Electronic Design

By BILL LAMIE, President and CEO, PX5 RTOS

Proprietary RTOS APIs Impede Development

Open standard RTOS application programming interfaces, such as POSIX pthread, help enhance software development.

A quick internet search shows a seemingly endless number of real-time operating systems (RTOSes), most based on proprietary application programming interfaces (APIs). That's certainly true for the more popular embedded RTOSes, e.g., FreeRTOS, ThreadX, and Zephyr. Some of these proprietary RTOS APIs provide adequate functionality.

However, each proprietary API impedes embedded development by requiring extensive developer training, constricting cross-platform code sharing, and effectively locking the application to the proprietary RTOS.

Developer training is time-consuming and expensive. In addition, proprietary API usage errors resulting in product defects are all too common. For device makers with MPU- (Linux) and MCU- (RTOS) based designs, sharing code between platforms is difficult, if not impossible.

Finally, since the application is locked into the RTOS, it's at the mercy of what processor and development tools are supported by that particular RTOS. Not having the ability to freely migrate your application to another hardware platform or development tool might be the most significant impediment of all.

But instead of focusing on the negative aspects of proprietary RTOS APIs, it's more productive to focus on solutions. Interestingly, the answer has been around since 1995. This is when the POSIX Threads standard (commonly called pthreads) was introduced (IEEE 1003.1c-a995). Not only is this an international standard, it's also the multithreading API in every embedded Linux distribution. Hence, POSIX pthreads is already the most popular API standard in the embedded industry.

Is the POSIX pthread API as functionally capable as proprietary RTOS APIs? Here's a brief overview of the POSIX pthread API alongside the most popular proprietary RTOS APIs.

Creating Threads in POSIX

Creating threads (also called tasks) is the most fundamental RTOS primitive. The POSIX *pthread create* API is ideally suited for embedded applications. Having only four arguments, it's much simpler than other popular RTOS APIs. The POSIX pthread API offers an optional attribute specification if additional configuration is required. Here are the thread creation APIs for POSIX pthreads, ThreadX, FreeRTOS, and Zephyr:

```
  IEEE POSIX pthread standard
           int   pthread_create(pthread_t* thread_handle,
                        pthread_attr* attributes,
                        void* (*thread_entry)(void *),
                        void * thread_arguments);

  Proprietary RTOS APIs

ThreadX:
      UINT  tx_thread_create(TX_THREAD*  thread_handle, CHAR * name,
                        VOID  (*thread_entry)(ULONG), ULONG thread_argument,
                        VOID *  stack_memory,  ULONG  stack_size,
                        UINT  priority, UINT preemption_threshold,
```

```
                            ULONG time_slice, UINT auto_start);
FreeRTOS:
       BaseType_t  xTaskCreateStatic(TaskFunction_t  pvTaskCode,
                            const char*  const  pcName,
                            const uint32_t  ulStackDepth,
                            void* const pvParameters, UBaseType_t  uxPriority,
                            StackType_t*  const puxStackBuffer,
                            StaticTask_t* const pxTaskBuffer);
Zephyr:
       k_tid_t  k_thread_create(struct k_thread* new_thread,
       k_thread_stack_t *stack,
                            size_t  stack_size, k_thread_entry_t thread_entry,
                            void* p1, void*  p2, void*  p3, int prio,
                            uint32_t  options, k_timeout_t delay);
```

USING MUTUAL EXCLUSION WITH POSIX

Mutual exclusion is required in embedded systems to coordinate access to shared resources. The mutual-exclusion primitives in POSIX are similar to the APIs in other RTOS. One difference in POSIX is that additional APIs offer timeouts for waiting on a mutex. Most proprietary RTOS APIs have an additional parameter that determines the timeout. Here are the typically used mutual-exclusion APIs for POSIX pthreads, ThreadX, FreeRTOS, and Zephyr:

```
  IEEE POSIX prthread API for creating a mutex
  POSIX:
  int  pthread_mutex_init(pthread_mutex_t*  mutex_handle,
                            pthread_mutexattr_t* mutex_attributes);

  Proprietary RTOS API for creating a mutex
  ThreadX:
  UINT  tx_mutex_create(TX_MUTEX* mutex_handle, CHAR* name,
                            UNIT inheritance_option);
FreeRTOS:
  SemaphoreHandle_t  xSemaphoreCreateMutex(void);

  Zephyr:
  int  k_mutex_init(struct k_mutex*  mutex);

  IEEE POSIX pthread API for locking a mutex
            int  pthread_mutex_lock(pthread_mutex_t*  mutex_handle);

  Proprietary RTOS API for locking a mutex
  ThreadX:
  UINT  tx_mutex_get(TX_MUTEX* mutex_handle, ULONG suspend_option);

  FreeRTOS:
  UBaseType_t  xSemaphoreTake(SemaphoreHandle_t  xSemaphore,
                            TickType_t  xTicksToWait);
Zephyr:
  int  k_mutex_lock(struct k_mutex*  mutex, k_timeout_t  timeout);

  IEEE POSIX pthread API for unlocking a mutex
            int  pthread_mutex_unlock(pthread_mutex_t*  mutex_handle);
```

```
Proprietary RTOS API for unlocking a mutex
ThreadX:
UINT  tx_mutex_put(TX_MUTEX*  mutex_handle);

FreeRTOS:
UBaseType_t  xSemaphoreGive(SemaphoreHandle_t  xSemaphore);

Zephyr:
int  k_mutex_unlock(struct k_mutex*  mutex);
```

MULTIPLE THREAD SYNCHRONIZATION USING POSIX

Synchronizing the execution of multiple threads is an important RTOS primitive. A classic example is the producer-consumer paradigm, whereby one thread processes information produced by another thread or interrupt handler.

Counting semaphores is often utilized in a producer-consumer fashion. The thread responsible for processing the information waits for a semaphore. When the information is ready, the producer sends the semaphore.

POSIX semaphore APIs are similar to proprietary RTOS semaphore APIs. One difference is that POSIX has additional APIs when a timeout is required to wait for a semaphore. Most proprietary RTOS APIs have an additional parameter that determines the timeout. Here are the commonly used semaphore APIs for POSIX, ThreadX, FreeRTOS, and Zephyr:

```
IEEE POSIX API to create a semaphore
          int  sem_init(sem_t*  semaphore_handle, int  pshared,
                                   unsigned int  value);

Proprietary RTOS APIs to create a semaphore
ThreadX:
UINT  tx_semaphore_create(TX_SEMAPHORE*  semaphore_handle,
                       CHAR*  name, ULONG  initial_count);

FreeRTOS:
SemaphoreHandle_t  xSemaphoreCreateCounting(UBaseType_t
     uxMaxCount, UBaseType_t  unInitialCount);

Zephyr:
int  k_sem_init(struct k_sem* sem, unsigned int  initial_count,
                       unsigned int  limit);

IEEE POSIX API to obtain a semaphore
          int  sem_wait(stm_t*  semaphore_handle);

Proprietary RTOS APIs to obtain a semaphore
ThreadX:
UINT  tx_semaphore_get(TX_SEMAPHORE*  semaphore_handle,
                                   ULONG suspend_option);

FreeRTOS:
UBaseType_t  xSemaphoreTake(SemaphoreHandle_t  xSemaphore,
                                   TickType_t  xTicksToWait);
Zephyr:
int  k_sem_take(struct k_sem*  sem, k_timeout_t  timeout);
```

```
IEEE POSIX API to post a semaphore
           int  sem_post(sem_t*  semaphore_handle);

Proprietary RTOS API to post a semaphore
ThreadX:
UINT  tx_semaphore_put(TX_SEMAPHORE*  semaphore_handle);

FreeRTOS:
UBaseType_t  xSemaphoreGive(SemaphoreHandle_t  xSemaphore);

Zephyr:
int  k_sem_give(struct k_sem*  sem);
```

Thread Communication in POSIX

Communicating information between multiple threads for processing is another important RTOS primitive. The thread responsible for processing the information can wait for a message from a queue. When the message is available, the waiting thread is given the message for processing and is resumed.

POSIX message-passing APIs are similar to proprietary RTOS APIs. There are differences, though. POSIX supports variable-sized messages and message priority. POSIX also has additional APIs when suspension timeouts are required. Most proprietary RTOS APIs have an additional parameter that determines the message waiting timeout. Here are the commonly used message queue APIs for POSIX, ThreadX, FreeRTOS, and Zephyr:

```
IEEE POSIX Message Queue Create API
           mqd_t mq_open(char* queue_name, int operation, mode_t mode,
                                  mq_attr* attributes);

Proprietary RTOS Message Queue Create APIs

ThreadX:
UINT  tx_queue_create(TX_QUEUE*  queue_handle, CHAR* name,
                                  ULONG message_size, VOID* queue_memory,
                                  ULONG memory_size);

FreeRTOS:
QueueHandle_t  xQueueCreateStatic(UBaseType_t  uxQueueLength,
                                  UBaseType_t  uxItemSize,
                                  uint8_t*  pucQueueStorageBuffer,
                                  StaticQueue_t*  pxQueueBuffer);

Zephyr:          void  k_msgq_init(struct k_msgq*  msgq, char*  buffer,
                                  size_t  msg_size, uint32_t max_msgs);

IEEE POSIX Message Send API
           int  mq_send(mqd_t  queue_handle, char* message,
                 size_t  message_size, unsigned int  message_priority);

Proprietary RTOS Message Send API
ThreadX:
UINT  tx_queue_send(TX_QUEUE*  queue_handle,
                 VOID* message_pointer, ULONG  suspend_option);

FreeRTOS:
```

```
BaseType_T  xQueueSend(QueueHandle_t  xQueue,
                 void*  pvItemToQueue, TickType_t  xTicksToWait);

Zephyr:          int  k_msgq_put(struct k_msgq* msgq, const void* data, k_
timeout_t timeout);

IEEE POSIX Message Receive API
           int  mq_receive(mqd_t  queue_handle, char* message,
                 size_t  message_size, unsigned int*  message_priority);

Proprietary RTOS Message Receive APIs
ThreadX:
UINT  tx_queue_receive(TX_QUEUE* queue_handle,
                 VOID* message_pointer, ULONG  suspend_option);

FreeRTOS:
BaseType_t  xQueueReceive(QueueHandle_t  xQueue, void*  pvBuffer,
                                   TickType_t  xTicksToWait);

Zephyr:
void*  k_msgq_get(struct k_msgq*  msgq, void* data,
                                   k_timeout_t  timeout);
```

POSIX PTHREAD API VS. PROPRIETARY RTOS APIs

It's clear that the POSIX pthread API is similar and just as capable as proprietary RTOS APIs for the most common multithreading primitives associated with thread management, mutual exclusion, synchronization, and message passing. The POSIX API is actually more straightforward in some cases—like creating a thread. Any missing functionality in POSIX can be augmented with API extensions.

Since most developers have some experience with POSIX pthreads, training and usage errors are significantly reduced or even eliminated. Sharing code with embedded Linux and/or moving to another RTOS that supports POSIX pthreads is easy. RTOS migration to the industry-standard IEEE POSIX pthread API promises to reduce time-to-market and enhance code reuse—welcome advances in our embedded industry.

Bill Lamie is President/CEO of PX5 RTOS. Bill has been in the commercial RTOS space for over 30 years – first with Accelerated Technology (acquired by Siemens) and then with Express Logic (acquired by Microsoft). Bill was also the sole author of Nucleus and ThreadX.