Smart.IO API

V0.1 July 25th, 2017
richard@imagecraft.com
Richard Man, ImageCraft, https://imagecraft.com/smartio

Once you have integrated the Smart.IO module into your hardware (see the **Hardware Integration Guide**) - which in the case of the Arduino Shield means that you just attach the shield to your Arduino or Arduino work-alike, then it's time to start writing the software! (First you may need to port the Host Software Layer, see the **Software Integration Guide** for details.)

With Smart.IO, your embedded program is linked with the **Smart.IO host interface layer**, a set of C functions which allow you to access SMart.IO features.

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.)

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" [1], 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.

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

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 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.

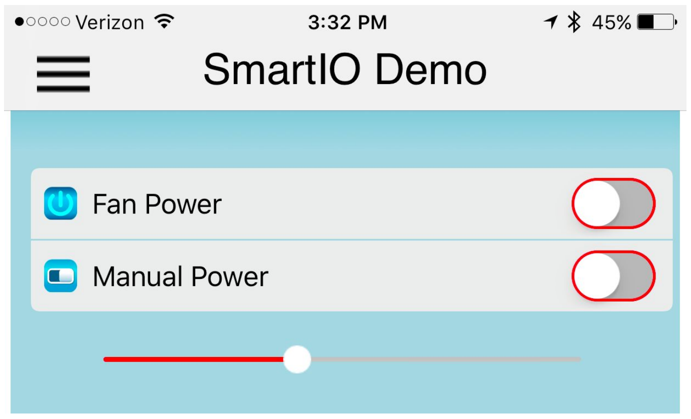
Alternative Callback Model

The callback mechanism is handled entirely in the host interface layer. Since the source code to the host interface layer is provided, you can replace it with a different operating model. For example, the host firmware may use an event-driven style or an RTOS, and the callback mechanism can be rewritten to work with such code. ImageCraft may provide alternative models in source form if demand warrants.

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 [2], contains a UI element, plus optionally an informative icon and a descriptive text. For example, this is a screencap of three 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);
```

[2] 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. It should be apparent that it is easy to build a modern usable UI with Smart.IO with just a few lines of code.

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.

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.

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 object must be copied in its entirety between the processors even though the standard C notation of “char *” is used to denote a string argument type.

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 (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. Finally, this block of memory is also used by the Smart.IO EEPROM read function to hold the read-out values.

API Function Descriptions

The following are short summaries of the API functions. Since 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(
      uint16_t host_sram_pool_size ,
      uint32_t security_id,
      void (*connect_callback)(void),
      void (*disconnect_callback)(void),
      uint16_t run_from_cache,
      uint16_t isSansSerifFont,
      uint16_t fontsize,
      uint32_t font_color);
```

host_sram_pool_size: the size of the SRAM block allocated for storing string data returning from the app. You should use the define constant HOST_SRAM_POOL_SIZE (modified by the firmware port as needed).

security_id: a 32-bit key specific to this user product. This is only used for a customized app. ImageCraft does not have a repository for user key values, and we recommend you use a random number. While collision with another product is possible, it will be highly unlikely if a good random number is used.

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_callback: 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.

run_from_cache: applicable to the paid version of the app only; otherwise, this argument has no effect. UI creation operations are cached by the paid app during the initial run. Subsequent runs require no further SPI transfers from the firmware, and thus can decrease the execution time significantly. This argument disables the cache. This should be set to 0 (do not disable) except for debugging or abnormal circumstances.

`isSansSerifFont`: if non-zero, this sets the default font style used to Sans Serif. The firmware can override the font for any specific object. The default is to use Sans Serif font.

`fontsize`: the acceptable values are 0, 1, and 2, representing small, medium, and large font sizes respectively. The default is 1 or medium size.

`font_color`: specifies the default font color using 32-bit web color values. The default is black, or 0x000000.

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 disallow/allow end user page navigation	`SmartIO_LockCurrentPage`

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 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`

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.

Commands to Add Items

For input elements that have items: expandable list, picker, multi-selector, checkboxes, and radio buttons, the following command is used to add a list item. They are inserted in the order in which they are called.

NOTE: Multi-selector and checkboxes can have only up to 16 list items (note: fewer than 16 is recommended for better visuals) since multiple items can be selected and a 16-bit bitmask is used to indicate which items are selected.

Command	**API Name**
Add a list item to an object, with an optional label	`SmartIO_AddListItem`

The number of `SmartIO_AddListItem` must match the number of entries specified in the original object creation call.

When a list item is selected or deselected, the index of the selected list item, or the bitmask of the selected list item in the cases of multi-selector and checkboxes, is returned to the firmware via callback functions.

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.

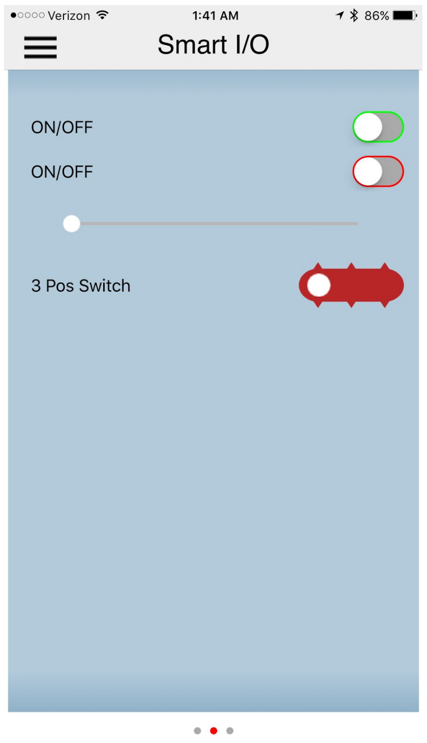
Auto Layout and Groups

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`

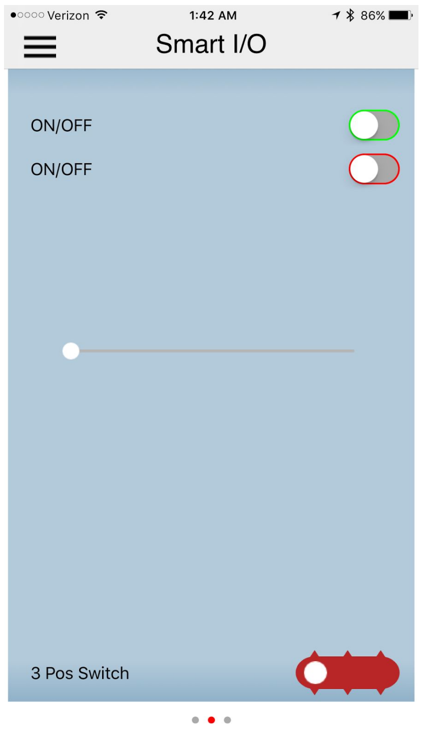
A spacer slice is a placeholder on the page. The firmware may create spacer slices anywhere on the page just as it would in creating a GU slice. When the function `SmartIO_AutoBalance` is called, the app calculates the vertical empty space not used by the GUI slices (and freeform slices, see later) and 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.

For example:



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.

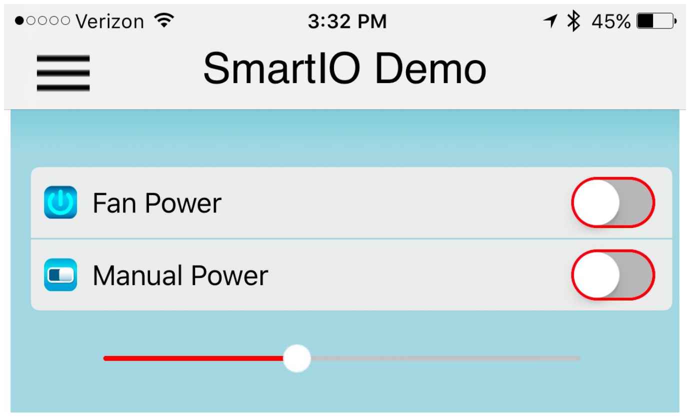
When the auto-balance command is executed, the result looks like this:



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 [3] 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:

[3] 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.



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 [4].

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`

[4] 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_SemiCircularGauge`
	`SmartIO_FFS_CircularGauge`
	`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		
	Sans Serif	Small: Medium (normal): Large:
	Serif	Small: Medium (normal): Large:

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/

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 A: UI Elements Graphics

Most UI elements have different “variations”, and future releases of Smart.IO software will support color themes. Taken together, this allows embedded products to sport their own look and feel.

This appendix shows the graphics for different variations of the UI elements. As this is just a snapshot of a release, graphics from the most recent release of Smart.IO app may look different.

Appendix B: smartio_api.h (function prototypes)