

Instalación de Micro-Ros en ESP32

Seguimiento del ejemplo 3.1 Ping-Pong Application with micro_ros_setup del libro Robot Operating System (ROS) de T. Anis Koubaa 7ma Edición

Prerrequisitos:

- Tener una computadora con Ubuntu 22.04 Jammy Jellyfish o posterior.
- Tener un ESP32 con un programador
- Tener conexión a internet
- Tener instalado el Visual Studio Code

Procedimiento:

1. Instalación de ROS 2 Humble Hawksbill en Ubuntu

Primero debemos de asegurarnos de que nuestro S.O. tenga una locación UTF-8 compatible. Para ello abriremos una terminal y ejecutaremos el siguiente comando:

```
locale # check for UTF-8
```

Luego, debemos de agregar el gestor de paquetes de ROS 2 a nuestras fuentes de paquetes. Para ello debemos asegurarnos de que el Repositorio Universal de Ubuntu (Repositorio de software mantenido por la comunidad) esté habilitado con los siguientes comandos:

```
sudo apt install software-properties-common  
sudo add-apt-repository universe
```

Luego agregamos el índice de ROS 2 GPG (GNU Privacy Guard) con apt para poder instalar los paquetes del repositorio de ROS de forma segura. Para ello debemos de asegurarnos de tener curl instalado:

```
sudo apt update && sudo apt install curl -y  
sudo curl -sSL  
https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o  
/usr/share/keyrings/ros-archive-keyring.gpg
```

Luego, se agrega el repositorio a la lista de fuentes:

```
echo "deb [arch=$(dpkg --print-architecture) signed-  
by=/usr/share/keyrings/ros-archive-keyring.gpg]  
http://packages.ros.org/ros2/ubuntu $(. /etc/os-release &&  
echo $UBUNTU_CODENAME) main" | sudo tee  
/etc/apt/sources.list.d/ros2.list > /dev/null
```

Posteriormente, actualizamos nuestro gestor de paquetes:

```
sudo apt update  
sudo apt upgrade
```

E instalamos los paquetes de ROS 2 Humble Hawksbill de escritorio (Opcional), sus librerías y herramientas de desarrollo. Para nuestra instalación solo son necesarias las librerías base:

```
sudo apt install ros-humble-ros-base  
sudo apt install ros-dev-tools
```

Por último inicializamos nuestro entorno de desarrollo:

```
source /opt/ros/humble/setup.bash
```

2. Crear área de trabajo para micro ROS

Ejecutamos los siguientes comandos para crear el área de trabajo:

```
mkdir microros_ws  
cd microros_ws  
git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro\_ros\_setup.git src/micro_ros_setup  
sudo apt update && rosdep update  
sudo rosdep init  
rosdep update  
rosdep install --from-path src --ignore-src -y  
sudo apt-get install python3-pip  
sudo apt install python3.10-venv  
colcon build  
source install/local_setup.bash
```

Una vez creada el área de trabajo procederemos a crear el firmware que nos permitirá crear nuestro agente microROS:

```
ros2 run micro_ros_setup create_agent_ws.sh  
ros2 run micro_ros_setup build_agent.sh  
source install/local_setup.bash
```

Una vez instalado el paquete de agente microRos, podremos levantarlo en una consola usando el siguiente comando:

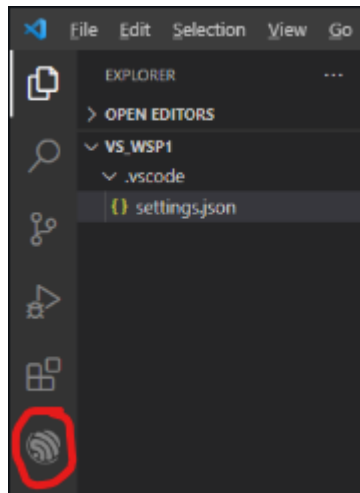
```
ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888 -v6
```

El cual nos debe dejar levantado el proceso de agente micro Ros para su posterior conexión con los nodos microRos. Estos nodos se crearán en las siguiente secciones.

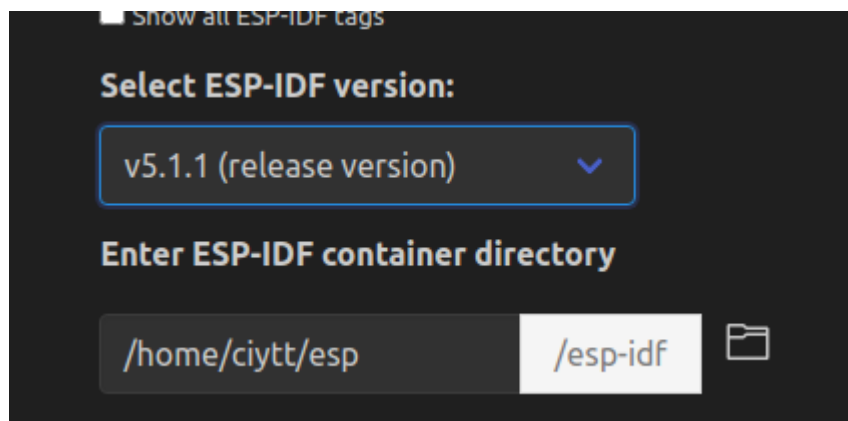
3. Instalar extensión Espressif

Abrimos Visual Studio Code e Instalamos la extensión Espressif IDF: [Espressif IDF - Visual Studio Marketplace](#).

Una vez instalada haremos click al ícono que se habrá creado en la barra de navegación del Visual Studio Code:



Esto nos abrirá una ventana para poder instalar todas las librerías necesarias para construir y subir nuestros programas a nuestro ESP32. En esta ventana seleccionamos como versión de ESP-IDF la última versión (Para el día de la fecha es la versión 5.1.1) y dejamos el directorio de instalación por defecto



Por último, le damos a aceptar y esperamos a que finalice la instalación.

4. Agregar microRos como componente de esp-idf

Primero debemos de situarnos dentro de la carpeta donde instalamos la extensión es espressif, usualmente:

```
cd ~/esp/esp-idf
```

Luego, nos situamos en la carpeta de componentes:

```
cd components
```

Y descargamos el componente de microRos para esp-idf:

```
git clone https://github.com/micro-ROS/micro_ros_espidf_component.git
```

5. Cargar microRos en ESP32

Ahora debemos de situarnos en alguno de los ejemplos del repositorio recientemente clonado (En este caso en int32_publisher)

```
cd micro_ros_espidf_component
. ~/esp/esp-idf/export.sh
cd examples/int32_publisher
pip3 install catkin_pkg lark-parser empy colcon-common-extensions
```

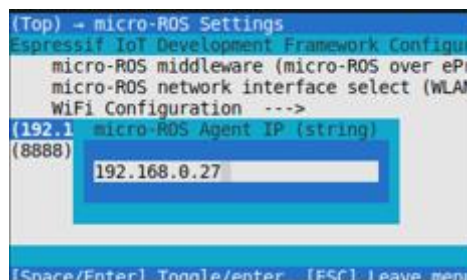
Configuramos el target para esp32

```
idf.py set-target esp32
```

Abrimos la configuración de la aplicación:

```
idf.py menuconfig
```

Dentro de este menú debemos de acceder a la sección micro-Ros configuration, luego a WiFi Configuration, donde pondremos las credenciales de la red a la cual se conectará nuestro ESP32. Luego volvemos a micro-Ros configuration para configurar la dirección ip y el puerto de nuestro agente ROS. Para ello pondremos la ip de nuestra computadora con Ubuntu y dejamos el puerto por defecto (8888).



Luego, salimos de la configuración guardando los cambios.

Ahora debemos de buildear nuestro ejemplo:

```
idf.py build
```

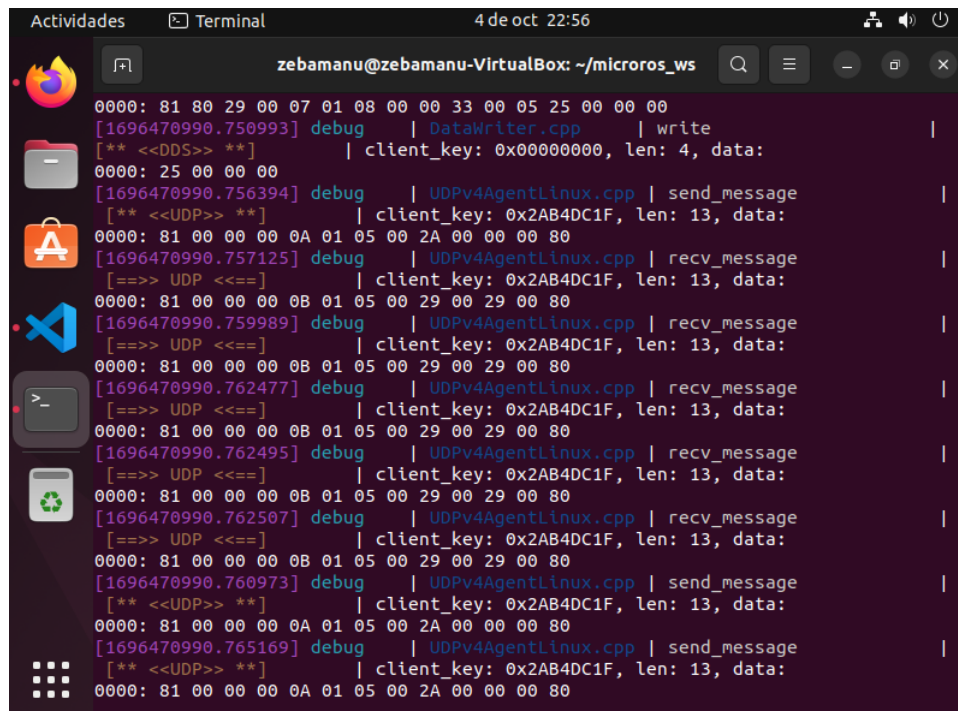
Una vez buildeado, conectamos nuestra esp32 a la computadora por USB, y luego flasheamos el MCU especificando el puerto al cual está conectado

```
idf.py -p PORT [-b BAUD] flash
```

Una vez subido el código al ESP32, podremos monitorear el correcto funcionamiento usando

```
idf.py monitor
```

Cabe destacar que para la correcta conexión del nodo a la red, debemos de asegurarnos de que nuestro agente microRos debe estar levantado en la ip y puertos previamente especificados. Si es este el caso deberíamos de ver una salida en terminal similar a la siguiente:



The screenshot shows a terminal window titled "Terminal" with the user "zebamanu@zebamanu-VirtualBox" and the directory "~/microros_ws". The terminal displays a series of debug logs from ROS. The logs show a sequence of messages being sent and received over UDP. The messages are hexadecimal strings, and the logs indicate the client key, length, and data for each message. The logs are as follows:

```
0000: 81 00 29 00 07 01 08 00 00 33 00 05 25 00 00 00
[1696470990.750993] debug | DataWriter.cpp | write
[** <<DDS>> **] | client_key: 0x00000000, len: 4, data:
0000: 25 00 00 00
[1696470990.756394] debug | UDPv4AgentLinux.cpp | send_message
[** <<UDP>> **] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 2A 00 00 00 80
[1696470990.757125] debug | UDPv4AgentLinux.cpp | recv_message
[==>> UDP <<==] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 29 00 29 00 80
[1696470990.759989] debug | UDPv4AgentLinux.cpp | recv_message
[==>> UDP <<==] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 29 00 29 00 80
[1696470990.762477] debug | UDPv4AgentLinux.cpp | recv_message
[==>> UDP <<==] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 29 00 29 00 80
[1696470990.762495] debug | UDPv4AgentLinux.cpp | recv_message
[==>> UDP <<==] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 29 00 29 00 80
[1696470990.762507] debug | UDPv4AgentLinux.cpp | recv_message
[==>> UDP <<==] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 29 00 29 00 80
[1696470990.760973] debug | UDPv4AgentLinux.cpp | send_message
[** <<UDP>> **] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 2A 00 00 00 80
[1696470990.765169] debug | UDPv4AgentLinux.cpp | send_message
[** <<UDP>> **] | client_key: 0x2AB4DC1F, len: 13, data:
0000: 81 00 00 00 0A 01 05 00 2A 00 00 00 80
```

En la cual nuestro nodo nos envía periódicamente un valor entero. Ahora ya estamos listos para cargar nuestros propios programas usando la arquitectura de micro-Ros.