2017

Smart.IO User Guide

https://imagecraft.com/smartio

V0.1 2017/12/14 richard@imagecraft.com

RICHARD MAN

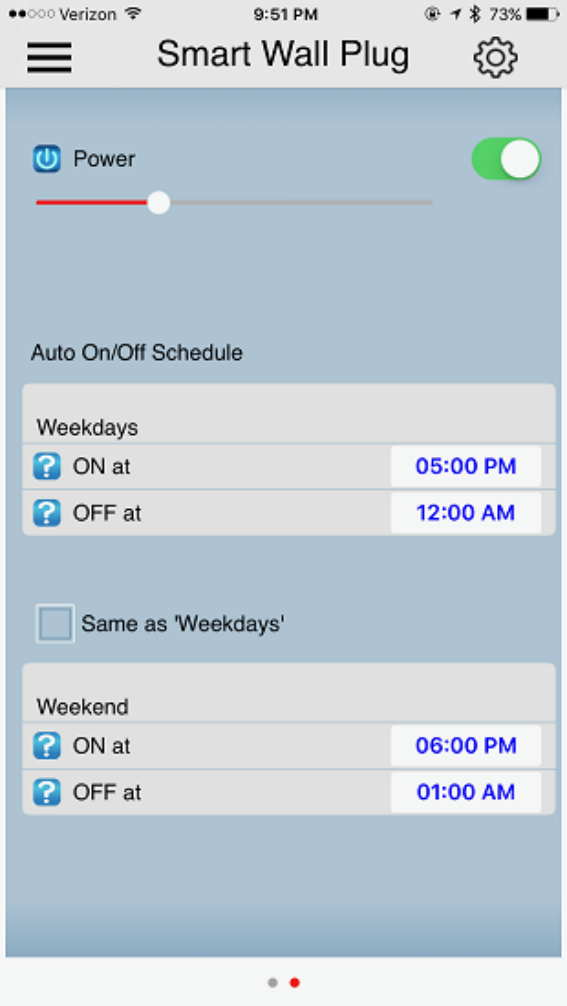
Table of Contents

OVERVIEW

Introduction

Smart.IO is a new way of creating UI (User Interface) for embedded designs. Indeed, the word UI is typically not associated with an embedded design unless and until it becomes a product. With Smart.IO, all that changes: ***with Smart.IO, an embedded designer can create a UI in a few lines of code, and without writing any wireless or app code, the UI will run on any iOS and Android devices***. The only investment is to add the Smart.IO hardware module to the design, and to build the firmware with the Smart.IO API library, which is provided in source code form. The system is compatible with nearly all microcontrollers.

This is a sample UI:



The UI has a on/off button, a slider, a few time selectors, and a checkbox. This is the entirety of the function that creates the UI, with the UI controls creation highlighted in red:

```
void CreateUI(void)
    {
```

```
tHandle p0, p1, p2, p3;
tHandle u0, u1, u2, u3, u4, u5, u6, u7, u8, u9, u10;

SmartIO_LoadCache(PRODUCT_ID, BUILD_ID);
p0 = SmartIO_MakePage();
SmartIO_AppTitle("Smart Wall Plug");
u0 = SmartIO_MakeOnOffButton(0, 0, 1, Button1);
SmartIO_AddText(u0, "Power");
SmartIO_SetSliceIcon(u0, SMARTIO_ICON_POWER);
u1 = SmartIO_MakeSlider(1, 0, 30, Slider1);
SmartIO_EnableIf(u0+1, u1+1, 0);

SmartIO_MakeSpacerSlice(2);
u2 = SmartIO_MakeLabel(0, 1, "Auto On/Off Schedule");
u3 = SmartIO_MakeLabel(0, 0, " Weekdays");
u4 = SmartIO_MakeTimeSelector(0, 0, "17:00", time_selector1);
SmartIO_AddText(u4, "ON at");
SmartIO_SetSliceIcon(u4, SMARTIO_ICON_QUERY);
u5 = SmartIO_MakeTimeSelector(0, 0, "0:00", time_selector2);
SmartIO_AddText(u5, "OFF at");
SmartIO_SetSliceIcon(u5, SMARTIO_ICON_QUERY);
SmartIO_MakeSpacerSlice(1);

u6 = SmartIO_MakeCheckboxes(1, 1, 0);
SmartIO_AddListItem(u6+1, "Same as 'Weekdays'");

u7 = SmartIO_MakeLabel(0, 0, " Weekend");
u8 = SmartIO_MakeTimeSelector(0, 0, "18:00", time_selector3);
SmartIO_AddText(u8, "ON at");
SmartIO_SetSliceIcon(u8, SMARTIO_ICON_QUERY);
u9 = SmartIO_MakeTimeSelector(0, 0, "1:00", time_selector4);
SmartIO_AddText(u9, "OFF at");
SmartIO_SetSliceIcon(u9, SMARTIO_ICON_QUERY);
SmartIO_GroupObjects(0, u3, u4, u5, u7, u8, u9, 0);
SmartIO_MakeSpacerSlice(3);
SmartIO_AutoBalance(p0);
SmartIO_SaveCache();
}
```

As seen in the example, it takes only one function call to create each UI element. The rest of the code adds text labels to UI elements and are the “magic stuff” that allow the UI to have very

similar balanced look and feel on any devices, regardless the smartphone OS or the device resolution.

Firmware Driven UI

In a Smart.IO system, the embedded firmware creates the UI. This is different and more efficient than the traditional method of “writing an app for an embedded product”, which involves separate UI developers to work with firmware engineers and then protocols must be designed and implemented. Changes in the UI app or the embedded firmware, due to design changes must be coordinated among separate engineers, introducing delays and potential errors.

With Smart.IO, the UI is embedded in the firmware, and thus eliminating many potential errors.

Where to Find Information

The main page for Smart.IO is here: https://imagecraft.com/smartio/

The software download page, including software updates, is here: https://imagecraft.com/download/smart-io-downloads

The documentation is here: https://imagecraft.com/documentation/smart-io-documentation

ImageCraft forums is here: https://imagecraft.com/forums

Overview of the Smart.IO System

The Smart.IO toolkit is made up of both a hardware module and software components. This table lists the components.

Component	Description
Smart.IO hardware module	Provides hardware for BLE communication and the firmware support to implement the Smart.IO API
Host Interface Layer	Source code in Standard C that an embedded user compiles with their host MCU firmware. Converts the Smart.IO API calls into command stream for the Smart.IO firmware.
Smart.IO API	The API that the embedded users call to build and interact with the UI
iOS/Android Smart.IO app	Smartphone app that interfaces with a Smart.IO-enabled device and provides the UI for the device

To use Smart.IO in your design:

1. Add the Smart.IO hardware module to your hardware. It interfaces to the host MCU via SPI.
2. Port the Host Interface Layer to your MCU and compiler, if it has not been done yet.
3. Add Smart.IO API calls to your firmware to create the UI
4. DONE!

All of these will be explained in details.

Customized App for Product Release

The Smart.IO App is a generic app that works with any Smart.IO enabled devices. When you are ready to release a product, we can customize an app specific to your device with your branding and logos and secured ID so the app will only work with your devices. ImageCraft can even provide more customization options such as your own graphics and performance enhancements. Visit our webpage https://imagecraft.com/smartio or contact us for details.

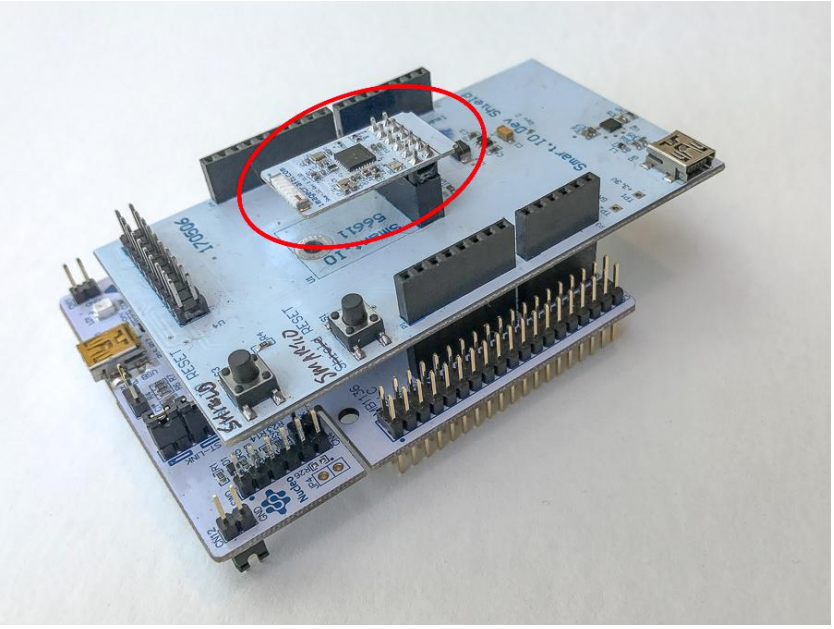
Evaluating Smart.IO Using the Smart.IO Starter Kit

If you have purchased the Smart.IO Starter Kit, you can start evaluating the Smart.IO technology immediately. However, having the Starter Kit is not mandatory: you can use other hardware to test Smart.IO and you may safely skip or skim this section if you do not have the Starter Kit.

The Smart.IO Starter kit contains:

1. A Smart.IO module (denoted by the red ellipse in the photo below)
2. An Arduino compatible shield (the board in the middle where the Smart.IO module is sitting on)
3. An ST-Nucleo-F411 MCU board (the bottom board)

Figure 1 Smart.IO Starter Kit



Note: only the Smart.IO module is an essential part of using Smart.IO. The Arduino shield and the ST-Nucleo are not necessary for using Smart.IO, and are part of the evaluation kit only. In an embedded user application, they may use any MCU of their choice, needing only to incorporate the Smart.IO module into their hardware designs.

ST-Nucleo Driver

When you receive the kit, connect a USB cable to the ST-Nucleo board at the BOTTOM of the stack. Windows should automatically install the driver. If not, please visit http://st.com and search for “ST-Nucleo driver”. The driver should enable the following features:

1. Windows virtual comport: you can find the comport number under “Control Panel->Device Manager->Ports (COM & LPT)”
2. ST-LINK debug pod
3. A virtual removable drive

The simplest way to download a new firmware to a ST-Nucleo is to drag and drop a suitable .bin file with the firmware to its virtual removable drive. For advanced development, you can also use the ST-LINK debug pod and a compiler/debugger toolset.

UI Demo

The ST-Nucleo board is pre-programmed with demo test firmware. The demo is included in the Host Interface Layer source zip file, which is available on the webpage http://imagecraft.com/download/smart-io-downloads.

To run the demo:

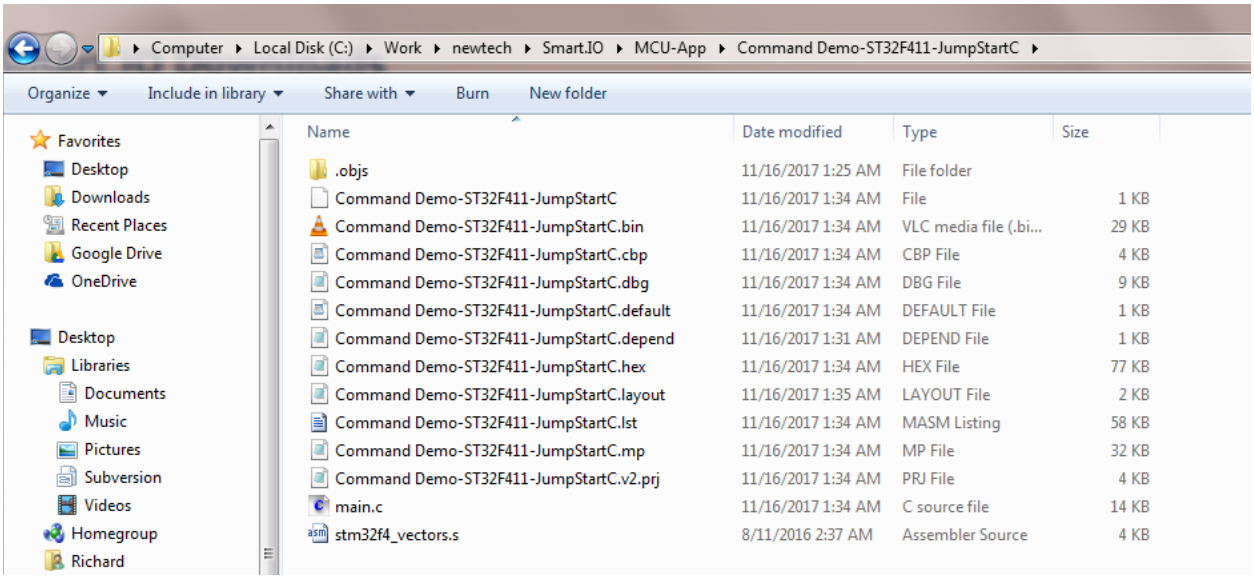
1. Connect a USB cable to the ST-Nucleo (the bottom board).
2. Install the Smart.IO iOS or Android smartphone app on your smartphone (search for “ImageCraft SmartIO” on the respective phone store).
3. Enable BLE (Bluetooth) access on the smartphone.
4. Invoke the Smart.IO app.
5. Connect to the Smart.IO on the app (on Android, the name may not be visible, depending on the Android version, but it will have a green dot next to the name).
6. After a few seconds, a sample UI page will pop up. Note that the Android version may take up to 10-20 seconds (but often shorter) due to Android BLE architecture. We are looking into improvements on this.

The sample UI is the “Smart Wall Plug” example in the first chapter.

Command Demo

There is another demo called the “Command Demo” included in the Host Interface Layer source code .zip file (the preloaded one is called the “UI Demo”). To run the Command demo, download the .zip file from our website http://imagecraft.com/download/smart-io-downloads. After unzipping the file, browse to the directory “MCU-App\Command Demo-ST32F411-JumpStartC”

Figure 2 Smart.IO "MCU-App" Directory



Then drag the file "Command Demo-ST32F411-JumpStartC.bin" to the ST-Nucleo's virtual drive folder (the .bin file for the UI Demo is also included in the zip file so you can revert to the UI Demo as needed). This will program the ST-Nucleo with the firmware.

To run the demo, you need to connect both the Smart.IO Arduino shield and the ST-Nucleo to your PC/Mac via USB cables, and set up terminal programs for display.

Setting up the terminal programs

There is one USB port on the Arduino shield, and another one on the ST-Nucleo. Drivers should be automatically installed by the OS. In case of any issues, the ST-Nucleo driver can be found on http://st.com (search for "ST-Nucleo USB driver"). The shield's USB port uses the FTDI serial-to-USB chip and its drivers can be found on http://ftdichip.com.

MacOS comes with built-in terminal program, and Windows users may use the "PuTTY" terminal program. Set the baudrate to 9600.

The ST-Nucleo is running an interactive program which allows you to type commands. The Smart.IO firmware is using the terminal (through the USB port on the Arduino shield) for diagnostic message display only.

Startup operations

1. Connect the USB cables to the starter kit and invoke the terminal programs. The kit gets its power from the USB cables.

2. Install the Smart.IO iOS or Android smartphone app on your smartphone (search for “ImageCraft SmartIO” on your respective phone’s app store).
3. Enable BLE (Bluetooth) access on the smartphone.
4. Invoke the Smart.IO app.
5. Connect to the Smart.IO on the app (on Android, the name might not be visible, depending on the Android version, but it will have a green dot next to the name).
6. If the terminal programs have been set up correctly, you should see sign-on messages from both boards.
7. Wait until you see “BLE connected” message on the ST-Nucleo terminal window before proceeding.
8. If you do not see the “BLE connected” message after 10 to 15 seconds, click on the gear icon on the app and select “Rescan devices”, and then restart from Step 5 and reset the kit.

Figure 3 BLE Scanning Page on the Phone App

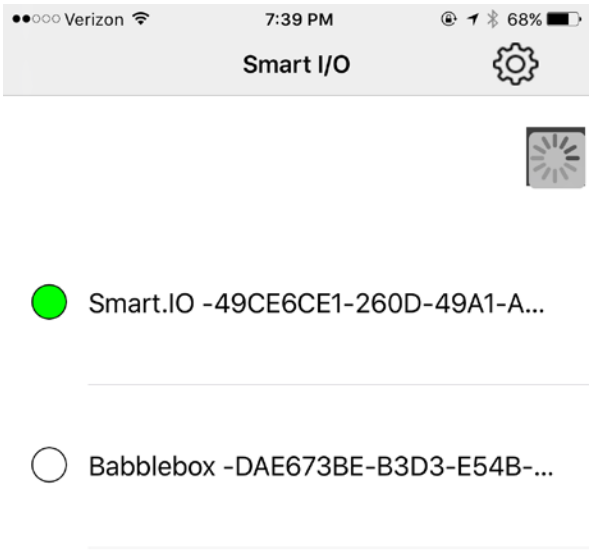


Figure 4 Sign-on message on the ST-Nucleo terminal window

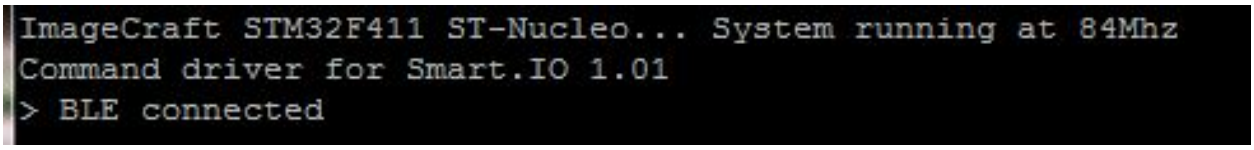
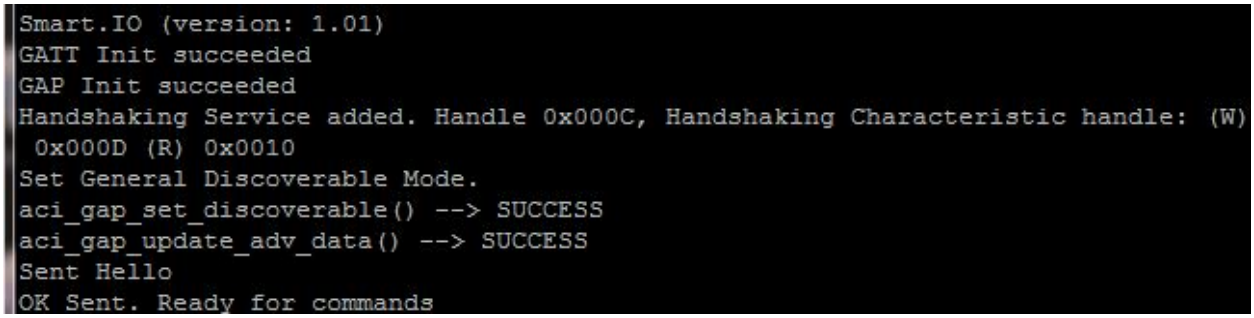


Figure 5 Sign-on message from the Arduino shield+Smart.IO module



Running the test program

Once connected, hit RETURN on the **ST-Nucleo terminal** (and NOT the Smart.IO window!), and you should see a prompt "> " at which you may type in a command followed by carriage return. For the purposes of this demo, all commands are in the form of "128 <number>" where <number> is 0 to 8[1].

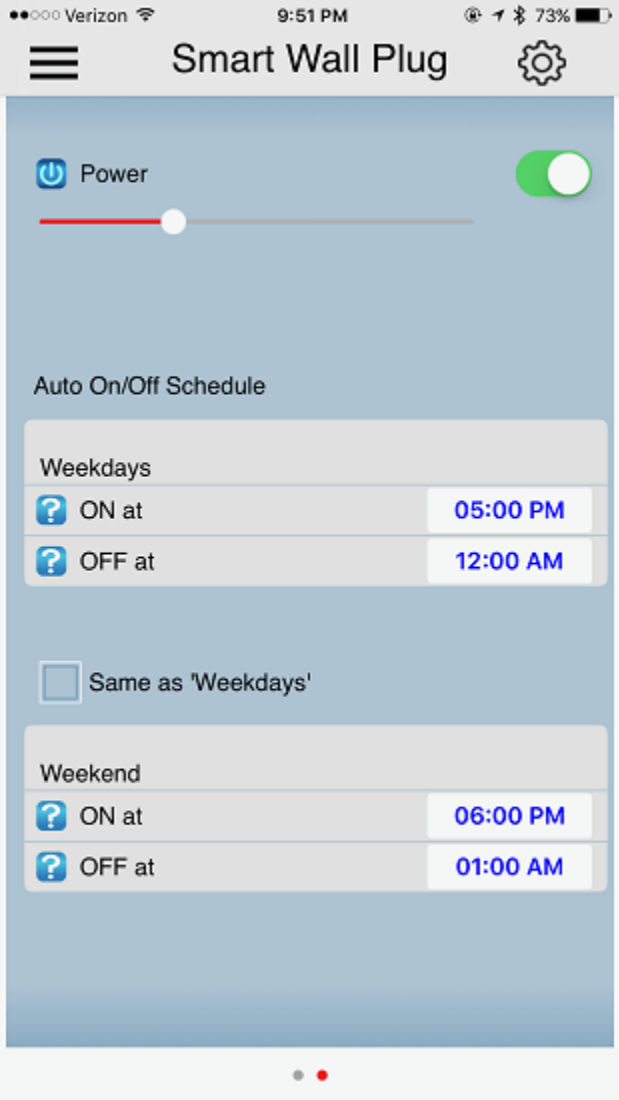
The test programs are summarized in the following table [2]. The most complex one is "128 4". You may either run the commands with or without resetting the boards and restarting the process. If you do not reset the boards, most tests create a new page with sample UI, and you can swipe sideways to access different pages.

Test Command	**Description**
128 0	Creates a "user application menu"
128 1	Demonstrates a textbox with font control
128 2	Smart Wall Plug UI, slightly simplified
128 3	Demonstrates checkboxes, radio buttons, and multiline textbox
128 4	Smart Wall Plug UI
128 5	Sample UI for a battery charger
128 6	UI for a Pokemon panel
128 7	Shows two text entry boxes for text input
128 8	Demonstrate non-UI Smart.IO API: generating random number, unique ID, and EEPROM storage

Running "128 4" should produce a screen looking like this, the same as the UI Demo:

[1] Future revisions of the test program may allow more test samples and the allowable number may increase.
[2] Again, this is subject to future revisions

Figure 6 "Smart Wall Plug" Sample UI



Changing the sliders or the time values send the updated values to the ST-Nucleo test program.

Modifying the test program

The source code of the test program and the “Host Interface Layer” (see the “Quick Start Guide to Smart.IO” in the documentation page) are available on the download page.

INTEGRATING WITH SMART.IO

Hardware Integration Guide

The Smart.IO module is designed to be easily incorporated into an embedded design with a host MCU. It provides the hardware and software interface to smartphones over the BLE [3] wireless interface. This document describes the hardware interface. Please see the separate document "Smart.IO Software Integration Guide" for the software interface and porting guide.

Physical dimensions:

- 15.5mm x 25.5mm (0.6” x 1”)
- 2x6 0.1in (2.54 x 2.54) male header

Power Requirements

- 3.3V Vdd
- Compatible with 3.3V Arduino with the Smart.IO Arduino shield
- Normal current draw: transmitting (BLE) 15mA @ +8dBm.
- Active: 2mA
- Sleep mode current draw: 17uA + 0.3uA for I2C EEPROM Standby

Smart.IO Module Placement Recommendations

To obtain the best performance for BLE, the following guidelines should be observed:

- Any host design electronic components should be kept away from the antenna (the white block) of the Smart.IO module
- Likewise, the PCB’s ground planes should be kept away from the same area
- The Smart.IO module should not be covered by a grounded metal case

5V System Compatibility

The Smart.IO module works with a Vdd supply voltage of 1.7V to 3.3V with a maximum of 3.6V. When working with 5V devices like some of the Atmel AVR and older MCUs, care must be taken to ensure that the Vdd input does not exceed 3.6V. Moreover, the I/O pins must be level-shifted between the two devices; for example, by using a device such as the TXB0108 8-channel bidirectional channel logic level converter, available here: https://www.adafruit.com/product/395 (if the link is no longer valid, just do a web search with the above descriptions as search terms).

[3] Bluetooth Low Energy, the wireless communication mechanism between the Smart.IO module and smartphones / smart devices

2x6 0.1” Header

The host hardware interfaces with the Smart.IO through the 2x6 0.1” male header. Looking down at the chip module with the header rows on top, pin 1 of the header is located at the lower right corner.

Pin Number (I/O)[4]	**Function**	**Pin Number (I/O)**	**Function**
2 / O	USART Tx	1 / I	USART Rx
4 / I	RESET	3 / I	Vdd[5]
6 / I	SWCLK	5 / I/O	SWDIO
8 / O	SPI MISO	7 / I	SPI MOSI
10 / I	SPI nCS	9 / I	SPI SCK
12 / I	GND	11 / O	Host IRQ / DIO7

Microcontroller Interface

The interface between the host MCU and the Smart.IO consists of:

- SPI - MOSI, MISO, SCK (clock), nCS (chip select)
- Host IRQ - interrupt signal (Smart.IO to MCU), active low. Also used for bootloader firmware update
- RESET - resetting the Smart.IO module, active low
- Vdd and GND

SPI

The MCU is the SPI master in this setup and drives the SPI clock. As multiple SPI slaves may sit on a single SPI bus, the nCS (Chip Select) is used by the SPI master to select the SPI slave which should respond to a particular transaction.

- SPI in 8-bit mode
- Maximum bus frequency is 1 MHz
- CPOL is 0 and CPHA is 1
- MSBit transmitted first
- nCS is active low

[4] (I/O) = from the point of view of the Smart.IO module
[5] +3.3V to 3.6V required

Host IRQ

To inform data availability from Smart.IO to the host MCU, the Host IRQ pin is used. This must be connected to a GPIO pin in the host MCU. On the MCU:

- Configured the connected pin as an input pin
- Input interrupt triggered by transition from low to high
- Signal is pulled high by the Smart.IO module
- Signal is in high impedance state

The pin is held high as long as data is being transmitted from Smart.IO. It's also used for updating firmware using the bootloader. See below.

Smart.IO RESET

The host MCU may use this pin to reset the Smart.IO module. This must be connected to a GPIO pin in the host MCU. On the MCU:

- Configured the connected pin as an output pin
- Normal state is level high
- Must be pulled high by either the MCU internal resistor or an external resistor
- Pull low for one to ten milliseconds to cause a Smart.IO hardware reset

UART Pins

In addition to the SPI and IRQ pins, other pins from the internal BlueNRG1 pins are brought out as well. These pins can be left unconnected in your hardware design, if you do not use their features.

To facilitate advanced debugging, the host firmware can invoke a Smart.IO API to emit debug info on the UART port (at 9600 baud).

UART can also be used in bootload mode. See below.

Bootloader Mode

You may put the BlueNRG1 in bootloader mode by resetting and pulling the DIO7 (Host IRQ) pin high. You will need to use the UART port for bootloader operations. The Smart.IO Arduino Shield available from ImageCraft is a simple to use option for using the bootload mode.

JTAG pins

The JTAG pins (SWCLK and SWDIO) are for flash programming using the JTAG/SWD port.

Arduino Style Shield

WARNING: All official AVR Arduino except Arduino "mini PRO" with Mega328P are 5V only, and will need logic level shifter to be compatible with the Smart.IO.

ImageCraft provides an optional Arduino shield with a dedicated socket for the Smart.IO module. The Smart.IO module and the shield are compatible with 3.3V Arduino-like systems. The Smart.IO module can draw power from either the shield's Micro-USB connector, or from the Arduino 3.3V pin.

Arduino Shield Header Pinouts

When mounted on the Arduino-style shield, the signals in the 2x6 header are routed to the following pins in the 10-pin Arduino header:

ST Nucleo-401 Pin Number	**Function**	**ST Nucleo-401 Pin Number**	**Function**
1/ PA9	Host IRQ / DIO7	6/ PA5	SPI SCK
2/ PC7	Smart.IO RESET	7/ GND	GND
3/ PB6	SPI CS	8/ Vss	Vss
4/ PA7	SPI MOSI		
5/ PA6	SPI MISO		

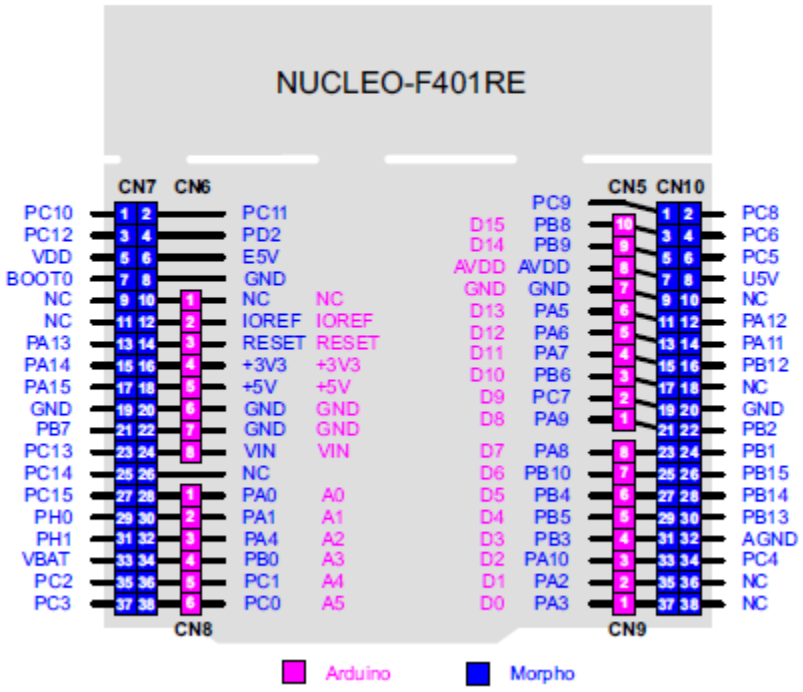
The rest of the 2x6 header pins are routed to the JTAG/SWD header and the UART port. See below.

ST-Nucleo Boards with Arduino-style Headers

This is the pinout diagram of the ST Nucleo-411 Arduino compatible board. Other ST Nucleo boards have very similar pinouts:

Figure 7 ST-Nucleo F401RE Pinouts

Figure 16. NUCLEO-F401RE



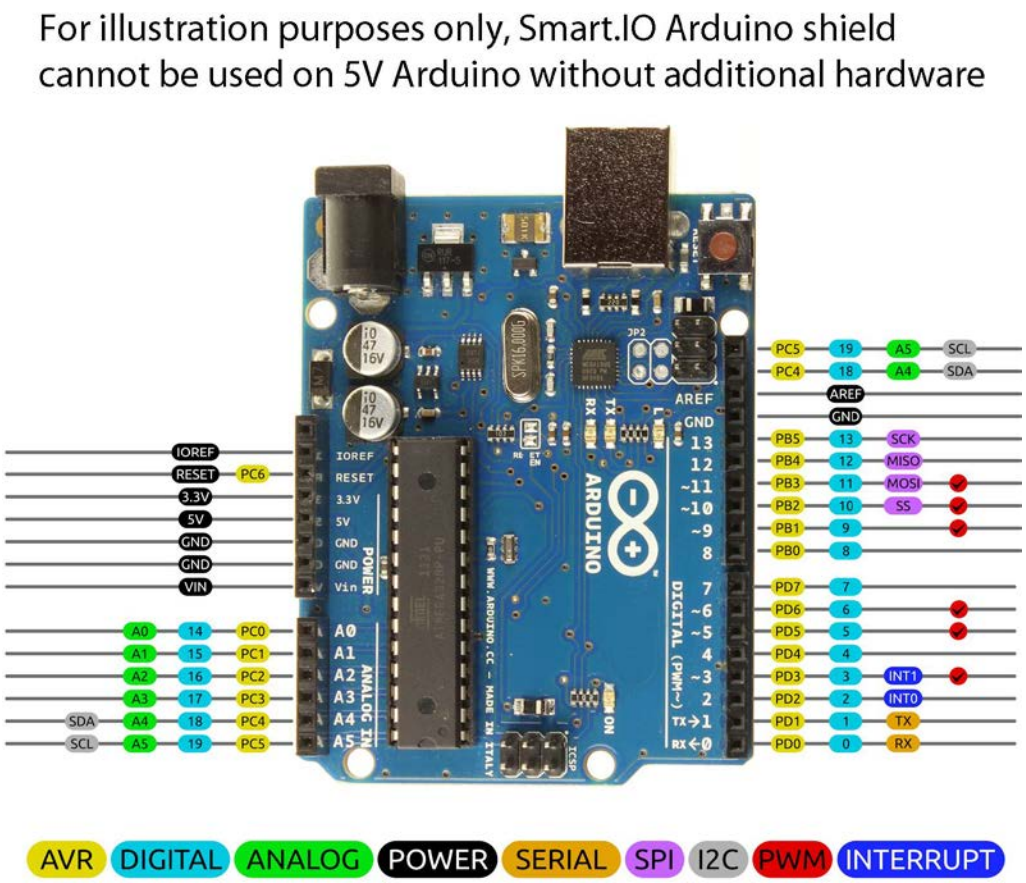
To put Smart.IO in bootload mode for a firmware upgrade with the ST Nucleo-411, you would jumper D8/PA9 and AVDD together while resetting the Smart.IO module.

AVR Arduino

WARNING: All official AVR Arduino except Arduino “mini PRO” with Mega328P are 5V only, and will need logic level shifter to be compatible with the Smart.IO.

This is the pinout diagram of an Arduino board. Note that the diagram uses the Uno R3, as it is considered the “standard” basic Arduino. However, the Uno R3 is 5V, and thus not directly compatible with Smart.IO. However, the Arduino Mini PRO with Mega328P is 3.3V compatible and has the exact same pinouts.

Figure 8 AVR Arduino Pinouts



To put Smart.IO in bootload mode for a firmware upgrade with the Arduino, you would jumper PB0 and AREF together while resetting the Smart.IO.

Arduino Shield JTAG Header Pinouts

Additionally, the Smart.IO Arduino shield board comes with a JTAG/SWD header. This is useful for programming the Smart.IO firmware using a JTAG/SWD pod such as the Segger JLINK or the ST ST-LINK. (The pinout details are not presented here since they are standard JTAG header pinouts.) The SWCLK and SWDIO signals from the 2x6 Smart.IO header are routed to this header.

FTDI/USB Micro-USB Connector

FTDI/USB is part of the original V1 release of the Smart.IO Arduino shield and is an optional component in the V2 release. It uses the FTDI industry standard driver. It provides a VCOM port to the Smart.IO module. Smart.IO API functions are provided to use the VCOM for debugging

purposes. It can also be used for bootload programming. The UART Tx and UART Rx from the 2x6 Smart.IO header are routed to the FTDI/USB chip (V1) or the 4-pin UART connector (V2).

Software Integration Guide

This chapter describes the software interface, including the porting guide for the Smart.IO host interface layer software. The API and graphics elements are described later.

The Smart.IO host interface layer is provided in Standard C source code and must be compiled and linked with the embedded firmware that uses Smart.IO. ImageCraft provides working ports of the host interface layer to the following platforms and thus you do not need to read this document if you are using one of these platforms.

Device	**Compiler**	**Flash Memory (bytes)**	**RAM (bytes)**
STM32F411	JumpStart C for Cortex	Under 10 K	1K - 2K
AVR	JumpStart C for AVR	Under 10K	1K - 2K

Smart.IO API

The host MCU firmware calls the Smart.IO API functions to create GUI objects and also to provide callback functions for host interface layer to invoke when input elements are changed (e.g. toggling an on/off switch or change the value of a slider).

Smart.IO API are a set of C functions (prefixed with “SmartIO_”). They implement the public software interface to Smart.IO. A sample call is:

```
tHandle SmartIO_MakeOnOffButton(
            uint16_t alignment,
            uint16_t variation,
            uint16_t initial_value,
            void (*callback)(uint16_t));
```

This creates an on/off button on the app screen. It returns a “handle” to the on/off button object to the calling firmware. See the chapter **Software API** for details.

Host Interface Layer Architecture

The Host Interface Layer consists of the following modules:

1. Functions that take the API data and translate it into command stream. These are machine-independent.
2. Functions that perform SPI data communication. These are MCU-specific.
3. Functions that initialize and access MCU hardware components. These are MCU-specific.

API are described in a later chapter. ImageCraft provides pre-built host interface layer in binary forms for some popular MCUs such as the STM32F4xx, Atmel AVR series. We will provide additional pre-built binaries as they become available. Please visit our page https://imagecraft.com/smartio/ for details.

Blocking API

When the host firmware makes a Smart.IO API function call, there is some communication and processing (including BLE [6] data transfers) overhead before a result is returned to the host firmware. To simplify the Host Interface Layer design and implementation, Smart.IO API functions are blocking and do not return until the result is returned (and processed) from the smartphone app. Thus from the point of view of the firmware, after it makes a Smart.IO API call, it may for some times before the function returns. The time period is usually short, under a millisecond, but sometimes may take longer due to BLE data communication overhead.

As most API functions are used for creating UI elements, they are usually run in the initialization phase of the host firmware, and the blocking nature should not affect the performance of the host MCU.

Interrupts

Smart.IO generates data to the host MCU from two different sources:

1. Synchronous: as part of the Smart.IO API call chain.
2. Asynchronous: when the end user interacts with the smartphone app and changes the value of an input element, the new value needs to be communicated to the host firmware.

As the host MCU is the SPI master, whenever there are data available, the Smart.IO module must inform the host MCU so that the host software can initiate an SPI transfer. This is done by using a hardware pin for host interrupt.

The actual interrupt handling is provided in the Smart.IO Host Interface Layer. However, the method to associate an interrupt handler to a hardware GPIO [7] pin is MCU specific.

Host Interface Layer Source Files

The following source files are provided:

- smartio_api.c/.h - the API functions. The host firmware uses these to access the Smart.IO functionality.
- smartio_interface.c/.h - the low-level access functions used by the API code.

[6] Bluetooth Low Energy, the wireless communication mechanism used in Smart.IO
[7] General Purpose Input Output

- smartio_hardware_interface.c/.h - the MCU specific hardware access functions. These must be ported to a specific MCU. The port is either provided by ImageCraft, or must be provided by the user.

Porting Tasks

To port the Smart.IO Host Interface Layer to a MCU, modify the smartio_hardware_interface.c file to perform the following:

Hardware Initializations

Configure the SPI as follows:

- SPI master
- 8-bit mode
- Maximum bus frequency is 1 MHz
- CPOL is 0, CPHA is 1
- MSBit transmitted first
- nCS (Chip Select) is active low

Configure the Host IRQ pin: the Host IRQ pin should be configured as input interrupt source, active high - i.e. interrupt should be triggered when the signal goes from low to high. The signal remains high as long as the Smart.IO is transmitting data to the MCU host. The signal is pulled-down by the Smart.IO module.

SPI Data Transfer

You must provide a function to read and write from the SPI interface. Since the API works in blocking mode, no interrupt mode is required.

Host IRQ Interrupt Handling

You must associate the provided interrupt handler with the Host IRQ interrupt. The interrupt should trigger when the signal goes from low to high.

Other Source Files in the Host Interface Layer

Other files in the Host Interface Layer should not be modified. They are written in Standard C and should be portable to most MCU.

Complete Data Flow of a Smart.IO API Call

Taking the above together, the following is the complete data flow of a single Smart.IO API call.

Host MCU Firmware	**Smart.IO Host Interface Layer**	**Smart.IO firmware**	**BLE [8] firmware**	**Smart.IO smartphone app**
Data flow direction →				
Smart.IO API call				
	Convert API request to command stream			
	Initiate SPI data transfer			
		Process SPI into BLE data		
			BLE transfer	
				GUI actions
← *Data flow direction*				
				Command response
			BLE transfer	
		Process BLE data into SPI stream		
		Pull Host IRQ high		
	Interrupt handler reads SPI data stream			
	Return data to host MCU			
Receive return value				

[8] ST BlueNRG-1 BLE SoC

Smartphone App

One may look at the Smart.IO app as a remote display manager: for example, the embedded system uses a simple C API function call (e.g. `SmartIO_MakeSlider`), and the app displays an interactive slider. When the app user (e.g. the user of the embedded device) adjusts the slider, the updated value is sent back to the embedded device for processing.

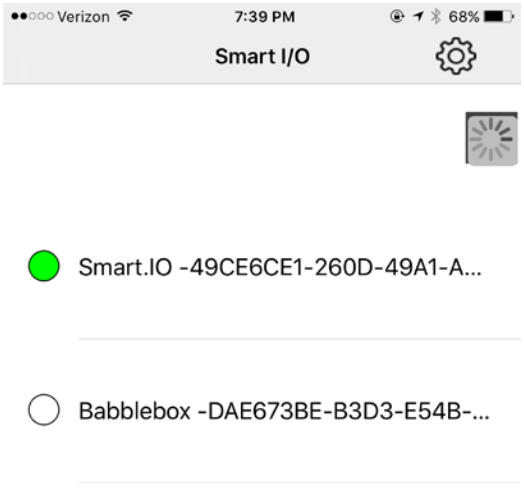
Once paired with a Smart.IO-enabled device, the embedded device sends UI commands through the BLE interface and the app interprets the commands, usually resulting in some UI element(s) being displayed.

A key innovation with Smart.IO API is that we utilize the concept of GUI slices, so that the generated UI will look good on any smartphones regardless of OS and screen resolution, from the iPhone 5 to the iPad Pro, and any Android devices. The embedded firmware engineer does not need to worry about screen resolution, exact pixel placement, etc.

Some Sample Screens and Additional Features

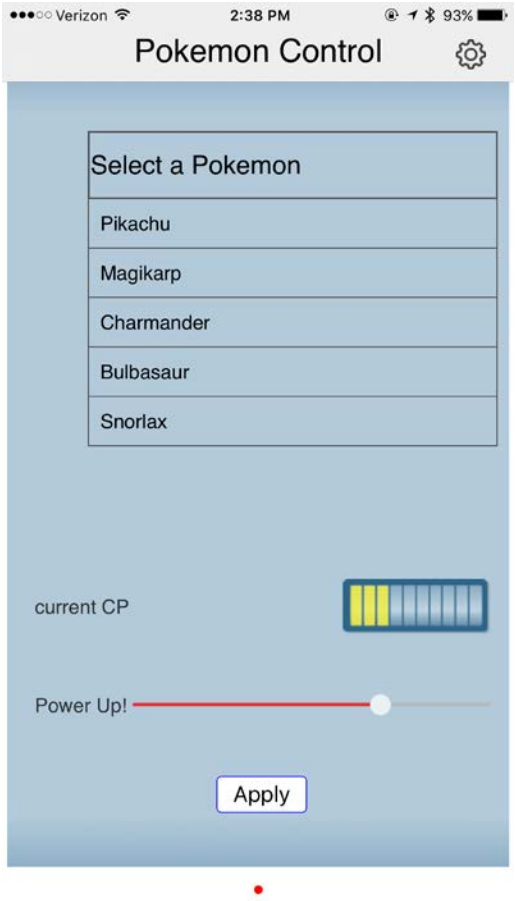
The Smart.IO app is designed to look for BLE UUIDs that conforms to a certain signature.

Figure 9 iOS Phone App Device Scanning Page



The Smart.IO API is designed to minimize BLE communication whenever possible. For example:

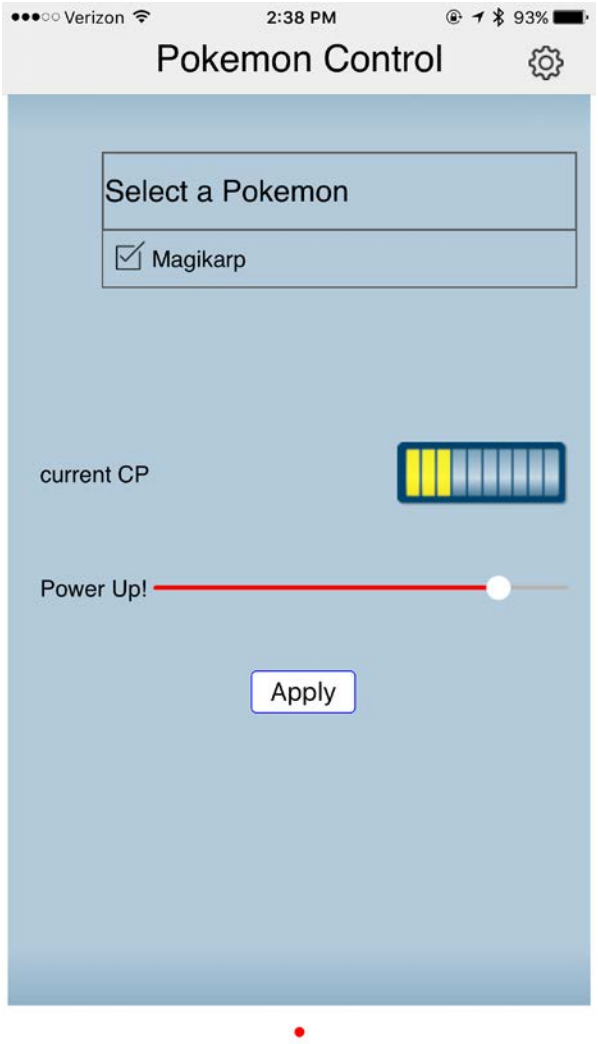
Figure 10 A Simulated Sample Pokémon Game Control App



In this screencap, while it is hard to see, the slider control is slightly dimmed and cannot be changed. This is because the embedded (UI) designer is using a Smart.IO API call to make the enabling of the slider depend on the state of the expandable list: when no simple Pokémon selection is currently being made, the slider is disabled. This dependency check and slider enabling is handled entirely by the app, and does not involve communication to the embedded firmware (which is expensive in terms of BLE communication overhead).

Here's the screencap of the state when one Pokémon has been selected:

Figure 11 Sample Pokémon Selection



Again, while it may be hard to see from a screencap, the slider control is now at full brightness, and the slider can be adjusted by the app user.

Caching - an In-App Purchase Option

To encourage widespread adoption of the Smart.IO technology, the basic app is offered free to users. However, one potential problem with having the UI generation in the embedded firmware, is that the UI code effectively resides on the embedded system, and must be sent over to the app. This can take a bit of due to BLE overhead.

Therefore: a handy solution is caching of the UI generation instructions. A complex UI page that may take 5-6 seconds to generate could be reduced to about a couple of seconds once caching has been enabled. NOTE: caching is only enabled in the paid-for app (an in-app purchase option is available.)

For a fully customized app, further time reductions can be facilitated so that the UI becomes effectively is as fast as a native app.

A Program Template

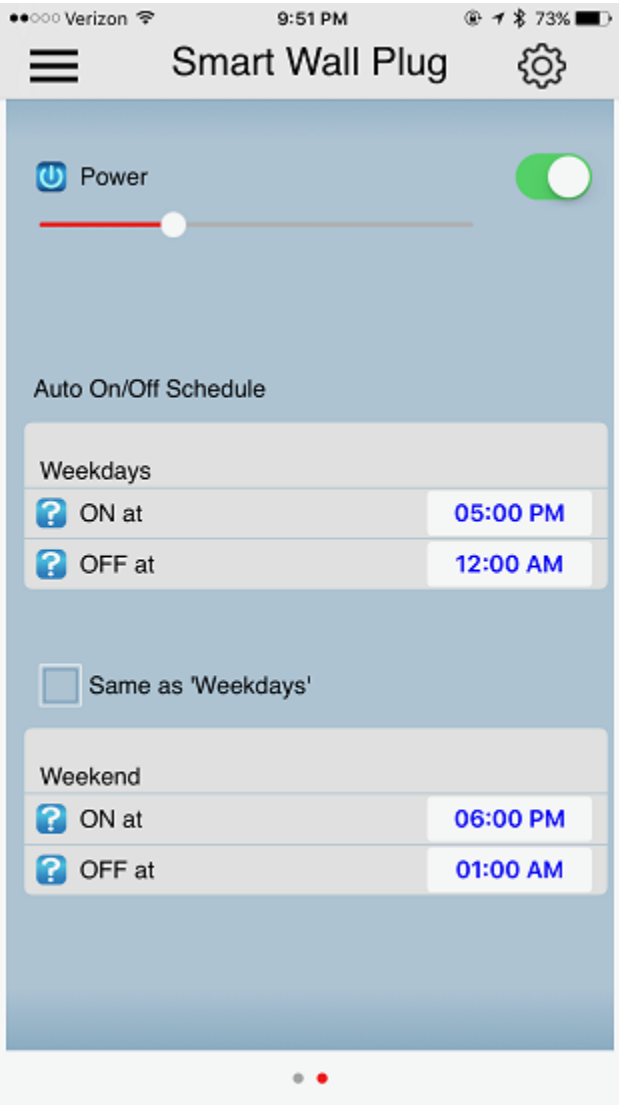
This chapter describes the basic structure of a program that uses Smart.IO.

Naming Conventions

All Smart.IO API functions start with the prefix `SmartIO_` , all other functions are supplied by the MCU firmware.

Example UI

The example used in this document is an embedded device that controls the power output of an FET controlled port using PWM. Such a device can be used to control the brightness level of an LED light or similar appliances. The example uses the ST32F411 MCU, and the FET is connected to PORTA pin 2. Timer2 is used to generate the PWM that drives the output.



In this document, we will only look at the upper two UI controls: the on/off Power button and the slider underneath. (The other portions of the UI are for implementing advanced scheduling features.)

As a reminder, this UI looks the same on both Android and iOS smartphones and tablets, without the MCU firmware engineer needing to know anything about the screen resolution or target OS.

Main Loop

The body of the main function looks like this:

```
int last_state = 0;
while (1)
    {
    while (last_state == connected)
        if (SPI_State == SPI_SMARTIO_ASYNC_REQUEST)
            SmartIO_ProcessUserInput();

    if (connected)
        {
        DelayMSecs(3);
        CreateUI();
        RestoreUIState();
        }
    else
        SaveUIState();
    last_state = connected;
    }
```

In the inner `while` loop, whenever the connection state is unchanged, the code checks for the condition "`if (SPI_State == SPI_SMARTIO_ASYNC_REQUEST)`". This condition is set by an interrupt handler, and becomes true only if there is asynchronous data coming from the Smart.IO module. When the condition is met, the conditional code calls the Smart.IO API function `SmartIO_ProcessUserInput()` to process the data.

If the connection changes, then the loop exits, and if this is a new connection, the firmware calls a function `CreateUI()` to create the UI.

The code above loops continuously, checking for the condition each time through the loop. Per usual embedded programming practice, in a device with power saving features, the MCU firmware can put the MCU in sleep mode and wake up the MCU whenever the connection state changes.

Save and Restore UI State

A side effect of letting the embedded firmware creating the UI via Smart.IO is that the state of the UI (e.g. the value of a slider, or the state of an on/off button) is not stored in the app, since the app can be run on different devices. One “logical” default potential option would be for the smartphone app to store the UI state “on the cloud”, but as that involves internet connectivity and therefore exposes the data to security risks, ImageCraft rejected this option.

Instead, the state is stored in the embedded system itself, in a permanent storage medium such as EEPROM. If the embedded system does not have its own EEPROM, two Smart.IO API functions provide read and write access to the Smart.IO module’s internal EEPROM. See the Appendix on how to implement the UI state functions, `SaveUIState()` and `RestoreUIState()`, using these API calls.

Note that if the embedded system is powered off, unless extraordinary measures are taken, then the above example scheme of calling the `RestoreUIState` function upon phone disconnect will (obviously) not work, as the MCU will stop running. This could impact the UI state when the device is restarted, so if this is important in the product design, then the firmware should save the UI state periodically, instead of only when the phone is disconnected, perhaps by using a timer.

Creating the UI

The following is an excerpt of the `CreateUI()` function. The important lines are highlighted in red,

```
void CreateUI()
    {
    tHandle p0, p1, p2, p3;
    tHandle u0, u1, u2, u3, u4, u5, u6, u7, u8, u9, u10;

    p0 = SmartIO_MakePage();
    SmartIO_AppTitle("Smart Wall Plug");
    u0 = SmartIO_MakeOnOffButton(0, 0, 1, Button1);
    SmartIO_AddText(u0, "Power");
    SmartIO_SetSliceIcon(u0, SMARTIO_ICON_POWER);
    u1 = SmartIO_MakeSlider(1, 0, 30, Slider1);
    SmartIO_UpdateSlider(u1+1, current_light_value);

    SmartIO_EnableIf(u0+1, u1+1, 0);
    …
    }
```

`tHandle` is a 16-bit integer type, defined as a C typedef. It is used by Smart.IO API to represent a “handle” to a UI object. You might notice that sometimes the handle value +1 (e.g. `u0+1`) is

used instead of just the handle value. This is an artifact of using a Smart.IO concept called a GUI Slice for device-independent GUI display. See other Smart.IO documentation for details.

The important functions are:

`SmartIO_MakePage()`: Smart.IO supports UI with multiple pages, so the first function in creating UI is to create a page using this function..

`SmartIO_AppTitle()`: changes the title, which is displayed at the top of the app screen.

`SmartIO_MakeOnOffButton()`: creates an on/off button control.

`SmartIO_MakeSlider()`: creates a slider.

`SmartIO_EnableIf()`: controls whether a set of controls is enabled or not, depending on the state of another UI control. In this example, the slider is only enabled if the on/off button is on. This feature allows a more responsive UI without requiring the MCU firmware to make every decision.

UI Callback Functions

When you call a Smart.IO API function to create an input control (such as a slider or an on/off button), you must also specify a callback function. When the end user (i.e. the person using the phone app) changes the value of an input control, the Smart.IO firmware calls the associated callback function with the new value as an argument. For example, in this sample UI:

```
u0 = SmartIO_MakeOnOffButton(0, 0, 1, Button1);
…
u1 = SmartIO_MakeSlider(1, 0, 30, Slider1);
```

The last argument of the two function calls `Button1` and `Slider1` are callback functions. The sample implementations look like these:

```
int current_light_level;

void Button1(uint16_t val)
    {
    if (val == 0)
        {
        // TURN OFF timer2 and clear output pin
        timer2.Disable();
        porta.MakeOutput(2, OSPEED_HIGH);
        porta.Clear(2);
        }
```

```
        else
            {
            // reactivate timer2
            porta.MakeAltFunction(2, 1, OSPEED_HIGH);
            timer2.Enable();
            timer2.ChangePWMDutyCycle(PA2_CHANNO, current_light_value);
            }
        }

    void Slider1(uint16_t val)
        {
        current_light_value = val;
        // CHANGE PWM value
        timer2.ChangePWMDutyCycle(PA2_CHANNO, current_light_value);
        }
```

The code should be self-explanatory. The low-level access to timer2 and PORTA are done using ImageCraft's JumpStart API, which makes it easy to perform such functions. Direct IO register access, or other support libraries such as ST's CubeMX generated code etc., can also be used.

Initial Setup

A single API call set up the Smart.IO environment:

```
extern void Connect_CB(void);
extern void Disconnect_CB(void);
…
SmartIO_Init(Connect_CB, Disconnect_CB);
```

`Connect_CB` is a callback function defined by the MCU firmware. It will be called when the Smart.IO phone app connects to the Smart.IO module.

`Disconnect_CB` is a callback function defined by the MCU firmware. It will be called when the Smart.IO phone app disconnects from the Smart.IO module.

These functions can be as basic as follow:

```
extern int connected;
extern int SPI_State;
…
void Connect_CB(void)
    {
    connected = 1;
```

```
    }

void Disconnect_CB(void)
    {
    SPI_State = SPI_IDLE;
    connected = 0;
    }
```

The main function is to set a global variable `connected`, indicating whether the phone app is connected or not. These functions work in conjunction with the main loop described above.

Summary

This is essentially all the code you need to create this example app UI. Notice there is no need to write wireless code, or to write the phone app yourself. All of that is taken care of by the Smart.IO toolkit.

More advanced UI features can be added. For example, the sample UI shows UI controls for allowing the end user to input auto on/off times based on weekday and weekend schedules. Implementing this is left as an exercise to the reader.

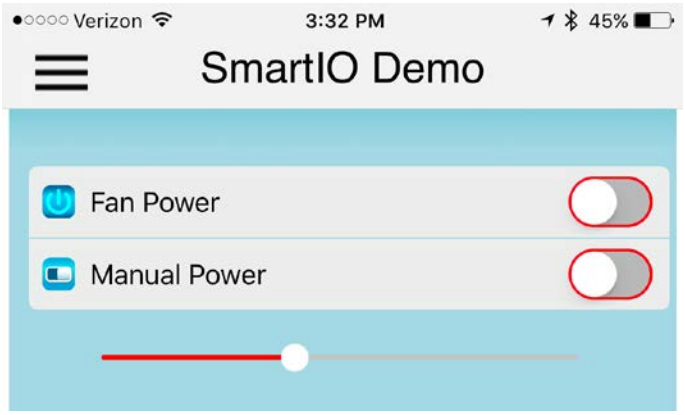
UI DESIGN

GUI Slice

To simplify building a UI that works for all phone devices with varying resolution and across two different types of OS (iOS and Android) with minimal effort from the firmware engineers, the Smart.IO UI toolkit uses the concept of GUI Slices.

A GUI slice, or slice [9], contains a UI element, plus optionally an informative icon and a descriptive text. For example, this is a screencap of three GUI slices:

Figure 12 Example with 3 GUI Slices



In vertical order:

1. An on/off button slice with a “power” icon and a label of “Fan Power”.
2. An on/off button slice with a “horizontal on/off” icon and a label of “Manual Power”.
3. A slider slice, whereby the end user can “slide” the control in either direction.

Note that the two on/off buttons are grouped together. The code fragment that generates the above looks like this:

```
// indentation here only to highlight primary GUI creation API
tHandle h1 = SmartIO_MakeOnOffButton(0, 0, 0, Button1);
    SmartIO_AddText(h1, "Fan Power");
    SmartIO_SetSliceIcon(h1, SMARTIO_ICON_POWER);
tHandle h2 = SmartIO_MakeOnOffButton(0, 0, 0, Button2);
    SmartIO_AddText(h2, "Manual Power");
    SmartIO_SetSliceIcon(h2, SMARTIO_ICON_H_ONOFF);
SmartIO_GroupObjects(0, h1, h2, 0);
tHandle h5 = SmartIO_MakeSlider(0, 0, 40, Slider1);
```

[9] There is another type of slice, the freeform slice, which will be introduced later. As GUI slice is the common case, so “slice” by itself always referred to a GUI slice.

The details of the functions will be explained later. GUI slices are laid out in the order they are called. For example, in the code fragment above, the order of the `SmartIO_Make...` calls correspond to the order of the UI elements on the page.

For cases where the firmware needs to have more precise placement control of GUI elements, a *Freeform Slice* may be used. (See description in later section.)

Virtual Screen Sizes and Screen Orientation

To simplify UI programming, the UI only operates in portrait orientation. This and the use of GUI slices and other Smart.IO features (e.g. built-in font support) allow a GUI to be created that looks optimal regardless of the OS and screen resolution of the target device. However, in some cases, it is important to have fine grain control regarding the placement of certain UI elements.

To address these issues, Smart.IO divides the screen into 320 virtual pixels wide, and maps the virtual pixels into the target device screen width. The app calculates the ratio between physical pixels to 320 virtual pixels and uses the same ratio to calculate the virtual pixel dimension of the height of the screen.

The height is almost never a factor in the UI as layouts are done using GUI slices, spacer slices, and the auto-balance command (the latter two are described later). The only instance where a problem may occur is when the firmware creates too many GUI elements for a single page. In that case, the app generates a vertical scroll bar for the user to navigate the full page. However, it is recommend that the firmware engineers to avoid this condition for a better looking UI.

List of UI Controls

This is a summary of all the available UI controls. More detailed information are given in the next section.

Input Elements

UI Element	Description	API Name
On/off button	An on/off switch	`SmartIO_MakeOnOffButton`
3-pos button	A switch with 3 positions	`SmartIO_Make3PosButton`
Incrementer	Increment / decrement control	`SmartIO_MakeIncrementer`
Slider	Slider	`SmartIO_MakeSlider`
Expandable list	A collapsible list to select one item. No more than 6 to 8 items should be on the list.	`SmartIO_MakeExpandableList`
Picker	A scrollable list to select one item. For use when large number of items are needed.	`SmartIO_MakePicker`
Multi-selector	A single or double rows of typically 2 to 6 items.	`SmartIO_MakeMultiSelector`
Number selector	Select a number with a low and high range	`SmartIO_MakeNumberSelector`
Time selector	Select a time in hours and minutes	`SmartIO_MakeTimeSelector`
Calendar selector	Select a calendar date	`SmartIO_MakeCalendarSelector`
Weekday Selector	Select a weekday (MON-SUN)	`SmartIO_MakeWeekdaySelector`
OK button	A single button. The label can be modified.	`SmartIO_MakeOK`
Cancel/OK button	Two button choice. The labels can be modified.	`SmartIO_MakeCancelOK`
OK “Link” button	Same as an :OK button” except that the it is linked to another UI element. See text below this table.	`SmartIO_MakeOKLinkTo`
Checkboxes	A group of checkboxes where multiple items can be selected.	`SmartIO_MakeCheckboxes`

Radio buttons	A group of radio buttons where one item can be selected.	`SmartIO_MakeRadioButtons`
Text entry	A box where text can be entered.	`SmartIO_MakeTextEntry`
Password entry	Same as “text entry” except that each character is replaced by * in the display.	`SmartIO_MakePasswordEntry`
Number entry	Same as “text entry” except that only numbers are accepted.	`SmartIO_MakeNumberEntry`

Output Controls

UI Element	**Description**	**API Name**
Text Box	Display text in a box with specified width (in virtual pixels). Also allow slice icon, slice label, and box alignment.	`SmartIO_MakeTextBox`
Multiline Text	Display text in a box that takes the full width of the screen	`SmartIO_MakeMultilineBox`
Counter	Display numeric digits in a bound box	`SmartIO_MakeCounter`
Progress Bar	Display progress (percentage) in a bar	`SmartIO_MakeProgressBar`
Progress Circle	Display progress (percentage) in a circular “bar”	`SmartIO_MakeProgressCircle`
Horizontal Gauge	Display quantity (percentage) in a horizontal gauge	`SmartIO_MakeHGauge`
Vertical Gauge	Display quantity (percentage) in a vertical gauge	`SmartIO_MakeVGauge`
Semicircular Gauge	Display quantity (percentage) in a semicircular gauge	`SmartIO_MakeSemiCircularGauge`
Circular Gauge	Display quantity (percentage) in a circular gauge	`SmartIO_MakeCircularGuage`
Battery Level	Display a battery icon with the charge level (20% increment)	`SmartIO_MakeBatteryLevel`
RGB Led	Display a “led” with on/off state, and one of the RGB (Red Green Blue) colors.	`SmartIO_MakeRGBLed`

Custom Horizontal Gauge	Display quantity (percentage) in a horizontal gauge with custom colors	`SmartIO_MakeCustomHGauge`
Custom Vertical Gauge	Display quantity (percentage) in a vertical gauge with custom colors	`SmartIO_MakeCustomVGauge`

There are other static UI elements, system menu API, etc. These will be explained in details later.

UI Elements

Input UI Elements

Before we look at the individual UI element, there is one auxiliary function we need to describe, for adding list items.

Adding List Items

For a UI element that contains multiple items, the function `SmartIO_AddListItem` adds a list item to the UI element. For example, after creating a checkboxes element (see below), the firmware calls this function to create the individual checkboxes with their labels.

handle	**label**
The handle of the UI element; e.g. return value of the “Make” function PLUS one.	Labels of the list items, in order of calls

Applicable UI elements:

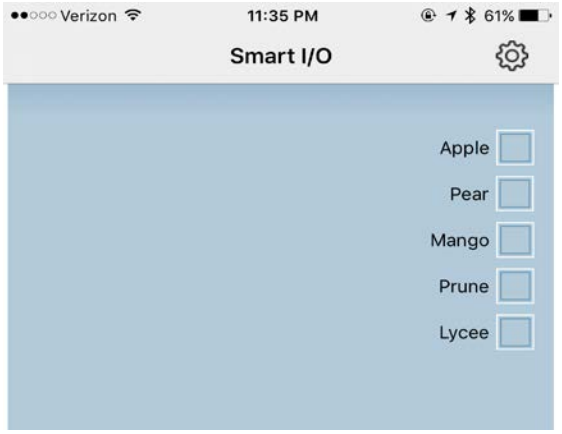
- Expandable list
- Picker
- Multi-Selector
- Checkboxes
- Radio buttons
- App menu

For example, to create a set of checkboxes with 5 items:

```
tHandle h = SmartIO_MakeCheckboxes(5, CheckboxesCB);
SmartIO_AddListItem(h+1, “Apple”);
SmartIO_AddListItem(h+1, “Pear”);
SmartIO_AddListItem(h+1, “Mango”);
SmartIO_AddListItem(h+1, “Prune”);
SmartIO_AddListItem(h+1, “Lychee”);
```

would create the following UI. As mentioned elsewhere, various `SmartIO_Make…` functions create a GUI slice that contains a UI element. The “handle” that is returned is the handle for the slice itself, and by design, handle+1 is the value for the handle of the UI element itself. Hence the argument to `SmartIO_AddListItem` is `h+1`.

Figure 13 Creating a 5-Element Checkboxes



Now we can move on to the actual UI elements.

On/Off buttons

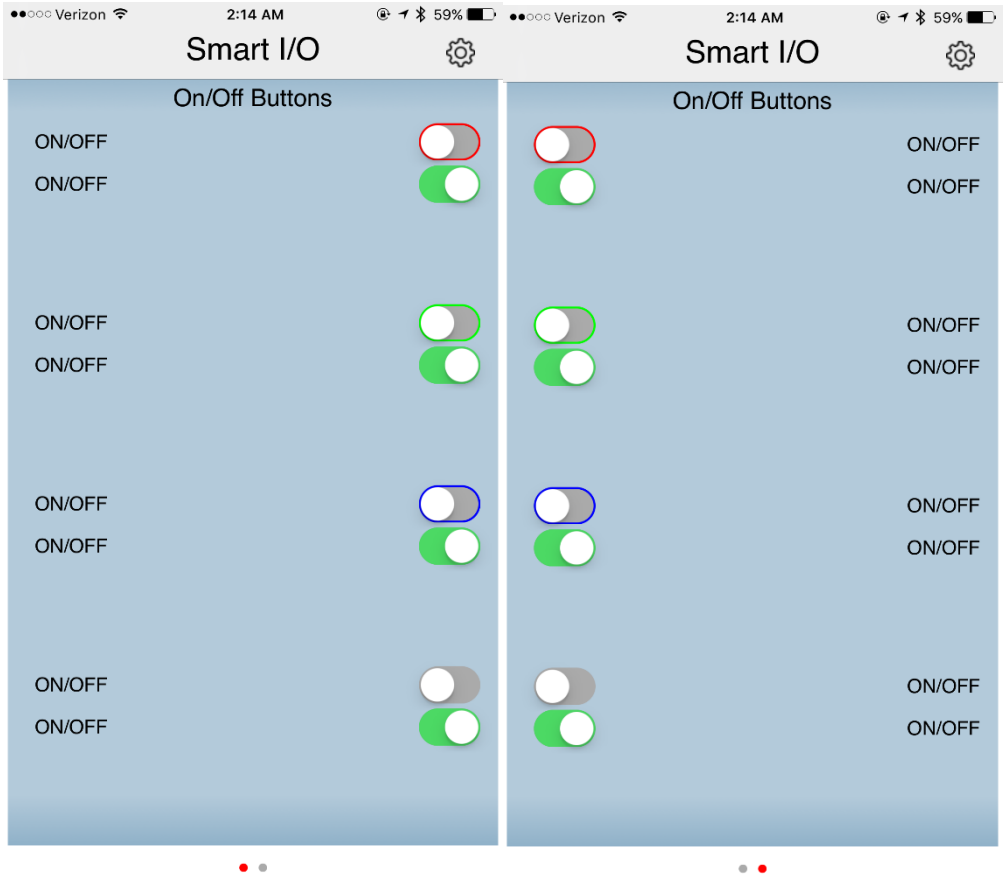
On/Off buttons allow the user to select an "on" or "off" position.

```
tHandle SmartIO_MakeOnOffButton
(uint16_t alignment, uint16_t variation, uint16_t initial_value,
 void (*callback)(uint16_t));
```

alignment	**variation**	**initial_value**	**callback**
0: right align 1: left align	0, 1, 2, 3	0: off 1: on	Callback function. Call with 0 or 1
return: handle value for the slice, handle+1 is the object ID of the UI element			

These screens show the variations 0, 1, 2, and 3 in vertical order. Both right and left alignments are shown. Later graphics will only show left alignment for brevity. For each set of two, the top one denotes the off state, and the bottom one denotes the on state.

Figure 14 Right and Left Alignment



3-Position Button

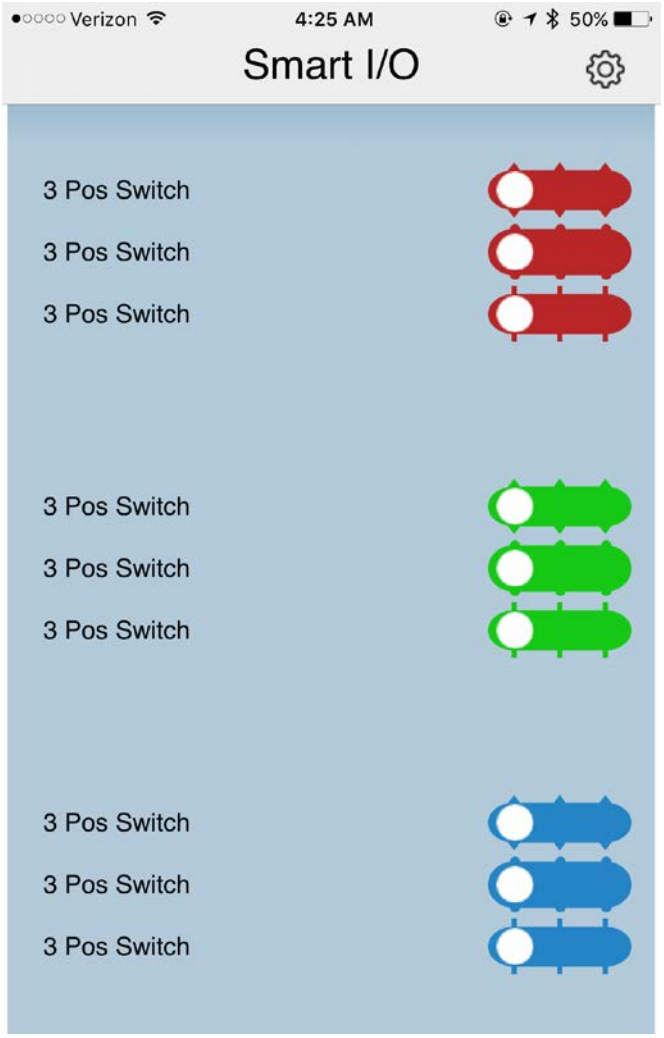
A 3-Position buttons allows one of three positions to be selected.

```
tHandle Make3PosButton
(uint16_t alignment, uint16_t variation, uint16_t initial_value,
 void (*callback)(uint16_t));
```

alignment	**variation**	**initial_value**	**callback**
0: right align 1: left align	0, 1, 2: red 3, 4, 5: green 6, 7, 8: blue	0: left 1: middle 2: right	Callback function. Call with 0, 1 or 2
return: handle value for the slice, handle+1 is the object ID of the UI element			

3-Position button comes in three different colors, each one has three variations in the shapes of the bumps around the edges.

Figure 15 3-Position Buttons



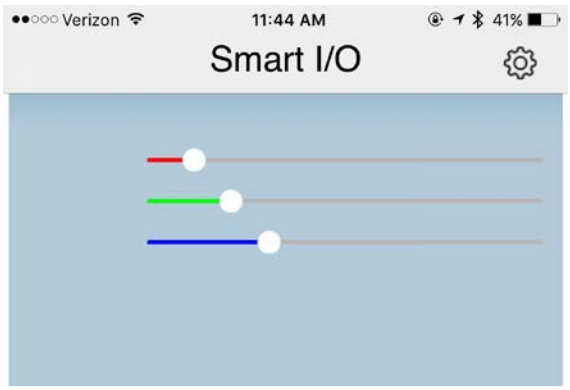
Slider

A slider allows the end user to select a value between the minimum value (which is usually 0%) and maximum (usually 100%).

```
tHandle MakeSlider
(uint16_t alignment, uint16_t variation, uint16_t initial_value,
 void (*callback)(uint16_t));
```

alignment	**variation**	**initial_value**	**callback**
0: right align 1: left align	0: red 1: green 2: blue	0 to 100	Callback function. Call with 0 to 100
return: handle value for the slice, handle+1 is the object ID of the UI element			

Figure 16 Sliders



Incrementer

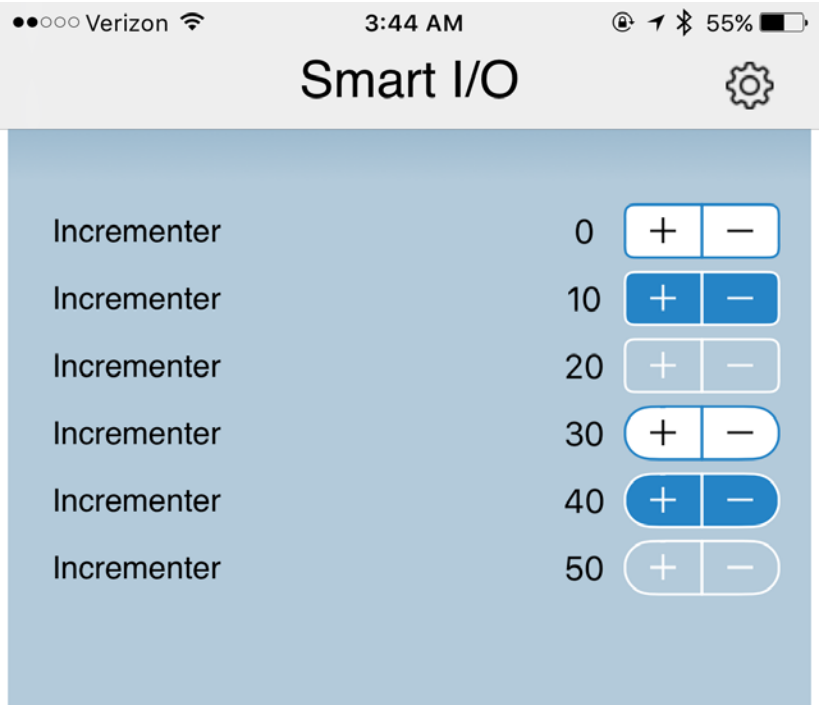
An incrementer displays a number and allows it to be incremented or decremented. The API allows low and high limits to be set.

```
tHandle MakeIncrementer
(uint16_t alignment, uint16_t variation, uint16_t initial_value,
 void (*callback)(uint16_t));
```

alignment	**variation**	**initial_value**	**callback**
0: right align 1: left align	0 to 5 with different fill colors and outline shapes. See screencap.	0 to 0xFFFFu	Callback function. Call with 0 to 0xFFFFu
return: handle value for the slice, handle+1 is the object ID of the UI element			

Variations 0 to 5 are shown in vertical order with different fill colors and box outline.

Figure 17 Incrementers



Expandable List

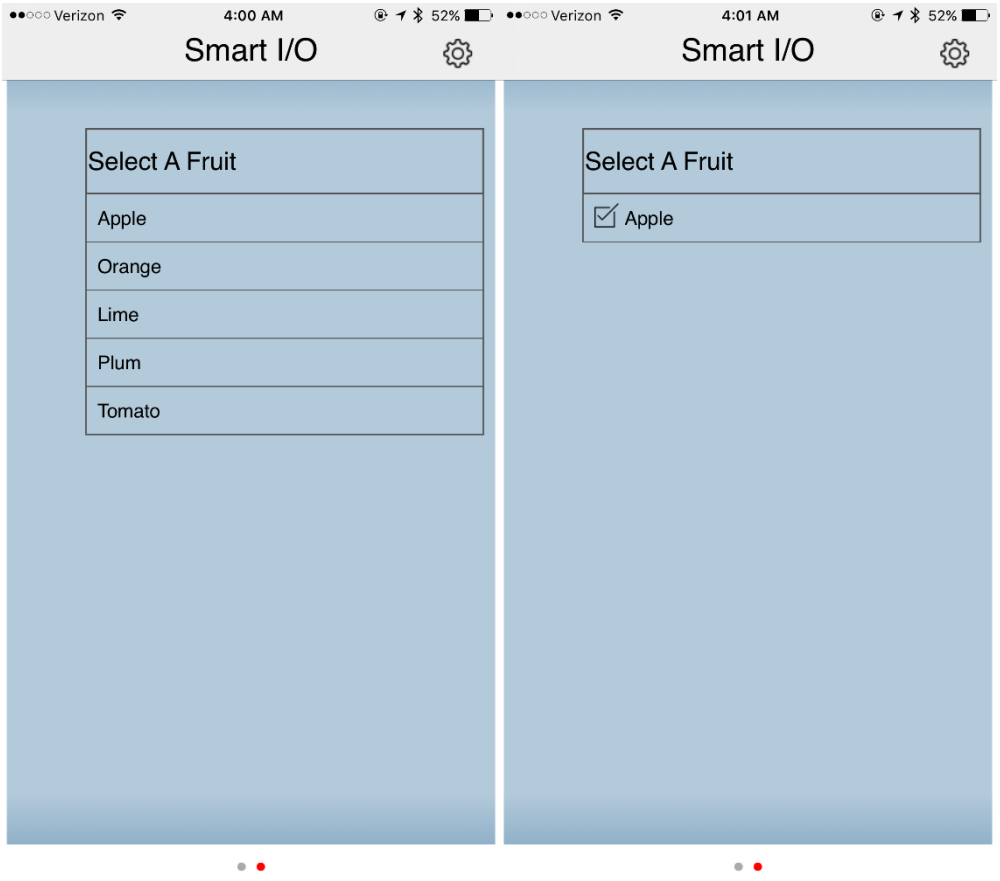
An expandable list allows no items or one item to be selected out of a list. It is recommended that an expandable list should contain no more than 5 to 6 list items. A longer list should use a "Picker" instead. Note that an expanded expandable list = no item is selected.

Use `SmartIO_AddListItem` to add entries to the list. Note: the first call to `SmartIO_AddListItem` is to create a label for the entire list and is not part of the `nentries` count. For example, in the screencaps below, "Select A Fruit" is the label for the list. Expandable list is the only UI with list items that has this label attribute.

`tHandle SmartIO_MakeExpandableList (uint16_t alignment, uint16_t nentries, void (*callback)(uint16_t));`		
`alignment`	**`nentries`**	**`callback`**
0: right align 1: left align	Number of entries	Callback function. Call with 0: no item is selected 1-nentries: selected item
return: handle value for the slice, handle+1 is the object ID of the UI element		

The left image is an expandable list (with “Select A Fruit” as its label) with no item selected. The right image is the same list with one item selected.

Figure 18 Expandable List: Unselected and Selected States



Picker

A picker is similar to an expandable list, except that the selection and presentation are different. The initial display is the selected or default element in a box with a drop down arrow next to it. When tapped, the list of elements is presented in a scrollable list. Note that while the initial state is no item is selected, once an item is selected, you cannot “select” no item, unlike an expandable list.

Use `SmartIO_AddListItem` to add entries to the list.

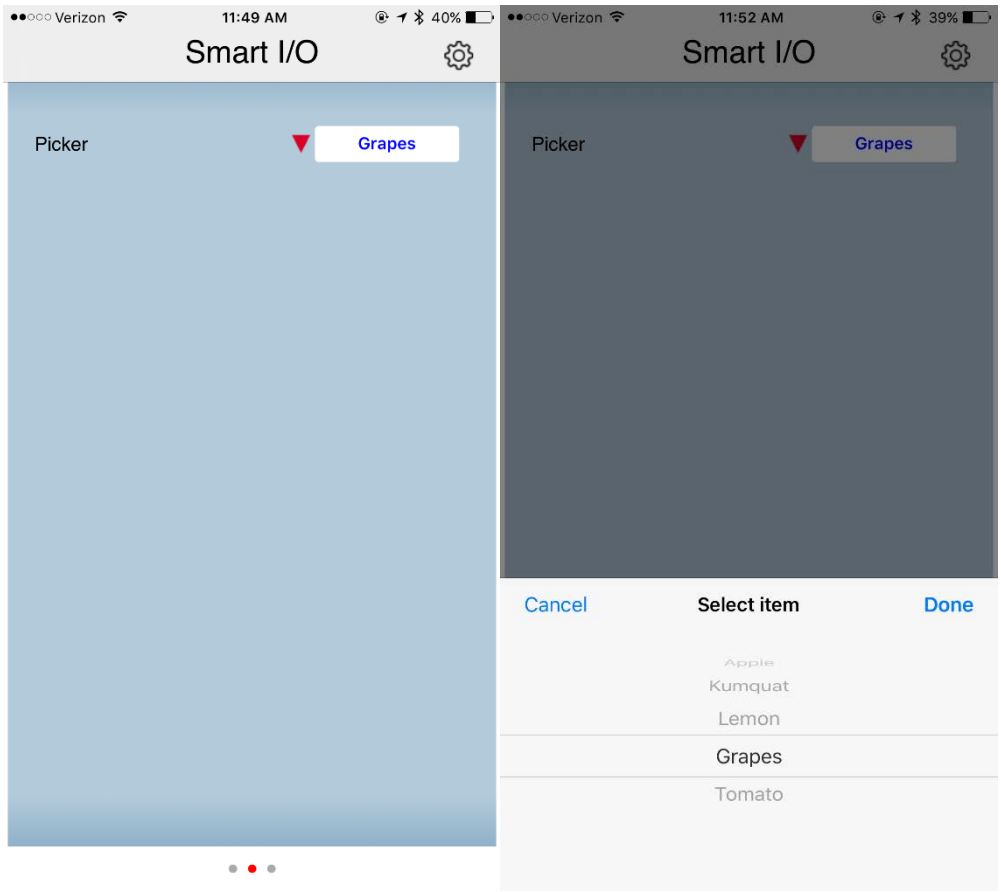
`tHandle SmartIO_MakePicker (uint16_t alignment, uint16_t nentries, void (*callback)(uint16_t));`		
`alignment`	**`nentries`**	**`callback`**
0: right align	Number of entries	Callback function. Call with

1: left align		1-nentries: selected item
return: handle value for the slice, handle+1 is the object ID of the UI element		

In the following, the left image is the unexpanded state of a picker. On the right is what the UI looks like when the picker is touched: a scrollable list shows up at the bottom allowing the app user to scroll and select an item.

Initially, the list can have no item selected, but once a selection is made, there is no provision to unselect all items (unless the firmware specifically adds a “-none-” item, which is still not quite the same).

Figure 19 Picker: Tap to Open Up the List of Items



Multi-Selector

A multi-selector is for a selection list of 2 to 6 items. The API includes an option to specify whether multiple items may be selected at a time, or whether only a single item may be selected.

Since multiple items may be selected, the argument to the callback is a 16-bit bitmask where bit 0 (LSB, least significant bit) represents whether item one is selected, and bit 1 (immediate left of bit 0), represents whether item two is selected, etc. For example, the bit mask 0x21 means that both item one and item six are selected.

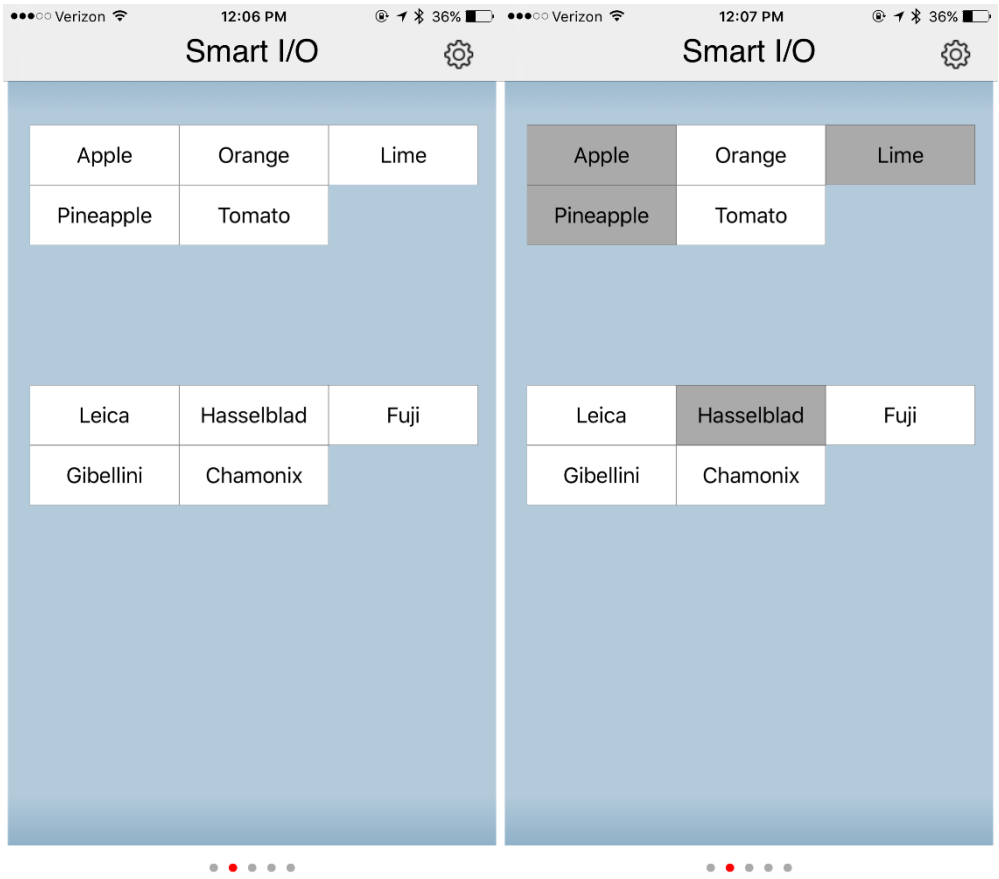
Use `SmartIO_AddListItem` to add entries to the list.

```
tHandle SmartIO_MakeMultiSelector
(uint16_t alignment, uint16_t nentries, uint16_t isSingleSelectOnly,
 void (*callback)(uint16_t));
```

alignment	**nentries**	**isSingleSelectOnly**	**callback**
0: right align 1: left align	Number of entries	0: allows multiple items to be selected 1: allows only one item to be selected	Callback function. Call with 1-nentries: selected item
return: handle value for the slice, handle+1 is the object ID of the UI element			

In the screencap below, the top selector allows multiple items to be selected, whereas the bottom selector only allows a single item to be selected. The left image shows two multi-selectors in their unselected states. On the right, some items have been selected. Although not obvious, the second control only allows a single selection, so only one item can be selected. The app user can deselect an item by tapping it again so it is possible that no item is selected in a picker. The callback function will be called whenever the selection changes.

Figure 20 Multi-Selectors: Unselected and Selected States



Number Selector

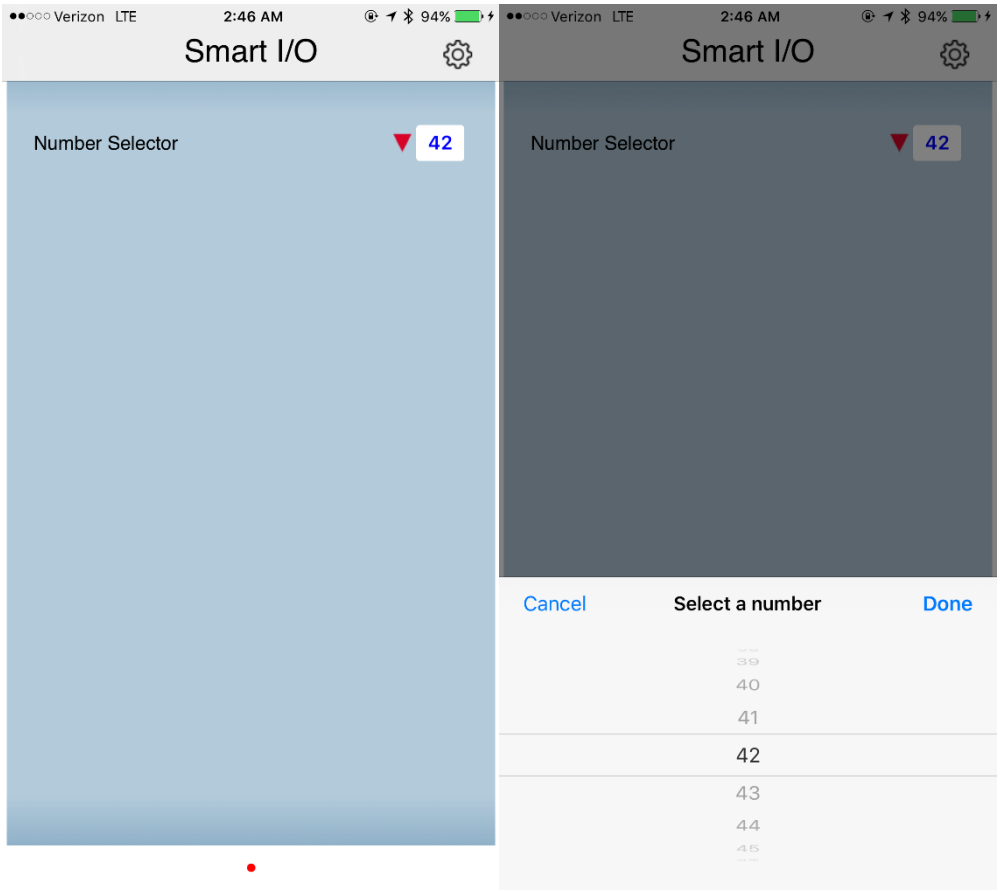
A number selector is a special kind of picker that displays a range of numbers. The low and high values of the range can be specified

```
tHandle SmartIO_MakeNumberSelector
(uint16_t alignment, uint16_t default_val, uint16_t low, uint16_t high,
 void (*callback)(uint32_t));
```

alignment	**default_val**	**low**	**high**	**callback**
0: right align 1: left align	The initial value to display	The lower bound of selectable numbers	The upper bound of selectable numbers	Callback function. Call with the selected number
return: handle value for the slice, handle+1 is the object ID of the UI element				

Since a number selector is a special case of the picker, the left image shows the selector in an unexpanded state. The right image shows what happens when the box is tapped: a scrollable list is shown with the range of numbers.

Figure 21 Number Selector: Unexpanded and Expanded States



Time Selector

A time selector allows the time of day to be selected. The initial time is specified as a C string in the form of “HH MM” in 24 hour format. Selected time is returned as a string in the same 24-hour format. The function supports two variations: 0 means that the display time is in 12-hour AM/PM format, and 1 means that the display time is in 24-hour format.

```
tHandle SmartIO_MakeTimeSelector
(uint16_t alignment, uint16_t display_24h, char *initial_value,
 void (*callback)(uint16_t hh, uint16_t mm));
```

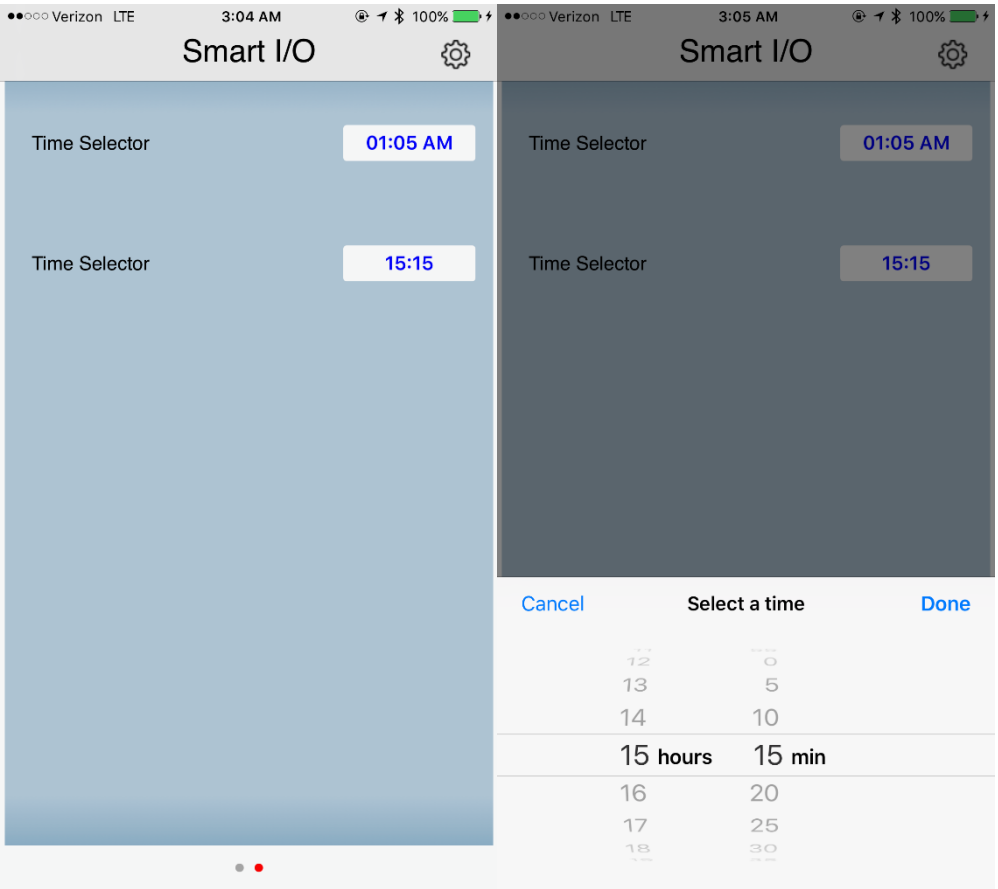
alignment	**display_24h**	**initial_value**	**callback**
0: right align 1: left align	0: display AM/PM 1: do not display AM/PM	The initial time in “HH:MM” 24-hour format	Call with the selected hours and minutes in 24-hour format

return: handle value for the slice, handle+1 is the object ID of the UI element

The left image shows 2 time selectors, the top one with AM/PM display and the bottom one in 24-hour display mode. When the entry box is touched, a time selection scrollable list is shown.

On iOS, currently times are selectable by 5-minute increments to make choice selection easier. We could add an option to allow single minute increments if there is demand for such a feature. Conversely, the Android app allows minute selection.

Figure 22 Time Selector: Tap to Select a Time Value



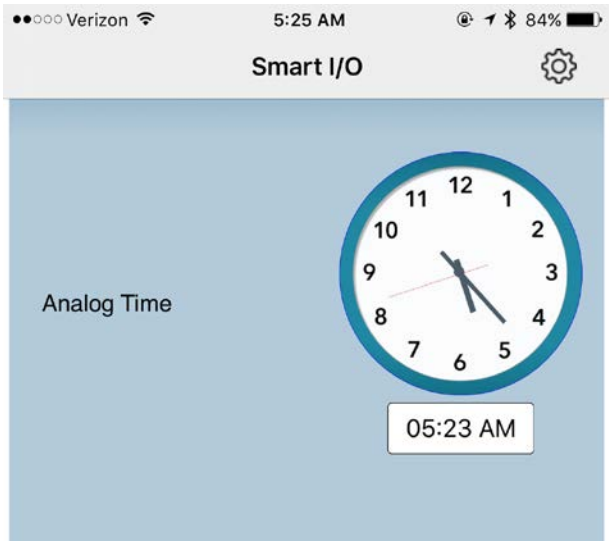
Analog Time Selector

An analog time selector displays an analog clock face, and allows a time to be selected by the end user by dragging the hour or minute hands.

Note: all arguments to this function are exactly the same as `SmartIO_TimeSelector`; the only difference is the UI visual which the functions create.

tHandle SmartIO_MakeAnalogTimeSelector (uint16_t alignment, uint16_t display_24h, char *initial_value, void (*callback)(uint16_t hh, uint16_t mm));			
alignment	**display_24h**	**initial_value**	**callback**
0: right align 1: left align	0: display AM/PM 1: do not display AM/PM	The initial time in “HH:MM” 24-hour format	Call with the selected hours and minutes in 24-hour format
return: handle value for the slice, handle+1 is the object ID of the UI element			

Figure 23 Analog Time Selector



Calendar Selector

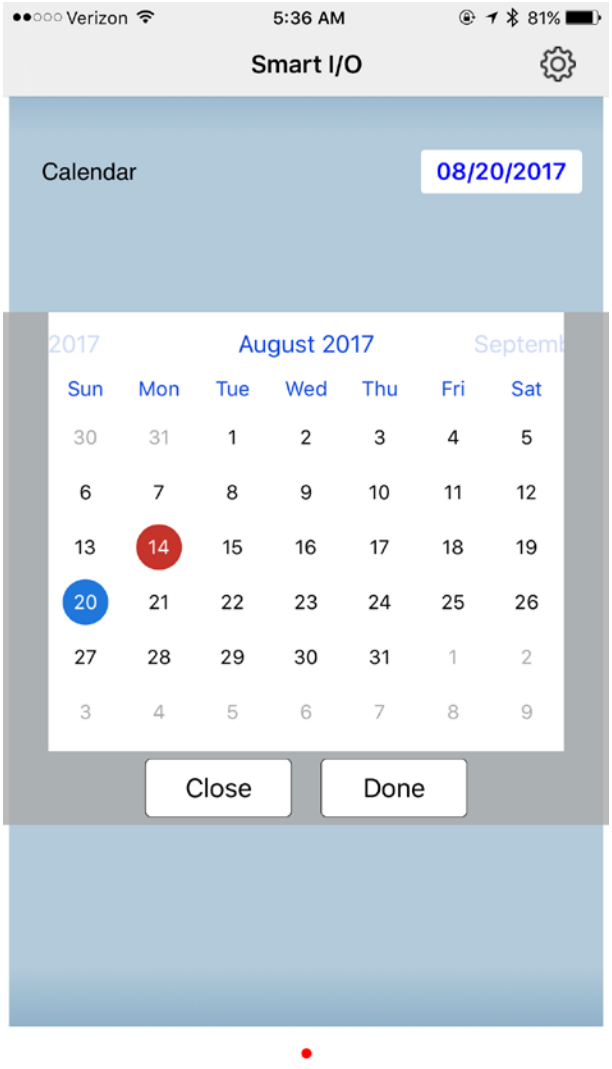
A calendar selector displays a date in the form of “MTH/DD/YYYY”, and when tapped, displays a full calendar. “MTH” is either a numeric value from 1 to 12 or a 3-letter month code.

tHandle SmartIO_MakeCalendarSelector (uint16_t alignment, char *initial_value, void (*callback)(char *));		
alignment	**initial_value**	**callback**
0: right align 1: left align	The initial date in “MTH DD YYYY” C string format. MTH can be a number 1 to 12, or the 3-letter code for months, e.g. JAN, FEB, and so on. DD is a number from 1 to 31.	Callback function. Call with C string in the form of “weekday MTH DD YYYY” where all 4 fields are numbers. ImageCraft provides a utility

	Note that the native OS may not support a calendar which is far in the past or future.	function `SmartIO_ConvertCalendarDay` to extract the return values as needed.
return: handle value for the slice, handle+1 is the object ID of the UI element		

The current date is in red and the selected date is in blue. The selected date is returned to the firmware in “weekday month date year” format where “weekday” is one to seven (one being Monday) and “month” is 1 to 12. You can tap to select a date and scroll to different months and years.

Figure 24 Calendar Selector



Weekday Selector

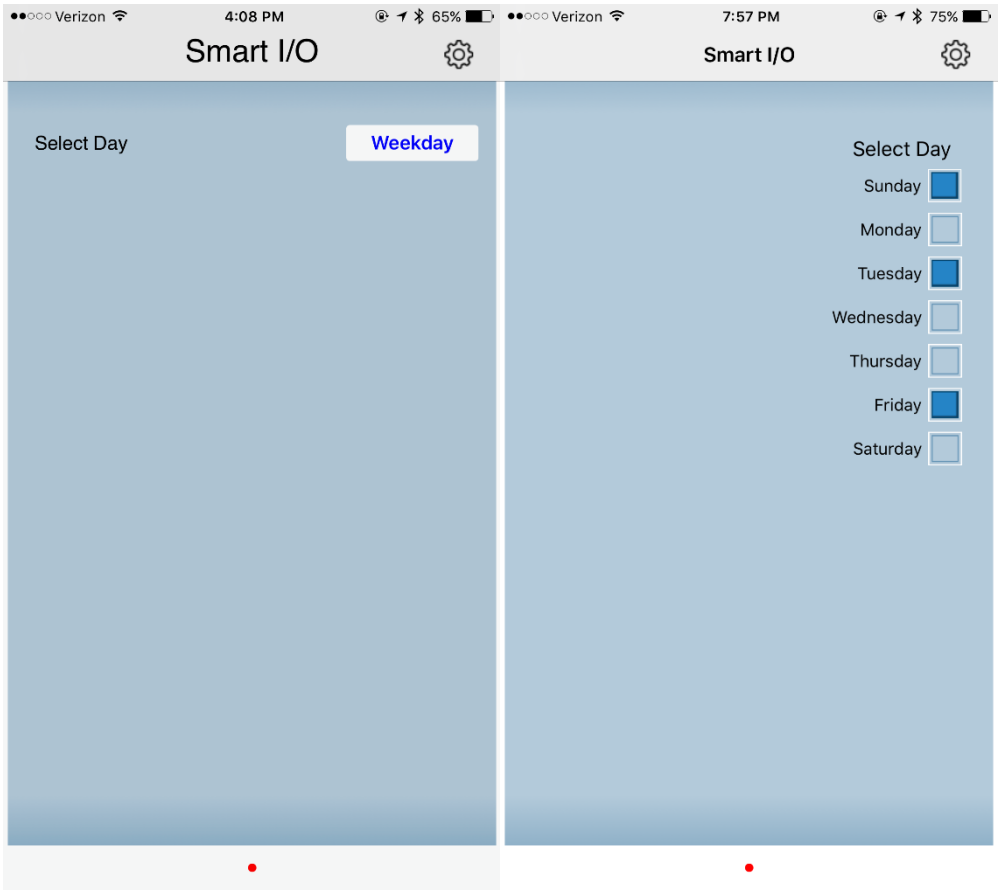
A weekday selector displays a list of weekdays. Multiple days may be selected. This is a special case of "checkboxes".

Since multiple days can be selected, a bitmask is used; bit 0 is Monday, bit 1 is Tuesday, etc. For example, 0x5 is Wednesday and Monday.

<table>
<tr><td colspan="3"><code>tHandle SmartIO_MakeWeekdaySelector
(uint16_t alignment, uint16_t initial_value, void (*callback)(uint16_t));</code></td></tr>
<tr><td><code>alignment</code></td><td><code>initial_value</code></td><td><code>callback</code></td></tr>
<tr><td>0: right align
1: left align</td><td>Bitmask of initial_value.</td><td>Callback function. Call with Bitmask of selected days.</td></tr>
<tr><td colspan="3">return: handle value for the slice, handle+1 is the object ID of the UI element</td></tr>
</table>

The left image shows an unexpanded weekday selector. When the box is touched, the selection box pops up and zero or more days can be selected.

Figure 25 Unexpanded and Expanded Day Selector

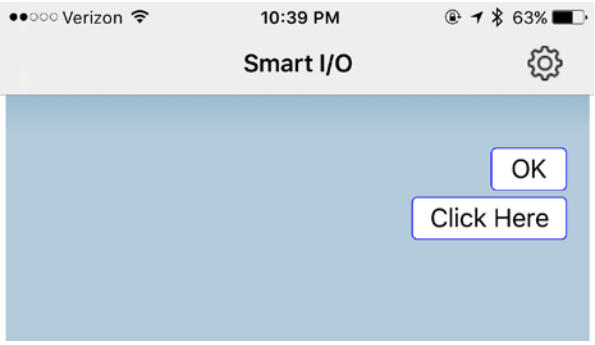


OK Button

An “OK” button is a clickable box. The callback function is called whenever the box is tapped.

`tHandle SmartIO_MakeOK (uint16_t alignment, char *text, void (*callback)(uint16_t));`		
`alignment`	**`text`**	**`callback`**
0: right align 1: left align	Label for the box. If null, then the word “OK” is used.	Callback function. Always call with the value 1.
return: handle value for the slice, handle+1 is the object ID of the UI element		

Figure 26 OK Button



OK_LINKTO Button

An OK_LINKTO button is exactly the same as an “OK” button, except that the function is called with the handle of a Popup UI (see below). When tapped, the Popup UI is displayed automatically.

For example, the firmware may create a popup informational page, and associate it with an OK_LINKTO box. Whenever the box is tapped, the informational page will be shown, without interaction from the embedded firmware. This reduces communication overhead between the App and the embedded system, resulting in a more responsive user experience.

A callback function is still called in case the embedded firmware wishes to register such event. For example, the OK button may be linked to a EULA (End User Legal Agreement) page and the embedded firmware may note the fact that the app user has opened the page.

```
tHandle SmartIO_MakeOKLinkTo
(uint16_t alignment, tHandle popup_handle, char *text,
 void (*callback)(uint16_t));
```

alignment	**popup_handle**	**text**	**callback**
0: right align 1: left align	Handle of the popup UI to display when tapped.	Label for the box. If null, then the word “OK” is used.	Callback function. Always call with the value 1.
return: handle value for the slice, handle+1 is the object ID of the UI element			

CANCEL/OK Button

CANCEL/OK is similar to OK except that one of two choices can be made. To specify the labels for both choices, use ‘|’ a divider, e.g. “Turn Left|Turn Right”.

```
tHandle SmartIO_MakeCancelOK
```

`(uint16_t alignment, char *text, void (*callback)(uint16_t));`		
alignment	**text**	**callback**
0: right align 1: left align	Label for the box. If null, then the words “CANCEL” and “OK” are used.	Callback function. 1: OK is clicked 2: Cancel is clicked

Checkboxes

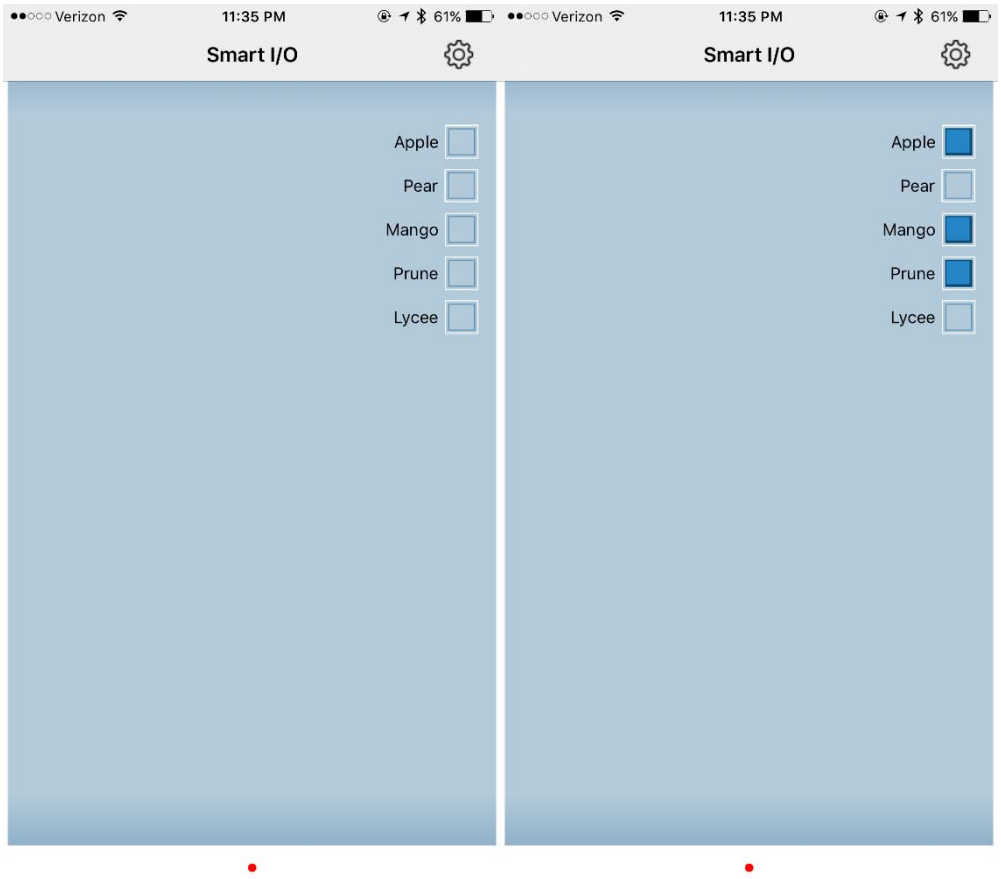
Checkboxes allow multiple items to be selected. Since multiple items may be selected, the argument to the callback is a 16-bit bitmask where bit 0 (LSB, least significant bit) represents whether item one is selected, and bit 1 represents whether item two is selected, etc. For example, the bit mask 0x21 means that item one and item six have been selected.

Use `SmartIO_AddListItem` to add entries to the list.

`tHandle SmartIO_MakeCheckboxes` `(uint16_t alignment, uint16_t nentries, void (*callback)(uint16_t));`		
alignment	**nentries**	**callback**
0: right align 1: left align	Number of entries	Callback function. Call with bitmask of selected items.
return: handle value for the slice, handle+1 is the object ID of the UI element		

The left image shows a list of checkboxes in the unselected state. The right image shows three items which have been selected.

Figure 27 Checkboxes: Unselected and Selected States



Radio Buttons

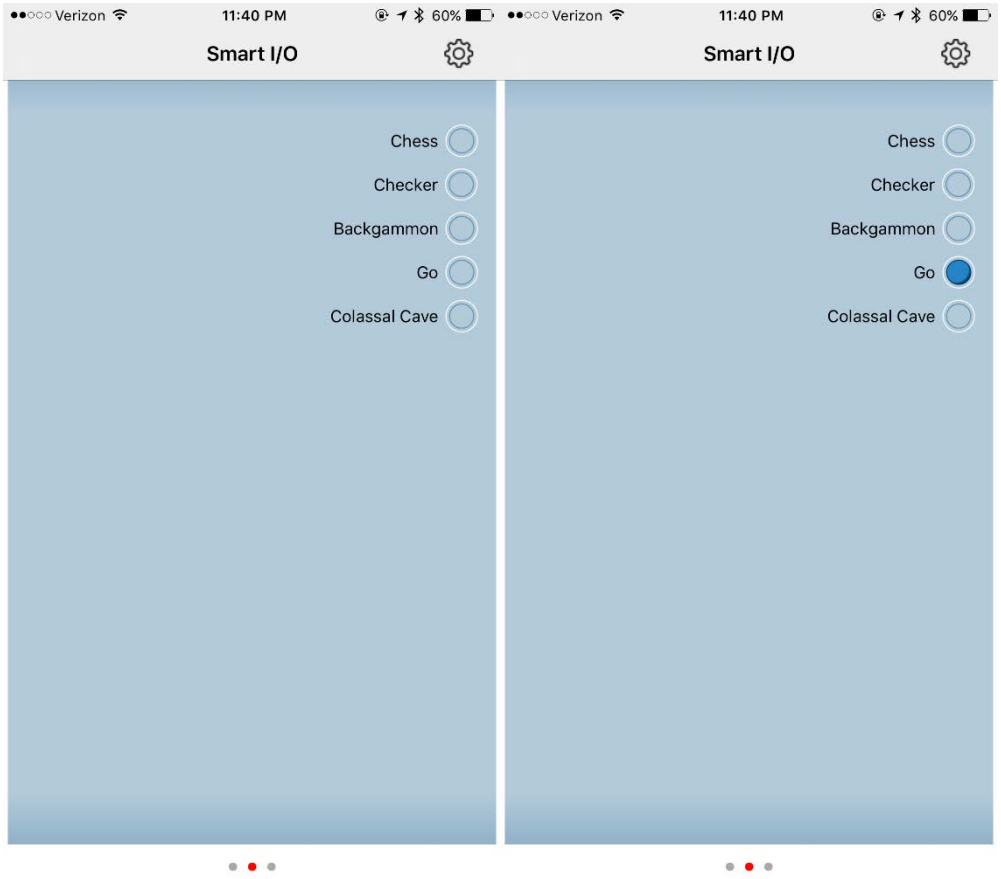
Radio buttons allow one of many items to be selected. Once any item has been selected, then one item will always be selected.

Use `SmartIO_AddListItem` to add entries to the list.

`tHandle SmartIO_MakeRadioButtons (uint16_t alignment, uint16_t nentries, void (*callback)(uint16_t));`		
`alignment`	**`nentries`**	**`callback`**
0: right align 1: left align	Number of entries	Callback function. Call with 1..nentries of the selected item.
return: handle value for the slice, handle+1 is the object ID of the UI element		

The left image shows a set of radio buttons. The right image shows that one item has been selected.

Figure 28 Radio Buttons: Unselected and Selected States



Text Entry

A text entry field is for entering a line of text. The input is sent to the callback function when the app user taps “Done” on the virtual keyboard (see below).

```
tHandle SmartIO_MakeTextEntry
(uint16_t alignment, uint16_t isRoundCorners, uint16_t nlines,
 void (*callback)(char *));
```

alignment	**isRoundCorners**	**nlines**	**callback**
0: right align 1: left align	0: input box has sharp corners 1: input box has round corners	Number of lines in the entry box. Default is one.	Callback function. Call with entered text.

return: handle value for the slice, handle+1 is the object ID of the UI element

The following image shows that you can type in the text entry with the device's virtual keyboard. Currently, the app user cannot enter more than one line of text in this field.

Figure 29 Text Entry Input Box

Password Entry

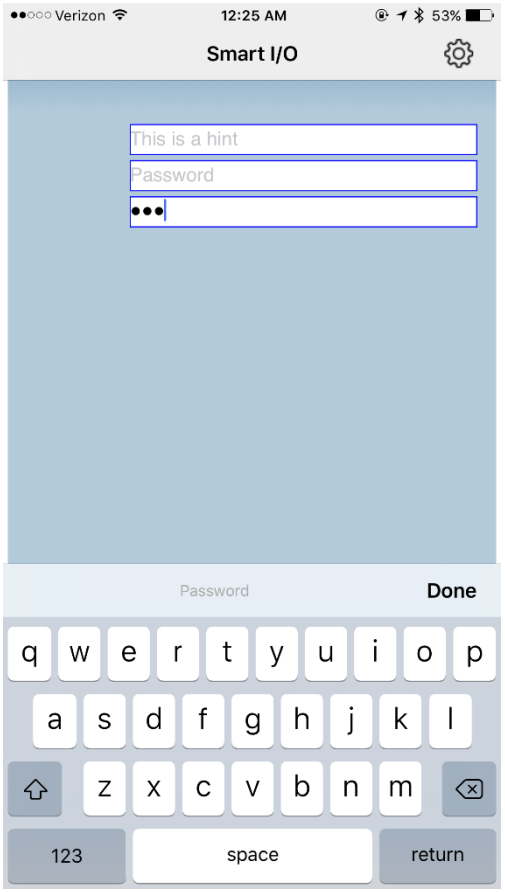
A password entry field is similar to text entry, except that each character is replaced by a '*' or a '•' after a slight delay as it is typed. The text is transferred to the Smart.IO using BLE protocol, so it relies on the BLE encryption capability to protect the text.

An empty password entry box has the word “Password” grayed out in the box. You can change it to any text you want by using the `SmartIO_AddText` [10] function.

`tHandle SmartIO_MakePasswordEntry (uint16_t alignment, void (*callback)(char *));`	
`alignment`	**`callback`**
0: right align 1: left align	Callback function. Call with entered text.
return: handle value for the slice, handle+1 is the object ID of the UI element	

Three password fields are shown here. The first one shows a hint that that the firmware can add to the field using the `SmartIO_AddText` function. The second one shows the default text. The third one shows the password as being entered with each character replaced by a dot (•).

Figure 30 Password Entry



[10] Per usual, keep in mind that the handle value that you pass to the `SmartIO_AddText` function is the returned value of the `SmartIO_MakePasswordEntry` PLUS one to access the UI handle value.

API INTRODUCTION

API Categories

The Smart.IO API is divided into these categories:

1. System / BLE initialization
2. Creating and managing UI elements
3. Updating the values of UI elements
4. Callback function mechanisms to accept end user input
5. System commands such as storing and retrieving UI values in Smart.IO EEPROM
6. Miscellaneous commands

The UI elements consist of:

1. Input elements such as on.off switches, sliders, text entry boxes, etc.
2. Output elements such as gauges, progress bars, text display, and even RGB LEDs, etc.
3. Static and other elements such as labels, informative icons, page navigation, etc.

Terminology

In this document, the following terms are used:

- *API* refers to the Smart.IO API
- *App* refers to the smartphone Smart.IO app running on either iOS or Android
- *Custom App* refers to a version of the Smart.IO app customized to a specific firmware/product
- *End User* refers to the person using the phone app
- *User* refers to the embedded programmer
- *Firmware* refers to embedded firmware written by the user.

App Version

The Smart.IO app can work with any Smart.IO device by default. There are three basic versions of the app:

1. Free generic version. This supports any Smart.IO-enabled product (unless explicitly prevented by a specific product) with no optimization.
2. Paid (low cost) version. This includes caching support to eliminate some UI creation BLE overhead.
3. Customized version. Embedded vendors may commission ImageCraft to customize a version of the app that is specific to their product with their own logos, branding, etc.

Visit our page https://imagecraft.com/smartio/ for details.

Error Conditions

The app detects error conditions and returns error statuses to the calling firmware; for example, if the firmware specifies an incorrect handle value for an operation. When designing the UI, it is up to the firmware engineers to ensure that the error conditions are checked, either by checking the return values from the app, or by running the generated UI and checking its operations. During development, the firmware may enable debug output through the UART port. (See details later.)

Memory Blocks

There are three processor spaces in the system (the user MCU, the Smart.IO module, and the smartphone itself), and pointers are not passed between the processor spaces as it makes no sense. Thus, a string or blocks of text must be copied in its entirety between the processors even though often the standard C notation of “char *” is used to denote a string. This applies to function argument and function results.

The host interface layer allocates a block of host SRAM as *transfer memory* to store incoming text data from the Smart.IO module to the host firmware. The size of the SRAM block is #define’d as HOST_SRAM_POOL_SIZE in smartio_api.h (default is 512 bytes), which should be sufficient for most uses in Smart.IO. You may modify this value, up to a maximum value of 4K bytes (and rebuild the host interface layer) to suit your system’s requirements. This value must also be sent to the Smart.IO firmware as an argument to the initialization function. See the `SmartIO_Initialize` API description.This block of memory is also used by the Smart.IO EEPROM read function to hold the read-out values.

In addition, Smart.IO module uses an internal buffer of 4K bytes to store any text that is transferred between the app and the host MCU. This is a hard limit that is independent of HOST_SRAM_POOL_SIZE and cannot be changed by the firmware.

UI Cache

Since the Smart.IO app is generic in nature, the specific UI must be built by the firmware issuing Smart.IO API function calls. Having a great-looking UI is important, but not if there is a lag for the UI to show up! Also, using Bluetooth technology means that the device and the phone must be paired, adding to the initial response time.

To address the issue with UI construction overhead, the paid and customized version of the Smart.IO app caches the UI creation instructions, so that after the first run, BLE overhead is significantly reduced in subsequent runs.

The customized app also allows transparent pairing of the app to a specific product, so that the pairing overhead can be eliminated as well.

Local Storage

Currently, Smart.IO does not (yet) allow storage and accessing data “on the cloud” [11], and as multiple phone devices may be used to access the embedded product, the firmware should store the UI element values (e.g. the state of an on/off switch) in local storage so it can update the UI element properly at startup. In the case that host hardware does not have persistent data storage, the Smart.IO contains a 2K-byte EEPROM that may be used by the firmware for this purpose. (The API is described later.)

API Dataflow

When the embedded firmware calls a Smart.IO API function, the behavior is as if the firmware has made a regular function call: the command is carried out, and some point later the function returns a result.

Internally, the host interface layer transfers the command to the Smart.IO module via the SPI interface. After some massaging, the command is transferred to the phone app via the BLE interface. After running the command, the phone app returns the result of executing the command to the Smart.IO module (again via the BLE interface). The Smart.IO firmware transfers the result to the host interface layer, which returns the result to the firmware call.

Firmware calls an API function
```
Firmware → (Smart.IO API) → SPI → Smart.IO firmware → (BLE) → App
```

The App (eventually) returns a result
```
App → (BLE) → Smart.IO firmware → SPI → (host interface layer) → Firmware
```

API calls are blocking; i.e. from the firmware point of view, it is just making a function call that returns a result. The SPI and BLE transfer overhead, and the time that the app needs to execute the command, are transparent to the firmware.

Callback Functions

Callback functions are used to receive new values from input UI elements. When the firmware creates an input element, it provides the address of a function that the host interface layer will invoke when the end user interacts with that UI element. A callback function accepts either an integer or a string as its parameter, depending on the UI element.

By necessity, a callback function is called from inside an interrupt handler (which is a part of the host interface layer). Therefore, to minimize disruption to the firmware execution, either the callback functions should return as soon as possible, or the firmware should be designed to use

[11] “Cloud access” involves strong security measures. Smart.IO will be further developed with robust security in mind for Cloud access in future releases.

an interrupt-driven execution model. This is a standard problem with dealing with interrupt driven code in firmware. Typically, a callback function that needs to run many instructions sets a global flag that is picked up by the normal execution flow later.

While the end user may modify an input element at any time, the Smart.IO firmware and the host interface layer are written such that callback functions are not called during the execution of an API call. The Smart.IO firmware stores up to a maximum of (16) callback invocations in its internal memory. The callback functions are invoked in the order in which they arrive.

While there is no mechanism for a callback function to indicate to the Smart.IO firmware that it is finished processing, callback functions are invoked on the host MCU side by interrupts. As most MCUs do not allow nested interrupts or the same interrupt to interrupt itself, this is not a problem.

Data Types, Strings and Transfer Memory

Smart.IO API defines the following data types:

- *tHandle* is a 16-bit unsigned integer. When the firmware creates a UI object, a *handle* to the object is returned by the app so the object can be referenced later.
- *tStatus* is a 16-bit signed integer. Some API functions return status indicating success or failure. In most cases, zero denotes success and non-zero denotes failure. Depending on the API, the status may have different failure values. (See smartio_api.h for details.)
- *label* is a string, e.g. “char *”

The “native” argument type is a 16-bit unsigned integer (uint16_t) as that can hold most values in the system. Using 16 bits versus 32 bits reduces unnecessary transfer overhead over the SPI and BLE. This also makes it friendlier to 8-bit embedded systems, such as the ones using the Atmel AVR MCU.

32-bit integers are used on occasion; for example, the “color” argument (e.g. font color) is a 32-bit value.

API Function Descriptions

As the API is evolving, no attempt is made here to fully document all the functions in detail. For up-to-date details, please refer to the smartio_api.h header file in the latest software host interface layer distribution.

Initialization Function

The firmware must call this function prior to any other Smart.IO API calls.

```
tStatus SmartIO_Initialize(
      void (*connect_callback)(void),
      void (*disconnect_callback)(void),
      );
```

connect_callback: this is the firmware function to call when a connection is established between Smart.IO and the phone app. The firmware should not make any Smart.IO API calls unless the systems are connected.

*disconnect_callbac*k: the firmware function to call when a connection is dropped, which could be due to the devices being too far apart, or abnormal operations. Invoking this callback function should cause the firmware to reset the state of all UI related functions.

After the initialization call, the firmware must create a new page before creating any UI elements.

Page Management

UI elements are organized into pages. All UI creation commands operate on “current page”, and the firmware can switch pages programmatically.

The end user may navigate to different pages using the native OS page navigation mechanism (e.g. on iOS, by swiping right or left), unless navigation is disabled by the firmware.

Command	**API Name**
Create a new page and set it as the current page	`SmartIO_MakePage`
Set the current page	`SmartIO_SetCurrentPage`
Set the page title	`SmartIO_PageTitle`
Display the specified page as the current page and disable user page navigation	`SmartIO_LockPage`
Unlock page and allow user page navigation	`SmartIO_UnlockPage`

Input UI Elements

Input elements comprise the largest group of UI elements in the Smart.IO API. Most elements have variations (different colors or shapes). (See Appendix for full details with graphics for all the variations.) These elements accept input from the end user. The input values are passed to the firmware via the callback mechanism. The general formats are:

```
// elements that have alignment, variation, and initial value
tHandle SmartIO_MakeXXX(
      uint16_t alignment,
      uint16_t variation,
      uint16_t initial_value,
      void (*callback)());

// elements that contain N entries (e.g. Picker, Expandable List)
tHandle SmartIO_MakeYYY(
      uint16_t nentries,
      void (*callback)());
```

The following input elements are provided:

UI Element	**Description**	**API Name**
On/off button	An on/off switch	`SmartIO_MakeOnOffButton`
3-pos button	A switch with 3 positions	`SmartIO_Make3PosButton`
Incrementer	Increment / decrement control	`SmartIO_MakeIncrementer`
Slider	Slider	`SmartIO_MakeSlider`
Expandable list	A collapsible list to select one item. No more than 6 to 8 items should be on the list.	`SmartIO_MakeExpandableList`
Picker	A scrollable list to select one item. For use when large number of items are needed.	`SmartIO_MakePicker`
Multi-selector	A single or double rows of typically 2 to 6 items.	`SmartIO_MakeMultiSelector`
Number selector	Select a number with a low and high range	`SmartIO_MakeNumberSelector`
Time selector	Select a time in hours and minutes	`SmartIO_MakeTimeSelector`
Calendar	Select a calendar date	`SmartIO_MakeCalendarSelector`

selector		
Weekday Selector	Select a weekday (MON-SUN)	`SmartIO_MakeWeekdaySelector`
OK button	A single button. The label can be modified.	`SmartIO_MakeOK`
Cancel/OK button	Two button choice. The labels can be modified.	`SmartIO_MakeCancelOK`
OK “Link” button	Same as an :OK button” except that the it is linked to another UI element. See text below this table.	`SmartIO_MakeOKLinkTo`
Checkboxes	A group of checkboxes where multiple items can be selected.	`SmartIO_MakeCheckboxes`
Radio buttons	A group of radio buttons where one item can be selected.	`SmartIO_MakeRadioButtons`
Text entry	A box where text can be entered.	`SmartIO_MakeTextEntry`
Password entry	Same as “text entry” except that each character is replaced by * in the display.	`SmartIO_MakePasswordEntry`
Number entry	Same as “text entry” except that only numbers are accepted.	`SmartIO_MakeNumberEntry`

An OK/Cancel button is usually used to elicit a response from the end user. For example, to prompt the end user to decide if they want to read the instructions:

{{ screen cap }}

[Show Instructions? [OK]]

If the end user taps on the OK button, the firmware is notified via the callback function. The firmware then can display a POPUP display (see later description) showing the instructions.

However, in cases like this, the firmware’s participation is not really necessary, and the back and forth communication adds overhead to the UI performance. The “OK Link button” UI element addresses this issue. Using this feature, the firmware first creates a POPUP element with the instruction text. Then the firmware creates the “OKLinkTo” element specifying the handle of the POPUP element as one of its arguments. Once done, the app then handles the end user interaction directly without involving the firmware.

Output UI Elements

These elements allow the firmware to display information or data to the end users. Customized versions of the app may use customized images for gauges.

UI Element	**Description**	**API Name**
Text Box	Display text in a box with specified width (in virtual pixels). Also allow slice icon, slice label, and box alignment.	`SmartIO_MakeTextBox`
Multiline Text	Display text in a box that takes the full width of the screen	`SmartIO_MakeMultilineBox`
Counter	Display numeric digits in a bound box	`SmartIO_MakeCounter`
Progress Bar	Display progress (percentage) in a bar	`SmartIO_MakeProgressBar`
Progress Circle	Display progress (percentage) in a circular “bar”	`SmartIO_MakeProgressCircle`
Horizontal Gauge	Display quantity (percentage) in a horizontal gauge	`SmartIO_MakeHGauge`
Vertical Gauge	Display quantity (percentage) in a vertical gauge	`SmartIO_MakeVGauge`
Semicircular Gauge	Display quantity (percentage) in a semicircular gauge	`SmartIO_MakeSemiCircularGauge`
Circular Gauge	Display quantity (percentage) in a circular gauge	`SmartIO_MakeCircularGuage`
Battery Level	Display a battery icon with the charge level (20% increment)	`SmartIO_MakeBatteryLevel`
RGB Led	Display a “led” with on/off state, and one of the RGB (Red Green Blue) colors.	`SmartIO_MakeRGBLed`
Custom Horizontal Gauge	Display quantity (percentage) in a horizontal gauge with custom colors	`SmartIO_MakeCustomHGauge`
Custom Vertical Gauge	Display quantity (percentage) in a vertical gauge with custom colors	`SmartIO_MakeCustomVGauge`

More advanced output elements such as charts, graphs and tables will be supported in a later release.

Auto Layout and Groups

To accomplish the goal of optimal-looking UI on all target devices, Smart.IO includes these features in addition to GUI slices: spacer slices, the auto-balance command, and the grouping command.

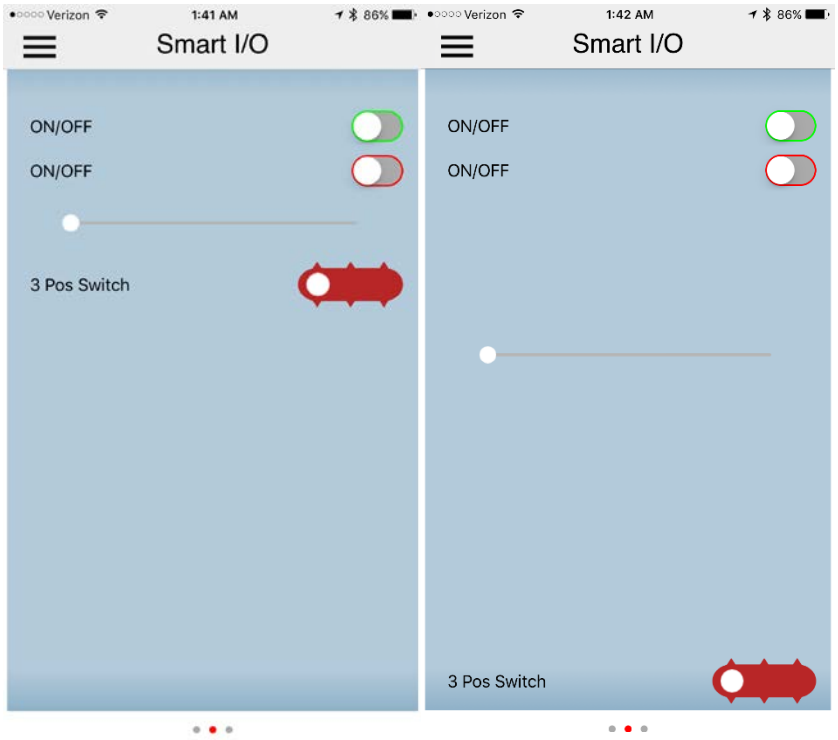
Command	**API Name**
Add a spacer slice	`SmartIO_SpacerSlice`
Auto-Balance the page layout	`SmartIO_AutoBalance`
Group GUI slices together	`SmartIO_GroupObjects`

A spacer slice is a placeholder on the page. The firmware may create spacer slices anywhere on the page just as it would in creating a GU slice. When the function `SmartIO_AutoBalance` is called, the app calculates the vertical empty space not used by the GUI slices (and freeform slices, see later) and divides the amount of free space by the number of spacer slices on the page. It then make each spacer slice take up that amount of vertical space. Thus, if there is one or more spacer slices between two GUI slices, an empty space is created in proportion to the number of spacer slices in-between.

Although not apparent, there is one spacer slice between the second on/off button and the slider button, and two spacer slices between the slider and the 3-pos switch. When created initially, they do not take up any space at all.

When the auto-balance command is executed, the result looks like this:

Figure 31 UI with Spacer Slices: Before and After Auto-Balance

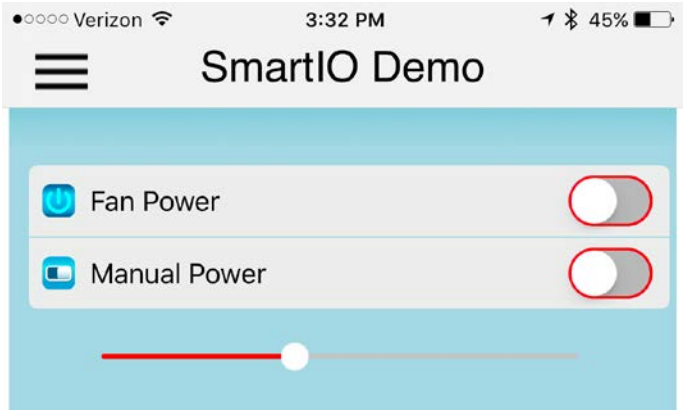


The empty space is evenly distributed to the spacer slices. In this way, auto-balance ensures that the UI page occupies the full height of the device display [12] and allows the firmware to control the amount of empty space between the UI elements.

To further enhance the look and feel, GUI slices can be grouped together using the `SmartIO_GroupObjects` call. Up to eight object handles can be specified at once, and any objects adjacent to each other that are on the list will be grouped together with a round corner group box:

[12] It is of course acceptable to create empty space at the bottom of the page by adding `SmartIO_SpacerSlice(s)` as the last UI elements(s) of the page, before invoking the auto-balance command.

Figure 32 Example of Object Grouping



In the example above, the two on/off switches are grouped together. You can specify multiple groups with a single call, and the app is smart enough that it will only group adjacent slices together.

It is acceptable for a group to have just a single slice. Grouping is for visual purposes only, and the objects within a group are not tied in other ways.

Enable-If Command

Another feature to make an easy-to-use UI is the Enable-If command. The command allows a group of UI elements to be enabled or disabled depending on the value or state of a parent UI element. A UI element that is disabled will be dimmed.

The “enable-state” of the dependent UI elements is handled by the app itself with no action from the firmware needed, thus making a more responsive UI.

Up to eight dependent UI handles can be specified at once. Multiple calls can be made to the same controlling parent if needed. It is an error to have a direct or indirect recursive enable-if relation.

The valid parent UI elements are:

UI Element	**Dependents Are Enabled If...**
On/Off Button	Switch is On
Expandable List	At least one item is selected
Picker	At least one item is selected
Number Selector	A number is selected
Weekday Selector	A weekday is selected
Time Selector	A time is selected
Multi-Selector	At least one item is selected
Checkboxes	At least one item is checked
Radio Buttons	An item is selected
Text Entry	Any text is entered
Number Entry	Any number is entered
Password Entry	Any text is entered

Update Functions

Update functions are used to set the values of either input or output elements. You can usually specify an element’s initial value in the object creation call (various `SmartIO_Make…` functions). These functions allow the firmware to modify them afterward.

Generally for input elements, the firmware should store the current states of the elements whenever they are changed in persistent memory, and then restore them during the next run of the app UI.

Command	**API Name**
Add text to an object. This can be used for adding a slice label, or text to a text box, etc.	`SmartIO_Addtext`
Clear text field	`SmartIO_ClearText`
Update an object with one integer attribute	`SmartIO_UpdateIntValue`
Update an object with two integer attributes	`SmartIO_UpdateIntValue2`
Update an object with three integer attributes	`SmartIO_UpdateIntValue3`
Update an object with a string attribute	`SmartIO_UpdateString`

Synonyms exist to give specific names to update different objects.

Real Name	**Synonyms**

`SmartIO_UpdateIntValue`	`SmartIO_UpdateOnOffButton`
	`SmartIO_Update3PosButton`
	`SmartIO_UpdateIncrementer`
	`SmartIO_UpdateSlider`
	`SmartIO_UpdateExpandableList`
	`SmartIO_UpdatePicker`
	`SmartIO_UpdateMultiSelector`
	`SmartIO_UpdateNumberSelector`
	`SmartIO_UpdateCheckboxes`
	`SmartIO_UpdateRadioButtons`
	`SmartIO_UpdateCounter`
	`SmartIO_UpdateProgressBar`
	`SmartIO_UpdateProgressCircle`
	`SmartIO_UpdateHGauge`
	`SmartIO_UpdateVGauge`
	`SmartIO_UpdateSemiCircularGauge`
	`SmartIO_UpdateCircularGauge`
	`SmartIO_UpdateCustomHGauge`
	`SmartIO_UpdateCustomVGauge`
	`SmartIO_UpdateBatteryLevel`

`SmartIO_UpdateIntValue2`	`SmartIO_UpdateRGBLed`
`SmartIO_UpdateString`	`SmartIO_UpdateTextBox` `SmartIO_UpdateMultilineBox`

Popups

A popup is for displaying a full page UI that takes the focus of the app. It is disabled by default, and when enabled, it has a close [X] gadget on the upper right. When enabled, the end user can dismiss the popup and returns to the regular app function by tapping on the close gadget.

Multiple popups elements can be linked. When linked, the bottom control displays a ← on the left if there is a previous popup and a → on the right if there is a next popup. For example, a long set of instructions can be broken up into multiple multiline text boxes with each one in a popup. The end user can read the popups page by page, navigating back and forth if needed, and close the popup display any time they choose.

The `SmartIO_Popup...` commands mirror the set of `SmartIO_Make...` commands for making GUI slices, and have similar arguments except that the Popup commands do not have alignment parameters. For example, `SmartIO_PopupOnOffButton` creates a popup with an on/off button just like `SmartIO_MakeOnOffButton` creates a GUI slice with an on/off button.

Appending multiple popups to the same source popup (with different calls to `SmartIO_AppendPopup` command) results in undefined behavior.

Command	**API Name**

Create a popup	`SmartIO_PopupOnOffButton`
	`SmartIO_Popup3PosButton`
	`SmartIO_PopupIncrementer`
	`SmartIO_PopupSlider`
	`SmartIO_PopupPicker`
	`SmartIO_PopupMultiSelector`
	`SmartIO_PopupNumberSelector`
	`SmartIO_PopupTimeSelector`
	`SmartIO_PopupCalendarSelector`
	`SmartIO_PopupWeekdaySelector`
	`SmartIO_PopupCheckboxes`
	`SmartIO_PopupRadioButtons`
	`SmartIO_PopupTextEntry`
	`SmartIO_PopupNumberEntry`
	`SmartIO_PopupPasswordEntry`
	`SmartIO_PopupCounter`
	`SmartIO_PopupProgressBar`
	`SmartIO_PopupProgressCircle`
	`SmartIO_PopupHGauge`
	`SmartIO_PopupVGauge`
	`SmartIO_PopupSemiCircularGauge`
	`SmartIO_PopupCircularGauge`
	`SmartIO_PopupBatteryLevel`
	`SmartIO_PopupRGBLed`

	SmartIO_PopupCustomHGauge SmartIO_PopupCustomVGauge SmartIO_PopupLabel SmartIO_PopupTextBox SmartIO_PopupMultilineBox
Append a popup to another	SmartIO_AppendPopup

Freeform Slices

A freeform slice is a “holding area” where the firmware may place one or more UI elements with fine grain placement control. When the firmware creates a freeform slice, it specifies the height of the slice in a number of virtual pixels. The width in virtual pixels is fixed at 320. The firmware must be careful not to make a freeform slice too high. Again, if all the UI slices do not fit into a particular device height-wise, then the app will create a scrollbar.

Once a freeform slice is created, the firmware creates UI elements within the freeform slice by specifying each object’s X and Y location, relative to the upper left corner of the freeform slice [13].

The `SmartIO_FFS_...` commands mirror the set of `SmartIO_Make...` commands for making GUI slices, and have similar arguments except that the freeform commands do not have alignment parameters and take location coordinates. For example, `SmartIO_FFS_OnOffButton` creates an on/off button in a freeform slice just like `SmartIO_MakeOnOffButton` creates a GUI slice with an on/off button.

Command	**API Name**
Create a freeform slice	`SmartIO_MakeFreeformSlice`

Create a popup	`SmartIO_FFS_OnOffButton`
	`SmartIO_FFS_3PosButton`
	`SmartIO_FFS_Incrementer`
	`SmartIO_FFS_Slider`
	`SmartIO_FFS_Picker`
	`SmartIO_FFS_MultiSelector`
	`SmartIO_FFS_NumberSelector`
	`SmartIO_FFS_TimeSelector`
	`SmartIO_FFS_CalendarSelector`
	`SmartIO_FFS_WeekdaySelector`
	`SmartIO_FFS_Checkboxes`
	`SmartIO_FFS_RadioButtons`
	`SmartIO_FFS_TextEntry`
	`SmartIO_FFS_NumberEntry`
	`SmartIO_FFS_PasswordEntry`
	`SmartIO_FFS_Counter`
	`SmartIO_FFS_ProgressBar`
	`SmartIO_FFS_ProgressCircle`
	`SmartIO_FFS_HGauge`
	`SmartIO_FFS_VGauge`
	`SmartIO_FFS_SemiCircularGauge`
	`SmartIO_FFS_CircularGauge`

[13] E.g. the upper left corner of a freeform slice has the coordinate 0,0. Across right (width) is the X coordinate and down (height) is the Y coordinate.

	SmartIO_FFS_BatteryLevel SmartIO_FFS_RGBLed SmartIO_FFS_CustomHGauge SmartIO_FFS_CustomVGauge SmartIO_FFS_Label SmartIO_FFS_TextBox SmartIO_FFS_MultilineBox

As the exact size of a UI element is defined by the app and is not known, the embedded engineers should ensure that the X,Y coordinate chosen for a UI element does not conflict with another UI element. This must be done by running the generated UI and tweaking the API calls as needed.

Popup Alerts

Alerts are for displaying critical information to the end users. They are predefined by the Smart.IO app but do have a few variations for the firmware to choose from.

Command	**API Name**
Display an alert	`SmartIO_PopupAlert`

They are not persistent UI elements, and are generated on-the-fly by the firmware. Once displayed, they disallow further end user interaction except for dismissing the alert.

UI Element States

A UI element has two attributes: enable / disable, and visible (show) vs. invisible (hide).

- Enable implies show
- Show does not imply enable

- Disable does not imply hide
- Hide implies disable

Command	**API Name**
Enable an object	`SmartIO_EnableObject`
Disable an object	`SmartIO_DisableObject`
Show an object	`SmartIO_ShowObject`
Hide an object	`SmartIO_HideObject`

Note: disabling or hiding an object does not remove the space it occupies on the screen.

Miscellaneous UI Functions

Deleting a GUI slice, freeform slice, popup, or a page will delete all UI elements contained within. Deleting the UI element that are part of a GUI slice (which only contains a single UI element) also deletes the GUI slice.

Some UI elements have fill colors, and the firmware can change it using the 32-bit web color value.

Command	**API Name**
Delete an object	`SmartIO_Delete`
Set the fill color	`SmartIO_FillColor`

Fonts

These are the font characteristics. By default, sans serif medium size font is used.

iOS (iPhone)		
Font name	**Type**	**Sizes**
HelveticaNeue	Sans Serif	Small: 12 Medium (normal): 16 Large:20
Helvetica	Serif	Small: 12 Medium (normal): 16 Large:20
iOS (iPad)		
Font name	**Type**	**Sizes**
HelveticaNeue	Sans Serif	Small:15 Medium (normal): 19 Large: 23
Helvetica	Serif	Small:15 Medium (normal): 19 Large: 23
Android		
Roboto	Sans Serif	Small:14 Medium (normal):18 Large:22
Noto Serif	Serif	Small:14 Medium (normal):18 Large:22

The customized app may use special fonts.

Text Control Codes

Individual text strings may contain control codes that change the text attributes. Control codes are prefixed with the % character:

Control Characters	**Effect**

%%	Output a single %
%B	Bold the following characters
%b	Un-bold
%I (capital letter i)	Italicize the following characters
%i	Un-italicize
%S	Use serif font for the following characters
%s	Use sans serif font
%0	Use small font size
%1	Use medium/normal font size
%2	Use large font size
%L	Use superscript
%L	Use subscript
%n	Use normal script
%d	Reset all attributes to default, equivalent to %b%i%s%1%n

These control codes work with respect to the app default of sans serif medium size font. That is, the font attributes specified in the `SmartIO_Initialize` call have no effect.

For example, “Hello %BWorld%b!%IIam Alive%i!!” displays as

Hello **World**!*I Am Alive*!!

Color Values

Color values are 24-bit web color codes, encoding three 8-bit RGB (Red, Green, Blue) values. Since there is no 24-bit data type, a full 32-bit value is used.

This web page (and a web search will show others if this site is unavailable) is a good resource for web color codes: http://htmlcolorcodes.com/

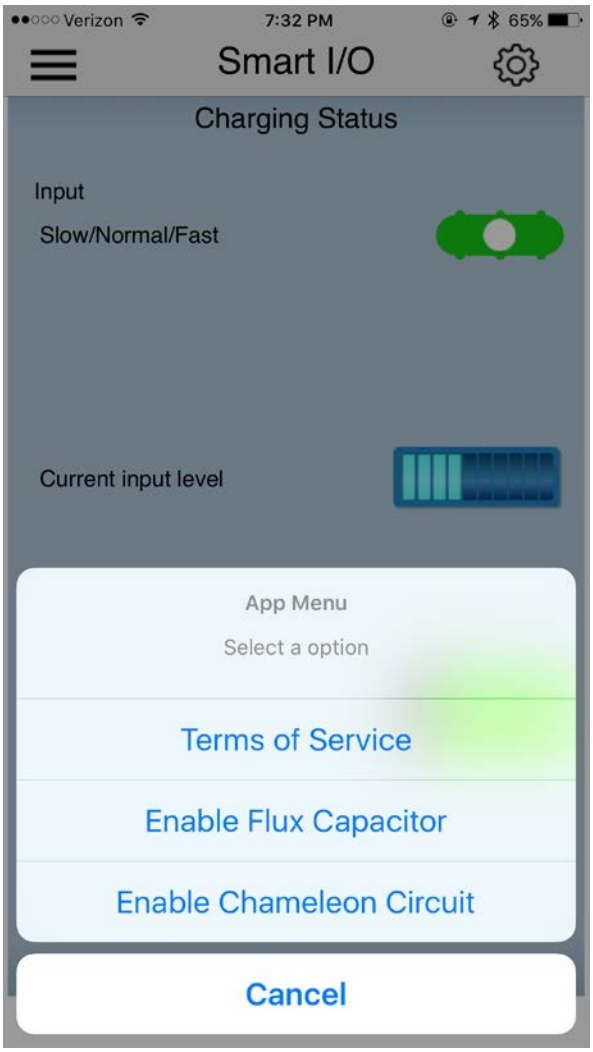
Miscellaneous

App Menu

On the top right of the display is a gear icon representing the System Menu, with commands/options specific to the Smart.IO app itself. The user UI generated by the firmware, may have its own set of menus as well. On iOS devices, this icon is displayed as 3 horizontal bars on the upper left corner.

The firmware creates the app menu by using the SmartIO_AddMenu function. Once created, it can call `SmartIO_AddListItem` as many times as it wishes (to the limit of the screen dimensions) to add menu entries. When the app user touches a menu entry, the callback function will be called.

`tHandle SmartIO_AddMenu(char *label, void (*callback)(uint16_t))`	
`label`	**`callback`**
0/null: no menu entry is created text: create first menu entry with “text”	Callback function. Call with the index of the menu entry that is tapped. 1: first entry was tapped 2: second entry was tapped ...
return: handle value for the menu object	



(The green blob is an artifact of a green UI element behind the "translucent" app menu display.)

App Title

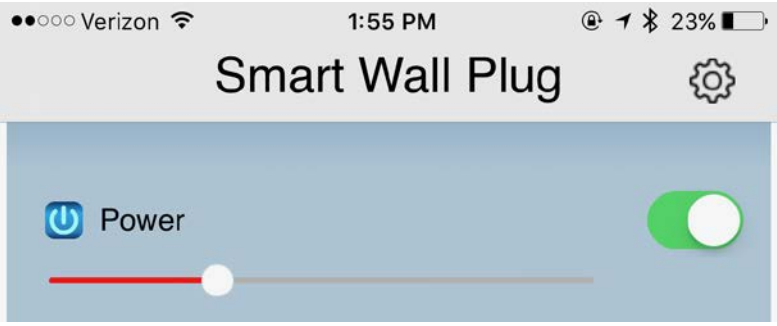
The firmware may change the app title by using the `SmartIO_AppTitle` function. It may call this any time, and the effect is immediate. The default title is “Smart I/O”. An app title is limited to 18 characters, with a shorter title preferred, as it may otherwise not fit cleanly within some device’s display.

In a customized version of the app, custom graphical icons may be used in app title (and other places).

```
tStatus SmartIO_AppTitle(char *title)
```

title
Title of the App

This example shows the firmware changing the app title to “Smart Wall Plug”.



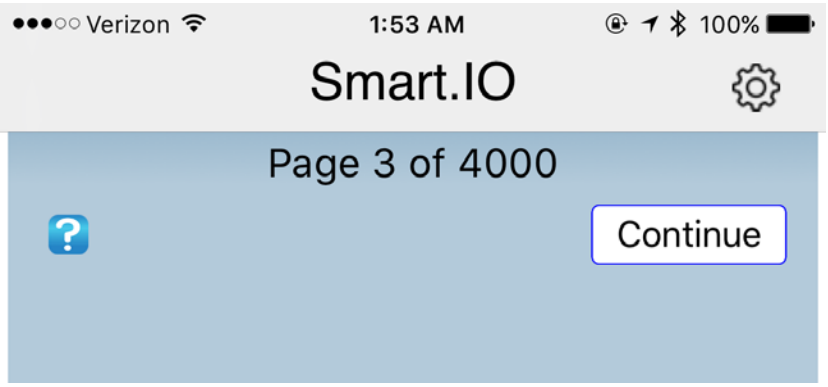
Page Title

Each page may have its own title too. They are displayed in smaller font than the app title. The length is also limited to 18 characters, and should preferably be shorter in most cases to better fit the display.

`tStatus SmartIO_PageTitle(tHandle page_id, char *title)`	
page_id	**title**
Handle of the page object	Title text

This example shows the page title changed to “Page 3 of 4000” and the use of an OK button labeled "continue".

Figure 33 Example of Page Title



EEPROM Commands

If the host MCU does not have persistent storage such as its own EEPROM, the Smart.IO module's 2K bytes EEPROM can be used by the firmware. This is useful for storing and retrieving the values of the UI elements so that the firmware can have an accurate display during different runs of the app, even if the app is run on different machines (obviously not concurrently).

Command	**API Name**
Read bytes from EEPROM	`SmartIO_ReadEEPROM`
Write bytes to EEPROM	`SmartIO_WriteEEPROM`

These functions take the byte address location in the EEPROM and a buffer as arguments to the functions. Reading from EEPROM always deposits the result in the Transfer Memory described in the early part of this document. The firmware must not try to read a block more than HOST_SRAM_POOL_SIZE bytes in a single call, or an error will result.

System Commands

The firmware invokes `SmartIO_StopResume` to "pause" the app, which is useful if the firmware or the embedded hardware needs to perform a long running task and it cannot allow UI interaction. The app displays a spinning circle indicating that it is busy, and no end user interaction is allowed. The app resumes when the firmware calls this function again, or when it makes any UI Smart.IO API call.

Under catastrophic circumstances, the firmware can reset the app to its original state using `SmartIO_Reset`. The BLE connection is kept alive, but the GUI will be reset.

Command	**API Name**
"Pause" the app	`SmartIO_StopResume`
Reset the app	`SmartIO_Reset`

Miscellaneous System Commands

The Smart.IO module has 3 (RGB) physical LEDs on board and can be controlled by the firmware individually. It also has a hardware random number generator, and each Smart.IO

module is guarantee to have a unique ID that is different from any other Smart.IO module. The set of functions below accesses these features.

Finally, for debugging Smart.IO operations, the firmware can enable debug output on the UART port. The UART runs at 9600 BAUD and you can use a FTDI serial to USB cable to send the output to a virtual COM port.

Command	**API Name**
Set the hardware LED	`SmartIO_SetLED`
Clear the hardware LED	`SmartIO_ClearLED`
Toggle the hardware LED	`SmartIO_ToggleLED`
Generate a random number	`SmartIO_GenerateRandomNumber`
Obtain a unique integer ID	`SmartIO_GetUniqueID`
Enable debug output in the UART	`SmartIO_UseDebugUART`

Phone Commands

These functions allow the firmware to access information from the phone.

Command	**API Name**
Obtain clock time	`SmartIO_GetPhoneTime`
Obtain GPS coordinate **(not yet implemented)**	`SmartIO_GetPhoneGPS`

APPENDIX

Appendix A: Using Smart.IO EEPROM for UI State Storage

To save and restore the UI state, you create a C structure and define fields to correspond to the UI controls that you want to store, e.g.

```
typedef struct {
      tHandle on_off_button;
      tHandle slider1_value;
} UI_STATE;

UI_STATE current_state;
```

Then, whenever the values are changed, they are stored into the global variable `current_state`:

```
void Button1(uint16_t val)
    {
    … // previous content
    current_state.on_off_button = val;
    }

void Slider1(uint16_t val)
    {
    … // previous content
    current_state.slider1_value = val;
    }
```

The Smart.IO read and write EEPROM functions take the following function signatures:

```
unsigned char *SmartIO_ReadEEPROM(uint16_t address, uint16_t length);
tStatus SmartIO_WriteEEPROM(uint16_t address, uint16_t length,
    unsigned char *buffer);
```

The read function takes an address, reads "`length`" bytes from the EEPROM starting at `address,` and returns the buffer. The write function takes a similar argument and also a pointer to the buffer (e.g. the address of a UI_STATE structure), and writes the content of the buffer to the EEPROM.

The MCU firmware must manage the address range used. If there is no other use for the EEPROM, you may store the UI_STATE starting at address 0.

The current version of the Smart.IO module contains 2K bytes of internal EEPROM. Future versions may include larger amounts of EEPROM. Note that the last 32 bytes is reserved for Smart.IO use, and is not accessible by the firmware MCU.

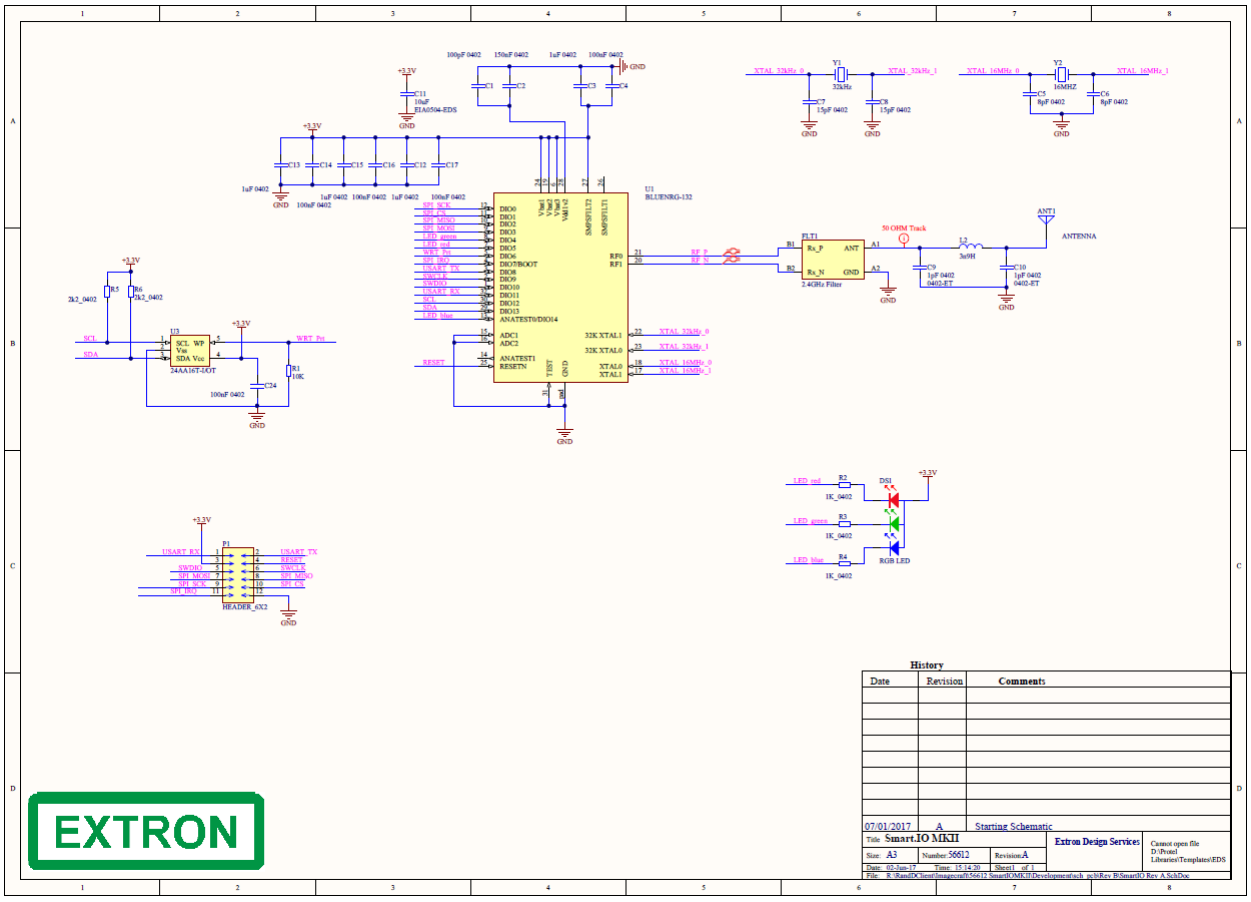
Saving and restoring the UI state then can be written as follows:

```
void SaveUIState(void)
    {
    SmartIO_WriteEEPROM(UI_STATE_ADDRESS, sizeof (current_state),
        (unsigned char *)&current_state);
    }

void RestoreUIState(void)
    {
    memcpy((unsigned char *)&current_state,
        SmartIO_ReadEEPROM(UI_STATE_ADDRESS, sizeof (current_state)),
        sizeof (current_state));

    SmartIO_UpdateOnOffButton(on_off_button_handle,
        current_state.on_off_button);
    SmartIO_UpdateSlider(slider1_handle, current_state.slider1_value);
    }
```

The handles to the on/off button and the slider can be stored in global variables after you create them, or they can be stored in the UI_STATE structure itself. The former is preferred since accessing the EEPROM takes time and you want to minimize the size of the UI_STATE structure as much as possible.

Appendix B: Smart.IO Module Schematic

See here for full size PDF



Appendix C: Smart.IO Arduino Shield Schematic

See here for full size PDF