App Note: Smart.IO Program Template

V0.1 Nov 16th, 2017
richard@imagecraft.com
Richard Man, ImageCraft, https://imagecraft.com/smartio

Smart.IO is a new way of creating flexible UI for your embedded products. To use Smart.IO, you build the MCU firmware with the Smart.IO “Host Interface Layer” code, which is supplied by ImageCraft in Standard C source form. These functions implement the Smart.IO API. This appnote describes the basic structure of a program that uses Smart.IO.

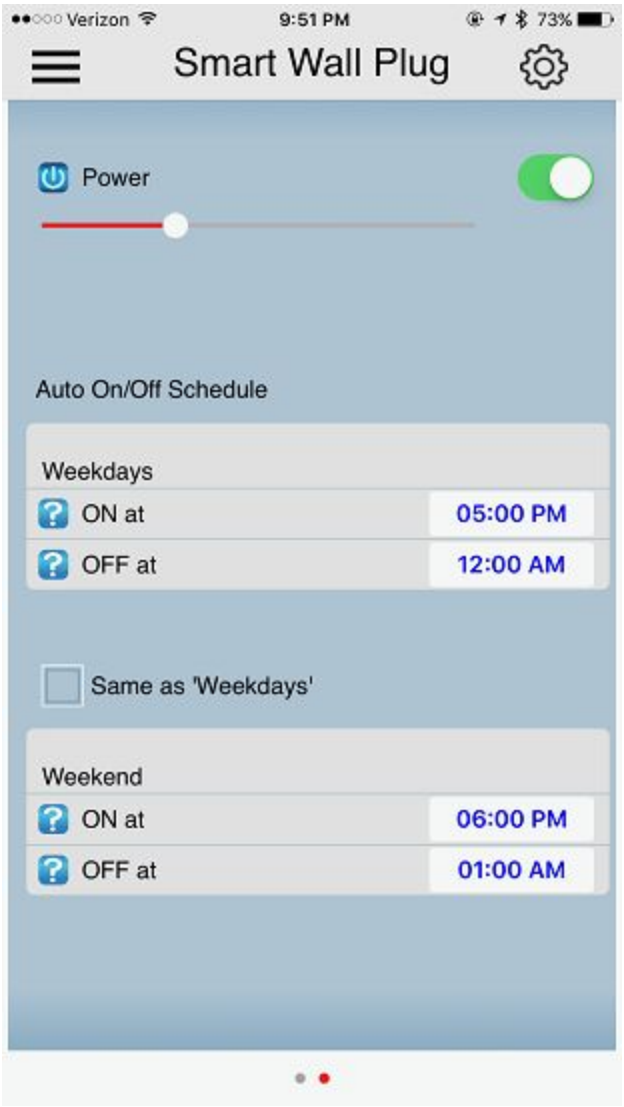
Source code can be found on the web page
https://imagecraft.com/download/smart-io-downloads

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.

APPENDIX: 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.