);

... continue
```

See Also

TS_RaiseThreadLevel, page 70

XX_ORThreadGateBits

Set the bits in a thread gate using logical OR.

Zones

1 `IS_ORThreadGateBits`
2 `TS_ORThreadGateBits`
3 `KS_ORThreadGateBits`

Synopsis

```
KSRC XX_ORThreadGateBits (THREAD thread,
   GATEKEY gatekey)
```

Inputs

thread	The handle of the thread containing the thread gate to change. The thread handle can be that of the Current Thread or it can be zero (0), representing the Current Thread.
gatekey	A mask containing the bits to set in the thread gate of the specified *thread*. A value of zero (0) is treated as a non-operation.

Description

The `XX_ORThreadGateBits` kernel service sets bits in the thread gate of the specified *thread*. Because the service uses a logical OR operation to set bits in the thread gate, the operation results in the thread gate having a non-zero value if *gatekey* is non-zero. As a result, the service schedules *thread*. The value of the thread gate remains intact until *thread* reads it and simultaneously resets it using the `TS_GetThreadGateLoadPreset` kernel service, or until a `XX_IncrThreadGate` or `XX_ORThreadGateBits` kernel service occurs before *thread* can execute.

If the content of *gatekey* is zero (0), there is no change to the thread gate and control returns to the caller without scheduling *thread*.

If an ISR calls this service, it causes the scheduling of *thread*. However, execution of *thread* cannot occur until the current ISR and all other ISRs are completed.

A preemption of the Current Thread may occur if *thread* is of a higher priority level than the Current Thread. In such a case, execution of *thread* is immediate. If *thread* is of the same or lower priority level, its execution does not occur until the termination of the Current Thread or at an even later time depending on the order number of the thread and the scheduling protocol in use for the given priority level.

Output

This service returns a KSRC value as follows:

- RC_GOOD if the service was successful.
- RC_GATE_OVERSIGNAL if gate contains bits that are set (already a one (1)) before the OR operation. This return code is not necessarily an error condition but the service reports it in case the caller needs to take action should it occur.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_THREAD if the specified thread ID is not valid.
- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.

Example

In Example 2-26, the Current Thread ORs a bit into the gate of the thread specified in THREADA, which has the additional effect of scheduling THREADA.

Example 2-26. Set Thread Gate Bits

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

/* OR bit in thread gate */
TS_ORThreadGateBits (THREADA, (GATEKEY)0x800);

... THREADA was scheduled, continue
```

See Also

XX_ClearThreadGateBits, page 23
XX_GetThreadGatePreset, page 54

XX_PresetThreadGate

Set the new thread gate value to the current thread gate preset value.

Zones

2 `TS_PresetThreadGate`
3 `KS_PresetThreadGate`

Synopsis

```
void XX_PresetThreadGate(THREAD thread)
```

Inputs

thread	The handle of the thread containing the thread gate to be set. The thread handle can be that of the Current Thread or it can be zero (0), representing the Current Thread.

Description

The `XX_PresetThreadGate` kernel service moves the content of the specified *thread*'s thread gate preset into that thread's thread gate value. The new thread gate value is put into effect immediately. There is no effect on the thread gate preset value.

Note: This service does not cause *thread* to be scheduled.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-27 on page 69, the Current Thread changes its thread gate to its thread gate preset values in preparation for taking some new operational path on its next execution cycle.

Example 2-27. Set Thread Gate with Thread Gate Preset

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kproject.h"

TS_PresetThreadGate(SELFTHREAD);

... continue
```

See Also

`XX_GetThreadGate`, page 50
`TS_GetThreadGateLoadPreset`, page 52
`XX_GetThreadGatePreset`, page 54
`XX_SetThreadGate`, page 78
`XX_SetThreadGatePreset`, page 80

TS_RaiseThreadLevel

Raise the Current Thread's execution priority level.

Synopsis

```
KSRC TS_RaiseThreadLevel (TLEVEL newlevel)
```

Input

newlevel	The new execution priority level for the Current Thread. It can be the handle of the level at the desired priority.

Description

The `TS_RaiseThreadLevel` kernel service temporarily raises the Current Thread's execution priority level to the value specified in *newlevel*. If the value of *newlevel* is zero (0), the service causes no change to the thread's priority level and returns a value indicative of the condition. If *newlevel* specifies an execution priority level less than the Current Thread's base execution priority level, the base execution priority level is substituted for the value of *newlevel* and the operation proceeds but with a notification of the condition.

After raising its execution priority, the thread must lower its execution priority level to the original level before completing operation by calling the `TS_LowerThreadLevel` service.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_ILLEGAL_LEVEL` if the new execution priority level is zero (0).
- `RC_REQUESTED_LEVEL_TOO_LOW` if the new execution priority level is lower than the Current Thread's base execution priority level.

Error

This service may generate the following fatal error code:

- `FE_ILLEGAL_LEVEL` if the specified level is not valid.

Example

In Example 2-28 on page 71, the Current Thread raises its current execution level to level 2, executes some function, and then lowers its level back to its previously defined value.

Example 2-28. Raise Current Thread Execution Priority Level

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

TLEVEL tlevel;

/* get current Thread's execution level */
tlevel = TS_GetThreadCurrentLevel ();

/* raise execution level to 2 */
TS_RaiseThreadLevel ((TLEVEL)2);

... perform some critical operation

/* restore execution level to previously defined value */
TS_LowerThreadLevel (tlevel);

... continue
```

See Also

`TS_LowerThreadLevel`, page 64

XX_ScheduleThread

Schedule execution of a thread.

Zones

1 `IS_ScheduleThread`
2 `TS_ScheduleThread`
3 `KS_ScheduleThread`

Synopsis

```
KSRC XX_ScheduleThread (THREAD thread)
```

Input

thread	The handle of the thread to schedule. A thread value of zero (0) is legal, allowing the Current Thread to schedule itself.

Description

The `XX_ScheduleThread` kernel service schedules the specified *thread* for execution.

If the Current Thread calls this service, *thread* preempts the Current Thread if *thread* has an execution priority level higher than the Current Thread. Otherwise, there is no preemption and *thread* begins execution after the completion of the Current Thread's operation, but not necessarily immediately afterwards.

If the Current Task (in Zone 3) calls this service, *thread* preempts the Current Task regardless of execution priority because *thread* executes in Zone 2.

If an ISR calls this service, the ISR must be completely serviced as well as any other ISRs and threads of higher execution priority levels before *thread* may begin its execution. If the Current Thread (Zone 2) is interrupted and is of lower execution priority than *thread*, the Current Thread is preempted to allow *thread* to start immediately.

Note: This service does not define or redefine *thread*'s argument pointer or environment argument pointer. As a consequence, the **RTXC/ss** Scheduler passes those two values as they exist at the time of *thread*'s next execution cycle.

> **Warning:** If a thread has been scheduled more than once since its last execution cycle, it is considered to be over scheduled. Regardless of the number of schedule requests in an over-scheduled condition, only one will cause the thread to execute. The condition is not necessarily an error but the **RTXC** Kernel reports the condition in case the caller needs to take special action.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_OVER_SCHEDULED` if the service attempts to schedule a thread and the thread has already been scheduled.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-29, the Current Thread schedules the thread specified in `THREADA` to execute.

Example 2-29. Schedule Thread Execution

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

if (TS_ScheduleThread (THREADA) != RC_GOOD)
{
   ... THREADA already scheduled
}
else
{
   ... THREADA was scheduled, continue
}

... continue
```

See Also

`TS_DisableThreadScheduler`, page 38
`TS_EnableThreadScheduler`, page 39
`XX_ScheduleThreadArg`, page 75

XX_ScheduleThreadArg

Schedule execution of a thread and define a new argument.

Zones

1 `IS_ScheduleThreadArg`
2 `TS_ScheduleThreadArg`
3 `KS_ScheduleThreadArg`

Synopsis

```
KSRC XX_ScheduleThreadArg (THREAD thread,
   void *parg)
```

Inputs

thread	The handle of the thread to schedule. A *thread* value of zero (0) is legal, allowing the Current Thread can schedule itself.
parg	A pointer to the execution argument of the specified *thread*.

Description

The `XX_ScheduleThreadArg` kernel service schedules the specified *thread* for execution using the arguments pointed to by *parg*.

If the Current Thread calls this service, *thread* preempts the Current Thread if *thread* has an execution priority level higher than the Current Thread. Otherwise, there is no preemption and *thread* begins execution after the completion of the Current Thread's operation.

If the Current Task (in Zone 3) calls this service, *thread* preempts the Current Task regardless of execution priority because *thread* executes in Zone 2.

If an ISR calls this service, the ISR must be completely serviced as well as any other ISRs and threads of higher execution priority levels before *thread* may begin its execution. If the Current Thread (Zone 2) is interrupted and is of lower execution priority than *thread*, the Current Thread is preempted to allow *thread* to start immediately.

Note: The *parg* argument can be a pointer or it can be a scalar datum. If the former, it should not be a null pointer (`(void *)0`). If used as a scalar, *parg* can be any legal value.

Warning: If a thread has been scheduled more than once since its last execution cycle, it is considered to be over scheduled. Regardless of the number of schedule requests in an over-scheduled condition, only one will cause the thread to execute. The condition is not necessarily an error but the **RTXC** Kernel reports the condition in case the caller needs to take special action.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_OVER_SCHEDULED` if the service attempts to schedule a thread and the thread has already been scheduled.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-30 on page 77, the Current Thread schedules the thread specified in `THREADA` to execute and defines a new argument.

Example 2-30. Schedule Thread Execution with New Argument

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

char newbuf[80];

if (TS_ScheduleThreadArg (THREADA, (void *)&newbuf[0]) != RC_GOOD)
{
... THREADA already scheduled
}
else
{
... THREADA was scheduled, continue
}

... continue
```

See Also

XX_ScheduleThread, page 72

XX_SetThreadGate

Set new thread gate and thread gate preset values.

Zones

2 `TS_SetThreadGate`
3 `KS_SetThreadGate`

Synopsis

```
void XX_SetThreadGate (THREAD thread,
   GATEKEY gatekey)
```

Inputs

thread	The handle of the thread containing the thread gate to being set. The thread handle can be that of the Current Thread, which is assumed if the thread handle is zero (0).
gatekey	The new value to store in the thread gate and thread gate preset of the specified *thread*.

Description

The `XX_SetThreadGate` kernel service moves the content of *gatekey* into the specified *thread*'s thread gate and thread gate preset. The new thread gate value is put into effect immediately. The new thread gate preset value does not have any effect until the next time *thread* is scheduled as the result of a call to the `XX_ClearThreadGateBits`, `XX_DecrThreadGate`, or `TS_GetThreadGateLoadPreset` services.

Note: This service does not cause *thread* to be scheduled.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-31, the Current Thread changes its thread gate and thread gate preset values in preparation for taking some new operational path on its next execution cycle.

Example 2-31. Set Thread Gate and Thread Gate Preset

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

TS_SetThreadGate (SELFTHREAD, (GATEKEY)5);

... continue
```

See Also

`XX_ClearThreadGateBits`, page 23
`XX_DecrThreadGate`, page 26
`XX_GetThreadGate`, page 50
`XX_IncrThreadGate`, page 60
`XX_ORThreadGateBits`, page 66

XX_SetThreadGatePreset

Set a new thread gate preset value.

Zones

2 TS_SetThreadGatePreset
3 KS_SetThreadGatePreset

Synopsis

```
void XX_SetThreadGatePreset (THREAD thread,
   GATEKEY gatekey)
```

Inputs

thread	The handle of the thread containing the thread gate being set. The thread handle can be that of the Current Thread, which is assumed if the thread handle is zero (0).
gatekey	The new value to store in the thread gate preset of the specified *thread*.

Description

The XX_SetThreadGatePreset kernel service moves the content of *gatekey* into the specified *thread*'s thread gate preset. The new thread gate preset value does not have any effect until the next time *thread* is scheduled as the result of a call to the XX_ClearThreadGateBits, XX_DecrThreadGate, or TS_GetThreadGateLoadPreset services.

Output

This service does not returns a value.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_THREAD if the specified thread ID is not valid.
- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.

Example

In Example 2-32 on page 81, the Current Thread changes its thread gate preset value in preparation for taking some new operational path on its next execution cycle.

Example 2-32. Set Thread Gate Preset

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

TS_SetThreadGatePreset (SELFTHREAD, (GATEKEY)5);

... continue
```

See Also

XX_GetThreadGatePreset, page 54

INIT_ThreadClassProp

Initialize the Thread object class properties.

Synopsis

```
KSRC INIT_ThreadClassProp
   (const KCLASSPROP *pclassprop)
```

Input

pclassprop A pointer to a Thread object class properties structure.

Description

During the **RTXC** Kernel initialization procedure (usually performed in Zone 3), you must define the kernel objects needed by the **RTXC** Kernel to perform the application. The `INIT_ThreadClassProp` kernel service allocates space for the Thread object class in system RAM. The amount of RAM to allocate, and all other properties of the class, should be specified in the structure pointed to by *pclassprop*.

The `KCLASSPROP` structure has the following organization:

```
typedef struct
{
  KATTR attributes;
  KOBJECT n_statics;        /* number of static objects */
  KOBJECT n_dynamics;       /* number of dynamic objects */
  short objsize;            /* used for calculating offsets */
  short totalsize;          /* used to alloc object array RAM */
  ksize_t namelen;          /* length of the name string */
  const char *pstaticnames;
} KCLASSPROP;
```

The *attributes* element of the Thread property structure supports the attributes and corresponding masks listed in Table 2-1 on page 45.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_NO_RAM` if the initialization fails because there is insufficient system RAM available.

Example

During system initialization, the startup code must initialize the Thread object class before using any kernel service for that class. In Example 2-33 on page 83, the system generation process produced a

KCLASSPROP structure containing the information about the kernel class necessary for its initialization. That structure is referenced externally to the code.

Example 2-33. Initialize Thread Object Class

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

extern const SYSPROP     sysprop;
extern const KCLASSPROP  threadclassprop;
extern const THREADPROP  threadprop[];

KSRC rtxcinit (void)
{
   KOBJECT objnum;
   KSRC ksrc;

   /* initialize the RTXCdsp workspace and class/object data */
   if ( (ksrc = INIT_SysProp (&sysprop)) != RC_GOOD)
      return ksrc;

   /* initialize the THREAD class/object data */
   if ((ksrc = INIT_ThreadClassProp (&threadclassprop)) != RC_GOOD)
      return ksrc;

   for (objnum = 1; objnum <= threadclassprop.n_statics; objnum++)
   {
      KS_DefThreadProp (objnum, &threadprop[objnum]);
   }

  ... continue with system initialization
```

See Also

KS_GetThreadClassProp, page 44

XX_UnscheduleThread

Unschedule execution of a thread.

Zones

`IS_ScheduleThread`
`TS_ScheduleThread`

Synopsis

```
void XX_UnscheduleThread (THREAD thread)
```

Input

thread — The handle of the thread to unschedule. A thread value of zero (0) is legal, allowing the Current Thread to unschedule itself.

Description

The `XX_UnscheduleThread` kernel service unschedules the execution of the specified thread.

Regardless of the zone from which the code entity calls this service, the specified *thread* is unscheduled. It does not receive control of the processor until such time as it is again scheduled and the **RTXC/ss** Scheduler grants it control of the processor.

Note: This service has no effect on a thread that is already executing.

Output

This service returns no value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 2-34 on page 85, the Current Thread unschedules the execution of the thread specified in `THREADA`.

Example 2-34. Unschedule Thread Execution

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */

TS_UnscheduleThread (THREADA);

... continue
```

See Also

XX_ScheduleThread

CHAPTER 3

Exception Services

In This Chapter

We describe the Exception kernel services in detail. The Exception services perform a limited number of special operations while CPU control is in an interrupt service routine.

KS_CloseException

End the use of a dynamic exception.

Synopsis

```
KSRC KS_CloseException (EXCPTN xeptn)
```

Input

xeptn	The handle of the exception to be closed.

Description

The `KS_CloseException` kernel service ends the Current Task's use of the dynamic exception specified in *xeptn*. When closing the exception, the kernel detaches the caller's use of it. If the caller is the last user of the exception, the service releases the exception to the free pool of dynamic exceptions for reuse. If there is at least one other task still using the exception, the kernel does not release the exception to the free pool but the service completes successfully.

Note: To use this service, you must enable the *Dynamics* attribute of the Exception class during system generation.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service is successful.
- `RC_STATIC_OBJECT` if the specified exception is not dynamic.
- `RC_OBJECT_NOT_INUSE` if the specified exception does not correspond to an active dynamic exception.
- `RC_OBJECT_INUSE` if the Current Task's use of the specified exception is closed but the exception remains open for use by other tasks.

Note: `RC_OBJECT_INUSE` does not necessarily indicate an error condition. The calling task must interpret its meaning.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_EXCEPTION` if the specified exception ID is not valid.

Example

Example 3-1 waits on a signal from another task indicating that it is time to close a dynamic exception. The handle of the dynamic exception is specified in *dynxeptn*. When the signal is received, the Current Task closes the associated exception.

Example 3-1. Close Exception

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

EXCPTN dynxeptn;
SEMA dynsema;

KS_TestSemaW (dynsema); /* wait for signal */

/* then close exception */
if (KS_CloseException (dynxeptn) != RC_GOOD)
{
   ...something is wrong. Deal with it here;
}
```

See Also

`KS_OpenException`, page 104
`XX_DefExceptionProp`, page 92

KS_DefExceptionName

Define the name of a previously opened exception.

Synopsis

```
KSRC KS_DefExceptionName (EXCPTN xeptn,
   const char *pname)
```

Inputs

xeptn	The handle of the exception being defined.
pname	A pointer to a null-terminated name string.

Description

The `KS_DefExceptionName` kernel service names or renames the dynamic exception specified in *xeptn*. The service uses the null-terminated string pointed to by *pname* for the exception's new name.

Static exceptions cannot be named or renamed under program control.

Note: To use this service, you must enable the *Dynamics* attribute of the Exception class during system generation.

This service does not check for duplicate exception names.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_STATIC_OBJECT` if the exception being named is static.
- `RC_OBJECT_NOT_FOUND` if the *Dynamics* attribute of the exception class is not enabled.
- `RC_OBJECT_NOT_INUSE` if the specified exception does not correspond to an active dynamic exception.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_EXCEPTION` if the specified exception ID is not valid.

Example

Example 3-2 assigns the name `NewExeptn` to the previously opened exception specified in the *dynxeptn* variable so other users may reference it by name.

Example 3-2. Define Exception Name

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

EXCPTN dynxeptn;

if (KS_DefExceptionName (dynxeptn, "NewExeptn") != RC_GOOD)
{
   ... Probably is a static exception. Deal with it here
}

... naming operation was successful. Continue
```

See Also

`KS_OpenException`, page 104
`KS_GetExceptionName`, page 98
`KS_LookupException`, page 102
`KS_UseException`, page 107

XX_DefExceptionProp

Define the properties of an exception.

Zones

2 TS_DefExceptionProp
3 KS_DefExceptionProp

Synopsis

```
void XX_DefExceptionProp (EXCPTN xeptn,
   const EXCPTNPROP *pxeptnprop)
```

Inputs

xeptn	The handle of the exception being defined.
pxeptnprop	A pointer to an exception properties structure.

Description

The `XX_DefExceptionProp` kernel service defines the properties of the exception specified in *xeptn* using the values contained in the `EXCPTNPROP` structure pointed to by *pxeptnprop*.

Example 3-3 shows the organization of the `EXCPTNPROP` structure.

Example 3-3. Exception Properties Structure

```
typedef struct
{
  KATTR attributes;       /* attributes */
  unsigned char level;    /* processor interrupt level (IPL) */
  unsigned char vector;   /* vector number*/
  void (*handler)(void);  /* ISR Prologue address */
} EXCPTNPROP;
```

Output

This service does not return a value.

Error

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_EXCEPTION` if the specified exception ID is not valid.
- `FE_NULL_EXCEPTIONHANDLER` if the specified exception handler address is null.

Example

Example 3-4 on page 93 allocates a dynamic exception with the following properties: The interrupt level is 5, the vector for the interrupt is 64. The exception function is `Handler`.

Example 3-4. Define Exception Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */
#include "ktask.h"        /* TASK5 */

EXCPTN dynxeptn;
static EXCPTNPROP xeptnprop;
extern void Handler (void);

if (KS_OpenException ((char *)0, &dynxeptn) != RC_GOOD)
{
   ... something wrong. Deal with it here
}

/* define the properties of the dynamic exception */
xeptnprop.attributes = 0;
xeptnprop.level = 5;
xeptnprop.vector = 64;
xeptnprop.handler = Handler;
KS_DefExceptionProp (dynxeptn, &xeptnprop);

   ...continue processing
```

See Also

`XX_GetExceptionProp`, page 100
`INIT_ExceptionClassProp`, page 94
`KS_OpenException`, page 104

INIT_ExceptionClassProp

Initialize the Exception object class properties.

Synopsis

```
KSRC INIT_ExceptionClassProp
   (const KCLASSPROP *pclassprop)
```

Inputs

pclassprop	A pointer to an exception object class properties structure.

Description

During the **RTXC** Kernel initialization procedure, you must define the kernel objects needed by the kernel to perform the application. The `INIT_ExceptionClassProp` kernel service allocates space for the exception object class in system RAM. The amount of RAM to allocate, and all other properties of the class, are specified in the `KCLASSPROP` structure pointed to by *pclassprop*.

Example 2-13 on page 44 shows the organization of the `KCLASSPROP` structure.

The *attributes* element of the Exception `KCLASSPROP` structure supports the class property attributes and masks listed in Table 3-1 on page 96.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_NO_RAM` if the initialization fails because there is insufficient system RAM available.

Example

During system initialization, the startup code must initialize the Exception object class before using any kernel service for that class. The system generation process produces a `KCLASSPROP` structure containing the information about the kernel object necessary for its initialization. Example 3-5 on page 95 references that structure externally to the code module.

Example 3-5. Initialize Exception Object Class

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

extern const SYSPROP sysprop;
extern const KCLASSPROP xeptnclassprop;

KSRC userinit (void)
{
   KSRC ksrc;

   /* initialize the kernel workspace and allocate RAM */
   /* for required classes, etc. */

   if ((ksrc = InitSysProp (&sysprop)) != RC_GOOD)
   {
      putline ("Kernel initialization failure");
      return (ksrc); /* end initialization process */
   }
   /* kernel is initialized */

   /* Need to initialize the necessary kernel object classes */

   /* Initialize the Exception kernel object class */
   if ((ksrc = INIT_ExceptionClassProp (&xeptnclassprop))
       != RC_GOOD)
   {
      putline ("No RAM for Exception init");
      return (ksrc); /* end initialization process */
   }

... Continue with system initialization
}
```

See Also

`INIT_SysProp`, page 284
`KS_GetExceptionClassProp`, page 96

KS_GetExceptionClassProp

Get the Exception object class properties.

Synopsis

```
const KCLASSPROP * KS_GetExceptionClassProp
   (int *pint)
```

Input

pint	A pointer to a variable in which to store the number of available dynamic exceptions.

Description

The `KS_GetExceptionClassProp` kernel service obtains a pointer to the `KCLASSPROP` structure that was used during system initialization by the `INIT_ExceptionClassProp` service to initialize the exception object class properties. If *pint* is not null (`(int *)0`), the service returns the number of available dynamic exceptions in the variable pointed to by *pint*.

Example 2-13 on page 44 shows the organization of the `KCLASSPROP` structure.

The *attributes* element of the Exception `KCLASSPROP` structure supports the class property attributes and corresponding masks listed in Table 3-1.

Table 3-1. Exception Class Attributes and Masks

Attribute	Mask
Static Names	`ATTR_STATIC_NAMES`
Dynamics	`ATTR_DYNAMICS`

Output

If successful, this service returns a pointer to a `KCLASSPROP` structure.

If the Exception class is not initialized, the service returns a null pointer (`(KCLASSPROP *)0`).

Example

Example 3-6 on page 97 accesses to the information contained in the `KCLASSPROP` structure for the Exception object class.

Example 3-6. Read Exception Object Class Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

KCLASSPROP *pxeptnclassprop;
int free_dyn;

/* Get the Exception kernel object class properties */
if ((pxeptnclassprop = KS_GetExceptionClassProp (&free_dyn))
    == (KCLASSPROP *)0)
{
   putline ("Exception Class not initialized");
}
else
{
   ... Exception class information is available for use
       "free_dyn" contains the number of available dynamic exceptions
}
```

See Also

`INIT_ExceptionClassProp`, page 94

KS_GetExceptionName

Get the name of an exception.

Synopsis

`char * KS_GetExceptionName (EXCPTN xeptn)`

Input

xeptn	The handle of the exception being queried.

Description

The `KS_GetExceptionName` kernel service obtains a pointer to the null-terminated string containing the name of the static or dynamic exception specified in *xeptn*.

Output

If the exception has a name, this service returns a pointer to the null-terminated name string.

If the exception has no name, the service returns a null pointer (`(char *)0`).

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_EXCEPTION` if the specified exception ID is not valid.

Example

Example 3-7 reports the name of the dynamic exception specified in *dynxeptn*.

Example 3-7. Read Exception Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

static char buf[128];

EXCPTN dynxeptn;
char *pname;

if ((pname = KS_GetExceptionName (dynxeptn)) == (char *)0)
   sprintf (buf, "Exception %d has no name", dynxeptn);
else
   sprintf (buf, "The name of Exception %d is %s", dynxeptn,
            pname);

putline (buf);
```

See Also

`KS_DefExceptionName`, page 90
`KS_OpenException`, page 104

XX_GetExceptionProp

Get the properties of an exception.

Zones

2 `TS_GetExceptionProp`
3 `KS_GetExceptionProp`

Synopsis

```
void XX_GetExceptionProp (EXCPTN xeptn,
   EXCPTNPROP *pxeptnprop)
```

Inputs

xeptn	The handle of the exception being queried.
pxeptnprop	A pointer to an exception properties structure.

Description

The `XX_GetExceptionProp` kernel service obtains all of the property values of the exception specified in *xeptn* in a single call. The service stores the property values in the `EXCPTNPROP` structure pointed to by *pxeptnprop*.

Example 3-3 on page 92 shows the organization of the `EXCPTNPROP` structure.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_EXCEPTION` if the specified exception ID is not valid.
- `FE_UNINITIALIZED_SEMA` if the specified semaphore has not yet been initialized.

Example

In Example 3-8 on page 101, the Current Task needs to change the interrupt level of the dynamic exception specified in *dynxeptn* to 3 but does not want to change any of the other properties. The task first obtains the current properties, then modifies the *level* element in the `EXCPTNPROP` structure. The task then uses `XX_DefExceptionProp` to redefine the properties of the exception.

Example 3-8. Read Exception Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

EXCPTN dynxeptn;
EXCPTNPROP xeptnprop;

/* get the current exception properties */
KS_GetExceptionProp (dynxeptn, &xeptnprop);

/* modify just the level element */
xeptnprop.level = 3;

/* define the new exception properties */
KS_DefExceptionProp (dynxeptn, &xeptnprop);
```

See Also

`XX_DefExceptionProp`, page 92

KS_LookupException

Look up an exception's name to get its handle.

Synopsis

```
KSRC KS_LookupException (const char *pname,
   EXCPTN *pxeptn)
```

Inputs

pname	A pointer to a null-terminated name string.
pxeptn	A pointer to a variable in which to store the matching exception's handle.

Description

The `KS_LookupException` kernel service obtains the handle of a static or dynamic exception whose name matches the null-terminated string pointed to by *pname*. The lookup process terminates when it finds a match between the specified string and a static or dynamic exception name or when it finds no match. The service stores the matching exception's handle in the variable pointed to by *pxeptn*. The service searches dynamic names, if any, first.

Note: To use this service on static mutexes, you must enable the *Static Names* attribute of the Exception class during system generation.

This service has no effect on the use registration of the specified exception by the Current Task.

The time required to perform this operation varies with the number of exception names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search succeeds. The service stores the matching exception's handle in the variable pointed to by *pxeptn*.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching exception name.

Example

In Example 3-9, the Current Task needs to use the dynamic exception named `Dynxeptn2`. If the exception is found, the task sends its exception handle to the console in a brief message.

Example 3-9. Look Up Exception by Name

```
#include <stdio.h>          /* standard i/o */
#include "rtxcapi.h"        /* RTXC Kernel Services prototypes */

static char buf[128];

EXCPTN dynxeptn;

/* lookup the exception name to see if it exists */
if (KS_LookupException ("Dynxeptn2", &dynxeptn) != RC_GOOD)
{
   putline ("Exception Dynxeptn2 name not found");
}
else  /* Exception exists */
{
   sprintf (buf, "Dynxeptn2 is Exception %d", dynxeptn);
   putline (buf);
}
```

See Also

`KS_DefExceptionName`, page 90
`KS_OpenException`, page 104

KS_OpenException

Allocate and name a dynamic exception.

Synopsis

```
KSRC KS_OpenException (const char *pname,
   EXCPTN *pxeptn)
```

Inputs

pname	A pointer to a null-terminated name string.
pxeptn	A pointer to a variable in which to store the allocated exception's handle.

Description

The `KS_OpenException` kernel service allocates, names, and obtains the handle of a dynamic exception. If a dynamic exception is available and there is no existing exception, static or dynamic, with a name matching the null-terminated string pointed to by *pname*, the service allocates a dynamic exception and applies the name referenced by *pname* to the new exception. The service stores the handle of the new dynamic exception in the variable pointed to by *pxeptn*. The kernel stores only the address of the name internally, which means that the same array cannot be used to build multiple dynamic exception names.

If *pname* is a null pointer (`(char *)0`), the service does not assign a name to the dynamic exception. However, if *pname* points to a null string, the name is legal as long as no other exception is already using a null string as its name.

If the service finds an existing exception with a matching name, it does not open a new exception and returns a value indicating an unsuccessful operation.

Note: To use this service, you must enable the *Dynamics* attribute of the Exception class during system generation.

If *pname* is not null (`(char *)0`), the time required to perform this operation is determined by the number of exception names in use.

If the pointer to the timer name is null, no search of exception names takes place and the time to perform the service is fixed. You can define the exception name at a later time with a call to the `KS_DefExceptionName` service.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_OBJECT_ALREADY_EXISTS` if the name search finds another exception whose name matches the given string.
- `RC_NO_OBJECT_AVAILABLE` if the name search finds no match but all dynamic exceptions are in use.

Example

Example 3-10 allocates a dynamic exception and names it `SCIChnl2xeptn`. If the name is found to be in use or if there are no dynamic exceptions available, the task sends an appropriate message to the console.

Example 3-10. Allocate and Name Exception

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

KSRC ksrc;
EXCPTN dynxeptn;

if ((ksrc = KS_OpenException ("SCIChnl2xeptn", &dynxeptn))
    != RC_GOOD)
{
   if (ksrc == RC_OBJECT_ALREADY_EXISTS)
      putline ("SCIChnl2xeptn name in use");
   else if (ksrc == RC_NO_OBJECT_AVAILABLE)
      putline ("No dynamic exceptions available");
   else
      putline ("Exceptions are not a defined class");
}
else
{
   ... Exception was opened correctly. Okay to use it now
}
```

See Also

`KS_CloseException`, page 88
`KS_LookupException`, page 102
`KS_UseException`, page 107

KS_UseException

Look up a dynamic exception by name and mark it for use.

Synopsis

```
KSRC KS_UseException (const char *pname,
   EXCPTN *pxeptn)
```

Inputs

pname	A pointer to a null-terminated name string.
pxeptn	A pointer to a variable in which to store the allocated exception's handle.

Description

The `KS_UseException` kernel service acquires the handle of a dynamic exception by looking up the null-terminated string pointed to by *pname* in the list of exception names. If there is a match, the service registers the exception for future use by the Current Task and stores the matching exception's handle in the variable pointed to by *pxeptn*. This procedure allows the Current Task to reference the dynamic exception successfully in subsequent kernel service calls.

Note: To use this service, you must enable the *Dynamics* attribute of the Exception class during system generation.

The time required to perform this operation varies with the number of exception names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search is successful. The service also stores the matching exception's handle in the variable pointed to by *pxeptn*.
- `RC_STATIC_OBJECT` if the given name belongs to a static exception.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching exception name.

Example

Example 3-11 locates a dynamic exception named `DynSCIxeptn3`, prepares it for subsequent use, and obtains its exception handle. It then sends a message to the console indicating the handle of the exception if successful or an error message if unsuccessful.

Example 3-11. Read Exception Handle and Register It

```
#include <stdio.h>          /* standard i/o */
#include "rtxcapi.h"        /* RTXC Kernel Services prototypes */

static char buf[128];

EXCPTN dynxeptn;

if (KS_UseException ("DynSCIxeptn3", &dynxeptn) != RC_GOOD)
{
   ... exception is either static or not found
   ... need to handle that here
}
else
{
   /* Exception was found and its handle is in dynxeptn. */
   sprintf (buf, "DynSCIxeptn3 is Exception %d", dynxeptn);
   putline (buf);
}
```

See Also

`XX_DefExceptionProp`, page 92
`KS_DefExceptionName`, page 90
`KS_OpenException`, page 104

CHAPTER 4

Pipe Services

In This Chapter

We describe the Pipe kernel services in detail. The Pipe kernel services move data between a single producer and a single consumer and maintain information about pipe states.

KS_ClosePipe

End the use of a dynamic pipe.

Synopsis

```
KSRC KS_ClosePipe (PIPE pipe)
```

Input

pipe	The handle of the pipe to close.

Description

The `KS_UsePipe` kernel service ends the Current Task's use of the specified dynamic *pipe*. When closing *pipe*, the kernel detaches the caller's use of it. If the caller is the last task associated with *pipe*, the kernel releases *pipe* to the free pool of dynamic pipes for reuse. If there is at least one other task still referencing *pipe*, the kernel does not release the pipe to the free pool but the service completes successfully.

Note: To use this service, you must enable the *Dynamics* attribute of the Pipe class during system generation.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_STATIC_OBJECT` if the specified pipe is not dynamic.
- `RC_OBJECT_NOT_INUSE` if the specified pipe does not correspond to an active dynamic pipe.
- `RC_OBJECT_INUSE` if the Current Task's use of the specified pipe is closed but the pipe remains associated with other tasks.

Note: The `KSRC` value does not necessarily indicate an error condition. The calling task must interpret its meaning.

Error

This service may generate the following fatal error code:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.

Example

In Example 4-1, the Current Task waits on a signal from another task indicating that it is time to close the active dynamic pipe specified in `dynpipe`. When the signal is received, the Current Task closes the associated pipe.

Example 4-1. Close Pipe Upon Receiving Signal

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

PIPE dynpipe;
SEMA dynsema;

KS_TestSema (dynsema);    /* wait for signal */

/* then close the pipe */
if (KS_ClosePipe (dynpipe) != RC_GOOD)
{
   ... something is wrong. Deal with it
}
... continue              /* pipe closed successfully */
```

See Also

`KS_OpenPipe`, page 140
`KS_UsePipe`, page 154

XX_DefPipeAction

Define action to perform following `XX_PutFullPipeBuf` or `XX_PutEmptyPipeBuf` services.

Zones

2 `TS_DefPipeAction`
3 `KS_DefPipeAction`

Synopsis

```
void XX_DefPipeAction (PIPE pipe, PIPEACTION action,
   THREAD thread, PIPECOND cond)
```

Input

pipe The handle of the pipe to be associated with the callback function.

action A code for the action to perform as follows:

- `SCHEDULETHREAD`—Schedule thread at the completion of the operation on *pipe*.
- `DECRTHREADGATE`—Decrement the thread gate value of *thread* upon completing the operation on *pipe*.

thread The handle of the thread on which to perform the end action operation.

cond A value of `PIPECOND` type specifying the action to take according to the completed pipe operation. The valid values are:

- `PUTFULL`—If the pipe operation puts full buffers into the pipe (`XX_PutFullPipeBuf`, `XX_PutFullGetEmptyPipeBuf`, `XX_JamFullPipeBuf`, or `XX_JamFullGetEmptyPipeBuf`).
- `PUTEMPTY`—For a pipe operation that puts empty buffers into a pipe (`XX_PutEmptyPipeBuf` or `XX_PutEmptyGetFullPipeBuf`).

Description

The `XX_DefPipeAction` kernel service defines the action to take following a service that puts a buffer (empty or full) into the specified *pipe*. The `XX_PutFullPipeBuf` or `XX_PutEmptyPipeBuf` services perform the specified end action operation, if defined, on the specified *thread* when the service completes. If the pipe service to put an empty or full buffer into the pipe is called from an ISR, the end action operation performs a Zone 1 service, `IS_ScheduleThread` or `IS_DecrThreadGate`, corresponding to the action code `SCHEDULETHREAD` or `DECRTHREADGATE`, respectively. If the pipe service to put an empty or full buffer into the pipe is called from a thread, the end action operation performs a Zone 2 service, `TS_ScheduleThread` or `TS_DecrThreadGate`, corresponding to the action code `SCHEDULETHREAD` or `DECRTHREADGATE`, respectively.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.
- `FE_INVALID_PIPECOND` if the specified pipe condition value is not either `PUTEMPTY` or `PUTFULL`.
- `FE_INVALID_PIPEACTION` if the specified pipe action value is not one of the four possible actions.
- `FE_ILLEGAL_THREAD` if the specified thread ID is not valid.
- `FE_UNINITIALIZED_THREAD` if the specified thread has not yet been initialized.

Example

In Example 4-2 on page 114, the gate for the thread specified in `THREADA` needs to be decremented every time a full buffer is put into the pipe specified in `PIPEA`. When the value of the `THREADA` thread gate reaches zero, `THREADA` is scheduled to execute. The Current Thread defines the action that is to take place on `PIPEA`.

Example 4-2. Define Pipe End Action Operation

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */
#include "kpipe.h"        /* PIPEA */

/* define pipe action on PIPEA to decrement gate */
TS_DefPipeAction (PIPEA, DECRTHREADGATE, THREADA, PUTFULL);

/* define the thread gate and thread gate preset for THREADA */
TS_DefThreadGate (THREADA, (GATEKEY)2);
... continue

/*  ISR device handler function that puts a full buffer into PIPEA */
void devhandler (void)
{
   ...first part of ISR

/* The following action puts the buffer into PIPEA and causes */
/* THREADA thread gate to be decremented because of the defined */
/* pipe action.  */
/* THREADA is scheduled if the decrement causes the thread */
/* gate to become zero (0) */

/* In the following statement, bufptr points to the full buffer */
/* and bufsize contains the size of the buffer as filled */

   IS_PutFullPipeBuf (PIPEA, bufptr, bufsize);

   ...more device handler
   return
}
```

XX_DefPipeProp

Define the properties of a pipe.

Zones

2 `TS_DefPipeProp`
3 `KS_DefPipeProp`

Synopsis

```
void XX_DefPipeProp (PIPE pipe, PIPEPROP ppipeprop)
```

Inputs

pipe	The handle of the pipe being defined.
ppipeprop	A pointer to a pipe properties structure.

Description

The `XX_DefPipeProp` kernel service defines the properties of the specified *pipe* using the values contained in the `PIPEPROP` structure pointed to by *ppipeprop*. You may use this service on static or dynamically allocated pipes. It is typically used to define a static pipe during system startup or a dynamic pipe during runtime which has been previously allocated with the `KS_OpenPipe` kernel service.

Example 4-3 shows the organization of the `PIPEPROP` structure.

Example 4-3. Pipe Properties Structure

```
typedef struct _pipeprop
{
   KATTR attributes;  /* pipe attributes */
   KCOUNT numbufs;    /* number of buffers */
   ksize_t bufsize;   /* maximum usable buffer size */
   void * buf;        /* pipe buffer base address */
   void ** fullbase;  /* base address of full buffers pointers */
   void ** freebase;  /* base address of free buffers pointers */
   int * sizebase;
} PIPEPROP;
```

When using this service with static pipes defined as part of the system configuration process, the properties are fully specified. In the case of static pipes, the pipe's buffers are generally allocated in a contiguous manner.

When using this service to define the properties of a dynamic pipe, it is possible to define the properties less than fully and still be able to make limited references to the pipe. It is possible to define the pipe buffer base address, `buf`, as a null pointer (`(void *)0`) and then allocate space for each buffer in the pipe, defining each by using the `XX_PutEmptyPipeBuf` kernel service until all buffers are defined. With this technique, the buffers are not necessarily allocated contiguously.

Output

This service does not return a value.

Error

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_ZERO_PIPENUMBUF` if the number of buffers in the specified pipe is zero.
- `FE_ZERO_PIPEBUFSIZE` if the buffer size in the specified pipe is zero.
- `FE_NULL_PIPEFULLBASE` if the specified Pipe full base address is null.
- `FE_NULL_PIPEFREEBASE` if the specified Pipe free base address is null.
- `FE_NULL_PIPESIZEBASE` if the specified Pipe base size address is null.

Examples

During system initialization, the startup routine must create and initialize the Pipe object class and define the properties of all the static Pipes before information can be passed through Pipes, as illustrated in Example 4-4 on page 117.

Example 4-4. Define Pipe Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

extern const KCLASSPROP pipeclassprop;
extern const PIPEPROP pipeprop[];

KSRC ksrc;
int objnum;

   /* initialize the PIPE class/object data */
   if ((ksrc = INIT_PipeClassProp (&pipeclassprop)) != RC_GOOD)
      return ksrc;

   for (objnum = 1; objnum <= pipeclassprop.n_statics; objnum++)
   {
      TS_DefPipeProp (objnum, &pipeprop[objnum]);
   }

... continue
```

See Also

`XX_GetPipeProp`, page 130

KS_DefPipeName

Define the name of a previously opened dynamic pipe.

Synopsis

```
KSRC KS_DefPipeName (PIPE pipe, const char *pname)
```

Inputs

pipe	The handle of the pipe being defined.
pname	A pointer to a null-terminated name string.

Description

The `KS_GetPipeName` kernel service names or renames the specified dynamic *pipe*. The service uses the null-terminated string pointed to by *pname* for the new name. The kernel only stores *pname* internally, which means the same array cannot be used to build multiple dynamic pipe names. Static pipes cannot be named or renamed under program control.

Note: To use this service, you must enable the *Dynamics* attribute of the Pipe class during system generation.

This service does not check for duplicate pipe names.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_STATIC_OBJECT` if the pipe being named is static.
- `RC_OBJECT_NOT_FOUND` if the *Dynamics* attribute of the Pipe class is not enabled.
- `RC_OBJECT_NOT_INUSE` if the dynamic pipe being named is still in the free pool of dynamic pipes.

Error

This service may generate the following fatal error code:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.

Example

Example 4-5 assigns the name `NewPipe` to the pipe specified in `dynpipe` so other users may reference it by name.

Example 4-5. Define Dynamic Pipe Name

```
#include "rtxcapi.h"       /* RTXC Kernel Service prototypes */

PIPE dynpipe;

if (KS_DefPipeName (dynpipe, "NewPipe") != RC_GOOD)
{
   ... Probably is a static pipe. Deal with it here.
}

... else the naming operation was successful. Continue
```

See Also

`KS_GetPipeName`, page 128
`KS_LookupPipe`, page 138
`KS_UsePipe`, page 154

XX_GetEmptyPipeBuf

Get an empty buffer from a specified pipe.

Zones

1 `IS_GetEmptyPipeBuf`
2 `TS_GetEmptyPipeBuf`
3 `KS_GetEmptyPipeBuf`

Synopsis

```
void * XX_GetEmptyPipeBuf (PIPE pipe)
```

Input

pipe	The handle of the pipe from which to get an empty buffer.

Description

The `KS_UsePipe` kernel service removes the next available empty buffer from the specified *pipe* and returns a pointer to the empty buffer to the caller. If there is no buffer available, the service returns a null pointer (`(void *)0`).

Output

This service returns a pointer to the empty buffer if a buffer is available. If no buffer is available, the service returns a null pointer (`(void *)0`).

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.

Example

In Example 4-6 on page 121, the Current Thread gets an empty buffer from the pipe specified in `PIPEA` and, if a valid buffer pointer is returned, performs some operation on the buffer.

Example 4-6. Get Empty Buffer from Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

void threadxyz ((void *)0, (void *)0)
{
   char *pipebuf;
   int pipebufsize;

   /* get size of pipe's buffers */
   pipebufsize = TS_GetPipeBufSize (PIPEA);

   /* get empty pipe buffer and test for success */
   if ((pipebuf = TS_GetEmptyPipeBuf (PIPEA)) == (char *)0);

   /* test if emtpy buffer is available */
   if (pipebuf == (char *)0)
   {
      ... no empty buffers, deal with it here
   }
   else
   {
      ... perform operation on fill the empty buffer
   }
   ...when buffer is full, put it into the pipe
   TS_PutFullPipeBuf (PIPEA, (void *)pipebuf, pipebufsize);
... continue
}
```

See Also

`KS_UsePipe`, page 154
`XX_JamFullGetEmptyPipeBuf`, page 132
`XX_JamFullPipeBuf`, page 136
`XX_PutEmptyGetFullPipeBuf`, page 144
`XX_PutEmptyPipeBuf`, page 147
`XX_PutFullGetEmptyPipeBuf`, page 149
`XX_PutFullPipeBuf`, page 152

XX_GetFullPipeBuf

Get a full buffer from a specified pipe.

Zones

1 `IS_GetFullPipeBuf`
2 `TS_GetFullPipeBuf`
3 `KS_GetFullPipeBuf`

Synopsis

```
void * XX_GetFullPipeBuf (PIPE pipe, int *pbufsize)
```

Inputs

pipe	The handle of the pipe from which to get a full buffer.
pbufsize	A pointer to a variable that will, upon completion of the service, contain the actual size of the full buffer whose pointer is being returned as the value of the service.

Description

The `XX_GetFullPipeBuf` kernel service removes the next available full buffer from the specified *pipe* and returns a pointer to the full buffer to the caller. If there is no buffer available, the service returns a null pointer (`(void *)0`) and the variable pointed to by *pbufsize* is set to 0.

Output

This service returns a pointer to the full buffer if a buffer is available. If no buffer is available, the service returns a null pointer (`(void *)0`) and the variable pointed to by *pbufsize* is set to 0.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.
- `FE_NULL_PIPEPBUFSIZE` if the pointer to the buffer size is null.

Example

In Example 4-7 on page 123, the Current Thread gets a full buffer from the pipe specified in `PIPEA` and, if a valid buffer pointer is returned, performs some operation on the buffer.

Example 4-7. Get Full Buffer from Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

void threadxyz ((void *)0, (void *)0)
{
   char *pipebuf;
   int bsize, i;

   /* get full pipe buffer and its size and test if OK */
   if ((pipebuf = TS_GetFullPipeBuf (PIPEA, &bsize)) == (char *)0);

   /* test if full buffer available */
   if (pipebuf == (char *)0)
   {
      ... no full buffers, deal with it here
   }
   else
   {
      for (i=0; i<=bsize; i++)
      {
      ... perform operation on full buffer
      }
      /* when buffer is empty, return it to the pipe */
      TS_PutEmptyPipeBuf (PIPEA, pipebuf);
   }

... continue
}
```

See Also

`XX_DefPipeProp`, page 115
`XX_JamFullGetEmptyPipeBuf`, page 132
`XX_JamFullPipeBuf`, page 136
`XX_PutEmptyGetFullPipeBuf`, page 144
`XX_PutEmptyPipeBuf`, page 147
`XX_PutFullGetEmptyPipeBuf`, page 149
`XX_PutFullPipeBuf`, page 152

XX_GetPipeBufSize

Get the maximum usable size of buffers in the specified pipe.

Zones

1 `IS_GetPipeBufSize`
2 `TS_GetPipeBufSize`
3 `KS_GetPipeBufSize`

Synopsis

```
int XX_GetPipeBufSize (PIPE pipe)
```

Input

pipe	The handle of the pipe being queried.

Description

The `XX_GetPipeBufSize` kernel service allows the caller to obtain the maximum usable size of buffers in the specified *pipe*.

Warning: It is possible that a pipe may contain buffers of unequal sizes. It is the responsibility of the programmer to ensure that all buffers used in a given pipe have sufficient RAM to meet or exceed the maximum useful buffer size specified for the pipe. Failure to do so may lead to undesirable or unpredictable results.

Output

This service returns an `int` type value containing the buffer size for *pipe*.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.

Example

In Example 4-8 on page 125, the Current Thread reads the buffer size and fills an empty buffer with data. It then puts the full buffer into the pipe specified in `PIPEA`.

Example 4-8. Read Pipe Buffer Size

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

int buffersize, i;
char *pipebuf;

/* get pipe buffer size */
buffersize = TS_GetPipeBufSize (PIPEA);

/* get empty pipe buffer */
pipebuf = TS_GetEmptyPipeBuf (PIPEA);

/* fill buffer with data */
for (i = 0; i <= buffersize; i++)
{
   ... put  entry into pipebuf
}

/* put full buffer into Pipe */
TS_PutFullPipeBuff (PIPEA, pipebuf, buffersize);

... continue
```

See Also

`XX_DefPipeProp`, page 115

KS_GetPipeClassProp

Get the Pipe class properties.

Synopsis

```
const KCLASSPROP * KS_GetPipeClassProp (int *pint)
```

Input

pint	A pointer to an integer variable in which to store the current number of unused dynamic pipes.

Description

The `KS_GetPipeClassProp` kernel service obtains a pointer to the `KCLASSPROP` structure that was used during system initialization by the `INIT_PipeClassProp` service to initialize the Pipe object class properties.

If the *pint* pointer contains a non-zero address, the current number of unused dynamic pipes is stored in the indicated address. If *pint* contains a null pointer (`(int *)0`), the service ignores the parameter. If the Pipe object class properties do not include the *Dynamics* attribute, the service stores a value of zero (0) at the address contained in *pint*.

The `KCLASSPROP` structure has the following organization:

```
typedef struct
{
   KATTR attributes;
   KOBJECT n_statics;        /* number of static objects */
   KOBJECT n_dynamics;       /* number of dynamic objects */
   short objsize;            /* used for calculating offsets */
   short totalsize;          /* used to alloc object array RAM */
   ksize_t namelen;          /* length of the name string */
   const char *pstaticnames;
} KCLASSPROP;
```

The attributes element of the Pipe property structure supports the class property attributes and corresponding masks listed in Table 4-1 on page 127.

Table 4-1. Pipe Class Attributes and Masks

Attribute	Mask
Static Names	ATTR_STATIC_NAMES
Dynamics	ATTR_DYNAMICS

Output

If successful, this service returns a pointer to a KCLASSPROP structure.

If the Pipe class is not initialized, the service returns a null pointer ((KCLASSPROP *)0).

If *pint* is not null ((int *)0), the service returns the number of available dynamic pipes in the variable pointed to by *pint*.

Example

In Example 4-9, the Current Pipe needs access to the information contained in the KCLASSPROP structure for the Pipe object class.

Example 4-9. Read Pipe Object Class Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KCLASSPROP *ppipeclassprop;
int free_dyn;

/* Get the pipe kernel object class properties */
if ((ppipeclassprop = KS_GetPipeClassProp (&free_dyn))
    == (KCLASSPROP *)0)
{
   putline ("Pipe Class not initialized");
}
else
{
   ... pipe object class properties are available for use
       "free_dyn" contains the number of available dynamic pipes
}
```

See Also

INIT_PipeClassProp, page 142

KS_GetPipeName

Get the pipe's name.

Synopsis

```
char * KS_GetPipeName (PIPE pipe)
```

Input

pipe	The handle of the pipe being queried.

Description

The `KS_GetPipeName` kernel service obtains a pointer to the null-terminated string containing the name of the specified *pipe*. The pipe may be static or dynamic.

Note: To use this service on static pipes, you must enable the *Static Names* attribute of the Pipe class during system generation.

Output

If *pipe* has a name, this service returns a pointer to the null-terminated name string.

If *pipe* has no name, the service returns a null pointer (`(char *)0`).

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.

Example

In Example 4-10 on page 129, the Current Task needs to report the name of the dynamic pipe specified in `dynpipe`.

Example 4-10. Read Pipe Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

static char buf[128];
PIPE dynpipe;
char *pname;

if ((pname = KS_GetPipeName (dynpipe)) == (char *)0)
   sprintf (buf, "Pipe %d has no name", dynpipe);
else
   sprintf (buf, "Pipe %d name is %s", dynpipe, pname);

putline (buf);
```

See Also

`KS_DefPipeName`, page 115
`KS_LookupPipe`, page 138
`XX_DefPipeProp`, page 115

XX_GetPipeProp

Get the pipe's properties.

Zones

2 TS_GetPipeProp
3 KS_GetPipeProp

Synopsis

```
void XX_GetPipeProp (PIPE pipe, PIPEPROP *ppipeprop)
```

Inputs

pipe	The handle of the pipe being queried.
ppipeprop	The pointer to the pipe property structure in which to store the properties of the specified *pipe*.

Description

The XX_GetPipeProp kernel service obtains all of the property values of the specified *pipe* in a single call. The *pipe* input argument may specify a static or a dynamic pipe. The service stores the property values in the PIPEPROP structure pointed to by *ppipeprop* and returns to the caller.

Example 4-3 on page 115 shows the organization of the PIPEPROP structure.

Output

This service returns *pipe*'s properties in the property structure pointed to by *ppipeprop*.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_PIPE if the specified pipe ID is not valid.
- FE_UNINITIALIZED_PIPE if the specified pipe has not yet been initialized.

Example

In Example 4-11 on page 131, the Current Thread reads the properties of the pipe specified in PIPEA, changes some value in the property structure, then redefines PIPEA with the new properties.

Example 4-11. Read Pipe Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

PIPEPROP  pipeprop;

/* get current Pipe Properties */
TS_GetPipeProp (PIPEA, &pipeprop);

... make changes to the properties

/* define the new Pipe properties */
TS_DefPipeProp (PIPEA, &pipeprop);

... continue
```

See Also

`XX_DefPipeProp`, page 115

XX_JamFullGetEmptyPipeBuf

Put a full buffer at the front of a pipe and then get an empty buffer from the same pipe.

Zones

`IS_JamFullGetEmptyPipeBuf`
`TS_JamFullGetEmptyPipeBuf`
`KS_JamFullGetEmptyPipeBuf`

Synopsis

```
void * XX_JamFullGetEmptyPipeBuf (PIPE pipe,
   void * pbuf, int bufsize, KSRC * pksrc)
```

Inputs

pipe	The handle of the pipe to use.
pbuf	The pointer to the full buffer to be put at the front of the specified *pipe*.
bufsize	The actual size of the buffer as filled. This number must be less than or equal to the maximum usable buffer size for the specified *pipe*.
pksrc	A pointer to `KSRC` type return code.

Description

The `XX_JamFullGetEmptyPipeBuf` kernel service allows the specified *pipe*'s producer to put a full buffer into *pipe* at its head rather than at the tail as is the normal case. At the same time, the service gets the next available empty buffer from the same pipe and returns the pointer to the empty buffer to the caller.

It is necessary for the producer to state the size of the buffer as filled so that *pipe*'s consumer knows how much data there is to process. The size of the filled buffer must be less than or equal to the maximum usable size of the buffers for *pipe*.

It is permissible to define *pbuf* as a null pointer (`(void *)0`) to indicate there is no full buffer to put into the pipe. If *pbuf* is null, the service ignores it and operates identically to the `XX_GetEmptyPipeBuf` service, returning the pointer to the next available empty buffer. This technique may be useful when employing a loop in a producer that uses the combination pipe operation. The first time through the loop, there is no full buffer but the service allocates an empty buffer allowing the producer to begin operation.

If *pbuf* is a null pointer, the service ignores the value of *bufsize*. Ideally, in this situation, *bufsize* would contain a value of zero (0).

Output

This service returns the pointer to the next available empty buffer in *pipe* if one is available. If the pipe contains no available empty buffer, the service returns a null pointer `((void *)0)`.

The service also returns a `KSRC` type value through the *pksrc* pointer indicating how the service performed. The possible values are:

- `RC_GOOD` if the service was successful.
- `RC_PIPE_FULL` if the specified pipe does not have room for another full buffer. The service may return a valid empty buffer address even though this `KSRC` value is passed back.
- `RC_PIPE_EMPTY` if the specified pipe does not have an available empty buffer. The service may return this code after successfully putting the full buffer into the pipe but not finding an available empty buffer. This code is redundant because the service would also return the null pointer for the empty buffer. It is provided for completeness.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.

Example

In Example 4-12 on page 134, the `threadA` producer thread has environment arguments in the `myenvargs` structure and the elements of the structure have been previously defined. The `buffer` element represents the address of the next buffer to fill and is initialized with a pointer to an empty buffer in `PIPEA`. The `maxbufsize` variable contains the maximum useful size of a buffer in `PIPEA`. When `threadA` executes, it receives the pointer to its environment arguments and uses the elements therein to preserve variables it needs to maintain between execution cycles, principally the `buffer` variable. When in operation, it fills the empty buffer in a loop until the buffer reaches the maximum useful size. The thread then jams the full buffer to the front of the pipe, simultaneously getting the next available empty buffer in the pipe. The empty buffer is stored in the environment argument `buffer` element to get ready for the next execution cycle of `threadA`. The example assumes that the `KSRC` value returned by the service is always `RC_GOOD`.

Example 4-12. Perform Fast Buffer Exchange at Front of Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

/* environment argument structure for threadA */
struct myenvargs
{
   char *buffer;
   int maxbufsize;
}

void threadA ((void *)0, (struct myenvargs *)myargs)
{
   int bufsize;
   KSRC ksrc;

   for (bufsize=0; bufsize <= myargs->maxbufsize; bufsize++)
   {
   ...fill the buffer
   }

   /* jam full buffer in front of pipe and get next empty buffer */
   myargs->buffer = TS_JamFullGetEmptyPipeBuf (PIPEA,
                      myargs->buffer, bufsize, &ksrc);
}
```

See Also

`XX_GetEmptyPipeBuf`, page 120
`XX_GetFullPipeBuf`, page 122
`XX_JamFullPipeBuf`, page 136
`XX_PutEmptyGetFullPipeBuf`, page 144
`XX_PutEmptyPipeBuf`, page 147
`XX_PutFullGetEmptyPipeBuf`, page 149
`XX_PutFullPipeBuf`, page 152

XX_JamFullPipeBuf

Put a full buffer at the front of a pipe.

Zones

1 IS_JamFullPipeBuf
2 TS_JamFullPipeBuf
3 KS_JamFullPipeBuf

Synopsis

```
KSRC XX_JamFullPipeBuf (PIPE pipe, void * pbuf,
   int bufsize)
```

Inputs

pipe	The handle of the pipe to use.
pbuf	The pointer to the full buffer to be put at the front of the specified *pipe*.
bufsize	The actual size of the buffer as filled. This number must be less than or equal to the maximum usable buffer size for the specified *pipe*.

Description

The `XX_JamFullPipeBuf` kernel service allows the specified *pipe*'s producer to put a full buffer into *pipe* at its head rather than at the tail as is the normal case.

It is necessary for the producer to state the size of the buffer as filled so that *pipe*'s consumer knows how much data there is to process. The size of the filled buffer must be less than or equal to the maximum usable size of the buffers for *pipe*.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_PIPE_FULL` if the specified pipe does not have room for another full buffer.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.

- FE_NULL_PIPEBUFFER if the specified Pipe buffer address is null.
- FE_ZERO_PIPEBUFSIZE if the buffer size in the specified pipe is zero.

Example

In Example 4-13, the Current Thread fills a buffer with data and jams this buffer in front of the pipe specified in PIPEA.

Example 4-13. Put Full Buffer at Front of Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

int bufsize;
char *fullbuf;

... fill the buffer, fullbuf

/* jam full buffer in front of pipe */
TS_JamFullPipeBuf (PIPEA, fullbuf, bufsize);

... continue
```

See Also

XX_DefPipeAction, page 112
XX_DefPipeProp, page 115
XX_JamFullGetEmptyPipeBuf, page 132
XX_PutEmptyGetFullPipeBuf, page 144
XX_PutEmptyPipeBuf, page 147
XX_PutFullGetEmptyPipeBuf, page 149
XX_PutFullPipeBuf, page 152

KS_LookupPipe

Look up a pipe by name to get its handle.

Synopsis

```
KSRC KS_LookupPipe (const char *pname, PIPE *ppipe)
```

Inputs

pname	A pointer to the null-terminated name string for the pipe.
ppipe	A pointer to a variable in which to store the matching handle, if found.

Description

The `KS_LookupPipe` service obtains the handle of a static or dynamic pipe whose name matches the null-terminated string pointed to by *pname*. The lookup process terminates when it finds a match between the specified string and a static or dynamic pipe name or when it finds no match. The service searches dynamic names, if any, first. If a match is found, the service stores the matching pipe's handle in the variable pointed to by *ppipe*.

Note: To use this service on static pipes, you must enable the *Static Names* attribute of the Pipe class during system generation.

This service has no effect on the registration of the specified pipe by the Current Task.

The time required to perform this operation varies with the number of pipe names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search succeeds. The service stores the handle of the pipe in the variable pointed to by *ppipe*.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching pipe name.

Example

In Example 4-14, the Current Task needs to use the dynamic pipe specified in `DynPipe2`. If the pipe name is found, the example outputs the pipe handle to the console in a brief message.

Example 4-14. Look Up Pipe by Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

PIPE dynpipe;
static char buf[128];

/* lookup the pipe name to see if it exists */
if (KS_LookupPipe ("DynPipe2", &dynpipe) != RC_GOOD)
{
   putline ("Pipe DynPipe2 name not found");
}
else  /* pipe exists */
{
   sprintf (buf, "DynPipe2 is pipe %d", dynpipe);
   putline (buf);
}
```

See Also

`KS_DefPipeName`, page 118
`KS_GetPipeName`, page 128

KS_OpenPipe

Allocate and name a dynamic pipe.

Synopsis

```
KSRC KS_OpenPipe (const char *pname, PIPE *ppipe)
```

Inputs

pname	A pointer to the null-terminated name string for the pipe.
ppipe	A pointer to a variable in which to store the handle of the allocated pipe.

Description

The `KS_OpenPipe` kernel service allocates, names, and obtains the handle of a dynamic pipe. If a dynamic pipe is available and there is no existing pipe, static or dynamic, with a name matching the null-terminated string pointed to by *pname*, the service allocates a dynamic pipe and applies the name referenced by *pname* to the new pipe. The service stores the handle of the new dynamic pipe in the variable pointed to by *ppipe*. The kernel stores only the address of the name internally, which means that the same array cannot be used to build multiple dynamic pipe names.

If *pname* is a null pointer `((char *)0)`, the service does not assign a name to the dynamic pipe. However, if *pname* points to a null string (`""`), the name is legal as long as no other pipe is already using a null string as its name.

If the service finds an existing pipe with a matching name, it does not open a new pipe and returns a value indicating an unsuccessful operation.

Note: To use this service, you must enable the *Dynamics* attribute of the Pipe class during system generation.

If the pointer to the pipe name is not null, the time required to perform this operation varies with the number of pipe names in use.

If the pointer to the pipe name is null, no search of pipe names takes place and the time to perform the service is fixed. You can define the pipe name at a later time with a call to the `KS_DefPipeName` service.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully. The service stores the handle of the new dynamic pipe in the variable pointed to by *ppipe*.
- `RC_OBJECT_ALREADY_EXISTS` if the name search finds another pipe whose name matches the specified string.
- `RC_NO_OBJECT_AVAILABLE` if the name search finds no match but all dynamic pipes are in use.

Example

Example 4-15 allocates a dynamic pipe and names it `DynPipe2`.

Example 4-15. Allocate Dynamic Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KSRC ksrc;
PIPE dynpipe;

if ((ksrc = KS_OpenPipe ("DynPipe2", &dynpipe)) != RC_GOOD)
{
   if (ksrc == RC_OBJECT_ALREADY_EXISTS)
      putline ("DynPipe2 pipe name in use");
   else if (ksrc == RC_NO_OBJECT_AVAILABLE)
      putline ("No dynamic pipes available");
   else
      putline ("Pipes are not a defined class");
}
else
{
   ... pipe was opened correctly. Okay to define its properties now
}
```

See Also

`KS_ClosePipe`, page 110
`KS_UsePipe`, page 154

INIT_PipeClassProp

Initialize the Pipe object class properties.

Synopsis

```
KSRC INIT_PipeClassProp
   (const KCLASSPROP *pclassprop)
```

Input

pclassprop	A pointer to a Pipe object class properties structure.

Description

During the **RTXC** Kernel initialization procedure (usually performed in Zone 3), you must define the kernel objects needed by the **RTXC** Kernel to perform the application. The `INIT_PipeClassProp` kernel service allocates space for the Pipe object class in system RAM. The amount of RAM to allocate, and all other properties of the class, are specified in the structure pointed to by *pclassprop*.

The `KCLASSPROP` structure has the following organization:

```
typedef struct
{
   KATTR attributes;
   KOBJECT n_statics;        /* number of static objects */
   KOBJECT n_dynamics;       /* number of dynamic objects */
   short objsize;            /* used for calculating offsets */
   short totalsize;          /* used to alloc object array RAM */
   ksize_t namelen;          /* length of the name string */
   const char *pstaticnames;
} KCLASSPROP;
```

The attributes element of the Pipe property structure supports the attributes and corresponding masks listed in Table 4-1 on page 127.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_NO_RAM` if the initialization fails because there is insufficient system RAM available.

Example

During system initialization, the startup code must initialize the Pipe object class before using any kernel service for that class, regardless of Zone. In Example 4-16 on page 143, the system generation process produced a KCLASSPROP structure containing the information about the kernel class necessary for its initialization. That structure is referenced externally to the code. The example outputs any error messages to the console.

Example 4-16. Initialize Pipe Object Class

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

extern const SYSPROP sysprop;
extern const KCLASSPROP pipeclassprop;

KSRC userinit (void)
{
   KSRC ksrc;
   static char buf[128];

   /* initialize the kernel workspace, allocate RAM for
      required classes, etc. */

   if ((ksrc = INIT_SysProp (&sysprop)) != RC_GOOD)
   {
      putline ("Kernel initialization failure");
      return (ksrc);  /* end initialization process */
   }
   /* kernel is initialized */

   /* Need to initialize the necessary kernel object classes */

   /* Initialize the Pipe Kernel Object class */
   if ((ksrc = INIT_PipeClassProp (&pipeclassprop)) != RC_GOOD)
   {
      putline ("Insufficient RAM for Pipe init.");
      return (ksrc);  /* end initialization process */
   }

... Continue with system initialization

}
```

See Also

KS_GetPipeClassProp, page 126

XX_PutEmptyGetFullPipeBuf

Put an empty buffer into a pipe and then get a full buffer from the same pipe.

Zones

1 IS_PutEmptyGetFullPipeBuf
2 TS_PutEmptyGetFullPipeBuf
3 KS_PutEmptyGetFullPipeBuf

Synopsis

```
void * XX_PutEmptyGetFullPipeBuf (PIPE pipe,
   void * pbuf, int *pbufsize, KSRC *pksrc)
```

Inputs

pipe	The handle of the pipe to use.
pbuf	The pointer to the empty buffer being returned to the specified *pipe*.
pbufsize	A pointer to a variable that will, upon completion of the service, contain the actual size of the full buffer, the pointer to which is being returned as the value of the function.
pksrc	A pointer to KSRC type return code.

Description

The XX_PutEmptyGetFullPipeBuf kernel service allows the specified *pipe*'s consumer to return an empty buffer to *pipe* and, at the same time, get the next available full buffer from *pipe*, returning the pointer to the full buffer to the caller.

It is necessary for the consumer to obtain the size of the buffer as filled by *pipe*'s producer so that the consumer knows how much data there is to process. It is the producer's responsibility to ensure that the size of the filled buffer is less than or equal to the maximum usable size of the buffers for *pipe*.

It is permissible to define *pbuf* as a null pointer ((void *)0) to indicate there is no empty buffer to return into the pipe. If *pbuf* is null, the service ignores the pointer and functions identically to the XX_GetFullPipeBuf kernel service, returning the pointer to the next available full buffer. This technique may be useful when operating a loop in a consumer that uses the combination pipe operation. The first time through the loop, there is no empty buffer to release but the service gets a full buffer, returning its address to allow the consumer to begin operation.

Output

This service returns the pointer to the next available full buffer in *pipe* if the service is successful. If not, it returns a null pointer `((void *)0)`.

The service also returns a `KSRC` type value through the *pksrc* pointer indicating how the service performed. The possible values are:

- `RC_GOOD` if the service was successful.
- `RC_PIPE_FULL` if the specified pipe does not have room for another full buffer. The service may return a valid empty buffer address even though this `KSRC` value is passed back.
- `RC_PIPE_EMPTY` if the specified pipe does not have an available empty buffer. The service may return this code after successfully putting the full buffer into the pipe but not finding an available empty buffer. This code is redundant because the service would also return the null pointer for the empty buffer. It is provided for completeness.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.
- `FE_NULL_PIPEPBUFSIZE` if the pointer to the buffer size is null.

Example

In Example 4-17 on page 146, the `threadA` thread is a pipe consumer having environment arguments in the `myenvargs` structure. The elements of the structure have been previously defined. The `buffer` element represents the address of the empty buffer to release and is initialized with a null pointer. The `maxbufsize` variable contains the maximum useful size of a buffer in `PIPEA`. When `threadA` executes, it receives the pointer to its environment arguments and uses the elements therein to preserve needed values between execution cycles, principally `buffer`. When in operation, it releases the buffer, presumed empty, to the pipe and simultaneously gets the address of the next available full buffer and the buffer's size. Having a full buffer, it processes the data in the buffer in a loop until the buffer is empty. The pointer to the now-empty buffer is stored in the environment argument `buffer` element to be ready for the next execution cycle of `threadA`. The example assumes that the `KSRC` value returned by the service is always `RC_GOOD`.

Example 4-17. Perform Consumer Fast Buffer Exchange on Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

/* environment argument structure for threadA */
struct myenvargs
{
   char *buffer;
   int maxbufsize;
}

void threadA ((void *)0, (struct myenvargs *)myargs)
{
   int bufsize, i;
   KSRC ksrc;

   /* put empty buffer into pipe and get next full buffer */
   myargs->buffer = TS_PutEmptyGetFullPipeBuf (PIPEA,
                       myargs->buffer, &bufsize, &ksrc);

   for (i=0; i<= bufsize; i++)
   {
   ...process the data in the buffer
   }
}
```

See Also

`XX_DefPipeProp`, page 115
`XX_GetPipeProp`, page 130
`XX_JamFullGetEmptyPipeBuf`, page 132
`XX_JamFullPipeBuf`, page 136
`XX_PutEmptyPipeBuf`, page 147
`XX_PutFullGetEmptyPipeBuf`, page 149
`XX_PutFullPipeBuf`, page 152

XX_PutEmptyPipeBuf

Return an empty buffer to a pipe.

Zones

1 IS_PutEmptyPipeBuf
2 TS_PutEmptyPipeBuf
3 KS_PutEmptyPipeBuf

Synopsis

```
KSRC XX_PutEmptyPipeBuf (PIPE pipe, void * pbuf)
```

Inputs

pipe	The handle of the pipe to use.
pbuf	The pointer to the empty buffer to be returned to the pipe.

Description

The `XX_PutEmptyPipeBuf` kernel service allows the specified *pipe*'s consumer to return an empty buffer to *pipe*. The address of the empty buffer is then available for future use by the pipe's producer.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_PIPE_FULL` if the specified pipe does not have room for another empty buffer.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.
- `FE_NULL_PIPEBUFFER` if the specified Pipe buffer address is null.

Example

In Example 4-18 on page 148, the Current Thread gets a full buffer, empties it and returns the buffer to the pipe specified in `PIPEA`.

Example 4-18. Return Empty Buffer to Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

int buffersize, i;
char *buffer;

/* first get a full buffer from PIPEA */
buffer = TS_GetFullPipeBuf (PIPEA, &buffersize);

for (i=0; i<=buffersize; i++)
{
... Process the data in the buffer

/* release empty buffer to pipe */
TS_PutEmptyPipeBuf (PIPEA, buffer);

... continue
```

See Also

`XX_DefPipeProp`, page 115
`XX_GetPipeProp`, page 130
`XX_JamFullGetEmptyPipeBuf`, page 132
`XX_JamFullPipeBuf`, page 136
`XX_PutEmptyGetFullPipeBuf`, page 144
`XX_PutFullGetEmptyPipeBuf`, page 149
`XX_PutFullPipeBuf`, page 152

XX_PutFullGetEmptyPipeBuf

Put a full buffer into a pipe and then get an empty buffer from the same pipe.

Zones

1 `IS_PutFullGetEmptyPipeBuf`
2 `TS_PutFullGetEmptyPipeBuf`
3 `KS_PutFullGetEmptyPipeBuf`

Synopsis

```
void * XX_PutFullGetEmptyPipeBuf (PIPE pipe,
   void *pbuf, int bufsize, KSRC *pksrc)
```

Inputs

pipe	The handle of the pipe to use.
pbuf	The pointer to the full buffer to be put into the specified *pipe*. A null pointer is valid.
bufsize	The actual size of the buffer as filled. This number must be less than or equal to the maximum usable buffer size for the specified *pipe*. If *pbuf* is a null pointer, the service ignores *bufsize*.
pksrc	A pointer to `KSRC` type return code.

Description

The `XX_PutFullGetEmptyPipeBuf` kernel service allows the specified *pipe*'s producer to put a full buffer into *pipe* and at the same time, get the next available empty buffer from *pipe*, returning the pointer to the empty buffer to the caller.

It is necessary for the producer to state the size of the buffer as filled so that *pipe*'s consumer knows how much data there is to process. The size of the filled buffer must be less than or equal to the maximum usable size of the buffers for *pipe*.

It is permissible to define *pbuf* as a null pointer (`(void *)0`) to indicate there is no full buffer to put into the pipe. If *pbuf* is null, the service ignores it and operates identically to the `XX_GetEmptyPipeBuf` service, returning the pointer to the next available empty buffer. This technique may be useful when employing a loop in a producer that uses the combination pipe operation. The first time through the loop, there is no full buffer but the service allocates an empty buffer allowing the producer to begin operation.

If *pbuf* is a null pointer, the service ignores the value of *bufsize*. Ideally, in this situation, *bufsize* would contain a value of zero (0).

Output

This service returns the pointer to the next available empty buffer if the service is successful. If not, it returns a null pointer (`(void *)0`).

The service also returns a `KSRC` type value through the *pksrc* pointer indicating how the service performed. The possible values are:

- `RC_GOOD` if the service was successful.
- `RC_PIPE_FULL` if the specified pipe does not have room for another full buffer. The service may return a valid empty buffer address even though this `KSRC` value is passed back.
- `RC_PIPE_EMPTY` if the specified pipe does not have an available empty buffer. The service may return this code after successfully putting the full buffer into the pipe but not finding an available empty buffer. This code is redundant because the service would also return the null pointer for the empty buffer. It is provided for completeness.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.

Example

In Example 4-19 on page 151, the `threadA` pipe producer thread has environment arguments in the `myenvargs` structure and the elements of the structure have been previously defined. The `buffer` element represents the address of the next empty buffer to process and is initialized with a valid pointer to an empty buffer in `PIPEA`, and `maxbufsize` contains the maximum useful size of a buffer in `PIPEA`. When `threadA` executes, it receives the pointer to its environment argument structure and uses the elements therein to preserve needed variables between execution cycles, principally the `buffer` variable. When in operation, it fills the buffer in a loop until the buffer reaches the maximum useful size. The thread then puts the full buffer into the pipe at its tail, simultaneously getting the next available empty buffer in the pipe. The address of the next empty buffer is stored in the environment argument `buffer` element to be ready for the next execution cycle of `threadA`. The example assumes that the `KSRC` value returned by the service is always `RC_GOOD`.

Example 4-19. Perform Fast Producer Buffer Exchange on Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

/* environment argument structure for threadA */
struct myenvargs
{
   char *buffer;
   int maxbufsize;
}

void threadA ((void *)0, (struct myenvargs *)myargs)
{
   int bufsize;
   KSRC ksrc;

   for (bufsize=0; bufsize<=myargs->maxbufsize; bufsize++)
   {
   ...fill the buffer
   }

   /* put full buffer at tail of pipe and get next empty buffer */
   myargs->buffer = TS_PutFullGetEmptyPipeBuf (PIPEA,
                        myargs->buffer, bufsize, &ksrc);
}
```

See Also

`XX_DefPipeProp`, page 115
`XX_GetPipeProp`, page 130
`XX_JamFullGetEmptyPipeBuf`, page 132
`XX_JamFullPipeBuf`, page 136
`XX_PutEmptyGetFullPipeBuf`, page 144
`XX_PutEmptyPipeBuf`, page 147
`XX_PutFullPipeBuf`, page 152

XX_PutFullPipeBuf

Put a full buffer into a pipe.

Zones

1 IS_PutFullPipeBuf
2 TS_PutFullPipeBuf
3 KS_PutFullPipeBuf

Synopsis

```
KSRC XX_PutFullPipeBuf (PIPE pipe, void * pbuf,
   int bufsize)
```

Inputs

pipe	The handle of the pipe to use.
pbuf	The pointer to the full buffer to be put into the specified *pipe*. The pointer must not be null.
bufsize	The actual size of the buffer as filled. This number must be less than or equal to the maximum usable buffer size for the specified *pipe*.

Description

The `XX_PutFullPipeBuf` kernel service allows the specified *pipe*'s producer to put a full buffer into *pipe* at its tail.

It is necessary for the producer to state the size of the buffer as filled so that *pipe*'s consumer knows how much data there is to process. The size of the filled buffer must be less than or equal to the maximum usable size of the buffers for *pipe*.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service was successful.
- `RC_PIPE_FULL` if the specified pipe does not have room for another full buffer.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_PIPE` if the specified pipe ID is not valid.
- `FE_UNINITIALIZED_PIPE` if the specified pipe has not yet been initialized.

- FE_NULL_PIPEBUFFER if the specified Pipe buffer address is null.
- FE_ZERO_PIPEBUFSIZE if the buffer size in the specified pipe is zero.

Example

In Example 4-20, the Current Thread fills a buffer and puts the buffer in the pipe specified in PIPEA.

Example 4-20. Put Full Buffer into Pipe

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kpipe.h"        /* PIPEA */

int buffersize;
void *buffer;

/* get pointer to an empty buffer */
buffer = TS_GetEmptyPipeBuf (PIPEA);

... fill the buffer until its size = maxbufsize for the pipe

/* put full buffer back into pipe */
TS_PutFullPipeBuf (PIPEA, buffer, &buffersize);

... continue
```

See Also

XX_DefPipeProp, page 115
XX_GetPipeProp, page 130
XX_JamFullGetEmptyPipeBuf, page 132
XX_JamFullPipeBuf, page 136
XX_PutEmptyGetFullPipeBuf, page 144
XX_PutEmptyPipeBuf, page 147
XX_PutFullGetEmptyPipeBuf, page 149

KS_UsePipe

Look up a dynamic pipe by name and mark it for use.

Synopsis

```
KSRC KS_UsePipe (const char *pname, PIPE *ppipe)
```

Inputs

pname	A pointer to a null-terminated name string.
ppipe	A pointer to a variable in which to store the matching handle, if found.

Description

The `KS_UsePipe` kernel service acquires the handle of a dynamic pipe by looking up the null-terminated string pointed to by *pname* in the list of pipe names. If there is a match, the service registers the pipe for future use by the Current Task and stores the matching handle in the variable pointed to by *ppipe*. This procedure allows the Current Task to reference the dynamic pipe successfully in subsequent kernel service calls.

Note: To use this service, you must enable the *Dynamics* attribute of the Pipe class during system generation.

The time required to perform this operation varies with the number of pipe names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search and registration is successful. The service stores the matching handle in the variable pointed to by *ppipe*.
- `RC_STATIC_OBJECT` if the specified name belongs to a static pipe.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching pipe name.

Example

Example 4-21 on page 155 locates the `DynPipe3` dynamic pipe by name and obtains the pipe handle. It then outputs a message to the console indicating the handle of the pipe if successful or an error message if unsuccessful.

Example 4-21. Read Pipe Handle and Register It

```
#include <stdio.h>          /* standard i/o */
#include "rtxcapi.h"        /* RTXC Kernel Service prototypes */

PIPE dynpipe;
KSRC ksrc;
static char buf[128];

if ((ksrc = KS_UsePipe ("DynPipe3", &dynpipe)) != RC_GOOD)
{
   if (ksrc == RC_STATIC_OBJECT)
      putline ("Pipe DynPipe3 is a static pipe");
   else
      putline ("Pipe DynPipe3 not found");
}
else
{
   /* pipe was found and its handle is in dynpipe. */
   sprintf (buf, "DynPipe3 is pipe %d", dynpipe);
   putline (buf);
}
```

See Also

`KS_DefPipeName`, page 118
`KS_GetPipeName`, page 128

CHAPTER 5

Event Source Services

In This Chapter

We describe the Event Source kernel services in detail. The Event Source services maintain and update accumulators to count the number of source events as well as to serve as the base for related Counters and Alarms.

XX_ClearEventSourceAttr

Clear one or more event source attributes.

Zones

2 `TS_ClearEventSourceAttr`
3 `KS_ClearEventSourceAttr`

Synopsis

```
void XX_ClearEventSourceAttr (EVNTSRC evntsrc,
   ATTRMASK amask)
```

Input

evntsrc	The handle of the event source containing the attributes to be cleared.
amask	A mask value containing the bits to clear in the attribute property of the specified event source.

Description

The `XX_ClearEventSourceAttr` kernel service clears bits in the attribute property of the event source specified in *evntsrc* according to the bits specified in *amask*.

The *attributes* element of an Event Source object supports the attribute and corresponding mask listed in Table 5-1 on page 164.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.
- `FE_UNINITIALIZED_EVNTSRC` if the specified event source has not yet been initialized.

Example

In Example 5-1 on page 159, the Current Thread clears the disable bit in the event source specified in `EVNTSRC1` to enable further processing of Events.

Example 5-1. Clear Event Source Attribute

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "keventsrc.h"    /* EVNTSRC1 */

/* clear disable bit in event source to re-enable */
TS_ClearEventSourceAttr (EVNTSRC1, ATTR_ENVTSRC_DISABLE);

... continue
```

See Also

`XX_SetEventSourceAttr`, page 185

KS_CloseEventSource

End the use of a dynamic event source.

Synopsis

```
KSRC KS_CloseEventSource (EVNTSRC evntsrc)
```

Input

evntsrc	The handle for an event source.

Description

The `KS_CloseEventSource` kernel service ends the Current Task's use of the dynamic event source specified in *evntsrc*. When closing *evntsrc*, the service detaches the caller's use of it. If the caller is the last user of *evntsrc*, the service releases *evntsrc* to the free pool of dynamic event sources for reuse. If there is at least one other task still using *evntsrc*, the service does not release *evntsrc* to the free pool but completes successfully.

Note: To use this service, you must enable the *Dynamics* attribute of the Event Source class during system generation.

Output

This service returns a `KSRC value` as follows:

- `RC_GOOD` if the service is successful.
- `RC_STATIC_OBJECT` if the specified event source is not dynamic.
- `RC_OBJECT_NOT_INUSE` if the specified event source does not correspond to an active dynamic event source.
- `RC_OBJECT_INUSE` if the Current Task's use of the specified event source is closed but the event source remains open for use by other tasks.

Note: `RC_OBJECT_INUSE` does not necessarily indicate an error condition. The calling task must interpret its meaning.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.

Example

In Example 5-2, the Current Task waits on a signal from another task indicating that it is time to close the dynamic event source specified in *dynevntsrc*. When the Current Task receives the signal, it closes the associated event source.

Example 5-2. Close Event Source

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

EVNTSRC dynevntsrc;
SEMA dynsema;

KS_TestSemaW (dynsema);    /* wait for signal */

KS_CloseEventSource (dynevntsrc); /* then close the event source */
```

See Also

`KS_OpenEventSource`, page 179
`KS_UseEventSource`, page 187

KS_DefEventSourceName

Define the name of a previously opened event source.

Synopsis

```
KSRC KS_DefEventSourceName (EVNTSRC evntsrc,
   const char *pname)
```

Inputs

evntsrc	The handle of the event source being defined.
pname	A pointer to a null-terminated name string.

Description

The `KS_DefEventSourceName` kernel service names or renames the dynamic event source specified in *evntsrc*. The service uses the null-terminated string pointed to by *pname* for *evntsrc*'s new name.

Static event sources cannot be named or renamed under program control.

Note: To use this service, you must enable the *Dynamics* attribute of the Event Source class during system generation.

This service does not check for duplicate event source names.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_STATIC_OBJECT` if the event source being named is static.
- `RC_OBJECT_NOT_FOUND` if the Dynamics attribute of the Event Source class is not enabled.
- `RC_OBJECT_NOT_INUSE` if the specified event source does not correspond to an active dynamic event source.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.

Example

Example 5-3 assigns the name `NewEventSource` to the event source specified in `dynevntsrc` so other users may reference it by name.

Example 5-3. Assign Event Source Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KSRC ksrc;
EVNTSRC dynevntsrc;

if ((ksrc = KS_DefEventSourceName (dynevntsrc, "NewEventSource"))
    != RC_GOOD)
{
   if (ksrc == RC_OBJECT_NOT_FOUND)
      putline ("Dynamic Event Sources are not enabled");
   else if (ksrc == RC_STATIC_OBJECT)
   {
      sprintf (buf, "Event Source %d is a static event source",
               dynevntsrc);
      putline (buf);
   }
   else
   {
      sprintf (buf, "Event Source %d is not active.",
               dynevntsrc);
      putline (buf);
   }
}

... naming operation was successful. Continue
```

See Also

`KS_OpenEventSource`, page 179
`KS_GetEventSourceName`, page 173
`KS_LookupEventSource`, page 177
`KS_UseEventSource`, page 187

XX_DefEventSourceProp

Define the event source's properties.

Zones

2 `TS_DefEventSourceProp`
3 `KS_DefEventSourceProp`

Synopsis

```
void KS_DefEventSourceProp (EVNTSRC evntsrc,
   const EVNTSRCPROP *pevntsrcprop)
```

Inputs

evntsrc	The handle of the event source being defined.
pevntsrcprop	A pointer to an Event Source properties structure.

Description

The `XX_DefEventSourceProp` kernel service defines the properties of the event source specified in *evntsrc* using the values contained in the `EVNTSRCPROP` structure pointed to by *pevntsrcprop*.

Example 5-4 shows the organization of the `EVNTSRCPROP` structure.

Example 5-4. Event Source Properties Structure

```
typedef struct
{
   KATTR attributes; /* Event Source attributes (DISABLE only) */
} EVNTSRCPROP;
```

The *attributes* element of an Event Source object supports the attribute and corresponding mask listed in Table 5-1.

Table 5-1. Event Source Attributes and Masks

Attribute	Mask
Disable	`ATTR_DISABLE`

Setting the *Disable* attribute disables processing of event source ticks with the `XX_ProcessEventSourceTick` service. Clearing the *Disable* attribute enables tick processing on the event source.

Note: Define a event source's properties only when the event source is not busy.
This kernel service is not intended to permit unrestricted enabling and disabling of a event source's *Disable* attribute. While no restrictions are placed on its frequency of use, you should use this service before the first use of the event source.

Output

This service does not return a value.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.

Example

During system initialization, the startup routine must create and initialize the Event Source object class and define the properties of all the static event sources before the system can process the events associated with the sources, as illustrated in Example 5-5.

Example 5-5. Define Event Source Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

extern const  KCLASSPROP evntsrcclassprop;
extern const  ENVSRCPROP evntsrcprop[];

KSRC ksrc;
int objnum;

/* initialize the Event Source class/object data */
if ((ksrc = INIT_EventSourceClassProp (&evntsrcclassprop))
    != RC_GOOD)
   return ksrc;

for (objnum = 1; objnum <= evntsrcclassprop.n_statics; objnum++)
{
   TS_DefEventSourceProp (objnum, &evntsrcprop[objnum]);
}

... continue
```

See Also

`XX_GetEventSourceProp`, page 175
`INIT_EventSourceClassProp`, page 167
`KS_OpenEventSource`, page 179

INIT_EventSourceClassProp

Initialize the Event Source object class properties.

Synopsis

```
KSRC INIT_EventSourceClassProp
   (const KCLASSPROP *pclassprop)
```

Input

pclassprop A pointer to a Event Source object class properties structure.

Description

During the RTXC initialization procedure, you must define the kernel objects needed by the kernel to perform the application. The `INIT_EventSourceClassProp` kernel service allocates space for the Event Source object class in system RAM. The amount of RAM to allocate, and all other properties of the class, are specified in the `KCLASSPROP` structure pointed to by *pclassprop*.

The `KCLASSPROP` structure has the following organization:

```
typedef struct
{
  KATTR attributes;
  KOBJECT n_statics;        /* number of static objects */
  KOBJECT n_dynamics;       /* number of dynamic objects */
  short objsize;            /* used for calculating offsets */
  short totalsize;          /* used to alloc object array RAM */
  ksize_t namelen;          /* length of the name string */
  const char *pstaticnames;
} KCLASSPROP;
```

The `attributes` element of the Event Source `KCLASSPROP` structure supports the class property attributes and corresponding masks listed in Table 5-2 on page 171.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_NO_RAM` if the initialization fails because there is insufficient system RAM available.

Example

During system initialization, the startup code must initialize the Event Source object class before using any kernel service for that class. The system generation process produces a KCLASSPROP structure containing the information about the kernel object necessary for its initialization. In Example 5-6, that structure is referenced externally to the code module.

Example 5-6. Initialize Event Source Object Class Properties

```
#include "rtxcapi.h"       /* RTXC Kernel Service prototypes */

extern const SYSPROP sysprop;
extern const KCLASSPROP evntsrcclassprop;

KSRC userinit (void)
{
   KSRC ksrc;

   /* Initialize the kernel workspace and allocate RAM */
   /* for required classes, etc. */
   if ((ksrc = INIT_SysProp (&sysprop)) != RC_GOOD)
   {
      putline ("Kernel initialization failure");
      return (ksrc); /* end initialization process */
   }

   /* Initialize the necessary kernel object classes */

   /* Initialize the Event Source kernel object class */
   if ((ksrc = INIT_EventSourceClassProp (&evntsrcclassprop))
       != RC_GOOD)
   {
      putline ("No RAM for Event Source init");
      return (ksrc); /* end initialization process */
   }

... Continue with system initialization

}
```

See Also

KS_GetEventSourceClassProp

XX_GetEventSourceAcc

Get the event sources's accumulator.

Zones

1 IS_GetEventSourceAcc
2 TS_GetEventSourceAcc
3 KS_GetEventSourceAcc

Synopsis

```
TICKS KS_GetEventSourceAcc (EVNTSRC evntsrc)
```

Input

evntsrc	The handle of the event source to be read.

Description

The `XX_GetEventSourceAcc` kernel service reads the event accumulator of the event source specified in *evntsrc* and returns the value read to the caller.

Output

This service returns the event accumulator of the specified event source to a variable of type `TICKS`.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.
- `FE_UNINITIALIZED_EVNTSRC` if the specified event source has not yet been initialized.

Example

In Example 5-7 on page 170, the Current Thread needs to know how many ticks have occurred on the event source specified in `EVNTSRC1`.

Example 5-7. Read Event Source Accumulator

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kevntsrc.h"     /* EVNTSRC1 */

TICKS currticks;

/* get current tick count on EVNTSRC1 */
currticks = TS_GetEventSourceAcc (EVNTSRC1);

... do something with currticks from EVNTSRC1

... continue
```

See Also

`XX_ProcessEventSourceTick`, page 181

KS_GetEventSourceClassProp

Get the Event Source object class properties.

Synopsis

```
const KCLASSPROP * KS_GetEventSourceClassProp
   (int *pint)
```

Input

pint A pointer to a variable in which to store the number of available dynamic event sources. The value of *pint* may be null (`(int *)0`), in which case the service does not return the number of dynamic event sources.

Description

The `KS_GetEventSourceClassProp` kernel service obtains a pointer to the `KCLASSPROP` structure that was used during system initialization by the `INIT_EventSourceClassProp` service to initialize the Event Source object class properties. If *pint* is not null (`(int *)0`), the service returns the number of available dynamic event sources in the variable pointed to by *pint*. If *pint* is null, the service does not return the number of dynamic event sources.

Table 2-13 on page 44 shows the organization of the `KCLASSPROP` structure.

The value of the *attributes* element of the Event Source `KCLASSPROP` structure is determined by the selections you make during the system configuration procedure. It supports the class property attributes and corresponding masks listed in Table 5-2.

Table 5-2. Event Source Class Attributes and Masks

Attribute	Mask
Static Names	`ATTR_STATIC_NAMES`
Dynamics	`ATTR_DYNAMICS`

Output

If successful, this service returns a pointer to a `KCLASSPROP` structure.

If the Event Source class is not initialized, the service returns a null pointer (`(KCLASSPROP *)0`).

If *pint* is not null, the service returns the number of available dynamic event sources, provided that the *Dynamics* attribute is enabled (Set). If the Dynamics attribute is disabled (Clear), the service stores a value of zero (0) in the variable pointed to by *pint*.

Example

In Example 5-8, the Current Task accesses the information contained in the `KCLASSPROP` structure for the Event Source class.

Example 5-8. Read Event Source Object Class Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KCLASSPROP *pevntsrcclassprop;
int free_dyn;

/* Get the event source kernel object class properties */
if ((pevntsrcclassprop = KS_GetEventSourceClassProp (&free_dyn))
    == (KCLASSPROP *)0)
{
   putline ("Event Source Class not initialized");
}
else
{
   ... event source object class properties are available for use
       "free_dyn" is the number of available dynamic event sources
}
```

See Also

`INIT_EventSourceClassProp`, page 167

KS_GetEventSourceName

Get the event source's name.

Synopsis

```
char * KS_GetEventSourceName (EVNTSRC evntsrc)
```

Input

evntsrc	The handle of the event source being queried.

Description

The `KS_GetEventSourceName` kernel service obtains a pointer to the null-terminated string containing the name of the event source specified in *evntsrc*. The event source may be static or dynamic.

Note: To use this service, you must select the *Dynamics* option for the Event Source class during system generation.

To use this service on static event sources, you must select the *Static Names* option for the Event Source class during system generation.

Output

If *evntsrc* has a name, this service returns a pointer to the null-terminated name string.

If *evntsrc* has no name, the service returns a null pointer (`(char *)0`).

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.

Example

In Example 5-9 on page 174, the Current Task reports the name of the dynamic event source specified in `dynevntsrc`.

Example 5-9. Read Event Source Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

static char buf[128];

EVNTSRC dynevntsrc;
char *pname;

if ((pname = KS_GetEventSourceName (dynevntsrc)) == (char *)0)
   sprintf (buf, "Event Source %d has no name", dynevntsrc);
else
   sprintf (buf, "Event Source %d name is %s", dynevntsrc, pname);

putline (buf); /* send buffer to console */
```

See Also

`KS_DefEventSourceName`, page 162
`KS_OpenEventSource`, page 179

XX_GetEventSourceProp

Get the event source's properties.

Zones

2 `TS_GetEventSourceProp`
3 `KS_GetEventSourceProp`

Synopsis

```
void XX_GetEventSourceProp (EVNTSRC evntsrc,
   EVNTSRCPROP *pevntsrcprop)
```

Inputs

evntsrc	The handle of the event source being queried.
pevntsrcprop	A pointer to an Event Source properties structure.

Description

The `XX_GetEventSourceProp` kernel service obtains all of the property values of the event source specified in *evntsrc* in a single call. The service stores the property values in the `EVNTSRCPROP` structure pointed to by *pevntsrcprop*.

Example 5-4 on page 164 shows the organization of the `EVNTSRCPROP` structure.

The *attributes* element of an Event Source object supports the attribute and corresponding mask listed in Table 5-1 on page 164.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.
- `FE_UNINITIALIZED_EVNTSRC` if the specified event source has not yet been initialized.

Example

In Example 5-10 on page 176, the Current Thread needs to know the status of the disable bit in the attributes of the event source specified in `EVNTSRC1`.

Example 5-10. Read Event Source Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kevntsrc.h"     /* EVNTSRC1 */

EVNTSRCPROP  evntsrcprop;

/* get current Event Source Properties */
TS_GetEventSourceProp (EVNTSRC1, &evntsrcprop);

/* is event source disable? */
if (evntsrcprop.attributes && ATTR_DISABLE)
{
   ... do something, Event Source is disabled
}

... continue
```

See Also

`XX_DefEventSourceProp`, page 164

KS_LookupEventSource

Look up an event source by its name to get its handle.

Synopsis

```
KSRC KS_LookupEventSource (const char *pname,
   EVNTSRC *pevntsrc)
```

Inputs

pname	A pointer to a null-terminated name string.
pevntsrc	A pointer to a variable in which to store the event source handle.

Description

The `KS_LookupEventSource` kernel service obtains the handle of the static or dynamic event source whose name matches the null-terminated string pointed to by *pname*. The lookup process terminates when it finds a match between the specified string and a static or dynamic event source name or when it finds no match. The service stores the handle of the matching event source in the variable pointed to by *pevntsrc*. The service searches dynamic names, if any, first.

Note: To use this service on static event sources, you must enable the *Static Names* attribute of the Event Source class during system generation.

This service has no effect on the registration of the specified event source by the Current Task.

The time required to perform this operation varies with the number of event source names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search succeeds. The service also stores the handle of the matching event source in the variable pointed to by *pevntsrc*.

- RC_OBJECT_NOT_FOUND if the service finds no matching event source name.

Example

In Example 5-11, the Current Task needs to use the dynamic event source named Chnl2EventSource. If the event source is found, the Current Task reads its accumulator.

Example 5-11. Look Up Event Source by Name

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

EVNTSRC dynevntsrc;
TICKS chnl2evnts;
KSRC ksrc;

/* lookup the event source name to see if it exists */
if (KS_LookupEventSource ("Chnl2EventSource", &dynevntsrc)
    != RC_GOOD)
{
   ... Event Source name not found. Deal with it
}
else  /* event source exists */
{
   /* get the event source's accumulator */
   chnl2evnts = KS_GetEventSourceAcc (dynevntsrc);

   ...OK to use accumulator for "Chnl2EventSource" now

}
```

See Also

KS_DefEventSourceName, page 162
KS_GetEventSourceName, page 173
KS_OpenEventSource, page 179

KS_OpenEventSource

Allocate and name a dynamic event source.

Synopsis

```
KSRC KS_OpenEventSource (const char *pname,
   EVNTSRC *pevntsrc)
```

Inputs

pname	A pointer to a null-terminated name string.
pevntsrc	A pointer to a variable in which to store the event source handle.

Description

The `KS_OpenEventSource` kernel service allocates, names, and obtains the handle of a dynamic event source. If a dynamic event source is available and there is no existing event source, static or dynamic, with a name matching the null-terminated string pointed to by *pname*, the service allocates a dynamic event source and applies the name referenced by *pname* to the new event source. The service stores the handle of the new dynamic event source in the variable pointed to by *pevntsrc*. The kernel stores only the address of the name internally, which means that the same array cannot be used to build multiple dynamic event source names.

If *pname* is null (`(char *)0`), the service does not assign a name to the dynamic event source. However, if *pname* points to a null string (`""`), the name is legal as long as no other event source is already using a null string as its name.

If the service finds an existing event source with a matching name, it does not open a new event source and returns a value indicating an unsuccessful operation.

Note: To use this service, you must enable the *Dynamics* attribute of the Event Source class during system generation.

If the pointer to the event source name is not null (`(char *)0`), the time required to perform this operation varies with the number of event source names in use.

If *pname* is null, no search of event source names takes place and the time to perform the service is fixed. You can define the event source name at a later time with a call to the `KS_DefEventSourceName` service.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully. The service also stores the handle of the allocated event source in the variable pointed to by *pevntsrc*.
- `RC_OBJECT_ALREADY_EXISTS` if the name search finds another event source whose name matches the specified string.
- `RC_NO_OBJECT_AVAILABLE` if the name search finds no match but all dynamic event sources are in use.

Example

Example 5-12 allocates a dynamic event source and names it `Chnl2EventSource`. If the name is already being used, the example outputs a message on the console.

Example 5-12. Allocate and Name Event Source

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KSRC ksrc;
EVNTSRC dynevntsrc;

if ((ksrc = KS_OpenEventSource ("Chnl2EventSource", &dynevntsrc))
    != RC_GOOD)
{
   if (ksrc == RC_OBJECT_ALREADY_EXISTS)
      putline ("Chnl2EventSource event source name in use");
   else if (ksrc == RC_NO_OBJECT_AVAILABLE)
      putline ("No dynamic event sources available");
   else
      putline ("Event Sources object class not defined");
}
... event source was opened correctly. Okay to use it now
```

See Also

`KS_CloseEventSource`, page 160
`KS_LookupEventSource`, page 177
`KS_UseEventSource`, page 187

XX_ProcessEventSourceTick

Process a tick on an event source.

Zones

1 `IS_ProcessEventSourceTick`
2 `TS_ProcessEventSourceTick`
3 `KS_ProcessEventSourceTick`

Synopsis

```
KSRC XX_ProcessEventSourceTick (EVNTSRC evntsrc,
   TICKS nevnts)
```

Inputs

evntsrc	The handle of the event source being updated with a new tick (or ticks).
nevnts	The number of ticks to process for the specified event source.

Description

Provided the `ATTR_DISABLED` attribute is cleared in the *attributes* property of the event source specified in *evntsrc*, the `XX_ProcessEventSourceTick` kernel service performs all of the **RTXC** Kernel-dependent functions necessary when an event source tick occurs, including updating of all counters associated with *evntsrc* and all alarms associated with those counters. The source of the tick may be external or an internal. The service may process more than one tick per call, as specified in *nevnts*.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if no alarm expiration occurred as a result of the event source tick.
- `RC_ALARM_EXPIRED` if an alarm expires on a counter associated with the specified event source as a result of the call to `XX_ProcessEventSourceTick`.
- `RC_EVNTSRC_DISABLED` if the specified event source has been disabled.

Example

In Example 5-13 on page 182, for diagnostic purposes, a clock interrupt service routine is tracking how many alarm expirations occur as a result of processing ticks from the `TIMEBASE` event source. The `ticks` variable specifies the number of ticks to process. The interrupt service routine uses a second event source, named `ALARMEXPS`, to accumulate the number of alarm expiration notifications it receives as a result of processing event source ticks on `TIMEBASE`.

Example 5-13. Process Source Event for Clock Tick

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "eventsrc.h"     /* defines ALARMEXPS */

Global TICKS ticks;

/* start of the interrupt service routine device handler */

...got a clock interrupt and have number of ticks
   to process in "ticks"

if (IS_ProcessEventSourceTick (TIMEBASE, ticks)
    == RC_ALARM_EXPIRED)
   IS_ProcessEventSourceTick (ALARMEXPS, (TICKS)1);

... continue
```

XX_SetEventSourceAcc

Set the event source's accumulator to a specified value.

Zones

2 `TS_SetEventSourceAcc`
3 `KS_SetEventSourceAcc`

Synopsis

```
void XX_SetEventSourceAcc (EVNTSRC evntsrc,
   TICKS ticks)
```

Inputs

evntsrc	The handle of the event source to be updated.
ticks	The value to store in the accumulator of the event source.

Description

The `XX_SetEventSourceAcc` kernel service sets the event accumulator of the event source specified in *evntsrc* to the value specified in *ticks*.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.
- `FE_UNINITIALIZED_EVNTSRC` if the specified event source has not yet been initialized.

Example

In Example 5-14 on page 184, the Current Thread needs to set the accumulator in the event source specified in `EVNTSRC1` to zero.

Example 5-14. Set Event Source Accumulator

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kevntsrc.h"     /* EVNTSRC1 */

/* set event source accumulator */
TS_SetEventSourceAcc (EVNTSRC1, (TICKS)0);

... continue
```

See Also

`XX_GetEventSourceAcc`, page 169
`XX_ProcessEventSourceTick`, page 181

XX_SetEventSourceAttr

Set one or more event source attributes.

Zones

2 `TS_SetEventSourceAttr`
3 `KS_SetEventSourceAttr`

Synopsis

```
void XX_SetEventSourceAttr (EVNTSRC evntsrc,
   ATTRMASK amask)
```

Inputs

evntsrc The handle of the event source containing the attributes to be set.

amask A mask value containing the bits to set in the *attribute* property of the event source specified in *evntsrc*.

Description

The `XX_SetEventSourceAttr` kernel service sets bits in the attribute property of the event source specified in *evntsrc* according to the bits specified in *amask*.

The *attributes* element of an Event Source object supports the attribute and corresponding mask listed in Table 5-1 on page 164.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_EVNTSRC` if the specified event source ID is not valid.
- `FE_UNINITIALIZED_EVNTSRC` if the specified event source has not yet been initialized.

Example

In Example 5-15 on page 186, the Current Thread needs to disable the event source specified in `EVNTSRC1` to prevent further processing of events for that event source.

Example 5-15. Set Event Source Attribute Bits

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kevntsrc.h"     /* EVNTSRC1 */

/* disable EVNTSRC1 */
TS_SetEventSourceAttr (EVNTSRC1, ATTR_EVNTSRC_DISABLE);

... continue
```

See Also

`XX_ClearEventSourceAttr`, page 158

KS_UseEventSource

Look up a dynamic event source by name and mark it for use.

Synopsis

```
KSRC KS_UseEventSource (const char *pname,
   EVNTSRC *pevntsrc)
```

Inputs

pname	A pointer to a null-terminated name string.
pevntsrc	A pointer to a variable in which to store the event source handle.

Description

The `KS_UseEventSource` kernel service acquires the handle of a dynamic event source by looking up the null-terminated string pointed to by *pname* in the list of event source names. If there is a match, the service registers the event source for future use by the Current Task and stores the handle of the matching event source in the variable pointed to by *pevntsrv*. This procedure allows the Current Task to reference the dynamic event source successfully in subsequent kernel service calls.

Note: To use this service, you must enable the *Dynamics* attribute of the Event Source class during system generation.

The time required to perform this operation varies with the number of event source names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search is successful. The service also stores the handle of the matching event source in the variable pointed to by *pevntsrc*.
- `RC_STATIC_OBJECT` if the specified name belongs to a static event source.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching event source name.

Example

Example 5-16 locates a dynamic event source named `DynMuxEventSource3` and obtains its handle for subsequent use.

Example 5-16. Read Event Source Handle and Register It

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KSRC ksrc;
EVNTSRC dynevntsrc;

if ((ksrc = KS_UseEventSource ("DynMuxEventSource3", &dynevntsrc))
    != RC_GOOD)
{
   if (ksrc == RC_STATIC_OBJECT)
      putline ("DynMuxEventSource3 is a static event source");
   else
      putline ("Event Source DynMuxEventSource3 name not found");
}

... event source was found and its handle is in dynevntsrc.
    Okay to use it now
```

See Also

`XX_DefEventSourceProp`, page 164
`XX_ClearEventSourceAttr`, page 158
`KS_OpenEventSource`, page 179

CHAPTER 6 Counter Services

In This Chapter

We describe the Counter kernel services in detail. The Counter services maintain and update accumulators for the number of counter ticks used for associated Alarms.

XX_ClearCounterAttr

Clear one or more attributes for a counter.

Zones

2 `TS_ClearCounterAttr`
3 `KS_ClearCounterAttr`

Synopsis

```
void XX_ClearCounterAttr (COUNTER counter,
   KATTRMASK amask)
```

Inputs

counter	The handle of the counter containing the attributes to be cleared.
amask	A mask value containing the bits to clear in the attribute property of the specified counter.

Description

The `XX_ClearCounterAttr` kernel service clears bits in the specified *counter*'s attribute property according to the bits specified in *amask*. For information about the Counter attributes, see “XX_ClearCounterAttr” on page 190.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.
- `FE_UNINITIALIZED_COUNTER` if the specified counter has not yet been initialized.

Example

In Example 6-1 on page 191, the Current Thread clears the disable bit in the counter specified in `COUNTER1` to enable further processing of events on this Counter.

Example 6-1. Clear Counter Attribute

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kcounter.h"     /* COUNTER1 */

/* clear disable bit in counter to re-enable */
TS_ClearCounterAttr (COUNTER1, ATTR_COUNTER_DISABLE);

... continue
```

See Also

`XX_SetCounterAttr`, page 220

KS_CloseCounter

End the use of a dynamic counter.

Synopsis

```
KSRC KS_CloseCounter (COUNTER counter)
```

Input

counter A handle for a dynamic counter.

Description

The `KS_CloseCounter` kernel service ends the Current Task's use of the specified dynamic *counter*. When closing *counter*, the service detaches the caller's use of it. If the caller is the last user of *counter*, the service releases *counter* to the free pool of dynamic counters for reuse. If there is at least one other task still using *counter*, the service does not release the counter to the free pool but completes successfully.

Note: To use this service, you must enable the *Dynamics* attribute of the Counter class during system generation.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service is successful.
- `RC_STATIC_OBJECT` if the specified counter is not dynamic.
- `RC_OBJECT_NOT_INUSE` if the specified counter does not correspond to an active dynamic counter.
- `RC_OBJECT_INUSE` if the Current Task's use of the specified counter is closed but the counter remains open for use by other tasks.

Note: `RC_OBJECT_INUSE` does not necessarily indicate an error condition. The calling task must interpret its meaning.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.

Example

In Example 6-2, the Current Task waits on a signal from another task indicating that it is time to close the dynamic counter specified in `dyncounter`. When the signal is received, the Current Task closes the associated counter.

Example 6-2. Close Counter

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

COUNTER dyncounter;
SEMA dynsema;

KS_TestSemaW (dynsema);        /* wait for signal */

KS_CloseCounter (dyncounter);  /* then close the counter */
```

See Also

`KS_OpenCounter`, page 216

INIT_CounterClassProp

Initialize the Counter object class properties.

Synopsis

```
KSRC INIT_CounterClassProp
   (const KCLASSPROP *pclassprop)
```

Input

pclassprop	A pointer to a Counter object class properties structure.

Description

During the **RTXC** Kernel initialization procedure, you must define the kernel objects needed by the kernel to perform the application. The `INIT_CounterClassProp` kernel service allocates space for the Counter object class in system RAM. The amount of RAM to allocate, and all other properties of the class, are specified in the `KCLASSPROP` structure pointed to by *pclassprop*.

Example 2-13 on page 44 shows the organization the `KCLASSPROP` structure.

The *attributes* element of the Counter `KCLASSPROP` structure supports the class property attributes and corresponding masks listed in Table 6-2 on page 204.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_NO_RAM` if the initialization fails because there is insufficient system RAM available.

Example

During system initialization, the startup code must initialize the Counter object class before using any kernel service for that class. The system generation process produces a `KCLASSPROP` structure containing the information about the kernel object necessary for its initialization. In Example 6-3 on page 195, that structure is referenced externally to the code module.

Example 6-3. Initialize Counter Object Class Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

extern const SYSPROP sysprop;
extern const KCLASSPROP counterclassprop;

KSRC userinit (void)
{
   KSRC ksrc;

   /* initialize the kernel workspace and allocate RAM */
   /* for required classes, etc. */

   if ((ksrc = INIT_SysProp (&sysprop)) != RC_GOOD)
   {
      putline ("Kernel initialization failure");
      return (ksrc); /* end initialization process */
   }
   /* kernel is initialized */

   /* Need to initialize the necessary kernel object classes */

   /* Initialize the Counter kernel object class */
   if ((ksrc = INIT_CounterClassProp (&counterclassprop))
       != RC_GOOD)
   {
      putline ("No RAM for Counter init");
      return (ksrc); /* end initialization process */
   }

... Continue with system initialization

}
```

See Also

`KS_GetCounterClassProp`, page 204

KS_DefCounterName

Define the name of a previously opened dynamic counter.

Synopsis

```
KSRC KS_DefCounterName (COUNTER counter,
   const char *pname)
```

Inputs

counter	The handle of the counter being defined.
pname	A pointer to a null-terminated name string.

Description

The `KS_DefCounterName` kernel service names or renames the specified dynamic *counter*. The service uses the null-terminated string pointed to by *pname* for the new name.

Static counters cannot be named or renamed under program control.

Note: To use this service, you must enable the *Dynamics* attribute of the Counter class during system generation.

This service does not check for duplicate counter names.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_STATIC_OBJECT` if the counter being named is static.
- `RC_OBJECT_NOT_FOUND` if the Dynamics attribute of the Counter class is not enabled.
- `RC_OBJECT_NOT_INUSE` if the specified counter does not correspond to an active dynamic counter.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.

Example

Example 6-4 assigns the name `NewCounter` to the counter specified in `dyncounter` so other users may reference it by name.

Example 6-4. Assign Counter Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

KSRC ksrc;
COUNTER dyncounter;

if ((ksrc = KS_DefCounterName (dyncounter, "NewCounter"))
    != RC_GOOD)
{
   if (ksrc == RC_OBJECT_NOT_FOUND)
     putline ("Dynamic Counters are not enabled");
   else if (ksrc == RC_STATIC_OBJECT)
   {
     sprintf (buf, "Counter %d is a static counter", dyncounter);
     putline (buf);
   }
   else
   {
     sprintf (buf, "Counter %d is not active.", dyncounter);
     putline (buf);
   }
}

... naming operation was successful. Continue
```

See Also

`KS_OpenCounter`, page 216
`KS_GetCounterName`, page 206
`KS_LookupCounter`, page 214
`KS_UseCounter`, page 222

XX_DefCounterProp

Define the counter's properties.

Zones

2 `TS_DefCounterProp`
3 `KS_DefCounterProp`

Synopsis

```
void XX_DefCounterProp (COUNTER counter,
   const COUNTERPROP *pcounterprop)
```

Inputs

counter	The handle of the counter being defined.
pcounterprop	A pointer to a Counter properties structure.

Description

The `XX_DefCounterProp` kernel service defines the properties of the specified *counter* using the values contained in the `COUNTERPROP` structure pointed to by *pcounterprop*.

Example 6-5 shows the organization of the `COUNTERPROP` structure.

Example 6-5. Counter Properties Structure

```
typedef struct
{
   KATTR attributes;    /* counter attributes */
   EVNTSRC evntsrc;
   KMODULUS modulus;
} COUNTERPROP;
```

The *attributes* element of a Counter object supports the attribute and corresponding mask listed in Table 6-1.

Table 6-1. Counter Attributes and Masks

Attribute	Mask
Counter Disable	`ATTR_COUNTER_DISABLE`
Systime Time Counter	`ATTR_COUNTER_TIMEBASE`
Tick Slice Counter	`ATTR_COUNTER_TICKSLICE`

The *Counter Disable* attribute controls updating of the counter's tick accumulator during a XX_ProcessEventSourceTick service. When you set ATTR_COUNTER_DISABLE, the counter's tick accumulator is frozen. When you clear the attribute, the counter's tick accumulator can be updated. The attribute is cleared by default.

The *Systime Time Counter* attribute controls the counter's use as the system timebase. The attribute is cleared by default. When you set ATTR_COUNTER_TIMEBASE using this service or the XX_SetCounterAttr service, the counter has special significance as the system timebase. Therefore, for kernel services in which the user wants to use the system timebase counter as a referenced object, the actual identity of the timebase counter can be represented by the construct (COUNTER)0 (or some suitable symbol defined as (COUNTER)0). This construct makes it possible to reference the system timebase counter without actually knowing its identity.

Note: The developer must ensure that one, and only one, counter in the system has the *Systime Time Counter* attribute enabled. The **RTXC** Kernel does not provide any checking to ensure only one counter has this attribute enabled. After clearing the ATTR_COUNTER_TIMEBASE attribute on one counter, you may enable it on another.

The *Tick Slice Counter* attribute controls the counter's use in tick slice scheduling by the **RTXC** Kernel. The attribute is cleared by default. When you set ATTR_COUNTER_TICKSLICE using this service or the XX_SetCounterAttr service, the counter is used as the source of counter ticks for tick sliced scheduling of tasks by the **RTXC/ms** Scheduler. In this manner, tasks can use any form of tick, not just time, for tick sliced scheduling.

Note: Like the *Systime Time Counter* attribute, there should be one, and only one, counter with the *Tick Slice Counter* attribute enabled at any given time. You may use different counters for the tick slice counter and the system timebase counter.

Define a counter's properties only when the counter is not busy.

This kernel service is not intended to permit unrestricted enabling and disabling of a counter's attributes. While no restrictions are placed on its frequency of use, you should use this service before the first use of *counter*.

If more than one counter has either the *Systime Time Counter* or *Tick Slice Counter* attributes enabled, the **RTXC** Kernel recognizes only the counter most recently defined that has either attribute set for their intended purposes. For information about setting Counter attributes, see "XX_SetCounterAttr" on page 220.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.
- `FE_ILLEGAL_EVENTSOURCE` if the specified event source ID is not valid.

Example

During system initialization, the startup routine must create and initialize the Counter object class and define the properties of all the static counters before the system can process the events on the counters, as illustrated in Example 6-6 on page 201.

Example 6-6. Define Counter Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

extern const  KCLASSPROP counterclassprop;
extern const  COUNTERPROP counterprop[];

KSRC ksrc;
int objnum;

   /* initialize the Counter class/object data */
   if ((ksrc = INIT_CounterClassProp (&counterclassprop))
       != RC_GOOD)
      return ksrc;

   for (objnum = 1; objnum <= counterclassprop.n_statics; objnum++)
   {
         TS_DefCounterProp (objnum, &counterprop[objnum]);
   }

... continue
```

See Also

`XX_GetCounterProp`, page 208
`INIT_CounterClassProp`, page 194
`KS_OpenCounter`, page 216

XX_GetCounterAcc

Get the counter's tick accumulator.

Zones

1 `IS_GetCounterAcc`
2 `TS_GetCounterAcc`
3 `KS_GetCounterAcc`

Synopsis

`TICKS XX_GetCounterAcc (COUNTER counter)`

Input

counter	The handle of the counter to be read.

Description

The `KS_GetCounterClassProp` kernel service reads the specified *counter*'s tick accumulator and returns the value to the caller.

Output

This service returns the tick accumulator value as a `TICKS` type value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.
- `FE_UNINITIALIZED_COUNTER` if the specified counter has not yet been initialized.

Example

In Example 6-7 on page 203, the Current Thread needs to know how many ticks have occurred on the counter specified in `COUNTER1` and on the counter used for the system timebase.

Example 6-7. Read Counter Accumulator

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kcounter.h"     /* COUNTER1 */

TICKS currticks;

/* get current tick count on COUNTER1 */
currticks = TS_GetCounterAcc (COUNTER1);

... do something with currticks from COUNTER1

currticks = TS_GetCounterAcc (TIMEBASE);

... do something with currticks from system time base

... continue
```

See Also

XX_GetElapsedCounterTicks, page 210

KS_GetCounterClassProp

Get the Counter object class properties.

Synopsis

```
const KCLASSPROP * KS_GetCounterClassProp
   (int *pint)
```

Input

pint	A pointer to a variable in which to store the number of available dynamic counters. This argument may be a null pointer (`(void *)0`).

Description

The `KS_GetCounterClassProp` kernel service obtains a pointer to the `KCLASSPROP` structure that was used during system initialization by the `INIT_CounterClassProp` kernel service to initialize the Counter object class properties. If *pint* is not null (`(int *)0`), the service returns the number of available dynamic counters in the variable pointed to by *pint*. If *pint* is null, the service does not return the number of available dynamic counters.

Example 2-13 on page 44 shows the organization of the `KCLASSPROP` structure.

The value of the *attributes* element of the Counter `KCLASSPROP` structure is determined by the selections you make during the system configuration procedure. It supports the class property attributes and corresponding masks listed in Table 6-2.

Table 6-2. Counter Class Attributes and Masks

Attribute	Mask
Static Names	`ATTR_STATIC_NAMES`
Dynamics	`ATTR_DYNAMICS`

Output

If successful, this service returns a pointer to a `KCLASSPROP` structure.

If the Counter class is not initialized, the service returns a null pointer (`(KCLASSPROP *)0`).

If *pint* is not null, the service returns the number of available dynamic counters, provided that the *Dynamics* attribute is enabled (set). If the *Dynamics* attribute is disabled (cleared), the service stores a value of zero (0) in the variable pointed to by *pint*.

Example

In Example 6-8, the Current Task accesses the information contained in the `KCLASSPROP` structure for the Counter object class.

Example 6-8. Read Counter Object Class Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

KCLASSPROP *pcounterclassprop;
int free_dyn;

/* Get the counter kernel object class properties */
if ((pcounterclassprop = KS_GetCounterClassProp (&free_dyn))
    == (KCLASSPROP *)0)
{
   putline ("Counter Class not initialized");
}
else
{
   ... counter object class properties are available for use
       "free_dyn" contains the number of available dynamic counteres
}
```

See Also

`INIT_CounterClassProp`, page 194

KS_GetCounterName

Get the counter's name.

Synopsis

```
char * KS_GetCounterName (COUNTER counter)
```

Input

counter	The handle of the counter being queried.

Description

The `KS_GetCounterName` kernel service obtains a pointer to the null-terminated string containing the name of the specified *counter*. The counter may be static or dynamic.

Note: To use this service on static counters, you must enable the *Static Names* attribute of the Counter class during system generation.

Output

If *counter* has a name, this service returns a pointer to the null-terminated name string.

If *counter* has no name, the service returns a null pointer (`(char *)0`).

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.

Example

In Example 6-9 on page 207, the Current Task reports the name of the dynamic counter specified in `dyncounter`.

Example 6-9. Read Counter Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

static char buf[128];

COUNTER dyncounter;
char *pname;

if ((pname = KS_GetCounterName (dyncounter)) == (char *)0)
   sprintf (buf, "Counter %d has no name", dyncounter);
else
   sprintf (buf, "Counter %d name is %s", dyncounter, pname);

putline (buf); /* send buffer to console */
```

See Also

`KS_DefCounterName`, page 196
`KS_OpenCounter`, page 216

XX_GetCounterProp

Get the counter's properties.

Zones

2 `TS_GetCounterProp`
3 `KS_GetCounterProp`

Synopsis

```
void XX_GetCounterProp (COUNTER counter,
   COUNTERPROP *pcounterprop)
```

Inputs

counter	The handle of the counter being queried.
pcounterprop	A pointer to an Counter properties structure.

Description

The `XX_GetCounterProp` service obtains all of the property values of the specified *counter* in a single call. The service stores the property values in the `COUNTERPROP` structure pointed to by *pcounterprop*.

The `COUNTERPROP` structure has the following organization:

```
typedef struct
{
   KATTR attributes;    /* Counter attributes (DISABLE only) */
   EVNTSRC evntsrc;
   KMODULUS modulus;
} COUNTERPROP;
```

For information about the Counter properties, see "XX_GetCounterProp" on page 208.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.
- `FE_UNINITIALIZED_COUNTER` if the specified counter has not yet been initialized.

Example

In Example 6-10, the Current Thread needs to know the status of the `ATTR_COUNTER_DISABLE` attribute for the counter specified in `COUNTER1`.

Example 6-10. Read Counter Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kcounter.h"     /* COUNTER1 */

COUNTERPROP  counterprop; /* a counter properties structure */

/* get current Counter Properties */
TS_GetCounterProp (COUNTER1, &counterprop);

/* is counter disabled? */
if (counterprop.attributes && ATTR_COUNTER_DISABLE)
{
   ... do some processing if Counter is disabled
}

... continue
```

See Also

`XX_DefCounterProp`, page 198

XX_GetElapsedCounterTicks

Compute the number of counter ticks that have elapsed between two events.

Zones

2 `TS_GetElapsedCounterTicks`
3 `KS_GetElapsedCounterTicks`

Synopsis

```
TICKS XX_GetElapsedCounterTicks (COUNTER counter,
   TICKS *pprevticks)
```

Inputs

counter	The handle of the counter to use for determining the number of elapsed ticks.
pprevticks	A pointer to a variable that contains the value of the tick accumulator for the specified *counter* at a previous event or point in time.

Description

The `XX_GetElapsedCounterTicks` service returns the number of ticks on counter that have elapsed between the current value of *counter*'s tick accumulator and a previous value of *counter*'s tick accumulator represented by the value pointed to by *pprevticks*. The service computes the difference between the current value of *counter*'s tick accumulator and the previous value and returns it to the caller as the number of elapsed ticks. The service then prepares for the next event by putting the current value of *counter*'s tick accumulator into the variable pointed to by pprevticks.

Correct calculation of an elapsed number of ticks requires two service calls. The first call puts the initial value of *counter*'s tick accumulator into the variable pointed to by *pprevticks* and should be done using either this service or the `XX_GetCounterAcc` kernel service The second call should use this service as it returns the number of ticks that have elapsed since the first call. Putting the current tick accumulator value into pprevticks allows you to measure sequential events with single calls to `XX_GetElapsedCounterTicks` after each subsequent period.

Accuracy of the elapsed count is limited by the tick frequency of the specified counter and is guaranteed to be less than the duration of one tick.

Note: If you use the `XX_GetElapsedCounterTicks` kernel service to initialize the variable at *pprevticks*, the `TICKS` value returned by that service call should be discarded because it is unreliable.

Output

This service returns the number of elapsed counter ticks as a `TICKS` type value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.
- `FE_UNINITIALIZED_COUNTER` if the specified counter has not yet been initialized.

Example

Example 6-11 on page 212 calculates the number of ticks on the system timebase counter, defined as `TIMEBASE`, that elapse between two consecutive states of an on/off switch, where the change-of-state event is associated with the `SWITCH` semaphore. The current state of the switch is unknown.

Example 6-11. Obtain Elapsed Counter Ticks between Two Events

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */
#include "ksema.h"        /* defines SWITCH */

TICKS timestamp, diff;

/* wait for the first, and change of state */
KS_TestSemaW (SWITCH);

/* initialize timestamp and disregard return value */
KS_GetElapsedCounterTicks (TIMEBASE, &timestamp);

/*-------------------------------------------------------------*/
/* initialization of timestamp could have been done by the     */
/* following:                                                  */
/* timebase = KS_GetCounterAcc (TIMEBASE);                     */
/*-------------------------------------------------------------*/

KS_TestSemaW (SWITCH);  /* wait for switch change event */
                       /* marking end of first state */

/* get elapsed time since t(0) */
diff = KS_GetElapsedCounterTicks (TIMEBASE, &timestamp);

... use the elapsed number of ticks in "diff" for something ...

KS_TestSemaW (SWITCH); /* wait for next switch change */
                       /* marking end of second state */

/* get elapsed time since start of second state */
diff = KS_GetElapsedCounterTicks (TIMEBASE, &timestamp);

... Use the second period’s elapsed time
```

KS_LookupCounter

Look up a counter by name to get its handle.

Synopsis

```
KSRC KS_LookupCounter (const char *pname,
   COUNTER *pcounter)
```

Inputs

pname	A pointer to a null-terminated name string.
pcounter	A pointer to a variable in which to store the counter handle.

Description

The `KS_LookupCounter` kernel service obtains the handle of the static or dynamic counter whose name matches the null-terminated string pointed to by *pname*. The lookup process terminates when it finds a match between the specified string and a static or dynamic counter name or when it finds no match. The service stores the handle of the matching counter in the variable pointed to by *pcounter*. The service searches dynamic names, if any, first.

Note: To use this service on static counters, you must enable the *Static Names* attribute of the Counter class during system generation.

This service has no effect on the registration of the specified counter by the Current Task.

The time required to perform this operation varies with the number of counter names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search succeeds. The service also stores the handle of the matching counter in the variable pointed to by *pcounter*.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching counter name.

Example

In Example 6-12, the Current Task needs to use the dynamic counter named `Chnl2Counter`. If the counter is found, the Current Task reads its accumulator.

Example 6-12. Look Up Counter by Name

```
#include "rtxcapi.h"       /* RTXC Kernel Services prototypes */

COUNTER dyncounter;
TICKS chnl2cnts;
KSRC ksrc;

/* lookup the counter name to see if it exists */
if (KS_LookupCounter ("Chnl2Counter", &dyncounter) != RC_GOOD)
{
   ... Counter name not found. Deal with it
}
else  /* counter exists */
{
   /* get the counter's accumulator */
   chnl2cnts = KS_GetCounterAcc (dyncounter);

   ok to use accumulator for "Chnl2Counter" now

}
```

See Also

`KS_DefCounterName`, page 196
`KS_GetCounterName`, page 206
`KS_OpenCounter`, page 216

KS_OpenCounter

Allocate and name a dynamic counter.

Synopsis

```
KSRC KS_OpenCounter (const char *pname,
   COUNTER *pcounter)
```

Inputs

pname	A pointer to a null-terminated name string.
pcounter	A pointer to a variable in which to store the counter handle.

Description

The `KS_OpenCounter` service allocates, names, and obtains the handle of a dynamic counter. If a dynamic counter is available and there is no existing counter, static or dynamic, with a name matching the null-terminated string pointed to by *pname*, the service allocates a dynamic counter and applies the name referenced by *pname* to the new counter. The service stores the handle of the new dynamic counter in the variable pointed to by *pcounter*. The kernel stores only the address of the name internally, which means that the same array cannot be used to build multiple dynamic counter names.

If *pname* is null (`(char *)0`), the service does not assign a name to the dynamic counter. However, if *pname* points to a null string (`""`), the name is legal as long as no other counter is already using a null string as its name.

If the service finds an existing counter with a matching name, it does not open a new counter and returns a value indicating an unsuccessful operation.

Note: To use this service, you must enable the *Dynamics* attribute of the Counter class during system generation.

If the pointer to the counter name is not null (`(char *)0`), the time required to perform this operation varies with the number of counter names in use.

If the pointer to the counter name is null, no search of counter names takes place and the time to perform the service is fixed. You can define the counter name at a later time with a call to the `KS_DefCounterName` service.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully. The service also stores the handle of the allocated counter in the variable pointed to by *pcounter*.
- `RC_OBJECT_ALREADY_EXISTS` if the name search finds another counter whose name matches the specified string.
- `RC_NO_OBJECT_AVAILABLE` if the name search finds no match but all dynamic counters are in use.

Example

Example 6-13 attempts to allocate a dynamic counter and names it `Chnl2Counter`. If the name is already being used, the example outputs a message on the console.

Example 6-13. Allocate and Name Counter

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

KSRC ksrc;
COUNTER dyncounter;

if ((ksrc = KS_OpenCounter ("Chnl2Counter", &dyncounter))
    != RC_GOOD)
{
   if (ksrc == RC_OBJECT_ALREADY_EXISTS)
      putline ("Chnl2Counter counter name in use");
   else if (ksrc == RC_NO_OBJECT_AVAILABLE)
      putline ("No dynamic counters available");
   else
      putline ("Counters object class not defined");
}

... counter was opened correctly. Okay to use it now
```

See Also

`KS_CloseCounter`, page 192
`KS_LookupCounter`, page 214
`KS_UseCounter`, page 222

XX_SetCounterAcc

Set the accumulator of a counter to a specified value.

Zones

2 `TS_SetCounterAcc`
3 `KS_SetCounterAcc`

Synopsis

```
void XX_SetCounterAcc (COUNTER counter, TICKS ticks)
```

Inputs

counter	The handle of the counter to be read.
ticks	The value to store in the accumulator of the counter.

Description

The `XX_SetCounterAcc` service sets the specified *counter*'s tick accumulator to the value in *ticks*.

This service is useful for setting a counter to a specific count that has some significance in engineering units. For example, you can easily establish an accurate real-time clock with one-second accuracy. First, set up a counter that increments its tick accumulator once per second. Then use a function to convert the current date and time to the number of elapsed seconds since a standard date (most runtime libraries include a function for conversion of dates and time to Base Universal Time beginning 1-JAN-1970). Finally, set the 1-Hz counter's tick accumulator to the resulting number of seconds since the base date with the `XX_SetCounterAcc` service.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.
- `FE_UNINITIALIZED_COUNTER` if the specified counter has not yet been initialized.

Example

In Example 6-14 on page 219, the Current Thread reads the number of ticks in `COUNTER1` that have occurred since the thread's last execution cycle. When the counter's tick accumulator is read, the thread sets the counter's tick accumulator to zero.

Example 6-14. Set Counter Accumulator

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kcounter.h"     /* COUNTER1 */

TICKS nticks;

/* first, get the number of ticks since last time thread executed */
nticks = GetCounterAcc (COUNTER1);

/* then set counter accumulator to 0 */
TS_SetCounterAcc (COUNTER1, (TICKS)0);

...do something with nticks

... continue
```

See Also

`KS_GetCounterClassProp`, page 204

XX_SetCounterAttr

Set one or more attributes for a counter.

Zones

2 `TS_SetCounterAttr`
3 `KS_SetCounterAttr`

Synopsis

```
void XX_SetCounterAttr (COUNTER counter,
   KATTRMASK amask)
```

Inputs

counter	The handle of the counter containing the attributes to be cleared.
amask	A mask value containing the bits to set in the attribute property of the specified counter.

Description

The `XX_SetCounterAttr` service sets bits in the specified *counter*'s attribute property according to the bits specified in *amask*. For information about the Counter attributes, see "XX_DefCounterProp" on page 198.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.
- `FE_UNINITIALIZED_COUNTER` if the specified counter has not yet been initialized.

Example

In Example 6-15 on page 221, the Current Thread needs to disable the counter specified in `COUNTER1` to prevent further processing of events by that counter.

Example 6-15. Set Counter Attribute Bits

```
#include "rtxcapi.h"     /* RTXC Kernel Service prototypes */
#include "kcounter.h"    /* COUNTER1 */

/* disable COUNTER1 */
TS_SetCounterAttr (COUNTER1, ATTR_COUNTER_DISABLE);

... continue
```

See Also

`XX_ClearCounterAttr`, page 190

KS_UseCounter

Look up a dynamic counter by name and mark it for use.

Synopsis

```
KSRC KS_UseCounter (const char *pname,
   COUNTER *pcounter)
```

Inputs

pname	A pointer to a null-terminated name string.
pcounter	A pointer to a variable in which to store the counter handle.

Description

The `KS_UseCounter` service acquires the handle of a dynamic counter by looking up the null-terminated string pointed to by *pname* in the list of counter names. If there is a match with a dynamic counter, the service registers the counter for future use by the Current Task and stores that counter's handle in the variable pointed to by *pcounter*. This procedure allows the Current Task to reference the dynamic counter successfully in subsequent kernel service calls.

Note: To use this service, you must enable the *Dynamics* attribute of the Counter class during system generation.

The time required to perform this operation varies with the number of counter names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search is successful. The service also stores the matching counter's handle in the variable pointed to by *pcounter*.
- `RC_STATIC_OBJECT` if the specified name belongs to a static counter.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching counter name.

Example

Example 6-16 on page 223 locates a dynamic counter named `DynMuxCounter3` and obtains its handle for subsequent use.

Example 6-16. Read Counter Handle and Register It

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

KSRC ksrc;
COUNTER dyncounter;

if ((ksrc = KS_UseCounter ("DynMuxCounter3", &dyncounter))
    != RC_GOOD)
{
   if (ksrc == RC_STATIC_OBJECT)
      putline ("DynMuxCounter3 is a static counter");
   else
      putline ("Counter DynMuxCounter3 name not found");
}

... counter was found and its handle is in dyncounter.
    Okay to use it now
```

See Also

`XX_DefCounterProp`, page 198
`XX_ClearCounterAttr`, page 190
`KS_OpenCounter`, page 216

CHAPTER 7

Alarm Services

In This Chapter

We describe the Alarm kernel services in detail. The Alarm services create, arm, and start alarms as well as disarm and stop them. Alarms are related to Counters in that alarms utilize the tick accumulators of Counters to determine when an alarm reaches its point of expiration.

XX_AbortAlarm

Abort an active alarm.

Zones

2 `TS_AbortAlarm`
3 `KS_AbortAlarm`

Synopsis

```
TICKS XX_AbortAlarm (ALARM alarm)
```

Input

alarm	The handle for the alarm to be aborted.

Description

The `XX_AbortAlarm` kernel service stops the specified *alarm* and removes it from the list of active alarms on its associated counter, thereby making it inactive. If the alarm is already inactive, this service has no effect on it.

All tasks waiting for the expiration of the alarm as a result of a previous call to `KS_TestAlarmW` or `KS_TestAlarmT` become unblocked and these services return a `KSRC` value of `RC_ALARM_ABORTED`.

In addition, if there is a *Alarm_Abort* (`AA`) semaphore associated with the alarm, then the service signals the `AA` semaphore.

Output

If the user aborts an active alarm, the service returns the number of counter ticks remaining on the alarm.

If the alarm was inactive when stopped, the service ignores the request and returns a value of zero (0) for remaining ticks.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

In Example 7-1 on page 227, the Current Task starts the static alarm specified in `ALARM1`. The alarm uses the counter specified in `TIMEBASE` and has an initial period of 150 msec and a cyclic period of 100 msec. After starting the alarm, the task waits for the alarm to expire before starting its procedure. It then runs periodically every 100 msec for a total of five iterations, after which it stops the alarm and continues processing.

Example 7-1. Abort Alarm

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kalarm.h"       /* defines ALARM1 */

int i;

/* start alarm with 150 ms initial period & 100 ms cycle period */
KS_ArmAlarm (ALARM1);

/* wait for alarm to expire */
KS_TestAlarmW (ALARM1, (TICKS *)0);

for (i = 0; i < 5; i++)  /* processing loop of task */
{
   /* wait on alarm expiration but ignore time remaining */
   KS_TestAlarmW (ALARM1, (TICKS *)0);

   ... Execute loop procedure, then wait for the next loop time
}
/* kill the alarm and ignore time remaining */
KS_AbortAlarm (ALARM1);

... continue
```

See Also

`XX_ArmAlarm`, page 230
`KS_TestAlarmW`, page 268

INIT_AlarmClassProp

Initialize the Alarm object class properties.

Synopsis

```
KSRC INIT_AlarmClassProp
   (const KCLASSPROP *pclassprop)
```

Input

pclassprop	A pointer to a Alarm object class properties structure.

Description

During the RTXC initialization procedure, you must define the kernel objects needed by the kernel to perform the application. The `INIT_AlarmClassProp` kernel service allocates space for the Alarm object class in system RAM. The amount of RAM to allocate, and all other properties of the class, are specified in the `KCLASSPROP` structure pointed to by *pclassprop*.

Example 2-13 on page 44 shows the organization of the `KCLASSPROP` structure.

The *attributes* element of the Alarm `KCLASSPROP` structure supports the class property attributes and corresponding masks listed in Table 7-1 on page 246.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_NO_RAM` if the initialization fails because there is insufficient system RAM available.

Example

During system initialization, the startup code must initialize the Alarm object class before using any kernel service for that class. In Example 7-2 on page 229, the system generation process produced a `KCLASSPROP` structure containing the information about the kernel object necessary for its initialization. The example references that structure externally to the code module.

Example 7-2. Initialize Alarm Object Class

```
#include "rtxcapi.h"       /* RTXC Kernel Service prototypes */

static char buf[128];

extern const SYSPROP sysprop;
extern const KCLASSPROP alarmclassprop;

KSRC userinit (void)
{
   KSRC ksrc;

   /* initialize the kernel workspace and allocate RAM */
   /* for required classes, etc. */

   if ((ksrc = INIT_SysProp (&sysprop)) != RC_GOOD)
   {
      putline ("Kernel initialization failure");
      return (ksrc); /* end initialization process */
   }
   /* kernel is initialized */

   /* Need to initialize the necessary kernel */
   /* object classes */

   /* Initialize the Alarm kernel object class */
   if ((ksrc = INIT_AlarmClassProp (&alarmclassprop))
       != RC_GOOD)
   {
      putline ("No RAM for Alarm init");
      return (ksrc); /* end initialization process */
   }

... Continue with system initialization
}
```

See Also

`XX_CancelAlarm`, page 232

XX_ArmAlarm

Arm and start an alarm.

Zones

2 `TS_ArmAlarm`
3 `KS_ArmAlarm`

Synopsis

```
KSRC XX_ArmAlarm (ALARM alarm)
```

Input

alarm	The handle of the alarm to be armed and started.

Description

The `XX_ArmAlarm` service arms and starts the specified *alarm*. Before performing this service on the alarm, you should define, through a call to `XX_DefAlarmProp`, its associated counter and the initial and cyclic interval properties in ticks appropriate to that counter.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the alarm is successfully started.
- `RC_ALARM_ACTIVE` if the service attempts to start an active alarm.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

Example 7-3 on page 231 arms and starts the static alarm specified in `ALARM1`. The task first tests the alarm to insure that it is not already active. If not active, it arms and starts the alarm using its defined properties. The alarm is to be relative to the counter specified in `TIMEBASE` and have an initial period of 150 msec and a cyclic period of 100 msec. After starting the alarm, the task waits for the alarm to expire before starting its procedure. It then runs periodically every 100 msec.

Example 7-3. Arm Alarm

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kalarm.h"       /* defines ALARM1 */

static ALARMPROP alarmprop;

if (KS_TestAlarm (ALARM1, (TICKS *)0) == RC_ALARM_INACTIVE)
/* alarm is inactive. Okay to arm it and start it */
   KS_ArmAlarm (ALARM1);

/* alarm is active */

for (;;)  /* processing loop of task */
{
   /* wait for alarm to expire */
   KS_TestAlarmW (ALARM1, (TICKS *)0);

   ... Do some processing, then wait for alarm to begin next loop
}
```

See Also

`XX_DefAlarmProp`, page 242
`KS_TestAlarm`, page 262
`XX_RearmAlarm`, page 260

XX_CancelAlarm

Make an active alarm inactive.

Zones

2 `TS_CancelAlarm`
3 `KS_CancelAlarm`

Synopsis

```
TICKS XX_CancelAlarm (ALARM alarm)
```

Input

alarm	The handle for the alarm to be canceled.

Description

The `XX_CancelAlarm` service stops the specified *alarm* and removes it from the list of active alarms on its associated counter, thereby making it inactive. If *alarm* is already inactive, this service has no effect on it.

All tasks waiting for the expiration of the alarm as a result of a previous call to `KS_TestAlarmW` or `KS_TestAlarmT` become unblocked and these services return a `KSRC` value of `RC_ALARM_CANCELED`.

Output

If the user cancels an active alarm, this service returns the number of counter ticks remaining on the alarm.

If the alarm was inactive when stopped, the service ignores the request and returns a value of zero (0) for remaining ticks.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

In Example 7-4 on page 233, the Current Task cancels the static alarm specified in `ALARM1` after it has gone through five expiration events.

Example 7-4. Cancel Alarm

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kalarm.h"       /* defines ALARM1 */

int i;
TICKS residual;

/* start alarm with 150 ms initial period and 100 ms cycle period */
KS_ArmAlarm (ALARM1);

for (i = 0; i < 5; i++)  /* processing loop of task */
{
   /* wait on alarm expiration but ignore time remaining */
   KS_TestAlarmW (ALARM1, (TICKS *)0);

   ... Execute loop procedure, then wait for the next loop time
}
/* kill the alarm and ignore time remaining */
KS_CancelAlarm (ALARM1);

... continue
```

See Also

`XX_ArmAlarm`, page 230
`KS_TestAlarmW`, page 268

KS_CloseAlarm

End the use of a dynamic alarm.

Synopsis

```
KSRC KS_CloseAlarm (ALARM alarm)
```

Input

alarm	The handle for the alarm.

Description

The `KS_CloseAlarm` kernel service ends the Current Task's use of the specified dynamic *alarm*. When closing *alarm*, the kernel detaches the caller's use of it. If the caller is the last user of *alarm*, the alarm is released to the free pool of dynamic alarms for reuse. If there is at least one other task still using the alarm, the kernel does not release the alarm to the free pool but the service completes successfully.

Note: To use this service, you must enable the Dynamics attribute of the Alarm class during system generation.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service is successful.
- `RC_STATIC_OBJECT` if the specified alarm is not dynamic.
- `RC_OBJECT_NOT_INUSE` if the specified alarm does not correspond to an active dynamic alarm.
- `RC_OBJECT_INUSE` if the Current Task's use of the specified alarm is closed but the alarm remains open for use by other tasks.

Note: `RC_OBJECT_INUSE` does not necessarily indicate an error condition. The calling task must interpret its meaning.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.

Example

In Example 7-5, the Current Task waits on a signal from another task indicating that it should close a dynamic alarm. The handle of the dynamic semaphore associated with the signal is specified in `dynsema`. The handle of the dynamic alarm is specified in dynalarm. When the signal is received, the Current Task closes the prescribed dynamic alarm.

Example 7-5. Close Alarm

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

ALARM dynalarm;           /* dynamic alarm’s handle stored here */
SEMA dynsema;             /* dynamic sema’s handle stored here  */
KSRC ksrc;

KS_TestSemaW (dynsema);    /* wait for signal */

/* then close the alarm */
if ((ksrc = KS_CloseAlarm (dynalarm)) != RC_GOOD)
{
   /* Something may be wrong, deal with it here */
}
... dynamic alarm is closed, continue
```

See Also

`XX_ArmAlarm`, page 230

XX_DefAlarmAction

Define action to perform following an alarm expiration.

Zones

2 `TS_DefAlarmAction`
3 `KS_DefAlarmAction`

Synopsis

```
KSRC XX_DefAlarmAction (ALARM alarm,
   ALARMACTION action, THREAD thread)
```

Inputs

alarm The handle of the alarm to be associated with the end action operation.

action A code for the action to perform as follows:

- `SCHEDULETHREAD`—Schedule *thread* at the expiration of *alarm*.
- `DECRTHREADGATE`—Decrement the thread gate value of *thread* upon the expiration of *alarm*.

thread The handle of the thread on which to perform the end action operation.

Description

The `XX_DefAlarmAction` service defines the action to take following the expiration of the specified *alarm*. The `XX_ProcessEventSourceTick` determines when an alarm expires. When an expiration occurs, the `XX_ProcessEventSourceTick` service performs the specified end action operation, if defined, on the specified thread. If the source event processing is called from an ISR, the end action operation must perform `IS_ScheduleThread` or `IS_DecrThreadGate`, corresponding to the action codes `SCHEDULETHREAD` or `DECRTHREADGATE`, respectively. If a Zone 2 thread processes the source event and determines that an alarm has expired, the end action operation must perform `TS_ScheduleThread` or `TS_DecrThreadGate`, corresponding to the action code `SCHEDULETHREAD` or `DECRTHREADGATE`, respectively.

Output

This service returns a KSRC value as follows:

- RC_GOOD if the service is successful.
- RC_ALARM_ACTIVE if the service attempts to start an active alarm.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_ALARM if the specified alarm ID is not valid.
- FE_UNINITIALIZED_ALARM if the specified alarm has not yet been initialized.
- FE_ILLEGAL_THREAD if the specified thread ID is not valid.
- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.
- FE_INVALID_ALARMACTION if the specified alarm action value is not one of the four possible actions.

Example

In Example 7-6, the thread specified in THREADA needs to be scheduled every 5 seconds. The Current Thread defines a SCHEDULETHREAD action to take place on the expiration of the ALARM1 static alarm, which is a cyclic alarm. The Current Thread then arms and starts the alarm.

Example 7-6. Define Alarm End Action Operation

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */
#include "kalarm.h"       /* ALARM1 */

/* define alarm action on ALARM1 to schedule thread */
TS_DefAlarmAction (ALARM1, SCHEDULETHREAD, THREADA);

TS_ArmAlarm (ALARM1);

... continue
```

XX_DefAlarmActionArm

Define the action to perform when an alarm expires and then arm and start the alarm.

Zones

2 `TS_DefAlarmActionArm`
3 `KS_DefAlarmActionArm`

Synopsis

```
KSRC XX_DefAlarmActionArm (ALARM alarm,
   ALARMACTION action, THREAD thread)
```

Inputs

alarm	The handle of the alarm to be associated with the end action operation.
action	A code for the action to perform as follows: `SCHEDULETHREAD`—Schedule *thread* at the expiration of *alarm*. `DECRTHREADGATE`—Decrement the thread gate value of *thread* upon the expiration of alarm.
thread	The handle of the thread on which to perform the end action operation.

Description

The `XX_DefAlarmActionArm` service arms the specified *alarm* and defines the action to take following its expiration. The `XX_ProcessEventSourceTick` service determines when an alarm expires. When an expiration occurs, the `XX_ProcessEventSourceTick` service performs the specified end action operation, if defined, on the specified thread. If the source event processing is called from an interrupt service routine, the end action operation must perform `IS_ScheduleThread` or `IS_DecrThreadGate`, corresponding to the action code `SCHEDULETHREAD` or `DECRTHREADGATE`, respectively. If a Zone 2 thread processes the source event and determines that an alarm has expired, the end action operation must perform `TS_ScheduleThread` or `TS_DecrThreadGate`, corresponding to the action code `SCHEDULETHREAD` or `DECRTHREADGATE`, respectively.

Output

This service returns a KSRC value as follows:

- RC_GOOD if the service is successful.
- RC_ALARM_ACTIVE if the service attempts to start an active alarm.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_ALARM if the specified alarm ID is not valid.
- FE_UNINITIALIZED_ALARM if the specified alarm has not yet been initialized.
- FE_ILLEGAL_THREAD if the specified thread ID is not valid.
- FE_UNINITIALIZED_THREAD if the specified thread has not yet been initialized.
- FE_INVALID_ALARMACTION if the specified alarm action value is not one of the four possible actions.

Example

In Example 7-7, the thread specified in THREADA needs to be scheduled every 5 seconds. The Current Thread defines a SCHEDULETHREAD action on the ALARM1 static alarm, which is a cyclic alarm. The alarm is then armed and started automatically.

Example 7-7. Arm Alarm and Define Alarm Expiration Action Operation

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kthread.h"      /* THREADA */
#include "kalarm.h"       /* ALARM1 */

/* define alarm action on ALARM1 to schedule a thread from an ISR */
TS_DefAlarmActionArm (ALARM1, SCHEDULETHREAD, THREADA);

... continue
```

KS_DefAlarmName

Define the name of a previously opened alarm.

Synopsis

```
KSRC KS_DefAlarmName (ALARM alarm,
   const char *pname)
```

Inputs

alarm	The handle of the alarm being defined.
pname	A pointer to a null-terminated name string.

Description

The `KS_DefAlarmName` kernel service names or renames the specified dynamic *alarm*. The service uses the null-terminated string pointed to by *pname* for the alarm's new name.

Static alarms cannot be named or renamed under program control.

Note: To use this service, you must enable the Dynamics attribute of the Alarm class during system generation.

This service does not check for duplicate alarm names.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_STATIC_OBJECT` if the alarm being named is static.
- `RC_OBJECT_NOT_FOUND` if the Dynamics attribute of the Alarm class is not enabled.
- `RC_OBJECT_NOT_INUSE` if the dynamic alarm being named does not correspond to an open dynamic alarm.

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.

Example

Example 7-8 assigns the name `NewAlarm` to the previously opened dynamic alarm specified in `dynalarm` so other users may reference it by name.

Example 7-8. Define Alarm Name

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

ALARM dynalarm;

if (KS_DefAlarmName (dynalarm, "NewAlarm") != RC_GOOD)
{
   ... Something may be wrong. Deal with it here
}
... naming operation was successful. Continue
```

See Also

`KS_OpenAlarm`, page 258
`KS_GetAlarmName`, page 248
`KS_LookupAlarm`, page 256
`KS_UseAlarm`, page 270

XX_DefAlarmProp

Define the properties of a alarm.

Zones

2 `TS_DefAlarmProp`
3 `KS_DefAlarmProp`

Synopsis

```
void XX_DefAlarmProp (ALARM alarm,
   const ALARMPROP *palarmprop)
```

Inputs

alarm	The handle of the alarm being defined.
palarmprop	A pointer to an Alarm properties structure.

Description

The `XX_DefAlarmProp` kernel service defines the properties of the specified *alarm* using the values contained in the `ALARMPROP` structure pointed to by *palarmprop*.

Example 7-9 shows the organization of the `ALARMPROP` structure.

Example 7-9. Alarm Properties Structure

```
typedef struct
{
   KATTR attributes;     /* alarm attributes */
   COUNTER counter;      /* counter associated with alarm */
   TICKS initial;        /* initial count period */
   TICKS recycle;        /* recycle count period */
} ALARMPROP;
```

The alarm attributes value is reserved for future use. The counter property specifies the counter the system will use to determine alarm expiration. The alarm's initial ticks value is specified in initial. The cyclic value is specified in recycle.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_ILLEGAL_COUNTER` if the specified counter ID is not valid.

Example

In Example 7-10, the Current Task defines the properties of the previously opened dynamic alarm specified in `dynalarm`. The attributes element is set to zero (0). The alarm uses the `TIMEBASE` counter, which is the counter for the system timebase. The duration of the alarm's initial period is 500 ms and the cyclic period is 200 ms. After the task defines the alarm's properties, it uses the alarm to time some processing on a periodic basis.

Example 7-10. Define Alarm Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kproject.h"     /* defines CLKTICK */

ALARM dynalarm;
static ALARMPROP alarmprop;

alarmprop.attributes = 0;
alarmprop.counter = TIMEBASE;
alarmprop.initial = (TICKS)500 / CLKTICK;
alarmprop.cycle = (TICKS)200 / CLKTICK;

KS_DefAlarmProp (dynalarm, &alarmprop);
KS_ArmAlarm (dynalarm);    /* start alarm now */

for (;;)
{
   /* wait for alarm to expire */
   KS_TestAlarmW (dynalarm, (TICKS *)0);

   ... perform some process, then wait for next period
}
```

See Also

`XX_GetAlarmProp`, page 250
`XX_GetAlarmTicks`, page 254
`KS_OpenAlarm`, page 258
`KS_TestAlarm`, page 262

KS_DefAlarmSema

Associate a semaphore with a alarm event.

Synopsis

```
void KS_DefAlarmSema (ALARM alarm, SEMA sema,
   AEVENT event)
```

Inputs

alarm	The handle of the alarm with which to associate the semaphore.
sema	The handle of the semaphore to associate with the alarm event.
event	An alarm event value.

Description

The `KS_DefAlarmSema` service associates the semaphore specified in *sema* with an *event*, either Alarm_Expired (AE) or Alarm_Aborted (AA), of the specified *alarm*.

The Alarm_Expired and Alarm_Aborted events have enumerated values of AE and AA, respectively. You should use one of these values when specifying the event argument.

Note: To use this service, you must enable the Semaphores attribute of the Alarm class during system generation.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.
- `FE_ILLEGAL_SEMA` if the specified semaphore ID is not valid.
- `FE_UNINITIALIZED_SEMA` if the specified semaphore has not yet been initialized.

- FE_INVALID_ALARMEVENT if the specified semaphore event value is not either AE or AA.

Example

In Example 7-11, the Current Task needs to know when either of two events occurs. The SWITCH1 event is associated with a switch closure. The task uses KS_DefAlarmSema to associate the ALARMXP semaphore with the Alarm_Expired (AE) event. Then the task waits for either event.

Example 7-11. Define Alarm Semaphore

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "ksema.h"        /* defines SWITCH1, ALARMXP */
#include "kalarm.h"       /* defines ALARM1 */

SEMA cause;
const SEMA semalist[] =
{
   SWITCH1,
   ALARMXP,
   (SEMA)0             /* null terminated list */
};

/* associate ALARMXP with the expiration of ALARM1 */
KS_DefAlarmSema (ALARM1, ALARMXP, AE);

for (;;)
{
   /* wait for either of 2 events */
   cause = KS_TestSemaMW (semalist);

   switch (cause)
   {
      case SWITCH1:
         ... process SWITCH1 event...
         break;

      case ALARMXP:
         ... process ALARMXP event...
         break;

   }  /* end of switch */
}  /* end of forever */
```

See Also

KS_GetAlarmSema, page 252

KS_GetAlarmClassProp

Get the Alarm object class properties.

Synopsis

```
const KCLASSPROP * KS_GetAlarmClassProp (int *pint)
```

Input

pint	A pointer to a variable in which to store the number of available dynamic alarms.

Description

The `KS_GetAlarmClassProp` kernel service obtains a pointer to the `KCLASSPROP` structure that was used during system initialization by the `INIT_AlarmClassProp` service to initialize the Alarm object class properties.

If *pint* contains a non-zero address, the service stores the current number of unused dynamic alarms in the indicated address. If *pint* contains a null pointer (`(int *)0`), the service ignores the parameter. If the Alarm object class properties do not include the *Dynamics* attribute, the service stores a value of zero (0) at the address contained in *pint*.

Example 2-13 on page 44 shows the organization of the `KCLASSPROP` structure.

The attributes element of the Alarm `KCLASSPROP` structure supports the class property attributes and corresponding masks listed in Table 7-1.

Table 7-1. Alarm Class Attributes and Masks

Attribute	Mask
Static Names	`ATTR_STATIC_NAMES`
Dynamics	`ATTR_DYNAMICS`
Semaphores	`ATTR_SEMAPHORES`

Output

If successful, this service returns a pointer to a `KCLASSPROP` structure.

If the Alarm class is not initialized, the service returns a null pointer `((KCLASSPROP *)0)`.

Example

Example 7-12 accesses the information contained in the `KCLASSPROP` structure for the Alarm object class.

Example 7-12. Read Alarm Object Class Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

KCLASSPROP *palarmclassprop;
int free_dyn;

/* Get the Alarm kernel object class properties */
if ((palarmclassprop = KS_GetAlarmClassProp (&free_dyn))
    == (KCLASSPROP *)0)
{
   putline ("Alarm Class not initialized");
}
else
{
   ... alarm object class info is available for use
       "free_dyn" contains the number of available dynamic alarms
}
```

See Also

`XX_GetAlarmTicks`, page 254

KS_GetAlarmName

Get the name of a alarm.

Synopsis

`char * KS_GetAlarmName (ALARM alarm)`

Input

alarm	The handle of the alarm being queried.

Description

The `KS_GetAlarmName` kernel service obtains a pointer to the null-terminated string containing the name of the specified static or dynamic *alarm*.

Output

If *alarm* has a name, this service returns a pointer to the null-terminated name string.

If *alarm* has no name, the service returns a null pointer (`(char *)0`).

Error

This service may generate the following fatal error code:

`FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.

Example

Example 7-13 reports the name of the dynamic alarm specified in `dynalarm`.

Example 7-13. Read Alarm Name

```
#include <stdio.h>        /* standard i/o */
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */

static char buf[128];
ALARM dynalarm;
char *pname;

if ((pname = KS_GetAlarmName (dynalarm)) == (char *)0)
   sprintf (buf, "Alarm %d has no name", dynalarm);
else
   sprintf (buf, "Alarm %d name is %s", dynalarm, pname);

putline (buf); /* send buffer to console */
```

See Also

`KS_DefAlarmName`, page 240
`KS_OpenAlarm`, page 258

XX_GetAlarmProp

Get the properties of a alarm.

Zones

2 `TS_GetAlarmProp`
3 `KS_GetAlarmProp`

Synopsis

```
void XX_GetAlarmProp (ALARM alarm,
   ALARMPROP *palarmprop)
```

Inputs

alarm	The handle of the alarm being queried.
palarmprop	A pointer to a Alarm properties structure.

Description

The `XX_GetAlarmProp` kernel service obtains all of the property values of the specified *alarm* in a single call. The service stores the property values in the `ALARMPROP` structure pointed to by *palarmprop*.

Example 7-9 on page 242 shows the organization of the `ALARMPROP` structure.

The alarm attributes value is reserved for future use. The *counter* property specifies the counter the system uses to determine alarm expiration. The alarm's initial ticks value is specified in *initial*. The cyclic value is specified in *recycle*.

Output

This service does not return a value.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

Example 7-14 on page 251 changes the cyclic period value of the dynamic alarm specified in `dynalarm` to 150 ms. The task first obtains the alarm's current properties then modifies the cyclic period element in the `ALARMPROP` structure. `XX_DefAlarmProp` is then used to redefine the properties of the alarm.

Example 7-14. Read Alarm Properties

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kproject.h"     /* defines CLKTICK */

ALARM dynalarm;
static ALARMPROP alarmprop;

/* get the current alarm properties */
KS_GetAlarmProp (dynalarm, &alarmprop);

/* modify just the cyclic period element */
alarmprop.recycle = (TICKS)150/CLKTICK;

/* define the new alarm properties */
KS_DefAlarmProp (dynalarm, &alarmprop);
```

See Also

`XX_DefAlarmProp`, page 242

KS_GetAlarmSema

Get the handle of the semaphore associated with a alarm event.

Synopsis

```
SEMA KS_GetAlarmSema (ALARM alarm, AEVENT event)
```

Inputs

alarm	The handle of the alarm being queried.
event	A alarm event value.

Description

The `KS_GetAlarmSema` kernel service obtains the handle of the semaphore associated with the alarm event for the specified static or dynamic *alarm*. The two possible alarm events are Alarm_Expired (`AE`) orAlarm_Aborted (`AA`) and the value of *event* must be either `AE` or `AA`.

You must have previously associated the semaphore and the alarm event through a call to `KS_DefAlarmSema`.

Note: To use this service, you must enable the Semaphores attribute of the Alarm class during system generation.

Output

If the alarm event and semaphore association exists, this service returns the handle of the semaphore as a `SEMA` type value.

If there is no such association for the alarm event, the service returns a `SEMA` value of zero (0).

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.
- `FE_INVALID_ALARMEVENT` if the specified semaphore event value is not either `AE` or `AA`.

Example

In Example 7-15 on page 253, the Current Task needs to know the handle of the semaphore associated with the specified alarm so it can initialize the semalist semaphore list for use in a multiple event wait kernel service request.

Example 7-15. Read Alarm Semaphore

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "ksema.h"        /* defines SEMA2, SEMA3 */

ALARM alarm;
SEMA cause;
static SEMA semalist[] =
{
   (SEMA)0,   /* to be filled in below */
   SEMA2,
   SEMA3,
   (SEMA)0;   /* null terminated list */
};
semalist[0] = KS_GetAlarmSema (alarm, AE);

/* got sema handle, wait on events */
cause = KS_TestSemaMW (semalist);
switch (cause)
{
   case SEMA2:                 /* test for SEMA2 */
      ... handle this case
      break;
   case SEMA3:                 /* test for SEMA3 */
      ... handle this case
      break;
   default:               /* test for alarm expired   */
      /* has to be this way because case arg must be a constant */
      if (cause == semalist[0])
      {
         ... handle this case
      }
      break;
}
... continue
```

See Also

KS_DefAlarmSema, page 244

XX_GetAlarmTicks

Get the number of counter ticks remaining until the expiration of an active alarm.

Zones

2 `TS_AbortAlarm`
3 `KS_AbortAlarm`

Synopsis

```
TICKS XX_GetAlarmTicks (ALARM alarm)
```

Input

alarm	The handle of the alarm being queried.

Description

The `XX_GetAlarmTicks` service obtains the number of counter ticks remaining on the specified *alarm* until it expires. The alarm must be active. This service does not affect the operation of the alarm.

Output

If *alarm* is active, this service returns a value of type `TICKS` containing the number of ticks remaining on the alarm.

If *alarm* is inactive, the service returns a `TICKS` value of zero (0).

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

In Example 7-16 on page 255, the Current Task needs to know how many ticks the static alarm, `ALARM1`, has to go before it expires.

Example 7-16. Read Number of Counter Ticks Remaining on Alarm

```
#include "rtxcapi.h"       /* RTXC Kernel Service prototypes */
#include "kalarm.h"        /* defines ALARM1 */

TICKS residual;

residual = KS_GetAlarmTicks (ALARM1);

...do something with the residual counter tick value

... continue
```

See Also

`KS_DefAlarmSema`, page 244

KS_LookupAlarm

Look up a alarm's name to get its handle.

Synopsis

```
KSRC KS_LookupAlarm (const char *pname,
   ALARM *palarm)
```

Inputs

pname	A pointer to a null-terminated name string.
palarm	A pointer to a variable in which to store the matching alarm's handle.

Description

The `KS_LookupAlarm` kernel service obtains the handle of a static or dynamic alarm whose name matches the null-terminated string pointed to by *pname*. The lookup process terminates when it finds a match between the specified string and a static or dynamic alarm name or when it finds no match. The service also stores the matching alarm's handle in the variable pointed to by *palarm*. The service searches dynamic names, if any, first.

Note: To use this service on a static alarm, you must enable the Static Names attribute of the Alarm class during system generation.

This service has no effect on the use registration of the specified alarm by the Current Task.

The time required to perform this operation varies with the number of alarm names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search succeeds. The service also stores the matching alarm's handle in the variable pointed to by palarm.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching alarm name.

Example

Example 7-17 looks for the dynamic alarm named `Chnl2Alarm`. If the alarm is found, the example sends its handle to the console.

Example 7-17. Look Up Alarm by Name

```
#include <stdio.h>          /* standard i/o */
#include "rtxcapi.h"        /* RTXC Kernel Service prototypes */

static char buf[128];       /* output buffer */

ALARM dynalarm;

/* lookup the semaphore name to see if it exists */
if (KS_LookupAlarm ("Chnl2Alarm", &dynalarm) != RC_GOOD)
{
   putline ("Alarm Chnl2Alarm not found");
}
else
{
   /* alarm exists, output its handle */
   sprintf (buf, "Chnl2Alarm is alarm %d", dynalarm);
   putline (buf);
}
... continue
```

See Also

`KS_DefAlarmName`, page 240
`KS_OpenAlarm`, page 258

KS_OpenAlarm

Allocate and name a dynamic alarm.

Synopsis

```
KSRC KS_OpenAlarm (const char *pname, ALARM *palarm)
```

Inputs

pname	A pointer to a null-terminated name string.
palarm	A pointer to a variable in which to store the allocated alarm's handle.

Description

If a dynamic alarm is available and no existing alarm, static or dynamic, has a name matching the null-terminated string pointed to by *pname*, the `KS_OpenAlarm` kernel service allocates a dynamic alarm and applies the name to the new alarm. The kernel stores only the address of the name internally, which means that the same array cannot be used to build multiple dynamic alarm names. The service stores the alarm's handle in the variable pointed to by *palarm*.

If *pname* is a null pointer (`(char *)0`), the service does not assign a name to the dynamic alarm. However, if *pname* points to a null string, the name is legal as long as no other alarm is already using a null string as its name.

If the service finds an existing alarm with a matching name, it does not open a new alarm and returns a value indicating the failure.

Note: To use this service, you must enable the Dynamics attribute of the Alarm class during system generation.

If the pointer to the alarm name is not null (`(char *)0`), the time required to perform this operation is determined by the number of alarm names in use. If the pointer to the alarm name is null, no search of alarm names takes place and the time to perform the service is fixed. You can define the alarm name at a later time with a call to the `KS_DefAlarmName` service.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the service completes successfully. The service stores the handle of the new dynamic alarm in the variable pointed to by *palarm*.
- `RC_OBJECT_ALREADY_EXISTS` if the name search finds another alarm whose name matches the given string.
- `RC_NO_OBJECT_AVAILABLE` if the name search finds no match but all dynamic alarms are in use.

Example

Example 7-18 allocates a dynamic alarm and names it `MuxChnl2Alarm`. If the name is found to be in use or if there are no dynamic alarms available, the example sends an appropriate message to the console.

Example 7-18. Allocate and Name Alarm

```
#include "rtxcapi.h"       /* RTXC Kernel Service prototypes */

KSRC ksrc;
ALARM dynalarm;

if ((ksrc = KS_OpenAlarm ("MuxChnl2Alarm", &dynalarm))
    != RC_GOOD)
{
   if (ksrc == RC_OBJECT_ALREADY_EXISTS)
      putline ("MuxChnl2Alarm alarm name in use");
   else if (ksrc == RC_NO_OBJECT_AVAILABLE)
      putline ("No dynamic alarms available");
   else
      putline ("Alarms are not a defined object class");
}
else
{
   ... alarm was opened correctly. Okay to use it now
}
```

See Also

`XX_ArmAlarm`, page 230
`KS_LookupAlarm`, page 256
`KS_UseAlarm`, page 270

XX_RearmAlarm

Rearm and restart an active alarm.

Zone

2 TS_RearmAlarm
3 KS_RearmAlarm

Synopsis

```
TICKS XX_RearmAlarm (ALARM alarm, TICKS newinitial,
   TICKS newcycle)
```

Inputs

alarm	The handle of the alarm to be rearmed and restarted.
newinitial	A value of type `TICKS` to be used as the new initial tick interval for the alarm.
newcyclel	A value of type `TICKS` to be used as the new recycle tick interval for the alarm.

Description

The `XX_RearmAlarm` kernel service changes the initial period, cyclic period, or both, of the specified *alarm*. If the alarm is inactive, this service is equivalent to a call to `XX_DefAlarmProp` followed by a call to `XX_ArmAlarm`. If the alarm is active when this request is made, the service disarms and stops the alarm and then rearms and restarts it with the new properties given by *newinitial* and *newcycle*. If the alarm is active, the service returns the number of counter ticks remaining on the alarm at the point of its disarming.

This service does not change the status of any task waiting for the expiration of the alarm or on either of the alarm event semaphores.

Output

If the alarm is active, this service returns the number of counter ticks remaining on the alarm when the service is called.

If the alarm is inactive, the service returns zero (0).

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

Example 7-19 illustrates a re-triggerable watchdog alarm where the Current Task, having previously opened a dynamic alarm, uses it with the `TIMEBASE` counter as a 250-msec one-shot alarm. When the task completes its processing, it rearms and restarts the alarm with the same initial period duration. Presumably, some other task is waiting on the expiration event should it occur.

Example 7-19. Rearm and Restart Alarm from Zone 3

```
#include "rtxcapi.h"          /* RTXC Kernel Service prototypes */
#include "kproject.h"         /* defines CLKTICK */

ALARM dynalarm;
static ALARMPROP alarmprop;

/* allocate a dynamic alarm for WDT, (name not important) */
if (KS_OpenAlarm ((char *)0, &dynalarm) != RC_GOOD)
{
   ... Deal with failure to open alarm
}

/* define the properties for a 250 msec one shot alarm */
alarmprop.attributes = 0;
alarmprop.counter = TIMEBASE;
alarmprop.initial = (TICKS)250/CLKTICK;
alarmprop.recycle = (TICKS)0;
KS_DefAlarmProp (dynalarm, &alarmprop);

/* start the alarm
if (KS_ArmAlarm (dynalarm) == RC_GOOD)
{
... WDT started. Do some processing
}

/* then restart the WDT as a 250 msec one-shot */
if (KS_RearmAlarm (dynalarm, (TICKS)250/CLKTICK, (TICKS)0) ==
    (TICKS)0)
{
   ...alarm had expired, may need to deal with that
}
else
   ...alarm not expired. Continue processing
```

See Also

`XX_AbortAlarm`, page 226
`XX_DefAlarmProp`, page 242
`XX_ArmAlarm`, page 230

KS_TestAlarm

Get the time, in ticks, remaining on an active alarm.

Synopsis

```
KSRC KS_TestAlarm (ALARM alarm, TICKS *pticks)
```

Inputs

alarm	The handle of the alarm being tested.
pticks	A pointer to a variable in which to store the number of ticks remaining on the alarm.

Description

The `KS_TestAlarm` kernel service tests the specified *alarm* and obtains the time remaining on it if it is active. The service puts the time remaining into the variable pointed to by *pticks*.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the alarm is active.
- `RC_ALARM_INACTIVE` if the alarm is not active.

If *alarm* is active and *pticks* is not null (`(TICKS *)0`), the service returns the number of ticks remaining on the alarm in the variable pointed to by *pticks*.

If *alarm* is not active and *pticks* is not null (`(TICKS *)0`), the service stores a value of zero (0) in the variable pointed to by *pticks*.

If *pticks* is null, the service ignores it and does not use it as a destination pointer.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

Example 7-20 on page 263 opens a dynamic alarm and defines the properties for a 500-msec, one-shot alarm. The task then associates the `TMRSEMA` semaphore with the expiration of the alarm and waits on `TMRSEMA` and another event associated with the `INTSEMA` semaphore. When either event occurs, the task tests the alarm and loads the remainder variable with the time remaining on the alarm. If the event associated with INTSEMA occurs, the task obtains the remaining time and stops the alarm. If the event was the alarm expiration, the value of remainder is zero (0).

Example 7-20. Test Alarm

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kproject.h"     /* defines CLKTICK */
#include "ksema.h"        /* defines INTSEMA & TMRSEMA */

ALARM dynalarm;
static ALARMPROP alarmprop;
TICKS remainder;
SEMA sema;
const SEMA semalist[] = { INTSEMA, TMRSEMA, (SEMA)0 };

/* allocate a dynamic alarm, name is unimportant */
if (KS_OpenAlarm ((char *)0, &dynalarm) != RC_GOOD)
{
   ... no alarms available if here
}
/* define the properties for a 500 msec one-shot alarm */
alarmprop.attributes = 0;
alarmprop.counter = TIMEBASE;
alarmprop.initial = (TICKS)500/CLKTICK;
alarmprop.cycle = (TICKS)0;
KS_DefAlarmProp (dynalarm, &alarmprop);

/* associate semaphore TMRSEMA with alarm expiration */
KS_DefAlarmSema (dynalarm, TMRSEMA);

/* start the allocated alarm and wait for the event or the alarm */
KS_ArmAlarm (dynalarm);
KS_TestSemaMW (semalist);/* disregard the returned sema */

/* test alarm to see if INTSEMA event occurred*/
if (KS_TestAlarm (dynalarm, &remainder) == RC_GOOD)
   KS_AbortAlarm (dynalarm); /* stop the alarm */
/* otherwise, alarm elapsed before event occurred

/* at this point both semaphores are back in a PENDING */
/* state and the alarm is in an INACTIVE state. */

... now do something with the remaining time
```

See Also

XX_AbortAlarm, page 226
XX_DefAlarmProp, page 242
KS_DefAlarmSema, page 244
KS_OpenAlarm, page 258
XX_ArmAlarm, page 230

KS_TestAlarmT

Wait a specified number of ticks for an alarm to expire.

Synopsis

```
KSRC KS_TestAlarmT (ALARM alarm, TICKS *pticks,
   COUNTER counter, TICKS tickout)
```

Inputs

alarm	The handle of the alarm being tested.
pticks	A pointer to a variable in which to store the number of ticks remaining on the alarm being tested.
counter	The handle of a counter to use for the internal alarm of duration ticks.
tickout	The number of ticks for the internal alarm on counter to wait for the expiration of the alarm being tested.

Description

The `KS_TestAlarmT` service waits for the expiration of the specified active *alarm*. When the service determines that *alarm* is active, the service starts an internal tickout alarm for the duration specified in *tickout* on the specified *counter,* and then blocks the Current Task. If *pticks* is not null (`(TICKS *)0`), the service returns the number of ticks remaining on the alarm in the variable pointed to by *pticks* when the task resumes.

The Current Task remains blocked until one of three events occurs.

- The alarm being tested expires.
- The specified number of ticks elapses.
- The alarm being tested is aborted.

When an alarm is armed, it may be aborted by another task. If so, the internal tickout alarm is stopped and the task waiting on the alarm being tested resumes and the `KS_TestAlarmT` service returns an indicator that the alarm was aborted.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the specified alarm being tested expires before the internal alarm expires. The service stores the number of ticks remaining on the alarm as a value of zero (0) at the address in pticks.

- RC_ALARM_INACTIVE if the specified alarm is not active when the service is called. In this case, the service returns immediately. The service returns a value of zero (0) for the remaining number of ticks on the alarm
- RC_ALARM_ABORTED if another task aborts the alarm being tested through the use of XX_AbortAlarm before the internal alarm expires. If so, the service stores, in the variable pointed to by pticks, the number of ticks remaining on the alarm being tested.
- RC_TICKOUT if the specified number of ticks elapses before the expiration of the specified alarm. In this case, the service returns the number of ticks remaining on the specified alarm at the address pointed to by pticks.

Errors

This service may generate one of the following fatal error codes:

- FE_ILLEGAL_ALARM if the specified alarm ID is not valid.
- FE_UNINITIALIZED_ALARM if the specified alarm has not yet been initialized.
- FE_ILLEGAL_COUNTER if the specified counter ID is not valid.
- FE_UNINITIALIZED_COUNTER if the specified counter has not yet been initialized.

Example

Example 7-21 on page 267 needs to synchronize with static alarm, ALARM1, started by another task, but sets up an internal tickout alarm of 50 msec to achieve synchronization. If the internal tickout alarm occurs before synchronizing with ALARM1, the task tries to sync up again. If ALARM1 is inactive or is aborted, the task takes special action.

Example 7-21. Test Alarm—Wait Number of Ticks for Expiration

```
#include "rtxcapi.h"       /* RTXC Kernel Service prototypes */
#include "kproject.h"      /* defines CLKTICK */
#include "kalarm.h"        /* defines ALARM1 */

KSRC retcode;

/* wait 50 msec for alarm to expire*/
while ((retcode = KS_TestAlarmT (ALARM1, (TICKS *)0, TIMEBASE,
       (TICKS)50/CLKTICK)) == RC_TICKOUT)
{
   ... No sync yet because timeout occurred.
       Do something useful
}

if (retcode != RC_GOOD)
{
   ... Either alarm was inactive or was aborted.
       Deal with it
}
else
{
   ... Alarm expired, Current Task is now in synch
       with ALARM1
}
```

See Also

`XX_AbortAlarm`, page 226
`XX_ArmAlarm`, page 230

KS_TestAlarmW

Wait for a alarm to expire.

Synopsis

```
KSRC KS_TestAlarmW (ALARM alarm, TICKS *pticks)
```

Inputs

alarm	The handle of the alarm being tested.
pticks	A pointer to a variable in which to store the number of ticks remaining on the alarm if aborted by another task or thread.

Description

The `KS_TestAlarmW` service waits for the expiration of the specified active *alarm*. The service blocks the requesting task until the expiration of the specified alarm. However, another task or thread may stop the alarm through the use of `XX_CancelAlarm` or `XX_AbortAlarm` and cause a premature resumption of the waiting task. In this case, the service stores the number of ticks remaining on the alarm at the point of being stopped in the variable pointed to by *pticks*, if *pticks* is not null `((TICKS *)0)`. If *pticks* is null, the service does not return the number of remaining ticks.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the alarm expires normally. The service returns zero (0) for the remaining ticks in the variable pointed to by *pticks*.
- `RC_ALARM_INACTIVE` if the alarm is inactive at the time of the service request. The service does not block the calling task and returns immediately, storing a value of zero (0) in the variable pointed to by *pticks*.
- `RC_ALARM_ABORTED` if another task aborts the alarm through the use of the `XX_AbortAlarm` kernel service. If this occurs, the number of ticks remaining on the alarm when aborted is stored in the variable pointed to by *pticks*.
- `RC_ALARM_CANCELLED` if another task stops the alarm through the use of the `XX_CancelAlarm` kernel service. If this occurs, the service stores the number of ticks remaining on the alarm when aborted in the variable pointed to by *pticks*.

Errors

This service may generate one of the following fatal error codes:

- `FE_ILLEGAL_ALARM` if the specified alarm ID is not valid.
- `FE_UNINITIALIZED_ALARM` if the specified alarm has not yet been initialized.

Example

Example 7-22 needs to generate a report periodically. It opens a dynamic alarm, defines the properties for a cyclic alarm, and starts the alarm using counter `TIMEBASE`. The report period is 30 seconds and the report is generated each time the alarm expires.

Example 7-22. Test Alarm—Wait for Expiration

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kproject.h"     /* defines CLKTICK */

ALARM dynalarm;
static ALARMPROP alarmprop;

/* open a dynamic alarm, name is unimportant */
if (KS_OpenAlarm ((char *)0, &dynalarm) != RC_GOOD)
{
   ... no alarms available. Deal with it here
}

/* define the properties for a 30 second cyclic alarm */
alarmprop.attributes = 0;
alarmprop.counter = TIMEBASE;
alarmprop.initial = (TICKS)30000/CLKTICK;
alarmprop.cycle = (TICKS)30000/CLKTICK;
KS_DefAlarmProp (dynalarm, &alarmprop);

/* start the alarm
KS_ArmAlarm (dynalarm);

for (;;)
{
   /* wait for the report period */
   KS_TestAlarmW (dynalarm, (TICKS *)0);

   ...generate periodic report
}
```

See Also

`XX_AbortAlarm`, page 226
`XX_ArmAlarm`, page 230

KS_UseAlarm

Look up a dynamic alarm by name and mark it for use.

Synopsis

```
KSRC KS_UseAlarm (const char *pname, ALARM *palarm)
```

Inputs

pname	A pointer to a null-terminated name string.
palarm	A pointer to a variable in which to store the matching alarm's handle.

Description

The `KS_UseAlarm` kernel service acquires the handle of a dynamic alarm by looking up the null-terminated string pointed to by *pname* in the list of alarm names. If there is a match, the service registers the alarm for future use by the Current Task and stores the matching alarm's handle in the variable pointed to by *palarm*. This procedure allows the Current Task to reference the dynamic alarm successfully in subsequent kernel service calls.

Note: To use this service, you must enable the Dynamics attribute of the Alarm class during system generation.

The time required to perform this operation varies with the number of alarm names in use.

Output

This service returns a `KSRC` value as follows:

- `RC_GOOD` if the search is successful. The service stores the matching alarm's handle in the variable pointed to by palarm.
- `RC_STATIC_OBJECT` if the given name belongs to a static alarm.
- `RC_OBJECT_NOT_FOUND` if the service finds no matching name.

Example

Example 7-23 on page 271 locates a dynamic alarm named `DynMuxAlarm3` and obtains its handle. After the handle is known, the task starts the alarm as a one-shot having an initial period duration of 500 milliseconds. The task sends a message to the console indicating the action taken.

Example 7-23. Read Alarm Handle and Register It

```
#include "rtxcapi.h"      /* RTXC Kernel Service prototypes */
#include "kproject.h"     /* defines CLKTICK */

KSRC ksrc;
ALARM dynalarm;
static ALARMPROP alarmprop;

if ((ksrc = KS_UseAlarm ("DynMuxAlarm3", &dynalarm)) != RC_GOOD)
{
   if (ksrc == RC_STATIC_OBJECT)
      putline ("DynMuxAlarm3 is a static alarm");
   else
      putline ("Alarm DynMuxAlarm3 not found");
}
else
{
   /* alarm was found and its handle is in dynalarm */.
   if (KS_TestAlarm (dynalarm, (TICKS *)0) != RC_GOOD)
   {
      /* alarm is not active, ok to use it */
      /* define the properties for a 500 msec alarm */
      alarmprop.attribute = 0;
      alarmprop.counter = TIMEBASE;
      alarmprop.initial = (TICKS)500/CLKTICK;
      alarmprop.cycle = (TICKS)0;
      KS_DefAlarmProp (dynalarm, &alarmprop);

      /* now start the alarm */
      KS_ArmAlarm (dynalarm);
      putline ("Alarm DynMuxAlarm3 is started");
      ... alarm started, do whatever is required
   }
   else
   {
      putline ("Alarm DynMuxAlarm3 is already active");
      ... alarm was already active, deal with that here
   }
}
```

See Also

XX_DefAlarmProp, page 242
KS_DefAlarmName, page 240
KS_OpenAlarm, page 258
KS_TestAlarm, page 262

CHAPTER 8

Special Services

In This Chapter

We describe the Special kernel services in detail. The Special services provide for user-defined extensions to the **RTXC** Kernel.

XX_AllocSysRAM

Allocate a block of system RAM.

Zones

2 `TS_AllocSysRAM`
3 `KS_AllocSysRAM`

Synopsis

```
void * XX_AllocSysRAM (ksize_t blksize)
```

Input

blksize	The size in bytes of the block of RAM to allocate.

Description

The `XX_AllocSysRAM` kernel service allocates a block of system RAM of size *blksize*. You define the amount of system RAM available to the kernel during the kernel generation process (that is, in the **RTXCgen** program). The kernel uses this RAM during **RTXC** Kernel initialization processing for its internal tables. The kernel keeps track of the amount of this RAM it needs and allows you to allocate any extra RAM from this area of memory.

Note: The **RTXC** Kernel provides no inverse function to release RAM allocated by this function.

Output

If successful, this service returns a pointer to the first address of the allocated block.

If the size of the requested block exceeds the amount of available system RAM, the service returns a null pointer (`(void *)0`).

Example

In Example 8-1 on page 275, the application needs a 256-byte block of system RAM. If the allocation is successful, the pointer to the block is to be stored in the *p* pointer. If there is not enough free RAM available, the task must take the appropriate action.

Example 8-1. Allocate System RAM from Zone 3

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

void *p;

if ((p = KS_AllocSysRAM (256)) == (void *)0)
{
   ... Deal with no memory available
}
else
{
   ... Allocation was successful
}
```

XX_DefFatalErrorHandler

Establish the system error function.

Zones

2 `TS_DefFatalErrorHandler`
3 `KS_DefFatalErrorHandler`

Synopsis

```
void XX_DefFatalErrorHandler
   (int (*errfunc) (void *))
```

Input

errfunc	The entry address for the error function.

Description

The `XX_DefFatalErrorHandler` kernel service establishes a function to which the **RTXC** Kernel branches upon detection of a fatal error. The *errfunc* argument specifies the entry address for the error function.

Output

This service does not return a value.

Example

Example 8-2 on page 277 defines the `kerror` function for receiving all fatal **RTXC** Kernel usage errors. The specified error function requires two arguments as shown in the example: the handle of the Current Task at the time of the error, *task*, and a pointer to that task's interrupt stack frame, *pinfo*. The error function returns an `int` type value. If the returned value is non-zero, the **RTXC** Kernel aborts the Current Task. The kernel ignores the error if the returned value is zero (0).

Example 8-2. Define Fatal Error Function

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

void fehandler (FEPACKET *fepacket); /* prototype for Error Handler */

KS_DefFatalErrorHandler (fehandler); /* define error handler function
*/
... continue

/* System Error Handler for Fatal RTXC Usage */
void fehandler (FEPACKET *fepacket)
{
   ...Do what has to be done here: display the point of error,
      kill the system, whatever is suitable to the application
   return (1); /* have RTXC abort Current Task */
}
```

See Also

XX_GetFatalErrorHandler, page 278

XX_GetFatalErrorHandler

Get the system error function.

Zones

2 `TS_GetFatalErrorHandler`
3 `KS_GetFatalErrorHandler`

Synopsis

`int (*)(void *)) XX_GetFatalErrorHandler (void)`

Inputs

This service has no inputs.

Description

The `XX_GetFatalErrorHandler` kernel service returns a pointer to the function registered to handle fatal system conditions by a previous `XX_DefFatalErrorHandler` call.

Output

The service returns a pointer to the error function installed by a previous call to `XX_DefFatalErrorHandler`.

If no error function has been installed, the kernel service returns a null function pointer (`(int (*)(void *)) 0`).

Example

Example 8-3 needs to know if an error function has been defined. If not, `XX_DefFatalErrorHandler` is used to establish `kerror`, a function external to the Current Task, as the system error handler.

Example 8-3. Read Fatal Error Function

```
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

extern void fehandler (FEPACKET *fepacket);

if (KS_GetFatalErrorHandler () == (void (*)(FEPACKET *fepacket))0)
   KS_DefFatalErrorHandler (fehandler);

...Error handler is now in place, continue
```

See Also

`XX_DefFatalErrorHandler`, page 276

XX_GetFreeSysRAMSize

Get the size of free system RAM.

Zones

2 `TS_GetFreeSysRAMSize`
3 `KS_GetFreeSysRAMSize`

Synopsis

`ksize_t XX_GetFreeSysRAMSize (void)`

Inputs

This service has no inputs.

Description

The `XX_GetFreeSysRAMSize` kernel service determines the amount of free system RAM that is available to the user.

Output

The service returns the number of remaining free bytes of system RAM.

Example

The task in Example 8-4 needs to allocate 2000 bytes of system RAM. It obtains the amount of available system RAM and prints a message if there is less than 2000 bytes.

Example 8-4. Read Amount of Available System RAM from Zone 3

```
#include <stdio.h>
#include "rtxcapi.h"      /* RTXC Kernel Services prototypes */

static char buffer[128];
ksize_t freeRAM;

if ((freeRAM = KS_GetFreeSysRAMSize ()) < 2000)
{
   sprintf (buf, "Only %d free bytes of System RAM", freeRAM);
   putline (buf);
}
else
{
   ... enough RAM available, continue initialization
}
```

See Also

`XX_AllocSysRAM`, page 274

KS_GetSysProp

Get the system properties.

Synopsis

`const SYSPROP * KS_GetSysProp (void)`

Inputs

This service has no inputs.

Description

The `KS_GetSysProp` kernel service returns a pointer to a `SYSPROP` structure containing the system properties used to initialize the system through the `INIT_SysProp` service.

Example 8-5 shows the organization of the `SYSPROP` structure.

Example 8-5. System Properties Structure

```
typedef struct
{
   KATTR attributes;        /* system attributes */
   unsigned long version;   /* kernel version number */
   char *sysrambase;        /* base address of system RAM */
   ksize_t sysramsize;      /* size (bytes) of system RAM */
   char *kernelstackbase;   /* base address of kernel stack */
   ksize_t kernelstacksize; /* size (bytes) of kernel stack */
   unsigned long reserve1;  /* reserved */
   unsigned long reserve2;  /* reserved */
} SYSPROP;
```

Output

The function always returns a pointer to a `SYSPROP` structure.

Example

Example 8-6 reads the clock rate that was established when the system was initialized and sends it to the console.

Example 8-6. Read System Properties

```
#include <stdio.h>       /* standard i/o */
#include "rtxcapi.h"     /* RTXC Kernel Services prototypes */
static char buf[128];
SYSPROP *psysprop = KS_GetSysProp ();

putline (buf);
... continue
```

See Also

`INIT_SysProp`, page 284

KS_GetVersion

Get the version number of the **RTXC** Kernel.

Synopsis

`unsigned long KS_GetVersion (void)`

Inputs

This service has no inputs.

Description

The `KS_GetVersion` kernel service returns the version number of the **RTXC** Kernel.

Output

The function returns a value that contains the version number formatted as follows:

Bits 31–16	System Use
Bits 15–08	Version number (hexadecimal)
Bits 07–00	Release number (hexadecimal)

Note: The developer defines bits 31 through 16 during system generation. This bit field is the developer's version number for the application.

Example

Example 8-7 on page 283 obtains the **RTXC** Kernel version number and displays it on the console.

Example 8-7. Read Version Number

```
#include <stdio.h>          /* standard i/o */
#include "rtxcapi.h"        /* RTXC Kernel Services prototypes */

static char buf[128];
union RTXCver
{
   unsigned long version;
   struct {
      unsigned short sysnum; /* reserved for system use */
      unsigned char ver;     /* version number */
      unsigned char rel;     /* release number */
   } vr;
}curVR;

curVR.version = KS_GetVersion (); /* get RTXC version */
sprintf (buf, "Current RTXC version.release is %d.%d",
         curVR.vr.ver, curVR.vr.rel);
putline (buf);  /* display version # */

... continue
```

INIT_SysProp

Initialize the RTXC system properties.

Synopsis

```
KSRC INIT_SysProp (const SYSPROP *psysprop)
```

Input

psysprop	A pointer to a SYSPROP structure.

Description

The `INIT_SysProp` service performs the required initialization procedure and must be called before any other RTXC kernel service or system function. It passes the system properties, as defined by the user during system generation and found in the `SYSPROP` structure pointed to by *psysprop*, to the kernel. The system properties specify information about how the RTXC Kernel is to operate.

Example 8-5 on page 280 shows the organization of the `SYSPROP` structure.

The system attributes specify the object classes that are defined for the application. The attributes element of the `SYSPROP` structure supports the attributes and corresponding masks listed in Table 8-1.

Table 8-1. System Attributes and Masks

Attribute	**Mask**
Tasks	`K_ATTR_TASKS`
Threads	`K_ATTR_THREADS`
Semaphores	`K_ATTR_SEMAPHORES`
Queues	`K_ATTR_QUEUES`
Mailboxes	`K_ATTR_MAILBOXES`
Partitions	`K_ATTR_PARTITIONS`
Pipes	`K_ATTR_PIPES`

Table 8-1. System Attributes and Masks *(continued)*

Attribute	Mask
Mutexes	K_ATTR_MUTEXES
Event Sources	K_ATTR_SOURCES
Counters	K_ATTR_COUNTERS
Alarms	K_ATTR_ALARMS
Exceptions	K_ATTR_EXCEPTIONS

Output

The service returns a KSRC value as follows:

- `RC_GOOD` if the service completes successfully.
- `RC_VERSION_MISMATCH` if the version number passed in the `SYSPROP` structure is different from the version stored within the RTXC Kernel.

Example

During system initialization, the startup code must initialize the kernel properties before initializing the needed kernel object classes. The system generation process produces a structure of type `SYSPROP` that contains the information about the system necessary for its initialization. Example 8-8 on page 286 externally references that structure and outputs any error messages to the console.

Example 8-8. Initialize Kernel Properties

```
#include "rtxcapi.h"      /* RTXC KC prototypes */

extern const SYSPROP sysprop;

KSRC userinit (void)
{
   KSRC ksrc;
   static char buf[128];

   /* initialize the system properties

   if ((ksrc = INIT_SysProp (&sysprop)) != RC_GOOD)
   {
      putline ("Kernel initialization failure\n");
      return ksrc;  /* end initialization process */
   }
   /* kernel is initialized */

   /* Proceed now with init of kernel object classes */

... Continue with system initialization

}
```

See Also

KS_GetSysProp, page 280

APPENDIX I Fatal Error Codes

This appendix lists the fatal error codes returned by **RTXC/ss** kernel services.

F

`FE_ILLEGAL_ALARM`
: The specified alarm ID is not valid.
- `KS_CloseAlarm` 234
- `KS_DefAlarmName` 240
- `KS_GetAlarmName` 248
- `KS_TestAlarm` 262
- `KS_TestAlarmT` 266
- `KS_TestAlarmW` 269
- `XX_AbortAlarm` 226
- `XX_ArmAlarm` 230
- `XX_CancelAlarm` 232
- `XX_DefAlarmAction` 237
- `XX_DefAlarmActionArm` 239
- `XX_DefAlarmProp` 242
- `XX_DefAlarmSema` 244
- `XX_GetAlarmProp` 250
- `XX_GetAlarmSema` 252
- `XX_GetAlarmTicks` 254
- `XX_RearmAlarm` 260

`FE_ILLEGAL_COUNTER`
: The specified counter ID is not valid.
- `KS_CloseCounter` 192
- `KS_DefCounterName` 196
- `KS_GetCounterName` 206
- `KS_GetElapsedCounterTicks` 211
- `KS_TestAlarmT` 266
- `XX_ClearCounterAttrib` 190
- `XX_DefAlarmProp` 242
- `XX_DefCounterProp` 200
- `XX_GetCounterAcc` 202
- `XX_GetCounterProp` 208
- `XX_SetCounterAcc` 218
- `XX_SetCounterAttrib` 220

`FE_ILLEGAL_EVENTSOURCE`
: The specified event source ID is not valid.
- `KS_CloseEventSource` 161
- `KS_DefEventSourceName` 162
- `KS_DefEventSourceProp` 165
- `KS_GetEventSourceName` 173
- `XX_ClearEventSourceAttr` 158
- `XX_DefCounterProp` 200
- `XX_GetEventSourceAcc` 169
- `XX_GetEventSourceProp` 175
- `XX_SetEventSourceAcc` 183
- `XX_SetEventSourceAttr` 185

`FE_ILLEGAL_EXCPTN`
: The specified Exception ID is not valid.
- `KS_CloseException` 88
- `KS_DefExceptionName` 90
- `KS_DefExceptionProp` 92
- `KS_GetExceptionName` 98
- `KS_GetExceptionProp` 100

`FE_ILLEGAL_LEVEL`
: The specified level is not valid.
- `KS_RaiseThreadLevel` 70
- `TS_LowerThreadLevel` 64

`FE_ILLEGAL_PIPE`
: The specified pipe ID is not valid.

FE_NULL_PIPEFULLBASE
The specified Pipe full base address is null.
XX_DefPipeProp 116

FE_NULL_PIPEPBUFSIZE
The pointer to the buffer size is null.
XX_GetFullPipeBuf 122
XX_PutEmptyGetFullPipeBuf 145

FE_NULL_PIPESIZEBASE
The specified Pipe base size address is null.
XX_DefPipeProp 116

FE_NULL_THREADENTRY
The specified Thread entry address is null.
XX_DefThreadEntry 30
XX_DefThreadProp 37

FE_UNINITIALIZED_ALARM
The specified alarm has not yet been initialized.
KS_TestAlarm 262
KS_TestAlarmT 266
KS_TestAlarmW 269
XX_AbortAlarm 226
XX_ArmAlarm 230
XX_CancelAlarm 232
XX_DefAlarmAction 237
XX_DefAlarmActionArm 239
XX_DefAlarmSema 244
XX_GetAlarmProp 250
XX_GetAlarmSema 252
XX_GetAlarmTicks 254
XX_RearmAlarm 260

FE_UNINITIALIZED_COUNTER
The specified counter has not yet been initialized.
KS_GetElapsedCounterTicks 211
KS_TestAlarmT 266
XX_ClearCounterAttrib 190
XX_GetCounterAcc 202
XX_GetCounterProp 208
XX_SetCounterAcc 218
XX_SetCounterAttrib 220

FE_UNINITIALIZED_EVENTSOURCE
The specified event source has not yet been initialized.
XX_ClearEventSource 158
XX_GetEventSourceAcc 169
XX_GetEventSourceProp 175
XX_SetEventSourceAcc 183
XX_SetEventSourceAttr 185

FE_UNINITIALIZED_EXCPTN
The specified Exception has not yet been initialized.
KS_GetExceptionProp 100

FE_UNINITIALIZED_PIPE
The specified pipe has not yet been initialized.
XX_DefPipeAction 113
XX_GetEmptyPipeBuf 120
XX_GetFullPipeBuf 122
XX_GetPipeBufSize 124
XX_GetPipeProp 130
XX_JamFullGetEmptyPipeBuf 133
XX_JamFullPipeBuf 136
XX_PutEmptyGetFullPipeBuf 145
XX_PutEmptyPipeBuf 147
XX_PutFullGetEmptyPipeBuf 150
XX_PutFullPipeBuf 152

FE_UNINITIALIZED_SEMA
KS_DefAlarmSema 244

FE_UNINITIALIZED_THREAD
The specified thread has not yet been initialized.
TS_GetThreadBaseLevel 42
XX_ClearThreadGateBits 24
XX_DecrThreadGate 27
XX_DefAlarmAction 237

Index

A

C

D

E

R

S

T