This porting guide describes the procedure to port from InnoOS to FreeRTOS for multiple generic scenarios.

# Create Tasks (Threads)

**Note**: Argument order differs between os\_create\_thread()and xTaskCreate()

|  |  |
| --- | --- |
| **FreeRTOS** | **InnoOS** |
| BaseType\_t xTaskCreate(TaskFunction\_t pvTaskCode, const char \* const pcName, configSTACK\_DEPTH\_TYPE usStackDepth, void \*pvParameters, UBaseType\_t uxPriority, TaskHandle\_t \*pxCreatedTask); | struct os\_thread \*os\_create\_thread(const char \*name, os\_entrypoint\_t entry, os\_threadarg\_t arg, uint32\_t flags, size\_t stacksz); |
| static void task(void \*arg)  {  int cnt = 0;  for(;;) {  printf("Count %d\n", cnt++);  vTaskDelay(1000,  portTICK\_PERIOD\_MS);  }  }  int main(void)  {  xTaskCreate(  task,  "task",  1024,  NULL,  1,  NULL);  return 0;  } | static void \* task(void \*arg)  {  int cnt = 0;  for (;;) {  os\_printf("Count %d\n", cnt++);  os\_sleep\_us(SYSTIME\_SEC(1),  OS\_TIMEOUT\_WAKEUP);  }  return NULL;  }  int main(void)  {  os\_create\_thread(  "task",  task,  NULL,  1,  1024);  return 0;  } |

Table 1: Create tasks (threads)

**Note**: The stack size is represented as number of words (4 byte) in FreeRTOS and number of bytes in InnOS.

Message Queues

In InnoOS, every thread gets a message queue at os\_create\_thread(). Hence, the message queues are not created separately. The message queues have no maximum number of items, and the items can be of different sizes.

Message type is used to differentiate between messages. The sender specifies to which thread the message should be sent. The receiver does not specify the queue, as it receives from its own thread queue. The receiver has no reception timeout, but there is a flag which decides if the call should wait or return immediately if there is no message in the reception queue for the moment.

It is also possible to use os\_recvmsg\_type to wait for a specific message type.

Following is an example with a Tx task sending messages to an Rx task:

**InnoOS**

|  |
| --- |
| #define MSG\_TYPE 100  struct os\_thread \*thread\_rx;  struct os\_thread \*thread\_tx;  struct my\_msg {  struct os\_msg msg;  int data;  };  static void \* tx(void \*arg)  {  for (;;) {  struct my\_msg \*msg = (struct my\_msg \*)os\_msg\_alloc(MSG\_TYPE, sizeof \*msg);  msg->data = 0xaddababe;  os\_sendmsg(thread\_rx, &msg->msg);  os\_sleep\_us(SYSTIME\_SEC(1), OS\_TIMEOUT\_WAKEUP);  }  return NULL;  }  static void \* rx(void \*arg)  {  for (;;) {  struct my\_msg \*rec = (struct my\_msg \*)os\_recvmsg(false);  os\_printf("Received %x from %s\n", rec->data,  os\_thread\_name(rec->msg.msg\_sender));  os\_msg\_release((struct os\_msg \*)rec);  }  return NULL;  }  int main(void)  {  thread\_tx = os\_create\_thread("tx", tx, NULL, 1, 1024);  thread\_rx = os\_create\_thread("rx", rx, NULL, 1, 1024);  return 0;  } |

In FreeRTOS, the message queue is created separately. Has a maximum size and an element size. xQueueSend and xQueueReceive specify the message queue, with specific timeouts on the duration for block in case of failure.

**FreeRTOS**

|  |
| --- |
| #define Q\_SIZE 5 /\* number of items in queue \*/  #define I\_SIZE sizeof(int) /\* size of each item \*/  #define Q\_TX\_TO 500 /\* timeout for send if queue is full \*/  #define Q\_RX\_TO 500 /\* timeout for reception if queue is empty \*/  xQueueHandle msg\_queue;  static void tx(void \*arg)  {  int item = 0xaddababe;  for(;;) {  if (xQueueSend(msg\_queue, &item, Q\_TX\_TO))  printf(“Send ok\n”);  else  printf(“Send failed\n”);  vTaskDelay(1000, portTICK\_PERIOD\_MS);  }  }  static void rx(void \*arg)  {  int item;  for(;;) {  if (xQueueReceive(msg\_queue, &item, Q\_RX\_TO)  printf(“Received %x\n”, item);  else  printf(“Reception failed\n”);  }  }  int main(void)  {  msg\_queue = xQueueCreate(Q\_SIZE, I\_SIZE);  xTaskCreate(Rx, "Rx", 1024, NULL, 1, NULL);  xTaskCreate(Tx, "Tx", 1024, NULL, 1, NULL);  vTaskStartScheduler();  return 0;  } |

# Soft Timers

In InnoOS, the timers are called callouts, and the APIs are prefixed with callout\_.

The preferred coding style is to use a struct including the timer (callout) and the parameters needed. For example: cnt. In the callback function, the pointer to the struct is captured via the container\_of macro.

**InnoOS**

|  |
| --- |
| #include <kernel/os.h>  #include <kernel/callout.h>  struct my\_state {  struct callout timer;  uint32\_t cnt;  } state;  static void \_\_irq  timer\_callback(struct callout \*co)  {  struct my\_state \*state = container\_of(co, struct my\_state, timer);  state->cnt++;  os\_printf("cnt: %u\n", state->cnt);  if (state->cnt < 10)  /\* Reschedule the timer \*/  callout\_schedule(&state->timer, SYSTIME\_SEC(1));  else  os\_printf("Ready\n");  }  int main(void)  {  callout\_init(&state.timer, timer\_callback);  callout\_schedule(&state.timer, SYSTIME\_SEC(1));  return 0;  } |

In FreeRTOS, there is a built-in counter that determines the number of times the counter has elapsed. There is also an auto-reload option where the timer is automatically restarted. In this example we use it to illustrate a difference between FreeRTOS and InnoOS.

For both FreeRTOS and InnoOS, the timer callback function will execute in the timer interrupt service context, and no blocking calls must be used. Preferably the work is handed over to another task/thread.

**FreeRTOS**

|  |
| --- |
| static void timer\_callback(TimerHandle\_t timer)  {  uint32\_t cnt = (uint32\_t)pvTimerGetTimerID(timer);  cnt++;  printf("cnt: %u\n", cnt);  if (cnt < 10)  /\* timer is auto-reloaded \*/  vTimerSetTimerID(timer, (void\*)cnt);  else {  xTimerStop(timer, 0);  printf("Ready\n");  }  }  int main(void)  {  TimerHandle\_t timer;  timer = xTimerCreate(  "timer",  1000/portTICK\_PERIOD\_MS,  pdTRUE, /\* auto-reloaded \*/  (void\*) 0,  timer\_callback);  xTimerStart(timer,0);  } |

# Semaphores

The differences between FreeRTOS and InnoOS when it comes to semaphores are very small. lists the functions required.

|  |  |
| --- | --- |
| **FreeRTOS** | **InnoOS** |
| xSemaphoreHandle semaphore; | struct os\_semaphore semaphore; |
| semaphore = xSemaphoreCreateCounting(1, 10); | os\_sem\_init(&semaphore, 1); |
| semaphore = xSemaphoreCreateMutex(); | os\_sem\_init(&semaphore, 1); |
| xSemaphoreTake(semaphore, timeout); | os\_sem\_wait\_timeout(&semaphore, timeout); |
| xSemaphoreGive(semaphore); | os\_sem\_post(&semaphore); |

Table 2: Semaphore – differences

InnoOS has the API os\_sem\_wait (&semaphore), which is without timeout, and which blocks until the semaphore is taken. This is the same behavior which is achieved in FreeRTOS if:

1. INCLUDE\_vTaskSuspend is set to '1'
2. Setting the timeout in xSemaphoreTake to portMAX\_DELAY.

xSemaphoreTake() as well as os\_sem\_wait() and os\_sem\_wait\_timeout() must not be used in interrupt context.

# Work Queue

Work queues are used to schedule functions to run in a specific thread context. Most commonly used to defer work from an interrupt handler that needs to run quickly to another function that may do the more heavy processing involved in serving the interrupt.

But FreeRTOS doesn’t have built support for Work Queue. However, this can be easily implemented using a thread and a message queue.

# Wait Queue

Wait queue is used for a task/thread to wait for an event.

But FreeRTOS doesn’t have built-in support for wait queue. However, this can be easily implemented using semaphores and message queue.