

KUBEN

Felix Strand

Brage Wiseth

KUBEN

2024-12-15

Contents

Introduction	3
First Principles	5
Circuitry	5
Structure and Routing	5
Components	6
Microcontroller	6
Programmer	7
Motorcontroller	8
Driver Gates	9
Power Supply	9
Inertial Measurement Unit	9
Micro Servos	11
OLED Screen	11
Mechanical	13
Overview	13
Casing	13
Mountings	13
Flywheel	13
Brakes	13
Computation	13
Overview	13
Sensor Data	13
Orientation	13
MPC/LDR	13
Software	13
Overview	13
State Machine	13
Computation	13
Testing	13

Introduction

In this project, we aim to design and build a self-balancing cube using reaction wheels, a type of control mechanism commonly used in spacecraft for attitude control. The goal is to create a compact, dynamically stable system capable of maintaining its equilibrium in three-dimensional space without external support. The cube, with its simple geometric structure, presents unique challenges in terms of balance and orientation control, making it an excellent platform for exploring the principles of feedback control systems and mechanical design.

Reaction wheels, which function by spinning at varying speeds to produce torques that counteract disturbances, will serve as the core actuators for balancing the cube. By carefully coordinating the speeds of these wheels, the system will be able to counteract any rotational forces, allowing the cube to maintain stability in any orientation.

This project will involve multiple stages, including the design and fabrication of the cube, the selection and integration of reaction wheels, the development of sensors to track its orientation, and the implementation of a control algorithm that adjusts the wheel speeds in real-time to correct any deviations. The project not only challenges our understanding of dynamic systems and control theory but also has the potential to inform future applications in robotics, aerospace, and precision engineering, where balance and stability are crucial.

Ultimately, the self-balancing cube will serve as a proof-of-concept for demonstrating the potential of reaction wheels in balancing and controlling small-scale systems, offering a hands-on opportunity to delve into the fascinating intersection of mechanical engineering, robotics, and control theory.

Contents

Introduction	3
First Principles	5
Circuitry	5
Structure and Routing	5
Components	6
Microcontroller	6
Programmer	7
Motorcontroller	8
Driver Gates	9
Power Supply	9
Inertial Measurement Unit	9
Micro Servos	11
OLED Screen	11
Mechanical	13
Overview	13
Casing	13
Mountings	13
Flywheel	13
Brakes	13
Computation	13
Overview	13
Sensor Data	13
Orientation	13
MPC/LDR	13
Software	13
Overview	13
State Machine	13
Computation	13
Testing	13

First Principles

How can we balance with just a set of spinning wheels that don't even touch the ground? Well the same way you and I would balance on a slack line or on a fence post, We pivot our arms around our torso kinda like how a plane would turn its wings when banking. How this makes us balance is not at all obvious, but it has something to do with angular momentum and torques, considering that we are rotating our body. Another hint is that by extending our arms further out or even using a rod to artificially extend our arms we can greatly enhance our ability to balance. By doing so we create torque, we can use this exact technique with flywheels only instead of it being two arms we essentially have infinite arms stacked in a circle.

Circuitry

The internal logic and the physical output of our system needs to be joined together by circuitry; a way to translate our calculations for balance onto the motors themselves. Our circuitry has three main responsibilities; input -> calculate -> output. Our input will be in form of sensor data indicating our cube's current angle, and a wanted state (do we want to balance? on which edge? so on). Our calculations will be about transforming our current input angles into a quantitative action for the motors to perform. The output will in turn be the adjustment of our motors to achieve balance. High level, this will require accelerometers and gyroscopes as our input, a microcontroller to perform the calculations, and motors to be adjusted for output. All this combined can be reasonably packed into a few circuits that contain a few sensors, a microcontroller, and motor driver capabilities.

This section of our documentation aims to provide context and reasoning behind our main circuit choices, as well as give a general overview of all components, their role in Cubitron specifically, and how they are interconnected to achieve a cohesive circuit with results.

Structure and Routing

The circuitry will consist of three different PCBs. Two PCBs are mounted on opposite corners of the cube, containing an Inertial Measurement Unit each. The last circuit will be mounted in the center of the cube, containing our main logic, motors, battery charger, driver gates; essentially the rest of our circuitry. Three BLDC motors that put motion to our flywheels are connected to our main circuit. Additionally, three micro-servos are positioned for braking mechanism for each flywheel. Cables between the main circuit, BLDC motors and sensor-PCBs will be of type PFC. They are a thin, compact and convenient snap-on cable to work with. They came standard with our chosen motors, and influenced the choice for the sensor PCBs. The micro-servo are cabled with individual insulated wires bundled together. It is important to be specific with wire lengths, as loose wires may disturb moving parts of the whole robot.

Components

To gain context of our implementation, this subsection will outline every integrated circuit we use, what their general purpose in a project is, and more specifically what their purpose is for Cubitron.

Microcontroller

Our microcontroller will be the ATSAME53J18A-MF, which has a 32-bit ARM Cortex-4 processor. Our main reasons for this microcontroller is the ceiling for calculations and peripherals. We expect to make reasonably heavy calculations in the context of an embedded system, which the 32 bit system with an in built Floating Point Unit is excellent for. Additionally, we need to connect two sensors and six motors, which makes the pool of peripherals we need to support quite large. We also have personal experience with the usage of these microcontrollers, with similar problem statements, and were satisfied with the physical capabilities of our microcontroller.

The microcontroller has the most central and complicated role of the whole circuitry. It will be the component responsible of combing the three main stages of our program runtime. The pipeline of Input -> Calculate -> Output has the microcontroller as a major component in all of them. Our sensors will both be sending their data through an I2C connection to our microcontroller. They will have an assigned interrupt pin each, which opens for having our sensor data be as close to real time as possible. From this sensor data, we need to calculate our current position in space, with regard to our 3D orientation, and calculate again how far this is from our intended position, and what motor actions we need to perform to get there. Lastly, our output stage will consist of sending that data to our TMC controllers, and adjusting our motors physically.

The TMC4671, which will be detailed later, needs a 25MHz clock signal to operate. This microcontroller has a Generic Clock Controller, which can be configured to generate a steady 25MHz clock signal to each of the TMC4671 ICs.

As you can imagine, our microcontroller is hooked up to a of external components in our circuitry, which is quite nice to have an overview over. Below is a table of the names of all external connections in the perspective of our microcontroller schematics, the endpoint they are connected to, the pin and pin type it is connected to, and a short description of its purpose:

Name	Endpoint	Pin	Pin Type	Function
SWDIO	Programmer	PA31	Digital	Dataline for programming of our microcontroller
SWCLK	Programmer	PA30	Digital	Clock signal for programming
ERR_LED	LED	PA12	Digital	LED indicating system error
PROCESS_LED	LED	PA13	Digital	LED indicating processing

Name	Endpoint	Pin	Pin Type	Function
IMU1 INT	IMU1	PA05	Digital	IMU1 data rdy interrupt
IMU1 SCL	IMU1	PA06	Digital	IMU1 Serial Clock for data transfer
IMU1 SDA	IMU1	PA07	Digital	IMU1 Serial data line
IMU2 INT	IMU2	PA23	Digital	IMU2 data rdy interrupt
IMU2 SCL	IMU2	PA24	Digital	IMU2 Serial Clock for data transfer
IMU2 SDA	IMU2	PA25	Digital	IMU2 Serial data line
OLED_SDA	OLED	PA08	Digital	OLED screen Serial Data Line
OLED_SCL	OLED	PA09	Digital	OLED screen Serial CLK
S1_SIG	Servo 1	PA17	Digital	Brake Servo 1 control signal
S2_SIG	Servo 2	PA18	Digital	Brake Servo 2 control signal
S3_SIG	Servo 3	PA19	Digital	Brake Servo 3 control signal
DB_TX	DB_HEADER	PA22	Digital	UART TX for debug purposes
DB_RX	DB_HEADER	PA22	Digital	UART RX for debug purposes

Programmer

We actually have two individual microcontrollers on this board, but the second one is intended for programming our main microcontroller. If you looked at the pin table for the ATSAME53J18A-MF, we have two entries with the endpoint “Programmer”, which refers to this component. The programmer equips a 32-bit ARM Cortex M0+ processor, and was chosen because of the close relation to the microcontroller above, and being able to act as a bridge for programming and debugging the Cortex-M4, with the firmware CMSIS-DAP. This explains the pins between the programmer and our main Cortex-M4.

CMSIS-DAP (Cortex Microcontroller Interface Standard) is an ARM standard for the DAP (Debug Access Port) that provides an USB interface to allow debugging and programming ARM Cortex-M microcontrollers. Our Cortex-M0 will have an USB-C interface through the physical connector of our microcontroller, which will communicate with the device we are flashing from (probably our computers). Furthermore, we can use DAP functionality for debugging and flashing our controller. We can program the flash memory through our USB interface, debug (stop execution, set breakpoints, view memory, etc...) our programs.

Read more about CMSIS-DAP:

https://arm-software.github.io/CMSIS_5/DAP/html/index.html

Name	Endpoint	Pin	Pin Type	Function
SWDIO	Microcontroller	PA30	Digital	Dataline for programming of our microcontroller
SWCLK	Microcontroller	PA31	Digital	Clock signal for programming
D+	Microcontroller	PA25	Digital	USB Interface
D-	Microcontroller	PA24	Digital	USB Interface

Motorcontroller

The TMC4671 allows for control of a BLDC motor, and will be tripled up for the control of three BLDC motors in total. The benefit of this is being allowed to control the motors completely separated, with no interference between the three. We will take use of supported field operated control (FOC), which allows for very precise and energy efficient moving of the motors.

The TMC471 communicates with our MCU using the SPI protocol. SPI fits our system well, as it allows our master (MCU) to have several slaves (motors) on fewer lines.

The TMC471 needs extensive input to ensure a precise output. It takes in hall sensor data directly from its opposing BLDC motor, and reads current sensors to regulate torque. Each of these signals are

We need to ensure our Digital Encoder/Hall Sensor signals, which usually operates on 5V are regulated down to 3.3V, which the 4671 asks for. As of right now, the digital encoders may be seen as redundant. Both the digital encoders and hall sensors are intended to provide the TMC4671 data about how our motors are positioned. Because of our motor choice having in-built hall sensors, that will be the simpler approach. Adding digital encoders in this implementation can be seen as overkill. We can add digital encoders on a second iteration if the extra resolution is needed.

We have disconnected the BRAKE pin. The pin is not intended for braking the motors, but for dissipating excess energy provided for the motors voltage feedback. Outside of scope for now.

REF_R, REF_H, REF_L are pins intended to be driven HIGH if we can tell if the motors are approaching the physical limits of motion. This is overkill for our project, as there are no real scenarios wherein we would deem this necessary. Our motors only control a flywheel clockwise or counter-clockwise. We wont ever be in a position where this is no more degrees we can spin in a given direction physically.

The ENI (Enable Input)/ENO (Enable Output) pins are two separate pins that co-operate. Enable Input essentially acts like a kill switch for the motor control in general. If we reach a bad state, we can drive ENI low, and the motors will chill out. ENO is an output pin, which just reflects the state of ENI, and is intended for more self-reliant embedded systems. Again, we control everything with SPI. Therefore we will have the ENI pin as a killswitch, while having ENO disconnected.

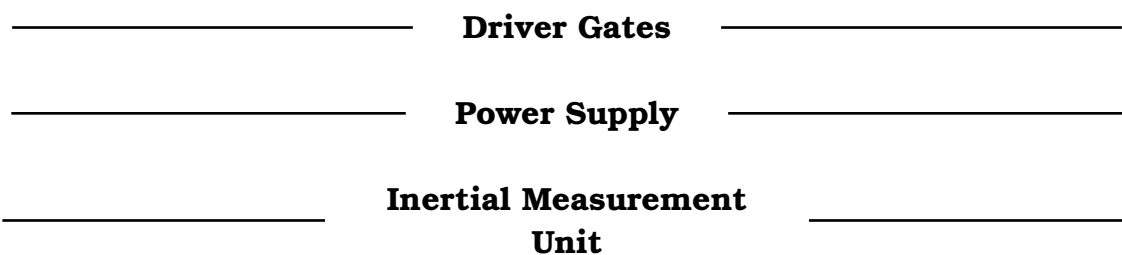
The STEP and DIR pins are not needed, as we will be communicating to the TMC4671 through SPI. STEP and DIR are for external motion controllers, like an FPGA steering the motors.

Lastly, we have a lot of PWM signals that we need to decide if we need or not. Let us go through the unneeded ones first:

PWM_I is PWM input for steering the motor. We use SPI, and therefore do not need this connected. PWM_IDLE_H / PWM_IDLE_L are for setting idle states. We don't need any specific idle states, because our cube will be flat on the ground when not in use, which does not require a specific motor position/speed/torque etc. Therefore this will not be connected.

Both of the above are PWM inputs in perspective of the TMC4671. The rest of the PWM signals are PWM outputs, which will be going to our driver gates. These are essential for motor operation, and is part of the core functionality of the controller. Therefore we route these outputs to the driver gates, which will in return drive the motors.

Lastly, we need to decide on whether we want debugging through SPI or UART.



We will in total have three separate circuits, wherein two of them will be dedicated to host a single IMU (Inertial Measurement Unit) each. The rationale for this is the positions we will mount the sensors, to acquire the information about our orientation in order to stabilize the system at a given angle. One IMU will be mounted on the corner which will be facing the ground during operation, and the other IMU will be on the corner straight above the corner facing the ground, so they form a straight line. We can then deduce the orientation and angle of the cube, in such a way that we know our cube is balancing straight if $\theta = \frac{\pi}{2}$ with respect to the ground, and $\Delta\theta \approx 0$.

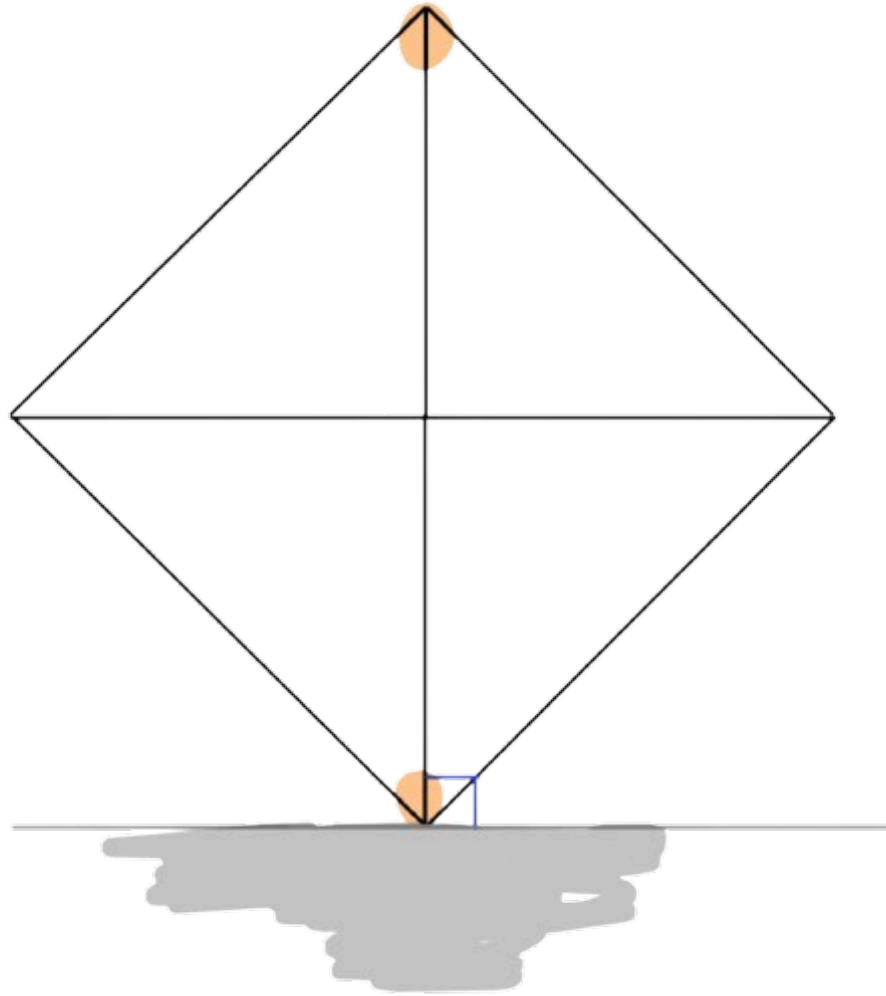


Figure 1: Orange indicates IMU positions. Blue highlights the angle of our IMU-line with respect to the ground.

The specific IC we will move forward is the ISM330DHC, it hoists two separate sensors; a 3D gyroscope and 3D accelerometer. The sensors are high accuracy and high performance, comfortable to use and reliable. The team has had previous experience with the specific IMU, and have written drivers for similar purposes that can be partially reused.

It communicates with either I2C or SPI, wherein we decided I2C is best for us. We want to make the internals as cable and hassle free as possible. I2C lets us only need to route five signals through a PFC cable, which will decrease the size of the female end of the connectors, the PCBs, and the cable itself. We are sacrificing a pin on the main microcontroller to route one Serialized Data signal to each microcontroller, with no I2C switch. The respective are instead both pulled high and communicated with on respective lines.

Because the IMU itself has such few pins, we can include the whole table and the intended pin connections:

Name	Endpoint	Pin	Pin Type	Function
VDD	3.3V	VDD	Power	Power supply
VDDIO	3.3V	VDDI	Power	Power supply
GND	Ground	GND	Power	Ground the circuit
SDA	Microcontroller	SDA	Digital	Serial Data to MCU
SDO/SA0	3.3V	SDO	LSB	Decides device ID
SCL	Microcontroller	PA24	Digital	USB Interface
CS	3.3V	CS	Digital	Must be set high to enable I2C
INT1	Microcontroller	INT1	Digital	Interrupt for data RDY
INT2	X	X	X	Disconnected
SDx	X	X	X	Disconnected
SCx	X	X	X	Disconnected
OCS_AUX	X	X	X	Disconnected
SDO_AUX	X	X	X	Disconnected

Micro Servos

The braking system requires one micro-servo for each flywheel to be able to stop their momentum as close to instant as possible. The micro-servos are a part of a larger mechanical brake functioning closely to the way a bike-brake works. From a circuitry perspective, the braking system is quite simple, and only requires pins on the MCU for control, and headers to connect the motors themselves to.

Name	Endpoint	Pin	Pin Type	Function
VDD	5V	VDD	Power	For Power
GND	Ground	GND	For Power	
Signal	MCU	Signal Pin	Steer the controller	

OLED Screen

Our OLED screen is added for personalization and options for communicating interesting internal states. The GROVE OLED SSD1306 is a small OLED display with a 64x48 resolution. It is just a fun addition to our circuit, and requires only

I2C pins for communication. We will be using the U8G2 library for displaying and communicating easily through I2C.

Name	Endpoint	Pin	Pin Type	Function
VDD	3.3V	VDD	Power	For Power
GND	Ground	GND	For Power	
SCL	MCU	SCL	Signal	Serial Clock for I2C
SDA	MCU	SDA	Signal	Serial data for I2C

_____	Mechanical	_____
_____	Overview	_____
_____	Casing	_____
_____	Mountings	_____
_____	Flywheel	_____
_____	Brakes	_____
_____	Computation	_____
_____	Overview	_____
_____	Sensor Data	_____
_____	Orientation	_____
_____	MPC/LDR	_____
_____	Software	_____
_____	Overview	_____
_____	State Machine	_____
_____	Computation	_____
_____	Testing	_____