Event Chaining

Enables Real-Time Systems to Respond to Multiple Real-Time Events More Efficiently

Innovative function callback capability permits responsiveness, while reducing overhead

January 2020

Microsoft

Contents

Introduction

Azure RTOS ThreadX provides several advanced technology features that can be beneficial during the development stage as well as during run-time. These features include real-time Event-Chaining, Application Notification “Callback” Functions, and many others. We will investigate the Event Chaining and Notification Callback Function topics in this paper.

Event-Chaining

Event-Chaining is a technique that enables a single RTOS action based on the occurrence of independent events. This is particularly useful in activating an application thread that is suspended on two or more resources. For example, suppose a single thread is responsible for processing messages from 5 or more message queues, and must suspend when no messages are available. Such resources might be messages being awaited in one or more queues, a semaphore from one of several cooperating threads, or an event in an event flags group. In general, Event-Chaining results in fewer threads, less overhead, and smaller RAM requirements. It also provides a highly flexible mechanism to handle synchronization requirements of more complex systems. Implementing this technique is a three-step process as follows:

1. Register one or more notification callback functions. We'll explain notification callback functions below.
2. The event occurs, and the registered notification callback function is automatically invoked. Each such function typically contains a *tx_semaphore_put* service call, which increments a "gatekeeper" semaphore which communicates to a waiting thread that a particular event has occurred. However, many other service calls could be used.
3. A thread, suspended on the "gatekeeper" semaphore mentioned above, is activated. Getting this semaphore signifies that one of the events in question has occurred and the thread determines which, and then performs the actions appropriate for that event.

There are three types of Event-Chaining available:

1. Queue Event-Chaining
2. Semaphore Event Chaining
3. Event Flags Group Event Chaining

A typical use for Event-Chaining is to create a mechanism for a thread to suspend on two or more objects. For example, this technique can be used to permit a thread to suspend on any of the following situations:

- Suspend on a queue, a semaphore, and an event flags group
- Suspend on a queue or a semaphore
- Suspend on a queue or an event flags group
- Suspend on two queues
- Suspend on three queues
- Suspend on four queues

An important advantage of the Event-Chaining technique is that one or more threads waiting for an event to occur can be activated automatically when the event occurs. In general, this technique will reduce the number of threads needed to respond to an event and will reduce the associated resources and overhead required for processing systems of this nature.

In this paper, we will focus on Queue Event Chaining. The principles are the same across all three types, so the process described below for Queue Event Chaining can be replicated for either of the other two types.

Notification Callback Functions

Some applications may find it advantageous to be notified whenever a message is placed on a queue. ThreadX provides this ability through the *tx_queue_send_notify* service. This service registers the supplied application notification function with the specified queue. ThreadX will subsequently invoke this application notification function whenever a message is sent to the queue. The processing within the application notification function is determined by the application; however, it typically consists of resuming the appropriate thread for processing the new message.

For example, the *tx_queue_send_notify(&my_queue, queue_notify)* function registers a callback function ("queue_notify") that would be called every time a message is sent to the specified queue ("my_queue").

Queue Event-Chaining

Suppose a single thread is responsible for processing messages from five different queues and must also suspend when no messages are available. This is easily accomplished by registering an application notification function for each queue and introducing an additional counting semaphore. Specifically, the application notification function performs a *tx_semaphore_put* whenever it is called (the semaphore count represents the total number of messages in all five queues). The processing thread suspends on this semaphore via the *tx_semaphore_get* service. When the semaphore is available (in this case, when a message is available!), the processing thread is resumed. It then interrogates each queue for a message, processes the found message, and performs another *tx_semaphore_get* to wait for the next message. Accomplishing this without event-chaining is quite difficult and likely would require more threads and/or additional application code. As noted, implementing Event-Chaining is a multiple-step process. Figure **1 : Template for Event-Chaining with a message queue**

contains a template that illustrates the components involved for Event-Chaining with a message queue.

1. Initialization TX_QUEUE my_queue; TX_SEMAPHORE gatekeeper; ULONG my_message[4]; tx_queue_send_notify (&my_queue, queue_notify); void queue_notify (TX_QUEUE *my_queue);	/* The queue, semaphore, and message declarations, the registration of the notification callback function, and the prototype for the notification callback function are usually placed in the tx_application_define function, which is part of the initialization process */

2a. Event Occurrence tx_queue_send (&my_queue, my_message, TX_NO_WAIT);	/* A message is sent to the queue somewhere in the application. Whenever a message is sent to this queue, the notification callback function is automatically invoked, thus causing the semaphore gatekeeper to be incremented. */

2b. Notification Callback Function Called	/* Notification callback function to increment the "gatekeeper" semaphore is called whenever a

void queue_notify (TX_QUEUE *my_queue)
{
  tx_semaphore_put (&gatekeeper);
}
	message has been sent to my_queue
	*/

3. Thread Activation tx_semaphore_get (&gatekeeper, TX_WAIT_FOREVER);	/* Somewhere in the application, a thread suspends on semaphore gatekeeper, which is equivalent to waiting for a message to appear on the queue */

Figure 1 : Template for Event-Chaining with a message queue

Sample System Using Event-Chaining

We will now study a complete sample system that uses Event-Chaining. The system is characterized in Figure 2.

All the thread suspension examples in previous chapters involved one thread waiting on one object, such as a mutex, a counting semaphore, an event flags group, or a message queue. In this sample system, we have 2 threads waiting on multiple objects. Specifically, threads wait for a message to appear on either *queue_1* or *queue_2*.

Speedy_thread has priority 5 and *slow_thread* has priority 15. We will use Event-Chaining to automatically increment the counting semaphore named "gatekeeper" whenever a message is sent to either *queue_1* or *queue_2*. We use two application timers to send messages to *queue_1* or *queue_2* at periodic time intervals and the threads wait for a message to appear.

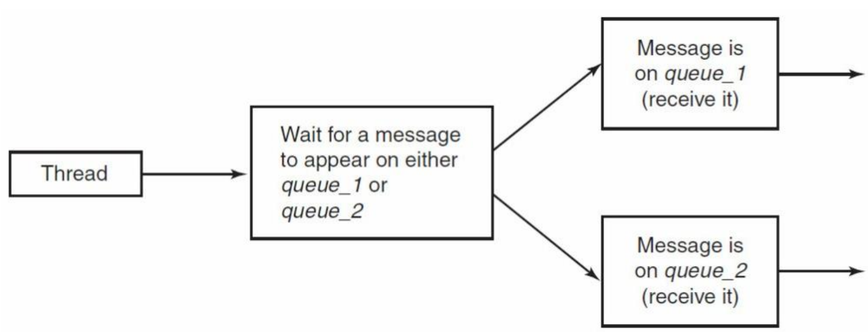

Figure 2. Multiple object suspension problem

Figure contains a description of the two activities for speedy_thread.

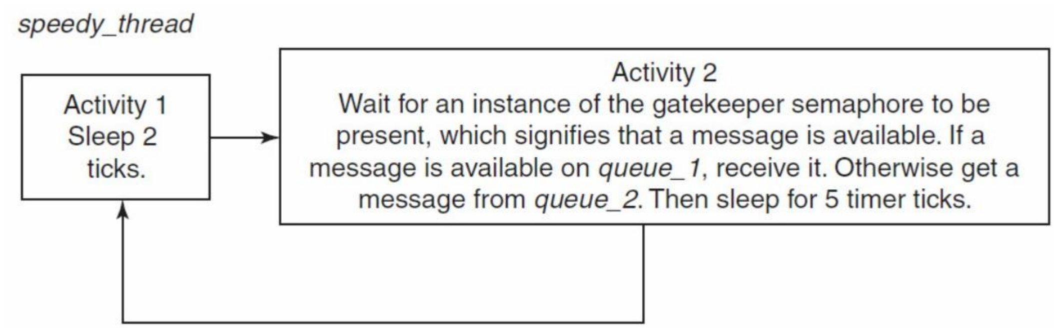

Figure 3. speedy_thread activities

Figure contains a description of the two activities for *slow_thread*.

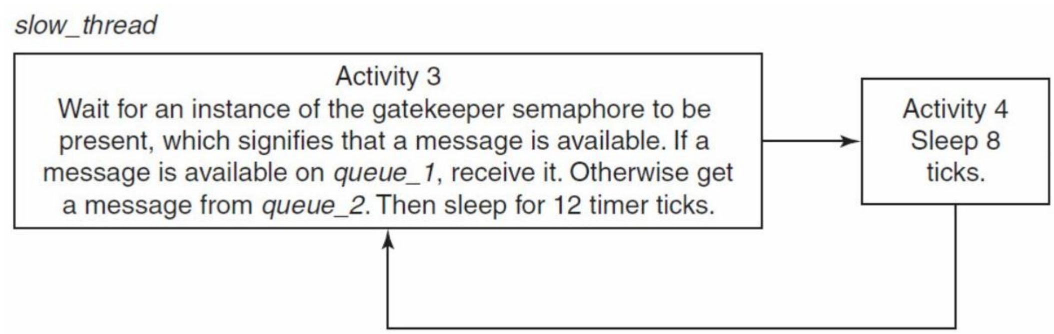

Figure 4. slow_thread activities

Listing for sample_system.c

The sample system named *sample_system.c* appears below; line numbers have been added for easy reference.

```
/* sample_system.c

    Create two threads, one byte pool, two message queues, three timers, and
    one counting semaphore. This is an example of multiple object suspension
    using Event-Chaining, i.e., speedy_thread and slow_thread wait for a
    message to appear on either of two queues */


   /*****************************************************/
   /*     Declarations, Definitions, and Prototypes    */
   /*****************************************************/

   #include “tx_api.h”
   #include <stdio.h>

   #define STACK_SIZE         1024
   #define BYTE_POOL_SIZE     9120
   #define NUMBER_OF_MESSAGES 100
   #define MESSAGE_SIZE       TX_1_ULONG
   #define QUEUE_SIZE         MESSAGE_SIZE*sizeof(ULONG)*NUMBER_OF_MESSAGES


   /* Define the ThreadX object control blocks... */

   TX_THREAD speedy_thread; /* higher priority thread */
   TX_THREAD slow_thread; /* lower priority thread */
```

```

TX_BYTE_POOL my_byte_pool; /* byte pool for stacks and queues */
TX_SEMAPHORE gatekeeper; /* indicate how many objects available */

TX_QUEUE queue_1; /* queue for multiple object suspension */
TX_QUEUE queue_2; /* queue for multiple object suspension */

TX_TIMER stats_timer; /* generate statistics at intervals */
TX_TIMER queue_timer_1; /* send message to queue_1 at intervals */
TX_TIMER queue_timer_2; /* send message to queue_2 at intervals */

/* Variables needed to get info about the message queue */
CHAR *info_queue_name;
TX_THREAD *first_suspended;
TX_QUEUE *next_queue;
ULONG enqueued_1=0, enqueued_2=0, suspended_count=0, available_storage=0;

/* Define the variables used in the sample application... */
ULONG speedy_thread_counter=0, total_speedy_time=0;
ULONG slow_thread_counter=0, total_slow_time=0;
ULONG send_message_1[TX_1_ULONG]={0X0}, send_message_2[TX_1_ULONG]={0X0};
ULONG receive_message_1[TX_1_ULONG], receive_message_2[TX_1_ULONG];

/* speedy_thread and slow_thread entry function prototypes */
void speedy_thread_entry(ULONG thread_input);
void slow_thread_entry(ULONG thread_input);

/* timer entry function prototypes */
void queue_timer_1_entry(ULONG thread_input);
void queue_timer_2_entry(ULONG thread_input);
void print_stats(ULONG);

/* event notification function prototypes used for Event-Chaining */
void queue_1_send_notify(TX_QUEUE *queue_1_ptr);
void queue_2_send_notify(TX_QUEUE *queue_2_ptr);


/****************************************************/
/*                 Main Entry Point                 */
/****************************************************/

/* Define main entry point. */

int main()
{
  /* Enter the ThreadX kernel. */
  tx_kernel_enter();
}


/****************************************************/
/*              Application Definitions             */
/****************************************************/


/* Define what the initial system looks like. */

void tx_application_define(void *first_unused_memory)
{
```

```

CHAR *speedy_stack_ptr;
CHAR *slow_stack_ptr;
CHAR *queue_1_ptr;
CHAR *queue_2_ptr;

  /* Create a byte memory pool from which to allocate the thread stacks. */
  tx_byte_pool_create(&my_byte_pool, “my_byte_pool”,
              first_unused_memory, BYTE_POOL_SIZE);

  /* Create threads, queues, the semaphore, timers, and register functions
     for Event-Chaining */

  /* Allocate the stack for speedy_thread. */
  tx_byte_allocate(&my_byte_pool, (VOID **) &speedy_stack_ptr, STACK_SIZE,
              TX_NO_WAIT);

  /* Create speedy_thread. */
  tx_thread_create(&speedy_thread, “speedy_thread”, speedy_thread_entry, 0,
              speedy_stack_ptr, STACK_SIZE, 5, 5, TX_NO_TIME_SLICE,
              TX_AUTO_START);

  /* Allocate the stack for slow_thread. */
  tx_byte_allocate(&my_byte_pool, (VOID **)&slow_stack_ptr, STACK_SIZE,
              TX_NO_WAIT);

  /* Create slow_thread */
  tx_thread_create(&slow_thread, “slow_thread”, slow_thread_entry, 1,
              slow_stack_ptr, STACK_SIZE, 15, 15, TX_NO_TIME_SLICE,
              TX_AUTO_START);

  /* Create the message queues used by both threads. */
  tx_byte_allocate(&my_byte_pool, (VOID **)&queue_1_ptr,
              QUEUE_SIZE, TX_NO_WAIT);

  tx_queue_create (&queue_1, “queue_1”, MESSAGE_SIZE,
              Queue_1_ptr, QUEUE_SIZE);

  tx_byte_allocate(&my_byte_pool, (VOID **) &queue_2_ptr,
              QUEUE_SIZE, TX_NO_WAIT);

  tx_queue_create (&queue_2, “queue_2”, MESSAGE_SIZE,
              Queue_2_ptr, QUEUE_SIZE);

  /* Create the gatekeeper semaphore that counts the available objects */
  tx_semaphore_create (&gatekeeper, “gatekeeper”, 0);

  /* Create and activate the stats timer */
  tx_timer_create (&stats_timer, “stats_timer”, print_stats,
              0x1234, 500, 500, TX_AUTO_ACTIVATE);

  /* Create and activate the timer to send messages to queue_1 */
  tx_timer_create (&queue_timer_1, “queue_timer”, queue_timer_1_entry,
              0x1234, 12, 12, TX_AUTO_ACTIVATE);

```

```
  /* Create and activate the timer to send messages to queue_2 */
  tx_timer_create (&queue_timer_2, “queue_timer”, queue_timer_2_entry,
               0x1234, 9, 9, TX_AUTO_ACTIVATE);

  /* Register the function to increment the gatekeeper semaphore when a
     message is sent to queue_1 */
  tx_queue_send_notify(&queue_1, queue_1_send_notify);

  /* Register the function to increment the gatekeeper semaphore when a
     message is sent to queue_2 */
  tx_queue_send_notify(&queue_2, queue_1_send_notify);
}


/****************************************************/
/*                Function Definitions              */
/****************************************************/


/* Entry function definition of speedy_thread
   it has a higher priority than slow_thread */

void speedy_thread_entry(ULONG thread_input)
{

ULONG start_time, cycle_time=0, current_time=0;
UINT status;

  /* This is the higher priority speedy_thread */

  while(1)
  {
    /* Get the starting time for this cycle */
    start_time = tx_time_get();

    /* Activity 1: 2 ticks. */
    tx_thread_sleep(2);

    /* Activity 2: 5 ticks. */
    /* wait for a message to appear on either one of the two queues */
    tx_semaphore_get (&gatekeeper, TX_WAIT_FOREVER);

    /* Determine whether a message queue_1 or queue_2 is available */
    status = tx_queue_receive (&queue_1, receive_message_1, TX_NO_WAIT);

    if (status == TX_SUCCESS)
        ; /* A message on queue_1 has been found-process */
    else
        /* Receive a message from queue_2 */
        tx_queue_receive (&queue_2, receive_message_2, TX_WAIT_FOREVER);

    tx_thread_sleep(5);

    /* Increment the thread counter and get timing info */
    speedy_thread_counter++;
    current_time = tx_time_get();
    cycle_time = current_time-start_time;
    total_speedy_time = total_speedy_time + cycle_time;
  }
```

```
}

/*****************************************************/

/* Entry function definition of slow_thread
   it has a lower priority than speedy_thread */

void slow_thread_entry(ULONG thread_input)
{

ULONG start_time, current_time=0, cycle_time=0;
UINT status;


  while(1)
  {
    /* Get the starting time for this cycle */
       start_time=tx_time_get();

    /* Activity 3-sleep 12 ticks. */
    /* wait for a message to appear on either one of the two queues */
    tx_semaphore_get (&gatekeeper, TX_WAIT_FOREVER);

    /* Determine whether a message queue_1 or queue_2 is available */
    status = tx_queue_receive (&queue_1, receive_message_1, TX_NO_WAIT);

    if (status == TX_SUCCESS)
        ; /* A message on queue_1 has been found-process */
    else
        /* Receive a message from queue_2 */
        tx_queue_receive (&queue_2, receive_message_2, TX_WAIT_FOREVER);

    tx_thread_sleep(12);


    /* Activity 4: 8 ticks. */
    tx_thread_sleep(8);

    /* Increment the thread counter and get timing info */
    slow_thread_counter++;

    current_time = tx_time_get();
    cycle_time = current_time-start_time;
    total_slow_time = total_slow_time + cycle_time;
  }
}

/*************************************************/
/* print statistics at specified times */
Void print_stats (ULONG invalue)
{
ULONG current_time, avg_slow_time, avg_speedy_time;

  If ((speedy_thread_counter>0) && (slow_thread_counter>0))
  {
      current_time = tx_time_get();
      avg_slow_time = total_slow_time/slow_thread_counter;
      avg_speedy_time = total_speedy_time/speedy_thread_counter;
      tx_queue_info_get (&queue_1, &info_queue_name, &enqueued_1,
```

```
         &available_storage, &first_suspended,
         &suspended_count, &next_queue);
   tx_queue_info_get (&queue_2, &info_queue_name, &enqueued_2,
         &available_storage, &first_suspended,
         &suspended_count, &next_queue);
  printf("\nEvent-Chaining: 2 threads waiting for 2 queues\n\n");
  printf(" Current Time:          %lu\n", current_time);
  printf(" speedy_thread counter:  %lu\n", speedy_thread_counter);
  printf(" speedy_thread avg time: %lu\n", avg_speedy_time);
  printf(" slow_thread counter:    %lu\n", slow_thread_counter);
  printf(" slow_thread avg time:   %lu\n", avg_slow_time);
  printf(" total # queue_1 messages sent:
  %lu\n", send_message_1[TX_1_ULONG-1]);
  printf(" total # queue_2 messages sent:
  %lu\n", send_message_2[TX_1_ULONG-1]);
  printf(" current # messages in queue_1:
  %lu\n", enqueued_1);
  printf(" current # messages in queue_2: %lu\n\n", enqueued_2);

 }
 else printf("Bypassing print_stats function, Current Time: %lu\n",
       tx_time_get());
}



/*****************************************************/
/* Send a message to queue_1 at specified times */
void queue_timer_1_entry (ULONG invalue)
{

 /* Send a message to queue_1 using the multiple object suspension approach
 */
 /* The gatekeeper semaphore keeps track of how many objects are
 available
   via the notification function */
 send_message_1[TX_1_ULONG-1]++;
 tx_queue_send (&queue_1, send_message_1, TX_NO_WAIT);

}

/*****************************************************/
/* Send a message to the queue at specified times */
void queue_timer_2_entry (ULONG invalue)
{

 /* Send a message to queue_2 using the multiple object suspension approach
 */
 /* The gatekeeper semaphore keeps track of how many objects are
 available
   via the notification function */
 send_message_2[TX_1_ULONG--1]++;
 tx_queue_send (&queue_2, send_message_2, TX_NO_WAIT);

}

/*****************************************************/
/* Notification function to increment gatekeeper semaphore
  whenever a message has been sent to queue_1 */
```

```
312    void queue_1_send_notify(TX_QUEUE *queue_ptr_1)
313    {
314      tx_semaphore_put (&gatekeeper);
315    }
316
317    /****************************************************/
318    /* Notification function to increment gatekeeper semaphore
319       whenever a message has been sent to queue_2 */
320    void queue_2_send_notify(TX_QUEUE *queue_ptr_2)
321    {
322      Tx_semaphore_put (&gatekeeper);
323    }
```

END Example code. Figure **5. Comments about sample system listing**

contains several comments about this listing, using the line numbers as references.

Lines	Comments
024 through 035	Declaration of system resources including threads, byte pool, semaphore, queues, and timers
037 through 047	Declaration of variables used in the system including parameters for the queue info get services
049 through 060	Declaration of prototypes for thread entry functions, timer entry function, and event notification functions
116 through 127	Creation of the two queues used for multiple object suspension
129 and 130	Creation of the gatekeeper semaphore used for Event-Chaining
132 through 142	Creation of the timer for display statistics at periodic intervals, and creation of the two timers to send messages to the queues at various intervals
144 through 150	Registration of the two functions that increment the gatekeeper semaphore whenever messages are sent to the queues
159 through 199	Entry function for Speedy Thread; lines 178 through 191 contain the implementation of Activity 2
203 through 244	Entry function for Slow Thread; lines 218 through 231 contain the implementation of Activity 3
247 through 276	Entry function for timer print stats, which includes calculating average cycle time, number of times through each cycle, and info get for the two queues

281 through 304	Entry functions for timers to send messages to queue_1 and queue_2 at periodic intervals
307 through 320	Entry functions for the notification callback functions; these functions increment semaphore gatekeeper whenever a message is send to either queue_1 or queue_2; these functions are essential to the Event-Chaining technique

Figure 5. Comments about sample system listing

Following is some sample output for this system after it has executed for 500 timer ticks, using information obtained from the tx_queue_info_get service:

Event-Chaining: 2 threads waiting for 2 queues	
Current Time:	500
speedy_thread counter:	69
speedy_thread avg time:	7
slow_thread counter:	24
slow_thread avg time:	20
total # *queue_1* messages sent:	41
total # *queue_2* messages sent:	55
current # messages in *queue_1*:	0
current # messages in *queue_2*:	1

Conclusion

Event-Chaining is one technique that uses notification callback functions to reduce the number of threads required to manage responses to multiple events in a real-time system. For more information about Event-Chaining, Callback Functions, or any of the other advanced technology features of Azure RTOS, please visit Azure.com/RTOS.

Microsoft