Firmware para prototipo de vehículo de operación remota (rover)

Versión del documento: 0.1.0

Firmware para prototipo de vehículo de operación remota (rover)
- Descripción de la propuesta
  - Marco de la propuesta
  - Rover (versión 1.0)
  - Componentes de software
- Diagrama en bloques de la placa de control
- Conexionado de hardware
  - NUCLEO - L298N
  - L298N - Motores
  - Conexión serie con Raspberry Pi 3B+
  - NUCLEO - MPU9250
  - NUCLEO - GPS
  - NUCLEO - LM393
- Modelado de SW
  - Diagrama de clases
  - Protocolo de comunicación
  - Lazo de control
- Repositorio de código
- Ejemplos
- Referencia del protocolo de comunicación
  - Comandos
    - 0x03 UPDATE_MOTOR_SPEEDS
  - Reportes
    - 0x80 GENERAL_TELEMETRY_REPORT
    - 0x81 REPORT_COMMAND_EXECUTION_STATUS
- Referencias

Descripción de la propuesta

Se propone como trabajo final para el curso de Introducción a Sistemas Embebidos de la carrera de Especialización en Inteligencia Artificial la implementación en la placa Nucleo F767ZI del firmware de la placa de control de un vehículo de operación remota (generalmente llamado *rover* por sus siglas en inglés: *Remotely Operated Vehicle for Emplacement & Reconnaissance*).

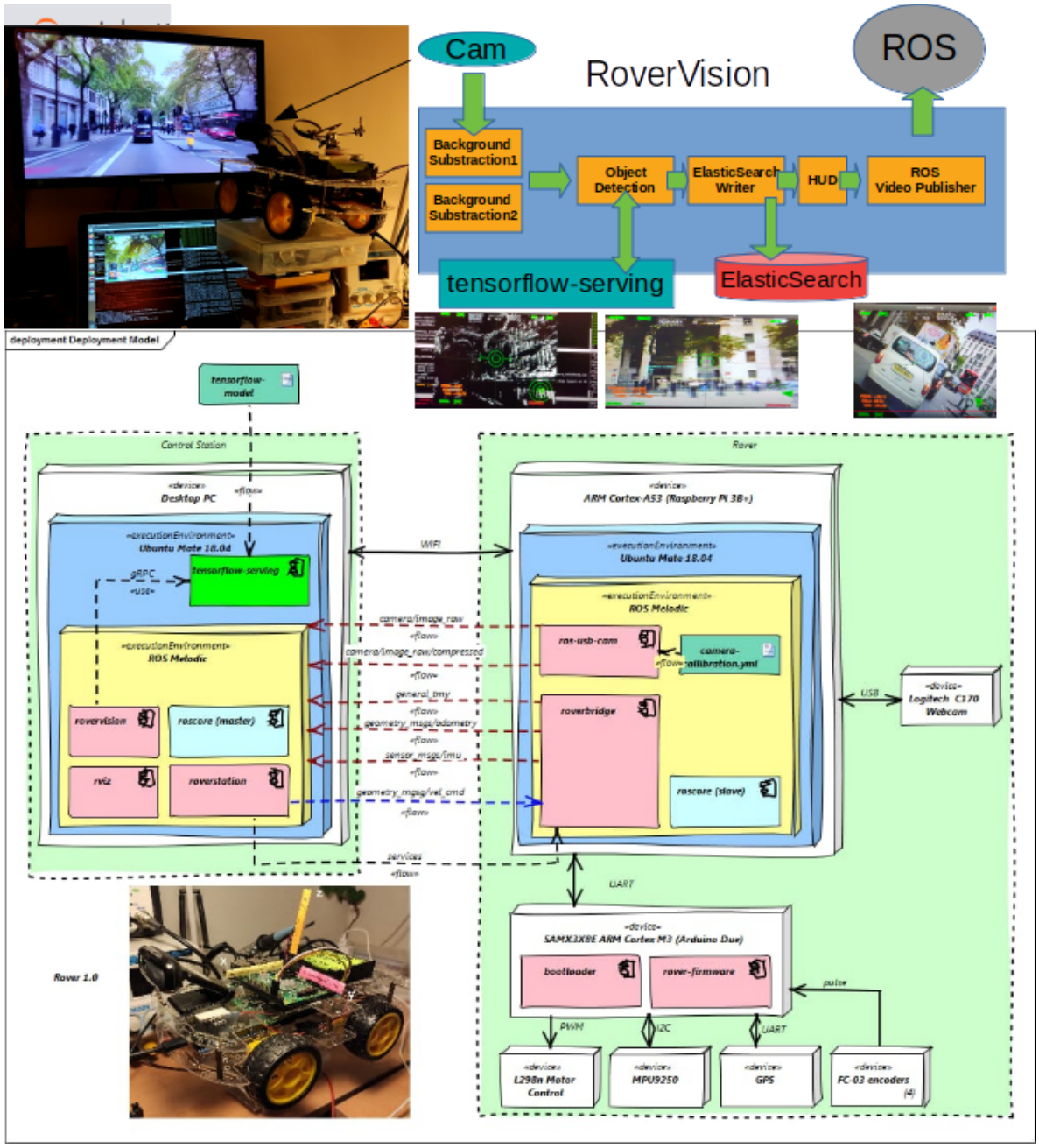
Marco de la propuesta

El presente trabajo es parte de un proyecto más grande en el que la placa de control del vehículo es uno de los subsistemas a desarrollar. El objetivo final del proyecto es contar con una plataforma para:

- Aprendizaje y prototipado de aplicaciones en el ambiente ROS (Robot Operative System)
- Aprendizaje y prototipado de aplicaciones de visión artificial.
- Experimentación con formas de navegación autónoma usando inteligencia artificial, comenzando por la detección de objetos en una escena e incorporando gradualmente otros sensores y técnicas: SLAM, etc.
- Prototipado de aplicaciones IoT en que se colecta algún tipo de información y se registra en bases de datos.

Se espera que, con excepción del firmware, el software desarrollado sea válido tanto para un rover real como para un modelo simulado (por ejemplo en Gazebo, Webots o V-Rep).

En el siguiente diagrama se muestra el sistema completo, para el que se se ha realizado una implementación preliminar del firmware en una placa arduino. También se utilizó una SBC Raspberry Pi 3B+ para ensayar la conexión serie con la placa de control.



Se utiliza una arquitectura típica para este tipo de sistemas, en las que como mínimo se dispone del vehículo y un software para operarlo (ground station).

Rover (versión 1.0)

Se utiliza el término **plataforma** para referirse al vehículo físico y su placa de control y **carga útil** a la computadora y otros sensores e instrumentos adicionales que permitan al vehículo cumplir una misión (obtener mediciones, manipular un brazo u otro mecanismo, explorar de forma autónoma, transmitir video, etc).

Esta división es habitual en este tipo de aplicaciones y tiene como objetivo separar las funciones mínimas que debe cumplir un robot teleoperado de aquellas más específicas del uso que se le quiere dar, permitiendo reutilizar el hardware y software de una misma plataforma en diferentes misiones.

A modo de ejemplo: si se trata de un drone se espera que la plataforma proporcione la capacidad de desplazar el vehículo, mantenerlo estabilizado y exponer su posición, actitud y velocidad pero no necesariamente esquivar obstáculos o definir una trayectoria.

Del mismo modo, se espera que un vehículo terrestre pueda ser comandado para avanzar a una velocidad determinada, detenerse, o cambiar su orientación, pero la planificación de trayectorias u otras tareas más complejas es competencia de otro subsistema.

En su primera versión el vehículo se construirá con los siguientes componentes:

Plataforma:

- 1x microcontrolador (placa NUCLEO-F767ZI) que implementa el lazo de control del vehículo.
- Sensores:
    - 4x tacómetros ópticos LM393.
    - 1x MPU9250.
    - 1x GPS Neo6M.
- Actuadores:
    - Módulo controlador de motores L298N.

Carga útil:

- Una computadora Raspberry Pi 3B+ para las funciones de alto nivel, como por ejemplo el software aplicativo, la comunicación con estación de control (WIFI), transmisión de video, etc. Se utiliza en la etapa temprana de desarrollo como sustituto más económico de una computadora más apropiada para la tarea, como por ejemplo NVIDIA Jetson.
    - Opcional: adaptador wireless USB para mejorar velocidad y rango de comunicación.
- Cámara USB: Logitech Webcam C170.

Alimentación:

- Se utilizan baterías 18650 para el microcontrolador y motores y una batería USB para la computadora.

Nota: para el chasis se utiliza el kit 4WD que incluye 4 motores DC c/ reducción y ruedas.

Componentes de software

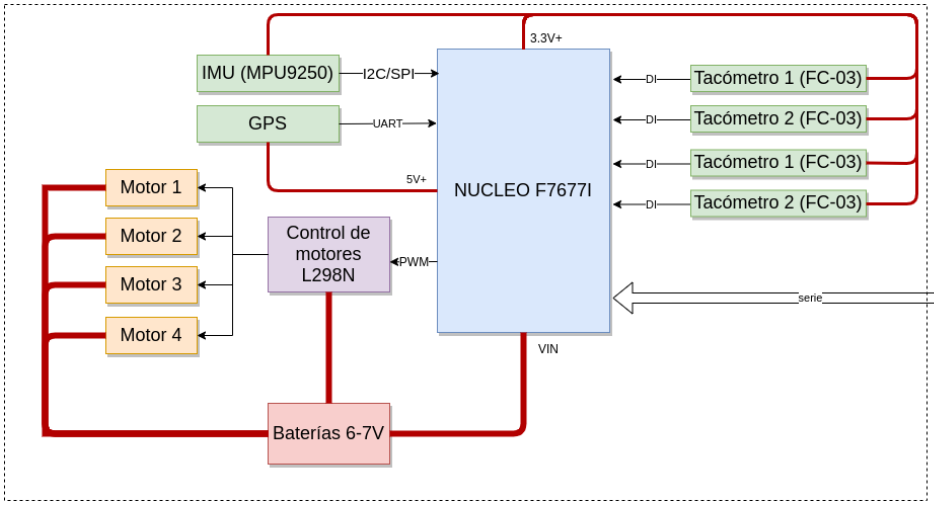
El código está organizado siguiendo la filosofía de ROS de separar las aplicaciones y conectarlas en una red TCP/IP donde se publican y se suscriben a datos de interés mediante un sistema de mensajería. Esto permite construir aplicaciones complejas aprovechando la extensa cantidad de componentes disponibles para todo tipo de tareas: planificación de trayectoria, corrección de distorsión de cámara, fusión de datos, IA, etc. además de facilitar la integración prácticamente con cualquier otro SW existente.

Puede hacerse una primera división del software entre el firmware y el código de aplicación. Para el trabajo final de este curso **únicamente se desarrollará el firmware**.

- Código de Aplicación (ARM o x86): desarrollado en Python o C++ usando ROS para ejecutarse en Ubuntu en ARM o x86, dependiendo de la exigencia de la tarea. Organizado como servicios/aplicaciones separadas:
    - **Bridge con ROS (roverbridge)**: comunicación con firmware por UART. Envío de comandos y recepción de telemetría usando un protocolo sencillo similar a MAVLINK. Publicación de telemetría en los formatos de ROS (posición de cuaternión de orientación del vehículo, variables de estado, contadores, etc.).
    - **Estación de control (roverstation)**: aplicación para teleoperación del rover con dos modos: modelo uniciclo o controlando directamente los motores.
    - **Sistema de visión (rovervision)**: cadena de procesamiento de video que funciona por bloques. Cada imagen entrante es recibida por cada componente de la cadena (se pueden configurar, habilitar y deshabilitar los componentes en tiempo de ejecución). Cada componente puede escribir información en un contexto global o publicarla o almacenarla en otro medio, de manera que se vaya enriqueciendo la información disponible de la escena, guardando alguna similaridad con el patrón de diseño blackboard. De este modo, un componente puede por ejemplo extraer el fondo, otro realizar detección de objetos, otro dibujar un HUD, etc.
- Firmware (NUCLEO F67ZI):
    - Código en C/C++ desarrollado para MbedOS siguiendo estándares de programación de software embebido.
    - Modelo de ejecución: único hilo (a confirmar).
    - Código con el lazo de sensado y control que comprende:
        - Lectura de sensores.
        - Control de actuadores.
        - Comunicación por puerto serie con computadora utilizando un protocolo propio basado en mensajes (similar a MAVLINK) que incluye:
            - Un conjunto de comandos básicos de control: establecer velocidad, detenerse, etc.
            - Publicación períodica de telemetría: estado del sistema, lecturas de sensores, etc.

Diagrama en bloques de la placa de control

El siguiente diagrama muestra los componentes del sistema



- La lectura del sensor IMU se implementará por I2C o SPI (los dos modos de comunicación soportados por el sensor).
- La lectura del GPS se realizará por UART.
- Los tacómetros se conectarán a pines de entrada digitales. Nota: se utilizarán capacitores para reducir el ruido en la lectura (no se muestran en el diagrama)
- El control de los motores se realizará por salidas pulsadas de ancho modulado (PWM).

Alimentación: se utilizarán dos baterías 18650 (8.4V) para alimentar el controlador de los motores, los motores y la placa. Para los sensores se utilizará la tensión de entrada recomendada en el datasheet de cada sensor utilizando las salidas de la placa NUCLEO (3.3V o 5V).

Conexionado de hardware

Sección en preparación para futura consulta de asignación de pines. Se completará más avanzado el proyecto.

NUCLEO - L298N

Pin L298N	Pin Nucleo
ENA	D0
IN1	D1
IN2	D2
IN3	D3
IN4	D4
ENB	D5

L298N - Motores

Pin L298N	Motor
OUT1	LF, LB (-)
OUT2	LF, LB (+)
OUT3	RF,RB(+)
OUT4	RF,RB(-)

Nota: con el vehículo apuntando hacia adelante: LF=Left/Front, RB=Right/Back,etc.

Conexión serie con Raspberry Pi 3B+

Pin NUCLEO	Pin Raspberry Pi 3B+
GND	6 (GND)
RX (?)	8 (TXD)
TX (?)	10 (RXD)

NUCLEO - MPU9250

Pin NUCLEO	Pin MPU9250
VCC	3.3V
GND	GND
SDA (?)	SDA
SCL (?)	SCL

NUCLEO - GPS

Pin NUCLEO	Pin GPS
GND	GND
RX (?)	Tx
TX (?)	Rx

NUCLEO - LM393

Pin Nucleo	Pin LM393
GND	GND
??	LF
??	RF
??	LB
??	RB

Modelado de SW

- Esta sección se completará más avanzado el proyecto.

Diagrama de clases

- Organización del código.

Protocolo de comunicación

Para comunicarse con la computadora, se utiliza un protocolo de SW basado en mensajes, donde cada paquete tiene el siguiente formato:

```
"PKT!<payload><crc16>\n"
```

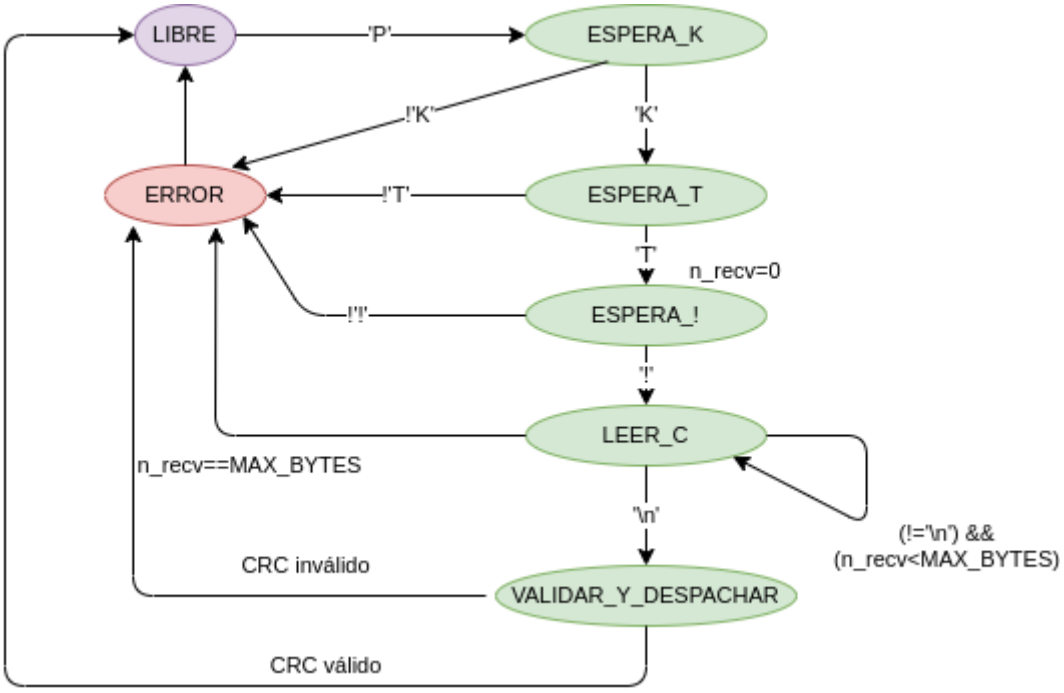
siendo:

- "PKT!" la palabra de sincronismo (4bytes).
- *payload* es el la carga útil del paquete, una secuencia de un máximo de 64 bytes.
- *crc16* es el código CRC del payload.
- \n es el byte terminador del paquete.

Se definen dos tipos de paquetes:

- **Comandos**: la carga útil se compone de un byte indicando un código de operación (OPCODE) seguido de datos específicos del comando.
- **Reportes**: la carga útil consiste en un byte indicando un código de reporte (REPORT_ID) seguido de datos específicos de cada reporte. Es mandatoria la implementación como mínimo de dos tipos de reporte:
    - Telemetría.
    - Resultado de ejecución de un comando.

El siguiente diagrama muestra la máquina de estados que se utiliza para reconocer el protocolo.



Secuencia nominal:

1. El estado inicial es *LIBRE* (es decir, se está a la espera de iniciar la recepción de un paquete).
2. Ante la llegada de un nuevo caracter, si el mismo coincide con el esperado para la secuencia de sincronismo se avanza al siguiente estado.
3. Una vez reconocido el último caracter de la secuencia de sincronismo se inicializa en cero un contador de bytes leidos *n_recv* y se pasa al estado de recepción de bytes *LEER_C*.
4. En el estado *LEER_C* cada nuevo byte recibido se almacena en un buffer y se incrementa *n_recv*. El máximo es *MAX_BYTES*=64.
5. Se sale de *LEER_C* si se recibe el caracter terminador. En este caso se pasa al estado de *VALIDAR_Y_DESPACHAR*.
6. En el estado de validación se realiza el chequeo de CRC, y si es válido se despacha el paquete (es decir, se ejecuta la función asociada a ese OPCODE). De lo contrario se pasa a *ERROR* con el código *CRC_INVÁLIDO*.

Error:

1. Recibir un caracter inesperado, superar la cantidad máxima de bytes de un paquete o un error en CRC llevará al estado de error. En este estado se incrementará el contador del tipo de error y se transicionará al estado inicial.

Lazo de control

- Esta sección se completará más avanzado el proyecto.
- Descripción de la secuencia del lazo y parámetros.

Repositorio de código

- Esta sección se completará más avanzado el proyecto.

Ejemplos

- Links a videos e imagenes.

- Esta sección se completará más avanzado el proyecto.

Referencia del protocolo de comunicación

A modo de ejemplo, se describen en esta sección los comandos y reportes a implementar. Se actualizarán una vez más avanzada la aplicación.

Comandos

Opcode	Mnemónico	Descripción	Parámetros
0x00	REQUEST_TMY	Solicitar telemetría. Se generará un reporte general de telemetría.	Ninguno.
0x01	LED_ON	Encender led de prueba.	Ninguno.
0x02	LED_OFF	Apagar led de prueba.	Ninguno.
0x03	UPDATE_MOTOR_SPEEDS	Actualizar velocidades de los motores.	Ver detalles más abajo.

0x03 UPDATE_MOTOR_SPEEDS

Actualizar las velocidades de los motores. Cada velocidad se indica como un int16 entre -255 y 255.

Byte Offset	Descripción
0	MSB de velocidad de motor A
1	LSB de velocidad de motor A
2	MSB de velocidad de motor B
3	LSB de velocidad de motor B
4	Flags. b01: habilitar motor A, b10: habilitar motor B.

LSB/MSB: less/most significant bit.

Reportes

0x80 GENERAL_TELEMETRY_REPORT

Este reporte se genera en respuesta a un comando REQUEST_TMY.

Offset	Parámetro	Descripción
0	REPORT_TELEMETRY_REQUEST(0x80)	Identificador del reporte.
1	TMY_PARAM_ACCEPTED_PACKETS	Cantidad de paquetes aceptados (un pedido de telemetría no incrementa este contador).
2	TMY_PARAM_REJECTED_PACKETS	Cantidad de paquetes rechzados (un pedido de telemetría no incrementa este contador).
3	TMY_PARAM_LAST_OPCODE	Último OPCODE recibido.
4	TMY_PARAM_LAST_ERROR	Código de error del último comando ejecutado.

0x81 REPORT_COMMAND_EXECUTION_STATUS

Este reporte se genera automáticamente como resultado de la ejecución de un comando.

Offset	Parámetro	Descripción
0	REPORT_COMMAND_EXECUTION_STATUS(0x81)	Identificador del reporte.
1	OPCODE	OPCODE ejecutado.
2	STATUS_CODE	Código de status de la ejecución.

Referencias

- Material del curso Introducción a los sistemas embebidos.
- Arquitectura para robots de exploración
    - Software Architecture for Planetary & Lunar Robotics
    - Software libre para drones y otros vehículos teleoperados:
        - Learning the Ardupilot codebase
        - Liprepilot system architecture
- Protocolos de telecomandos y telemetría para sistemas de aviónica:

    - ECSS-E-ST-70-41C – Telemetry and telecommand packet utilization (15 April 2016)
    - MAVLink Developer Guide
- Documentación de MBedOS:
    - MBed API references and tutorials - Standard Pin Names
    - MBed API references and tutorials - I2C
    - MBed API references and tutorials - BufferedSerial
    - MBed API references and tutorials - PwmOut
    - MBed API references and tutorials - CRC