Getting started with the ESP8266 and xPL v1.1

What is the ESP8266?

The ESP8266 is a small, inexpensive (~$3) chipset designed to connect embedded processors to a Wifi network. Its small size and low price make it ideal for IoT (Internet of Things) applications. Out of the box, it is designed to communicate with a host processor using a simple serial interface, using modem-like ‘AT’ commands. Many users have used this to interface with Arduinos and similar processors.

An ESP-01 module

What has the hacker community excited though, is the onboard 32 bit processor, which is user accessible. With the hard work of the pioneers from the ESP8266 online community, some tools and an SDK have been made available.
Most ESP8266 modules have 4mb of flash memory onboard, with about 240kB-440kB available for programs, and about 80kB of RAM, which makes this module more powerful than most Arduinos. Also, anywhere between 0 and 9 GPIOs are available, with SPI, I2C available in addition to the standard serial port.

What is xPL?

xPL is an open protocol intended to permit the control and monitoring of home automation devices. The primary design goal of xPL is to provide a rich set of features and functionality, whilst maintaining an elegant, uncomplicated message structure. The protocol includes complete discovery and auto-configuration capabilities which support a fully “plug-n-play” architecture – essential to ensure a good end-user experience.
Many developers have contributed interfaces for many Home Automation devices and systems.

About the xPL for ESP8266 project

This small project will allow you to turn an ESP8266 module into an xPL node. I used an ESP-01 variant (pictured above), since it has two available GPIOs, GPIO0 and GPIO2. Although these GPIOs are used during the boot process, with some care, they can be used by applications. The application currently does two things:

- Responds to xPL X10.BASIC ON and OFF commands, on a user-defined address, to turn on or off GPIO0. I have connected this to a LED for easy confirmation.
- Generates an xPL X10.BASIC trigger command based on the state of GPIO2. Grounding GPIO2 generates an ON trigger, releasing it generates an OFF trigger.

To give users the most flexibility, I built this example using the FreeRTOS-based SDK supplied by the makers of the chip, Espressif. FreeRTOS is an open-source RTOS (Real-Time Operating System) for embedded processors. The code is written in C. For the xPL code, I mainly converted Olivier Lebrun's xPL for Arduino code to straight C.

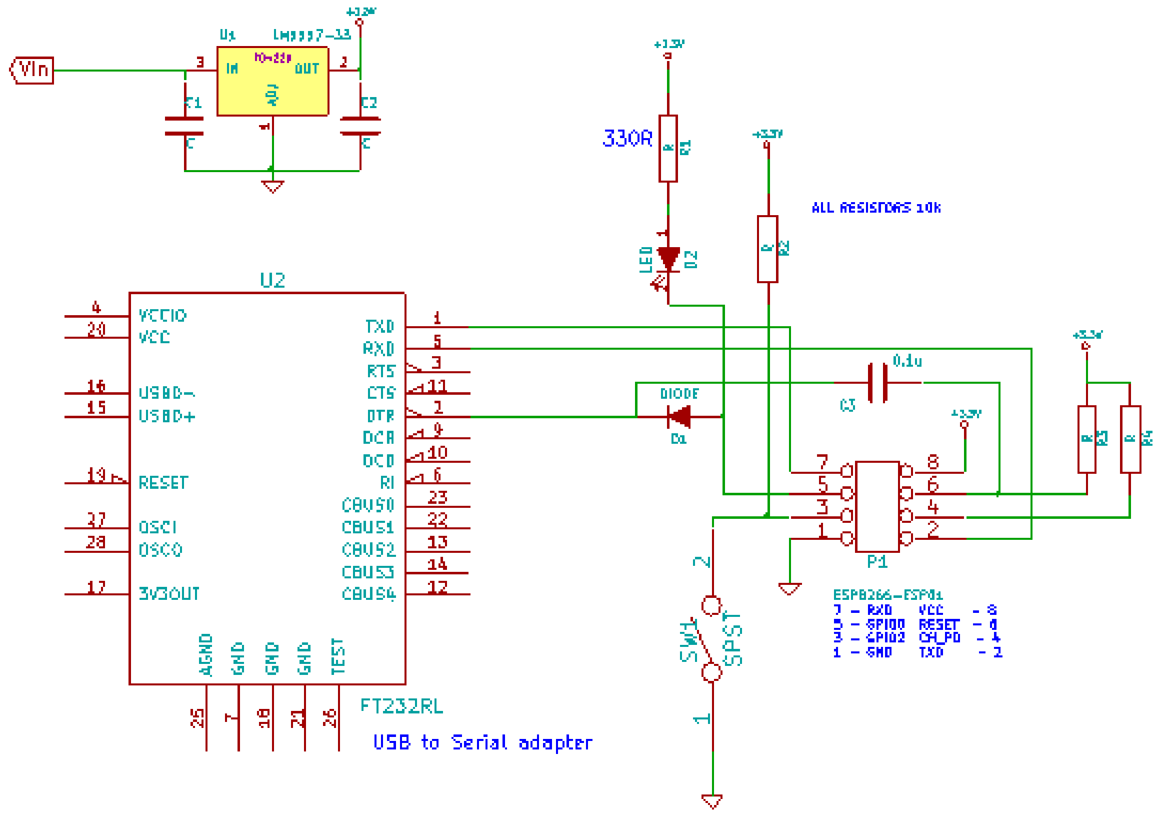
Just getting the tools up and running and figuring out ways to work around the limitations of the SDK was a challenge, but this document will allow you to hit the ground running to develop your own application.

Connecting the board

The ESP8266 runs on 3.3V and is NOT 5V TOLERANT, so you cannot just connect it to an Arduino. As the chip can draw up to 300mA when transmitting, you will need a good power supply. If in doubt, use a large capacitor (>100uf) to regulate the supply, near the chip. Failure to do so will result in instability. Since the firmware is uploaded through the serial interface, you will typically use a USB to serial adapter. Make sure that you get one that works at 3.3V! Here is the one I use:

FT232RL USB to Serial

This one clearly has a 3V/5V switch, so you can use it safely.



ESP-01 Hardware connections

To get the chip into firmware update mode, the chip needs to be reset while GPIO0 is held low and GPIO2 is held high. The above circuit uses the USB-serial adapter's DTR line to pull GPIO0 low and reset the chip via a small capacitor. GPIO0 needs to be held down for about 15ms after reset for the chip to go into firmware update mode. After that, GPIO0 needs to be released, as the bootloader then reconfigures GPIO0 as an output. All of this is handled automatically by the flash tool.

Toolset installation

The following lists all the software components I have gathered in order to create and upload firmware for the ESP8266. Most of these components have been supplied by other hobbyists and are subject to change or move. Even the official SDKs supplied by Espressif seem to evolve rather quickly.
This particular version of the toolset is designed to work under Windows 7 or 8. Linux users have other options available. In particular, Espressif has released a VM file that you can run within VirtualBox, with all the tools pre-installed. It runs within a Lubuntu environment.

Here is a step by step guide:

- Create a directory to put all your ESP stuff in. I put mine in **E:\ESP8266**, but obviously yours will be different.
- Get Python 2.7X. Install it where you want. I put it in **E:\Python27**.
- You need PySerial (http://pyserial.sourceforge.net/).
- Get minGW (http://www.mingw.org/). In the package manager, choose **mingw-developer-toolkit** and **mingw-base**. This will give you **make.exe** and some important utilities, like **mkdir.exe** that the makefiles will need.
- Get the sdk (Espressif RTOS SDK). Install it under your chosen directory. Mine is in **E:\ESP8266\esp_iot_rtos_sdk-master.**
- Download the missing libraries for the SDK: FreeRTOS libraries for SDK. Add them to the **E:\ESP8266\esp_iot_rtos_sdk\lib** directory.
- Get the pre-built compiler. Stick it in your chosen directory. Mine is in **E:\ESP8266\xtensa-lx106-elf**.
- Get the missing xtensa includes for gcc here. It is the file **xtensa.tar.bz2.** Extract them into the compiler subdirectory: **E:\ESP8266\xtensa-lx106-elf\xtensa-lx106-elf\include.**
- Get the source code for the xPL example project. Install it into a directory under your chosen project directory. Mine are in **E:\ESP8266\Apps\xPL-ESP8266.**
- Now you need to correctly setup your path. You can of course add things to your global user path, but I like to setup a separate command prompt for things like this. I created a file called **Vars.cmd**, that sets up the path for working with the ESP toolset:

  -
    ```
    @echo off
    PATH=%PATH%;E:\ESP8266\xtensa-lx106-elf\bin;E:\MinGW\bin;E:\MinGW\msys\1.0\bin
    ```
    - Then I setup a desktop shortcut for a command prompt:**%comspec% /k E:\ESP8266\VARS.cmd**. When I click this I get a command prompt with the proper path already set.
    - For the same reason, I also created a batch file that sets up the path before calling make:

-
  ```
  @echo off
  call E:\ESP8266\vars.cmd
  make %1 %2 %3 %4 %5
  ```

- So in my project setup, in Visual Studio, I use ESPMAKE instead of make.
- Modify the makefile in the xPL_ESP8266 project so that it reflects your locations for the tools and sdk. You should be able to compile and flash by simply typing ‘make’ or ‘make flash’ in the xPL_ESP8266 directory.
- Get the enhanced serial terminal and firmware tool. You need the modified Python tool to flash the chip, **esptool-py.py.** This version is enhanced to reset the chip and bring it into firmware update mode automatically by using DTR. It also connects to Termie, the serial terminal application to share the serial port connection.
- Compile ‘Termie’ the serial terminal. I modified this program to use a named pipe to communicate with the **esptool-py** flasher application. Thus it knows when to disconnect from the serial port when you want to flash the chip.
- You can use Visual Studio as your development IDE, by adding your project as a C++ Makefile project. I have included sample project files. You need to modify the project properties to setup your make command and add the include directories in the configuration so that intellisense can show you function definitions in the editor. I recommend that you get the free Visual Studio 2013 community edition..

Things to watch out for.

You cannot use most of the standard ‘C’ libraries, as they are not compatible with the runtime. The sdk does provide many common functions, like ‘printf’, ‘memcpy’, and similar, but I had to scrounge the Internets to get source code to some missing functions, like ‘sscanf’, ‘strlcpy’.
By default, all code goes into iram, which is only 32k, so it is easy to overflow that segment by linking in a bunch of stuff from the libraries or from your own code.
Use ICACHE_FLASH_ATTR on your functions to make sure they go into irom instead. Irom resides off-chip on a 4M flash ROM, thus leaving about 240-440kB for applications.
Although the final compiled program appears huge (about 180k in irom, and 28k in iram), most of this is library code that runs the Wifi and IP stacks. The actual user application is something like 8k.

Questions?

The ESP8266 community forums are an important resource, along with the wiki. For xPL related queries, the xPL forums will be a good place to start.

Have fun!

Pierre Bénard

xsc.peteben@neverbox.com

Version History

V1.0 Initial release

V1.1 Streamlined tools installation by indicating the location of a 32-bit compiler. Changed default directory to **E:\ESP8266.** Removed dependence on esptool.exe by using the python esptool-py to generate firmware images. Updated link to ‘Termie’. Added link to Python 2.7. Revised library warning. Added ‘What is xPL’ section.