Autonomous Runway Detection

# RTOS Configuration Parameters:

# Tasks created:

## Client-Side Tasks:

1. Run Application

Name of the task: "Run"  
Task handle variable: ‘Run\_application\_handle’  
Priority: 3  
Function run: vRunApplication()

This task is started after the network is online and initialized. This task is the main task which creates all the sub-tasks to run our client-side program. This function creates the initialize client socket task with a high priority and waits for its completion. Then, it creates Canny edge detection task, RSA encryption task and the TCP send task. Here, its gives away control to the canny filter tasks and awaits notification to create the next set of tasks for the next frame.

1. Load Image

This is not an actual task. It is a function which loads the image and stores the loaded image in a global variable after the network is initialized. This is done for the sake of simplicity and run-time reduction for best utilizing our sample image. When this application is to be deployed in real-life scenario, this can be converted to a task and be initialised from the above run application task.

1. Initialize client socket

Name of the task: “Client”  
Task handle variable: ‘client\_handle’  
Priority: 5  
Function run: initialise\_client\_socket()

This is the first task run by the Run-Application task to set up the TCP client socket and connect it to the server socket with an its IP address. The actual creation and connection happens in vCreateTCPClientSocket() function present in TCP\_client.c. This outer function present in main.c only provides with the server IP address, and notifies the Run-Application task that it has done its job. The run application task waits on this notification.

After completing its work, its priority is significantly decreased to 2 (just above idle task priority), and waits in background for the client socket to be available for the send function.

1. Canny Edge Detection Task

Name of the task: “Canny”  
Task handle variable: ‘canny\_handle’  
Priority: 7  
Function run: CannyFilter()

This function awaits notification from the run-application task. This ensures that all the tasks are initialized properly before executing this function. Here, edge detection parameters are provided, edge detection is performed, the output is put in a queue, a notification is given the next sequential task of compression and encryption and lastly, it deletes itself.

Refer the below Edge Detection section for more details.

1. Data Compression and RSA Encryption

Name of the task: “encrypt”  
Task handle variable: ‘encryption\_handle’  
Priority: 6  
Function run: RSAencryption()

This function awaits notification from the edge detection task. This ensures a sequential processing of data. Here, the image with edges detected is received from the queue, it is processed and then the processed data is put back into the same queue. Then, a notification is given the next sequential task of sending the data over TCP and lastly, it deletes itself.

The important thing to notice over here is that the data is compressed using linear compression before it is encrypted. This saves processing time for encryption and also saves time to send the data later on.

It also puts an end of frame byte.

Refer the below Encryption section for more details.

1. Send data

Name of the task: “send”  
Task handle variable: ‘send\_handle’  
Priority: 6  
Function run: vTCPSend()

This function uses the previously created and connected client socket to send the data frame. The actual processing happens in vSendMessage() function present in TCP\_client.c file.

## Server-Side Tasks:

1. Initialize server socket

Name of the task: "Initialise Server"  
Task handle variable: NA  
Priority: 2  
Function run: vCreateTCPServerSocket()

This function creates the server socket, binds it to port 7, sets it in listening state and then accepts any new connections and creates a new socket to receive the data. Further, this task then creates a new task to receive data for every new connection.

1. Receive TCP data

Name of the task: "Receive Data"  
Task handle variable: NA  
Priority: 3  
Function run: xRecieveTCPData()

This function receives the data and stores it in a buffer. When end of file is reached, this creates a new task for landing guidance processing using the received data. This task has a higher priority than the main server task itself since we are receiving one frame at a time.

1. Landing Guidance

Name of the task: "Landing guidance"  
Task handle variable: NA  
Priority: Idle task priority  
Function run: Landing\_guidance()

This is a dummy function for the sake of representational purposes only. In the real-time deployment scenario, the actual server-side decryption and processing is supposed to take place here.

# Edge Detection:

The given Canny edge detection is modified in such a way that it has the following reusable function blocks:

* Load\_bmp()
* Save\_bmp()
* Canny\_edge\_detection()
* Gaussian\_filter()
* Convolution()

Here in our project, we only use the load\_bmp() and the canny\_edge\_detection() functions. However, for demonstration purposes, that is to save the sample outputs the save\_bmp() is also used.

Original image versus edge detected image can be seen below:   
These outputs are also stored in the “outputs” sub-folder.

# Encryption:

The given RSA encryption is modified in such a way that it has the following reusable function blocks:

* Encrypt() & decrypt()
* Rsa\_encryption() & Rsa\_decryption()

These functions take in an array of data, generate their own encryption/decryption key with the default prime numbers: 37 & 41 (High enough for secure results) and encrypt/decrypt.

**Important modifications:**

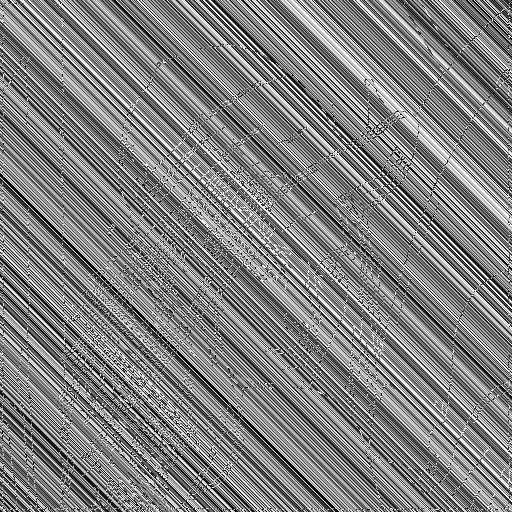
The code has been modified to first compress the array of data using linear compression. This significantly reduces the size of the array by almost 90%. Then, this can be easily encrypted quickly and securely. Further, to enhance the security, based on the iterator value, a combination is added:

en\_msg[i] = msg[i]+(i/512)+(i%512);

For the sake of demonstration, the images below show before and after encryption using this process. However, to store the output back in an image format, it has to be decompressed. So, compression itself is avoided for demonstration.

Decrypted vs Encrypted:

These outputs are also stored in the “outputs” sub-folder.

# TCP Client Implementation:

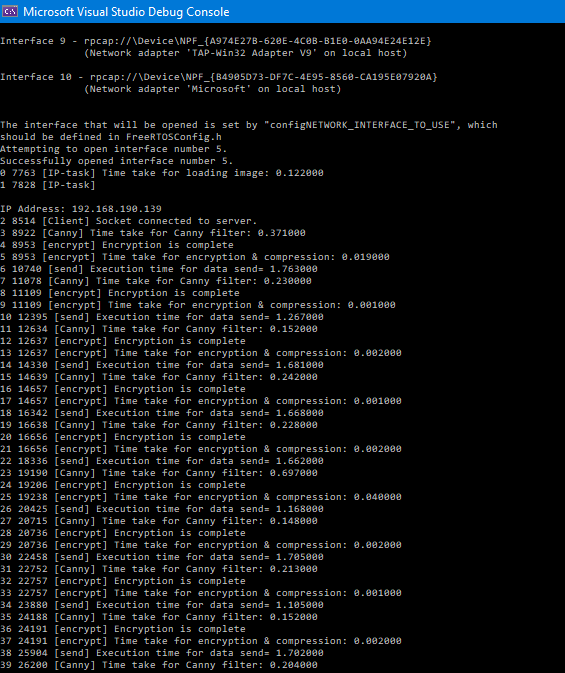
The following functionalities have been encoded in the TCP client module:

1. Create client socket  
2. Create a buffer to send the file.  
3. Connect the socket to the remote server socket  
4. Send data to the remote server socket  
5. Shutdown the socket  
6. Disconnect the socket  
7. Close the socket

However, in this project, we only get it use: Creation of the client socket, creation of file buffer, connecting the socket to the remote server socket and sending the data to the remote server socket. This is because, we intend to keep this connection running and keep on sending in frames.

If this were implemented in the real world, it can be scaled up and the other functionalities can be used to properly close the socket.

Screenshot of the transmission of data.



# Communication between the tasks:

The communication between the tasks takes place in two forms: Queue and notifications.

A queue is used to transfer the data in each of its stages of processing. It is because of it’s obvious advantages over global variables as taught in our course. We can reliably control which task/thread is reading or writing the data at a certain pointer address and the other tasks will wait out until something is once again put into the queue.

Task notifications on the other hand are light weight alternatives to binary semaphores provided in FreeRTOS. Instead of declaring a semaphore, its handle and utilizing it, a task can give notification to another task and the secondary task can await for the notification and is put into suspended state till then. Since we are using a sequential computation logic (where loading of image happens before the edge detection which happens before the compression and encryption which certainly happens before sending the data), it is advantageous to lock the tasks in this manner.

# TCP Server Implementation:

The following functionalities have been encoded in the TCP server module:

1. Create server socket

2. Create a buffer to receive the file.

3. Bind the socket to a port

4. Set the socket in listening state and accept incoming connections

5. Receive the data and store in the buffer

6. Shutdown the socket

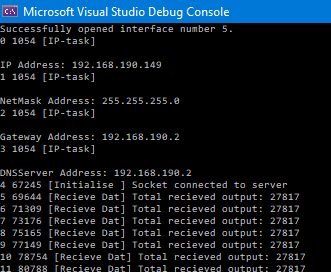
7. Close the socket

8. Landing guidance module

However, in this project, we only get it use: Creation of the server socket, creation of file buffer, binding to a port and listening on the socket, receiving the data, and storing it in the buffer.

If this were implemented in the real world, it can be scaled up and the other functionalities can be used to properly close the socket and guide the plane.

Screenshot of received data.



# Discussion and reference:

## Relevant techniques and hardware for interconnecting the camera in a real-world platform

Resolution/ Picture quality:

It must be sufficiently high so that the edges can easily be detected. At the same time, it must be sufficiently low such that it does not computationally burden the processing hardware.

Energy consumption

Throughput requirement of the communication: Speed of the communication.

Reliability

Financial estimate of such a system.

## Reference to the choice of hardware and Operating System

Performance

Energy consumption

Availability

Cost of the OS

Relevance to the industry