Managed Complexity in Embedded Control Systems for UAVs, Drones and Robotics

Harry Direen, Ph.D., P.E.
DireenTech Inc.
www.direentech.com
October 5, 2017

Introduction

Embedded software for autonomous control systems such as UAVs (Unmanned Ariel Vehicle / Drones) and robots can be 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. When first jumping into a project the tendency is to think… how hard can it be to write a little software to control a drone? After all what needs to be done:

- Sends some waypoints to the drone's autopilot
- Send/Receive a few messages to a ground station
- Capture some images and do a little image processing
- …

There is a strong tendency in software development to under estimate the system requirements. The tendency is to focus on the lower level aspects such as how to send waypoints to the autopilot, or send messages to/from the ground station, or focus on the fun image processing. Once the lower level code is figured out… some type of loop structure to pull all the pieces together is thrown in and wallah we are done… right? This process kind-of, sort-of works so isn't that good enough? The problem is as the system complexity grows, and it always seems to, the code becomes more and more spaghettish. New features become harder and harder to add without breaking the current code. The house of cards becomes very tenuous.

There are much better ways to handle the software system level design and I will be outlining one approach that works very well for the systems I have designed. The approach is not new, but I believe tends to be over looked. I developed the system as part of my work innovating control software for UAVs at the US Air Force Academy. I have used the approach in a variety of other projects and products including a wheelchair control system for the NeuroGroove project (www.direentech.com). To make life easier DireenTech developed software libraries which may be obtained at: https://github.com/rdireen/rabitcsharp and https://github.com/rdireen/rabitcpp.

UAV Control

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 highlight many of the issues that arise in embedded systems control.

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.

Modular Multi-threaded Control

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 on separate software threads allows the system to take advantage of and distribute the processing over multiple processor cores.

For instance, the UAV Communications is separated into a module that formats and sends messages out to other UAVs and the ground station while also receiving messages from other UAVs and the ground station and passed the received message to the proper module. The communications module acts like a post office. The communications module is not concerned with what is in a message, i.e. it is not responsible for generating messages or with responding to a received message. The communications module simply takes messages generated by other modules, formats them to the correct transport format, and sends the message to the intended location(s) (other UAV(s) and/or ground station). Messages received by the communications module are reformatted from the transport format to the internal system format, and send to the proper module(s). This isolates the other modules from having to handle communications of messages to outside systems.

Many embedded system processors are multi-core processors. Even the small, inexpensive, Raspberry Pi (www.raspberrypi.org) computers have a quad-ARM-core processor. 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. For embedded systems that only have single core processors, there is still an advantage of using multi-threaded programing techniques. A multi-threaded approach provides a clean separation of modules.

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 (sic) Multithreaded Management system software libraries developed by DireenTech are designed to hide most of the difficulties of building a multithreaded system and to provide a relatively easy to use base for embedded control systems. C++ and .NET/Mono versions of the libraries can be obtained from GitHub at:

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

Rabit (sic) is a multi-threaded management system library, or framework, which may be used as the basis for autonomous UAVs/Drones, robots, and many other embedded control systems.

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 threading and locking operations.

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. The grey-ish boxes ("Shutdown Manager", "Sleep", and "Wake-Up Events") are part of the underlying Rabit Manager.

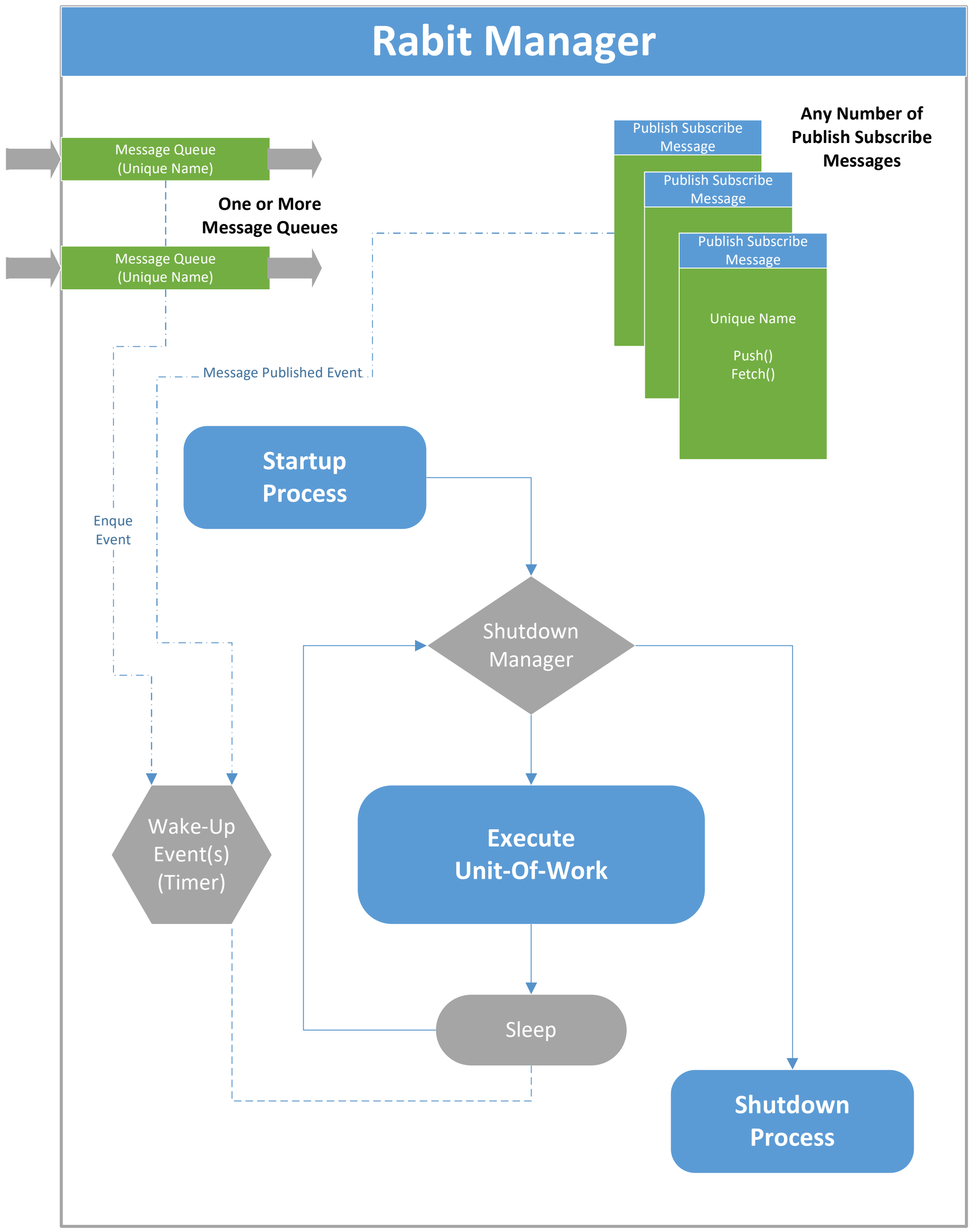
Rabit Manager
Message Queue (Unique Name)
Message Queue (Unique Name)
One or More Message Queues
Publish Subscribe Message
Publish Subscribe Message
Publish Subscribe Message
Unique Name
Push()
Fetch()
Any Number of Publish Subscribe Messages
Message Published Event
Enque Event
Startup Process
Shutdown Manager
Execute Unit-Of-Work
Wake-Up Event(s) (Timer)
Sleep
Shutdown Process

Figure 1 Rabit Manager

The Rabit system provides two mechanisms which safely communicate between managers. The two mechanisms are publish- subscribe messages and message queues. Each has distinct advantages. The messaging system is fully described in the section: **Information Sharing between Managers.**

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.

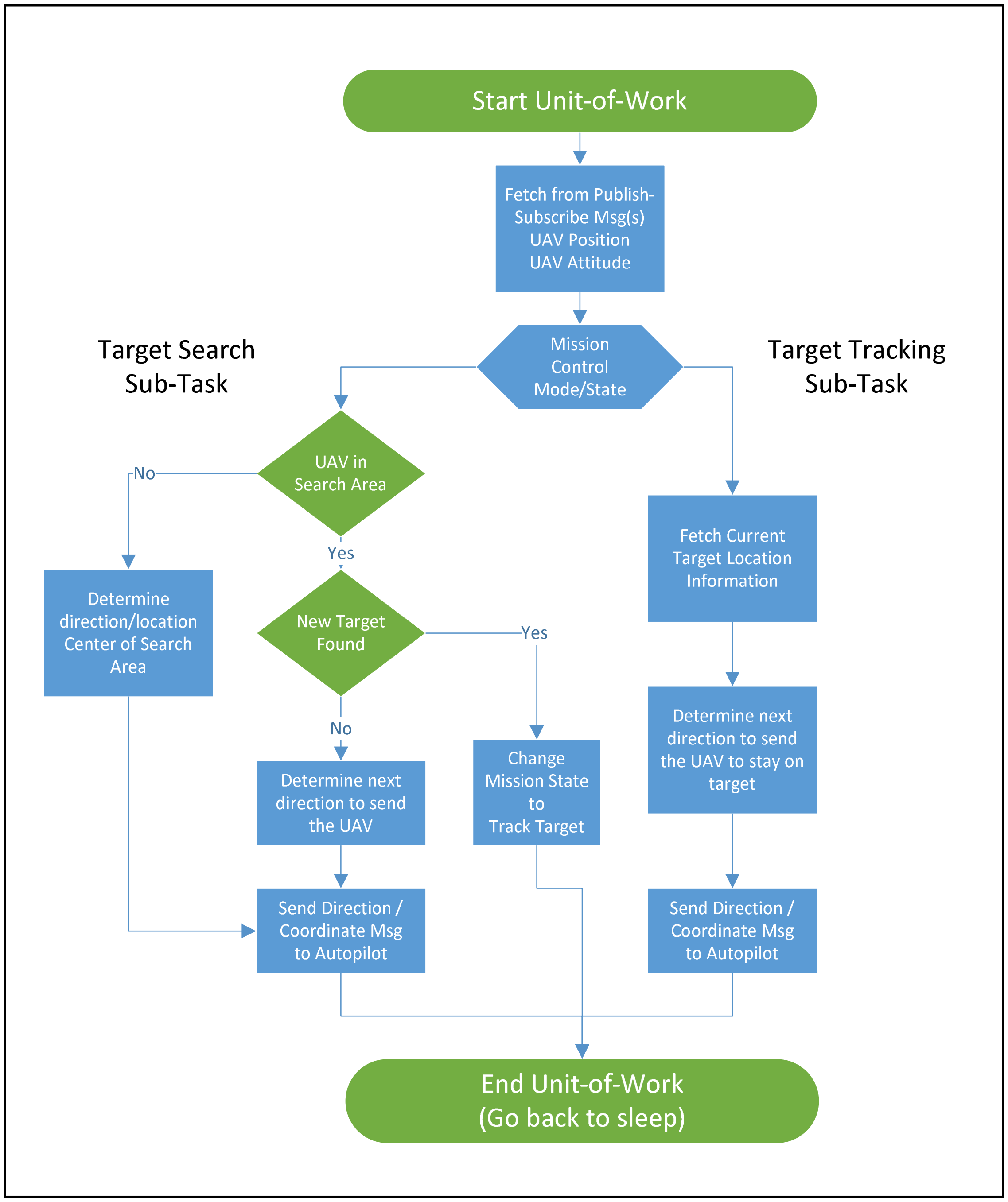
Start Unit-of-Work
Fetch from Publish-Subscribe Msg(s) UAV Position UAV Attitude
Mission Control Mode/State
Target Search Sub-Task
Target Tracking Sub-Task
UAV in Search Area
No
Yes
Determine direction/location Center of Search Area
New Target Found
Yes
No
Determine next direction to send the UAV
Change Mission State to Track Target
Fetch Current Target Location Information
Determine next direction to send the UAV to stay on target
Send Direction / Coordinate Msg to Autopilot
Send Direction / Coordinate Msg to Autopilot
End Unit-of-Work (Go back to sleep)

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.
    a. If the UAV is outside the bounds of the search area, determine the location/center of the search area.
    b. 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.
    c. 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.
    a. If a target is found… switch to the Target Tracking mode.
    b. 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 other managers (running on independent threads) 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 other UAVs or ground station; 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:
    a. Calculate the target's physical ground location
    b. 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 it is 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. 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 another 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. When 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.

3. 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 it's 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 established 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. The manager 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 a manager is responsible for fetching the latest copy of the message data when that 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.

Summary

Embedded software for autonomous control systems is typically very 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, control system. It is very import to manage the complexity of these systems by building upon a robust underlying system framework. The DireenTech Rabit library is a proven, multi-threaded, management system framework with witch robust, feature rich, expandable, embedded control system can be built upon. The Rabit library is being used to control:

- UAV's at the US Air Force Academy's Center for Unmanned Aircraft Systems:
    - https://www.usafa.edu/research/research-centers/center-unmanned-aircraft-systems
    - https://direentech.com/portfolio/drone-research-project/
- The NeuroGroove wheelchair and upcoming racecar control systems:
    - https://direentech.com/portfolio/neurogroove-project/
    - https://www.usafa.edu/the-neurogroove-project/
    - http://falcibiosystems.org/
    - http://www.falconworks.org/projects