konashi

Objective-C | Android

Apps

konashi inspector (iOS)

konashi inspector (Android)

Hardware

INTRODUCTION

+ Concept
+ Architecture
+ Versions

SPECIFICATIONS

+ Supported devices
+ Appearance
+ Schematic

CORE FUNCTIONS

+ Digital
+ Analog
+ PWM
+ Communication
+ Bluetooth Low Energy
+ Event-driven

API Reference

CONSTANTS

+ Pin name
+ PIO
+ AIO
+ PWM
+ UART
+ I^2C
+ SPI
+ Function return
+ Events

BASE

+ initialize
+ find
+ findWithName
+ disconnect
+ isConnected
+ peripheralName

PROMISE

+ done
+ fail
+ then

EVENTS

+ addObserver
+ removeObserver

DIGITAL I/O (PIO)

+ pinMode
+ pinModeAll
+ pinPullup
+ pinPullupAll
+ digitalRead
+ digitalReadAll
+ digitalWrite
+ digitalWriteAll

ANALOG I/O (AIO)

+ analogReference
+ analogReadRequest
+ analogRead
+ analogWrite

PWM

+ pwmMode
+ pwmPeriod
+ pwmDuty
+ pwmLedDrive

UART

+ uartMode
+ uartBaudrate
+ uartWrite

I^2C

+ i2cMode
+ i2cStartCondition
+ i2cRestartCondition
+ i2cStopCondition
+ i2cWrite
+ i2cReadRequest
+ i2cRead

SPI

+ spiMode
+ spiWrite
+ spiReadData
+ spiReadRequest

HARDWARE CONTROL

+ reset
+ batteryLevelReadRequest
+ batteryLevelRead
+ signalStrengthReadRequest
+ signalStrengthRead

Extension Board

CONSTANTS FOR EXTENSION BOARD

+ ADC Extension Board
+ AC Drive Extension Board
+ Grove Extension Board

ADC EXTENSION BOARD

+ Abstract
+ init
+ read
+ readDiff
+ selectPowerMode

AC DRIVE EXTENSION BOARD

+ Abstract
+ init
+ onDrive
+ offDrive
+ updateDuty
+ selectFreq

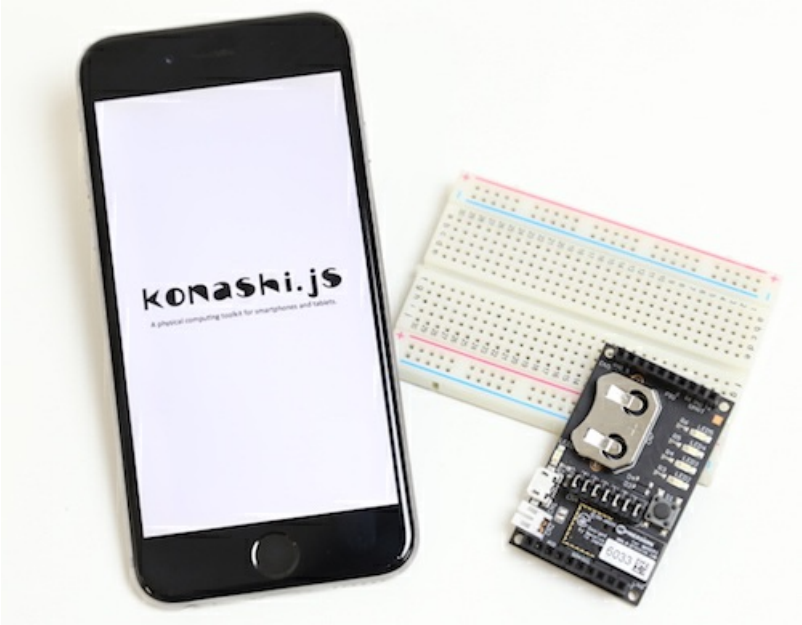
GROVE EXTENSION BOARD

+ Abstract
+ writeDigitalPort
+ readDigitalPort
+ readAnalogPort

Hardware

Introduction

Concept



既存のコンピュータの機能上の制約を超え、日常生活環境に沿った人（身体）とコンピュータとの新しいインタラクションを探る、そうすることで身の回りの世界に新しい広がりが生まれるのではないか。 フィジカル・コンピューティングはこのような考えのもと、ニューヨーク大学 Dan O'Sullivan 教授により提案(※1)されました。

フィジカル・コンピューティングの思想に沿ったインタフェースの開発を行う場合、ソフトウェアのみならずハードウェア、意匠デザインなどの幅広い要素を含めて考える必要があります。 つまり、それぞれの分野で一定のスキルが必要となるため、人とコンピュータとのインタラクションについて深く議論することは容易ではありませんでした。

このような背景から、これまでフィジカル・コンピューティングのツールキットとして「GAINER」「FUNNEL」「Arduino」などが開発、提供されました。 これらはコンピュータの入出力機能を外部に拡張する小規模なコンピュータであり、エンジニア以外にもわかりやすいように説明や開発環境が工夫されています。 モジュール化された入出力機能とその扱いやすさから、フィジカル・コンピューティングを考えるためのツールとして、現在、多くのアーティスト、デザイナ、エンジニアの間で広く普及し、その相互連携に貢献しています。

しかし現在、我々の日常生活環境における “コンピュータ” という存在は、これまでの「デスクトップ型」や「ノートブック型」の端末ではなく「スマートフォン型」や「タブレット型」の端末へと移行しているのではないでしょうか。

このような考えから、弊社ではスマートフォンやタブレットで利用可能かつ、これらの開発環境で扱うことが可能なツールキットとして “konashi” の開発をおこないました。

(※1) Tom I. and Dan O., Physical Computing: Sensing and Controlling the Physical World with Computers, Thomson, (May, 2004).

Architecture

konashiはiPhoneアプリから簡単にコントロールできるようになっています。

独自にkonashiと同じような、無線接続で簡単にコントロールできるものを実現するためには、

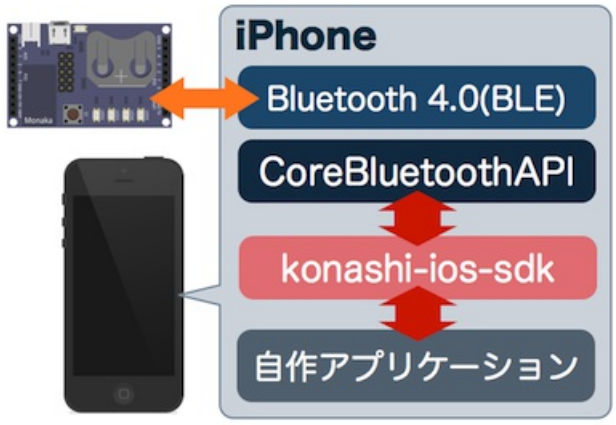
- Bluetooh Low Energy(以下BLE) のしくみの理解
- BLEモジュールを搭載したMPUの調達とBLEの通信を含めたプログラミング
- 上記MPUとBLEで通信するiPhoneアプリの開発

を行う必要があります。また、iPhoneアプリでBLEを扱うためには、CoreBluetoothAPIのハンドリングやMPUとの複雑なデータのやり取りが必要となってきます。

そのため、konashiでは、

- konashi独自のBLEのServiceやCharacteristicsの開発
- BLEモジュール+MPUのプログラミング
- iPhoneでのCoreBluetoothAPIのハンドリングを konashi-ios-sdk でラッピング

を行なっているので、konashiとiPhoneアプリの通信のハンドリングを意識することやMPUのプログラミングをせずに、iPhoneアプリからkonashiを簡単にコントロールすることができるようになっています。



また、konashiを操作するためのAPIもシンプルなものがほとんどです。たとえば、konashiのデジタルポートPIO0の出力をHIGHにするには以下のコードだけで完結します。

ObjectiveC | Android

```
[Konashi digitalWrite:PIO0 value:HIGH];
```

このコードにより、iPhoneアプリからkonashiに対して「PIO0の出力をHIGHに」という命令が送られ、konashiに内蔵されているMPUがそれを解釈しPIO0の出力をHIGHにするようになっています。

Versions

2013年1月の konashi の発売以降、konashi 同様にふるまう konashi 互換ハードウェアが複数登場しています。ここではそれらを紹介し、特徴を示します。

詳しいハードウェアのドキュメントはこちらをご覧ください。

機能の差分の詳細については、各API仕様のドキュメント内で、このように背景色を変えて説明します。

商品名	販売時期	販売者	Revision	特徴、konashiに対する機能の差分
konashi	2013年1月31日～	ユカイ工学株式会社	T1.0.0	最初の konashi
koshian	2014年9月～	株式会社マクニカ	1.0.0	PIO,PWM,A/D,D/A未対応 I2Cの一度に送受信できるバイト数が、20から16に変更 BLEのService,CharacteristicのUUIDを変更 OTAファームウェアアップデートに対応 koshianについて詳しくはこちら [www.m-pression.com/]
konashi 2	2014年12月～	ユカイ工学株式会社	2.0.0以降	PIOのピン数の変更(8->6) DAC(アナログ出力)未対応 UARTの対応baudrateを追加 UARTで複数バイトを一度に送受信を可能に I2Cで一度に送受信できるバイト数を、20から16に変更 BLEのService,CharacteristicのUUIDを変更 PWMを出せるピンがPIO0-2の3本のみに変更 PWM波形の仕様変更 OTAファームウェアアップデートに対応
konashi 3	2018年1月～	ユカイ工学株式会社	3.0.0以降	konashi2.0と同様、I2C master、SPI masterの機能が使えます PIOのピン数、PWMを出せるピン数はPIO0~7の8本とkonashi1の頃に戻りました PWM波形の仕様変更(ハードPWMからソフトPWMになり精度は低下、最小単位50us、周期とDutyの上限は2^32[us]) 電流出力DAC(アナログ出力)に対応(AIO0,1のうち同時には1本のみ出力可能) BLEのService,CharacteristicのUUIDはOTA Service（下記）以外konashi2.0のまま OTAファームウェアアップデートに対応(プロトコルはkonashi2.0と異なり、[Silicon Labs OTA Service] になります。)

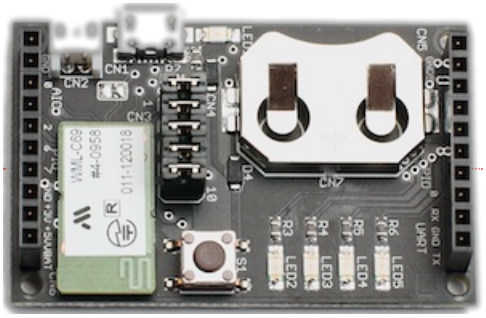
Specifications

Supported devices

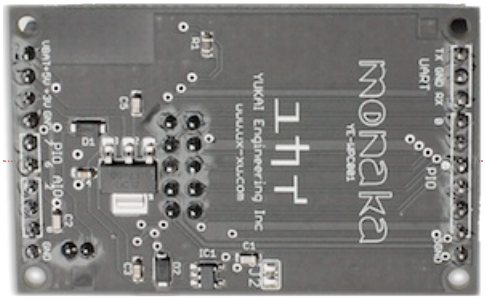
シリーズ名	機種	使用OS
iPhone	iPhone 4S, iPhone 5, iPhone 5S, iPhone 5C, iPhone 6, iPhone 6 Plus,iPhone 7,iPhone 8	iOS7.1 ～
iPad	iPad Air, iPad mini, iPad(第4世代 / 2012年11月モデル), iPad(第3世代 / 2012年3月モデル)	iOS7.1 ～
iPod touch	iPod touch(第5世代) (注) iPod touch(第４世代)には 対応しておりません 2013/02/13修正	iOS7.1 ～

Appearance

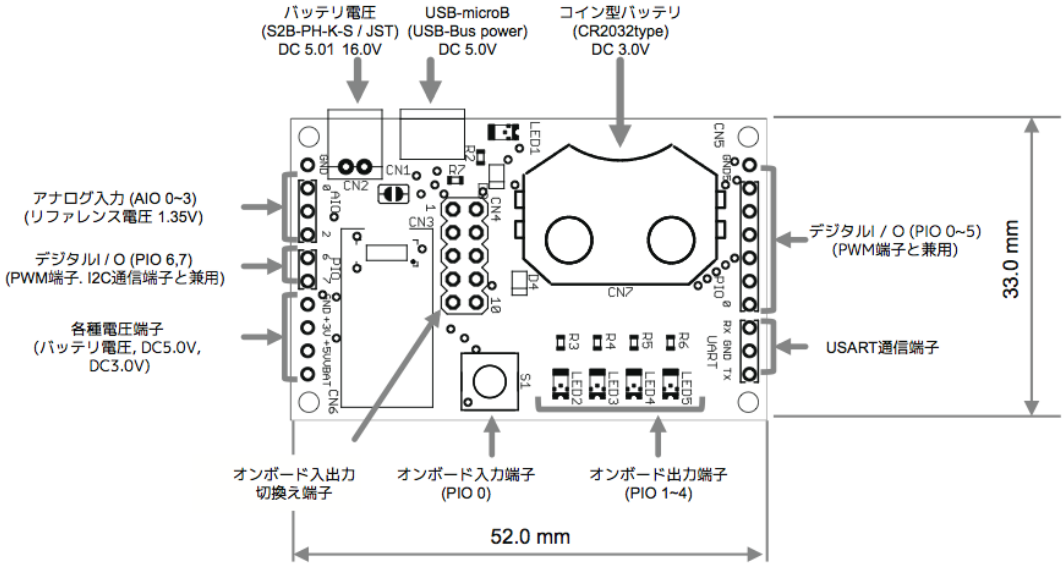
konashi 2.0



Top view

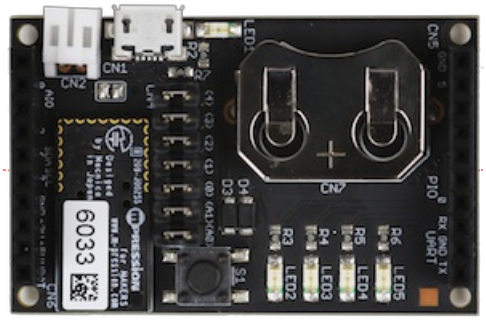


Bottom view

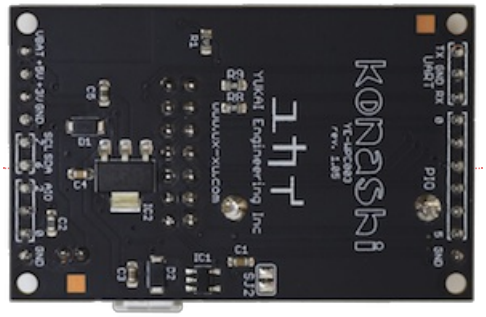


Layout

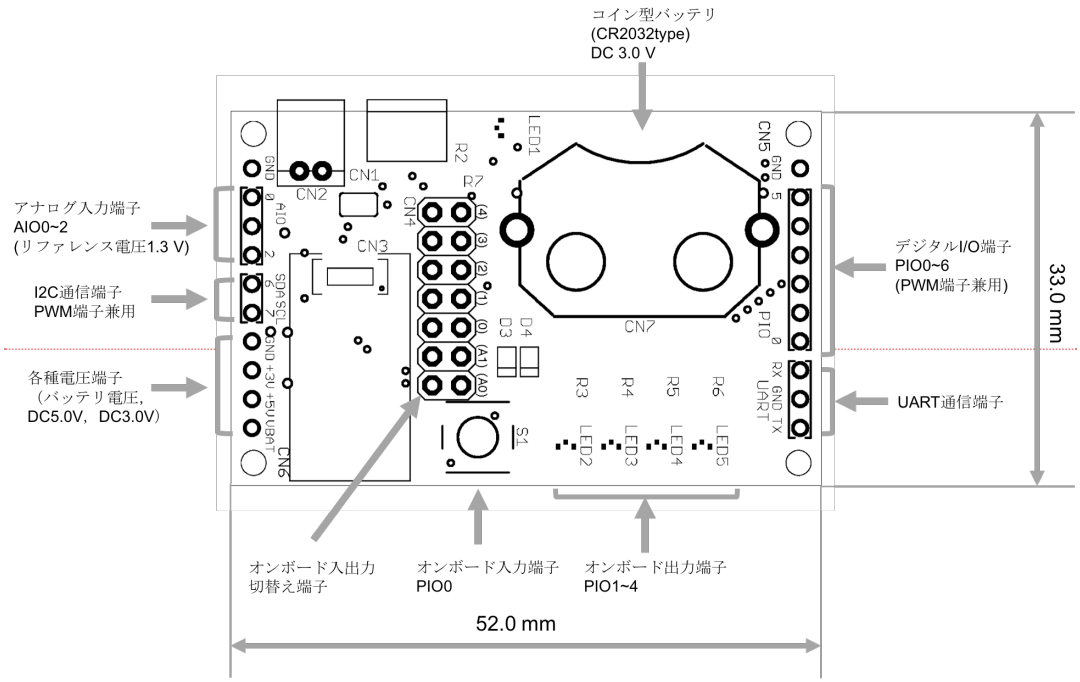
konashi 3.0



Top view



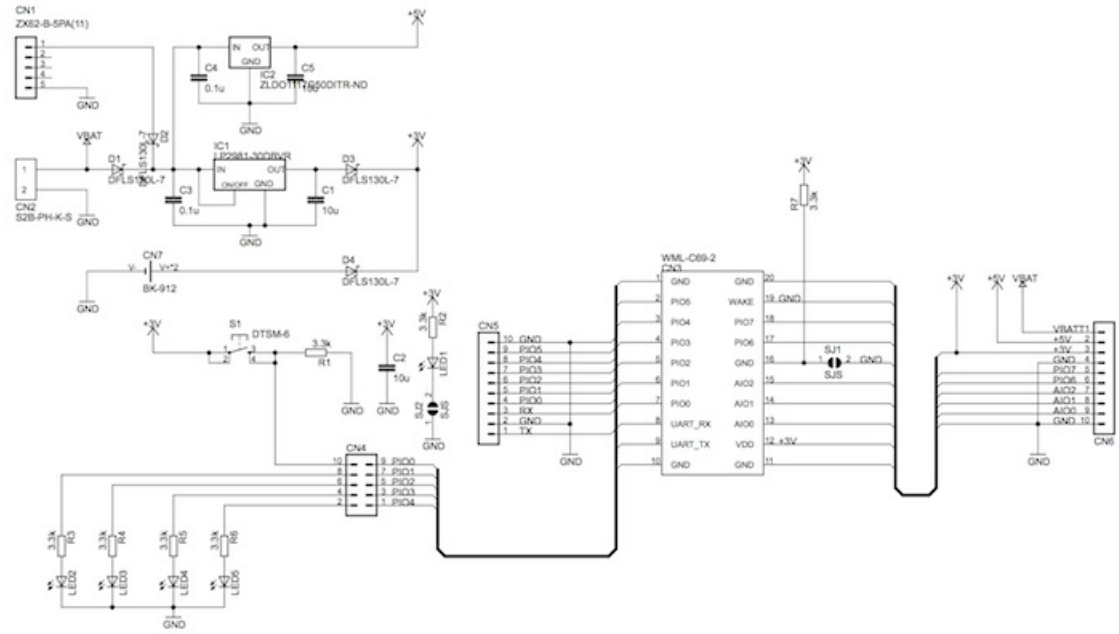
Bottom view



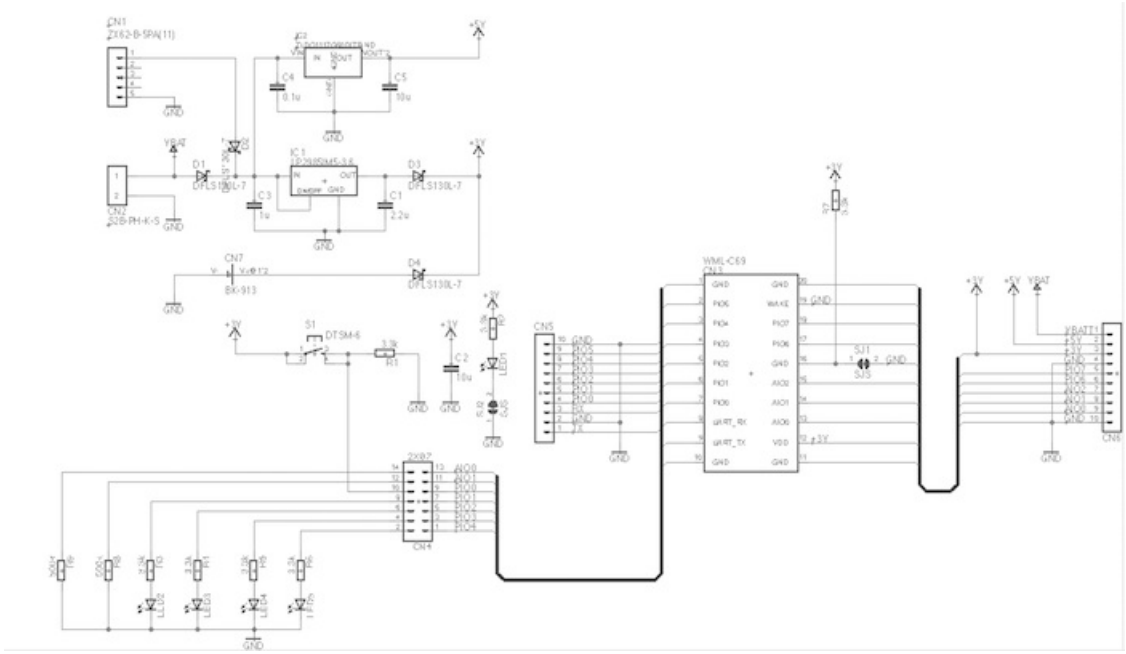
Layout

Schematic

konashi2.0



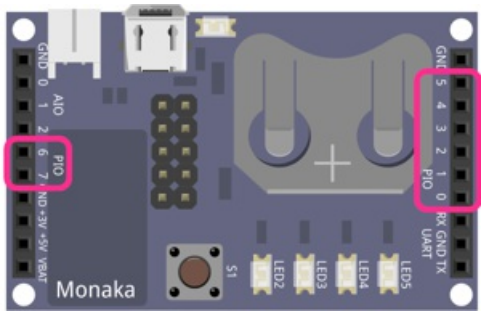
konashi3.0



konashi 2 ではIC1の電圧が 3.0 [V] で後段のダイオードで 0.3 [V] 電圧が落ちるため、konashi 3 ではIC1 電圧を 3.6 [V] にしました。センサデバイスなど繋げるICによっては電源電圧が最低でも 3.0 [V]　必要なデバイスがあるのでIC1を 3.6 [V]　にしたことで接続できるICの選択肢を広げました。

Core functions

Digital



konashiには、8つのデジタルI/Oピンが搭載されています。 デジタルI/Oピンでは、HighとLowの2つの状態を入力/出力することができます。

konashiの場合、デジタルI/Oの基準電圧は3Vですので、3Vまたは0Vの電圧を入力/出力することができます。

なお、初期状態ではすべてのデジタルI/Oは 入力 として設定されています。

デジタルI/Oピンを入力に設定した場合、pinMode, pinModeAll 関数で内部プルアップを設定することも可能です。

pinModeAll, digitalReadAll での戻り値や、pinPullupAll, digitalWriteAll での引数は、PIO0〜PIO7を8bit(1byte)として表現しています。どのビットがどのピン番号に対応しているかを以下に示します。

MSB(7bit目)　　LSB(0bit目)

PIO7	PIO6	PIO5	PIO4	PIO3	PIO2	PIO1	PIO0
I2C_SCL	I2C_SDA		LED5	LED4	LED3	LED2	S1

PIO0が0bit目(LSB)に、PIO7が7bit目(MSB)です。

koshianはPIO未対応です。
konashi 2 は PIO6, PIO7 はI2C専用になり、PIOとしては使えません。

konashi 3 は PIO0~7全て使用可能です。

また、それぞれのビットの0/1がなにを表現するかはそれぞれの関数によって異なります。

関数	bit: 0	bit: 1
pinModeAll(入出力設定)	入力設定	出力設定
pinPullupAll(プルアップ設定)	プルアップ無効	プルアップ有効
digitalWriteAll(出力の状態を設定)	LOW(0V)	HIGH(3V)
digitalReadAll(入力の状態を設定)	LOW(0V)	HIGH(3V)

例えば、PIO0(S1)を入力に、それ以外のPIOを出力に pinModeAll で設定する場合、入力=0、出力=1なので、以下のように11111110(254)と設定します。

ObjectiveC Android

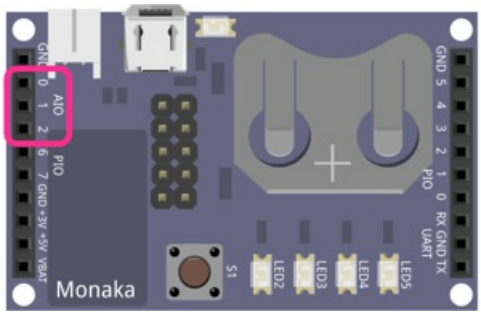
```
[Konashi pinModeAll:0b11111110]
```

また、PIO0(S1)がHIGH、それ以外がLOWの状態だった時に digitalReadAll を実行すると、Objective-Cで1(00000001)は戻り値として返ってきます。

ObjectiveC Android

```
int input = [Konashi digitalReadAll];
NSLog(@"input = %d", input);   // input = 1(00000001)
```

Analog



konashi には、3つのアナログI/Oピンが搭載されています。 アナログI/Oピンは、mV単位で入力されている電圧の値を取得したり(ADC)、指定の電圧を出力することができます(DAC)。

konashiの場合、アナログI/Oの基準電圧は1.3Vですので、0Vから1.3Vまでの電圧をmV単位で入力/出力することができます。

なお、初期上ではすべてのアナログI/Oは ADC(入力) として設定されています。

注意点として、ADC と DAC を同時に使用することはできません。ADC として3ピンを使用するか、DAC を使用するかのどちらかとなります。

また、DAC は同時に1ピンしか使用できません。他のピンをDACすると他のピンは出力がリセットされます(ADCになる)。

konashi 2では、DAC機能は未対応です。

konashi 3では、ADC,DAC機能は共に対応しています。

またkonashi 3のDAC機能について以下に説明をします。

- konashi 3のDACは電流出力型になっています。そのためベースボード上のショートピンで500kOhmのプルダウン抵抗を接続しています。AIO0,AIO1を入力として使うときに、ボード上の電流電圧変換回路が余計な場合はCN4のショートピンを外して使ってみてください。
- DACの出力電流は0.05~1.6[μA]が出力されます。
- DACの出力電圧はCN4のショートピンでショートしている場合は出力電圧は25~800[mV]が出力されます。

- DACを二つ使用することはできないですが、DACとADC２つを同時に使用することは可能です。

PWM

PWM(Pulse Width Modulation：パルス幅変調）は、ピンのON/OFFを繰り返すことでパスルを出力し、ONの時間(デューティ比)をコントロールする制御方式です。モータの回転速度やLEDの光の強さを制御するときによく使われる方式です。

konashi には、デジタルI/O(PIO)のすべてのピンをPWMモードに設定することが可能です。

konashi の PWM を使うににあたって、pwmMode, pwmPeriod, pwmDuty 関数でデューティ比や周期を決めるモード の他に、LEDの明るさを0〜100%で指定して簡単にドライブできるモード（ pwmLedDrive 関数を使用）があります。どの PWM モードにするかは、pwmMode 関数で指定できます。

konashi の PWM はソフトウェアPWMで実装されているため、短い間隔でデューティ比を変更するなど、konashi側でBLE系の処理が連続して走る状態になると、一瞬だけ、指定したデューティー比を正確に出力できなくなる場合があります。つきましては、konashiに対するBLEのアクセスを連続して行わないようにするか、デューティ比がずれることがあっても影響を受けないものを制御の対象としてください。

また、設定できる PWM の周期の最低値は `1000 [us]` です。

konashi 2 は、PIO0-PIO2の3本のみPWM出力として指定可能です。
またkonashi 2では、PWM波形の仕様が以下のとおり変更になりました。

- パルス幅は20usecの倍数（一般的なサーボモーターであれば約3.6度刻みで動かすことが出来る解像度です）
- 周期は40usec〜20,460usecが設定可能です。範囲外を指定した場合は設定可能範囲に制限されます

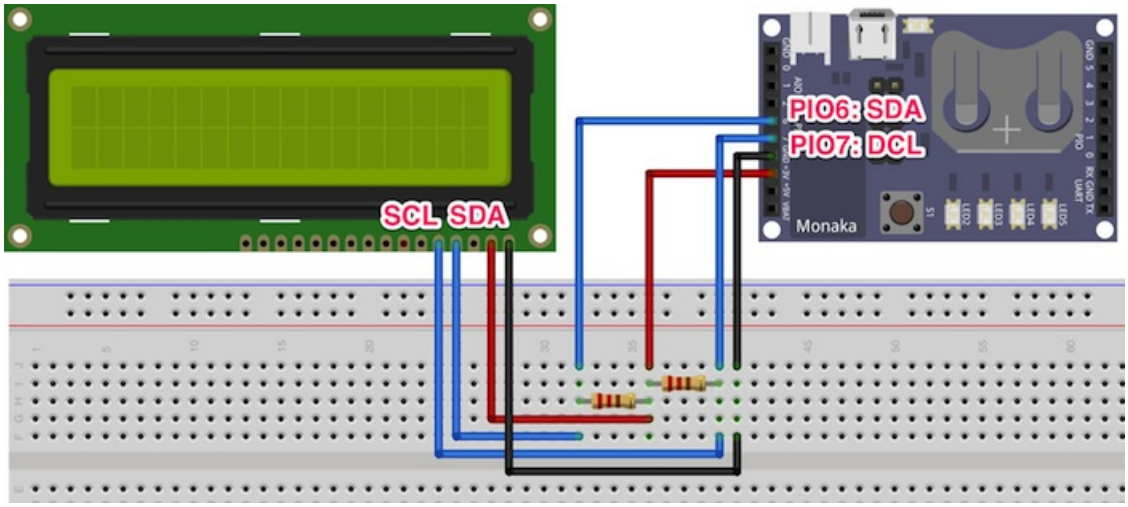
konashi 3 は、PIO0~7全てのピンがPWM出力として指定可能です。
またkonashi 3のPWM波形の仕様は以下のとおりになります。

- 周期、duty共に50[us]が最小になります。逆に最大は2^32[us](42.5分)まで設定可能です。

Communication

konashiには、他のデバイスとシリアル通信するための機能として、I²CとUARTとSPIの3種類に対応しています。

I²C



I²Cで利用する信号線は、シリアルデータ（SDA）とシリアルクロック（SCL）の2本のみです。

この通信規格は電子機器制御用のシンプルなバスシステムとして開発されたもので、規格の詳細はNXPセミコンダクター社のサイトから知ることができます。（詳細は以下の参考文献をご参照ください）

参考文献： I2Cバス仕様書 バージョン2.1 (NXPセミコンダクター社) （PDFファイル / 780KB）

konashi は、I2Cのマスタとして動作し、接続された周辺機器（スレーブ）と通信をおこないます。

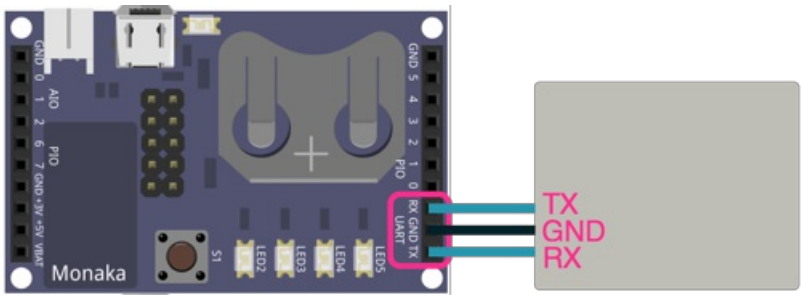
2つの信号線SDA, SCLは、konashi ではそれぞれ PIO6 (SDA)、PIO7 (SCL) が対応し ています。 上記の図のように接続し、デバイスの通信プロトコルに応じてプログラムを作成することで、I2C対応のLCDやセンサなどのデバイスと通信することができます。

注意点として、SDA と SCL には必ずプルアップ抵抗を挿入してください。

konashiライブラリ のI2C用関数 を利用することで、効率よくI2C 通信のプログラミングが可能です。

konashi ではI2Cで一度に送受信できるバイト数が 18Byte ですが、koshian、konashi 2 ,konashi 3では 16Byte に変更しました。

UART



UARTは調歩同期方式によるシリアル通信を行う機能の総称であり、konashi ではこれをもちいて RS-232(ANSI/TIA/EIA-232-F-1997) に準拠したシリアル通信をおこなうことができます。

konashi は、送信データ線（TX）と受信データ線(RX)の2本の信号線を利用して、UARTでのシリアル通信を行います。 上記の図のように接続し、プログラムを作成することでPCなどの周辺機器と簡単に通信することができます。

信号の電圧は 3V です。

設定できる通信速度は Constants / UART をご覧ください。

バージョン毎に取り得るUARTの通信速度 [bps] は以下のとおりです。

- konashi : 2400, 9600
- konashi 2 : 9600, 19200, 38400, 57600, 76800, 115200
- konashi 3 : 9600, 19200, 38400, 57600, 76800, 115200

また、konashiでは、一度に 1Byte ずつしか送受信できなかったところ、konashi 2,konashi 3では 18Byte まで一度に読み書きできるようになりました。

データを送信する場合は uartWrite を実行してください。

データを受信する場合は、`KONASHI_EVENT_UART_RX_COMPLETE` イベントを addObserver でキャッチしてください。

ObjectiveC Android

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.

    [Konashi initialize];

    [Konashi addObserver:self selector:@selector(recvUartRx) name:KONASHI_EVENT_UART_RX_COMPLETE];
}

- (void) recvUartRx
{
    NSLog(@"UartRx %d", [Konashi uartRead]);
}

- (IBAction)find:(id)sender {
    [Konashi find];
}
```

SPI

SPI通信で扱う信号線は

CS	PIO2
MISO	PIO3
MOSI	PIO4
CLK	PIO5

となっています。 Konashiのライブラリの SPI用関数 を利用することで、効率よくSPI通信のプログラムをすることが可能です。

konashi2 ではSPIを使うにはOTAが必要でしたが、 konashi3 ではOTA不要でSPIを使うことができます。

Bluetooth Low Energy

Bluetooth Low Energy(以下、BLE)はBluetooth SIG によって策定された低消費電力版Bluetoothであり、iPhone や iPad, 最新のMacBook にも搭載されるようになりました。

konashi は、このBLEを利用してiPhoneやiPadと接続されます。

konashi は 初めてのユーザにも簡単に使用していただけるように設計しているため、CPUがスリープに入っている時間が少ないなど、省電力設計になっておりません。普通のBLEデバイスよりも消費電力が大きくなっておりますのでご注意ください。

以下に、konashi のServiceやCharacteristicsのUUIDを示します。

konashi 2 ,konashi 3では ServiceやCharacteristicsのUUIDを変更しています。

Services

Name	UUID (konashi)	UUID (konashi 2,konashi 3)
Konashi Service	FF00	229BFF00-03FB-40DA-98A7-B0DEF65C2D4B

Characteristics

PIO

Name	UUID (konashi)	UUID (konashi 2,konashi 3)
PIO Setting	3000	229B3000-03FB-40DA-98A7-B0DEF65C2D4B
PIO PullUp	3001	229B3001-03FB-40DA-98A7-B0DEF65C2D4B
PIO Output	3002	229B3002-03FB-40DA-98A7-B0DEF65C2D4B
PIO Input Notification	3003	229B3003-03FB-40DA-98A7-B0DEF65C2D4B

PWM

Name	UUID (konashi)	UUID (konashi 2,konashi 3)
PWM Config	3004	229B3004-03FB-40DA-98A7-B0DEF65C2D4B
PWM Parameter	3005	229B3005-03FB-40DA-98A7-B0DEF65C2D4B
PWM Duty	3006	229B3006-03FB-40DA-98A7-B0DEF65C2D4B

Analog

Name	UUID (konashi)	UUID (konashi 2,konashi 3)
Analog Drive	3007	229B3007-03FB-40DA-98A7-B0DEF65C2D4B
Analog Read 0	3008	229B3008-03FB-40DA-98A7-B0DEF65C2D4B
Analog Read 1	3009	229B3009-03FB-40DA-98A7-B0DEF65C2D4B
Analog Read 2	300A	229B300A-03FB-40DA-98A7-B0DEF65C2D4B

I^2C

Name	UUID (konashi)	UUID (konashi 2,konashi 3)
I2C Config	300B	229B300B-03FB-40DA-98A7-B0DEF65C2D4B
I2C Start Stop	300C	229B300C-03FB-40DA-98A7-B0DEF65C2D4B
I2C Write	300D	229B300D-03FB-40DA-98A7-B0DEF65C2D4B
I2C Read Parameter	300E	229B300E-03FB-40DA-98A7-B0DEF65C2D4B
I2C Read	300F	229B300F-03FB-40DA-98A7-B0DEF65C2D4B

UART

Name	UUID (konashi)	UUID (konashi 2,konashi 3)
UART Config	3010	229B3010-03FB-40DA-98A7-B0DEF65C2D4B
UART Baud Rate	3011	229B3011-03FB-40DA-98A7-B0DEF65C2D4B
UART TX	3012	229B3012-03FB-40DA-98A7-B0DEF65C2D4B
UART RX Notification	3013	229B3013-03FB-40DA-98A7-B0DEF65C2D4B

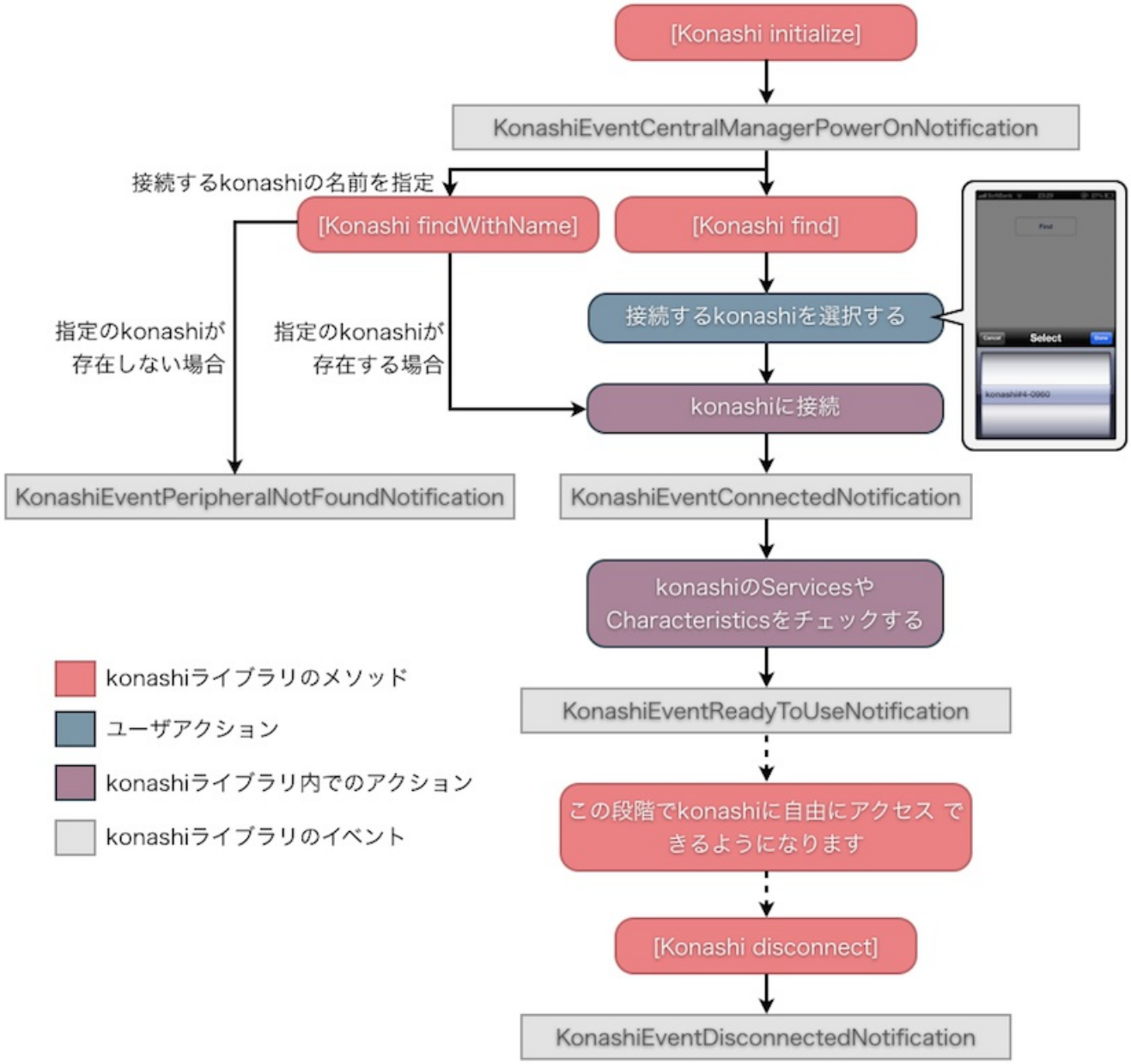
Hardware control

Name	UUID (konashi)	UUID (konashi 2,konashi 3)
Hardware Reset	3014	229B3014-03FB-40DA-98A7-B0DEF65C2D4B
Low Battery Notification	3015	229B3015-03FB-40DA-98A7-B0DEF65C2D4B

Event-driven

konashiはiPhoneとは無線で接続されているため、konashiの状態を取得するにも数msほど通信時間を要します。取得できるまで待機する場合スレッドにロックが掛かってしまうため、基本的に非同期でデータを取得することになります。そのため、取得完了時にはイベントという形で、iPhoneアプリに対して通知されます。

konashiを使うにあたっての、基本的なイベントサイクルは以下のようになります。灰色の部分がアプリを起動してから、konashiの接続を切断するまでに発行されるイベントです。



また上記とは別に、Read系のAPI実行したあとに取得完了イベントも発行されます。これらのイベントを取得するためには、addObserver という関数を使用し、イベントがあったときに実行される関数(イベントオブザーバ)を登録します。イベントの種類は Constants - Events をご覧ください。

API Reference

iOS SDKに関するより詳しいドキュメントはこちらを参照ください。

Constants

Pin name

ObjectiveC Android

KonashiDigitalIO0	0	デジタルI/Oの0ピン目
KonashiDigitalIO1	1	デジタルI/Oの1ピン目
KonashiDigitalIO2	2	デジタルI/Oの2ピン目
KonashiDigitalIO3	3	デジタルI/Oの3ピン目
KonashiDigitalIO4	4	デジタルI/Oの4ピン目
KonashiDigitalIO5	5	デジタルI/Oの5ピン目
KonashiDigitalIO6	6	デジタルI/Oの6ピン目
KonashiDigitalIO7	7	デジタルI/Oの7ピン目
KonashiS1	0	タクトスイッチ（ジャンパ をショートすることで、 デジタルI/Oの0ピン目 に接続されます）
KonashiLED2	1	赤色LED（ジャンパ をショートすることで、 デジタルI/Oの1ピン目 に接続されます）
KonashiLED3	2	赤色LED（ジャンパ をショートすることで、 デジタルI/Oの2ピン目 に接続されます）
KonashiLED4	3	赤色LED（ジャンパ をショートすることで、 デジタルI/Oの3ピン目 に接続されます）
KonashiLED5	4	赤色LED（ジャンパ をショートすることで、 デジタルI/Oの4ピン目 に接続されます）
KonashiAnalogIO0	0	アナログI/Oの0ピン目
KonashiAnalogIO1	1	アナログI/Oの1ピン目
KonashiAnalogIO2	2	アナログI/Oの2ピン目
KonashiI2C_SDA	6	I^2CのSDAのピン(デジタルI/Oの6ピン目)
KonashiI2C_SCL	7	I^2CのSCLのピン(デジタルI/Oの7ピン目)

PIO

ObjectiveC Android

KonashiLevelHigh	1	ピンの出力をHIGH(3V)にする
KonashiLevelLow	0	ピンの出力をLOW(0V)にする
KonashiPinModeOutput	1	ピンの入出力設定を出力に
KonashiPinModeInput	0	ピンの入出力設定を入力に
KonashiPinModePullup	1	ピンのプルアップ設定をON
KonashiPinModeNoPulls	0	ピンのプルアップ設定をOFF

AIO

ObjectiveC Android

Konashi.analogReference	1300	アナログ入出力の基準電圧 1300mV

PWM

ObjectiveC Android

KonashiPWMModeDisable	0	指定したPIOをPWMとして使用しない(デジタルI/Oとして使用)
KonashiPWMModeEnable	1	指定したPIOをPWMとして使用する
KonashiPWMModeEnableLED	2	指定したPIOをLEDモードとしてPWMとして使用する
KonashiLEDPeriod	10000	LEDモード時のPWMの周期は10ms

UART

ObjectiveC Android

KonashiUartModeDisable	0	UART無効
KonashiUartModeEnable	1	UART有効
KonashiUartBaudrateRate2K4	0x000a	2400bps
KonashiUartBaudrateRate9K6	0x0028	9600bps
KonashiUartBaudrateRate19K2	0x0050	19200bps
KonashiUartBaudrateRate38K4	0x00a0	38400pbs
KonashiUartBaudrateRate57K6	0x00f0	57600pbs
KonashiUartBaudrateRate76K8	0x0140	76800pbs
KonashiUartBaudrateRate115K2	0x01e0	115200pbs

I^2C

ObjectiveC Android

KonashiI2CModeDisable	0	I^2Cを無効にする
KonashiI2CModeEnable	1	I^2Cを有効にする(100kbpsモードがデフォルト)
KonashiI2CModeEnable100K	1	100kbpsモードでI2Cを有効にする
KonashiI2CModeEnable400K	2	400kbpsモードでI2Cを有効にする
KonashiI2CConditionStop	0	ストップコンディション
KonashiI2CConditionStart	1	スタートコンディション
KonashiI2CConditionRestart	2	リスタートコンディション

SPI

ObjectiveC | Android

KonashiSPIModeEnableCPOL0CPHA0	0	クロックを正論理にし、0から1に切り替わるタイミングでデータを取り込む
KonashiSPIModeEnableCPOL0CPHA1	1	クロックを正論理にし、1から0に切り替わるタイミングでデータを取り込む
KonashiSPIModeEnableCPOL1CPHA0	2	クロックを負論理にし、0から1に切り替わるタイミングでデータを取り込む
KonashiSPIModeEnableCPOL1CPHA1	3	クロックを負論理にし、1から0に切り替わるタイミングでデータを取り込む
KonashiSPIModeDisable	-1	SPI通信を無効にする
KonashiSPISpeed200K	20	通信速度を200kbsに設定する
KonashiSPISpeed500K	50	通信速度を500kbsに設定する
KonashiSPISpeed1M	100	通信速度を1Mbsに設定する
KonashiSPISpeed2M/td>	200	通信速度を2Mbsに設定する
KonashiSPISpeed3M	300	通信速度を3Mbsに設定する
KonashiSPISpeed6M	600	通信速度を6Mbsに設定する
KonashiSPIBitOrderLSBFirst	0	LSBからデータを転送する
KonashiSPIBitOrderMSBFirst	1	MSBからデータを転送する

Function return

ObjectiveC | Android

KonashiResultSuccess	0	成功時
KonashiResultFailure	1	失敗時

Events

ObjectiveC | Android

KonashiEventCentralManagerPowerOnNotification	CoreBluetoothのセントラルマネージャが起動した時
KonashiEventPeripheralNotFoundNotification	findWithNameで指定した名前のkonashiが見つからなかった時
KonashiEventPeripheralFoundNotification	findWithNameで指定した名前のkonashiが見つかった時
KonashiEventConnectedNotification	konashiに接続した時(まだこの時はkonashiが使える状態ではありません)
KonashiEventDisconnectedNotification	konashiとの接続を切断した時
KonashiEventReadyToUseNotification	konashiに接続完了した時(この時からkonashiにアクセスできるようになります)
KonashiEventDigitalIODidUpdateNotification	PIOの入力の状態が変化した時
KonashiEventAnalogIODidUpdateNotification	AIOのどれかのピンの電圧が取得できた時
KonashiEventAnalogIO0DidUpdateNotification	AIO0の電圧が取得できた時
KonashiEventAnalogIO1DidUpdateNotification	AIO1の電圧が取得できた時
KonashiEventAnalogIO2DidUpdateNotification	AIO2の電圧が取得できた時
KonashiEventI2CReadCompleteNotification	I^2Cからデータを受信した時
KonashiEventUartRxCompleteNotification	UARTのRxからデータを受信した時
KonashiEventSpiWriteCompleteHandler	SPI経由でのデータ書き込み完了時に呼び出されます。呼びだされた瞬間からSPIモジュールから受け取るデータを取得することができます。
KonashiEventSpiReadCompleteHandler	spiReadRequest メソッドを用いてデータを受信した時に呼び出されます。
KonashiEventBatteryLevelDidUpdateNotification	konashiのバッテリーのレベルを取得できた時
KonashiEventSignalStrengthDidUpdateNotification	konashiの電波強度を取得できた時

Base

initialize

Description

konashiの初期化を行います。

一番最初に表示されるViewControllerのviewDidLoadなど、konashiを使う前に必ず `initialize` をしてください。

Syntax

```
[Konashi initialize];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess` (0), 失敗時: `KonashiResultFailure` (-1)

find

ObjectiveC Android

Description

iPhone周辺のkonashiを探します。 この関数を実行した後、周りにあるkonashiのリストが出現します。リストに列挙されているkonashiのひとつをクリックすると、konashiに自動的に接続されます。その後、KonashiEventConnectedNotification KonashiEventReadyToUseNotificationのイベントが発火するので、事前にこれらのイベントを addObserver でキャッチできるようにしておいてください。本来、KonashiEventCentralManagerPowerOnNotification のイベント以前に find を実行しても無効ですが、この場合に限り、 KonashiEventCentralManagerPowerOnNotificationのイベント後に自動的に find が遅延実行されるように調整されています。

Syntax

```
[Konashi find];
```

Parametar

なし

Returns

成功時: `KonashiResultSuccess` (0), 失敗時: `KonashiResultFailure` (-1)

findWithName

ObjectiveC Android

Description

konashiの名前を指定して接続します。 find の場合はkonashiのリストが出現しますが、findWithName を実行した場合はリストが出ずに自動的に接続されます。 名前に関しては、 find を実行することによって下から出現するリストでリストアップされる `konashi#4-0452` などの文字列です。konashi#*-**** の*部分の数字は、konashiの緑色チップのシール上に記載されている番号と同じです。 もし、指定した名前が見つからない場合は KonashiEventPeripheralNotFoundNotificationが発火されます。 本来、KonashiEventCentralManagerPowerOnNotification のイベント以前に `findWithName` を実行しても無効ですが、この場合に限り、KonashiEventCentralManagerPowerOnNotification のイベント後に自動的に `findWithName` が遅延実行されるように調整されています。

Syntax

```
[Konashi findWithName:(NSString *)name];
```

Parameters

name	NSString*	接続したいkonashiの名前。例："konashi#4-0452"

Returns

成功時: `KonashiResultSuccess` (0), 失敗時: `KonashiResultFailure` (-1)

Examples

konashi#4-0452 のkonashiを探して接続する

```
[Konashi findWithName:@"konashi#4-0452"];
```

disconnect

Description

konashiとの接続を解除します。

> **注意**
> iOS6.1より古いiOSの場合、Core Bluetooth APIにバグがあり、この関数は正常に動作しません。iOS6.1以上では正しく動作します。

Syntax

ObjectiveC | Android

```
[Konashi disconnect];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess` (0), 失敗時: `KonashiResultFailure` (-1)

isConnected

Description

konashiと接続中かを返します。

KonashiEventConnectedNotification のイベントが発火するタイミングで `TRUE` となります。それ以前は `FALSE` です。

ObjectiveC | Android

Syntax

```
[Konashi isConnected];
```

Parameters

なし

Returns

BOOL

peripheralName

ObjectiveC | Android

Description

接続中のkonashiの名前を返します。 konashiに接続していない状態で peripheralName を実行すると空文字 `@""` が返ります。

Syntax

```
[Konashi peripheralName];
```

Parameters

なし

Returns

`NSString*` (例: konashi#4-0452)

Example

```
NSString* name = [Konashi peripheralName];
NSLog(name);
```

Events

addObserver(on)/addListener

ObjectiveC Android

Description

konashiに関するイベントをキャッチすることができます。 konashiとiPhoneは BLEで繋がっているため、konashiの状態やピンの状態は非同期で取得することになります。 たとえば、AIOピンの電圧を取得するには、analogReadRequestでkonashiにリクエストを送り、KonashiEventAnalogIODidUpdateNotification という取得完了イベントを受信した後にAIOピンの電圧を参照できるようになります。どのようなイベントがあるかは Constants / Eventsを参照してください。

Syntax

```
[Konashi addObserver:(id)notificationObserver selector:(SEL)notificationSelector name:(NSString*)notificationName];
```

Parameters

notificationObserver	id	オブザーバを指定します
selector	SEL	イベント発火時に呼び出される関数を指定します
name	NSString*	イベント名を指定します。詳細は Constants - Events を参照してください。

Returns

成功時: `KonashiResultSuccess` (0), 失敗時: `KonashiResultFailure` (-1)

Example

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [Konashi initialize];
    [Konashi addObserver:self selector:@selector(konashiConnected) name:KonashiEventConnectedNotification];
}

- (void) konashiConnected
{
    NSLog(@"CONNECTED");
}
```

removeObserver/removeListener

ObjectiveC | Android

Description

addObserverで行ったkonashiイベントオブザーバを削除します。

Syntax

```
[Konashi addObserver:(id)notificationObserver selector:(SEL)notificationSelector name:(NSString*)notificationName];
```

Parameters

notificationObserver	id	オブザーバを指定します
selector	SEL	イベント発火時に呼び出される関数を指定します
name	NSString*	イベント名を指定します。詳細は Constants - Events を参照してください。

Returns

成功時: `KonashiResultSuccess` (0), 失敗時: `KonashiResultFailure` (-1)

Example

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [Konashi initialize];
    [Konashi addObserver:self selector:@selector(konashiConnected) name:KonashiEventConnectedNotification];
}

- (void) konashiConnected
{
    NSLog(@"CONNECTED");
}
```

Android Promise

done

Description

PromiseオブジェクトのActionが成功したときに実行するcallbackの設定を行います。 詳細はjdeferred/jdeferred · GitHubをご覧ください。

Syntax

```
//pmはPromiseオブジェクト
pm.done((DoneCallback<T>)callback);
```

Parameters

callback	DoneCallback	設定するcallback。ジェネリクスTは受け取る結果の型、すなわちPromiseオブジェクトのジェネリクスの1番目の型

Returns

`Promise<T, BletiaException, Object>`

Example

LED2をOUTPUTにする。成功した場合Succeededと表示する。

```
mKonashiManager.pinMode(Konashi.LED2, Konashi.OUTPUT).done(new DoneCallback<BluetoothGattCharacteristic>() {
    @Override
    public void onDone(BluetoothGattCharacteristic result) {
        Log.d("Konashi", "Succeeded");
    }
});
```

fail

Description

PromiseオブジェクトのActionが失敗したときに実行するcallbackの設定を行います。 詳細はjdeferred/jdeferred · GitHubをご覧ください。

Syntax

```
//pmはPromiseオブジェクト
pm.fail((FailCallback<T>)callback);
```

Parameters

callback	FailCallback	設定するcallback。ジェネリクスTは受け取る結果の型、すなわちPromiseオブジェクトのジェネリクスの1番目の型

Returns

`Promise<T, BletiaException, Object>`

Example

LED2をOUTPUTにする。失敗した場合Failedと表示する。

```
mKonashiManager.pinMode(Konashi.LED2, Konashi.OUTPUT).fail(new FailCallback<BluetoothGattCharacteristic>() {
    @Override
    public void onFail(BluetoothGattCharacteristic result) {
        Log.d("Konashi", "Failed");
    }
});
```

then

Description

PromiseオブジェクトのActionが成功/失敗したときに実行するcallbackの設定を行います。 詳細はjdeferred/jdeferred · GitHubをご覧ください。

Syntax

```
//pmはPromiseオブジェクト
pm.then((DoneCallback<T>)callback1);
pm.then((DoneCallback<T>)callback1, (FailCallback<T>)callback2);
```

```
pm.then((DonePipe)pipe);
```

Parameters

callback1	DoneCallback	成功した際に呼ばれるcallback。ジェネリクスTは受け取る結果の型、すなわちPromiseオブジェクトのジェネリクスの1番目の型
callback2	FailCallback	(Optional)失敗した際に呼ばれるcallback。ジェネリクスTは受け取る結果の型、すなわちPromiseオブジェクトのジェネリクスの1番目の型
pipe	DonePipe	設定するpipe

Returns

`Promise<T, BletiaException, Object>`

Example

LED2をOUTPUTにする。成功した場合Succeededと表示し失敗した場合Failedと表示する。

```
mKonashiManager.pinMode(Konashi.LED2, Konashi.OUTPUT).then(new DoneCallback<BluetoothGattCharacteristic>() {
    @Override
    public void onDone(BluetoothGattCharacteristic result) {
        Log.d("Konashi", "Succeeded");
    }
}, new FailCallback<BluetoothGattCharacteristic>() {
    @Override
    public void onFail(BluetoothGattCharacteristic result) {
        Log.d("Konashi", "Failed");
    }
});
```

Digital I/O (PIO)

pinMode

Description

PIOのピンを入力として使うか、出力として使うかの設定を行います。詳細は Core functions / Digital をご覧ください。

ObjectiveC | Android

Syntax

```
[Konashi pinMode:(int)pin mode:(int)mode];
```

Parameters

pin	int	設定するPIOのピン名。設定可能なピン名は Constants / Pin name をご覧ください。
mode	int	ピンに設定するモード。`KonashiPinModeInput` か `KonashiPinModeOutput` が設定できます。詳細は Constants / PIO をご覧ください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

LED2をOUTPUTにする

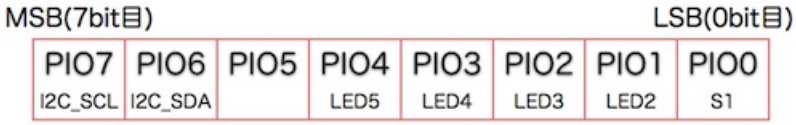
```
[Konashi pinMode:LED2 mode:OUTPUT];
```

pinModeAll

ObjectiveC | Android

Description

PIOの特定のピンの出力状態を設定します。 この関数での戻り値は、PIO0〜PIO7の入力状態が8bit(1byte)で表現されます。bitとピンの対応は以下です。



それぞれのビットで、0は入力を、1は出力を表現しています。 例えばこの関数で、PIO0(S1)を入力に、それ以外のPIOを出力に設定する場合、入力=0、出力=1なので、以下のように11111110(254)と設定します。

```
[Konashi pinModeAll:0b11111110];
```

詳細は Core functions / Digital をご覧ください。

Syntax

```
[Konashi pinModeAll:(int)mode];
```

Parameters

mode	int	PIO0 〜 PIO7 の計8ピンの設定。`OUTPUT` を1、`INPUT` を0として 0x00 〜 0xFF を指定してください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

PIOすべてのピンをOUTPUTにする

```
[Konashi pinModeAll:0xFF];
```

pinPullup

Description

PIOのピンをプルアップするかの設定を行います。

初期状態では、PIOはプルアップされていません。詳細は Core Functions / Digital の項をご覧ください。

ObjectiveC | Android

Syntax

```
[Konashi pinPullup:(int)pin mode:(int)mode];
```

Parameters

pin	int	設定するPIOのピン名。設定可能なピン名は Constants / Pin name をご覧ください。
mode	int	ピンをプルアップするかの設定。`KonashiPinModePullup` か `KonashiPinModeNoPulls` が設定できます。詳細は Constants / PIO 定数をご覧ください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

PIO7をプルアップする

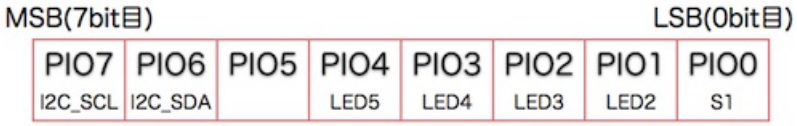
```
[Konashi pinPullup:PIO7 mode:PULLUP];
```

pinPullupAll

ObjectiveC | Android

Description

PIOの特定のピンの出力状態を設定します。 この関数での戻り値は、PIO0〜PIO7の入力状態が8bit(1byte)で表現されます。bitとピンの対応は以下です。



それぞれのビットでは、プルアップ無効を0、プルアップ有効を1として表現します。 例えばこの関数で、PIO0(S1)をプルアップし、それ以外はプルアップ無効にする場合、以下のように00000001(1)と設定します。

```
[Konashi pinPullupAll:0b00000001];
```

詳細は Core functions / Digital をご覧ください。

Syntax

```
[Konashi pinPullupAll:(int)mode];
```

Parameters

pin	int	設定するPIOのピン名。設定可能なピン名は Constants / Pin name をご覧ください。
mode	int	PIO0 〜 PIO7 の計8ピンのプルアップの設定。0x00 〜 0xFF を指定してください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

PIOのすべてのピンをプルアップする

```
[Konashi pinPullupAll:0xFF];  // 0b11111111, 255
```

digitalRead

Description

PIOの特定のピンの入力状態を取得します。

ピンの入力状態を取得する前に、必ず pinMode, pinModeAll でピンのモードを入力にしておいてください。出力モードの場合は正しい入力状態を取得することはできません。

詳細は Core functions / Digital をご覧ください。

ObjectiveC Android

Syntax

```
[Konashi digitalRead:(int)pin];
```

Parameters

pin	int	PIOのピン名。指定可能なピン名は Constants / Pin name をご覧ください。

Returns

`KonashiLevelHigh` もしくは `KonashiLevelLow`

Example

S1の入力の状態を取得する

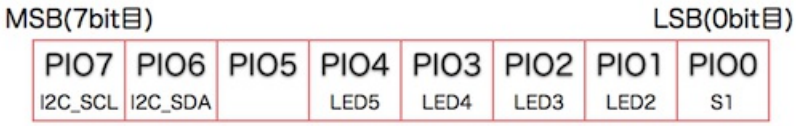
```
[Konashi digitalRead:S1];
```

digitalReadAll

ObjectiveC | Android

Description

PIOのすべてのピンの状態を取得します。 この関数での戻り値は、PIO0〜PIO7の入力状態が8bit(1byte)で表現されます。bitとピンの対応は以下です。



それぞれのビットで、0はLOW(0V)を、1はHIGH(3V)を表現しています。 例えば、PIO0(S1)がHIGH、それ以外がLOWの状態だった時にこの関数を実行すると、00000001(1)が戻り値として返ってきます。

```
int input = [Konashi digitalReadAll];
NSLog(@"input = %d", input);   // input = 1
```

詳細は Core functions / Digital をご覧ください。

Syntax

```
[Konashi digitalReadAll];
```

Parameters

なし

Returns

PIOの8ピンの入力情報(0x00〜0xFF)

Example

PIOのすべてのピンの入力状態を取得する。

```
[Konashi digitalReadAll];
```

digitalWrite

Description

PIOの特定のピンの出力状態を設定します。

詳細は Core functions / Digital をご覧ください。

ObjectiveC | Android

Syntax

```
[Konashi digitalWrite:(int)pin value:(int)value];
```

Parameters

pin	int	PIOのピン名。指定可能なピン名は Constants / Pin name をご覧ください。
value	int	設定するPIOの出力状態。`KonashiLevelHigh` もしくは `KonashiLevelLow` が指定可能です。詳細は Constants / PIO をご覧ください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

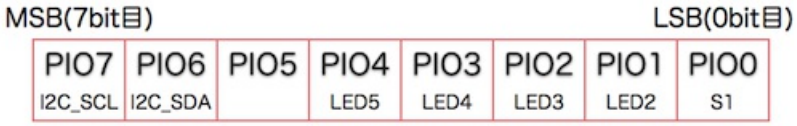
```
[Konashi digitalWrite:LED2 value:HIGH];
```

digitalWriteAll

ObjectiveC | Android

Description

PIOの特定のピンの出力状態を設定します。 この関数での戻り値は、PIO0〜PIO7の入力状態が8bit(1byte)で表現されます。bitとピンの対応は以下です。



それぞれのビットで、0はLOW(0V)を、1はHIGH(3V)を表現しています。 例えば、この関数でPIO3(LED4)をHIGH、それ以外をLOWの状態にする場合、00001000(8)を引数に設定します。

```
[Konashi digitalWriteAll:0b00001000];
```

詳細は Core functions / Digital をご覧ください。

Syntax

```
[Konashi digitalReadAll];
```

Parameters

value	int	PIO0〜PIO7の出力に設定する値。 0x00 〜 0xFF が設定可能です。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

PIOのすべてのピンをHIGHにする

```
[Konashi digitalWriteAll:0xFF];
```

Analog I/O (AIO)

analogReference

Description

アナログ入出力の基準電圧を返します。

Syntax

ObjectiveC | Android

```
[Konashi analogReference];
```

Parameters

なし

Returns

1300 が返ります(mV)。

analogReadRequest

Description

AIO の指定のピンの入力電圧を取得するリクエストを konashi に送ります。

この関数は konashi にリクエストを送るものなので、実際に値を取得するには、`KonashiEventAnalogIODidUpdateNotification` もしくは Constants / Events に定義されている入力電圧取得完了イベントを addObserver でキャッチした後、analogRead で値を取得できます。

アナログの機能に関しては、 Core functions / Analog をご覧ください。

Syntax

ObjectiveC

```
[Konashi analogReadRequest:(int)pin];
```

Parameters

ObjectiveC

pin	int	AIOのピン名。指定可能なピン名は `KonashiAnalogIO0` , `KonashiAnalogIO1` , `KonashiAnalogIO2` です。詳細は Constants / Pin name をご覧ください。

Returns

成功時: `KonashiResultSuccess` , 失敗時: `KonashiResultFailure`

Example

AIO0の入力電圧を取得する

ObjectiveC

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [Konashi initialize];

    [Konashi addObserver:self selector:@selector(konashiReady) name:KONASHI_EVENT_READY];
    [Konashi addObserver:self selector:@selector(readAio0) name:KONASHI_EVENT_UPDATE_ANALOG_VALUE_AIO0];
}

// konashiを探すボタンタップのアクション
- (IBAction)findKonashi:(id)sender {
    [Konashi find];
}

// AIO0電圧取得ボタンタップのアクション
- (IBAction)requestReadAio0:(id)sender {
    [Konashi analogReadRequest:AIO0];
}

- (void) readAio0
{
    NSLog(@"READ_AIO0: %d", [Konashi analogRead:AIO0]);
}
```

analogRead

ObjectiveC Android

Description

AIO の指定のピンの入力電圧を取得します。この関数で取得できる値は前回の analogReadRequest 時に取得した電圧です。konashiのAIOピンの電圧を取得したい場合は、まずで analogReadRequest konashi に取得リクエストを送り、

```
KonashiEventAnalogIODidUpdateNotification
```

もしくは Constants / Events に定義されている入力電圧取得完了イベントを addObserverでキャッチした後、 この関数で値を取得できます。 アナログの機能に関しては、 Core functions / Analogをご覧ください。

Syntax

```
[Konashi analogRead:(int)pin];
```

Parameters

pin	int	AIOのピン名。指定可能なピン名は `KonashiAnalogIO0` , `KonashiAnalogIO1` , `KonashiAnalogIO2` です。詳細は Constants / Pin name をご覧ください。

Returns

0 〜 1300 までの、mV単位の値が返ります。

Examples

```
- (void)viewDidLoad
      {
      [super viewDidLoad];

      [Konashi initialize];

      [Konashi addObserver:self selector:@selector(konashiReady) name:KONASHI_EVENT_READY];
      [Konashi addObserver:self selector:@selector(readAio0) name:KONASHI_EVENT_UPDATE_ANALOG_VALUE_AIO0];
      }

      // konashiを探すボタンタップのアクション
      - (IBAction)findKonashi:(id)sender {
      [Konashi find];
      }

      // AIO0電圧取得ボタンタップのアクション
      - (IBAction)requestReadAio0:(id)sender {
      [Konashi analogReadRequest:AIO0];
      }

      - (void) readAio0
      {
      NSLog(@"READ_AIO0: %d", [Konashi analogRead:AIO0]);
      }
```

analogWrite

ObjectiveC Android

Description

AIO の指定のピンに任意の電圧を出力します。 指定できる最大の電圧は 1300 [mV] です。 アナログの機能に関しては、 Core functions / Analogをご覧ください。

Syntax

```
[Konashi analogWrite:(int)pin milliVolt:(int)milliVolt];
```

Parameters

pin	int	AIOのピン名。指定可能なピン名は `KonashiAnalogIO0` , `KonashiAnalogIO1` , `KonashiAnalogIO2` です。詳細は Constants / Pin name をご覧ください。
milliVolt	int	設定する電圧をmVで指定します。0 〜 1300 まで設定可能です。

Returns

成功時: `KonashiResultSuccess` (0), 失敗時: `KonashiResultFailure` (-1)

Example

```
[Konashi analogWrite:AIO0 milliVolt:1000];
```

PWM

pwmMode

Description

PIO の指定のピンを PWM として使用する/しないかを設定します。

PIO のいずれのピンも PWMモード に設定できます。

`KonashiPWMModeEnable` モードを指定する場合は、事前に pwmPeriod, pwmDuty で周期とONになる時間を指定してください。

PWM の詳細は Core functions / PWM をご覧ください。

ObjectiveC Android

Syntax

```
[Konashi pwmMode:(int)pin mode:(int)mode];
```

Parameters

pin	int	PWMモードの設定をするPIOのピン名。`KonashiDigitalIO0` 〜 `KonashiDigitalIO7` が設定可能です。
mode	int	設定するPWMのモード。`KonashiPWMModeDisable`, `KonashiPWMModeEnable`, `KonashiPWMModeEnableLED` が設定できます。詳細は Constants / PWM をご覧ください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

LED2を周期10ms、デューティ5msのPWMにする。

```
[Konashi pwmMode:LED2 mode:KONASHI_PWM_ENABLE];
[Konashi pwmPeriod:LED2 period:10000];
[Konashi pwmDuty:LED2 duty:5000];
```

pwmPeriod

Description

指定のピンのPWM周期を設定します。

周期の単位はマイクロ秒(us)で指定してください。

PWM の詳細は Core functions / PWM をご覧ください。

ObjectiveC | Android

Syntax

```
[Konashi pwmPeriod:(int)pin period:(unsigned int)period];
```

Parameters

pin	int	PIOのピン名。`KonashiDigitalIO0` 〜 `KonashiDigitalIO7` が設定可能です。
period	unsigned int	周期。単位はマイクロ秒(us)で32bitで指定してください。最大2^{32}us = 71.5分です。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

LED2を周期10msにする。

```
[Konashi pwmPeriod:LED2 period:10000];
```

pwmDuty

Description

指定のピンのPWMのデューティ(ONになっている時間)を設定します。

単位はマイクロ秒(us)で指定してください。

PWM の詳細は Core functions / PWM をご覧ください。

ObjectiveC Android

Syntax

```
[Konashi pwmDuty:(int)pin duty:(unsigned int)duty];
```

Parameters

pin	int	PIOのピン名。`KonashiDigitalIO0` 〜 `KonashiDigitalIO7` が設定可能です。
duty	unsigned int	デューティ。単位はマイクロ秒(us)で32bitで指定してください。最大2^{32}us = 71.5分です。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

LED2のデューティを5msにセットする

```
[Konashi pwmDuty:LED2 duty:5000];
```

pwmLedDrive

Description

指定のピンのLEDの明るさを0%〜100%で指定します。

`pwmLedDrive` 関数を使うには pwmMode で `KonashiPWMModeEnableLED` を指定してください。

PWM の詳細は Core functions / PWM をご覧ください。

ObjectiveC Android

Syntax

```
mKonashiManager.pwmLedDrive((int)pin, (double|float)dutyRatio);
```

Parameters

pin	int	PIOのピン名。`KonashiDigitalIO0` 〜 `KonashiDigitalIO7` が設定可能です。
ratio	int	LEDの明るさ。0〜100 をしてしてください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

LED2の明るさを30%にする

```
[Konashi pwmMode:LED2 mode:KONASHI_PWM_ENABLE_LED_MODE];
[Konashi pwmLedDrive:LED2 dutyRatio:30];
```

UART

uartMode

Description

UART の有効/無効を設定します。

有効にする前に、uartBaudrate でボーレートを設定しておいてください。

UART の詳細は Core functions / Communication - UART をご覧ください。

ObjectiveC | Android

Syntax

```
[Konashi uartMode:(int)mode];
```

Parameters

mode	int	設定するUARTのモード。`KonashiUartModeDisable`, `KonashiUartModeEnable` が設定できます。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

UARTを有効にする

```
[Konashi uartMode:KONASHI_UART_ENABLE];
```

uartBaudrate

Description

UART の通信速度を設定します。

UART の詳細は Core functions / Communication - UART をご覧ください。

ObjectiveC Android

Syntax

```
[Konashi uartBaudrate:(int)baudrate];
```

Parameters

baudrate	int	UARTの通信速度。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

UARTの通信速度を9600bpsにする

```
[Konashi uartBaudrate:KONASHI_UART_RATE_9K6];
```

uartWrite

Description

UART でデータを1バイト送信します。

UART の詳細は Core functions / Communication - UART をご覧ください。

ObjectiveC Android

Syntax

```
[Konashi uartWrite:(unsigned char)data];
```

Parameters

data	unsigned char	送信するデータ。1byteです。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

'A'という文字をUARTで送る

```
[Konashi uartWrite:'A'];
```

I2C

i2cMode

Description

I²C を有効/無効を設定します。

I²C で通信できる速度として 100kbps と 400kbps があり `mode` 引数で指定します。

I²C の詳細は Core functions / Communication - I²C をご覧ください。

ObjectiveC Android

Syntax

```
[Konashi i2cMode:(int)mode];
```

Parameters

mode	int	設定するI²Cのモード。`KonashiI2CModeDisable` , `KonashiI2CModeEnable` , `KonashiI2CModeEnable100K` , `KonashiI2CModeEnable400K` が設定できます。`KonashiI2CModeEnable` と `KonashiI2CModeEnable100K` は等価です。

Returns

成功時: `KonashiResultSuccess` , 失敗時: `KonashiResultFailure`

Example

I²C を100kbps(デフォルト)の通信速度で有効にする。

```
[Konashi i2cMode:KONASHI_I2C_ENABLE];
```

i2cStartCondition

ObjectiveC Android

Description

I²C のスタートコンディションを発行します。 事前に i2cMode で I²C を有効にしておいてください。 I²C の詳細は Core functions / Communication - I²Cをご覧ください。

Syntax

```
[Konashi i2cStartCondition];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess` , 失敗時: `KonashiResultFailure`

i2cRestartCondition

ObjectiveC Android

Description

I²C のリスタートコンディションを発行します。 事前に i2cModeで I²C を有効にしておいてください。 I²C の詳細は Core functions / Communication - I²Cをご覧ください。

Syntax

```
[Konashi i2cRestartCondition];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

i2cStopCondition

ObjectiveC Android

Description

I²C のストップコンディションを発行します。 事前に i2cModeで I²C を有効にしておいてください。 I²C の詳細は Core functions / Communication - I²Cをご覧ください。

Syntax

```
[Konashi i2cStopCondition];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

i2cWrite

ObjectiveC Android

Description

I²C で指定したアドレスにデータを書き込みます。 事前に i2cModeで I²C を有効にしておいてください。 I²C の詳細は Core functions / Communication - I²Cをご覧ください。

Syntax

```
[Konashi i2cWrite:(int)length data:(unsigned char*)data address:(unsigned char)address];
```

Parameters

length	int	書き込むデータ(byte)の長さ
data	unsigned char*	書き込むデータ
address	unsigned char	書き込み先アドレス

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

i2cReadRequest

Description

I²C で指定したアドレスからデータを読み込むリクエストを行います。

この関数はリクエストを行うだけでデータは取得できません。実際に値を取得するには、`KonashiEventI2CReadCompleteNotification` を addObserver でキャッチした後、i2cRead で値を取得できます。

Syntax

ObjectiveC

```
[Konashi i2cReadRequest:(int)length address:(unsigned char)address];
```

Parameters

length	int	読み込むデータの長さ
address	unsigned char	読み込み先のアドレス

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

i2cRead

ObjectiveC Android

Description

konashi が I²C から読み込んだデータを取得します。この関数で取得できる値は前回の i2cReadRequest時に取得したデータです。

Syntax

```
[Konashi i2cRead:(int)length data:(unsigned char*)data];
```

Parameters

length	int	読み込むデータの長さ
data	unsigned char*	読み込んだデータを格納するポインタ

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

SPI

spiMode/spiConfig

Description

SPIのモードを設定します。

SPIで設定出来るのは、動作モードと動作速度、ビットオーダーになります。

ObjectiveC | Android

Syntax

```
[Konashi spiMode:(int)mode speed:(int)speed bitOrder:(int)bitOrder];
```

Parameters

mode	int	SPI通信のモードを設定する。`KonashiSPIModeEnableCPOL0CPHA0`, `KonashiSPIModeEnableCPOL0CPHA1`, `KonashiSPIModeEnableCPOL1CPHA0`, `KonashiSPIModeEnableCPOL1CPHA1` が設定可能です。
speed	int	設定するSPIのモード。`KonashiSPISpeedSpeed200K` ~ `KonashiSPISpeedSpeed6M`, を設定できます。詳細は Constants / SPI をご覧ください。
bitOrder	int	設定するSPIのモード。`KonashiSPIBitOrderLsbFirst` , `KonashiSPIBitOrderMsbFirst` が設定できます。詳細は Constants / SPI をご覧ください。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

spiWrite

Description

SPI経由でデータを書き込みます。

ObjectiveC Android

Syntax

```
[Konashi spiWrite:(NSData *)data];
```

Parameters

data　　NSData*　SPI通信で送信するデータ。

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

0x61~0x6bまでのデータをSPI経由で送信する。

```
Byte data[11] = {0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6a, 0x6b};
      [Konashi spiWrite:[NSData dataWithBytes:&data length:11]];
```

spiReadData/spiRead

Description

SPI経由でデータを読み込みます。

Syntax

ObjectiveC Android

```
[Konashi spiReadData];
```

Parameters

なし

Returns

ObjectiveC Android

読み込んだデータ

spiReadRequest

Description

SPI経由で取得したデータを読み込むリクエストを行います。koshianからiOSデバイスにデータを転送します。

Syntax

ObjectiveC

```
[Konashi spiReadRequest];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

SPI経由でデータを読み込み出力をする。

ObjectiveC

```
[[Konashi shared] setSpiWriteCompleteHandler:^{
        [Konashi spiReadRequest];
}];
[[Konashi shared] setSpiReadCompleteHandler:^(NSData *data) {
    self.spiLogTextView.text = [data description];
    NSLog(@"SPI Read %@", [data description]);
}];
```

Hardware Control

reset

ObjectiveC Android

Description

konashi を再起動します。 konashi が再起動すると、自動的にBLEのコネクションは切断されてしまいます。

Syntax

```
[Konashi reset];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

```
[Konashi reset];
```

batteryLevelReadRequest

Description

konashi のバッテリ残量を取得するリクエストを konashi に送ります。

この関数は konashi にリクエストを送るものなので、実際に値を取得するには、`KonashiEventBatteryLevelDidUpdateNotification` (バッテリ残量取得完了イベント)を addObserver でキャッチした後、batteryLevelRead で値を取得できます。

Syntax

ObjectiveC

```
[Konashi batteryLevelReadRequest];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

konashi のバッテリ残量を取得する

ObjectiveC

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [Konashi initialize];

    [Konashi addObserver:self selector:@selector(konashiReady) name:KONASHI_EVENT_READY];
    [Konashi addObserver:self selector:@selector(battery) KONASHI_EVENT_UPDATE_BATTERY_LEVEL];
}

// konashiを探すボタンタップのアクション
- (IBAction)findKonashi:(id)sender {
    [Konashi find];
}

// バッテリ残量取得ボタンタップのアクション
- (IBAction)batteryLevelReadRequest:(id)sender {
    [Konashi batteryLevelReadRequest];
}

- (void) battery
{
    NSLog(@"READ_BATTERY: %d", [Konashi batteryLevelRead]);
}
```

batteryLevelRead

ObjectiveC Android

Description

konashi のバッテリ残量を取得します。この関数で取得できる値は前回の batteryLevelReadReques 時に取得した残量です。 konashiの現在のバッテリ残量を取得したい場合は、まず batteryLevelReadRequesでkonashi に取得リクエストを送り、`KonashiEventBatteryLevelDidUpdateNotification` を addObserverでキャッチした後、この関数で値を取得できます。

Syntax

```
[Konashi batteryLevelRead];
```

Parameters

なし

Returns

`0 ～ 100` のパーセント単位でバッテリ残量が返ります。

Example

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [Konashi initialize];

    [Konashi addObserver:self selector:@selector(konashiReady) name:KONASHI_EVENT_READY];
    [Konashi addObserver:self selector:@selector(battery) KONASHI_EVENT_UPDATE_BATTERY_LEVEL];
}

// konashiを探すボタンタップのアクション
- (IBAction)findKonashi:(id)sender {
    [Konashi find];
}

// バッテリ残量取得ボタンタップのアクション
- (IBAction)batteryLevelReadRequest:(id)sender {
    [Konashi batteryLevelReadRequest];
}

- (void) battery
{
    NSLog(@"READ_BATTERY: %d", [Konashi batteryLevelRead]);
}
```

signalStrengthReadRequest

Description

konashi の電波強度を取得するリクエストを行います。

この関数はリクエストを行うだけでデータは取得できません。実際に値を取得するには、`KonashiEventSignalStrengthDidUpdateNotification` (電波強度取得完了イベント)を addObserver でキャッチした後、signalStrengthRead で値を取得できます。

Syntax

ObjectiveC

```
[Konashi signalStrengthReadRequest];
```

Parameters

なし

Returns

成功時: `KonashiResultSuccess`, 失敗時: `KonashiResultFailure`

Example

konashi の電波強度を取得する

ObjectiveC

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [Konashi initialize];

    [Konashi addObserver:self selector:@selector(konashiReady) name:KONASHI_EVENT_READY];
    [Konashi addObserver:self selector:@selector(strength) KONASHI_EVENT_UPDATE_SIGNAL_STRENGTH];
}

// konashiを探すボタンタップのアクション
- (IBAction)findKonashi:(id)sender {
    [Konashi find];
}

// 電波強度取得ボタンタップのアクション
- (IBAction)signalStrengthReadRequest:(id)sender {
    [Konashi signalStrengthReadRequest];
}

- (void) strength
{
    NSLog(@"READ_STRENGTH: %d", [Konashi signalStrengthRead]);
}
```

signalStrengthRead

ObjectiveC Android

Description

konashi の電波強度を取得します。この関数で取得できる値は前回の signalStrengthReadRequest 時に取得した強度(db)です。距離が近いと-40db, 距離が遠いと-90db程度になります。 konashiの現在の電波強度を取得したい場合は、まず signalStrengthReadRequest で konashi に取得リクエストを送り、 `KonashiEventSignalStrengthDidUpdateNotification` を addObserver でキャッチした後、 この関数で値を取得できます。

Syntax

```
[Konashi signalStrengthRead];
```

Parameters

なし

Returns

電波強度がdbで返ります。

Example

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [Konashi initialize];

    [Konashi addObserver:self selector:@selector(konashiReady) name:KONASHI_EVENT_READY];
    [Konashi addObserver:self selector:@selector(battery) KONASHI_EVENT_UPDATE_BATTERY_LEVEL];
}

// konashiを探すボタンタップのアクション
- (IBAction)findKonashi:(id)sender {
    [Konashi find];
}

// バッテリ残量取得ボタンタップのアクション
- (IBAction)batteryLevelReadRequest:(id)sender {
    [Konashi batteryLevelReadRequest];
}

- (void) battery
{
    NSLog(@"READ_BATTERY: %d", [Konashi batteryLevelRead]);
}
```

Extension Boards

AD変換拡張ボード

ObjectiveC | Android

KONASHI_ADC_CH0	0	AD変換ボードのチャンネル0
KONASHI_ADC_CH1	1	AD変換ボードのチャンネル1
KONASHI_ADC_CH2	2	AD変換ボードのチャンネル2
KONASHI_ADC_CH3	3	AD変換ボードのチャンネル3
KONASHI_ADC_CH4	4	AD変換ボードのチャンネル4
KONASHI_ADC_CH5	5	AD変換ボードのチャンネル5
KONASHI_ADC_CH6	6	AD変換ボードのチャンネル6
KONASHI_ADC_CH7	7	AD変換ボードのチャンネル7
KONASHI_ADC_CH0_CH1	0	チャンネル0とチャンネル1の差動入力
KONASHI_ADC_CH2_CH3	1	チャンネル2とチャンネル3の差動入力
KONASHI_ADC_CH4_CH5	2	チャンネル4とチャンネル5の差動入力
KONASHI_ADC_CH6_CH7	3	チャンネル6とチャンネル7の差動入力
KONASHI_ADC_CH1_CH0	4	チャンネル1とチャンネル0の差動入力
KONASHI_ADC_CH3_CH2	5	チャンネル3とチャンネル2の差動入力
KONASHI_ADC_CH5_CH4	6	チャンネル5とチャンネル4の差動入力
KONASHI_ADC_CH7_CH6	7	チャンネル7とチャンネル6の差動入力
KONASHI_ADC_ADDR_00	0x48	スイッチを00に設定した際のI2Cアドレス
KONASHI_ADC_ADDR_01	0x49	スイッチを01に設定した際のI2Cアドレス
KONASHI_ADC_ADDR_10	0x4a	スイッチを10に設定した際のI2Cアドレス
KONASHI_ADC_ADDR_11	0x4b	スイッチを11に設定した際のI2Cアドレス
KONASHI_ADC_REFOFF_ADCOFF	0	参照電圧とAD変換器のパワーをオフ
KONASHI_ADC_REFOFF_ADCON	1	参照電圧のパワーをオフ、AD変換器のパワーをオン
KONASHI_ADC_REFON_ADCOFF	2	参照電圧のパワーをオン、AD変換器のパワーをオフ
KONASHI_ADC_REFON_ADCON	3	参照電圧とAD変換器のパワーをオン

AC調光拡張ボード

ObjectiveC | Android

KONASHI_AC_MODE_ONOFF	0	ON/OFFモード
KONASHI_AC_MODE_PWM	1	PWMモード
KONASHI_PWM_AC_PERIOD	10000	PWMモードの周期
KONASHI_AC_FREQ_50HZ	50	コンセントの周波数50Hz(東日本)
KONASHI_AC_FREQ_60HZ	60	コンセントの周波数60Hz(西日本)

Grove拡張ボード

ObjectiveC | Android

KonashiDigitalIO0	0	デジタルI/Oの0ピン目
KonashiDigitalIO1	1	デジタルI/Oの1ピン目
KonashiDigitalIO2	2	デジタルI/Oの2ピン目
KonashiDigitalIO3	3	デジタルI/Oの3ピン目
KonashiDigitalIO4	4	デジタルI/Oの4ピン目
KonashiDigitalIO5	5	デジタルI/Oの5ピン目
KonashiDigitalIO6	6	デジタルI/Oの6ピン目
KonashiDigitalIO7	7	デジタルI/Oの7ピン目
KonashiAnalogIO0	0	アナログI/Oの0ピン目
KonashiAnalogIO1	1	アナログI/Oの1ピン目
KonashiAnalogIO2	2	アナログI/Oの2ピン目

konashi AD変換拡張ボード(YE-EX001)

konashi AD変換拡張ボードは、フィジカル・コンピューティング・ツールキットkonashiのアナログ入力を、I²Cを用いて拡張するためのインタフェース基板です。 Groveセンサモジュールをそのまま接続して使用することができます。

initADC / init

Description

AD変換基板の初期化を行います。AD変換基板のアドレスを指定し、アナログの電圧値をI²C経由で読み取ることができるように設定を行います。

Syntax

ObjectiveC | Android

```
[Konashi initADC:address];
```

Parameters

ObjectiveC | Android

address　AD変換拡張ボードのアドレスを指定します。指定できる値はExtensionBoard/Constants/ADCをご覧ください。

Returns

なし

Example

スイッチS1がそれぞれOFF,OFF(アドレス 0x48)のときの初期化

ObjectiveC | Android

```
[Konashi initADC:ADDR_00];
```

readADCWithChannel / read

Description

チャンネル番号を指定して、AD変換基板からデータを受け取ります。

Syntax

ObjectiveC | Android

```
[Konashi readADCWithChannel:channel];
```

Parameters

ObjectiveC | Android

channel　AD変換拡張ボードのチャンネルを指定します。指定できる値はExtensionBoard/Constants/ADCをご覧ください。

Returns

なし

Example

チャンネル0から読み取り

ObjectiveC | Android

```
[Konashi readADCWithChannel:KONASHI_ADC_CH0];
```

readDiffADCWithChannels / readDiff

Description

チャンネル番号を指定して、AD変換基板から差動をとったデータを受け取ります。 この機能を使うと、チャンネル間の電圧の差を取得することができます。

Syntax

ObjectiveC | Android

```
[Konashi readDiffADCWithChannels:channels];
```

Parameters

ObjectiveC | Android

channel　AD変換拡張ボードのチャンネルのペアにあたる値を指定します。指定できる値はExtensionBoard/Constants/ADCをご覧ください。

Returns

なし

Example

チャンネル1を基準電圧(0V)としたときのチャンネル0の値を取得

ObjectiveC | Android

```
[Konashi readDiffADCWithChannels:KONASHI_ADC_CH0_CH1];
```

selectADCPowerMode / selectPowerMode

Description

AD変換拡張ボードに搭載されているICのモードを切り替えます。 IC内部の各機能への電源の供給をON/OFFすることができます。

Syntax

ObjectiveC | Android

```
[Konashi selectADCPowerMode:mode];
```

Parameters

ObjectiveC | Android

mode　AD変換拡張ボードの電源モードを指定します。指定できる値はExtensionBoard/Constants/ADCをご覧ください。

Returns

なし

Example

AD変換機能と参照電圧機能をONにする

ObjectiveC | Android

```
[Konashi selectADCPowerMode:KONASHI_ADC_REFON_ADCON];
```

konashi AC調光拡張ボード(YE-EX003)

konashi AC調光拡張ボード(YE-EX003)は、フィジカル・コンピューティング・ツールキットkonashi(YE-WPC001)のディジタル出力とPWMを使用して、コンセントからの電源を使用するライトやヒータなどの出力を制御する拡張ボードです。

initACDrive / init

Description

AC調光拡張ボードで使用するピンの初期化を行います。

Syntax

ObjectiveC | Android

```
[Konashi initACDrive:mode freq:freq];
```

Parameters

ObjectiveC | Android

mode	AC調光拡張ボードの動作モードを指定します。指定できる値はExtensionBoard/Constants/ACDriveをご覧ください。
freq	使用するコンセントの周波数を指定します。西日本では60Hz、東日本では50Hzです。

Returns

なし

Example

PWMモードを東日本で使用するために初期化する

ObjectiveC | Android

```
[Konashi initACDrive:KONASHI_AC_MODE_PWM freq:KONASHI_AC_FREQ_50HZ];
```

onACDrive / on

Description

ON/OFFモードのとき、出力をONにします。

Syntax

ObjectiveC | Android

```
[Konashi onACDrive];
```

Parameters

なし

Returns

なし

offACDrive / off

Description

ON/OFFモードのとき、出力をOFFにします。

Syntax

ObjectiveC Android

```
[Konashi offACDrive];
```

Parameters

なし

Returns

なし

updateACDriveDuty / updateDuty

Description

AC調光拡張ボードのDuty比を設定します。 この関数を使用するには，PWMモードに設定している必要があります.

Syntax

ObjectiveC Android

```
[Konashi updateACDriveDuty:ratio];
```

Parameters

ObjectiveC Android

ratio	Duty比を1から100で指定します。

Returns

なし

Example

Duty比を50%に設定する

ObjectiveC Android

```
[Konashi updateACDriveDuty:50];
```

selectACDriveFreq / selectFreq

Description

AC調光拡張ボードで使用するコンセントの周波数を設定します。

Syntax

ObjectiveC | Android

```
[Konashi selectACDriveFreq:freq];
```

Parameters

ObjectiveC | Android

freq　　使用するコンセントの周波数を指定します。指定できる値はExtensionBoard/Constants/ACDriveをご覧ください。

Returns

なし

Example

PWMモードを東日本で使用するために周波数を設定する

ObjectiveC | Android

```
[Konashi selectACDriveFreq:KONASHI_AC_FREQ_50HZ];
```

konashi Grove拡張ボード(YE-EX004)

konashi Grove拡張ボード(YE-EX004)は、フィジカル・コンピューティング・ツールキットkonashi(YE-WPC001)の入出力ピンで、Groveモジュールを使用できるようにする拡張ボードです。

konashi 2 では、PIOのピン数が変更(8->6)になっているため、該当するポートが使えません。

writeGroveDigitalPort / digitalWrite

Description

Grove拡張ボードのディジタルポートの出力状態を設定します。

この関数はGrove拡張ボード向けに digitalWrite 関数のニックネームとして定義されています。 プログラムの動作は digitalWrite 関数と同じです。

Syntax

ObjectiveC | Android

```
[Konashi writeGroveDigitalPort:(int)port];
```

readGroveDigitalPort / digitalRead

Description

Grove拡張ボードのディジタルポートの値を取得します。

この関数はGrove拡張ボード向けに digitalRead 関数のニックネームとして定義されています。 プログラムの動作は digitalRead 関数と同じです。

Syntax

ObjectiveC | Android

```
[Konashi readGroveDigitalPort:(int)port];
```

readGroveAnalogPort / analogReadRequest

Description

Grove拡張ボードのアナログポートの値を取得するリクエストを konashi に送ります。

この関数はGrove拡張ボード向けに analogReadRequest 関数のニックネームとして定義されています。 プログラムの動作は analogReadRequest 関数と同じです。

Syntax

ObjectiveC | Android

```
[Konashi readGroveAnalogPort:(int)port];
```


Contact us: contact@ux-xu.com