Lab Units 5 & 6 Develop a Digital Twin for Conceptualization of a Robotic Cell

- Creating a Simple Robotic Cell Using RobotStudio
- Introduction to RAPID Programming for Robotic Control
- Development and Configuration of Smart Components in RobotStudio
- Connecting ABB Robots to the Cloud for Data Collection
- Virtual Commissioning and Analysis of Robotic Cells

Objective

To provide hands-on experience in using RobotStudio, RAPID programming language, and Smart Components for virtual commissioning of robotic cells. Students will also learn how to connect ABB robots to the cloud and collect real-time data.

Class Work (2 hours)

1. **Creating a Simple Robotic Cell** (30 minutes)
    - **Objective**: Set up a simple robotic work cell within RobotStudio.
    - **Activity**:
        - Create a new project and import a standard ABB robot model.
        - Add a basic workpiece and define the workspace.
        - Simulate basic robot movements and analyze workspace interactions.
2. **Introduction to RAPID Programming** (40 minutes)
    - **Objective**: Understand and write simple RAPID programs.
    - **Activity**:
        - Overview of RAPID syntax and structure.
        - Write a basic program to control the robotic arm for pick-and-place operations.
        - Run the program in the simulation and make real-time adjustments.
3. **Creating Smart Components in RobotStudio** (30 minutes)
    - **Objective**: Learn how to create and configure Smart Components for enhanced simulations.
    - **Activity**:
        - Define Smart Components and their applications.
        - Create simple Smart Components (e.g., sensors or conveyors) and link them to the RAPID program.
        - Test component functionality in the robotic cell.

Individual Work (10 hours)

1. **Further RAPID Programming** (3 hours)
    - **Objective**: Develop more complex RAPID programs.
    - **Activity**:
        - Create a more detailed RAPID program for a specific task (e.g., welding, sorting, or assembly).
        - Incorporate error handling and optimization techniques.
        - Validate the program in simulation, ensuring efficiency.
2. **Advanced Smart Components** (2 hours)
    - **Objective**: Create and configure advanced Smart Components in RobotStudio.
    - **Activity**:
        - Develop custom Smart Components that interact with the robot in more complex ways.
        - Example: Create a conveyor belt with sensors to stop and start the robot based on the workpiece position.
3. **Connecting to the Cloud and Data Collection** (3 hours)
    - **Objective**: Connect the robotic cell to a real ABB robot and collect data.
    - **Activity**:
        - Learn how to connect the ABB robot to the cloud using Python, nodeRED, and Siemens PLC technologies.
        - Collect real-time data from the robot's sensors (e.g., position, torque) and send it to a cloud platform for monitoring.
        - Use the collected data to improve the virtual commissioning setup.
4. **Virtual Commissioning and Analysis** (2 hours)
    - **Objective**: Finalize the virtual commissioning process and analyze results.
    - **Activity**:
        - Refine the robotic cell simulation based on collected data.
        - Test and simulate scenarios, making final adjustments to the RAPID programs and Smart Components.
        - Document findings and prepare a summary report on the virtual commissioning process.

Expected Outcomes:

- Create a functional digital twin of a robotic cell using RobotStudio.
- Write and test RAPID programs for controlling ABB robots.
- Design and implement Smart Components for enhanced automation.

- Connect the robotic cell to the cloud, and collect and analyze real-time data from a real ABB robot.
- Develop insights from virtual commissioning and refine processes based on performance data.

++++++++++++++++++++++++

Tutorial: Creating and Using Smart Components in RobotStudio

Prerequisites:

- Install **RobotStudio** (latest version)
- Familiarity with the **RobotStudio interface** and basic simulation

Step 1: Set Up the Project

1. **Open RobotStudio**:
    - Create a new station and import your robot (e.g., an ABB robot model).
    - Add a simple workpiece (e.g., a box or pallet) into the workspace.
2. **Enable Smart Components**:
    - Navigate to the **"Home"** tab, and in the "Component" section, select **"Smart Components"** to access this feature.
    - You can see a list of pre-built Smart Components (e.g., sensors, clamps, grippers, and conveyors) that you can customize.

Step 2: Create a Custom Smart Component

1. **Open the Smart Components Wizard**:
    - Go to the **"Smart Components"** tab and select **"New Smart Component"**. This will start the wizard for creating custom components.
2. **Define Component Behavior**:
    - **Triggers**: Set what will trigger your Smart Component (e.g., a signal, sensor detection, or a specific condition like an object's presence).
    - **Actions**: Define what the Smart Component should do when triggered. For example, a conveyor can move, a gripper can close, or a signal can be sent to another component.
3. **Add Connections**:
    - Smart Components often need to interact with other components. In the wizard, you can connect the Smart Component to your robot or other objects in the environment (e.g., workpieces or sensors).

Step 3: Example – Creating a Conveyor System with a Sensor

Objective: Create a conveyor belt that moves workpieces when a sensor detects their presence.

1. **Add Conveyor**:

    - Go to the **"Component Library"**, select a **Conveyor** (you can use a predefined one).
    - Position it in your station so that the robot can interact with it.

2. **Add a Sensor**:
    - In the **Smart Components Library**, choose a **Proximity Sensor** and place it along the conveyor belt.
    - This sensor will detect the workpiece on the conveyor.
3. **Configure Sensor Trigger**:
    - Open the sensor's properties and set the trigger to detect objects within a certain range. This trigger will activate the conveyor belt when the workpiece is detected.
4. **Create Action for the Conveyor**:
    - In the Smart Component Wizard, link the sensor trigger to an **Action** that starts the conveyor movement (e.g., setting a motor to drive the conveyor belt).
5. **Test the Setup**:
    - In the simulation, place a workpiece on the conveyor.
    - As the sensor detects the workpiece, the conveyor should start moving it.

Step 4: Linking Smart Components to a RAPID Program

1. **Set up a RAPID Program**:
    - Go to the **RAPID** programming tab and create a simple program for the robot, such as picking a workpiece from the conveyor.
2. **Trigger the RAPID Program with a Smart Component**:
    - In the Smart Components Wizard, you can set the **sensor's trigger** to send a signal to the robot. When the workpiece is detected, the robot can execute the RAPID program to pick or interact with the workpiece.
3. **Simulation**:
    - Run the simulation to test the behavior.
    - The sensor should detect the workpiece, the conveyor will move it, and the robot will pick it based on the RAPID program.

Step 5: Advanced Smart Component Features

1. **Custom Parameters**:
    - You can add **custom parameters** to your Smart Components, such as speed, timing, or control logic. This allows for more granular control over how components behave.
2. **Linking Multiple Components**:

- Multiple Smart Components can interact with each other. For example, a series of conveyors can work together to transport a product through different stages of a robotic operation.

3. **Using Signals**:
    - Smart Components can generate and listen to **signals**. These signals can create complex logic and interactions between components and robots.

Step 6: Saving and Exporting Smart Components

1. **Save the Smart Component**:
    - Once the Smart Component is configured, save it as part of your project for reuse.
    - You can also export the Smart Component for other projects.
2. **Export Station**:
    - If needed, export the entire station as a project file, which includes your robotic cell, Smart Components, and RAPID programs.

++++++++++++++++++

Connecting an ABB Robot to the Cloud Using Python, Node-RED, and Siemens PLC

Connecting an ABB robot to the cloud enables real-time data monitoring, remote control, and advanced analytics, enhancing the capabilities of your digital twin. This tutorial will guide you through the process of integrating an ABB robot with cloud services using **Python**, **Node-RED**, and a **Siemens PLC**. By the end of this guide, you will be able to collect data from the robot, process it, and send it to the cloud for further analysis.

Prerequisites

1. **Hardware and Software Requirements:**
    - **ABB Robot** with RIO (Robot Interface for OPC UA) or equivalent communication module.
    - **Siemens PLC** (e.g., Siemens S7-1200/1500 series).
    - **Computer** with **RobotStudio** installed.
    - **Internet Connection** for cloud access.
    - **Python** installed on your computer.
    - **Node-RED** installed (can run locally or on a server).
    - **Cloud Platform Account** (e.g., AWS IoT, Azure IoT Hub, or IBM Cloud).
2. **Software Libraries and Tools:**
    - **python-telegram-bot** or **opcua** library for Python.
    - **node-red-contrib-siemens-s7** or equivalent Node-RED nodes for Siemens PLC communication.

    - **MQTT** broker setup (often integrated within cloud platforms).

3. **Basic Knowledge:**
    - Familiarity with ABB RAPID programming.
    - Basic understanding of Python programming.
    - Understanding of PLC programming and networking.
    - Basic knowledge of Node-RED workflows.

Step 1: Setting Up Communication Between ABB Robot and Siemens PLC

1.1 Configure the Siemens PLC:

- **Connect PLC to Network:**
    - Ensure your Siemens PLC is connected to the same local network as your ABB robot and the computer running RobotStudio.
- **Configure PLC Communication:**
    - Use **TIA Portal** to set up the PLC's IP address and ensure it is reachable.
    - Create data blocks (DB) in the PLC for storing robot data (e.g., position, status).

1.2 Configure ABB Robot Communication:

- **Set Up Robot Communication:**
    - In **RobotStudio**, navigate to **"Controller"** > **"Communication"** settings.
    - Enable **OPC UA** or **TCP/IP** communication based on your setup.
    - Assign an IP address to the robot's communication module.
- **Create RAPID Program to Send Data to PLC:**
    - Write a RAPID program that collects necessary data (e.g., joint angles, task status) and sends it to the Siemens PLC via the configured communication protocol.

```
PROC SendDataToPLC()

    ! Example: Send joint positions to PLC

    VAR num joint1 := CRobT(\ToolData).joint1;

    VAR num joint2 := CRobT(\ToolData).joint2;

    ! Add code to send these variables to PLC via TCP/IP or OPC UA

ENDPROC
```

- **Trigger Data Transmission:**
    - Integrate the SendDataToPLC procedure within your robot's operational cycle, ensuring data is sent periodically or upon specific events.

Step 2: Collecting Data from Siemens PLC Using Node-RED

2.1 Install and Configure Node-RED:

- **Installation:**

    - Install Node-RED on your computer or a dedicated server. You can follow the official installation guide.
- **Start Node-RED:**
    - Launch Node-RED and access the interface via http://localhost:1880.

2.2 Install Siemens PLC Nodes:

- **Add Siemens Nodes:**
    - In Node-RED, go to **"Manage Palette"** > **"Install"** and search for node-red-contrib-s7.
    - Install the **node-red-contrib-s7** package to enable communication with Siemens PLCs.

2.3 Create a Node-RED Flow to Read Data from PLC:

1. **Add S7 Read Node:**
    - Drag the **"s7 in"** node onto the workspace.
    - Configure the node with your PLC's IP address, rack, and slot numbers.
2. **Define Data Points:**
    - Specify the data blocks (DB) and addresses from which to read data (e.g., DB1.DBD0 for joint1).
3. **Process Data:**
    - Use **function nodes** to format or process the incoming data as needed.
4. **Send Data to Cloud:**
    - Add an **MQTT out node** or use a specific cloud node (e.g., AWS IoT, Azure) to publish the data to your cloud platform.

Example Flow:

Function Node Example:

```
// Function to format PLC data
msg.payload = {
    joint1: msg.payload[0], // Assuming DB1.DBD0 is joint1
    joint2: msg.payload[1]
};
return msg;
```

Step 3: Processing Data with Python

3.1 Set Up Python Environment:

- **Install Required Libraries:**

```
pip install paho-mqtt opcua
```

3.2 Connect to MQTT Broker:

- **Python Script to Subscribe to MQTT Topics:**

```
import paho.mqtt.client as mqtt
import json
# Define MQTT settings
MQTT_BROKER = "your_mqtt_broker_address"
MQTT_PORT = 1883
MQTT_TOPIC = "abb_robot/data"
# Callback when a message is received
def on_message(client, userdata, msg):
    try:
        data = json.loads(msg.payload.decode())
        print(f"Received data: {data}")
        # Add processing logic here (e.g., store in database, trigger alerts)
    except json.JSONDecodeError:
        print("Failed to decode JSON")
# Set up MQTT client
client = mqtt.Client()
client.on_message = on_message
client.connect(MQTT_BROKER, MQTT_PORT, 60)
client.subscribe(MQTT_TOPIC)
# Start the loop
client.loop_forever()
```

3.3 Advanced Data Processing:

- **Real-Time Analytics:**
    - Implement real-time analytics, such as anomaly detection or predictive maintenance, using Python libraries like **pandas**, **numpy**, or **scikit-learn**.
- **Storing Data:**
    - Integrate with databases (e.g., InfluxDB for time-series data, MongoDB) to store the incoming data for historical analysis.

Step 4: Integrating Cloud Services

4.1 Choose a Cloud Platform:

- **Popular Options:**
    - **AWS IoT Core**
    - **Microsoft Azure IoT Hub**
    - **IBM Watson IoT**
    - **Google Cloud IoT**
- **Example: AWS IoT Core Setup**

4.2 Configure Cloud IoT Services:

1. **Create an IoT Thing:**
    - In AWS IoT Core, create a new IoT Thing representing your ABB robot.
2. **Set Up MQTT Topics:**
    - Define MQTT topics (e.g., abb_robot/data) to which Node-RED will publish data.
3. **Secure Communication:**
    - Generate and download security certificates.
    - Configure your MQTT client (Node-RED) to use these certificates for secure communication.

4.3 Modify Node-RED Flow for Cloud Integration:

- **Use Cloud-Specific Nodes:**
    - Install and configure nodes specific to your chosen cloud platform (e.g., node-red-contrib-aws-iot).
- **Configure MQTT Out Node:**
    - Update the MQTT out node with cloud broker details, including security settings.

Example: Publishing to AWS IoT Core

// Configure MQTT out node with AWS IoT endpoint and certificates

Step 5: Visualizing and Monitoring Data in the Cloud

5.1 Set Up Cloud Data Visualization:

- **AWS Example:**
    - Use **AWS IoT Analytics** or **Amazon QuickSight** to visualize incoming data.
    - Create dashboards to monitor robot performance, detect anomalies, and analyze operational efficiency.

5.2 Create Dashboards:

1. **Define Metrics:**
    - Choose key metrics to display, such as joint positions, task completion times, and error rates.
2. **Design Visual Elements:**
    - Use charts, graphs, and gauges to represent data intuitively.
3. **Set Up Alerts:**
    - Configure alerts for specific conditions (e.g., unexpected robot behavior, system failures).

Example: AWS QuickSight Dashboard

- **Connect QuickSight to AWS IoT Analytics or your data store.**
- **Create visualizations:**
    - Line charts for joint angles over time.
    - Bar charts for task completion rates.
    - Heatmaps for identifying high-error areas.

Step 6: Enhancing Connectivity and Automation with Node-RED

6.1 Implement Control Commands from Cloud to Robot:

- **Bidirectional Communication:**
    - Allow cloud-based commands to control the robot, such as starting/stopping tasks or adjusting parameters.
- **Node-RED Flow for Commands:**

Function Node Example:

```
// Example: Receive a command to start the robot
if (msg.payload.command === "start_robot") {
    // Format RAPID command or PLC write action
    msg.payload = "START";
    return msg;
}
return null;
```

6.2 Secure the Communication:

- **Authentication and Authorization:**
    - Ensure that only authorized users or systems can send commands to the robot.
- **Use Encrypted Channels:**
    - Utilize TLS/SSL for all data transmissions to protect against interception.

Step 7: Testing and Validation

7.1 Simulate Robot Operations:

- **Run Simulations in RobotStudio:**
    - Test your RAPID programs and Smart Components to ensure they behave as expected.
- **Validate Data Flow:**
    - Check that data is correctly sent from the robot to the PLC, processed by Node-RED, and uploaded to the cloud.

7.2 Monitor System Performance:

- **Check Latency:**

    - Ensure that data is transmitted in real-time with minimal delays.
- **Verify Data Accuracy:**
    - Cross-check the data collected in the cloud with actual robot operations to ensure accuracy.

7.3 Troubleshoot Issues:

- **Common Issues:**
    - Network connectivity problems.
    - Incorrect PLC or robot configurations.
    - Security certificate errors.
- **Debugging Steps:**
    - Use logs in Node-RED and Python scripts to identify where data transmission is failing.
    - Verify all IP addresses, ports, and credentials.
    - Ensure that firewall settings allow necessary traffic.

Step 8: Documentation and Reporting

8.1 Document the Setup:

- **Architecture Diagram:**
    - Create a diagram illustrating the communication flow between the ABB robot, Siemens PLC, Node-RED, Python scripts, and the cloud platform.
- **Configuration Details:**
    - Record all IP addresses, port numbers, and credentials used in the setup.

8.2 Prepare a Summary Report:

- **Include:**
    - Objectives and scope of the project.
    - Step-by-step configuration and integration process.
    - Challenges faced and solutions implemented.
    - Screenshots of Node-RED flows, Python scripts, and cloud dashboards.
    - Future improvements and potential enhancements.

+++++++++++++

To transfer data from a RAPID server running on an ABB robot to a Python client, and then to Google Drive as a file, we can break the process down into several steps.

Step 1: Set Up Communication Between RAPID Server and Python Client

1.1 Set Up RAPID Server on ABB Robot

The ABB robot's RAPID program will need to send data via a TCP/IP connection. Below is an example of how to set up a simple RAPID program to send data to a Python client.

Example RAPID Code:

```
VAR socketdev client_socket;

VAR string message;

VAR num joint1, joint2;

PROC SendDataToPython()

    ! Open a TCP connection to the Python client

    Open client_socket\InetAddr:="192.168.1.100",\Port:=1025;

    ! Check if the connection is established

    IF IsConnected(client_socket) THEN

        joint1 := CRobT(\ToolData).joint1;

        joint2 := CRobT(\ToolData).joint2;

        ! Create a message containing joint data

        message := "joint1:" + NumToStr(joint1, 3) + ", joint2:" + NumToStr(joint2, 3);

        ! Send the data to the Python client

        Write client_socket, message;

        ! Close the connection after sending the message

        Close client_socket;

    ENDIF

ENDPROC
```

This program opens a TCP/IP socket connection to the Python client, sends data (joint1 and joint2), and then closes the connection.

Replace 192.168.1.100 with the IP address of the machine running the Python client.

Use port 1025 or any other available port on the client machine.

1.2 Set Up Python Client to Receive Data

You can use Python's socket library to set up a TCP client that receives data from the ABB RAPID server.

Python Code to Receive Data:

```
import socket

# Set up the server IP and port (must match the RAPID program)

HOST = '192.168.1.100'  # Replace with your local IP address

PORT = 1025

def receive_data_from_robot():

    # Create a socket and bind it to the host and port

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:

        s.bind((HOST, PORT))

        s.listen()

        print(f"Listening on {HOST}:{PORT}...")
        # Accept the connection from the robot
        conn, addr = s.accept()
        with conn:
            print(f"Connected by {addr}")
            data = conn.recv(1024).decode()
            # Print the data received from the robot
            print(f"Received data: {data}")
            # Save data to a file locally
            with open("robot_data.txt", "w") as file:
                file.write(data)
            return "robot_data.txt"
# Test the function to receive data
local_file = receive_data_from_robot()
```

This code creates a TCP server that listens on the specified HOST and PORT.

When data is received, it writes the data to a text file named robot_data.txt.

Step 2: Upload the File to Google Drive

To upload the file to Google Drive, we can use the **Google Drive API**. You will first need to set up Google Drive API access in your Python script.

2.1 Set Up Google Drive API

1. **Enable Google Drive API**:
    - Go to the Google Cloud Console.
    - Create a new project or use an existing one.
    - Enable the **Google Drive API** for your project.
    - Create **OAuth 2.0 credentials** for the API.
    - Download the credentials.json file and store it in the same directory as your Python script.
2. **Install Required Libraries**:
    - Install the necessary libraries using pip:

```
pip install --upgrade google-api-python-client google-auth-httplib2 google-auth-oauthlib
```

2.2 Python Code to Upload the File to Google Drive

Python Script to Upload the File:

```
from googleapiclient.discovery import build
from googleapiclient.http import MediaFileUpload
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
import os
```

```
# Scopes for Google Drive API
SCOPES = ['https://www.googleapis.com/auth/drive.file']
def authenticate_google_drive():
    creds = None
    # Load credentials from the 'token.json' file, if it exists
    if os.path.exists('token.json'):
        creds = Credentials.from_authorized_user_file('token.json', SCOPES)
    # If no valid credentials, prompt user to log in
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file('credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        # Save credentials for future use
        with open('token.json', 'w') as token:
            token.write(creds.to_json())
    return creds
def upload_file_to_google_drive(file_path):
    # Authenticate and build the Google Drive service
    creds = authenticate_google_drive()
    service = build('drive', 'v3', credentials=creds)
    # Define the file metadata and upload file to Google Drive
    file_metadata = {'name': os.path.basename(file_path)}
    media = MediaFileUpload(file_path, mimetype='text/plain')
    # Upload file and return the file ID
    file = service.files().create(body=file_metadata, media_body=media, fields='id').execute()
    print(f"File uploaded to Google Drive with ID: {file.get('id')}")
# Upload the local file created by the Python client
upload_file_to_google_drive(local_file)
```

The function authenticate_google_drive() handles the authentication with Google Drive.

The function upload_file_to_google_drive() uploads the local file (e.g., robot_data.txt) to Google Drive.

Step 3: Running the Full Process

1. **Ensure the RAPID program is running** on the ABB robot, which will send data via TCP/IP to the Python client.
2. **Run the Python script** to listen for the incoming data from the ABB robot, save it locally, and upload it to Google Drive.

+++++++++++++

Automating with Task Scheduler (Windows)

For Windows, you can use **Task Scheduler** to run the Python script automatically at specific intervals.

Steps:

1. **Open Task Scheduler**:
   - Press Win + R, type taskschd.msc, and press Enter.
2. **Create a New Task**:
   - Click on **Create Basic Task** in the Task Scheduler interface.
3. **Set a Name and Trigger**:
   - Name your task (e.g., "Upload ABB Data").
   - Choose a trigger based on your requirement (e.g., "Daily", "Weekly", or "When a specific event is logged").
4. **Set the Action to Start a Program**:
   - Select **Start a Program** as the action.
   - Browse to the location of your Python executable (e.g., python.exe) in the **Program/script** field.
   - In the **Add arguments** field, add the path to your Python script (e.g., C:\path\to\your_script.py).
5. **Set the Schedule**:
   - Specify when and how often you want the task to run (e.g., every 1 hour or every day at a specific time).
6. **Save the Task**:
   - Save the task, and the system will automatically run the Python script according to the schedule you set.

+++++++++++

Automate Upload Based on New File Event

You can use a Python library called **watchdog** to monitor a directory for changes (e.g., when a new file is created by the RAPID server). Whenever a new file is detected, the Python script will automatically upload it to Google Drive.

Steps:

1. **Install watchdog**:

```
pip install watchdog
```

Python Script for Monitoring Directory:

```
import time

import os

from watchdog.observers import Observer

from watchdog.events import FileSystemEventHandler
from googleapiclient.discovery import build
from googleapiclient.http import MediaFileUpload
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request
# Google Drive authentication function
def authenticate_google_drive():
    SCOPES = ['https://www.googleapis.com/auth/drive.file']
    creds = None
    if os.path.exists('token.json'):
        creds = Credentials.from_authorized_user_file('token.json', SCOPES)
    if not creds or not creds.valid:
        if creds and creds.expired and creds.refresh_token:
            creds.refresh(Request())
        else:
            flow = InstalledAppFlow.from_client_secrets_file('credentials.json', SCOPES)
            creds = flow.run_local_server(port=0)
        with open('token.json', 'w') as token:
            token.write(creds.to_json())
    return creds
# Function to upload file to Google Drive
def upload_file_to_google_drive(file_path):
    creds = authenticate_google_drive()
    service = build('drive', 'v3', credentials=creds)
    file_metadata = {'name': os.path.basename(file_path)}
    media = MediaFileUpload(file_path, mimetype='text/plain')
    file = service.files().create(body=file_metadata, media_body=media, fields='id').execute()
    print(f"File uploaded to Google Drive with ID: {file.get('id')}")
# Event handler to monitor the directory
class Watcher(FileSystemEventHandler):
    def on_created(self, event):
        if not event.is_directory:
            print(f"New file detected: {event.src_path}")
            upload_file_to_google_drive(event.src_path)
# Main function to start the observer
if __name__ == "__main__":
    path = "/path/to/your/directory"  # Directory to monitor
    event_handler = Watcher()
    observer = Observer()
    observer.schedule(event_handler, path, recursive=False)
    observer.start()
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()
```

This script monitors a directory for new files using the watchdog library.

When a new file is detected (e.g., a data file generated by your robot), it automatically triggers the upload process to Google Drive.

++++++++++

Automating the Whole Workflow Using Node-RED (Optional)

If you're already using **Node-RED** to collect data, you can automate the upload to Google Drive as part of your Node-RED workflow.

1. **Install the Google Drive Node**: In the Node-RED palette, install the **node-red-contrib-google-drive** package to integrate with Google Drive.
2. **Create a Node-RED Flow**:
   - Use a combination of **file nodes** (to write the data locally) and **Google Drive nodes** (to upload files automatically).
   - You can set up a trigger whenever new data is received from the robot, and then the data is uploaded to Google Drive in real-time.

++++++++++++++

You can use **MQTT** in this scenario for automating data transfer from the RAPID server to Python and then further to Google Drive.

Overview of Using MQTT:

1. **Robot with RAPID server sends data** to an MQTT broker via an MQTT client.
2. **Python client subscribes** to the MQTT broker, receives data in real-time, and processes it.
3. The Python script can then **upload the data to Google Drive**.

Step 1: Set Up an MQTT Broker

- You need an **MQTT broker** that handles the messages between the RAPID server and the Python client. You can use a local MQTT broker like **Mosquitto**, or a cloud-based broker like **HiveMQ**, **AWS IoT**, or **Google Cloud IoT**.

Option 1: Install Mosquitto Locally

If you're running this locally, you can install **Mosquitto** as the MQTT broker.

Windows: Download and install from the Mosquitto website.

Start the broker:

```
mosquito
```

Option 2: Use Cloud-Based MQTT Broker

If you want to use a cloud-based broker like HiveMQ or AWS IoT, you need to configure the broker with appropriate credentials and topics.

Step 2: Modify RAPID Program to Publish Data via MQTT

For your RAPID server on the ABB robot, you will need an MQTT client to publish data to the MQTT broker. ABB robots might not have a native MQTT client in RAPID, but you can use **Node-RED** on an intermediary device (like a Raspberry Pi or a PC) to convert RAPID messages to MQTT.

1. **Configure Node-RED to Listen to Robot Data**:
   Use TCP/IP or OPC UA in **Node-RED** to receive data from the RAPID program.

2. **Convert and Publish Data**: After receiving the data from RAPID, use Node-RED's **MQTT output node** to publish it to your MQTT broker.

RAPID Example (TCP/IP to Node-RED):

```
PROC SendDataToNodeRED()

    VAR socketdev client_socket;

    VAR string message;

    VAR num joint1, joint2;

    ! Open a TCP connection to Node-RED

    Open client_socket\InetAddr:="192.168.1.100",\Port:=1025;

    ! Check if connection is established

    IF IsConnected(client_socket) THEN

        joint1 := CRobT(\ToolData).joint1;

        joint2 := CRobT(\ToolData).joint2;

        ! Create a message containing joint data

        message := "joint1:" + NumToStr(joint1, 3) + ", joint2:" + NumToStr(joint2, 3);

        ! Send data to Node-RED

        Write client_socket, message;

        ! Close connection after sending the message

        Close client_socket;

    ENDIF

ENDPROC
```

In **Node-RED**, add a TCP node to listen for this data, and then connect it to an MQTT output node to publish the data to your MQTT broker.

Step 3: Set Up the Python MQTT Client to Receive Data

Now, configure a Python script to act as an **MQTT subscriber**. This script will listen for messages from the MQTT broker, save the data into a file, and then upload the file to Google Drive.

Python Script to Receive Data and Upload to Google Drive:

1. **Install the required libraries**:

```
pip install paho-mqtt google-api-python-client google-auth-httplib2 google-auth-oauthlib
```

Python Code:

```python
import paho.mqtt.client as mqtt

import os

from googleapiclient.discovery import build

from googleapiclient.http import MediaFileUpload

from google.oauth2.credentials import Credentials

from google_auth_oauthlib.flow import InstalledAppFlow

from google.auth.transport.requests import Request

# MQTT settings

MQTT_BROKER = "localhost"  # Use the IP or hostname of your broker

MQTT_PORT = 1883

MQTT_TOPIC = "abb_robot/data"

# Google Drive authentication

SCOPES = ['https://www.googleapis.com/auth/drive.file']

def authenticate_google_drive():

    creds = None

    if os.path.exists('token.json'):

        creds = Credentials.from_authorized_user_file('token.json', SCOPES)

    if not creds or not creds.valid:

        if creds and creds.expired and creds.refresh_token:

            creds.refresh(Request())

        else:

            flow = InstalledAppFlow.from_client_secrets_file('credentials.json', SCOPES)

            creds = flow.run_local_server(port=0)

        with open('token.json', 'w') as token:

            token.write(creds.to_json())

    return creds

def upload_file_to_google_drive(file_path):

    creds = authenticate_google_drive()

    service = build('drive', 'v3', credentials=creds)

    file_metadata = {'name': os.path.basename(file_path)}

    media = MediaFileUpload(file_path, mimetype='text/plain')

    file = service.files().create(body=file_metadata, media_body=media, fields='id').execute()

    print(f"File uploaded to Google Drive with ID: {file.get('id')}")

# MQTT on_message callback

def on_message(client, userdata, msg):

    data = msg.payload.decode()

    print(f"Received message: {data}")

    # Save the data to a local file

    with open("robot_data.txt", "w") as file:

        file.write(data)
    # Automatically upload the file to Google Drive
    upload_file_to_google_drive("robot_data.txt")
# MQTT client setup
client = mqtt.Client()
client.on_message = on_message
client.connect(MQTT_BROKER, MQTT_PORT, 60)
client.subscribe(MQTT_TOPIC)
# Start the loop to listen for incoming messages
client.loop_forever()
```

The script connects to the MQTT broker and subscribes to the topic abb_robot/data.

Whenever data is published to this topic by the robot (via Node-RED), the Python script receives the message and writes it to a local file (robot_data.txt).

After writing the data, it automatically uploads the file to **Google Drive** using the Google Drive API.

Step 4: Testing the Setup

1. **Start the MQTT Broker** (Mosquitto or cloud-based).
2. **Ensure the RAPID program is running** and sending data to Node-RED.
3. **Configure Node-RED to publish data to MQTT** (use the same topic as the Python client subscribes to).
4. **Run the Python script**. It should start listening for data.
5. When Node-RED publishes data to the MQTT topic, the Python client should receive it, save it to a file, and then upload that file to Google Drive.

Automation and Customization:

- **Automate the process**: You can run the Python script in the background using **cron** (Linux/macOS) or **Task Scheduler** (Windows).
- **Security**: Use MQTT over **TLS/SSL** for secure communication, especially when using cloud-based brokers.
- **Data format**: You can structure the data in **JSON** format for easier parsing, modification, and storage.

Example JSON Data:

```
{
    "joint1": 45.0,
    "joint2": 30.2
}
```

This setup gives you a **real-time, event-driven solution** where data is transferred from your ABB robot to the MQTT broker, then captured by a Python client and automatically uploaded to Google Drive.

Advantages of Using MQTT:

1. **Scalability**: MQTT allows multiple clients (subscribers) to listen to the same topic, enabling scalable systems.
2. **Efficiency**: MQTT is a lightweight protocol, which makes it ideal for scenarios where bandwidth or processing power is limited.
3. **Real-time Data Transfer**: The MQTT client in Python will receive the data as soon as it's published, making it suitable for real-time applications.