Smart.IO Host Interface Layer

V0.2 September 14th, 2017
richard@imagecraft.com
Richard Man, ImageCraft, https://imagecraft.com/smartio

To use the Smart.IO toolkit, you include the Smart.IO Host Interface Layer source code with your host MCU firmware build project. The Smart.IO Host Interface Layer source is written in Standard C, and is available from the ImageCraft website. Only a few functions are specific to the particular hardware MCU, so porting to a new MCU or to a different compiler should be easy in most cases.

You can download the latest reference ports from this webpage: https://imagecraft.com/download/smart-io-downloads. The base reference port is for the ST-Nucleo-F411RE[1] based on the STM32F411RE MCU using ImageCraft's JumpStart C for Cortex compiler.

We will update the page to include as many reference ports for other MCUs and compilers as possible. If you have created a port to a host MCU not listed on that page, and are willing to share your efforts, please contact us at info@imagecraft.com

The MCU←→ Smart.IO Interface

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

- 4-pin SPI - MOSI, MISO, SCK (clock), nCS (chip select)
- Host IRQ - interrupt signal from Smart.IO to MCU, normally low
- RESET - resetting the Smart.IO module, normally high
- Vss and GND

6 pins are used for communication and reset, and 2 pins are used for Vss and GND, for a total of 8 pins. Using the ST-Nucleo-F411 as an example, the following connections are used:

Function	**Pin**	**F411 Device Alternate Function Code**

[1] For the purposes of running the Host Interface Layer, this port is also compatible with the F401 and many other ST F4xx devices, as the only difference that matters here is the memory map (e.g. SRAM sizes), so a simple changing of the target device setting would allow the port to work for these mostly-compatible devices.

SPI1 SCK (Clock)	PA5	AF5
SPI1 MOSI	PA7	AF5
SPI1 MISO	PA6	AF5
SPI1 nCS (Chip Select)	PB6	N/A
Host IRQ	PA9	N/A
Smart.IO RESET	PC7	N/A

SPI Properties

The SPI interface should be set to:

- 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

The STM32F411 has multiple SPI units, and in this case, SPI1 is used.

Host IRQ

The host IRQ pin should be:

- Configured as an input pin
- Set input interrupt triggered by transition from low to high

Furthermore:

- Signal is pulled high by the Smart.IO module. The host MCU should not use an internal pull-up or an external resistor
- Signal is in high impedance state

Smart.IO RESET

The Reset pin should be set as:

- Configured as an output pin
- Normal state is level high
- Must be pulled high by either the MCU internal resistor or an external resistor

Source File Directory Structure

The Host Interface Layer sources are available as a ZIP archive file on https://imagecraft.com/download/smart-io-downloads. When unzipped, the F411 reference port has the following directory structure

```
Host_Interface_Layer\
      smartio_api.c
      smartio_interface.c
      smartio_api.h
      smartio_interface.h
      smartio_hardware_interface.h
      < JumpStart C for Cortex project files >
      main.c      ← sample test driver
      < compiler project files >

      STM32F411-ICCV8\        ← ST-Nucleo-F4x1 specific files
      smartio_hardware_interface.c
      handlers.c
      stm32F4_vectors.s
```

When you are creating your own port to other MCUs, you should use a similar directory structure and put the MCU-specific files under a single subdirectory, using a similar naming scheme of <MCU>-<compiler>. (In this case, ICCV8 refers to version 8 of the ImageCraft C Compiler). You would replace `main.c` with your own firmware program, and other MCU specific files (`handlers.c` and `stm32f4x_vectors.s`) with files specific to your MCU and compiler.

The MCU-Dependent Files

Primarily, only a single file `smartio_hardware_interface.c` needs to be ported to a different MCU or compiler. This file define functions to send and receive data from the SPI port connected to the Smart.IO hardware module.

The functions as declared in `smartio_hardware_interface.h` are [2]:

- `void SmartIO_HardwareInit(void (*IRQ_ISR)(void));`
  Set up the Smart.IO hardware interface and register `IRQ_ISR` as the interrupt handler for host IRQ interrupt.

[2] Per usual, information in this document may be outdated as compared to the latest source. The source files should be considered as the most up-to-date document.

The host MCU must initialize the SPI, host IRQ, and the Smart.IO reset pins as described above.

- `void SmartIO_SPI_SendBytes(unsigned char *sendbuf, int sendlen);`
  Send a stream of bytes to the SPI port connected to the Smart.IO hardware. `sendbuf` contains the data to be sent, and `sendlen` is the number of bytes to be sent.

- `Int SmartIO_SPI_ReadBytes(unsigned char *replybuf, int buflen);`
  Read a stream of bytes from the SPI port. `replybuf` is the buffer to receive the data and `buflen` is the length of the buffer. The number of bytes to read is sent as the first two bytes, low byte first.

  As part of the Smart.IO-defined communication protocol, the host MCU must release nCS (i.e. pull it high) as soon as all the bytes are read. However, the transaction is not completed until Smart.IO releases the host IRQ. See next function.

- `void SmartIO_SPI_FinishRead(void);`
  As the Smart.IO module is a SPI slave, data coming from it employs a special ImageCraft defined protocol. One aspect is that the the host IRQ is used by Smart.IO to indicate that the SPI transaction has completed. .

  During a SPI read, the host IRQ is pulled low by the Smart.IO module. This routine waits until the host IRQ is pulled high again, signifying that the read transaction is complete.

- `void SmartIO_Error(int n, ...);`
  When the host interface layer detects an error condition, it calls this function to report it. The error code, n, is an enumeration defined in `smartio_interface.h`.

  In the reference port, this function prints out an error message in the UART port attached to the ST-Nucleo board. In a product design, a blinking LED with coded messages or other mechanism can be used.

Depending on the MCU and the compiler support, these functions can be as short as just a few lines each. For example, with the ImageCraft compiler's JumpStart API, these functions can be implemented quite trivially.

The remaining file, `handlers.c,` is the file containing the default interrupt handlers source, and stm32F4_vectors.s sets up the interrupt vector entries.

Other Customizable Items

In `smartio_api.h,` there are two defines that you may adjust based on your environment:

```
#define HOST_SRAM_POOL_SIZE         256
#define CALLBACK_TABLE_SIZE         100
```

These constants define two structures in SRAM, and therefore you must make sure that they fit within the host MCU SRAM limits. The first structure is exactly `HOST_SRAM_POOL_SIZE` bytes large and the second structure is approximately `8*CALLBACK_TABLE_SIZE` bytes large.

The `HOST_SRAM_POOL_SIZE` is the size of the read-back buffer used by Smart.IO to return data to the MCU firmware, either as a result of an API call, or containing the value of an input UI element that has been changed by the app user (e.g.: the app user changes a slider). Most of the time, return data is small in sizes, just a few bytes. However, there are two instances where larger amounts of data are returned:

1. As a result of a `SmartIO_ReadEEPROM` function to read data stored in the EEPROM. In this case, the amount read is an argument to the API function, and thus is controlled by the MCU firmware.
2. As a result of the app user typing on a text entry input box. While most UIs would not need or want large amounts of text input, nevertheless, the app user may either by choice or accidentally enter large amounts of text, which are then passed back to the MCU firmware.

Any returned data that overflows `HOST_SRAM_POOL_SIZE` will be discarded silently. Moreover, the Smart.IO firmware itself maintains an internal buffer that is guaranteed to be at least (but might not be more than) 2K bytes. Also note that the size of the EEPROM in the Smart.IO module is 2K bytes (minus 32 bytes for internal Smart.IO use). If you do not need the space for the EEPROM function, you should define a smaller value for `HOST_SRAM_POOL_SIZE`; e.g.: 256 (bytes) should suffice for most situations.

`CALLBACK_TABLE_SIZE` is the size of the “CALLBACK” table. When the host firmware calls an API to create an input element, it supplies a callback function so that Smart.IO will invoke that function when the app user changes the value of that input element. The Host Interface Layer uses this callback table to store information related to the callback functions.

`CALLBACK_TABLE_SIZE` is bound by the number of input elements the firmware creates for the UI. The default value of 100 is more than enough for most UIs and can be adjusted downward in most cases.

Portable Files

The following files are also in the Host Interface Layer but should not require any modification by the embedded users:

- `smartio_api.c`
- `smartio_interface.c`
- The header files `smartio_api.h, smartio_interface.h, and smartio_hardware_interface.h`

Demonstration Driver

The reference port also includes a driver file `main.c`. Besides calling `SmartIO_HardwareInit` to initialize the Smart.IO hardware interface, it also sets up the ST-Nucleo UART at 9600 baud so that it can accept commands from the UART port. You can use a terminal program such as (free) PuTTY to talk to the ST-Nucleo.

Once you use the smartphone Smart.IO app to pair with the Smart.IO module using Bluetooth, then you can type on the terminal connected to the ST-Nucleo to control the app display. In particular, if you type "128 <digit>" where <digit> is 0 to 8 [3], a sample UI will be generated on the screen. You can then modify the test code to see the results of API calls. (There are some considerations related to the UI cache that are not covered here; see the Smart.IO Software User Guide for details.)

By looking at the source code in main.c, you may notice that with this interface you can actually test out the effect of an API call by typing a command code and arguments for the command. This feature is for internal use only, and will not be documented here.

The demonstration program is further described in the document "Evaluating Smart.IO Using the Smart.IO Starter Kit" available on the Smart.IO documentation page https://imagecraft.com/documentation/smart-io-documentation.

[3] Again, more examples may have been added. See the source code for details.