Integrated Brain ESP-IDF Component

This is the MCU firmware for a custom PCB designed to control, supply power to, and monitor a four-wheeled robot. It's built on a master-slave architecture, with an ESP32 as the slave and an SBC (such as an NVIDIA Jetson or Raspberry Pi) as the master.

Related projects:

- Integrated Brain PCB
- Integrated Brain Python Library

Note that this project has a lot of different registers, addresses, and buffers. These are referenced several times throughout the documentation, but in the interest of comprehensibility, they are all listed towards the end of the documentation only.

Overall design

Let's get an overview of the different concepts and categories of stuff (such as functions) involved in this component.

Direct and register-based tasks

In general, there are two types of functions in the component:

- **Direct task**

  These functions are typical functions. You call them (and pass in arguments if relevant) and the function will try to do whatever you told it to. Exactly how it behaves is completely transparent in this scenario: for example, if you told motor 3 to move forward with a PWM of 127, that's exactly what the system will attempt to do, and nothing else.

- **Register-based task**

  In the interest of having a flexible system that can establish duplex (two-way) communication with an SBC, there is an extensive system of registers and state variables. Many functions write to these, and many others read from them to decide what actions to perform. You (the programmer) doesn't have the luxury of specifying the 127 (PWM) or forward (direction) as literal values in this situation.

The Doxygen documentation for each function states its type, so you can easily tell which is which.

Topics/themes

- **PCA9635 PWM Demux**

  To overcome the low number of GPIOs in the ESP32, this project uses the PCA driver to give PWM signals to the motor drivers and their corresponding LEDs. There are methods around initializing the driver, setting it's mode, setting specific PWM values etc. These are all under `pca_driver.h`.

- **Current sense**

The motor drivers have a current sense output that can be read by the ESP32 (with the use of a shunt resistor). The current sense is read by the ADC and converted to a digital value. The initialization, calibration, and reading of the current sense are all under `current_sense.h`.

- **Encoder**

  The motors have encoders that can be read by the ESP32 to provide their angular velocity. The initialization, reading, and resetting of the encoders are all under `encoder.h`.

- **Control of the robot and that of the motors is separate**

  The PCA drivers file `pca_driver.h` is responsible for controlling the motors and LEDs, and tends to deal with the lower-level functions of one motor at a time. The `four-wheel-robot.h` file is responsible for controlling four motors at a time, and includes some functions to instead treat the system as a cohesive robot. Functions in `four-wheel-robot.h` use functions from `pca_driver.h` quite heavily.

- **Structure**

  In general, "direct" functions are used to handle things like initialization and register-based functions are used to handle the actual tasks. Either these functions directly, or your custom functions, are expected to be registered as RTOS tasks using `xTaskCreate()` (built-in to FreeRTOS and ESP-IDF) since they will run perpetually.

I2C Communication with the SBC

This is one of the relatively elaborate parts of the firmware, so it deserves some more attention. In general, the ESP32 acts as a slave to the SBC. The SBC can both issue commands to the ESP32, and request data such as sensor readings and motor status.

It is important you understand how this works, so we'll go over it in detail.

- **The concept of a pseudo register**

  In the world of I2C-based devices, it is common for the slave to have a register. Each register then has a defined memory location (an address) and a clear purpose. For example, setting the value of the register at `0x00` sets the PCA driver mode, telling it what to do with further commands from the master.

  Since there is a wide variety of data we wish to transfer between the ESP32 and the SBC, this project adopts a similar system. Instead of dealing with actual memory addresses in the ESP32, we use a "pseudo register" system. To the SBC, this seems as though the ESP32 has registers like any ol' I2C device. However, in reality, it just places the byte that's supposed to the register address into a `switch() case` structure.

  Many other functions in the firmware behave in that way, so it's important to recognize what the term "register" refers to in this system. Unfortunately, devices like the PCA driver and the IMU do have true registers, so I'll provide a list of all pseudo later in this documentation.

`int-brain-sbc-registers.h` gives the addresses of most of these pseudo registers that the SBC has to deal with. Additionally, addresses are split into sending and receiving addresses, to make it easier to determine what the I2C bus is doing at any given time.

- **I2C buffers**

  Additionally, whenever the master (SBC) initiates a communication, it is expected that the first byte is the pseudo register address. The ESP32 will then read this byte and decide what to do based on it. While parsing, this byte is stored in `_sbc_i2c1_register`.

  Keeping in mind the interrupt-based nature of I2C communication in the ESP32 system [source], two I2C buffers are defined in this system.

  - One of them `_sbc_i2c1_receive_buffer` is the data (such as motor speeds) that should go from the SBC to the ESP32. When the ESP32 has finished reading the data, it is available in this buffer.
  - The other `_sbc_i2c1_send_buffer` is the data (such as encoder positions) that should go from the ESP32 to the SBC. While parsing the pseudo register `_sbc_i2c1_register`, the ESP32 will fill this buffer with the relevant data, and the master can read it whenever it wants. Note that this is a FIFO buffer, so care should be taken that whatever data is written to it is read by the master (and not left to clog up the buffer).

Motor command systems

Controlling a motor has two basic aspects: the direction and the speed, given as forward/backward/stop and a PWM value respectively.

Motor modes allow us to incorporate more functionality that can help in development of more safety features, make it easier to take emergency action, and issue the same speed to all motors at once.

These modes are controlled by a pseudo register, and pairs of bits define each mode.

Safety modes

- **Unsafe mode**

  The motor obeys your commands directly, without any modifications from the firmware. This is the default mode.

- **Protect stall mode**

  A stall is defined either when the motor exceeds a certain current threshold or when the encoder position is not changing, given a PWM value. To protect the motor, it will be stopped if a stall is detected.

- **Protect disconnect mode**

  If the motor is disconnected, the current sense will read zero, given a minimum PWM value. If this is detected, the motor will be stopped. Optionally, that specific motor's LED will blink to let you debug easily.

- **Protect stall and disconnect mode**

  This is a combination of the two modes above. If either a stall or a disconnect is detected, the motor will be stopped.

Speed modes

- **Stop mode:** The motor will stop, regardless of the PWM value (PWM = 0).
- **Max mode:** The motor will run at full speed, regardless of the PWM value (PWM = 255).
- **Command speed mode:** The motor will run at the PWM value you specify for that particular motor.
- **Speed register mode:** All motors will run at the same speed, given by the value in the speed register.

Enable modes

This leverages the OUTPUT ENABLED pin of the PCA driver. If the driver is disabled, all PWM values will be 0, and all motors will stop.

This value for this mode may simply be enabled or disabled.

Motor direction

The DRV8251A driver has the ability to actively brake the motor instead of letting it coast to a stop. This is done by setting the direction to brake. This gives us a total of four possible directions for each motor: idle, forward, backward, and brake.

Motor command functions

Function name	Speed control type	Direction control type	Safety mode	Speed mode	Enable mode
`set_motor_speeds()`	Array for individual speeds	[unused]	Not set-able	Not set-able	Not set-able
`set_common_motor_speed()`	Single speed for all motors	[unused]	Not set-able	Not set-able	Not set-able
`set_motor_directions()`	[unused]	Array for individual directions	Not set-able	Not set-able	Not set-able
`set_bot_direction()`	[unused]	Variable for unified robot direction	Not set-able	Not set-able	Not set-able
`set_motor_mode_register()`	[unused]	[unused]	Set-able by variable	Set-able by variable	Set-able by variable

Function name	Speed control type	Direction control type	Safety mode	Speed mode	Enable mode
`set_motor_safety_mode()`	[unused]	[unused]	Set-able by variable	[unused]	[unused]
`set_motor_speed_mode()`	[unused]	[unused]	[unused]	Set-able by variable	[unused]
`enable_motor_output()`	[unused]	[unused]	[unused]	[unused]	Forced to `ENABLED`
`disable_motor_output()`	[unused]	[unused]	[unused]	[unused]	Forced to `DISABLED`

After calling any of these, you must also call `publish_motor_command()` to actually send the commands to the motors through the PCA driver.

I2C Scheme

Addresses

Host	Slave	Address
ESP32	MPU9250	0x68 or 0x69
ESP32	PCA9635	0x40
Raspberry Pi	ESP32	0x30

Note that in many of the registers below, the address is given as a range. In such cases, each address in range corresponds to a particular motor. This feature allows you to use a shorter I2C transmission if you do not want to address all motors at once.

Raspberry Pi request from ESP32 (send) registers

- `FIRST_SLAVE_REQUEST_ADDRESS = 0x00`
- `LAST_SLAVE_REQUEST_ADDRESS = 0x9F`

Description	"Register"	Data type	Length
Encoder positions	0x10	4 bytes per motor (int)	16 bytes
Encoder position per motor	0x11 - 0x14	4 bytes (int)	4 bytes
Motor speeds	0x15	4 bytes per motor (int)	16 bytes
Motor speed per motor	0x16 - 0x19	4 bytes (int)	4 bytes

Description	"Register"	Data type	Length
Motor current	0x20	4 bytes per motor (int)	16 bytes
Motor current per motor	0x21 - 0x24	4 bytes (int)	16 bytes
Motor stall status	0x30	1 byte total (bool)	1 byte
Motor disconnect status	0x31	1 byte total (bool)	1 byte
IMU data (acceleration)	0x40	4 bytes per axis (int)	12 bytes
IMU data (gyroscope)	0x41	4 bytes per axis (int)	12 bytes
IMU data (magnetometer)	0x42	4 bytes per axis (int)	12 bytes
IMU data (quaternion)	0x43	4 bytes per axis (int)	16 bytes
Battery voltage	0x50	4 bytes (int)	4 bytes

Raspberry Pi command to ESP32 (receive) registers

- `FIRST_SLAVE_COMMAND_ADDRESS = 0xA0`
- `LAST_SLAVE_COMMAND_ADDRESS = 0xFF`

Description	"Register"	Data type	Length
Motor operation mode	0xA0	1 byte total (int)	1 byte
Standard motor data	0xA1	1 byte per motor + 1 byte (int + int)	5 bytes
Standard motor data per motor	0xA2 - 0xA5	2 bytes (int + int)	2 bytes
Motor directions	0xA6	1 byte total (int)	1 byte
Motor direction per motor	0xA7 - 0xAA	1 byte (int)	1 byte
Motor speeds	0xAB	1 byte per motor (int)	4 bytes
Motor speed per motor	0xAC - 0xAF	1 byte (int)	1 byte
Robot direction	0xB0	1 byte total (int)	1 byte
Common speed	0xB1	2 bytes (int)	2 bytes

The "standard motor data" is simply means that the first byte specifies motor direction and next byte(s) specify the motor speed(s). This functionality is made available since it's common to send both these parameters together in practical robots.

Bit breakdowns

[0xB0] Robot direction

Value	Description
0	Front

Value	Description
1	Back
2	Rotate clockwise
3	Rotate counterclockwise
4	Front left
5	Front right
6	Back left
7	Back right

[Common motor breakdown]

The motor stall status, disconnect status, and direction registers all obey this scheme. Two bits are used to specify the value for any one motor, so a single byte is sufficient for all four motors.

Bit	Motor
0, 1	0
2, 3	1
4, 5	2
6, 7	3

[0x30] Motor stall status

Value	Description
00	Not stalled
01	Stalled
10	[UNUSED]
11	[UNUSED]

[0x31] Motor disconnect status

Value	Description
00	Connected
01	Disconnected
10	[UNUSED]
11	[UNUSED]

[0xA6] Motor directions

Value	Description
00	Idle
01	Forward
10	Backward
11	Brake

[0xA0] Motor operation mode

This register does not follow the common motor breakdown scheme. Instead, each pair of bits controls a different aspect of the motor operation mode. The modes set here apply to all motors.

Bits 1, 0: Safety

Value	Description
00	Unsafe
01	Protect stall
10	Protect disconnect
11	Protect stall and disconnect

Bits 3, 2: Speed

Value	Description
00	0 (stop)
01	255 (max)
10	command speed
11	speed register (0xB1)

Bit 5, 4: Enable

Value	Description
00	Disable
01	Enable
10	[UNUSED]
11	[UNUSED]

Bits 6 - 7: Unused