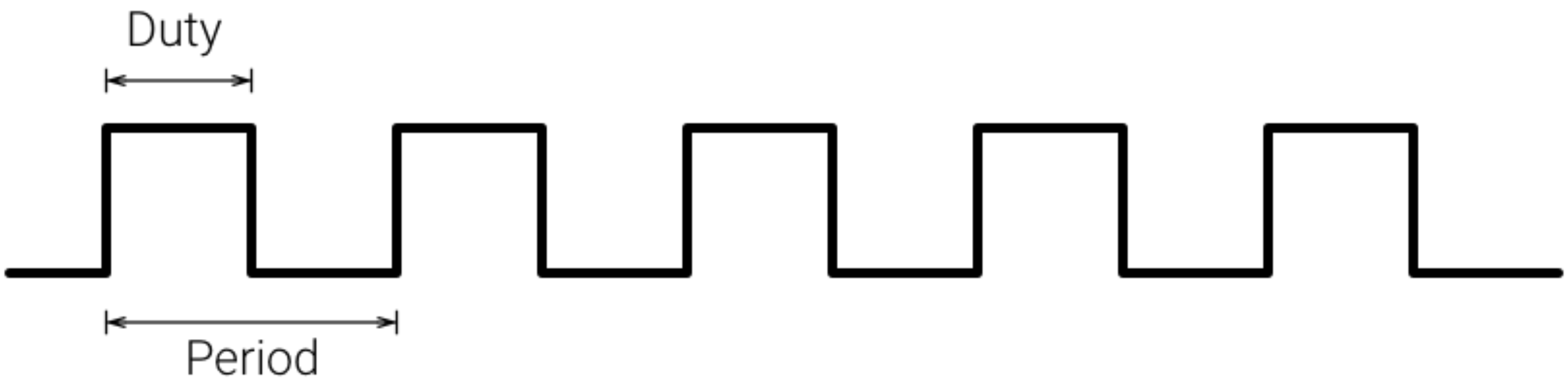
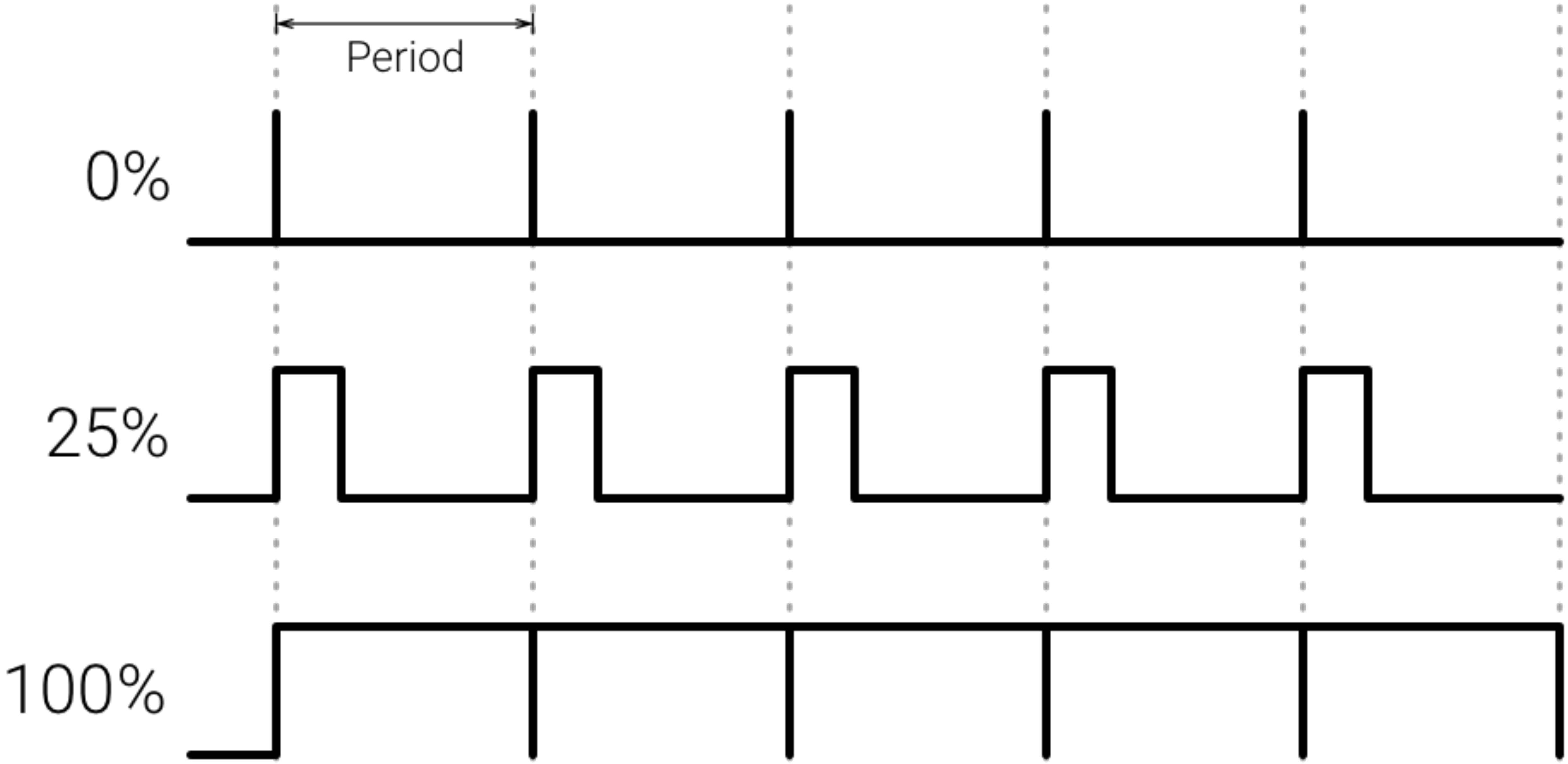
Lecture #4
Developer Platform

Android Things 2019

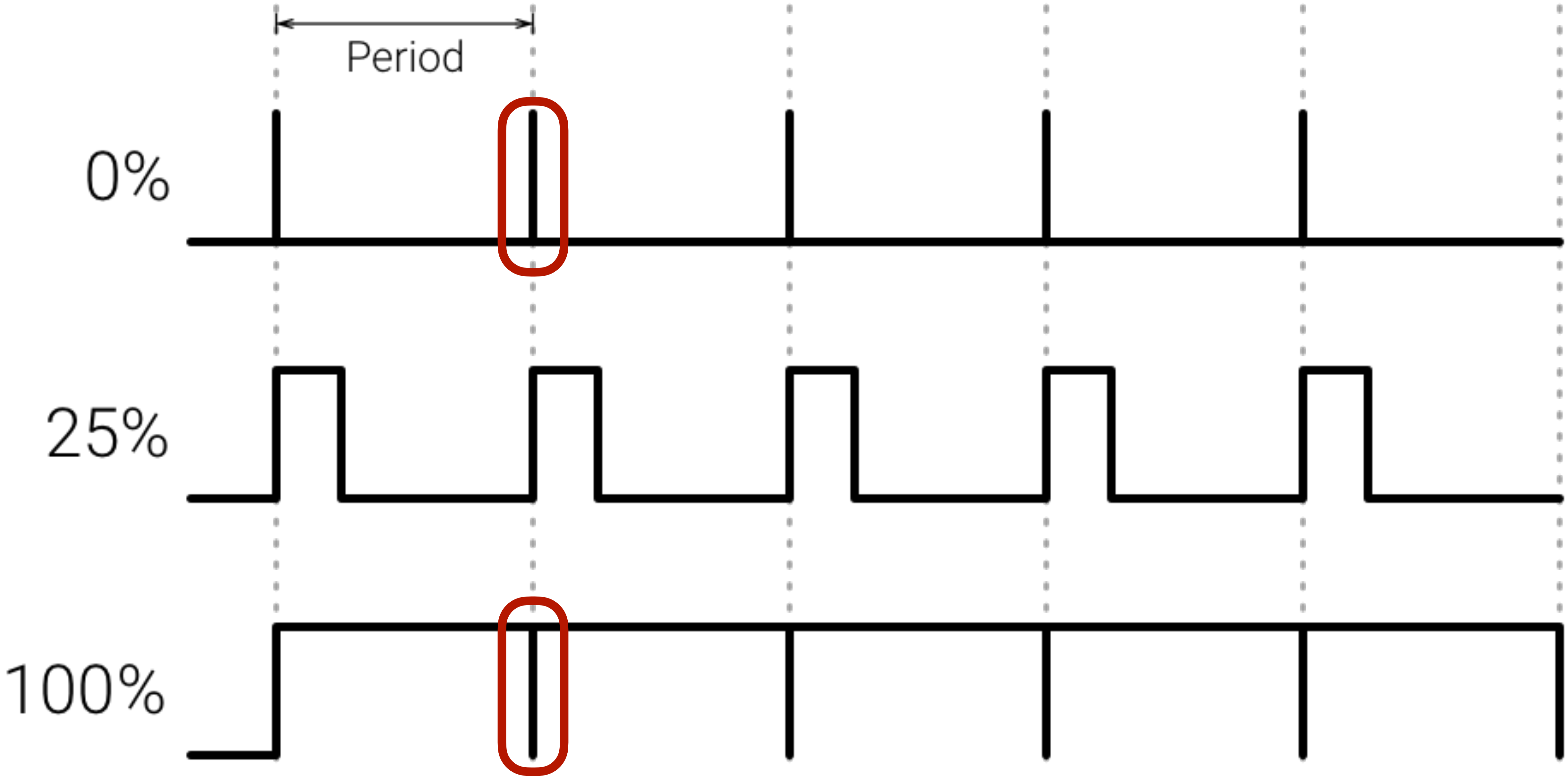
Pulse Width Modulation

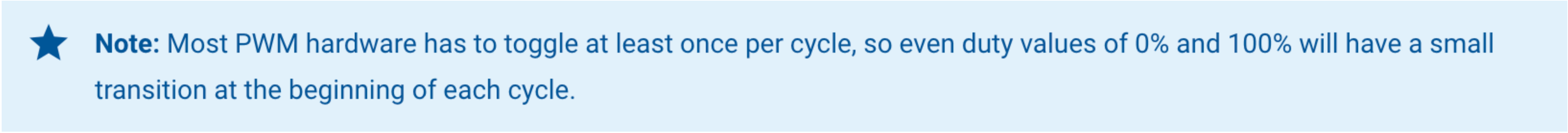


Pulse Width Modulation



Pulse Width Modulation





★ **Note:** Most PWM hardware has to toggle at least once per cycle, so even duty values of 0% and 100% will have a small transition at the beginning of each cycle.

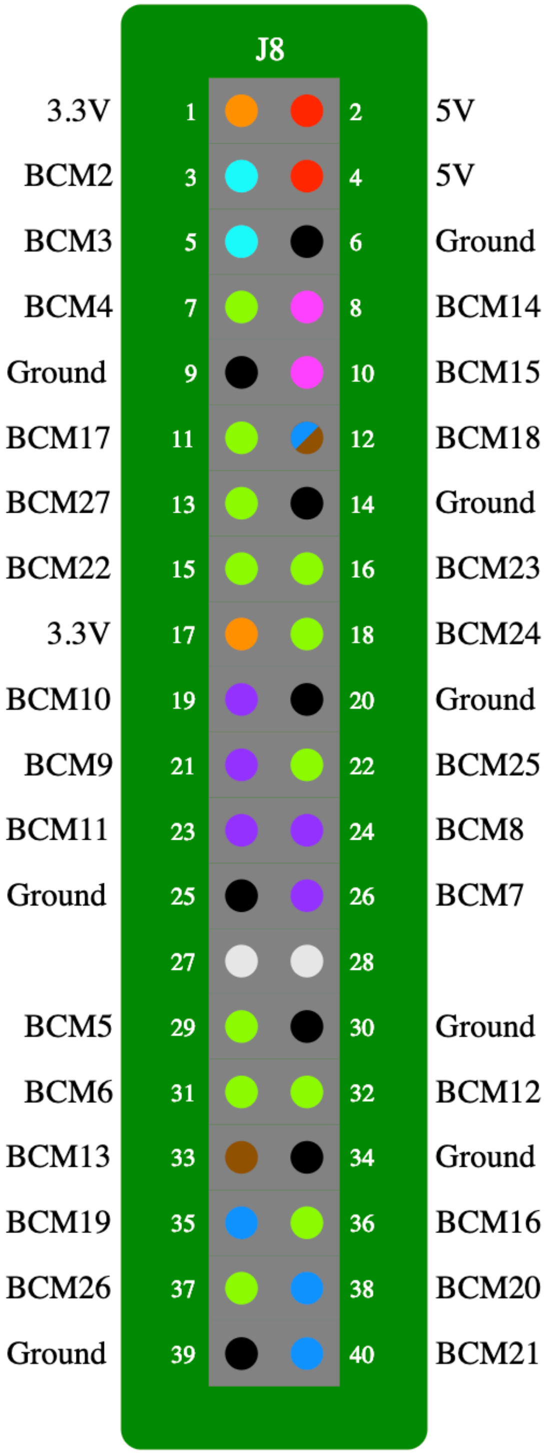
Permissions

```
<uses-permission
     android:name="com.google.android.things.permission.USE_PERIPHERAL_IO" />
```

Managing the connection

```
val manager = PeripheralManager.getInstance()
val portList: List<String> = manager.pwmList
if (portList.isEmpty()) {
    Log.i(TAG, "No PWM port available on this device.")
} else {
    Log.i(TAG, "List of available ports: $portList")
}
```

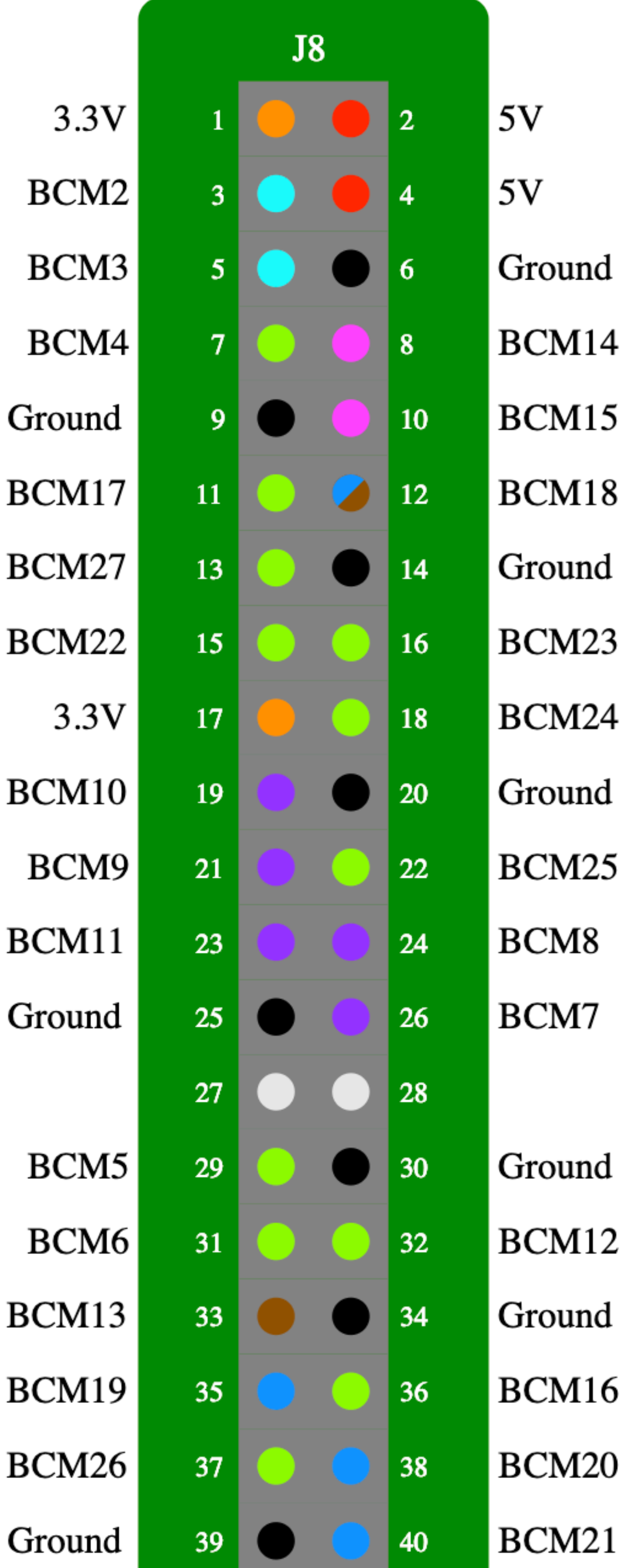
Pinout



J8

3.3V	1	2	5V
BCM2	3	4	5V
BCM3	5	6	Ground
BCM4	7	8	BCM14
Ground	9	10	BCM15
BCM17	11	12	BCM18
BCM27	13	14	Ground
BCM22	15	16	BCM23
3.3V	17	18	BCM24
BCM10	19	20	Ground
BCM9	21	22	BCM25
BCM11	23	24	BCM8
Ground	25	26	BCM7
	27	28	
BCM5	29	30	Ground
BCM6	31	32	BCM12
BCM13	33	34	Ground
BCM19	35	36	BCM16
BCM26	37	38	BCM20
Ground	39	40	BCM21

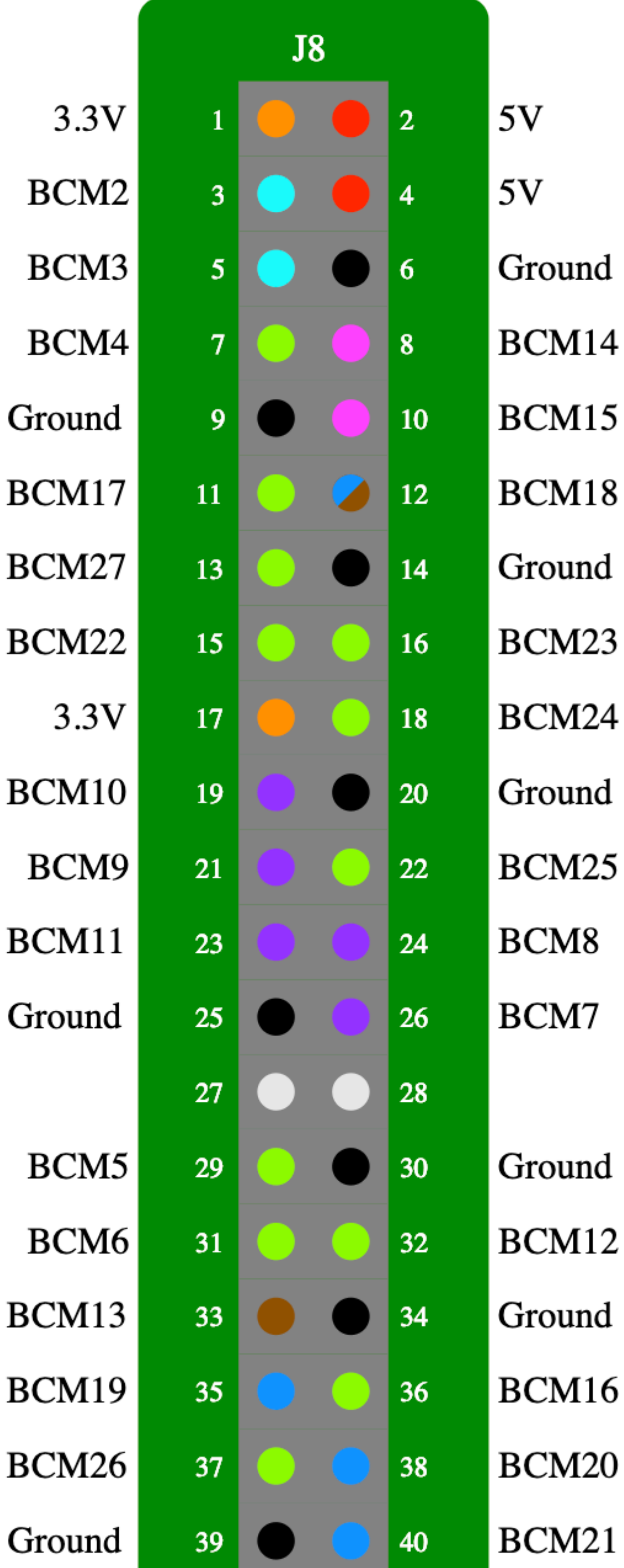
GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	



J8

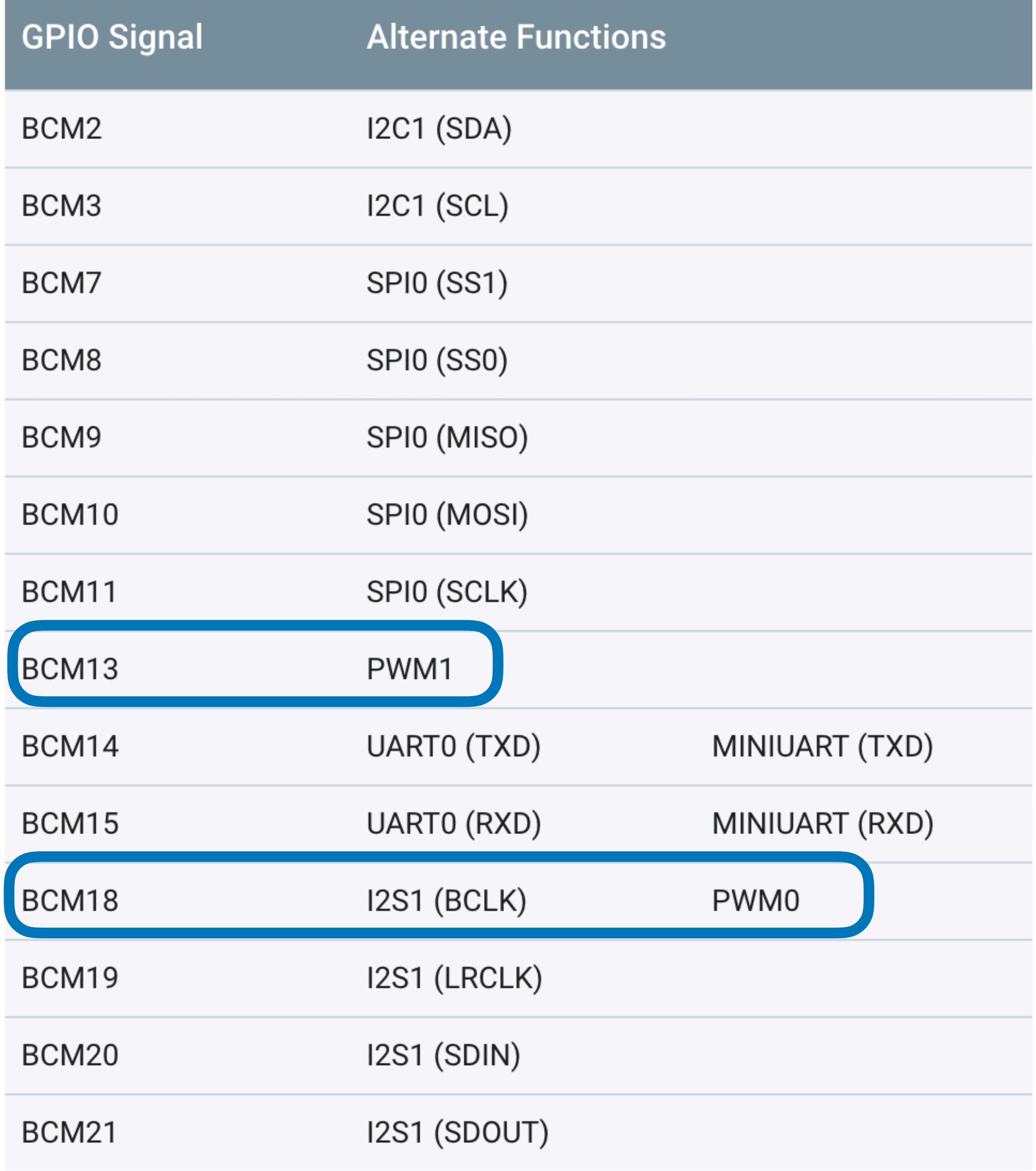
3.3V	1	2	5V
BCM2	3	4	5V
BCM3	5	6	Ground
BCM4	7	8	BCM14
Ground	9	10	BCM15
BCM17	11	12	BCM18
BCM27	13	14	Ground
BCM22	15	16	BCM23
3.3V	17	18	BCM24
BCM10	19	20	Ground
BCM9	21	22	BCM25
BCM11	23	24	BCM8
Ground	25	26	BCM7
	27	28	
BCM5	29	30	Ground
BCM6	31	32	BCM12
BCM13	33	34	Ground
BCM19	35	36	BCM16
BCM26	37	38	BCM20
Ground	39	40	BCM21

GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

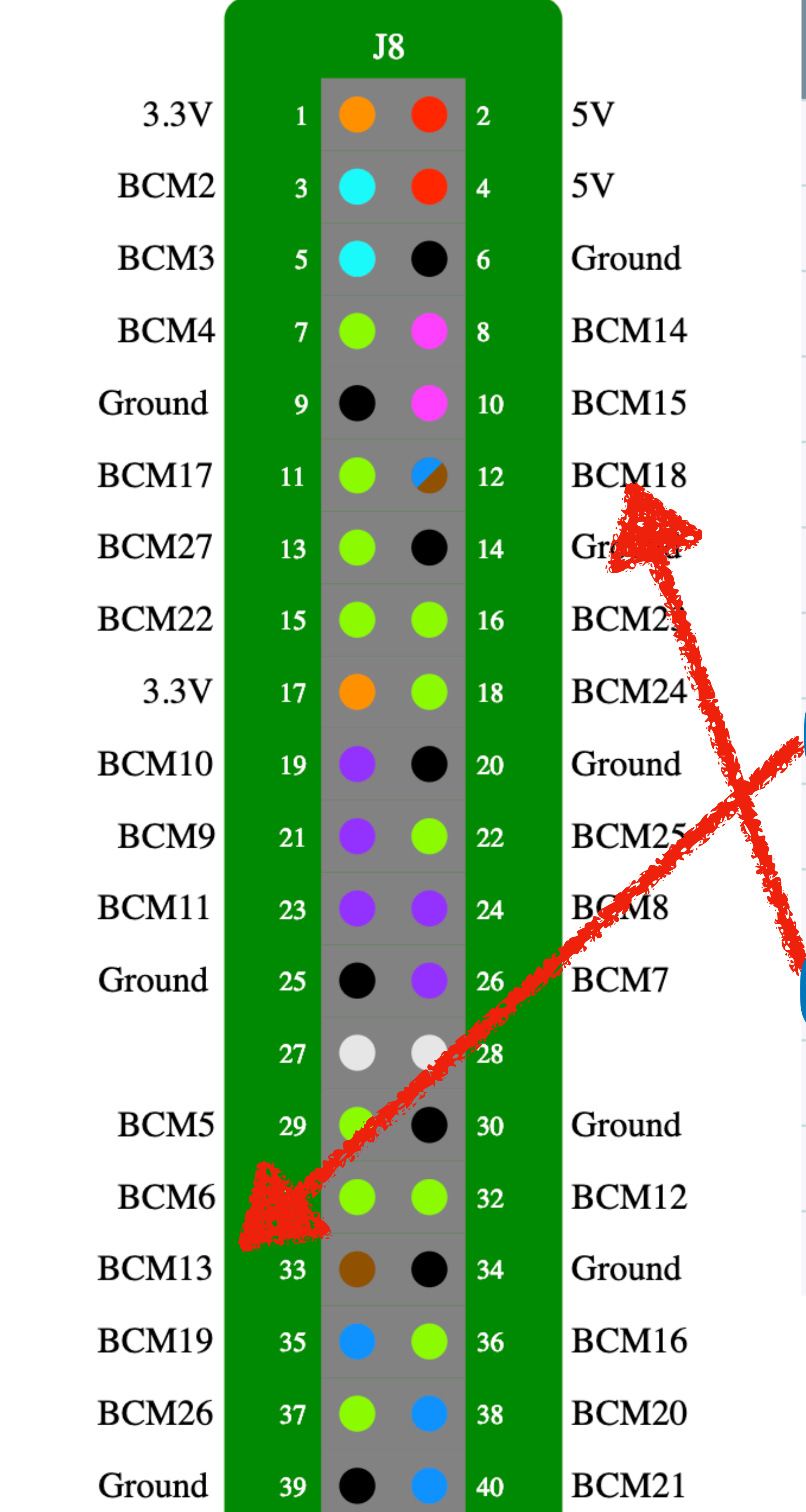


J8

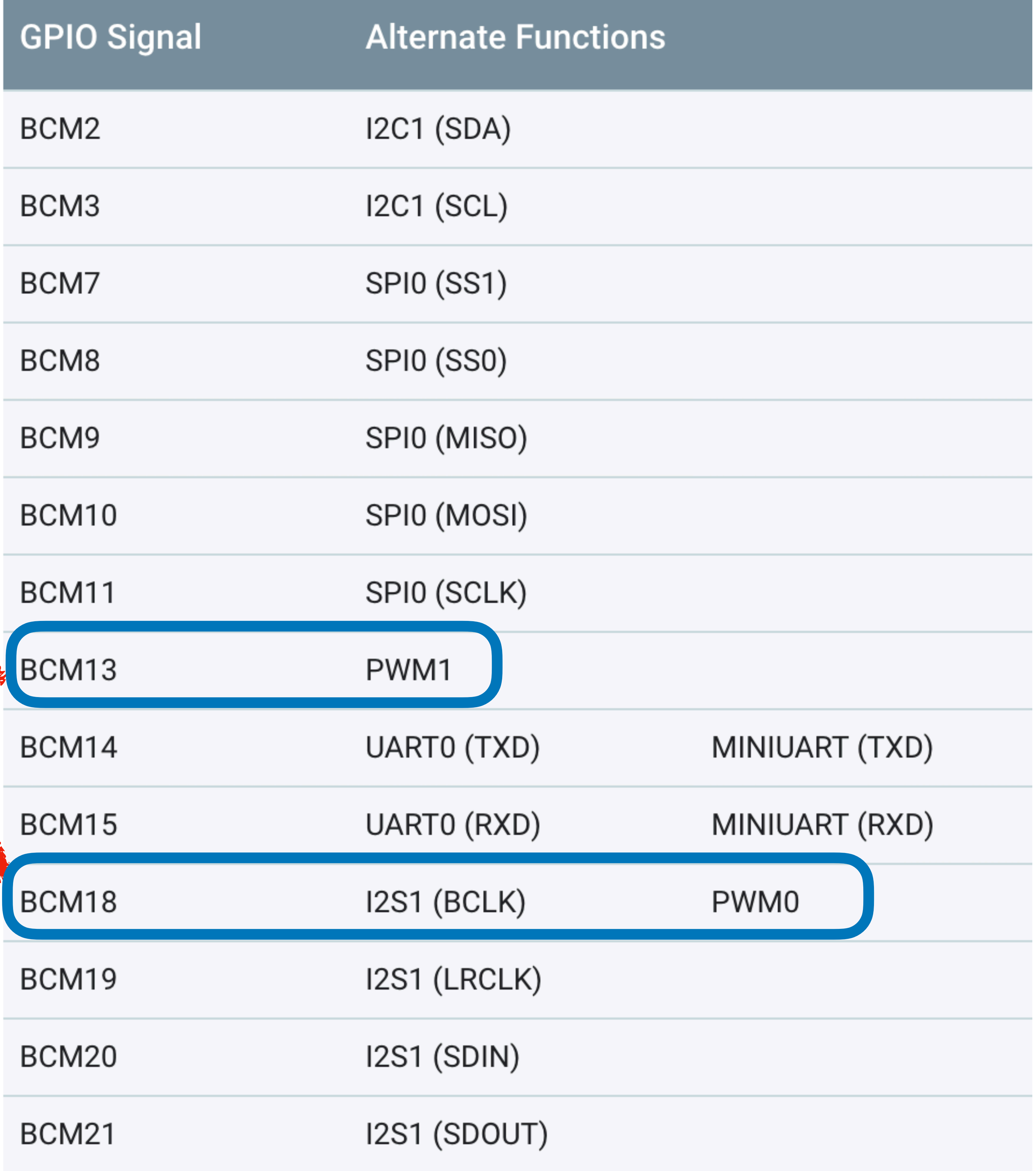
3.3V	1	2	5V
BCM2	3	4	5V
BCM3	5	6	Ground
BCM4	7	8	BCM14
Ground	9	10	BCM15
BCM17	11	12	BCM18
BCM27	13	14	Ground
BCM22	15	16	BCM23
3.3V	17	18	BCM24
BCM10	19	20	Ground
BCM9	21	22	BCM25
BCM11	23	24	BCM8
Ground	25	26	BCM7
	27	28	
BCM5	29	30	Ground
BCM6	31	32	BCM12
BCM13	33	34	Ground
BCM19	35	36	BCM16
BCM26	37	38	BCM20
Ground	39	40	BCM21



GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	



	J8		
3.3V	1	2	5V
BCM2	3	4	5V
BCM3	5	6	Ground
BCM4	7	8	BCM14
Ground	9	10	BCM15
BCM17	11	12	BCM18
BCM27	13	14	Ground
BCM22	15	16	BCM23
3.3V	17	18	BCM24
BCM10	19	20	Ground
BCM9	21	22	BCM25
BCM11	23	24	BCM8
Ground	25	26	BCM7
	27	28	
BCM5	29	30	Ground
BCM6	31	32	BCM12
BCM13	33	34	Ground
BCM19	35	36	BCM16
BCM26	37	38	BCM20
Ground	39	40	BCM21

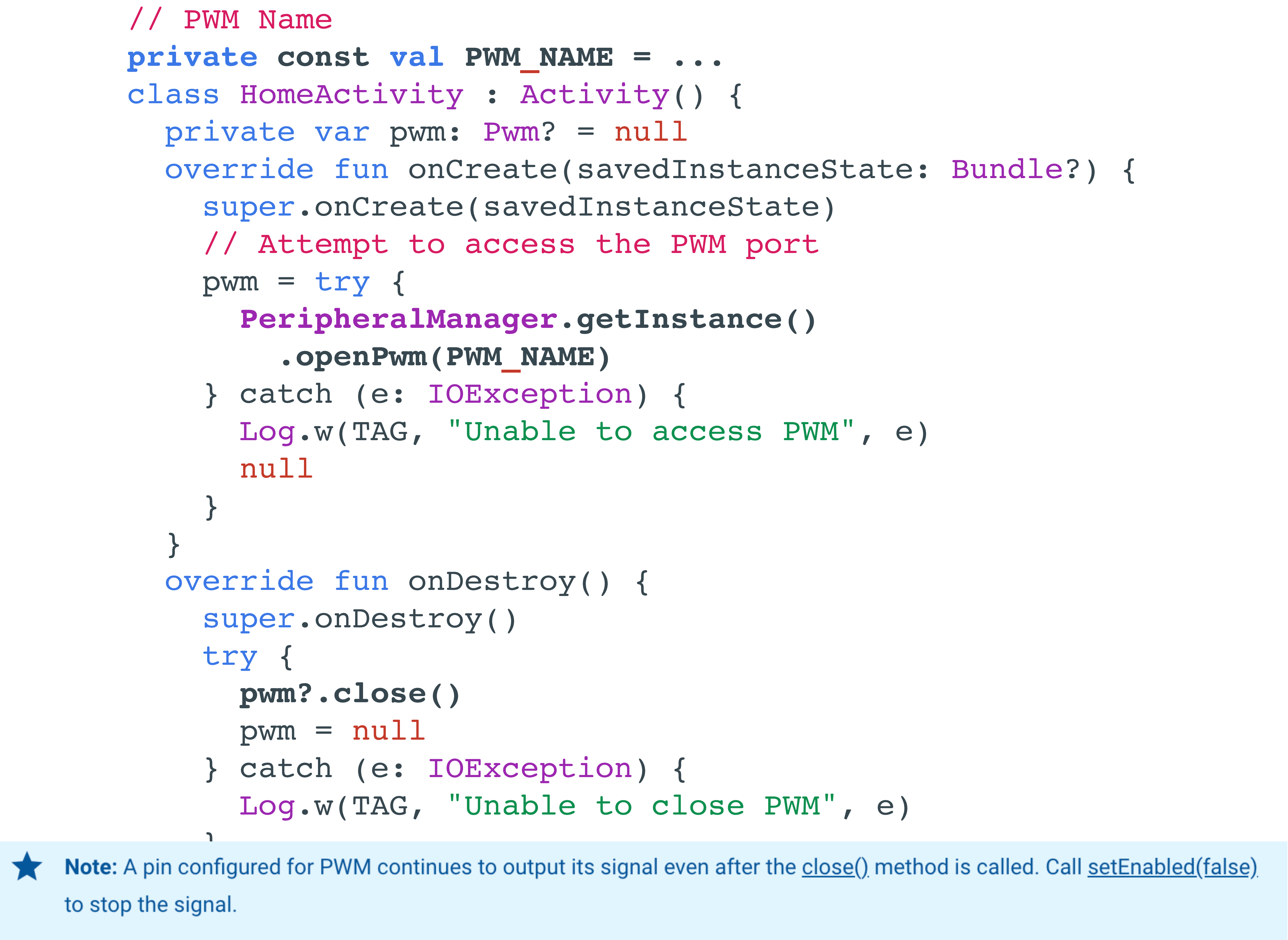


GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

Access the PWM port

```
// PWM Name
private const val PWM_NAME = ...
class HomeActivity : Activity() {
  private var pwm: Pwm? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the PWM port
    pwm = try {
      PeripheralManager.getInstance()
        .openPwm(PWM_NAME)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access PWM", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
    try {
      pwm?.close()
      pwm = null
    } catch (e: IOException) {
      Log.w(TAG, "Unable to close PWM", e)
    }
  }
}
```

Access the PWM port



```
// PWM Name
private const val PWM_NAME = ...
class HomeActivity : Activity() {
  private var pwm: Pwm? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the PWM port
    pwm = try {
      PeripheralManager.getInstance()
        .openPwm(PWM_NAME)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access PWM", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
    try {
      pwm?.close()
      pwm = null
    } catch (e: IOException) {
      Log.w(TAG, "Unable to close PWM", e)
    }
```

★ **Note:** A pin configured for PWM continues to output its signal even after the close() method is called. Call setEnabled(false) to stop the signal.

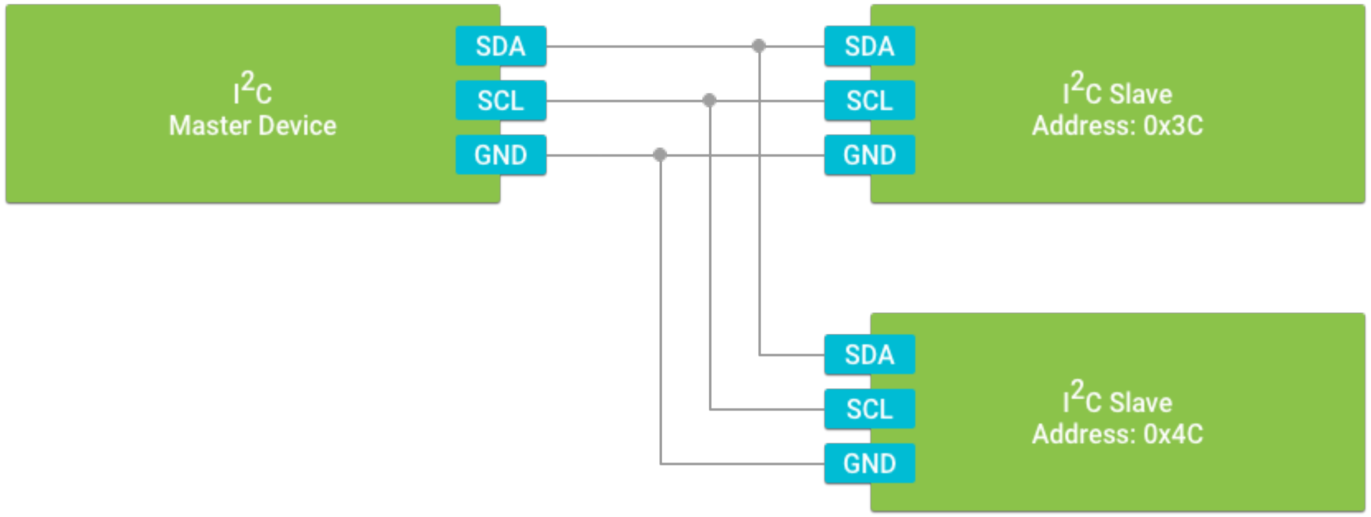
DEMO

Controlling the PWM signal

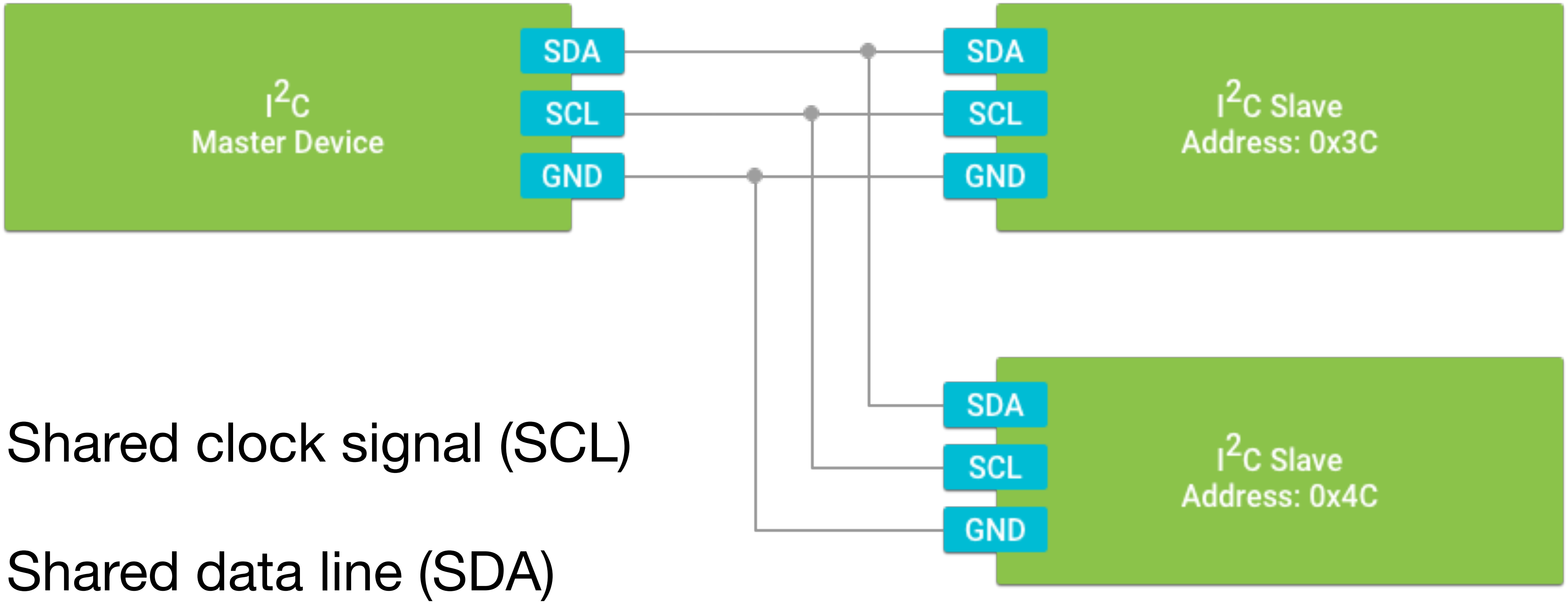
```kotlin
@Throws(IOException::class)
fun initializePwm(pwm: Pwm) {
  pwm.apply {
    setPwmFrequencyHz(120.0)
    setPwmDutyCycle(25.0)

    // Enable the PWM signal
    setEnabled(true)
  }
}
```

Inter-Integrated Circuit I2C

- I2C is a synchronous serial interface.
  - 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**.

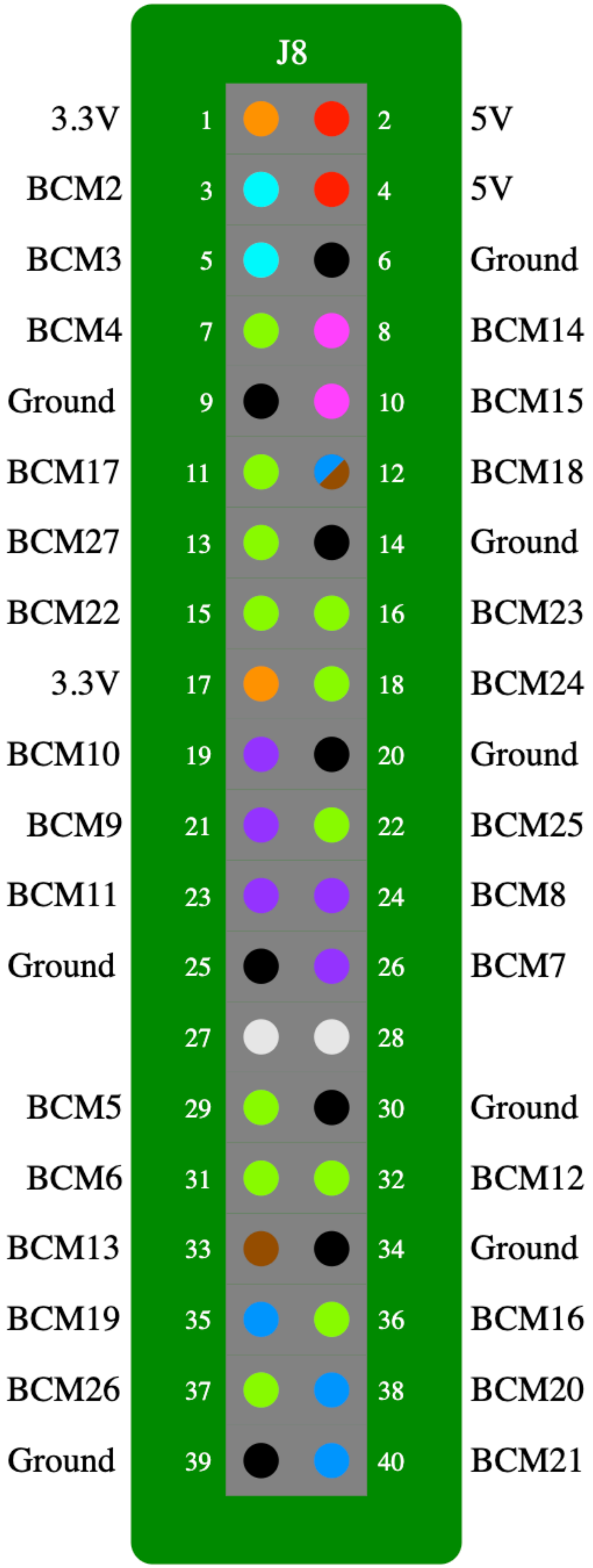


Inter-Integrated Circuit I2C

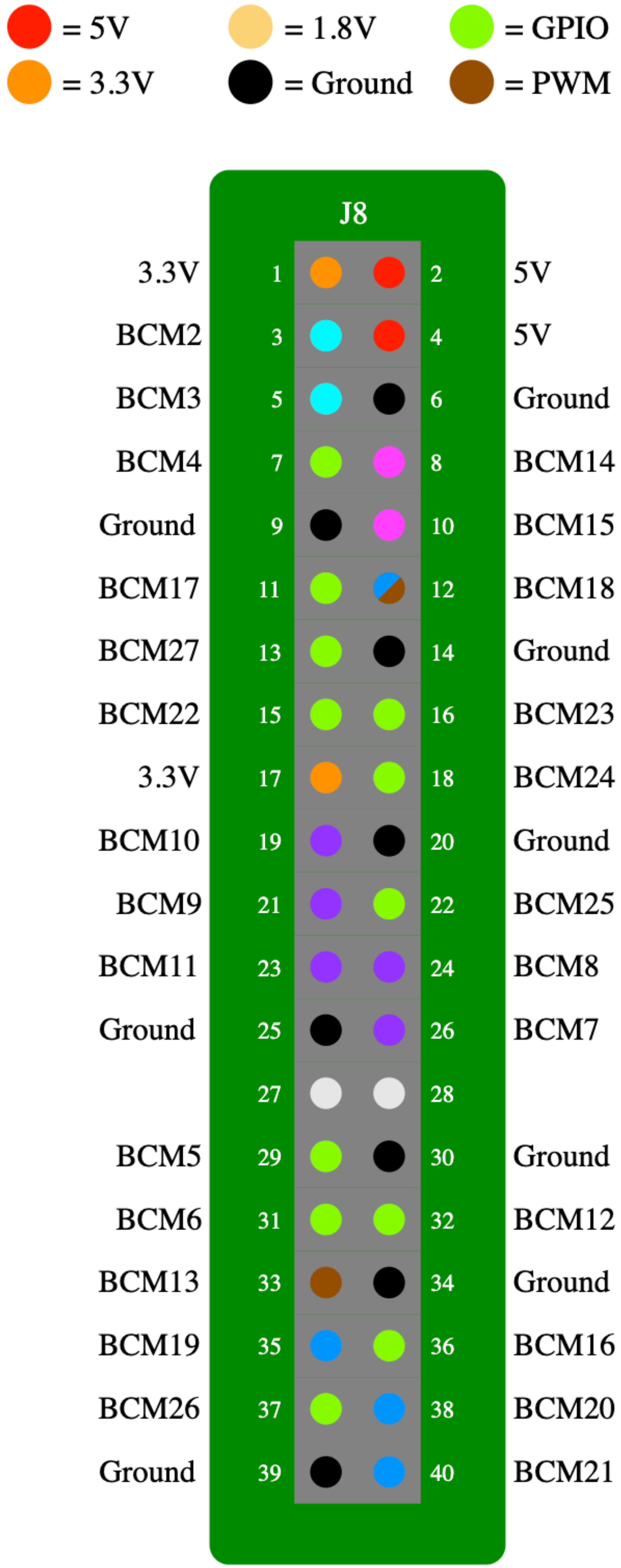


- Shared clock signal (SCL)
- Shared data line (SDA)
- Common ground reference (GND)

● = 5V ● = 1.8V ● = GPIO ● = I2C ● = SPI
● = 3.3V ● = Ground ● = PWM ● = I2S ● = UART

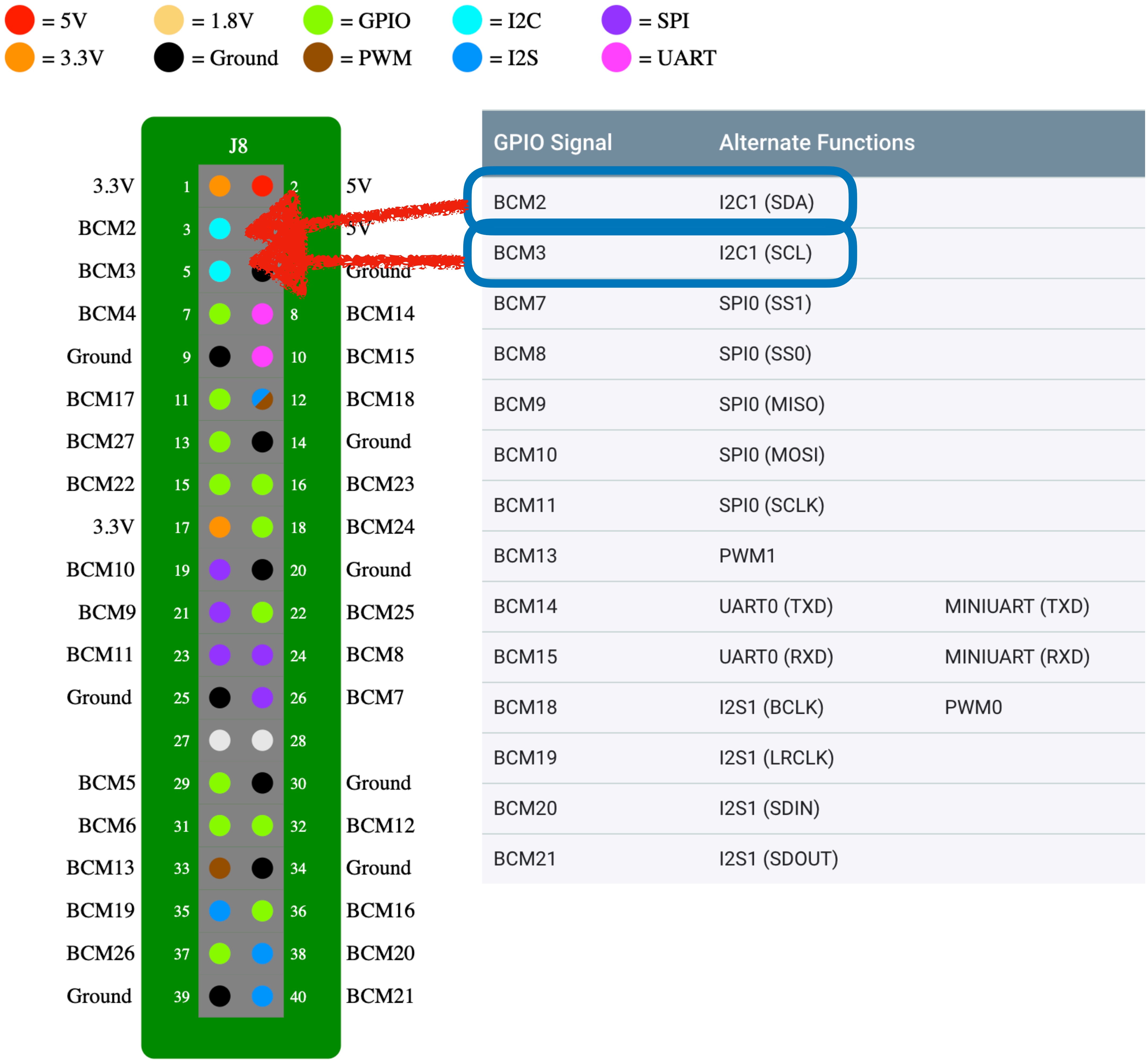


GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	



3.3V	1	2	5V
BCM2	3	4	5V
BCM3	5	6	Ground
BCM4	7	8	BCM14
Ground	9	10	BCM15
BCM17	11	12	BCM18
BCM27	13	14	Ground
BCM22	15	16	BCM23
3.3V	17	18	BCM24
BCM10	19	20	Ground
BCM9	21	22	BCM25
BCM11	23	24	BCM8
Ground	25	26	BCM7
	27	28	
BCM5	29	30	Ground
BCM6	31	32	BCM12
BCM13	33	34	Ground
BCM19	35	36	BCM16
BCM26	37	38	BCM20
Ground	39	40	BCM21

GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	



● = 5V ● = 1.8V ● = GPIO ● = I2C ● = SPI
● = 3.3V ● = Ground ● = PWM ● = I2S ● = UART

J8

3.3V	1	2	5V
BCM2	3	4	5V
BCM3	5	6	Ground
BCM4	7	8	BCM14
Ground	9	10	BCM15
BCM17	11	12	BCM18
BCM27	13	14	Ground
BCM22	15	16	BCM23
3.3V	17	18	BCM24
BCM10	19	20	Ground
BCM9	21	22	BCM25
BCM11	23	24	BCM8
Ground	25	26	BCM7
	27	28	
BCM5	29	30	Ground
BCM6	31	32	BCM12
BCM13	33	34	Ground
BCM19	35	36	BCM16
BCM26	37	38	BCM20
Ground	39	40	BCM21

GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

Adding the required permissions

```
<uses-permission
     android:name="com.google.android.things.permission.USE_PERIPHERAL_IO" />
```

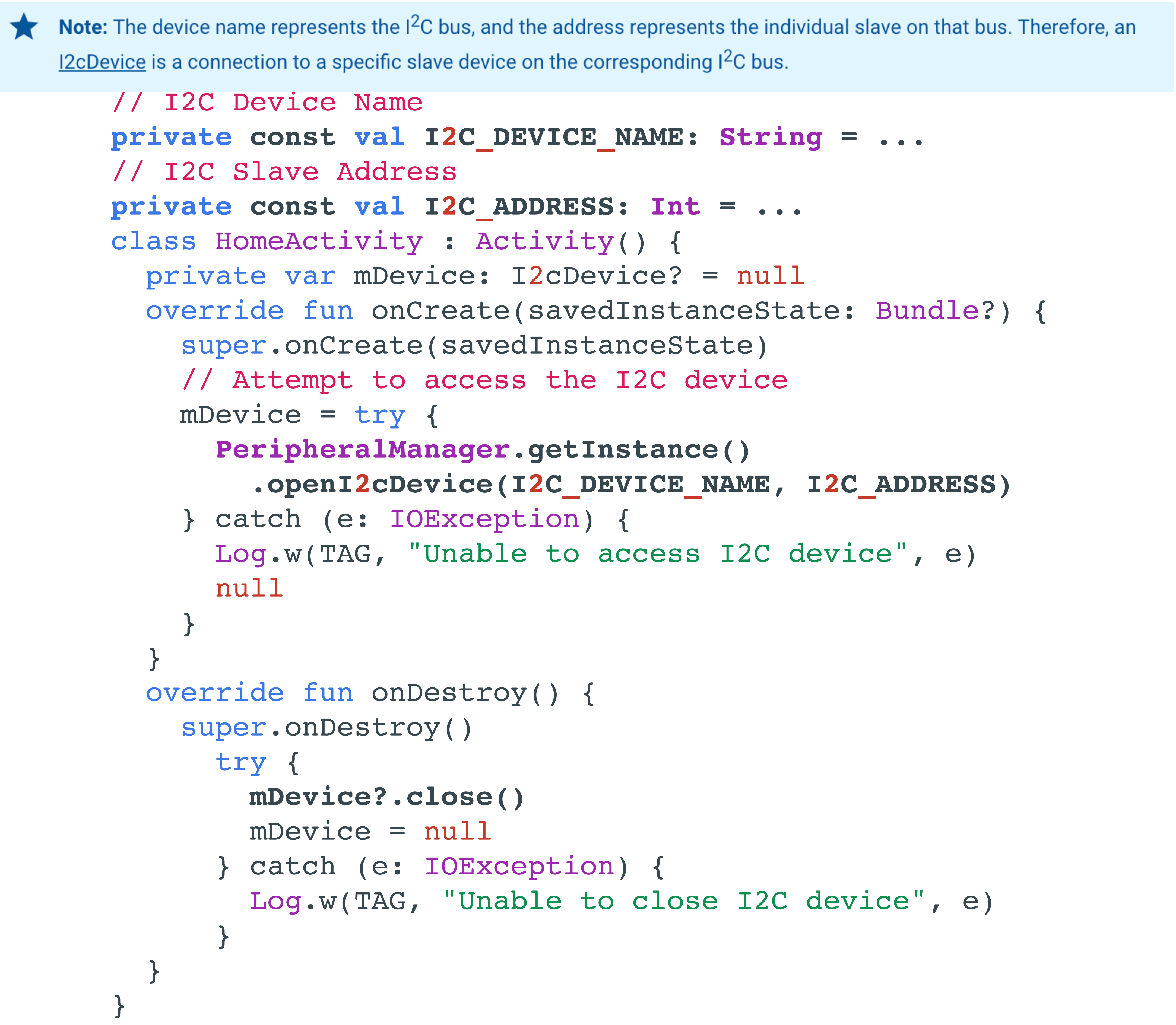
Managing the slave device connection

```
val manager = PeripheralManager.getInstance()
val deviceList: List<String> = manager.i2cBusList
if (deviceList.isEmpty()) {
  Log.i(TAG, "No I2C bus available on this device.")
} else {
  Log.i(TAG, "List of available devices: $deviceList")
}
```

Access the I2C device

```
// I2C Device Name
private const val I2C_DEVICE_NAME: String = ...
// I2C Slave Address
private const val I2C_ADDRESS: Int = ...
class HomeActivity : Activity() {
  private var mDevice: I2cDevice? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the I2C device
    mDevice = try {
      PeripheralManager.getInstance()
        .openI2cDevice(I2C_DEVICE_NAME, I2C_ADDRESS)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access I2C device", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
      try {
        mDevice?.close()
        mDevice = null
      } catch (e: IOException) {
        Log.w(TAG, "Unable to close I2C device", e)
      }
```

```kotlin
// I2C Device Name
private const val I2C_DEVICE_NAME: String = ...
// I2C Slave Address
private const val I2C_ADDRESS: Int = ...
class HomeActivity : Activity() {
  private var mDevice: I2cDevice? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the I2C device
    mDevice = try {
      PeripheralManager.getInstance()
        .openI2cDevice(I2C_DEVICE_NAME, I2C_ADDRESS)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access I2C device", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
      try {
        mDevice?.close()
        mDevice = null
      } catch (e: IOException) {
        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.

```kotlin
// I2C Device Name
private const val I2C_DEVICE_NAME: String = ...
// I2C Slave Address
private const val I2C_ADDRESS: Int = ...
class HomeActivity : Activity() {
  private var mDevice: I2cDevice? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the I2C device
    mDevice = try {
      PeripheralManager.getInstance()
        .openI2cDevice(I2C_DEVICE_NAME, I2C_ADDRESS)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access I2C device", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
      try {
        mDevice?.close()
        mDevice = null
      } catch (e: IOException) {
        Log.w(TAG, "Unable to close I2C device", e)
      }
  }
}
```

Determine available addresses

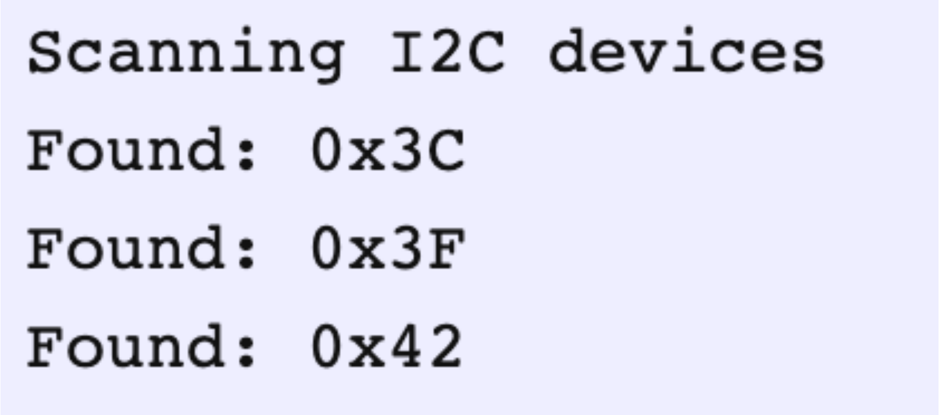
```
fun PeripheralManager.scanI2cAvailableAddresses(i2cName: String): List<Int> {
  return (0..127).filter { address ->
    with(openI2cDevice(i2cName, address)) {
      try {
        write(ByteArray(1), 1)
        true
      } catch (e: IOException) {
        false
      } finally {
        close()
      }
    }
  }
}
```

Determine available addresses

```
fun PeripheralManager.scanI2cAvailableAddresses(i2cName: String): List<Int> {
  return (0..127).filter { address ->
    with(openI2cDevice(i2cName, address)) {
      try {
        write(ByteArray(1), 1)
        true
      } catch (e: IOException) {
        false
      } finally {
        close()
      }
    }
  }
}

Log.i(TAG, "Scanning I2C devices")
manager.scanI2cAvailableAddresses(I2C_BUS_NAME)
    .map { String.format(Locale.US, "0x%02X", it) }
    .forEach { address -> Log.i(TAG, "Found: $address") }
}
```

Determine available addresses

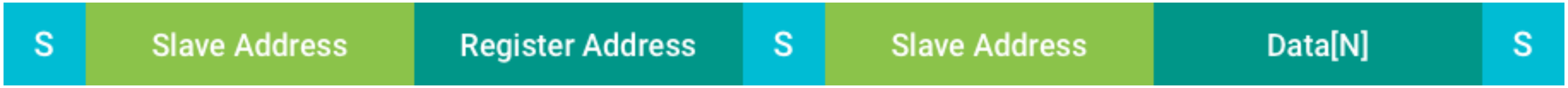
```
fun PeripheralManager.scanI2cAvailableAddresses(i2cName: String): List<Int> {
  return (0..127).filter { address ->
    with(openI2cDevice(i2cName, address)) {
      try {
        write(ByteArray(1), 1)
        true
      } catch (e: IOException) {
        false
      } finally {
        close()
      }
    }
  }
}
```



```
Scanning I2C devices
Found: 0x3C
Found: 0x3F
Found: 0x42
```

```
Log.i(TAG, "Scanning I2C devices")
manager.scanI2cAvailableAddresses(I2C_BUS_NAME)
    .map { String.format(Locale.US, "0x%02X", it) }
    .forEach { address -> Log.i(TAG, "Found: $address") }
}
```

Interacting with registers



S | Slave Address | Register Address | S | Slave Address | Data[N] | S

- Byte Data: **readRegByte**() and **writeRegByte**() Read or write a single 8-bit register value.

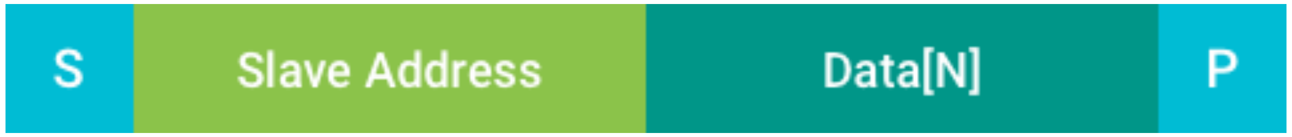
- Word Data: **readRegWord**() and **writeRegWord**() Read or write two consecutive register values as a 16-bit little-endian word. The first register address corresponds to the least significant byte (LSB) in the word, followed by the most significant byte (MSB).

- Block Data: **readRegBuffer**() and **writeRegBuffer**() Read or write up to 32 consecutive register values as an array.

Interacting with registers

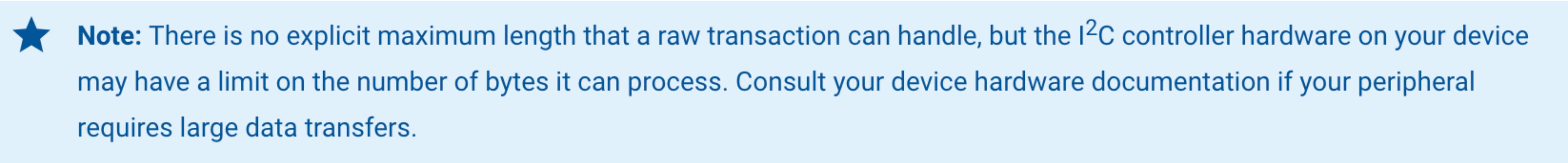
```
// Modify the contents of a single register
@Throws(IOException::class)
fun setRegisterFlag(device: I2cDevice, address: Int) {
    // Read one register from slave
    var value = device.readRegByte(address)
    // Set bit 6
    value = value or 0x40
    // Write the updated value back to slave
    device.writeRegByte(address, value)
}

// Read a register block
@Throws(IOException::class)
fun readCalibration(device: I2cDevice, startAddress: Int): ByteArray {
    // Read three consecutive register values
    return ByteArray(3).also { data ->
        device.readRegBuffer(startAddress, data, data.size)
    }
}
```

Transferring raw data



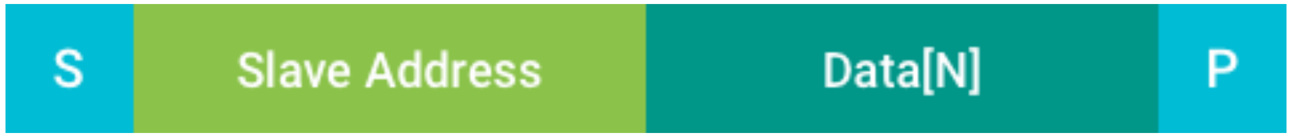
S	Slave Address	Data[N]	P

```
@Throws(IOException::class)
fun writeBuffer(device: I2cDevice, buffer: ByteArray) {
    device.write(buffer, buffer.size).also { count ->
        Log.d(TAG, "Wrote $count bytes over I2C.")
    }
}
```

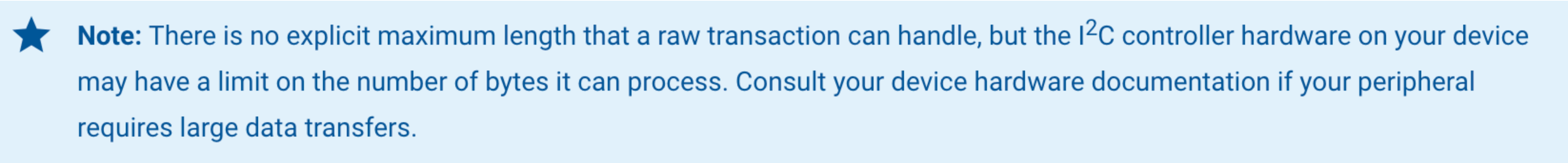


★ **Note:** There is no explicit maximum length that a raw transaction can handle, but the I^2C controller hardware on your device may have a limit on the number of bytes it can process. Consult your device hardware documentation if your peripheral requires large data transfers.

Transferring raw data



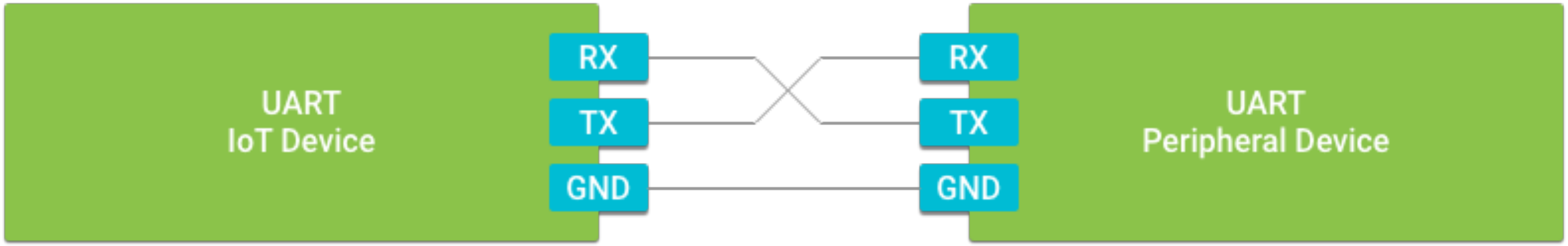
```
@Throws(IOException::class)
fun writeBuffer(device: I2cDevice, buffer: ByteArray) {
    device.write(buffer, buffer.size).also { count ->
        Log.d(TAG, "Wrote $count bytes over I2C.")
    }
}
```



★ **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.

https://github.com/Nilhcem/i2cfun-androidthings/

Universal Asynchronous Receiver Transmitter - UART

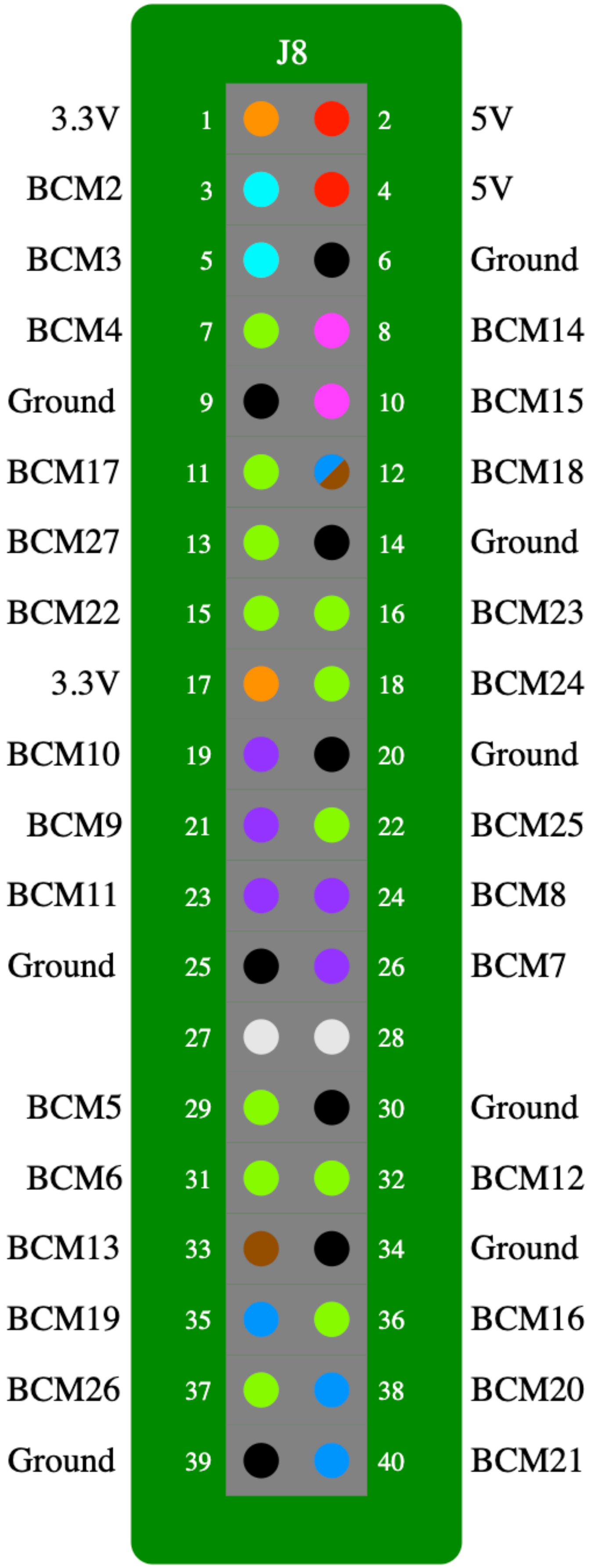


- GPS modules.
- LCD displays.

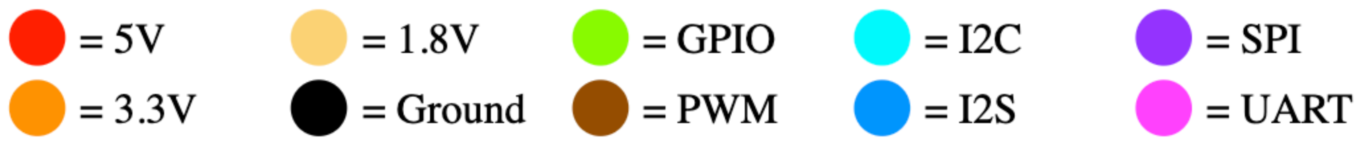
- 3-Wire ports include data receive (RX), data transmit (TX), and ground reference (GND) signals.
- 5-Wire ports add request to send (RTS) and clear to send (CTS) signals used for hardware flow control.

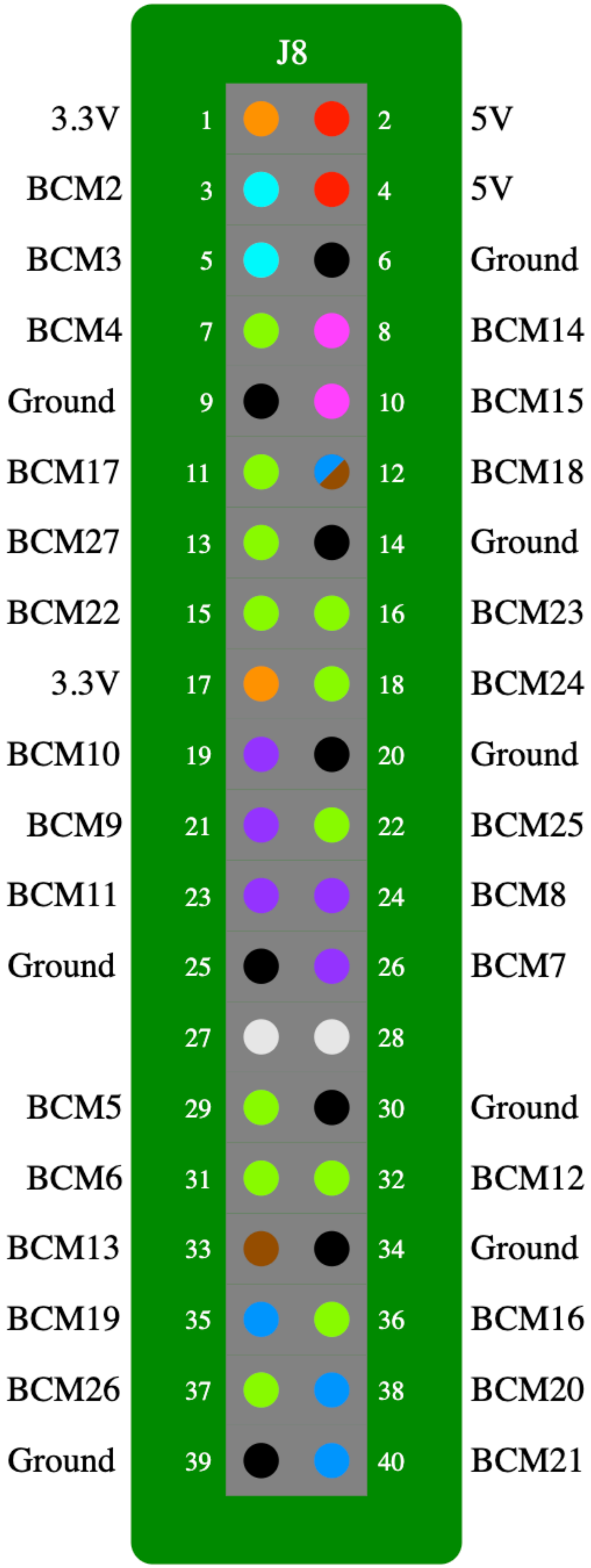
🔴 = 5V 🟡 = 1.8V 🟢 = GPIO 🔵 = I2C 🟣 = SPI

🟠 = 3.3V ⚫ = Ground 🟤 = PWM 🔵 = I2S 🟣 = UART



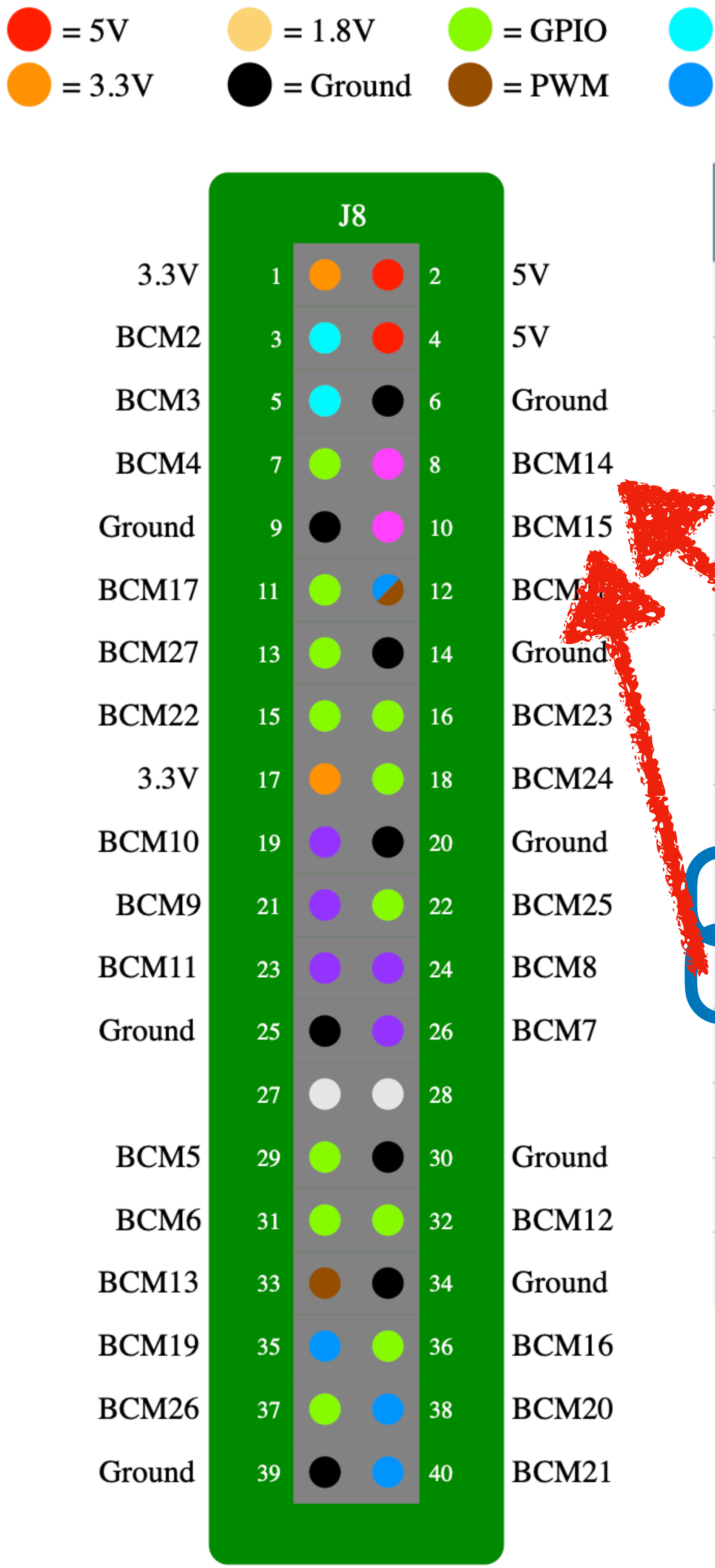
GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	





3.3V	1	2	5V
BCM2	3	4	5V
BCM3	5	6	Ground
BCM4	7	8	BCM14
Ground	9	10	BCM15
BCM17	11	12	BCM18
BCM27	13	14	Ground
BCM22	15	16	BCM23
3.3V	17	18	BCM24
BCM10	19	20	Ground
BCM9	21	22	BCM25
BCM11	23	24	BCM8
Ground	25	26	BCM7
	27	28	
BCM5	29	30	Ground
BCM6	31	32	BCM12
BCM13	33	34	Ground
BCM19	35	36	BCM16
BCM26	37	38	BCM20
Ground	39	40	BCM21

GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	



GPIO Signal	Alternate Functions	
BCM2	I2C1 (SDA)	
BCM3	I2C1 (SCL)	
BCM7	SPI0 (SS1)	
BCM8	SPI0 (SS0)	
BCM9	SPI0 (MISO)	
BCM10	SPI0 (MOSI)	
BCM11	SPI0 (SCLK)	
BCM13	PWM1	
BCM14	UART0 (TXD)	MINIUART (TXD)
BCM15	UART0 (RXD)	MINIUART (RXD)
BCM18	I2S1 (BCLK)	PWM0
BCM19	I2S1 (LRCLK)	
BCM20	I2S1 (SDIN)	
BCM21	I2S1 (SDOUT)	

Managing the connection

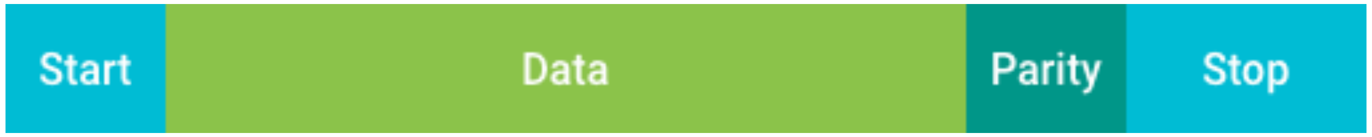
```
<uses-permission
android:name="com.google.android.things.permission.USE_PERIPHERAL_IO" />
```

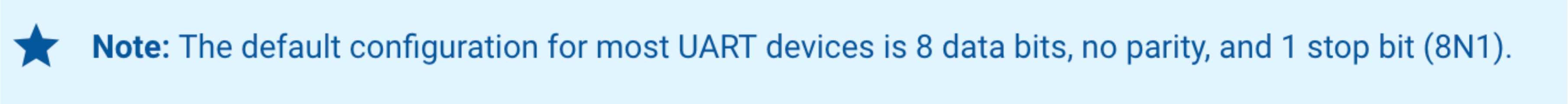
```
val manager = PeripheralManager.getInstance()
val deviceList: List<String> = manager.uartDeviceList
if (deviceList.isEmpty()) {
  Log.i(TAG, "No UART port available on this device.")
} else {
  Log.i(TAG, "List of available devices: $deviceList")
}
```

Access UART Device

```
// UART Device Name
private val UART_DEVICE_NAME: String = ...
class HomeActivity : Activity() {
  private var mDevice: UartDevice? = null
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Attempt to access the UART device
    mDevice = try {
      PeripheralManager.getInstance()
        .openUartDevice(UART_DEVICE_NAME)
    } catch (e: IOException) {
      Log.w(TAG, "Unable to access UART device", e)
      null
    }
  }
  override fun onDestroy() {
    super.onDestroy()
    try {
      mDevice?.close()
      mDevice = null
    } catch (e: IOException) {
      Log.w(TAG, "Unable to close UART device", e)
    }
  }
}
```

Configuring port parameters




★ **Note:** The default configuration for most UART devices is 8 data bits, no parity, and 1 stop bit (8N1).

```
@Throws(IOException::class)
fun configureUartFrame(uart: UartDevice) {
  uart.apply {
    // Configure the UART port
    setBaudrate(115200)
    setDataSize(8)
    setParity(UartDevice.PARITY_NONE)
    setStopBits(1)
  }
}
```

https://developer.android.com/things/sdk/pio/uart

Transmitting outgoing data

```
@Throws(IOException::class)
fun writeUartData(uart: UartDevice) {
  val count = uart.run {
    ByteArray(...).let { buffer ->
      write(buffer, buffer.size)
    }
  }
  Log.d(TAG, "Wrote $count bytes to peripheral")
}
```

Listening for incoming data

```kotlin
@Throws(IOException::class)
fun readUartBuffer(uart: UartDevice) {
  // Maximum amount of data to read at one time
  val maxCount = ...

  uart.apply {
    ByteArray(maxCount).also { buffer ->
      var count: Int = read(buffer, buffer.size)
      while (count > 0) {
        Log.d(TAG, "Read $count bytes from peripheral")
        count = read(buffer, buffer.size)
      }
    }
  }
}
```

```
class HomeActivity : Activity() {
  private var mDevice: UartDevice? = null
  ...
  override fun onStart() {
    super.onStart()
    // Begin listening for interrupt events
    mDevice?.registerUartDeviceCallback(uartCallback)
  }
  override fun onStop() {
    super.onStop()
    // Interrupt events no longer necessary
    mDevice?.unregisterUartDeviceCallback(uartCallback)
  }
  private val uartCallback = object : UartDeviceCallback {
    override fun onUartDeviceDataAvailable(uart: UartDevice): Boolean {
      // Read available data from the UART device
      try {
        readUartBuffer(uart)
      } catch (e: IOException) {
        Log.w(TAG, "Unable to access UART device", e)
      }
      // Continue listening for more interrupts
      return true
    }
    override fun onUartDeviceError(uart: UartDevice?, error: Int) {
      Log.w(TAG, "$uart: Error event $error")
    }
  }
}
```

Lecture outcomes

- Understand PWM and I2C.
- Transfer data using UART.