```
import random
import json
import boto3

# This program simulates a parking lot where cars with 7-digit license plates can park in random spots.
# The ParkingLot class initializes a parking lot of a given size with a specified parking spot size.
# The Car class represents a car with a license plate and methods to park the car in the parking lot.
# The main function simulates the process of parking a list of cars in random spots until the parking lot is full.

class ParkingLot:
    def __init__(self, size_in_sqft, spot_length=8, spot_width=12):
        """
        Initialize the parking lot with a given size and spot dimensions.
        Calculates the number of spots based on the size and spot dimensions.
        """
        self.spot_size = spot_length * spot_width
        self.num_spots = size_in_sqft // self.spot_size
        self.spots = [None] * self.num_spots

        if self.spot_size > size_in_sqft:
            raise ValueError("spot_size cannot be more than size_in_sqft.")

    def is_full(self):
        """
        Check if the parking lot is full.
        Returns True if there are no empty spots, False otherwise.
        """
        for spot in self.spots:
            if spot is None:
                return False
        return True

    def find_random_empty_spot(self):
        """
        Find a random empty spot in the parking lot.
        Returns the index of an empty spot, or None if the lot is full.
        """
        empty_spots = [i for i, spot in enumerate(self.spots) if spot is None]
        return random.choice(empty_spots) if empty_spots else None

    def map_vehicles_to_spots(self):
        """
        Create a JSON object mapping vehicles to their parked spots.
        Returns the JSON object.
        """
        mapping = {}
```

```
        for i, spot in enumerate(self.spots):
            if spot is not None:
                mapping[str(i)] = str(spot)  # Using string representation of spot number and license plate
        return mapping

def save_to_s3(json_data, bucket_name, file_name):
    """
    Save JSON data to a file and upload it to an S3 bucket.
    """
    with open(file_name, 'w') as f:
        json.dump(json_data, f)

    s3 = boto3.client('s3')
    s3.upload_file(file_name, bucket_name, file_name)

class Car:
    def __init__(self, license_plate):
        """
        Initialize the car with a given license plate.
        Raises a ValueError if the license plate is not a 7 digit alphanumeric string.
        """
        if len(license_plate) != 7 or not license_plate.isalnum():
            raise ValueError("License plate must be a 7 digit alphanumeric string.")
        self.license_plate = license_plate

    def __str__(self):
        """
        Return the license plate as the string representation of the car.
        """
        return self.license_plate

    def park(self, parking_lot, spot_number):
        """
        Attempt to park the car in the given spot number of the parking lot.
        Returns a tuple (success, message) indicating whether the parking was successful and a message.
        """
        if parking_lot.spots[spot_number] is None:
            parking_lot.spots[spot_number] = self
            return True, f"Car with license plate {self.license_plate} parked successfully in spot {spot_number}."
        else:
            return False, f"Spot {spot_number} is already occupied."

def main(cars, parking_lot, s3_bucket_name, file_name):
    """
    Simulate parking each car in the list of cars into random spots in the parking lot.
    Continues until all cars are parked or the parking lot is full.
```

```
    At the end, save the mapping of vehicles to spots in a JSON file and upload it to an S3
bucket.
    """
    for car in cars:
        if parking_lot.is_full():
            print("Parking lot is full. Exiting program.")
            break

        while True:
            spot_number = parking_lot.find_random_empty_spot()
            if spot_number is None:
                print("Parking lot is full. Exiting program.")
                break
            success, message = car.park(parking_lot, spot_number)
            print(message)
            if success:
                break

    # Create JSON mapping and save it to a file
    mapping = parking_lot.map_vehicles_to_spots()
    save_to_s3(mapping, s3_bucket_name, file_name)
    print(f"Vehicle to spot mapping saved to {file_name} and uploaded to S3 bucket
{s3_bucket_name}.")

if __name__ == "__main__":
    # Example usage: Create a parking lot and a list of cars, then try to park them.
    parking_lot = ParkingLot(size_in_sqft=2000, spot_length=10, spot_width=12)


cars=[Car("".join(random.choices("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789",
k=7))) for _ in range(parking_lot.total_spots)]

    s3_bucket_name = "your-s3-bucket-name"
    file_name = "vehicle_spot_mapping.json"

    main(cars, parking_lot, s3_bucket_name, file_name)
```

Test Cases

Positive Test Cases:

Valid Parking and Mapping:
Scenario: Park cars in the parking lot and verify the vehicle-to-spot mapping.
Steps:
Initialize a parking lot and cars.
Park cars in specific spots.

Call map_vehicles_to_spots and verify the generated mapping matches the expected JSON-like dictionary.
Expected Outcome: The mapping should accurately reflect which cars are parked in which spots.

```
def test_positive_valid_parking_and_mapping():
    parking_lot = ParkingLot(size_in_sqft=480, spot_length=8, spot_width=12)
    cars = [Car("ABC1234"), Car("XYZ5678"), Car("LMN3456")]

    # Park cars in specific spots
    parking_lot.spots[0] = cars[0]  # Car "ABC1234" in spot 0
    parking_lot.spots[2] = cars[1]  # Car "XYZ5678" in spot 2
    parking_lot.spots[4] = cars[2]  # Car "LMN3456" in spot 4

    # Expected mapping
    expected_mapping = {
        "0": "ABC1234",
        "2": "XYZ5678",
        "4": "LMN3456"
    }

    # Get actual mapping
    actual_mapping = parking_lot.map_vehicles_to_spots()

    # Assert that actual mapping matches expected mapping
    assert actual_mapping == expected_mapping
```

Negative Test Cases

1. Parking Lot Full:

Scenario: Attempt to park more cars than the parking lot can accommodate.
Steps:
Initialize a parking lot with a small number of spots.
Try to park more cars than there are spots.
Verify that no additional cars can be parked once the lot is full.
Expected Outcome: Cars should not be able to park once the parking lot is full.

```
def test_negative_parking_lot_full():
    parking_lot = ParkingLot(size_in_sqft=96, spot_length=8, spot_width=12)  # Only 1 spot available
    car1 = Car("ABC1234")
    car2 = Car("XYZ5678")

    # Park the first car
```

```
    parking_lot.spots[0] = car1

    # Attempt to park the second car
    success, message = car2.park(parking_lot, 0)

    # Assert that parking is not successful
    assert not success
    assert message == "Spot 0 is already occupied."
```

---

2. Invalid License Plate:

Scenario: Attempt to create a car with an invalid license plate (not 7 characters or not alphanumeric).
Steps:
Initialize a car with an invalid license plate.
Verify that a ValueError is raised during initialization.
Expected Outcome: Initialization should fail with a ValueError due to an invalid license plate.

```
def test_negative_invalid_license_plate():
    try:
        car = Car("ABC12345")  # Invalid: More than 7 characters
        assert False  # Should not reach here
    except ValueError as e:
        assert str(e) == "License plate must be a 7 digit alphanumeric string."

    try:
        car = Car("ABC 123")  # Invalid: Contains spaces
        assert False  # Should not reach here
    except ValueError as e:
        assert str(e) == "License plate must be a 7 digit alphanumeric string."
```