Application Note

Using the RZK Semaphore Object

AN018502-0908

Abstract

This application note describes how to use Zilog's Real-Time Kernel (RZK) semaphore object for thread synchronization and memory protection.

In the demo application described in this document, two threads are created:

1. Thread 1–Writes to a buffer
2. Thread 2–Reads from the same buffer

A binary semaphore is created for thread synchronization and for sharing the buffer between the two threads.

Note: *The source code file associated with this application note,* `AN0185-SC01.zip` *is available for download at* www.zilog.com.

RZK Overview

The RZK is a real-time, preemptive, multitasking kernel designed for time-critical embedded applications. It can be used with Zilog's eZ80Acclaim!® family of microcontroller unit (MCU) and microprocessors.

Features of RZK

The RZK is compact with minimal footprint, and can be accommodated in the Flash area of the target processor. RZK allows rapid context switching between threads, besides offering quick interrupt responses. It features a pre-emptive, priority-based, multi-tasking scheduler. It also provides timing support for delays, time-outs, and other periodic events. You can take advantage of RZK's time-slicing option with adjustable time slices. In addition to various configuration options, RZK provides priority inheritance facility.

RZK Objects

RZK objects used for real-time application development are threads, message queues, event groups, semaphores, Timers, partitions and regions (memory objects), and interrupts. This application note focuses on the use of RZK's semaphore object.

For more information on RZK, refer to *Zilog Real-Time Kernel Reference Manual (RM0006)* that is available for download at www.zilog.com. It is also available along with RZK software.

Discussion

This section contains a brief overview of semaphores in general, and discusses the features of RZK's semaphore object in more detail.

Semaphore Overview

In programming, semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system (OS) resources.

In multitasking systems, a semaphore is a variable that indicates the status of a common resource. It is used to lock the resource that is being used. A process requiring the resource checks the availability of the semaphore to determine the resource's status. Semaphores can be binary (0 or 1) or can have additional values. Semaphores are one of the techniques for interprocess communication (IPC).

Semaphore Objects in RZK

In RZK, the semaphore is an optional object and because it uses a mutual-exclusion mechanism (such as IPC) it is not directed to a specific thread. The semaphore handle is a unique value that points to a static Semaphore Control Block. Each semaphore is also identified by a name. This name is a fixed-length ASCII string. To keep the size of the Semaphore Control Block compact, the name field is omitted for the non-debug mode.

You must decide the maximum number of semaphore objects that can be created. This number is referred to as a *Static Creation Value*.

RZK semaphores are highly optimized and are the primary means for addressing the requirements of both mutual exclusion and task synchronization.

The different types of semaphore object that can be used in RZK are discussed below.

Binary Semaphore

A binary semaphore is a general-purpose semaphore used for synchronization and for mutual exclusion. A binary semaphore can be viewed as a flag that is either available or unavailable. A thread tries to acquire a binary semaphore using the `RZKAcquireSemaphore()` API. The outcome depends on whether the semaphore is available or unavailable at the time of the call. If the semaphore is available, the calling thread acquires it and then the semaphore becomes unavailable for other threads. If the semaphore is unavailable, the thread is put on a queue of blocked threads. It then enters a state of pending on the semaphore.

A thread releases a binary semaphore using the `RZKReleaseSemaphore()` API. The outcome again depends on whether the semaphore is available or unavailable at the time of the call. If the semaphore is available, releasing the semaphore has no effect. If one or more tasks are pending on the semaphore's availability, the first thread in the queue of pending threads is unblocked; and the semaphore is acquired by that thread. The semaphore remains unavailable for the rest of the pending threads.

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

Counting Semaphore

Counting semaphores are similar to the binary semaphore, but differ as more than one thread can acquire a semaphore at a time. You need to specify how many threads can acquire the semaphore at a time. The counting semaphores also maintain a count of the number of times the semaphore is acquired by a thread. These semaphores are optimized for protecting and synchronizing multiple instances of a resource.

A thread trying to acquire a semaphore can get blocked when the number of threads already with semaphore(s) equals the initial count. You can also specify threads not to block on a required semaphore. When this option is chosen, a TIMEOUT error occurs.

Developing the Application Using the RZK Semaphore Object

All RZK objects use the kernel services for resource management and provide a set of APIs as an interface to the user application.

The `semaphore_test.c` file, refer to `AN0185-SC01.zip`, contains an application entry point `RZKApplicationEntry()`, within which threads are created. Threads can also be created within any other thread. By default, the `RZKApplicationEntry()` is the first function to execute in RZK when the execution starts.

The RZKApplicationEntry() initializes the UART port, which is connected to a PC COM port. The UART port is initialized to 57600 bps, 1stop bit, no parity, no flow control. The HyperTerminal application is used to display the messages on the console during the different stages of thread execution.

The RZKApplicationEntry() calls the init_array() function to initialize a buffer called msg with zeros.

Within the RZKApplicationEntry() thread, Thread1 and Thread2 are created using the RZKCreateThread() API. Both these threads carry the same priority and are created with round-robin scheduling and the auto-start feature (which means that the thread created first gets into the ready state immediately). In the RZK Semaphore Object application, Thread1 starts executing immediately after the RZKApplicationEntry() thread execution is complete.

Thread1 writes to a character buffer called msg and Thread2 reads from the same msg buffer. Because the msg buffer can be accessed globally by both the threads, memory protection and the task synchronization gain importance. Task synchronization and memory protection are achieved with the use of a binary semaphore.

A binary semaphore is created within the RZKApplicationEntry() using the RZKCreateSemaphore() API. This API is structured as follows:

```
RZK_SEMAPHOREHANDLE_t RZKCreateSemaphore
(RZK_NAME_tszName[MAX_OBJECT_NAME_LEN],

COUNT_t uInitialCount,
RZK_RECV_ATTRIB_et etAttrib);
```

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

For the RZK Semaphore Object application, the semaphore is created with a maximum semaphore count of 1 (MAX_SEM_COUNT = 1) to make it a binary semaphore, and the receive order is FIFO. Therefore, the first thread to execute acquires the semaphore when the semaphore becomes available, irrespective of thread priority.

The RZKCreateSemaphore() function returns a handle to the semaphore if it is created successfully or else it returns NULL. If NULL is returned, the API sets an error value in the thread's Thread Control Block. RZKGetErrorNum() API is then called to retrieve the error number stored in the Thread Control Block.

After the semaphore is created, it must be acquired by the thread requiring to use the resource. In this case, Thread1 must write to the msg buffer or Thread2 must read from the same buffer. For Thread1 or Thread2 to acquire the binary semaphore, the RZKAquireSemaphore() API is used. This API is structured as follows:

```
RZK_STATUS_t RZKAcquireSemaphore
(RZK_SEMAPHOREHANDLE_t hSemaphore,
TICK_ttBlockTime);
```

If the semaphore is not immediately available (that is, the other thread has acquired the semaphore), Thread1 or Thread2 wait for the duration specified in tBlockTime.

When either of the threads, Thread1 or Thread2, must release a semaphore, the `RZKReleaseSemaphore()` API is used. This API is structured as follows:

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

Sequence of Events

Thread1 acquires the binary semaphore and in its first iteration, the `msg` buffer is filled with the characters from *A* to *Z* and from *a* to *z* (totally 52 locations). While Thread1 executes, context switching occurs and Thread2, which is in ready queue starts executing. Thread2 blocks on the binary semaphore, because Thread1 has not yet released it. In effect Thread2 is unable to access the `msg` buffer. The `msg` buffer is thus protected from Thread2, until it is completely updated by Thread1.

When Thread1 completes writing to buffer, it releases the semaphore and tries to start the second iteration, which is to fill the buffer with characters from *z* to *a* and from *Z* to *A* (totally 52 locations). However, as soon as Thread1 releases the semaphore, Thread2 acquires the Semaphore and starts reading from the buffer. Thread2 prints the contents of the buffer on the HyperTerminal console.

Note: *While printing each character, the letter T1 or T2 is prefixed to the character to identify the thread that is executing. For example: T1A, T1B, indicates that Thread1 is writing data A and B, to the buffer. T2A, T2B, indicates that Thread2 is reading data A and B from the buffer.*

Thread2 releases the semaphore only after reading the buffer completely. This is when Thread1, which is blocked on the semaphore, acquires it and starts the second iteration. It then fills the buffer from *z* to *a* and *Z* to *A* (totally 52 locations). Again after the semaphore is released by Thread1, Thread2 acquires it and prints data on the HyperTerminal console.

See Appendix A—Flowcharts on page 7 for the flowcharts displaying the sequence of events described previously.

Note: *If the* `#define EN_SEMAPHORE` *in the* `semaphore_test.c` *file is commented out, incorrect data is read from the shared memory area (*`msg` *buffer). This emphasizes that not using semaphores within the RZK context will impact adversely on thread synchronization and memory protection.*

Executing the RZK Semaphore Object Application

The RZK Semaphore Object Demo described in this application note requires the eZ80F91 Development Board and RZK. For the Demo execution, the application source file must be added and integrated to the RZK project.

Adding Application Source File to the RZK Project

This section contains the details of adding the RZK Semaphore Object Demo file to the RZK project.

The RZK software is available on the www.zilog.com. It is downloaded to a PC with a user registration key. RZK is installed in any location as specified by you. Its default location is `C:\Program Files\Zilog`.

Perform the following steps to add the Demo files to the RZK directory:

1. Download RZK, browse to the location where RZK is downloaded, and open the `..\eZ80F91\Sample Programs` folder.
2. Download the `AN0185-SC01.zip` file and extract its contents to a folder on your PC.

Note that there is a folder `\Semaphore` within the extracted folder.

3. Copy the `\Semaphore` folder and paste it into the `..\eZ80F91\Sample Programs` folder. Note that there are two folders within the `\Semaphore` folder. These folders are:
   (a) `\IntFlash`–Containing the `semaphore.pro` file to be downloaded into internal Flash Memory of the MCU.
   (b) `\Ram`–Containing the `semaphore.pro` file to be downloaded into the RAM memory of the MCU.

4. Launch ZDS II and open the project file, `semaphore.pro` located in the path: `..\eZ80F91\Sample Programs\Semaphore\IntFlash or RAM`.

Note: *Ensure not modifying any of the project settings in any of the project files.*

Running the RZK Semaphore Object Demo

To run the RZK Semaphore Object Demo perform the following steps:

1. Compile and build the `semaphore.pro` project, and download the resultant file to the eZ80F91 target board, using Zilog's ZDS II-IDE.

Note: *Use the Flash Loader option in ZDS II to download to internal Flash. For information on using ZDS II-IDE, refer to Zilog Developer Studio II-eZ80Acclaim!® User Manual (UM0144).*

2. Launch the HyperTerminal with settings 8-N-1 and with a baud rate of 57600 kbps.
3. Run the program and observe the display on the HyperTerminal console. Figure 1 is a snapshot of the display.

```
Starting to create RZK threads
Creating thread #1
Creating thread #2
Creating a Binary semaphore

Started Thread1

T1A:T1B:T1C:T1D:T1E:T1F:T1G:T1H:
Started Thread2
T1I:T1J:T1K:T1L:T1M:T1N:T1O:T1P:T1Q:T1R:T1S:T1T:T1U:T1V:T1W:T1X:T1Y:T1Z:T1
T1c:T1d:T1e:T1f:T1g:T1h:T1i:T1j:T1k:T1l:T1m:T1n:T1o:T1p:T1q:T1r:T1s:T1t:T1
T1w:T1x:T1y:T1z:
 T2A: T2B: T2C: T2D: T2E: T2F: T2G: T2H: T2I: T2J: T2K: T2L: T2M: T2N: T2O
 T2Q: T2R: T2S: T2T: T2U: T2V: T2W: T2X

Started Thread1
: T2Y: T2Z: T2a: T2b: T2c: T2d: T2e: T2f: T2g: T2h: T2i: T2j: T2k: T2l: T2
: T2o: T2p: T2q: T2r: T2s: T2t: T2u: T2v: T2w: T2x: T2y: T2z:
T1z:T1y:T1x:T1w:T1v:T1u:T1t:T1s:T1r:T1q:T1p:T1o:T1n:T1m:T1l:T1k:T1j:T1i:T1
T1f:T1e:T1d:T1c:T1b:T1a:T1Z:T1Y:T1X:T1W:T1V:T1U:T1T:T1S:T1R:T1Q:T1P:T1

Started Thread2
O:T1N:T1M:T1L:T1K:T1J:T1I:T1H:T1G:T1F:T1E:T1D:T1C:T1B:T1A:
 T2z: T2y: T2x: T2w: T2v: T2u: T2t: T2s: T2r: T2q: T2p: T2o: T2n: T2m: T2l
 T2j: T2i: T2h: T2g: T2f: T2e: T2d: T2c: T2b: T2a:
 T2Z: T2Y: T2X: T2W: T2V: T2U: T2T: T2S: T2R: T2Q: T2P: T2O: T2N: T2M: T2L
 T2J: T2I: T2H: T2G: T2F: T2E: T2D: T2C: T2B: T2A:
```

Figure 1. Data Read from msg Buffer

Summary

This application note demonstrates how a semaphore is used for thread synchronization and for protection of the shared memory area in RZK.

References

The documents associated with RZK and the eZ80F91 MCU available on www.zilog.com are provided below:

- Zilog Real-Time Kernel Reference Manual (RM0006)
- Zilog Real-Time Kernel User Manual (UM0075)
- RZK Developer's Kit Quick Start Guide (QS0027)
- eZ80F91 Module Product Specification (PS0193)
- Zilog Developer Studio II--eZ80Acclaim!® User Manual (UM0144)
- eZ80® CPU User Manual (UM0077)

Appendix A—Flowcharts

The flowcharts in this appendix display how semaphores are used with RZK. Figure 2 displays the program flow when using a semaphore.

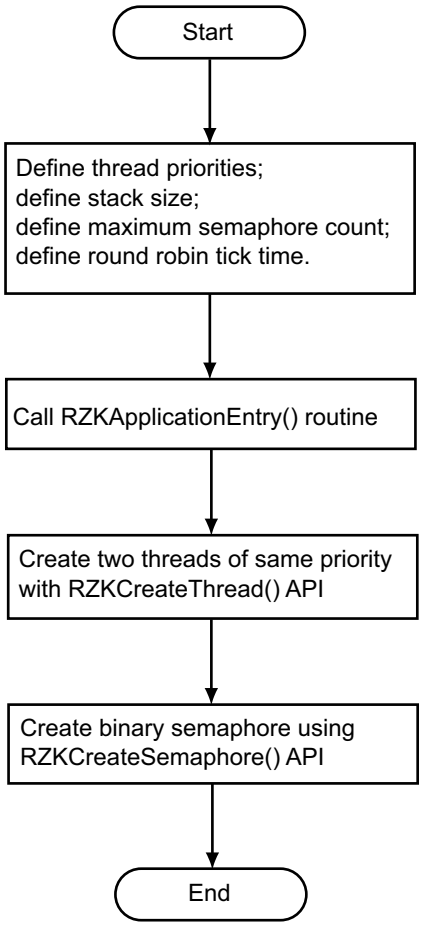

Figure 2. Flowchart for Using Semaphores in RZK

Figure 3 displays the sequence of events when Thread 2 requires the semaphore that Thread 1 has previously acquired.

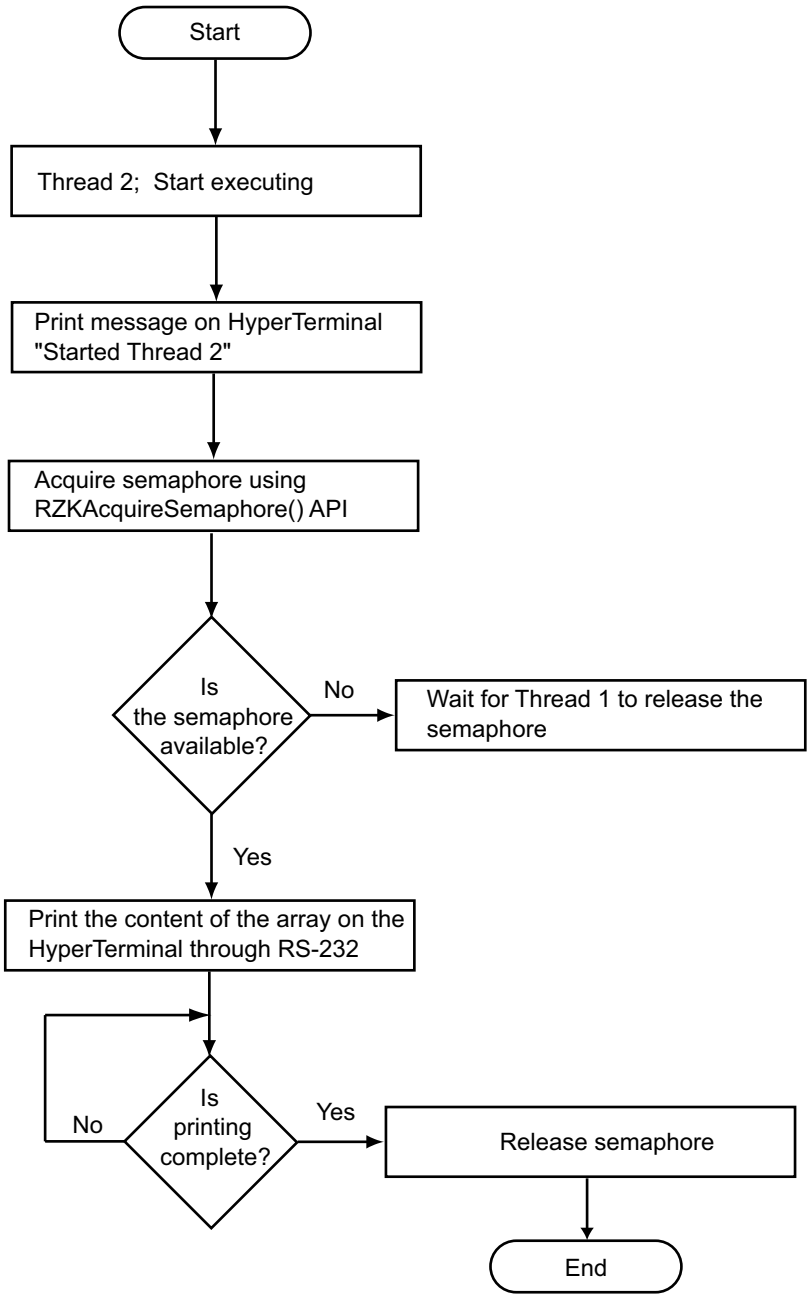

Figure 3. Flowchart when Thread 2 Requires the Semaphore

Figure 4 displays the sequence of event when Thread 1 requires the semaphore that Thread 2 has previously acquired.

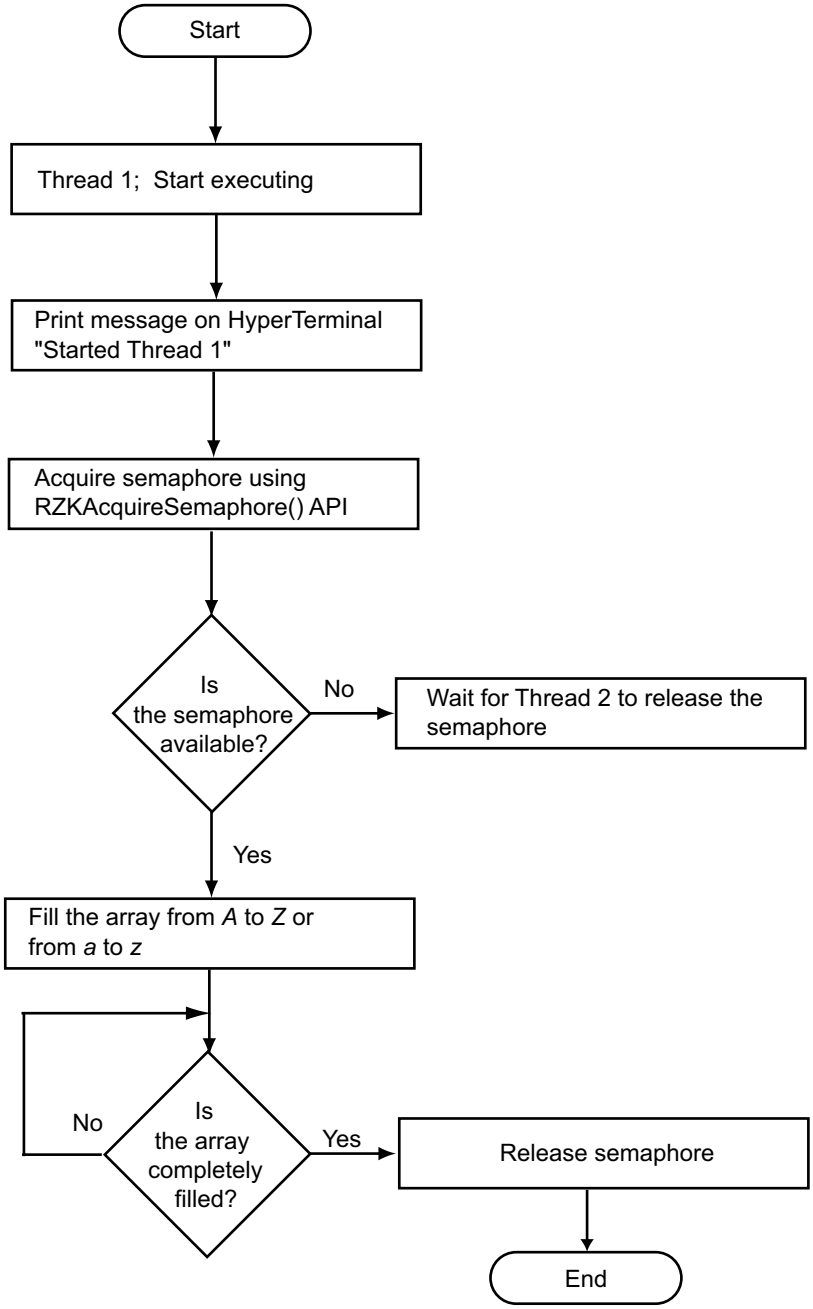

Figure 4. Flowchart when Thread 1 Requires the Semaphore

Warning: DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

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

As used herein

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

Document Disclaimer