Projektbeschreibung Studienarbeit Mikrocontroller (Jonas Karl) – Szenarioerkennung mit Weboberfläche (L475E-IOT01A2)

Dieses Projekt realisiert eine **szenariobasierte Kollisionswarnung** mit Sensordaten-Auswertung und **Visualisierung über einen integrierten Webserver**.

I. Ablauf des Programms

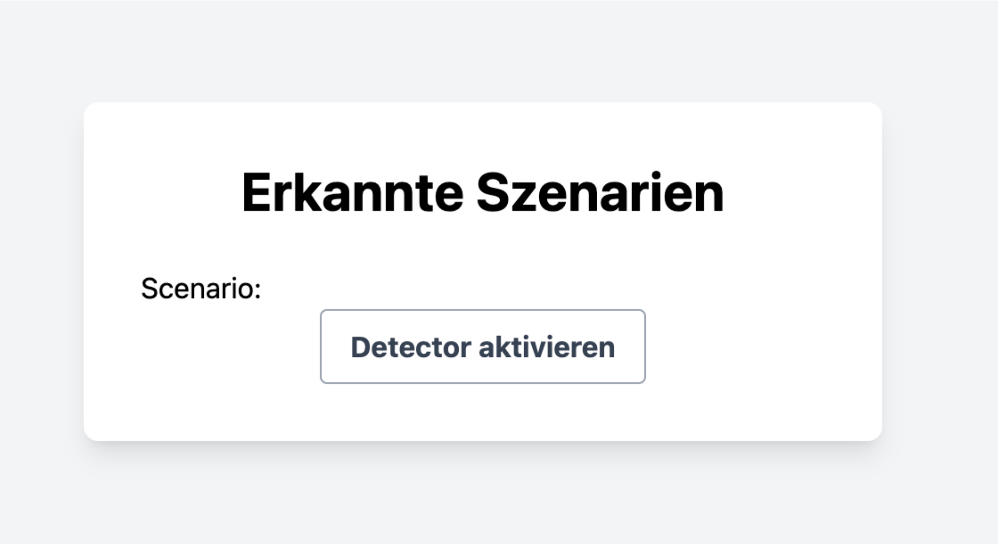
1. **Systeminitialisierung**
   - Initialisierung der Peripherie: UART, Clock, Timer, Beschleunigungssensor, Gyroskop, ToF-Distanzsensor.
   - Kalibrierung von Beschleunigungs- und Gyrosensor.
   - Start des Webservers auf dem Board.
2. **Webservermodus**
   - Der integrierte HTTP-Server liefert eine Webseite.
   - Die Webseite zeigt die aktuelle Szenario-Erkennung und bietet einen Button zum Starten des Detektors.
   - Nach Klick auf „Detector aktivieren":
     - Der Webserver wird gestoppt.
     - Die Sensordatenerfassung (Timer-Interrupt-gesteuert) wird aktiviert.
3. **Szenarioerkennung**
   - Alle 100 ms werden Distanz- und Bewegungsdaten vom Sensor gelesen.
   - Ein gleitender Mittelwert der Distanz wird gebildet.
   - Die Geschwindigkeit wird als $\Delta s / \Delta t$ berechnet.
   - Eine Zustandsmaschine mit drei Zuständen erkennt, ob ein mögliches Kollisionsszenario vorliegt:
     - **NO_SCENARIO**: Nichts erkannt
     - **POSSIBLE_SCENARIO**: Annäherung mit negativer Geschwindigkeit
     - **SCENARIO_DETECTED**: Kollisionsgefahr (nicht mehr rechtzeitig bremsbar)
4. **Kollisionsszenario erkannt**
   - Sensorwerte und Berechnungen (Stopping Distance, Geschwindigkeit etc.) werden im Textformat gespeichert.
   - Timer wird gestoppt.

- Webserver wird erneut gestartet.
- Das erkannte Szenario wird per JSON an die Webseite übermittelt und angezeigt.

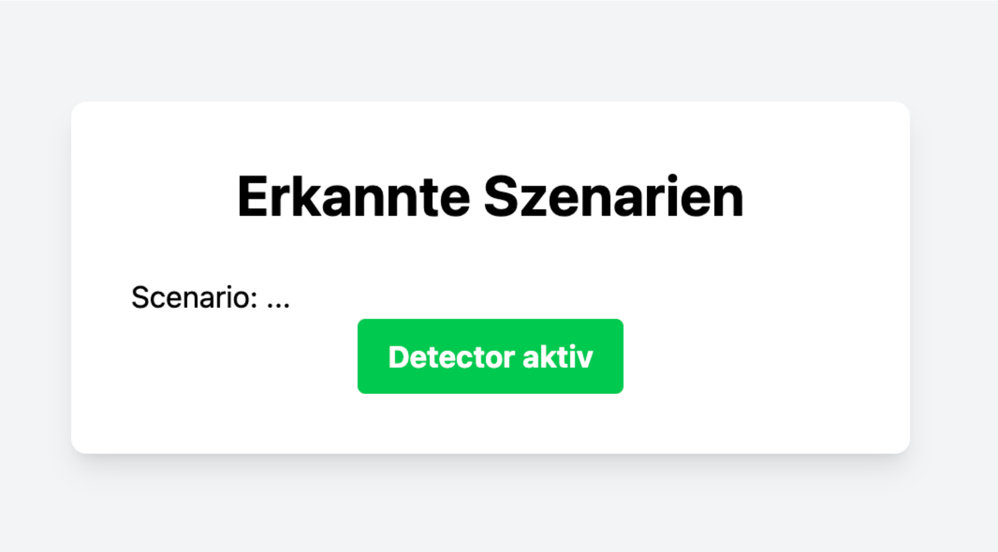
II. Web-Oberfläche

- HTML + TailwindCSS.
- Periodischer Abruf (fetch('/data')) holt alle 2 Sekunden den aktuellen scenario_text.
- Wird ein Szenario erkannt, wird die entsprechende Beschreibung in der Oberfläche angezeigt.

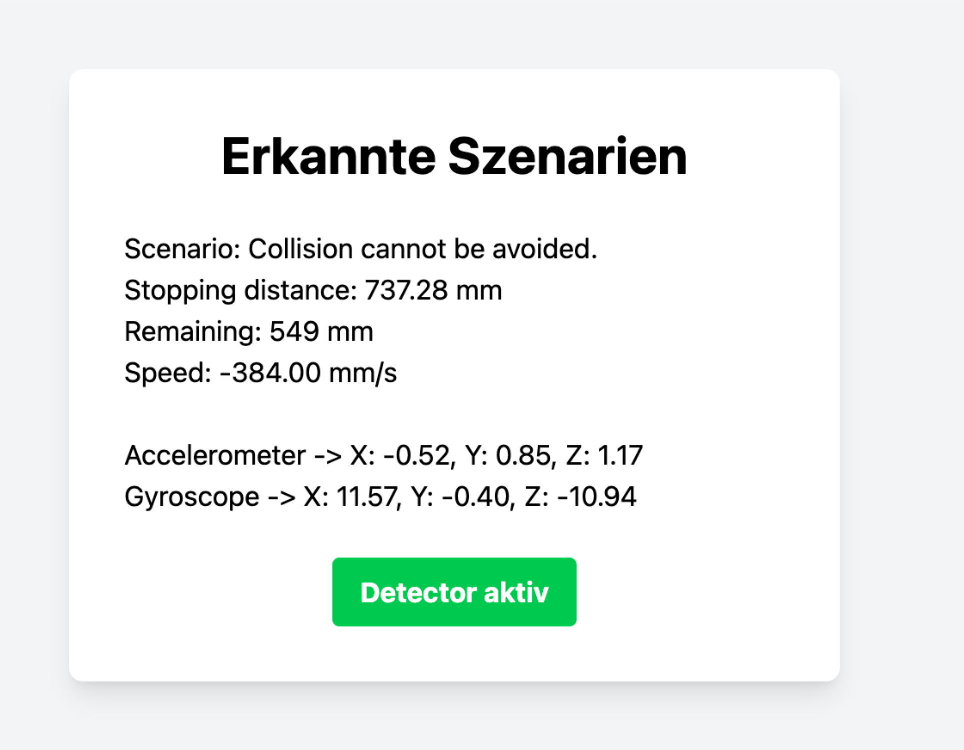
1. **Initialer Aufruf der Webseite, Detektor wurde noch nicht aktiviert.**



2. **Nach einem Klick auf „Detector aktivieren" wird der Button grün. Nun beginnt die Szenarienerkennung. Eine 5-sekündige Anfrage an den Webserver prüft, ob ein Szenario erkannt wurde.**



3. **Ein Szenario wurde erkannt, die Details werden entsprechend dargestellt.**



III. Ziel des Programms & Entscheidungsgründe

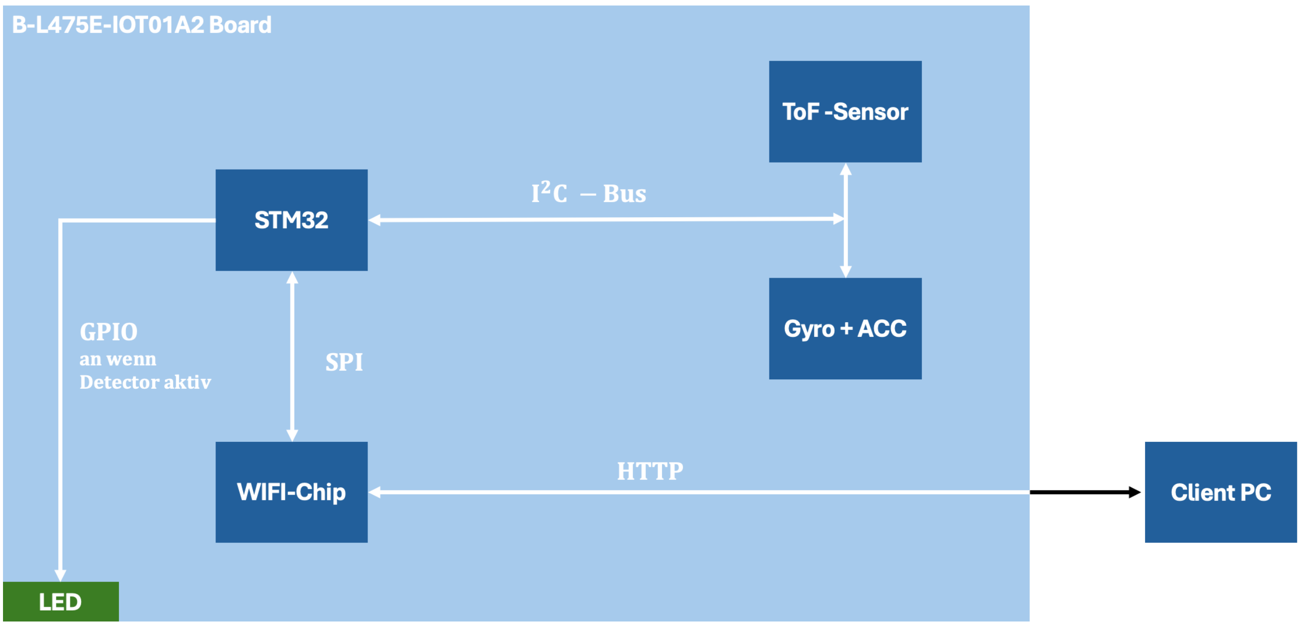
Ein System zur Gefahrenerkennung mit direkter Visualisierung über WiFi und Webbrowser. Es kombiniert Sensorfusion, einfache Zustandslogik und WLAN-Kommunikation auf einem Mikrocontroller.

Warum wird der Webserver nach aktivieren des Detektors gestoppt? Leider ist der WiFi-Stack sehr empfindlich gegen Interrupts, um die Stabilität zu erhöhen, wird dieser daher während der Sensorverarbeitung deaktiviert.

Warum wird die Geschwindigkeit nur über den Range-Sensor ermittelt? Leider hat das Accelerometer trotz vorheriger Kalibrierung & Filterung zu viel Drift und Rauschen und ist deshalb nicht ideal. Der Range-Sensor ist genau und lässt zudem die relative Geschwindigkeit ermitteln (vorteilhaft z.B. bei vorrausfahrendem Fahrzeug).

IV. Komponenten und Kommunikation

Komponente	Beschreibung
ToF (VL53L0X)	Time-of-Flight Sensor misst kontinuierlich die Distanz zum Objekt.
Accelerometer	Misst Beschleunigung in drei Achsen (XYZ).
Gyroskop	Misst Rotationsgeschwindigkeit in drei Achsen (XYZ).
STM32L4 MCU	Führt Datenerfassung, Filterung, Zustandslogik und Webserver aus.
Szenario-Logik	Zustandsautomat erkennt mögliche und tatsächliche Gefahrensituationen.
Webserver	Stellt HTML-Webseite bereit & JSON Daten bereit.
WLAN-Modul (ES-WIFI)	Verbindung zu Client über HTTP.
Webinterface (Client)	HTML-basierte Benutzeroberfläche zur Anzeige des aktuellen Szenarios.



V. Verbesserungsvorschläge

- Trennung der Logik in mehrere Dateien (aktuell viel in main.c)
- Paralleler Betrieb von WIFI-Server und Detektor

VI. Doku von main.c

main.c File Reference

Combined Webserver and Scenario Detection for STM32L475E-IOT01A2. More...

```
#include "main.h"
```

Macros

#define	**SSID** "WLAN-070011"
#define	**PASSWORD** "xxx"
#define	**PORT** 80
#define	**SOCKET** 0
#define	**WIFI_WRITE_TIMEOUT** 10000
#define	**WIFI_READ_TIMEOUT** 10000
#define	**ALPHA** 0.1f
#define	**DIST_SAMPLE_INTERVAL** 0.5f
#define	**CALIB_SAMPLES** 100
#define	**DIST_AVG_SAMPLES** 3
#define	**MAX_DECEL** 100.0f
#define	**TERMINAL_USE**
#define	**LOG**(a)
#define	**PUTCHAR_PROTOTYPE** int fputc(int **ch**, FILE *f)

Enumerations

enum	**ScenarioState** { **NO_SCENARIO** = 0 , **POSSIBLE_SCENARIO** , **SCENARIO_DETECTED** }

Functions

static void	**SystemClock_Config** (void) Configures the system clock to run at 80 MHz using PLL with MSI source.
static void	**MX_TIM2_Init** (void) Initializes TIM2 as a periodic interrupt timer for sensor readings.
static void	**calibrateSensors** () Calibrates accelerometer and gyroscope sensors.
static void	**read_sensors** () Reads sensor data and updates distance and speed.
static void	**check_scenario** () Evaluates the current sensor data to determine the presence of a critical driving scenario.
static int	**wifi_start** (void) Initializes the WiFi module and prints its MAC address.
static int	**wifi_connect** (void) Connects the WiFi module to the specified SSID and retrieves an IP address.

static int	**wifi_server** (void) Starts and manages a simple HTTP server on the WiFi module.
static bool	**WebServerProcess** (void) Processes incoming HTTP requests from the connected client.
static WIFI_Status_t	**SendWebPage** (void) Sends the main HTML UI page to the client via WiFi.
static WIFI_Status_t	**SendDataJson** (void) Sends scenario data as a JSON HTTP response.
static void	**read_vehicle_position_sensors** () Reads and filters vehicle position sensor data (accelerometer & gyroscope).
int	**main** (void) Application entry point.
void	**HAL_GPIO_EXTI_Callback** (uint16_t GPIO_Pin) External interrupt callback handler.
void	**SPI3_IRQHandler** (void) SPI3 interrupt handler.
void	**TIM2_IRQHandler** (void) TIM2 interrupt handler.
void	**HAL_TIM_PeriodElapsedCallback** (TIM_HandleTypeDef *htim) Callback for TIM2 periodic interrupts.

Variables

volatile **ScenarioState**	**scenario_state**
static int16_t	**acc_offset** [3] = {0}
static float	**gyro_offset** [3] = {0.0f}
static int	**distance** = 0
static uint16_t	**distance_buffer** [**DIST_AVG_SAMPLES**]
static uint32_t	**distance_sum** = 0
static uint8_t	**distance_index** = 0
static uint16_t	**distance_avg** = 0
static uint16_t	**previous_distance** = 0
static float	**speed** = 0.0f
static float	**acc_filtered** [3] = {0}
static float	**gyro_filtered** [3] = {0}
static uint8_t	**http** [4096]
static uint8_t	**IP_Addr** [4]
static int	**detectorStateUI** = 0
static char	**scenario_text** [1024]
UART_HandleTypeDef	**hDiscoUart**
TIM_HandleTypeDef	**htim2**
	PUTCHAR_PROTOTYPE

Redirects the printf output to UART for debugging.

return **ch**

Detailed Description

Combined Webserver and Scenario Detection for STM32L475E-IOT01A2.

Author

Jonas

Attention

This software is provided AS-IS under the terms in the LICENSE file.

Macro Definition Documentation

◆ ALPHA

```
#define ALPHA   0.1f
```

Low-pass filter alpha

◆ CALIB_SAMPLES

```
#define CALIB_SAMPLES   100
```

Number of calibration samples

◆ DIST_AVG_SAMPLES

```
#define DIST_AVG_SAMPLES   3
```

Moving average window size

◆ DIST_SAMPLE_INTERVAL

#define DIST_SAMPLE_INTERVAL 0.5f

Distance sampling interval (s)

◆ LOG

#define LOG (a)

Value:

```
printf a
```

◆ MAX_DECEL

#define MAX_DECEL 100.0f

Maximum deceleration (mm/s²)

◆ PASSWORD

#define PASSWORD ""

WiFi password

◆ PORT

#define PORT 80

HTTP server port

◆ PUTCHAR_PROTOTYPE

#define PUTCHAR_PROTOTYPE int fputc(int **ch**, FILE *f)

◆ SOCKET

```
#define SOCKET   0
```

TCP socket index

◆ SSID

```
#define SSID   "WLAN-070011"
```

WiFi SSID

◆ TERMINAL_USE

```
#define TERMINAL_USE
```

Use in Terminal

◆ WIFI_READ_TIMEOUT

```
#define WIFI_READ_TIMEOUT   10000
```

Read timeout (ms)

◆ WIFI_WRITE_TIMEOUT

```
#define WIFI_WRITE_TIMEOUT   10000
```

Write timeout (ms)

Enumeration Type Documentation

◆ ScenarioState

enum **ScenarioState**

Enumerator	
NO_SCENARIO	No collision scenario
POSSIBLE_SCENARIO	Potential collision scenario
SCENARIO_DETECTED	Collision detected

Function Documentation

◆ calibrateSensors()

void calibrateSensors (void) static

Calibrates accelerometer and gyroscope sensors.

This function collects multiple samples from the accelerometer and gyroscope to calculate and store offset values for each axis. These offsets are later used to filter sensor readings and reduce measurement errors.

◆ check_scenario()

void check_scenario (void) static

Evaluates the current sensor data to determine the presence of a critical driving scenario.

This function implements a simple finite-state machine to detect potentially dangerous driving scenarios based on distance and speed measurements. It transitions between states depending on whether a collision is likely or unavoidable. When a scenario is detected, sensor data is collected and formatted into a status message that is sent to the web client.

◆ HAL_GPIO_EXTI_Callback()

void HAL_GPIO_EXTI_Callback (uint16_t GPIO_Pin)

External interrupt callback handler.

Called when an external interrupt is triggered.

Parameters

GPIO_Pin The pin number which triggered the interrupt

Return values

None

◆ HAL_TIM_PeriodElapsedCallback()

void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef * htim)

Callback for TIM2 periodic interrupts.

Triggered every 100ms to update sensor values.

Parameters

htim Timer handle

Return values

None

◆ main()

int main (void)

Application entry point.

Return values

None

◆ MX_TIM2_Init()

void MX_TIM2_Init (void) static

Initializes TIM2 as a periodic interrupt timer for sensor readings.

Configures TIM2 to trigger every 100ms. Used to periodically update sensor data.

Return values

None

◆ read_sensors()

void read_sensors (void) static

Reads sensor data and updates distance and speed.

This function is typically called periodically by a timer interrupt. It reads the current distance from the proximity sensor, updates a moving average buffer, calculates the speed based on distance change, and triggers scenario detection logic if active.

◆ read_vehicle_position_sensors()

void read_vehicle_position_sensors (void) static

Reads and filters vehicle position sensor data (accelerometer & gyroscope).

This function reads raw data from the accelerometer and gyroscope sensors, applies calibration offsets, and filters the values using an exponential moving average to reduce noise.

◆ SendDataJson()

WIFI_Status_t SendDataJson (void) static

Sends scenario data as a JSON HTTP response.

This function generates a JSON string containing the scenario analysis result, formats it into an HTTP response, and sends it via the WiFi socket. If the send operation fails, it sends a fallback HTTP 500 error response.

Return values

WIFI_STATUS_OK	if the response was sent successfully.
WIFI_STATUS_ERROR	if sending failed or byte count mismatch occurred.

◆ SendWebPage()

static WIFI_Status_t SendWebPage (void) static

Sends the main HTML UI page to the client via WiFi.

This function constructs an HTML response including embedded JavaScript for periodically requesting scenario data via fetch. It includes a simple UI to display the current detection state and toggle the detector via a form.

Return values

WIFI_STATUS_OK	if the full page was successfully sent.
WIFI_STATUS_ERROR	if the page could not be sent completely.

◆ SPI3_IRQHandler()

void SPI3_IRQHandler (void)

SPI3 interrupt handler.

Forwards the interrupt to HAL SPI IRQ handler.

◆ SystemClock_Config()

static void SystemClock_Config (void) static

Configures the system clock to run at 80 MHz using PLL with MSI source.

This function sets up the oscillator, PLL multipliers/dividers and configures the AHB and APB buses for optimal performance. It will block in an infinite loop if any error occurs during configuration.

Return values

None

◆ TIM2_IRQHandler()

void TIM2_IRQHandler (void)

TIM2 interrupt handler.

Calls the HAL TIM IRQ handler for TIM2.

◆ WebServerProcess()

static bool WebServerProcess (void) `static`

Processes incoming HTTP requests from the connected client.

This function reads data from the WiFi socket, detects whether the request is a `GET /data`, `GET /`, or `POST`, and responds accordingly by sending scenario data or reloading the UI. It also handles state changes via POST.

Return values

- **true** if the server should stop (e.g., detector activated).
- **false** to continue accepting client connections.

◆ wifi_connect()

int wifi_connect (void) `static`

Connects the WiFi module to the specified SSID and retrieves an IP address.

This function initializes the WiFi module using the `wifi_start()` function, connects to the configured network, and retrieves the IP address.

Return values

- **0** if the connection and IP retrieval were successful.
- **-1** if connection or IP acquisition failed.

◆ wifi_server()

int wifi_server (void) `static`

Starts and manages a simple HTTP server on the WiFi module.

This function connects to WiFi, starts the HTTP server on the specified port, waits for incoming connections, and processes each request using `WebServerProcess()`. The loop runs until the `stopserver` flag is set.

Return values

- **0** if the server ran and stopped successfully.
- **-1** if an error occurred during connection or server operation.

◆ wifi_start()

```
static int wifi_start ( void )
```
static

Initializes the WiFi module and prints its MAC address.

This function initializes the WiFi hardware and attempts to retrieve the module's MAC address for logging purposes. If initialization or MAC retrieval fails, it returns an error.

Return values

- **0** WiFi module initialized successfully
- **-1** Initialization failed or MAC address could not be retrieved

Variable Documentation

◆ acc_filtered

```
float acc_filtered[3] = {0}
```
static

Low-pass filtered accel data

◆ acc_offset

```
int16_t acc_offset[3] = {0}
```
static

Accelerometer offsets

◆ ch

```
return ch
```

◆ detectorStateUI

```
int detectorStateUI = 0
```
static

UI button state (detector on/off)

◆ distance

int distance = 0

static

Current Distance

◆ distance_avg

uint16_t distance_avg = 0

static

Filtered distance

◆ distance_buffer

uint16_t distance_buffer[**DIST_AVG_SAMPLES**]

static

Circular buffer for distance avg

◆ distance_index

uint8_t distance_index = 0

static

Index in distance buffer

◆ distance_sum

uint32_t distance_sum = 0

static

Sum of valid distances

◆ gyro_filtered

float gyro_filtered[3] = {0}

static

Low-pass filtered gyro data

◆ gyro_offset

float gyro_offset[3] = {0.0f} static

Gyro offsets

◆ hDiscoUart

UART_HandleTypeDef hDiscoUart extern

UART handle for debug prints

◆ htim2

TIM_HandleTypeDef htim2

Timer2 handle for sensor interrupts

◆ http

uint8_t http[4096] static

Buffer for HTML/JSON responses

◆ IP_Addr

uint8_t IP_Addr[4] static

Acquired IP address

◆ previous_distance

uint16_t previous_distance = 0 static

Last distance reading

◆ PUTCHAR_PROTOTYPE

PUTCHAR_PROTOTYPE

Initial value:

```
{
  HAL_UART_Transmit(&hDiscoUart, (uint8_t *)&ch, 1, 0xFFFF)
```

Redirects the printf output to UART for debugging.

Allows standard output functions like printf to write via the UART interface.

Parameters

ch Character to write

Return values

Written character

◆ scenario_state

volatile **ScenarioState** scenario_state

Initial value:

```
=
    NO_SCENARIO
```

Current scenario FSM state

◆ scenario_text

char scenario_text[1024] static

HTML-formatted scenario message

◆ speed

float speed = 0.0f static

Computed speed (mm/s)