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
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Event-Chaining</td>
<td>4</td>
</tr>
<tr>
<td>Notification Callback Functions</td>
<td>4</td>
</tr>
<tr>
<td>Queue Event-Chaining</td>
<td>5</td>
</tr>
<tr>
<td>Sample System Using Event-Chaining</td>
<td>6</td>
</tr>
<tr>
<td>Sample System Listing for <code>sample_system.c</code></td>
<td>7</td>
</tr>
<tr>
<td>Conclusion</td>
<td>14</td>
</tr>
</tbody>
</table>
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.
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                                                                                   | /* 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 */ |
| 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 */ |
| TX_SEMAPHORE gatekeeper;                                                                             | /* 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 */ |
| ULONG my_message[4];                                                                                 | /* 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 */ |
| tx_queue_send_notify (&my_queue, queue_notify);                                                       | /* 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. */ |
| void queue_notify (TX_QUEUE *my_queue);                                                            | /* 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. */ |
| 2a. Event Occurrence                                                                                | /* 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. */ |
| 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 */

Scheduling Enables Real-Time Systems to Achieve Higher Performance // 5
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.
Scheduling Enables Real-Time Systems to Achieve Higher Performance

**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.

```c
000 /* sample_system.c */
001
002 Create two threads, one byte pool, two message queues, three timers, and
003 one counting semaphore. This is an example of multiple object suspension
004 using Event-Chaining, i.e., speedy_thread and slow_thread wait for a
005 message to appear on either of two queues */
006
007 /*******************************************************************************/
008 /*     Declarations, Definitions, and Prototypes */
009 /*******************************************************************************/
010
011 #include "tx_api.h"
012 #include <stdio.h>
013 #define STACK_SIZE         1024
014 #define BYTE_POOL_SIZE     9120
015 #define NUMBER_OF_MESSAGES 100
016 #define MESSAGE_SIZE       TX_1_ULONG
017 #define QUEUE_SIZE         MESSAGE_SIZE*sizeof(ULONG)*NUMBER_OF_MESSAGES
018
019 /* Define the ThreadX object control blocks... */
020
021 TX_THREAD speedy_thread; /* higher priority thread */
022 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)
{
085  CHAR *speedy_stack_ptr;
086  CHAR *slow_stack_ptr;
087  CHAR *queue_1_ptr;
088  CHAR *queue_2_ptr;
089
090  /* Create a byte memory pool from which to allocate the thread stacks. */
091  tx_byte_pool_create(&my_byte_pool, "my_byte_pool",
                        first_unused_memory, BYTE_POOL_SIZE);
092
093  /* Create threads, queues, the semaphore, timers, and register functions
   for Event-Chaining */
094
095  /* Allocate the stack for speedy_thread. */
096  tx_byte_allocate(&my_byte_pool, (VOID **) &speedy_stack_ptr, STACK_SIZE,
                        TX_NO_WAIT);
097
098  /* Create speedy_thread. */
099  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);
100
101  /* Allocate the stack for slow_thread. */
102  tx_byte_allocate(&my_byte_pool, (VOID **) &slow_stack_ptr, STACK_SIZE,
                        TX_NO_WAIT);
103
104  /* Create slow_thread */
105  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);
106
107  /* Create the message queues used by both threads. */
108  tx_byte_allocate(&my_byte_pool, (VOID **) &queue_1_ptr, QUEUE_SIZE,
                        TX_NO_WAIT);
109
110  /* Create the gatekeeper semaphore that counts the available objects */
111  tx_semaphore_create(&gatekeeper, "gatekeeper", 0);
112
113  /* Create and activate the stats timer */
114  tx_timer_create(&stats_timer, "stats_timer", print_stats,
                        0x1234, 500, 500, TX_AUTO_ACTIVATE);
115
116  /* Create and activate the timer to send messages to queue_1 */
117  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 */

140 tx_timer_create (&queue_timer_2, “queue_timer”, queue_timer_2_entry,
141 0x1234, 9, 9, TX_AUTO_ACTIVATE);

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

148 /* Register the function to increment the gatekeeper semaphore when a
149  message is sent to queue_2 */
150 tx_queue_send_notify(&queue_2, queue_1_send_notify);
151 }
152
154 /* Entry function definition of speedy_thread
155  it has a higher priority than slow_thread */
156
160 void speedy_thread_entry(ULONG thread_input)
161 {
162  ULONG start_time, cycle_time=0, current_time=0;
163  UINT status;
164  
166  /* This is the higher priority speedy_thread */
167  while(1)
168  {
169    /* Get the starting time for this cycle */
170    start_time = tx_time_get();
171  
172    /* Activity 1: 2 ticks. */
173    tx_thread_sleep(2);
174  
175    /* Activity 2: 5 ticks. */
176    /* wait for a message to appear on either one of the two queues */
177    tx_semaphore_get (&gatekeeper, TX_WAIT_FOREVER);
178  
179    /* Determine whether a message queue_1 or queue_2 is available */
180    status = tx_queue_receive (&queue_1, receive_message_1, TX_NO_WAIT);
181  
183      if (status == TX_SUCCESS)
184        ; /* A message on queue_1 has been found-process */
187      else
188        /* Receive a message from queue_2 */
189        tx_queue_receive (&queue_2, receive_message_2, TX_WAIT_FOREVER);
190  
192  
194    /* Increment the thread counter and get timing info */
195    speedy_thread_counter++;
196    current_time = tx_time_get();
197    cycle_time = current_time-start_time;
198    total_speedy_time = total_speedy_time + cycle_time;
199  }
Scheduling Enables Real-Time Systems to Achieve Higher Performance

199 }
200 /*******************************************************/
201 /* Entry function definition of slow_thread */
202 /* it has a lower priority than speedy_thread */
203 void slow_thread_entry(ULONG thread_input)
204 {
205 
206 ULONG start_time, current_time=0, cycle_time=0;
207 UINT status;
208 
209 while(1)
210 {
211 
212 /* Get the starting time for this cycle */
213 start_time=tx_time_get();
214 
215 /* Activity 3: sleep 12 ticks. */
216 /* wait for a message to appear on either one of the two queues */
217 tx_semaphore_get (&gatekeeper, TX_WAIT_FOREVER);
218 
219 /* Determine whether a message queue_1 or queue_2 is available */
220 status = tx_queue_receive (&queue_1, receive_message_1, TX_NO_WAIT);
221 
222 if (status == TX_SUCCESS)
223 ; /* A message on queue_1 has been found-process */
224 else
225 /* Receive a message from queue_2 */
226 tx_queue_receive (&queue_2, receive_message_2, TX_WAIT_FOREVER);
227 
228 tx_thread_sleep(12);
229 
230 /* Activity 4: 8 ticks. */
231 tx_thread_sleep(8);
232 
233 /* Increment the thread counter and get timing info */
234 slow_thread_counter++;
235 
236 current_time = tx_time_get();
237 cycle_time = current_time-start_time;
238 total_slow_time = total_slow_time + cycle_time;
239 }
240 
241 /* print statistics at specified times */
242 void print_stats (ULONG invalue)
243 {
244 
245 ULONG current_time, avg_slow_time, avg_speedy_time;
246 
247 if ((speedy_thread_counter>0) && (slow_thread_counter>0))
248 {
249 
250 current_time = tx_time_get();
251 avg_slow_time = total_slow_time/slow_thread_counter;
252 avg_speedy_time = total_speedy_time/speedy_thread_counter;
253 tx_queue_info_get (&queue_1, &info_queue_name, &enqueued_1,
Scheduling Enables Real-Time Systems to Achieve Higher Performance

258 &available_storage, &first_suspended,
259 &suspended_count, &next_queue);
260 tx_queue_info_get (&queue_2, &info_queue_name, &enqueued_2,
261 &available_storage, &first_suspended,
262 &suspended_count, &next_queue);
263 printf("\nEvent-Chaining: 2 threads waiting for 2 queues\n\n");
264 printf(" Current Time: %lu\n", current_time);
265 printf(" speedy_thread counter: %lu\n", speedy_thread_counter);
266 printf(" speedy_thread avg time: %lu\n", avg_speedy_time);
267 printf(" slow_thread counter: %lu\n", slow_thread_counter);
268 printf(" slow_thread avg time: %lu\n", avg_slow_time);
269 printf(" total # queue_1 messages sent: %lu\n", send_message_1[TX_1_ULONG-1]);
270 printf(" total # queue_2 messages sent: %lu\n", send_message_2[TX_1_ULONG-1]);
271 printf(" current # messages in queue_1: %lu\n", enqueued_1);
272 printf(" current # messages in queue_2: %lu\n", enqueued_2);
273 }
274 else printf("Bypassing print_stats function, Current Time: %lu\n",
275 tx_time_get());
276 }
277
278 }  /* Send a message to queue_1 at specified times */
279 void queue_timer_1_entry (ULONG invalue)
280 {
281 /* Send a message to queue_1 using the multiple object suspension approach */
282 /* The gatekeeper semaphore keeps track of how many objects are available
283 via the notification function */
284 send_message_1[TX_1_ULONG-1]++;
285 tx_queue_send (&queue_1, send_message_1, TX_NO_WAIT);
286 }
287
288 }  /* Send a message to the queue at specified times */
289 void queue_timer_2_entry (ULONG invalue)
290 {
291 /* Send a message to queue_2 using the multiple object suspension approach */
292 /* The gatekeeper semaphore keeps track of how many objects are available
293 via the notification function */
294 send_message_2[TX_1_ULONG-1]++;
295 tx_queue_send (&queue_2, send_message_2, TX_NO_WAIT);
296 }
297
298 }  /* 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 */
319    void queue_2_send_notify(TX_QUEUE *queue_ptr_2)
320    {
321        Tx_semaphore_put (&gatekeeper);
322    }

END Example code. Figure 5. Comments about sample system listing
contains several comments about this listing, using the line numbers as references.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>024 through 035</td>
<td>Declaration of system resources including threads, byte pool, semaphore, queues, and timers</td>
</tr>
<tr>
<td>037 through 047</td>
<td>Declaration of variables used in the system including parameters for the queue info get services</td>
</tr>
<tr>
<td>049 through 060</td>
<td>Declaration of prototypes for thread entry functions, timer entry function, and event notification functions</td>
</tr>
<tr>
<td>116 through 127</td>
<td>Creation of the two queues used for multiple object suspension</td>
</tr>
<tr>
<td>129 and 130</td>
<td>Creation of the gatekeeper semaphore used for Event-Chaining</td>
</tr>
<tr>
<td>132 through 142</td>
<td>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</td>
</tr>
<tr>
<td>144 through 150</td>
<td>Registration of the two functions that increment the gatekeeper semaphore whenever messages are sent to the queues</td>
</tr>
<tr>
<td>159 through 199</td>
<td>Entry function for Speedy Thread; lines 178 through 191 contain the implementation of Activity 2</td>
</tr>
<tr>
<td>203 through 244</td>
<td>Entry function for Slow Thread; lines 218 through 231 contain the implementation of Activity 3</td>
</tr>
<tr>
<td>247 through 276</td>
<td>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</td>
</tr>
</tbody>
</table>
Entry functions for timers to send messages to queue_1 and queue_2 at periodic intervals

Entry functions for the notification callback functions; these functions increment semaphore gatekeeper whenever a message is sent 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:

<table>
<thead>
<tr>
<th>Event-Chaining: 2 threads waiting for 2 queues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Time: 500</td>
</tr>
<tr>
<td>speedy_thread counter: 69</td>
</tr>
<tr>
<td>speedy_thread avg time: 7</td>
</tr>
<tr>
<td>slow_thread counter: 24</td>
</tr>
<tr>
<td>slow_thread avg time: 20</td>
</tr>
<tr>
<td>total # queue_1 messages sent: 41</td>
</tr>
<tr>
<td>total # queue_2 messages sent: 55</td>
</tr>
<tr>
<td>current # messages in queue_1: 0</td>
</tr>
<tr>
<td>current # messages in queue_2: 1</td>
</tr>
</tbody>
</table>

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.