**Rabit Multitrhreaded Manager**

Harry Direen, PhD

Aug. 29, 2015

Complex software systems such as robotics and drone control may be broken into simpler, easier to design and code modules. There is a real advantage to break the system into relatively independent modules that run on separate threads. Breaking a system into relatively independent modules allows the designer and coder to focus one aspect of the system’s operation without having to worry about how other aspects of the system operate. Running modules in separate threads allows the system to take advantage of the additional power of multiple core computers. The problem is that multithreaded systems are more difficult to design and there are issues of thread synchronization and the protection of shared resources that if not handled properly will cause insidiously difficult bugs in the system. The Rabit Multithreaded Management system is designed to hide most of the difficulties of building a multithreaded system.

The Rabit Multithreaded Management system concept and design was developed in part at the US Air Force Academy’s UAS (Unmanned Ariel Systems) research group for controlling UAVs (Unmanned Ariel Vehicles) or drones. I will use examples from the control of UAVs/drones to provide concrete examples of how to use the Rabit system.

The Rabit system is designed around managers. Each manager runs on a separate thread which is designed to handle the operation of some aspect of the system. For instance, the control software for a drone has a number of aspects or functions it must handle. The drone has an avionics system that controls the flight of the aircraft. The avionics system takes inputs such as where to fly the aircraft to next, the altitude of flight, and the air speed. The avionics system will report back the aircrafts current location, altitude, air speed, and other parameters. One aspect of the drone’s software control system is to communicate with the avionics system. Another aspect of the drone’s software system is to communicate with the ground station and possibly with other drones. A further aspect of the software control system is to make decisions on where to fly the aircraft based upon mission goals. The drone’s software control system can become very complex quickly as the functional requirements increase.

One way to break the drone’s control system into separate modules is to have a module for handling all the communications with the avionic’s system; a module for communicating with the ground station and other drones; and a module for mission control, or desiring where to fly the drone next. Using the Rabit system, a manager is established for each module. This allows each module to run on a separate thread and handle its operation independently of the other modules.

Blah… blah… blah…

**Hello World Example**

It would be wrong not to have a “Hello World” Example… so here we go. This example will create one Rabit Manager that prints out “Hello World”.

* Start a new .NET or Mono command line project and add “Rabit” to the references.
* Create a new class / file with the name: “PrintManager” “and add a “using Rabit” statement.
* Change the Class PrintManager to inherit from RabitManager:

PrintManager : RabitManager

* Add a constructor as shown below. A manager name is established with the constructor. The only requirement is that the manager name be unique for all managers in the system you build. Using the same name as the class works well:

public PrintManager ()

: base("PrintManager")

{

}

* Override the “ExecuteUnitOfWork()” method and simply add the print “Hello World” statement. The ExecuteUnitOfWork method is the only method that has to be overridden. This will complete our simple Rabit Manager class.

public override void ExecuteUnitOfWork ()

{

Console.WriteLine ("Hello World");

}

* Open or create the file that contains the “Main()” method (Program.cs). Add a “using Rabit” statement at the top.
* Instantiate a PrintManager. Instantiate a Rabit Reactor. Add the manager to the reactor and then run the reactor:

class MainClass

{

public static void Main (string[] args)

{

PrintManager printMgr = new PrintManager ();

RabitReactor reactor = new RabitReactor ();

reactor.AddManager (printMgr);

reactor.Run ();

}

}

Build and run the program. The Hello World manager will start printing out the message “Hello World” once a second until the program is shutdown. Use “CTRL-C” to shut the program down.

You can change the rate that Hello World is printed out by adding a line to the print manager constructor:

WakeUpTimeDelaySeconds = 2.5; //print once every 2.5 seconds

Or

WakeUpTimeDelayMilliseconds = 250; //print once ever 250 millisconds

The ExecuteUnitOfWork() method is contained within an infinite loop with a sleep for the given period of time after the ExecuteUnitOfWork is called. In the next example you will see other ways that the loop can be “woken up”.

That was a very poor and un-exciting use of the Rabit Multithreaded Management system. If all you have is a single manager there is not much point in using the Rabit system.

The next example will use two managers. One manager will be responsible for printing out whatever message is sent to it, while the other manager will be responsible for reading input from the user via the command line and sending the message to the print manager.

Before we jump into the example, a little more overview of the Rabit Multithreaded Management system is required. Information is shared between managers using messages. Messages are sent from one manager to another manager by one of two mechanisms: 1) Publish-Subscribe, or 2) Message Queues. Each mechanism has distinct advantages for its use case.

For Publish-Subscribe messages, one manager is responsible for publishing or posting the messages information. All other managers that have need of the information may subscribe to the message. When a manager subscribes to a message, the manager has access to the information posted by another manager. One example of a use for a publish-subscribe message from the UAV/Drone example is drone position and velocity data obtained from the avionics system. Let’s say one manager is responsible for communicating with the avionics system. This avionics manager will obtain drone position and velocity data from the avionics system and post the data to a DronePositionVelocity Message. Other managers that need the position/velocity data may fetch the data as they need it. It is quite possible the avionics manager may post the position and velocity data more often that it is used by other manager. That is ok… the other managers are only concerned with obtaining the latest position velocity information when they need it.

A message queue is a second method of passing messages from one manager to another. A manager has a message queue and any other manager can send a message to that manager by putting a message in the queue (first-in-first-out pipeline of messages). The manager that has the queue will pull messages from the queue on at a time and process the message. With message queues the receiving manager will get each and every message sent to it. Messages may build up in the queue while the manager is doing other processing. On average the receiving manager must be able to keep up with the messages put into the queue.

**Rabit Read Print Example**

This example builds on the Hello World example. You may start from that example or start a new project. I assume you have a PrintManager and the “Main” program with a Rabit Reactor.

* Our next step will be to create a “DataMessage”. Create a new class file and name the file/class as DataMessage.
* All messages must inherit from the Rabit “Message” class, so add “Message” to the DataMessage:

public DataMessage : Message

* Add a string data item to the DataMessage which will be the data of interest to be sent between the managers:

public string DataValue = null;

* Add a constructor to the message. Each message will have a unique name and it is best to simply use the class name as the message name. This name may be used in a “case” statement to determine the message type. The constructor will be:

public DataMessage()

: base("DataMessage")

{

}

* Each message must override the CopyMessage() method. The CopyMessage method is crucial to the Rabit’s publish-subscribe mechanism. Each data item in the message must be copied across as shown in the example:

public override void CopyMessage(Message msg)

{

base.CopyMessage(msg);

DataMessage dmsg = (DataMessage)msg;

DataValue = dmsg.DataValue;

}

* For simple messages, this is all we need. If the message contains arrays, lists, dictionaries or other complex types, the CopyMessage must take this into account and the user must override the Clone() method. The complete DataMessage class is:

class DataMessage : Message

{

public string DataValue = null;

public DataMessage()

: base("DataMessage")

{

}

public override void CopyMessage(Message msg)

{

base.CopyMessage(msg);

DataMessage dmsg = (DataMessage)msg;

DataValue = dmsg.DataValue;

}

}

* Add a ReadCmdLineManager in the same way the PrintManager was added. The example manager is shown below.

class ReadCmdLineManager : RabitManager

{

private DataMessage dataMessage;

public ReadCmdLineManager()

: base("ReadCmdLineManager")

{

dataMessage = new DataMessage();

AddPublishSubscribeMessage("DataToPrint", dataMessage);

WakeUpTimeDelaySeconds = 0;

}

public override void ExecuteUnitOfWork()

{

Console.Write("Enter Command: ");

string cmdLine = Console.ReadLine();

Console.WriteLine("\n");

dataMessage.DataValue = cmdLine;

string cmdLineLC = cmdLine.Trim().ToLower();

if(cmdLineLC.StartsWith("quit") || cmdLineLC.StartsWith("exit"))

{

this.MgrControl.ShutDownAllManagers = true;

MgrControl.PostMessage();

}

else if (cmdLineLC.Contains("queue"))

{

AddMessageToQueue("PrintManager", dataMessage.Clone());

}

else

{

dataMessage.PostMessage();

}

}

}

The ReadCmdLineManager inherits from the RabitManager as before. Notice that we have added a “dataMessage” to the manager. In the constructor we instantiate the dataMessage. After instantiating the message, we inform the Rabit Management system that this message is going to be a Publish-Subscribe message by calling the “AddPublishSubscribe( string messageName, Message msg) method. A critical item is the “mesageName”. The message name must be unique from all other messages used in the system… but the exact same message name must be used for the same message being shared via the publish-subscribe mechanism. In our case, the DataMessage is going to be shared between the ReadCmdLineManager and the Print Manager. In the Print Manager we will use the exact same message name: “DataToPrint”. The message names are case sensitive.

In the “ExecuteUnitOfWork” method, we will read a line of data from the user, make a couple of evaluations of what is in the data, and then send the data as a message to the PrintManager. In this example we will use both Publish-Subscibe and Message Queues to send the data to the PrintManager. Our first focus will be on the Publish-Subscribe mechanism.

Notice in the ReadCmdLineManager I have set the WakeUpTimeDelaySeconds = 0. The ExecuteUnitOfWork method makes a blocking call to the console.ReadLine() method. The blocking call effectively stops the ReadCmdLineManager until the user enters data and hits the “Enter” or “Return” key. Since the ReadCmdLineManager is being blocked due to this call, there is no reason to have the ReadCmdLineManager thread going into a thread sleep or wait state after the ExecuteUnitOfWork. After processing the new user’s data input, the ReadCmdLineManager will simply recall the ExecuteUnitOfWork which will block waiting for new input from the user.

Now back to the PrintManager. The PrintManager needs a copy of the DataMessage and needs to be changed so that it will print what is in the DataMessage. Open the PrintManager and make the changes per the example below:

class PrintManager : RabitManager

{

private DataMessage dataMsg;

public PrintManager()

: base("PrintManager")

{

dataMsg = new DataMessage();

AddPublishSubscribeMessage("DataToPrint", dataMsg);

WakeUpTimeDelaySeconds = 10.0;

}

public override void ExecuteUnitOfWork()

{

bool msgPrint = false;

if (dataMsg.FetchMessage() && dataMsg.DataValue != null )

{

Console.WriteLine("Publish Subsribe Message = {0}\n",

dataMsg.DataValue);

msgPrint = true;

}

while (MgrMessageQueue.NoMessagesInQueue() > 0)

{

object msg = MgrMessageQueue.getMessage();

if (msg is DataMessage)

{

DataMessage dmsg = (DataMessage)msg;

Console.WriteLine("Queue Message = {0}\n", dmsg.DataValue);

msgPrint = true;

}

}

if (!msgPrint)

{

Console.WriteLine("PrintManager is waiting for a Message");

}

}

}

Notice that we have added a DataMessage to the PrintManager. The dataMsg does not have to be the same as in the ReadCmdLineManager. In the constructor the dataMsg is instantiated and then added to the Rabit Publish-Subscribe message system. Notice that the “DataToPrint” message name is exactly the same as that used in the ReadCmdLineMessage. If different names are used, the messages will be added to the publish-subscribe message system, but they will not be connected between the two managers.

Notice that I have set the WakeUpTimeDelaySeconds = 10. This will help highlight points to be made as we run the example.

Open the Main program and add the new ReadCmdLineManager to the Rabit Reactor:

static void Main(string[] args)

{

ReadCmdLineManager ReadCmdLineMgr = new ReadCmdLineManager();

PrintManager PrintMgr = new PrintManager();

RabitReactor reactor = new RabitReactor();

reactor.AddManager(ReadCmdLineMgr);

reactor.AddManager(PrintMgr);

reactor.Run();

}

Now build and run the example program.

Enter any data or text on the command line and hit the enter key. The data will be printed out by the PrintManager. Because we set the WakeupTimeDelaySeconds to 10 seconds, it may take up to 10 seconds for the data to print.

Try typing several lines of data quickly (under 10 seconds) on the command line. Notice that only the last data message will be printed out. The ReadCmdLineManager Posts each data line entered with the dataMessage.PostMessage(); line in the ExecuteUnitOfWork. The PrintManager uses a dataMsg.Fetch() method in its ExecuteUnitOfWork method. Since the PrintManager has a 10 second time delay between each call to the ExecuteUnitOfWork method, only the last update to the data message will be Fetched and printed. The Fetch() method returns a flag which will be true if the time stamp on the posted message has changed. Each Message has a “Timestamp” attribute which is automatically updated when a message is posted.

If you enter a command line that starts with “quit” or “exit” the ReadCmdLineManager’s ExecuteUnitOfWork will post a shutdown command to all managers, and the Rabit Manager system will shut down. The “MgrControl” message is part of the Rabit system. Any manager that sets the ShutdownAllManagers flag to true and posts the message will cause the Rabit system to shutdown.

Notice that when we added the DataMessage to the PrintManager and the ReadCmdLineManager, we used exactly the same process. The only difference is which manager call the Post method and which managers call the Fetch method. Only none manager should Post to the message, but as many managers that need the information may Fetch the message.

Waiting for the PrintManager to wake up after posting a message is not necessarily desirable. We can use an event associated with the DataMessage to wake up the Print Manager whenever the ReadCmdLineManager posts to the messages. In the PrintManager constructor, right after adding the dataMsg to the Rabit Publish Subscribe system, add the statement:

dataMsg.RegisterEventWithGlobalMessage(wakeUpManagerEH);

This associates a manager wakeup event with the dataMsg. The PrintManager will wakeup anytime the ReadCmdLineManager posts new data to the data message. After adding the above line (or un-commenting the line) build and run the program. Now when you enter new data on the command line, the ReadCmdLineManager Posts the new data, the PrintManager should wake up and print the data.

If the word “queue” is added to any command line data entry, the data will be sent by a message queue from the ReadCmdLineManager to the PrintManager. The Rabit Manager system automatically sets up a message queue for each manager. The message queue name will be the same as the Manager’s name. To send a message via a message queue to another manager simply used the method: AddMessageToQueue(managerName, message) as seen in the ExecuteUnitOfWork method of the ReadCmdLineManager. Notice that the message going into the queue is cloned. The prevents the receiving manager from simply receiving a reference to the message which can cause some weird bugs.

The Rabit system automatically registers a wake-up event with the Manager’s message queue, so that the manager will wake up whenever a message is placed in its queue. The registration occurs in the Manager’s default Startup() method.

Each Manager has a Startup() and Shutdown() method that may be overridden. The Startup() method is where initialization code for the manager should be placed. Any shutdown code may be placed in the Shutdown() method.

An interesting test to run would be to override the PrintManager’s Startup() method with and empty method:

override public void Startup() {}

This will keep eliminate the wake-up event associated with the PrintManager’s message queue. Now build and run the program. If you enter a number of data items on the command line in rapid succession, you should see that the messages build up in the PrintManager’s message queue. When the PrintManager wakes up after 10 seconds, it will print all the messages one after the other. Notice the difference between this operation and that of the Publish-Subscribe message operation.