Android Developers

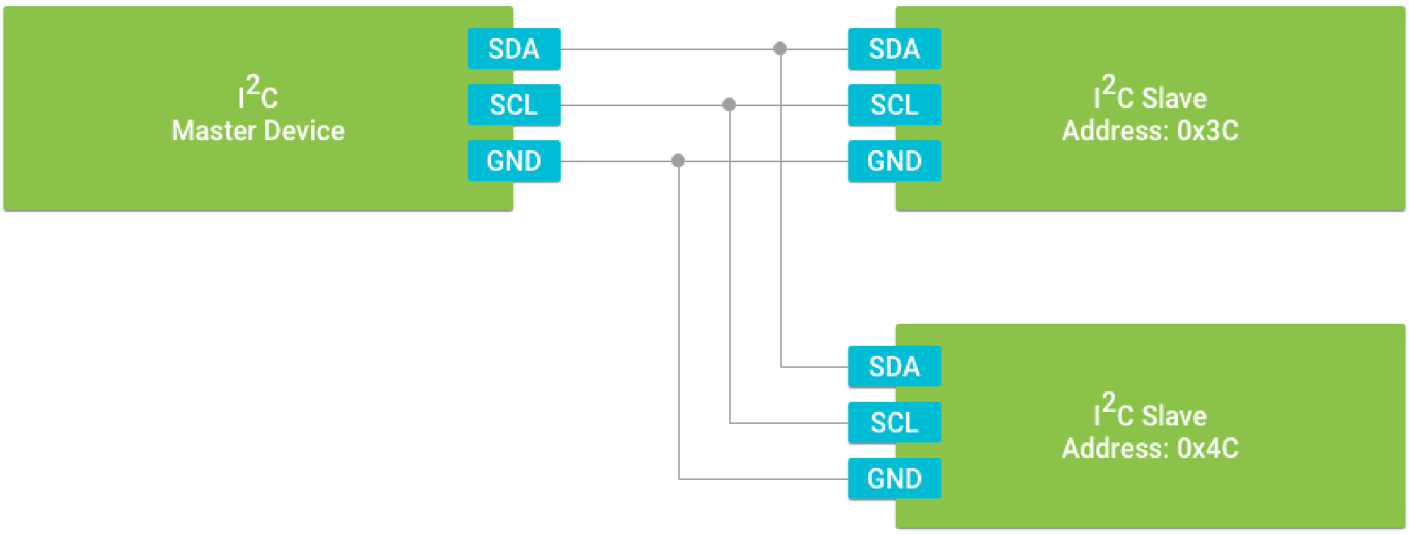
I2C

The Inter-Integrated Circuit (https://en.wikipedia.org/wiki/I%C2%B2C) (IIC or I^2C) bus connects simple peripheral devices with small data payloads. Sensors and actuators are common use cases for I^2C. Examples include accelerometers, thermometers, LCD displays, and motor drivers.

I^2C is a ***synchronous*** serial interface, which means it relies on a shared clock signal to synchronize data transfer between devices. The device in control of triggering the clock signal is known as the ***master***. All other connected peripherals are known as ***slaves***. Each device is connected to the same set of data signals to form a ***bus***.

I^2C devices connect using a 3-Wire interface consisting of:

- Shared clock signal (SCL)
- Shared data line (SDA)
- Common ground reference (GND)



Because all data is transferred over one wire, I^2C only supports ***half-duplex*** communication. All communication is initiated by the master device, and the slave must respond once the master's transmission is complete.

I^2C supports multiple slave devices connected along the same bus. Unlike SPI (https://developer.android.com/things/sdk/pio/spi.html), slave devices are addressed using the I^2C software protocol. Each device is programmed with a unique address and only responds to transmissions the master sends to that address. Every slave device must have an address, even if the bus contains only a single slave.

Managing the slave device connection

device names from `PeripheralManagerService` using `getI2cBusList()`:

```
PeripheralManagerService manager = new PeripheralManagerService();
List<String> deviceList = manager.getI2cBusList();
if (deviceList.isEmpty()) {
    Log.i(TAG, "No I2C bus available on this device.");
} else {
    Log.i(TAG, "List of available devices: " + deviceList);
}
```

Once you know the target device name, use `PeripheralManagerService` to connect to that device. When you are done communicating with the peripheral device, close the connection to free up resources. Additionally, you cannot open a new connection to the device until the existing connection is closed. To close the connection, use the device's `close()` method.

```
public class HomeActivity extends Activity {
    // I2C Device Name
    private static final String I2C_DEVICE_NAME = ...;
    // I2C Slave Address
    private static final int I2C_ADDRESS = ...;

    private I2cDevice mDevice;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Attempt to access the I2C device
        try {
            PeripheralManagerService manager = new PeripheralManagerService();
            mDevice = manager.openI2cDevice(I2C_DEVICE_NAME, I2C_ADDRESS);
        } catch (IOException e) {
            Log.w(TAG, "Unable to access I2C device", e);
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();

        if (mDevice != null) {
            try {
                mDevice.close();
                mDevice = null;
            } catch (IOException e) {
                Log.w(TAG, "Unable to close I2C device", e);
            }
        }
    }
```

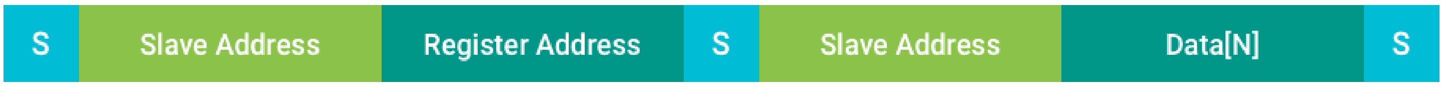
> **Note:** The device name represents the I^{2}C bus, and the address represents the individual slave on that bus. Therefore, an `I2cDevice` is a connection to a specific slave device on the corresponding I^{2}C bus.

Interacting with registers

I^{2}C slave devices organize their contents into either readable or writable registers (individual bytes of data referenced by an address value):

- Readable registers - Contains data the slave wants to report to the master, such as sensor values or status flags.
- Writable registers - Contains configuration data that the master can control.

A common protocol implementation known as System Management Bus (https://en.wikipedia.org/wiki/System_Management_Bus) (SMBus) exists on top of I^{2}C to interact with register data in a standard way. SMBus commands consist of two I^{2}C transactions as follows:



The first transaction identifies the register address to access, and the second reads or writes the data at that address. Logical data on a slave device may often take up multiple bytes, and thus encompass multiple register addresses. The register address provided to the API is always the first register to reference.

> **Note:** Per SMBus protocol, the device will send a "repeated start" condition between the address and data transactions.

Peripheral I/O provides three types of SMBus commands for accessing register data:

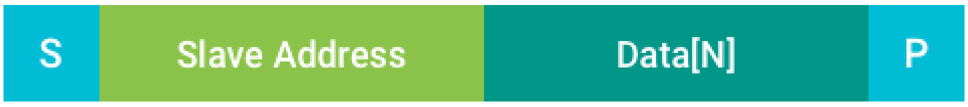
- **Byte Data** - `readRegByte()` and `writeRegByte()` Read or write a single 8-bit register value.
- **Word Data** - `readRegWord()` and `writeRegWord()` Read or write two consecutive register values as a 16-bit little-endian word. The first register address corresponds to the least significant byte (LSB) in the word, followed by the most significant byte (MSB).
- **Block Data** - `readRegBuffer()` and `writeRegBuffer()` Read or write up to 32 consecutive register values as an array.

```
// Modify the contents of a single register
public void setRegisterFlag(I2cDevice device, int address) throws IOException {
    // Read one register from slave
    byte value = device.readRegByte(address);
    // Set bit 6
    value |= 0x40;
    // Write the updated value back to slave
    device.writeRegByte(address, value);
}
```

```
    byte[] data = new byte[3];
    device.readRegBuffer(startAddress, data, data.length);
    return data;
}
```

Transferring raw data

When interacting with an I^{2}C peripheral that defines its registers differently than SMBus -- or perhaps doesn't use registers at all -- use the raw `read()` and `write()` methods for full control over the data bytes transmitted across the wire. These methods will execute a single I^{2}C transaction as follows:



With raw transfers, the device will send a single start condition before the transfer and a single stop condition after. It is not possible to combine multiple transactions with a "repeated start" condition.

> **Note:** There is no explicit maximum length that a raw transaction can handle, but the I^{2}C controller hardware on your device may have a limit on the number of bytes it can process. Consult your device hardware documentation if your peripheral requires large data transfers.

The following code sample show you how to construct a raw byte buffer and write it to an I^{2}C slave:

```
public void writeBuffer(I2cDevice device, byte[] buffer) throws IOException {
    int count = device.write(buffer, buffer.length);
    Log.d(TAG, "Wrote " + count + " bytes over I2C.");
}
```