This porting guide describes porting of the ZephyrOS code to InnoOS for a few common scenarios.

## ZephyrOS to InnoOS

### Create Threads

In InnoOS, a thread is created using the function os\_create\_thread(..). The function os\_create\_thread, takes five input parameters:

1. Name of the thread to create
2. Entry function of the thread task
3. Argument(s) of the entry function
4. Priority of the thread
5. Stack size of the thread

Following is the prototype of the function os\_create\_thread:

|  |
| --- |
| struct os\_thread \*os\_create\_thread(const char \*name, os\_entrypoint\_t entry, os\_threadarg\_t arg, uint32\_t flags, size\_t stacksz); |

Contrary to ZephyrOS, in InnoOS the entry function always returns NULL. It is not possible to pass more than one argument to the entry function. An alternative is to use a pointer to a struct to pass more than one argument.

The following example provides a comparison between Zephyr and InnoOS, on how a thread is created:

|  |  |
| --- | --- |
| **ZephyrOS** | **InnoOS** |
| static void task(void)  {  int cnt = 0;  for (;;){  printk(“Count %d\n”,cnt++);  k\_sleep(K\_SECONDS(1));  }  }    int main(void)  {  K\_THREAD\_STACK\_DEFINE(stack,1024);  static struct k\_thread my\_thread;  k\_thread\_create(  &my\_thread,  stack,  K\_THREAD\_STACK\_SIZEOF(stack),  (k\_thread\_entry\_t) task,  NULL,  NULL,  NULL,  1,  0,  K\_NO\_WAIT) ;  return 0;  } | static void \* task(void \*arg)  {  int cnt = 0;  for (;;) {  os\_printf("Count %d\n", cnt++);  os\_sleep\_us(SYSTIME\_SEC(1),  }  return NULL; }  int main(void)  {  os\_create\_thread(  "task",  task,  NULL,  1,  1024);  return 0;  } |

Table 1: Create a thread

### Message Queues

In ZephyrOS, the message queue must be initiated separately. To do so, it is required to allocate a message buffer that is aligned. In addition to this, in ZephyrOS, the receiver is required to specify the message queue from which it wants to get the message from.

There is also an option to peek a message from the queue, without removing it from the queue.

In InnoOS, every thread gets a message queue when a thread is created (i.e., after calling os\_create\_thread()). Hence, the message queues are not created separately.

In InnoOS, message queues have no maximum number of items, and each message on the queue can be of different sizes. InnoOS uses the message type (field) 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 it from its own thread queue. The receiver has no reception timeout, but there is a flag which tells if the call should wait or return immediately if there is no message in the reception queue for the moment.

Additionally, in InnoOS, it is also possible to use an os\_recvmsg\_type to wait for a specific message type.

|  |  |
| --- | --- |
| **ZephyrOS** | **InnoOS** |
| struct k\_msgq msgq;  K\_THREAD\_STACK\_DEFINE(stack1, 1024);  K\_THREAD\_STACK\_DEFINE(stack2, 1024);    static void tx()  {  int item = 0xaddababe;  for(;;) {  if (k\_msgq\_put(&msgq, &item,  K\_SECONDS(0.5))==0)  printk("Send ok\n");  else  printk("Send failed\n");  k\_sleep(K\_SECONDS(1));  }  }    static void rx()  {  int item;  for(;;) {  if (k\_msgq\_get(&msgq, &item,  K\_SECONDS(0.5))==0)  printk("Received %x\n",item);  else  printk("Reception failed\n");  k\_sleep(K\_SECONDS(1));  }  }    int main()  {  char \_\_aligned(4) msg\_buffer[5\*sizeof(void\*)];  k\_msgq\_init(&msgq, msg\_buffer, sizeof(void\*), 5);    static struct k\_thread thread\_rx;  static struct k\_thread thread\_tx;    k\_thread\_create(&thread\_tx, stack1,  K\_THREAD\_STACK\_SIZEOF(stack1),  (k\_thread\_entry\_t) tx, NULL, NULL, NULL,  1, 0, K\_NO\_WAIT) ;  k\_thread\_create(&thread\_rx, stack2,  K\_THREAD\_STACK\_SIZEOF(stack2),  (k\_thread\_entry\_t) rx, NULL, NULL, NULL,  1, 0, K\_NO\_WAIT) ;  return 0;  } | #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;  } |

Table 2: Message queues

### Soft Timers

Table 3 depicts an example where the timer is run multiple times. ZephyrOS does not have a callback which counts the number of times the timer has elapsed in total. It only has a status function which checks the number of times it has elapsed since the status was last read and then resets it to zero. Because of this, there is a need to have a global count which will keep track of the number of times the timer has elapsed in total.

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 this example.

In the callback function, the pointer to the struct is returned via the container\_of macro. This will lead to a lot of different coding opportunities with InnoOS.

In InnoOS, there is an additional function called callout\_scedule\_at(struct callout \*co, uint32\_t time)which offers the possibility to schedule the callout to start after a few microseconds.

|  |  |
| --- | --- |
| **ZephyrOS** | **InnoOS** |
| uint32\_t cnt=0;  struct k\_timer timer;    static void  timer\_callback()  {  cnt+=k\_timer\_status\_get(&timer);  printk("cnt: %u\n", cnt);  if (cnt<10)  /\* timer is auto-reloaded \*/  k\_timer\_start(&timer, K\_SECONDS(1),  K\_NO\_WAIT);  else {  k\_timer\_stop(&timer);  printk("Ready!\n");  }  }    int  main(void)  {  k\_timer\_init(&timer, timer\_callback, NULL);  k\_timer\_start(&timer, K\_SECONDS(1),  K\_NO\_WAIT);  return 0;  } | #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;  } |

Table 3: Soft timers

### Semaphores

The differences between ZephyrOS and InnoOS are minor when it comes to semaphores. Table 4 depicts the different functions.

|  |  |
| --- | --- |
| **ZephyrOS** | **InnoOS** |
| struct k\_sem semaphore;  k\_sem\_init(&semaphore, 1, MAX\_VALUE);  k\_sem\_take(&semaphore, timeout);  k\_sem\_give(&semaphore); | struct os\_semaphore semaphore;  os\_sem\_init(&semaphore, 1); os\_sem\_wait\_timeout(&semaphore, timeout);  os\_sem\_post(&semaphore); |

Table 4: Semaphores

InnoOS has the API os\_sem\_wait(&semaphore), which is without a timeout and blocks until the semaphore is taken. The same behavior can be achieved in ZephyrOS using k\_sem\_take, if the timeout argument is set to K\_MAX\_FOREVER.

### Work Queue

ZephyrOS and InnoOS work queues are similar. They can write to the systems work queue and create new work queues. Following is an example of both writing to the systems work queue:

|  |  |
| --- | --- |
| **ZephyrOS** | **InnoOS** |
| struct my\_state{  struct k\_work mining;  };  static void  working\_in\_a\_coalmine(struct k\_work \*work)  {  struct my\_state \*state = CONTAINER\_OF(work,  struct my\_state, mining);    // do some work...  }  static void  interrupt\_service\_receive(struct my\_state \*state)  {  k\_work\_submit(&state->mining);  }  static void  init\_mining(struct my\_state \*state)  {  //Associate the work function with the struct k\_work object  k\_work\_init(&state->mining, working\_in\_a\_coalmine);  } | struct my\_state {  struct os\_work mining;  };  static void  working\_in\_a\_coalmine(struct os\_work \*work)  {  struct my\_state \*state = container\_of(work, struct my\_state,  mining);    // do some work...  }  static void  interrupt\_service\_receive(struct my\_state \*state)  {  os\_queue\_system\_work(&state->mining);  }  static void  init\_mining(struct my\_state \*state)  {  //Associate the work function with the struct os\_work object  os\_init\_work(&state->mining, working\_in\_a\_coalmine);  } |

Table 5: Work queue