FLO

Hierarchical Distributed Dataflow

FloHW (hardware) Package

April 2025

Contents

The **FloHW** package builds on the **FloBox**[1] package (Fig. 2.2.a), to provide a library of remote Flo device implementations for specific pieces of hardware (HW).

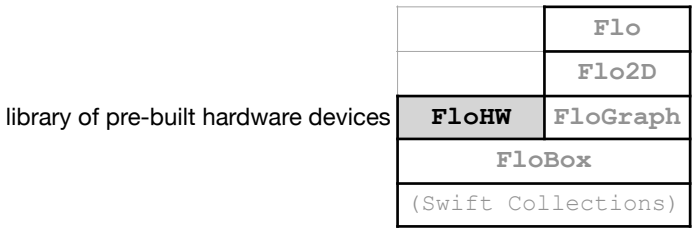

Fig. 2.2.a: FloHW package

The library currently contains Swift implementations of:

- General purpose UART[2] and i2c[3] wire communication protocols.
- Configuration and read/write of the Raspberry Pi[4] GPIO pins.
- Flo device wrappers for the following hardware components:
  - BME280[5] : pressure/temperature/humidity sensor
  - BNO055[6] : absolute orientation sensor (9 d.o.f. inertial measurement unit)
  - HC-SR04[7] : ultrasonic distance sensor
  - PCA 9685[8] : 16 channel servo driver
  - SSD1306[9] : 128x64 dot matrix OLED display panel.

In this document:

Section 1. **Hardware Protocols**: describes the UART, i2c and Raspberry Pi implementations in some detail, since we expect these to be of most use for developers;

Section 1. **Hardware Protocols**: introduces the device library, with an illustrative example.

[1] https://github.com/kk-0129/FloBox

[2] https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[3] https://en.wikipedia.org/wiki/I²C

[4] https://www.raspberrypi.com/

[5] https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf

[6] e.g. https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor/overview

[7] e.g. https://www.alldatasheet.com/datasheet-pdf/download/1132203/ETC2/HC-SR04.html

[8] https://learn.adafruit.com/16-channel-pwm-servo-driver/overview

[9] https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf

1. Hardware Protocols

This section describes the UART, i2c and Raspberry Pi utility classes in the **FloHW** package.

1.1. UART

The UART class, shown in Fig. 1.1.a, provides low-level access to a UART (Universal Asynchronous Receiver-Transmitter) communication channel. The class supports the standard range of UART configuration options (see Fig. 1.1.b) - defined as enumerations (**enum** types):

- **Baud = { 2400, 4800, 9600, 19200, 38400, 576, 128000, 115200 }**
- **BitsPerChar = { Six, Seven, Eight }**
- **StopBites = { One, Two }**
- **Parity = { None, Odd, Even }**

It also has methods to modify/monitor the channel's connection status, and to read/write data.

```
public class UART{
    // variables
    public let port:String
    public let config:Config
    public var connected:Bool{…}{
    // initialise
    public init(_ port:String,_ config:Config){…}
    // functions
    public func connect(){…}
    public func disconnect(){…}
    public func writeData(_ value:[UInt8]){…}
    public func readData(_ buf:inout [UInt8])->Int?{…}
}
```

Fig. 1.1.a: UART Class

```
public extension UART{
    public struct Config{
        let baud: Baud
        let bpc: BitsPerChar
        let stp: StopBits
        let par: Parity
   }
}
```

Fig. 1.1.b: UART extension - configuration options

The code snippet in Fig. 1.1.c illustrates the use of the **UART** class - in this particular case: code running on a Raspberry Pi communicating over a USB port registered in the Pi's **/dev** (device driver) registry as **/tty.SLAB_USBtoUART**.

```
let conf = UART.Config(._128000, .Eight, .One, .None)
let uart = UART("tty.SLAB_USBtoUART", conf)
if uart.connect(){
    var buf = [UInt8](repeating:0,count:128)
    while true{
        if let n = uart.readData(&buf),n > 0{
            // do something with the data returned in 'buf'
        }
    }
}
```

Fig. 1.1.c: UART example

1.2. i2c

The **I2C** class, shown in Figs. 1.2.a (and extension 1.2.b) provides low-level access to an i²c (Inter-Integrated Circuit) serial communication channel, typically used for short distance links between processors and low speed peripherals. The class initialiser, in Fig. 1.2.a, just requires the **id**s of the peripheral device and communication bus. The other methods provide various options for reading and writing data over the channel.

```
public class I2C{
    // variables
    public let bus:Int
    public let id:Int
    // initialiser
    public init(_ id:UInt8,_ bus:UInt8 = 1){…}
    // functions
    public func read8()->UInt8?{…}
    public func read8(_ channel:UInt8)->UInt8?{…}
    public func read16(_ channel:UInt8)->UInt16?{…}
    public func read(_ channel:UInt8,_ n:Int)->[UInt8]?{…}
    public func write8(_ v:UInt8){…}
    public func write8(_ channel:UInt8,_ value:UInt8){…}
    public func write16(_ channel:UInt8,_ value:UInt16){…}
    public func write(_ channel:UInt8,_ values:[UInt8]){…}
    public func writeI2C(_ channel:UInt8,_ values:[UInt8]){…}
}
```

Fig 1.2.a: I2C class

An example of the use of the **I2C** class is given in Section 2.1 (BNO055 example) below.

1.3. Raspberry Pi

The **RPi** class, Fig. 1.3.a, provides a simple wrapper for accessing the Raspberry Pis GPIO (general purpose input/output) pins - adopting Swift's subscript to access pins by index. The **Pin** class provides variables to specify whether the pin is to be used as input or output (and if output, to dynamically flip its state between high and low), and whether it requires a pull up or pull down resistor (or neither - *cf.* the **Pull** enum in Fig. 1.3.c)

```
public class RPi{
    // functions
    public static subscript(_ u:UInt)->Pin?
    // pins
    public final class Pin{
        public var out:Bool{…}
        public var high:Bool{…}
        public var pull:Pull{…}
    }
    public enum Pull{ case Neither, Down, Up }
}
```

Fig. 1.3.a: RPi (Raspberry Pi) class

The **RPi** class can only be instantiated on a Raspberry Pi itself running Linux.

2. Device Library

As stated earlier, the **FloHW** package contains a small library of fully functional **Device.Box** (defined in the FloBox package) implementations for the following specific pieces of hardware:

- BME280 : temperature/humidity sensor
- BNO055 : absolute orientation sensor (9 d.o.f. inertial measurement unit)
- HC-SR04 : ultrasonic distance sensor
- PCA 9685 : 16 channel servo driver
- SSD1306 : 128x64 dot matrix OLED display panel

A detailed description of these implementations is beyond the present scope, but for illustrative purposes the subsection below briefly describes the BNO055 **Device.Box** implementation.

2.1. BNO055

The BNO055 is an inertial measurement unit (IMU), described as a system in a package comprising a microprocessor with a triaxial accelerometer, gyroscope and geo-magnetic - to provide an overall 9-axis absolute orientation sensor. The code snippet in Fig. 2.1.a is part of a Swift **Device.Box** wrapper for the BNO055 - whose public elements just define a **Skin**, with:

- an application-specific **name** supplied to the constructor **BNO055** constructor,
- no inputs (an empty list),
- a list of 5 outputs giving: the absolute orientation of the device (in Euler angles), the local direction of gravity (as a 3D Euclidean vector), and the accelerometer, gyroscope and geo-magnetic readings from which orientation and gravity are computed.

Note that, because there are no inputs, the publish method is redundant and does nothing.

Behind the scenes, the **BNO055** class just implements a timer, which periodically polls the device (using the **I2C** instance passed to the constructor) for its sensor readings.

```
public class BNO055: Device.Box{
    // static functions
    static func skin(_ n:String)->Skin{
        return Skin(n,[:],[
            "euler":Euler,
            "accelerometer":XYZ,
            "gyroscope":XYZ,
            "magnetometer":XYZ,
            "gravity":XYZ
        ])
    }
    // variables
    public let skin:Skin
    // innitialiser
    init(_ n:String,_ i2c:I2C){
        self.skin = BNO055.skin(n)
        …
    }
    // functions
    public var callback:(([Ports.ID:Event])->())?
    public func publish(_ inputs:[Ports.ID:Event]){ /* do nothing */ }
}
```

Fig. 2.1.a: BNO055 (device) class

See Figs. 2.1.2.1.c & 2.1.2.1.e for struct types **XYZ** & **Euler**

This ends the description of the **FloHW** package.