**Rabit Multithreaded Manager Framework**

Harry Direen, PhD

Randal Direen, PhD

DireenTech Inc. ([www.direentech.com](http://www.direentech.com))

Under research contract for:

Academy Center for UAS Research

Department of Electrical and Computer Engineering

HQ USAF/DFEC

2354 Fairchild Drive

USAF Academy, CO 80840-6236

Dec, 2016

# Introduction

Software for autonomous systems such as UAVs (Unmanned Ariel Vehicle / Drones) and robots is hugely complex. The complexity derives from all the tasks that must be accomplished and work in unison in order to have a feature rich, smoothly operating, robust, autonomous system. I have spent over 5 years developing software for autonomous UAVs at the US Air Force Academy in the Academy Center for UAS Research. The design methodologies and software described here has come out of that research. The software has been used on multiple programs in the Academy Center for UAS Research and has been proven to be a very robust design approach. The design concepts are not new or unique; the point is they are design approaches that work and work well for autonomous systems and many other software systems.

I will set the stage by describing overall key tasks and functionalities of the UAVs we have flown at the Air Force Academy. The concrete examples should help in grasping, and better understanding the software design approach and why the particular design approach was taken.

Projects at the Academy Center for UAS Research typically involve multiple, coordinated UAVs running a mission autonomously, meaning that once the UAVs are in the air, all flight control and mission control of the UAV are handled by the on-board computer and not by a human operator. The UAVs communicate between each other and communicate with a ground station. The ground station operator monitors the mission progress of the UAVs and may update mission parameters through the ground station software. Each UAV has its own onboard computer and operates autonomously, making decisions on where and how to fly the aircraft based on its own sensor data and data from other participating UAVs to meet the goals of the mission. Each aircraft has an autopilot. The autopilot handles the actual flight control of the aircraft and provides aircraft position, velocity and attitude information (vehicle latitude, longitude, roll, pitch, yaw, and related information) to the autonomous operating system. Each aircraft typically has a camera system used to take pictures of and autonomously identify targets of interest. A mission usually involves flying to an area to search; searching an area for targets of interest; calling in other UAVs to help validate a target; and tracking moving targets; all handled autonomously by the aircraft.

A breakdown of the primary subsystems of the UAV is:

* Communications: wirelessly communicating with a ground station and communicating between other UAVs and possibly other autonomous systems. Communications use well defined messages.
* Autopilot Interface: sending command messages to the autopilot and receiving information from the autopilot. Information coming from the autopilot typically includes:
  + Latitude and Longitude aircraft position
  + Altitude of the aircraft
  + Velocity vectors
  + Attitude (roll, pitch, yaw) data along with rates of change of these items
  + Other status information
* Image Processing: a camera for capturing images along with the image processing required to identify targets in those images along with the geolocation of the targets.
* Sensor Fusion: Target information can come from the UAV’s image processing and may come from other UAVs and sensor systems. Sensor fusion is responsible for fusing target information from multiple sources and providing a best estimate of a given targets location. Sensor fusion has the capability to keep track of multiple targets simultaneously.
* Mission Control: mission control is responsible for taking current aircraft location, mission state, target information, and any other parameters deemed necessary and determine where to direct the aircraft to next. Mission control is responsible for all the high-level operational control of the aircraft to carry out the given mission.
* Aircraft health monitoring: This module monitors fuel/battery levels, communications status and other parameter associated with the health of the aircraft. If fuel levels become low or communications with the ground station is lost, then the aircraft can be sent home.

As can be seen from the outline above, control of an autonomous vehicle is complex and results in a highly complex software system. As is well known in the industry, complex system must be broken down into manageable subsystems in order to handle the complexity in a robust, reliable, and feature expandable way. The ensuing describes the details of the approach taken in software design of the UAV control software. C++ and .NET/Mono versions of the underlying base library can be obtained from GitHub at:

* https://github.com/rdireen/rabitcpp
* https://github.com/rdireen/rabitcsharp

These libraries provide the underlying software system used in the control of UAVs. They do not provide any of the UAV’s operational software. Rabit (sic) is a multi-threaded management system library, or framework, which may be used as the basis for autonomous UAVs/Drones, robots, and other systems.

Complex software systems such as robotics and drone control may be, and should be, broken into simpler, easier to design and code modules. There is a real advantage to breaking 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 on 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. Single threaded software can only take advantage of one of the processor cores, leaving the other cores un-used by the software system. 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.

# Rabit

The Rabit (sic) Multi-threaded Management System is a system composed of managers that run on separate threads; and a messaging system to safely communicate between the managers. The thread safety features are hidden in the back-ground so that a user does not have to be concerned with the specific operation.

Rabit 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, in the UAV system a separate manager was established for each of the primary subsystems noted above (Communications, Autopilot Interface, Image Sensor, …). For those familiar with ROS (Robot Operating System) a Rabit Manager corresponds loosely with a ROS Task. One of the key differences between ROS and Rabit is that Rabit runs as a single mutli-threaded process where each manager is running on a separate thread. In ROS each Task is a separate operating system process. It would be quite possible, and maybe desirable, to use Rabit in the design of a complex ROS Task. The goal of this paper is to describe Rabit and how to use Rabit and not to compare Rabit with ROS, so that is all I will say in comparing the two systems.

## Rabit Manager

Figure 1 shows a block diagram of a Rabit Manager. A Rabit Manager is composed of an Execute Unit-of-Work, which is supplied by the user along with optional Startup and Shutdown processes; any number of Publish-Subscribe Messages; and one or more Message Queues.



Figure 1 Rabit Manager

The publish-subscribe messages and message queues support the operation of the Unit-of-Work and communications between managers. The user establishes the publish-subscribe messages required to support the manager’s subsystem requirements and adds the messages to the manager. One message queue comes automatically with each manager for receiving messages from other managers. The user may add more message queues to meet the subsystem’s requirement.

The user is responsible for writing the Execute Unit-of-Work process/method which is part of the Manager’s continuous loop. The user may also supply startup and shutdown processes/methods. The grey boxes (“Shutdown Manager”, “Sleep”, and “Wake-Up Events”) are part of the underlying Rabit Manager.

Any number of events can be used to wake-up the manager from its sleep state. There is a settable timer that guarantees the manager will be woken up each time the timer times out. The default is once a second. The user has the option of tying a wake-up event to one or more publish-subscribe messages that will cause a wake-up whenever another manager up-dates the message. Each message queue can be setup to create a wake-up event when a message is pushed onto the queue. The key to remember is that there are a number of ways to wake the manager up so that it goes around its loop and executes another pass of the unit-of-work.

A Rabit Manager runs in a continuous loop with a “sleep” at the end of the loop. A manager is started at the beginning of a program and designed to run for the entire time the program (control system) is running. A Manager is designed to run a “Unit-of-Work” in each pass of its internal loop. So what does it mean to run a unit-of-work?

Let’s take the Mission Control subsystem of the UAV’s operation as an example. Figure 2 is a rough flowchart of some of Mission Control’s responsibilities. The details are not important or complete. The flowchart simply gives an example of things that can be accomplished in a unit-of-work. The flowchart shows a couple of sub-tasks (Target Search and Target Tracking) that are selected based upon a Mission Control Operational Mode or State Variable.



Figure 2 Mission Control Unit-of-Work

Suppose the UAV is sent to a search an area with the responsibility of finding a target in that area and if a target is found… track the target. We will assume we start of in the “Find Target” mode

.

A unit of work might compose the steps of:

1. Get the UAV’s current position, velocity vector and attitude information. This information is stored in publish-subscribe messages so those messages are fetched at the start of the process.
2. Check the Mission Control Mode of Operation and take the path based on that mode (assume the Target Search mode).
3. Check to see if the UAV is within the bounds of the search area.
   1. If the UAV is outside the bounds of the search area, determine the location/center of the search area.
   2. Send a message to the Autopilot Interface Manager with the direction or location coordinates of the search area. This message will be sent to the Autopilot Interface Manager’s message queue.
   3. End unit-of-work… which puts the manager back into the sleep mode.
4. Check to see if a new target was found. This might come from a message in the manager’s receive queue which would have been inserted there by Sensor Fusion.
   1. If a target is found… switch to the Target Tracking mode.
   2. End unit-of-work… which puts the manager back into the sleep mode.
5. Using some search algorithm, determine the next location or direction to send the UAV. Part of the algorithm will be to keep the UAV within the bounds of the search area.
6. Send a message to the Autopilot Interface Manager with the direction or location coordinates of the search area. This message will be sent to the Autopilot Interface Manager’s message queue.
7. End unit-of-work… which puts the manager back into the sleep mode.

The key concept here is that a Manager is sleeping (idle). An event wakes the manager up. The manager handles the unit-of-work (task) at hand, and then goes back to sleep. The manager will be forever (as long as the program is running) being woken up, handling its unit-of-work, going back to sleep, only to be woken up again at the next wake-up event. If a manager is designed well, and does not have too arduous of a task, the manager will spend most of its time sleeping which allows managers with a difficult, time-consuming task such as image processing, to have more of the processor’s compute cycles.

A Manager’s task should be designed to be as independent as possible from other manager’s tasks. For instance, the Mission Control manager should not be concerned with how messages are sent to or received from the ground station or other UAVs. The Mission Control manager should not be concerned with how Image Processing processes images to find targets or how Sensor Fusion combines and filters target information. Mission control should only obtain UAV location and current target information and make decisions based on the latest location and target data.

As another example, let’s look at the Communications Manager. The communications manager is responsible for sending and receiving messages from the ground station, other UAVs, and possibly other autonomous systems. In our system, the communications manager acts a lot like a post office. The communications manager is not concerned with what the contents of message is, it is only concerned with formatting a message to the correct format for transmission and sending it to the right location (ground station, UAV x, …) on the transmit side. On the receive side the communications manager receives messages from the various locations/entities, formats the message into the correct internal format, and posts the message to the correct manager or managers for their consumption. I will give a few more details of the Communications Manager after discussing some of the messaging features of Rabit.

The Image Processing Manager has the well-defined task of capturing images from a camera sensor, processing the image to identify specific targets, and then pass the target information on to Sensor Fusion. In each execute-unit-of-work loop of the Image Processing Manager, the manager:

1. Captures an image from the camera sensor
2. Obtains the latest UAV position and camera pointing angle
3. Processes the image to find targets located in the image
4. If a target or targets are found:
   1. Calculate the target’s physical ground location
   2. Send the target type and location information to Sensor Fusion
5. Go to sleep until woken by event which returns to step 1. (Actual sleep time is typically very short or zero).

The Image processing mode will typically be dependent on the mission behavior. Image processing like the other managers focus on its task of processing images without being concerned at all with what is done with the resulting target information. This keeps the design of the module straight forward, which is a primary feature of the Rabit Management system. Each manager handles its task without being concerned with other manager’s tasks.

### Startup Process

The Startup Process is an optional user provided process/method. This method is called by the Rabit system before the manager enters its main loop (reference Figure 1). The Startup process is used to initialize any resources and processes the manager’s Unit-of-Work requires. For instance, in the UAV system, the Autopilot Interface manager must communicate with the autopilot over a serial interface. The startup process for this manager will be responsible for initializing that communications interface.

### Shutdown Process

The Shutdown Process is an optional user provided process/method. This method is called by the Rabit system before the manager is shut down and exited. This is where resources used by the manger can be cleaned up or shut down before the manager is shut down.

### Wake-Up Events

As noted above, a manager runs in a continuous loop calling the Unit-of-Work each time through the loop (reference Figure 1). After executing the Unit-of-Work, the manager goes into a sleep state. Going into a sleep state frees up the CPU for use by other managers and other processes running on the computer. The Wakeup Events Block is responsible for waking up the manager from the sleep state so that the Unit-of-Work can be executed.

The general philosophy in Rabit is that once the manager is woken up, do to the occurrence of an event, it is best to carry out the all the functionality within the Unit-of-Work rather than some small subset of the work related to a specific event. The reason for this is that there is significant overhead in an operating system context switch to a manager’s operating thread. Therefore it is typically more efficient to run all the code in the manager’s Unit-of-Work at one time and then release the thread context by going back to sleep, than is to have to wake up more often and only perform a small subset of the Unit-of-Work. The user of Rabit has control of what operations an event triggers; but what I have found in general, that works best and keeps the manager’s code straight forward, is simply to use events to wake up the manager from the sleep state and execute the complete Unit-of-Work.

Rabit supports a number of events that can be used to wake up the manager from the sleep state. These are:

1. Timer Event: a timer is started when the manager enters the sleep state and a wake-up event will be triggered when the timer is complete. The default for this timer is 1 second. The user can set this timer to most any value that makes since. A lower bound is around 10 milliseconds. Operating systems and frameworks such as .NET have lower bounds on context switches, so it does not make sense to go below this boundary. If the use is relying primarily on other events to wake up the manager, the timeout can be set to a large number. The timeout cannot be disabled, but the timeout can be set to a large number to effectively disable it. Even if other events are being used as the primary mechanism to wake up the manager, I prefer to keep a timeout at a reasonable value so that there is a guarantee that the manager will be woken up periodically to run the Unit-of-Work and verify that anything that needs to be done is done. The timeout can be set by the Unit-of-Work to actively change the timeout period based upon the systems state or mode of operation. The user is responsible for taking advantage of the timeout event to meet the overall system’s requirements.
2. Publish-Subscribe Message Event: A wakeup event can be generated when a manager publishes new data to a publish-subscribe message. A manager that wishes to be woken up whenever a specific message is updated will subscribe to that message’s wakeup event. For example, let’s say that a publish-subscribe message is used to set the course/direction of the UAV. Both the Mission Control and the Autopilot Interface managers will have copies of this message. The Mission Control manager will be responsible for setting and updating the content of the message. The Autopilot Interface manager will be responsible for reading the message and sending the course/direction information to the autopilot. The Autopilot Interface manager will subscribe to a wakeup event that wakes the manager up whenever the message is updated. Whenever Mission Control publishes new course/direction information to the message, the Autopilot Manager will automatically wakeup, and as part of its Unit-of-Work, read the course/direction message and send the information to the autopilot.

It is a user’s responsibility to determine which, if any, publish-subscribe messages should be used to wake up a manager and to subscribe to the wakeup event. The code for subscribing to a wakeup event is typically added to the manager’s constructor code.

1. Message Queue Events: There are two types of events associated with message queues: push event and a pop event. The push event occurs when a message is pushed onto the queue and a pop event occurs when a message is popped off the queue. The most common use will be to wake up a manager when a message is pushed onto the manager’s queue. For instance, in the UAV system, the Communications Manager is responsible for taking message sent to the manager from any other manager and sending the message out to other UAVs or to the ground station. The Communications Manager subscribes to its receive message queue’s push event. Whenever any other manager pushes a message to be sent into the message queue, the Communications manager will wake up, and as part of its Unit-of-Work pull whatever messages are on the receive queue, format the message and send the message to its destination. Once the Communications Manager is woken up, it should process all the messages in the receive queue.

The queue pop event could be used by a manager to wake up after another manager pops a message from its message queue. This is not as common of an event to use, but I have found it useful to help control some overall system timing.

It is a user’s responsibility to determine which, if any, queue events should be used to wake up a manager and to subscribe to the event. The code for subscribing to a wakeup event is typically added to the manager’s constructor code.

Rabit wakeup events do not stack up. Whichever wakeup event occurs first will cause the manager to wake up from its sleep state and start executing the Unit-of-Work. All other wakeup events that occur essentially at the same time will be lost. If wakeup events occur during the Unit-of-Work execution time, Rabit will note this and instead of going to sleep at the end of the Unit-of-Work will clear the events and start the next execute Unit-of-Work. If no other events occur during this time, the manager will enter the sleep state at the end of the Unit-of-Work.

## Information Sharing between Managers

In any practical system, managers must be able to communicate between each other. Rabit provides two distinct, thread-safe, methods of communicating between managers. The two communications methods are:

* Publish Subscribe messages
* Message Queues

Each communication method has its advantages and disadvantages. Together, the two methods complement each other and provide a rich form of thread-safe, reliable, communications between the managers.

## Publish-Subscribe Messages

Publish-Subscribe messages allow a manager to publish information in a message that all other managers will have access to as needed. For instance, in the UAV example, the Autopilot Interface Manager receives UAV location and attitude data from the autopilot on a regular basis. A UAV Position-Attitude Publish-Subscribe message is set up that contains data such as:

* UAV Latitude
* UAV Longitude
* UAV Altitude
* UAV Velocity Vector
* UAV Pitch Angle
* UAV Roll Angle
* UAV Yaw Angle
* GPS Time stamp when data was captured.

The Autopilot Interface Manager collects this data from the autopilot several times a second and simply posts the UAV Location message. This makes the position and attitude information available to all the other managers that subscribe to the UAV Position-Attitude message. Image processing must tag every image captured with the information. So when image processing captures a new image, the image processing manager will fetch the latest copy of the UAV Position-Attitude message and include this data with the image. The Mission Control Manager requires the current UAV position and velocity information to compute next flight control directions and to perform other mission processes. The Mission Control Manager simply fetches the UAV Position-Attitude message as needed which supplies the latest position and velocity information. The Communications Manager fetches the UAV position and velocity message on a regular basis in order to send this information to the ground station and other UAVs for situational awareness purposes.

A publish-subscribe message is ideal for the example given above and any other similar situation. One manager is responsible for capturing or generating specific information that other managers need and simply posts the information message as it is captured or generated. The manager posting the information is not concerned with how the information is used by other managers; the manager simply keeps the information up-to-date. The receiving managers that subscribe to a given message are not concerned with who makes the information available or how the information is generated; the receiving managers simply fetch the information in the message as that manager needs the information.

A publish-subscribe message is typically used where the receiving manager is not concerned with obtaining the given information every time it is updated, but simply requires the latest copy of the data. In the example above, image processing might take some time to process an image. In the meantime, the Autopilot Interface manager might publish updates to the UAV Position-Attitude message several times during the processing of a single image. The Image processing manager only cares about getting the UAV’s current position and attitude information when it captures a new image and is not at all concerned with the fact that the information might be updated several times during the processing of each image.

In the Rabit system, each manager keeps a local copy of a given publish-subscribe message. A global copy of the message is hidden behind the scene. The manager that is responsible for keeping the information up-to-date simply fills in his local copy of the message with data captured or generated by the manager. When the data is filled into the local copy of the message, the manager “Posts” the message. When the message is posted, Rabit thread-locks the global copy of the message and copies the data into the global message and then releases the lock. Rabit also time-stamps the global message whenever a post occurs. A manager that wishes to receive information from a publish-subscribe message will create a local copy of the message and subscribe to the message. Any time the manager wants or needs a copy of the latest information stored in the global copy, the manager uses the “Fetch” method of the message. When a Fetch is issued, Rabit goes to the global copy of the message and checks the timestamp of the global message and the manager’s copy of the message. If the timestamps are exactly equal, it is assumed the manager has the latest message data. This prevents having to obtain a thread-lock. If the timestamps are not equal, Rabit will thread-lock the global message, and copy the data, including the timestamp, from the global message into the manager’s local copy. After copying the data, Rabit releases the lock on the global message.

One advantage of Rabit’s publish-subscribe message system is that the manager is responsible for fetching the latest copy of the message data when the manager deems it appropriate. This prevents the manager from starting to use a messages’ data only to have the data change spontaneously out from under the manager during its use which could cause unpredictable results.

Rabit’s Publish-Subscribe message system supports event triggers. A manager may subscribe to a message’s event trigger that will trigger whenever another manager posts new data to the message. The most common use of this trigger will be to wake a manager up from its sleep state when new information it posted to the message. Rabit also supports user-defined events triggered by a message post. This is less common, but can be used to meet a system’s needs.

### Message Queues

Rabit supports thread-safe message queues as another independent method of safely sending messages from one Manager to another Manager. In Rabit, one manager is the receiver of messages put into a particular message queue while one or more other managers can send messages to the receiving manager by simply pushing messages into that message queue. For example, in the UAV system, the Communications Manager has a message queue that will contain messages the other managers wish to send outside the UAV such as to the ground station or to one or more other UAVs. The Mission Control manager might create and send a message to another UAV to request help validating a target, or the manager might send a message to the ground station to verify a target. The Mission Control manager simply creates the message and pushes it into the Communications Manager’s message queue. The Communications Manager will pop the message from the message queue, format the message for transmit, determine the intended destination and send the message on its way. The Communications Manager does not care what the message is or where it was generated, the manager simply determines the destination based on header info in the message, formats the message for transport, and sends the message on its way.

In the opposite direction, the Mission Control Manager and Sensor Fusion Manager may have their own receive queues. If the Communications Manager receives a response message from another UAV or the ground station for target verification, the Communications manager will reformate the message into an internal message format, determine the message is for Mission Control, and simply push the message into Mission Control’s message queue. If the Communications Manager receives a target information message destined for Sensor Fusion, then the Communications manager will reformate the message into an internal message format and push the message into Sensor Fusion’s message queue.

Message queues are important for cases where each and every message must be processed by the receiving manager. The manager is not interested in just the latest message data, but each and every message dataset that comes in.

Rabit Message Queues support event triggers. An event can be triggered whenever a message is pushed into a message queue, and an event can be triggered whenever a message is popped from a message queue. A typical use would be to wake a manager up whenever a new message is pushed into its receive message queue. In the UAV case, the Communications Manager can subscribe to a wakeup event whenever another manager pushes a message to be transmitted into his message queue. The Communications Manager will then immediately wakeup when there is a new message to be sent.

## Rabit Summary

The Rabit Multi-threaded Management System is basically just managers that run on separate threads, and provides a means to safely communicate between the managers. The thread safety features are all hidden in the back-ground so that this user does not have to be concerned with the specific operation. While this is a fairly simple and straight forward system, it is very powerful. The Rabit system allows the user to build highly complex, robust systems in a simple, straight forward manner.

The next sections provide tutorials on how to use Rabit. These tutorials show how to setup and use most of the key features of Rabit.

# Using Rabit

An outline of how to setup and use Rabit is given below. In addition there is a Tutorials section that provides a couple of concrete examples of using Rabit. Source code for the tutorial examples is included with Rabit.

The basic steps involved in creating a Rabit application are fairly simple:

1. Create Rabit Managers, one for each significant task that makes up the application. Each manager must inherit from RabitManager. Each manager must at a minimum:
   1. Override the ExecuteUnitOfWork () method with the task code for that manager.
   2. Set the wake-up time for the manager (WakeUpTimeDelaySeconds = 1.0). The default wake-up time/rate is once per second.
   3. Optionally override the manager’s Startup() method. All initialization code for the manager should be included in the startup method. This method will be called before the first ExecuteUnitOfWork method is called.
   4. Optionally override the manager’s Shutdown() method. This method will be called when the manager is shut down and may be used to clean up any resources the manager uses.
2. Create messages for communicating between the managers. Each message must inherit from RabitMessage. The user must at a minimum override the CopyMessage() method of each message.
3. For Publish-Subscribe messages: each manager that is going to use a specific publish-subscribe message should create a local copy of the message, and then register the message with Rabit: AddPublishSubscribeMessage(“uniqueMessageName”, userMsg). The “uniqueMessageName” is unique to the message type. The exact same message name must be used by each manager that is registering the same publish-subscribe message.
4. Message Queues: by default, each Rabit Manager will have a message queue associated with the manager. Other managers can send messages to the specific manager using the method: AddMessageToQueue ("PrintManager", dataMessage.Clone ()). If other message queues are required for the application, they can be added.
5. Subscribe to wake-up events. This is optional. A manager may subscribe to wake-up events associated with any publish-subscribe message or message queue.
6. Application Main() program: in the application’s main program:
   1. Instantiate each manager
   2. Instantiate a RabitReactor. The Rabit Reactor is responsible for keeping track of and running the managers.
   3. Add each manager to the RabitReactor: reactor.AddManager(manager\_x).
   4. Run/Start the RabitReactor: reactor.Run() . The Run method is a blocking call. The main program will block on this call until all the managers have shutdown. It is a good idea to have some mechanism designed in for shutting the system down.

# Tutorials

The tutorials below are shown using C# in .NET/Mono. There is also a C++ version. The example code is provided with the Rabit source code at the following GitHub locations:

* https://github.com/rdireen/rabitcpp
* https://github.com/rdireen/rabitcsharp

**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:

public PrintManager ()

: base()

{

}

* 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 every 250 milliseconds

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 Framework 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 Framework 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. The constructor will be:

public DataMessage()

: base()

{

}

* Each message must override the CopyMessage() method. The CopyMessage method is crucial to the Rabit Framework’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 : RabitMessage

{

public string DataValue = null;

public DataMessage()

: base()

{

}

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()

{

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 Framework 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 “messageName”. 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-Subscribe 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()

{

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 Subscribe 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 Framework 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 calls the Post method and which managers call the Fetch method. Only one 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 Framework 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. This prevents the receiving manager from simply receiving a reference to the message which can cause some weird bugs.

The Rabit Framework 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 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 of the messages one after the other. Notice the difference between this operation and that of the Publish-Subscribe message operation.

## Definitions

Put Definitions here… I think a definitions section would be useful for re-emphasizing your concepts such as manager, task, etc.