Análisis software del sistema de tracción con ROS 1

https://github.com/racarla96/ros1_caddy_ai2_rbcar_robot/tree/indigo-devel/curtis_motordrive

A continuación se presenta un análisis general del código contenido en el archivo **curtis_controller.cpp** del paquete *curtis_controller*, que forma parte del sistema de control para un motor drive basado en ROS:

---

1. Arquitectura General y Objetivos

- **Propósito:**
  El controlador (*CurtisController*) se encarga de gestionar la comunicación entre la red CAN y el sistema ROS para controlar un motor (o conjunto de motores) de la plataforma. Esto incluye la recepción de comandos (ya sea de tipo “throttle” o “twist”), la lectura de datos desde el bus CAN y la publicación del estado del sistema.
- **Componentes Clave:**
  - **Interfaz CAN:** Se utiliza un objeto `PCan` (representado por `can_dev`) para la comunicación con el bus CAN.
  - **Control del Motor:** El objeto `MasterDrive` se encarga de enviar comandos de aceleración (throttle) y de procesar los mensajes CAN que provienen del drive.
  - **Integración ROS:**
    - **Publicadores:** Publica el estado general en el topic `"state"` y datos específicos del drive en `"master_drive"` y `"slave_drive"`.
    - **Suscriptores:** Se suscribe a comandos de control (por ejemplo, `"command_throttle"` o `"command_twist"`) dependiendo del modo configurado.
    - **Servicios:** Posee un servicio para resetear el controlador.

---

2. Máquina de Estados

El funcionamiento central del controlador se organiza en una máquina de estados implementada en el método `controlLoop()`, la cual revisa el valor de la variable `state` y ejecuta la función correspondiente. Los estados son definidos mediante constantes (probablemente en el paquete

`robotnik_msgs::State`):

- **INIT_STATE:**
  - **Función:** `initState()`
  - **Acción:** Llama a `setup()` para inicializar los subcomponentes (CAN, drive, etc.).
  - **Transición:** Si la inicialización es exitosa, cambia al estado **READY_STATE**.
- **STANDBY_STATE:**
  - **Función:** `standbyState()`
  - **Acción:** Se enfoca en la lectura de mensajes del bus CAN (llamando a `readCANMessages()`), pero sin enviar comandos de throttle.
- **READY_STATE:**
  - **Función:** `readyState()`
  - **Acción:**
    - Lee mensajes CAN y, en base a ellos, gestiona la comunicación con el drive.
    - Verifica condiciones de seguridad, como la existencia de la interlock.
    - Envía el valor de aceleración (throttle) a través de `master_drive->sendThrottle()`, aplicando límites para evitar exceder el rango permitido.
  - **Control de Fallos:** Si falla la lectura de mensajes CAN o se detecta un error al enviar el throttle, se transita a **FAILURE_STATE**.
- **FAILURE_STATE:**
  - **Función:** `failureState()`
  - **Acción:** Permite una recuperación tras un tiempo predefinido (controlado por `failure_recover_time`). Se intenta reinicializar el sistema, y si tiene éxito, se cambia a **READY_STATE**.
- **SHUTDOWN_STATE:**
  - **Función:** `shutdownState()`
  - **Acción:** Ejecuta el proceso de apagado (`shutdown()`) de los dispositivos y transita de vuelta al estado **INIT_STATE**.
- **EMERGENCY_STATE:**
  - **Función:** `emergencyState()`
  - **Acción:** Actualmente se define pero sin implementación. Se reserva para tratar situaciones de emergencia.

---

3. Flujo de Ejecución y Control

- **Inicio y Bucle de Control:**
  El método `start()` configura ROS (llamando a `rosSetup()`), activa la bandera `running` y ejecuta el `controlLoop()`, el cual se ejecuta a una frecuencia determinada (`desired_freq_`) leída desde el servidor de parámetros o con un valor por defecto.
- **Control Loop:**
  Dentro de este ciclo:
  - Se registra el tiempo para calcular la frecuencia real de ejecución.
  - Se ejecuta el bloque correspondiente al estado actual (con `switch(state)`).

- Se llaman a funciones comunes en todas las iteraciones a través de `allState()`, que entre otras cosas, implementa un watchdog (para poner a cero el throttle si no se reciben comandos en un tiempo determinado) y publica el estado a ROS.
- **Detección y Procesamiento de Mensajes CAN:**
  El método `readCANMessages()` intenta leer varios mensajes (en un bucle) del bus CAN y, para cada mensaje leído, llama a `master_drive->processCANMessage()`. Esto es fundamental para actualizar el estado del drive (por ejemplo, datos de velocidad, fault codes, interlock, etc.).

---

4. Interacción mediante ROS

- **Suscriptores:**
  Dependiendo del modo configurado (definido en el parámetro `"mode"`), se suscribe a:
  - `"command_throttle"`: Para recibir comandos de aceleración directa.
  - `"command_twist"`: Para recibir comandos en forma de twist (probablemente para convertir velocidad lineal y angular a un comando de throttle).
- **Publicadores:**
  - **Estado del Controlador:** Publica un mensaje del tipo `robotnik_msgs::State` que incluye el estado actual, la frecuencia deseada y la real, así como una descripción en cadena del estado.
  - **Datos del Motor:** Publica datos específicos de la conducción (por ejemplo, velocidad, interlock, fallos) en dos topics: `"master_drive"` y `"slave_drive"` con mensajes del tipo `curtis_msgs::DriveData`.
- **Servicio de Reset:**
  Se implementa un servicio denominado `"reset"` que permite reiniciar el controlador (a través de la función `resetSrv`).

---

5. Gestión de Errores y Seguridad

- **Watchdog:**
  En el método `allState()`, se monitoriza el tiempo desde el último comando recibido. Si se supera un tiempo límite (`WATCHDOG_TIMEOUT`), se fuerza la reducción del throttle a cero para evitar movimientos inesperados.
- **Control de Interlock:**
  El sistema verifica constantemente el estado del interlock:
  - Si no está activado, se establece el throttle deseado a cero para prevenir el funcionamiento sin la debida seguridad.
  - Se detecta la transición en el interlock (por ejemplo, cuando se recupera) y se emiten mensajes de advertencia.

- **Transiciones por Error:**
  Si se producen errores en la lectura de mensajes CAN o en el envío de comandos a través del drive, el sistema transita al estado **FAILURE_STATE** y posteriormente intenta una recuperación.

Conclusión

El código del *CurtisController* está organizado de forma modular y robusta, utilizando una máquina de estados que permite gestionar de manera clara las diferentes fases de la operación:

- **Inicialización y Configuración (INIT_STATE y rosSetup):** Se leen parámetros, se configuran los dispositivos CAN y se preparan los publishers/subscribers de ROS.
- **Operación Normal (READY_STATE):** Se realizan lecturas y escrituras constantes sobre el bus CAN, se envían comandos al drive y se publica el estado.
- **Gestión de Fallos (FAILURE_STATE y EMERGENCY_STATE):** Se implementa una estrategia para detectar fallos y recuperar el control tras un tiempo de espera.
- **Apagado Seguro (SHUTDOWN_STATE):** Se asegura la liberación de recursos y se prepara el sistema para una nueva inicialización.

En resumen, se trata de un controlador para sistemas embebidos en robótica que integra comunicación a bajo nivel (CAN) con la capa de mensajería y servicios de ROS, siguiendo un patrón basado en estados que facilita tanto la operación normal como el manejo de situaciones de error o emergencia.

Revision #1
Created 23 March 2025 14:57:04 by Rafael Carbonell Lázaro
Updated 22 May 2026 10:46:15 by Rafael Carbonell Lázaro