/* The using System line means that you are using the System library in your * project. Which gives you some useful classes and functions like Console class * or the WriteLine function/method. */ using System;

/* Include the Dictionary, to extend a Van OS message with generic fields. */ using System.Collections.Generic;

/* The Thread class is defined in the System. Threading namespace that must be * imported before you can use any threading related types, like Mutex. */ using System.Threading;

/* The namespace VOS - Van OS - is used to organize its code, and it is a * container for VOS classes. Namespace also solves the problem of naming * conflict. / *namespace VOS {* / Static class to hold global members, etc. / *static class Globals {* / Use the static modifier to declare a static member, which * belongs to the type itself rather than to a specific * object. */

```
    /* If the condition being tested is not met, an exception is
     * thrown. */
    public static void Assert(bool cond) {
        /* Test the exception condition. */
        if (! cond)
            return;

        /* Raised when a method call is invalid in an object's
         * current state. */
        throw new InvalidOperationException("Invalid operation");
    }
}

/* Everything in C# VOS is associated with thread or thread input queue
 * classes and objects, along with its attributes and methods. */

/* T_msg is the message class of a Van OS thread input queue with
 * i class members: j fields and k methods. */
class T_msg  {
    /* A delegate is an object which refers to a method or you can
     * say it is a reference type variable that can hold a reference
     * to the methods. Delegates in C# are similar to the function
     * pointer in C/C++. It provides a way which tells which method
     * is to be called when an event is triggered. */
            public delegate void Tm_cb(T_msg msg);

    /* Successor of the thread input queue. */
    public T_msg  next;

    /* Callback or reference to a function to process the input
     * message. It is expected to execute this field at a given time
     * in the thread context. */
    public Tm_cb  cb;

    /* Dictionary is a generic collection which is generally
     * used to store key/value pairs: declare a Dictionary
     * containing just the type object and then cast your
     * results. */
    public Dictionary<string, object>  param;

    /* The constructor T_msg() is a special method that is used
     * to initialize the T_msg object implicitely associated with
     * the new VOS.T_msg(() method. */
    public T_msg(Tm_cb msg_cb) {
        /* Save the  reference to a function to process the input
         * message in the thread context. */
        cb   = msg_cb;

        /* Create the Dictionary object, to extend the input
         * message with any key value pair optinally: e.g.
         * msg.param.Add("name",  "msg_1");
         * msg.param.Add("count", 0); ... */
        param = new Dictionary<string, object>();
    }

    /* You stop referencing them and let the garbage collector take
     * them. When you want to free the object, add the following
     * line: obj = null; The the garbage collector if free to delete
     * the object (provided there are no other pointer to the object
     * that keeps it alive.) */
}

/* T_queue is the input queue class of a Van OS thread with i class
 * members: j fields and k methods. */
class T_queue {
    /* Synchronize the access to the protected message queue. */
    public Mutex  mutex;

    /* First empty queue element. */
    private T_msg  anchor;

    /* Last empty queue element. */
    private T_msg  stopper;

    /* Current number of the input queue elements. */
    public int    count;
```

```
/* If 1, a client executes Tq_send(). */
public int    busy_send;

/* Calculate the next message from the thread input queue. */
public T_msg Tq_receive() {
    T_msg  msg;

    /* Wait until it is safe to enter the critical section. */
    this.mutex.WaitOne();

    /* Test the queue state. */
    if (this.count < 1) {
        /* Release the mutex to leave the critical section. */
        this.mutex.ReleaseMutex();

        /* The thread input queue is empty. */
        return null;
    }

    /* Update the number of the queue elements. */
    this.count--;

    /* Get the first queue element. */
    msg = this.anchor.next;

    /* Calculate the new queue start. */
    this.anchor.next = msg.next;

    /* Test the outside located limits of the input
     * queue. */
    if (this.count < 1) {
        /* Test the queue state. */
        Globals.Assert(this.anchor.next != this.stopper);

        /* As of now, the input queue is empty. */
        this.stopper.next = this.anchor;
    }

    /* Release the mutex to leave the critical section. */
    this.mutex.ReleaseMutex();

    /* Return the reference to current input message. */
    return msg;
}

/* Extend the input queue of a Van OS thread and resume it. */
```

```
public void Tq_send(Thd t, T_msg msg) {
    int is_running;

    /* Entry condition. */
    Globals.Assert(t == null || msg == null ||
             msg.cb == null);

    /* Change the state of this operation. */
    this.busy_send = 1;

    /* Wait until it is safe to enter the critical section
     * of the input queue. */
    this.mutex.WaitOne();

    /* Insert the new message at the end of the queue. */
    this.stopper.next.next = msg;
    msg.next = this.stopper;
    this.stopper.next = msg;

    /* Update the number of the queue elements. */
    this.count++;

    /* Change the execution state of the thread. */
    is_running = t.is_running;
    t.is_running = 1;

    /* Release the mutex to leave the critical section. */
    this.mutex.ReleaseMutex();

    /* Test the execution state of the thread. */
    if (is_running == 0) {
        /* Resume this thread blocked in the thread
         * control semaphore. */
        t.suspend_this.Release();
    }

    /* Change the state of this operation. */
    this.busy_send = 0;
}

/* The constructor T_queue() is a special method that is used
 * to initialize the T_queue object implicitely associated with
 * the new VOS.T_queue() method. */
    public T_queue() {

    /* Create the mutex, to synchronize the access to the
```

```
         * protected message queue. */
        this.mutex = new Mutex();

        /* To create the queue element object, use the keyword new with
         * the initialization arguments: */

        /* Define the excluded limits of a Van OS thread input
         * queue like ] ... [ or first and last element of the
         * Van OS thread input queue. Note: these messages shall
         * never be consumed, therefore the new argument
         * callback is null. */
        this.anchor = new VOS.T_msg(null);
        this.stopper = new VOS.T_msg(null);

        /* Link the first and last input queue element. */
        this.anchor.next  = this.stopper;
        this.stopper.next = this.anchor;

        /* Initialize the boundary conditions of a thread input
         * queue. */
        this.count = 0;
    }
}

/* Thd is the Van OS thread class with i class members: j fields and k
 * methods. */
class Thd {
    /* The public keyword is called an access modifier, which
     * specifies that the fields are accessible for other classes. */

    /* Execution states of a Van OS thread. */
    public enum Exec_s { VOS_THD_BOOT, VOS_THD_READY, VOS_THD_KILL,
        VOS_THD_INV };

    /* Initial execution state of a Van OS thread. */
    public Exec_s  exec_s;

    /* Synchronize the access to the multi thread access to the
     * thread state. */
    public Mutex  mutex;

    /* Name of a Van OS thread. */
    public string  name;

    /* Reference to a C sharp thread object.
     * Once the task assigned to a Thread is completed, that thread
     * will be terminated and we don’t need to worry about it. */
    public Thread  thd;

    /* Input queue of a Van OS thread. */
    public T_queue  queue;

    /* A semaphore, that shall suspend this thread, installed by the
     * superordinate thread. */
    public Semaphore  suspend_this;

    /* The parent shall be blocked until this thread has been
     * started. */
    public Semaphore  suspend_p;

    /* 1, if this thread is running on any CPU. */
    public int is_running;

    /* Extend the input queue of a Van OS thread and resume it. */
    public void Thd_send(T_msg msg) {
        /* Extend the input queue of a Van OS thread and resume it. */
        this.queue.Tq_send(this, msg);
    }

    /* Private members are accessible only within the body of the
     * class or the struct in which they are declared. */

    /* Process all current messages. */
    private void Thd_receive() {
        T_msg  msg;

        /* Process the received messages. */
        for (;;) {
            /* Get the next thread input message. */
            msg = this.queue.Tq_receive();

            /* Test the message state. */
            if (msg == null)
                break;

            /* Execute the message actions. */
            msg.cb(msg);

            /* Free the consumed message. */
            msg = null;
        }
    }
```

```
/* Suspend the thread, if the message queue is empty or it is
 * alive. */
private bool Thd_suspend() {
    T_queue  q;
    int  count;

    /* Get the reference to the thread input queue. */
    q = this.queue;

    /* Wait until it is safe to enter the critical section
     * of the input queue. */
    q.mutex.WaitOne();

    /* If the input queue is empty, prepare the suspend
     * operation for this thread. */
    this.is_running = 0;

    /* Copy the filling level of the queue. */
    count = q.count;

    /* Leave the critical section. */
    q.mutex.ReleaseMutex();

    /* If the input queue is empty, suspend the thread or
     * terminate the thread. */
    if (count < 1) {
        /* Test the shutdown request for this thread. */
        if (this.exec_s == Exec_s.VOS_THD_KILL)
            return false;

        /* Suspend this thread, until it is resumed by a
         * input message or shutdown trigger. */
        this.suspend_this.WaitOne();
    }

    /* Precondition: either at least there is a pending
     * input message or this thread shall be killed. */

    /* This thread is running on any CPU. */
    this.is_running = 1;

    /* Test the shutdown request for this thread. */
    if (this.exec_s == Exec_s.VOS_THD_KILL)
        return false;
    /* Final condition: there are pending input messages
     * and this thread is alive. */
    return true;
}

/* A new thread shall execute the callback method Tm_cb(). */
public void Thd_cb() {
    /* Update the Van OS thread state. */
    exec_s = Exec_s.VOS_THD_READY;

    /* Resume the parent thread, which has created and
     * started this thread. */
    this.suspend_p.Release();

    /* Process all received messages or accept the shutdown
     * request for this thread. */
    for (;;) {
        /* Suspend the thread, if the message queue is
         * empty or the thread is alive. */
        if (! Thd_suspend()) {
            /* This thread shall be killed. */
            break;
        }

        /* Process all current messages. */
        Thd_receive();
    }
}

/* Shutdown a Van OS thread. */
public void Thd_destroy() {
    /* Get, modify and test the thread state. */
    this.mutex.WaitOne();

    /* Entry condition. */
    Globals.Assert(this.exec_s != Exec_s.VOS_THD_READY);

    /* Change the thread state. */
    this.exec_s = Exec_s.VOS_THD_KILL;

    /* Test the state of the send operation. */
    Globals.Assert(this.queue.busy_send != 0);

    /* Leave the critical section. */
    this.mutex.ReleaseMutex();
    /* Resume this Van OS thread in Thd_suspend(). */
    this.suspend_this.Release();

}

/* A constructor is a special method that is used to initialize
 * objects. The advantage of a constructor, is that it is called
 * when an object of a class is created. It can be used to set
 * initial values for fields: */
public Thd(string n) {
    /* Initial the execution state of a Van OS thread. */
    exec_s = Exec_s.VOS_THD_BOOT;

    /* Define the thread name. */
    name = n;

    /* Create the mutex to Synchronize the access to the
     * multi thread access to the thread state. */
    mutex = new Mutex();

    /* Create a semaphore that can satisfy up to 1
     * concurrent request. Use an initial count of zero, so
     * that the entire semaphore count is initially owned by
     * this thread. */
    suspend_this = new Semaphore(0, 1);

    /* Create a semaphore to suspend the parent thread. */
    suspend_p = new Semaphore(0, 1);

    /* Allocate a C sharp thread object. */
    thd = new Thread(new ThreadStart(Thd_cb));
    Console.WriteLine("{0}: exec_s = {1}", this.name, this.exec_s);

    /* Allocate the input queue for this thread. */
    queue = new T_queue();

    /* Start the thread, which shall execute the callback
     * method Thd_cb(). */
    thd.Start();

    /* The parent thread shall be blocked until this thread
     * has been started. */
    suspend_p.WaitOne();

    /* Exit condition. */
```

```
        Globals.Assert(this.exec_s != Exec_s.VOS_THD_READY);
    }
}

/* Test the thread concept. */
class Prg {
    /* A semaphore, that shall suspend the Main() thread. */
    private static Semaphore  suspend;

    /* Max. number of the generated test messages. */
    private const int limit = 4;

    /* Current number of the generated test messages. */
    private static int count;

            /* Define the method to process a thread input message. */
            private static void msg_cb(T_msg msg) {
        string  s;
        int  i;

        /* Use the Dictionary key as index to get the value, but
         * then you need to cast the results. */
        s = (string) msg.param["name"];
        i = (int)    msg.param["count"];

        Console.WriteLine("msg_cb: name = {0}, count = {1}", s, i);

        /* Update the message counter. */
        count++;

        /* Test the exit condition for the main thread. */
        if (count < limit)
            return;

        /* Resume the main thread. */
        suspend.Release();
            }

    /* The Main method is the entry point of a C# application. When the
     * application is started, the Main method is the first method that is
     * invoked. */
    static void Main() {
        /* Spefify a Van OS thread. */
        Thd thd_x;

        /* Current message counter. */
```

```
int  i;

/* Spefify a Van OS thread input message. */
T_msg  msg;

/* Message name. */
string  n;

/* Create the thread control semaphore. */
suspend = new Semaphore(0, 1);

/* Initialize the message counter. */
count = 0;

/* To create a Van OS thread object, specify the class
 * name, followed by the object name, and use the keyword new: */
thd_x = new Thd("thd_x");

/* Produce some messages for the thread input queue. */
for (i = 0; i < limit; i++) {
    /* Create the input message for the Van OS
     * thread with the message processing method
     * msg_cb(), see above. */
    msg = new T_msg(msg_cb);

    /* String.Format performs the same operation a C
     * snprintf(), but do note that the format
     * strings are in a different format. */
    n = string.Format("msg-{0}", i);

    /* Extend the Van OS input message with generic
     * parameters: add further message fields or
     * key/value pairs to the message Dictionary
     * using the Dictionary Add() method. */
    msg.param.Add("name",  n);
    msg.param.Add("count", i);

    /* Extend the input queue of a Van OS thread and resume it. */
    thd_x.Thd_send(msg);
}

/* Suspend the main thread, until the test thread has
 * done its job. */
suspend.WaitOne();

/* Shutdown the test thread. */
```

```
        thd_x.Thd_destroy();

        /* Force garbage collector to run. */
        GC.Collect();

        /* Suspends the current thread until the thread that is
         * processing the queue of finalizers has emptied that
         * queue. */
        GC.WaitForPendingFinalizers();
    }
}
```

}