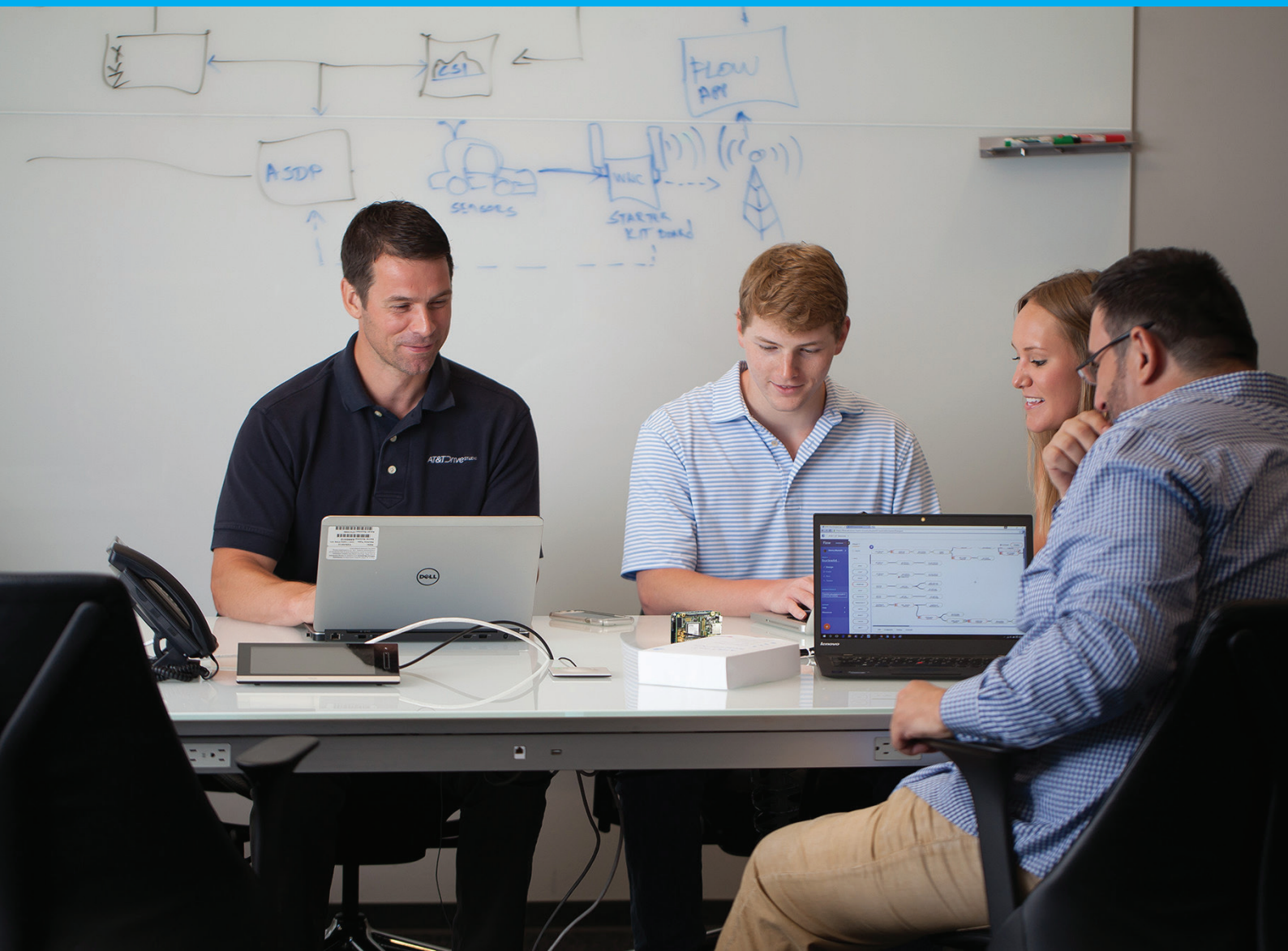
AT&T IoT Starter Kit
Getting Started Guide



AT&T IoT Starter Kit (2nd Generation) User's Guide

If you purchased an AT&T IoT Starter Kit (2nd Generation) and just want to get started with the QuickStart Demo, click here to jump right to that page in Chapter 1.

Revision 0.40 (Monday, December 24, 2018)

Table of Contents

Table of Code Listings

License for Code Examples

Preface

Work in Progress

As of September 2018, this User's Guide is a work in progress. It currently includes the first two chapters. The goal being to add a new chapter, or two, each month throughout 2018.

The next section highlights the first two chapters that are available, as well as the additional chapters planned for development

What this Book Covers

The 2nd generation IoT Starter Kit (SK2) development module runs a version of Linux and can be programmed with either the C language or with Python. It has quite a few peripherals that allow users to connect to various sensors and systems. What makes it an Internet of Things (IoT) module is its ability to connect to the world via an on-board LTE radio. These various capabilities and services are discussed in the following chapters.

Chapter 1 - Getting Started

The first chapter is all about getting started with the SK2. After a brief introduction to the kit itself, you'll begin by registering your kit's SIM card and then working through the QuickStart application that comes programmed into the SK2.

The second part of the QuickStart lets you view the sensor data sent by your SK2 on a pre-built Asset Tracking dashboard that located in the AT&T IoT Marketplace.

Chapter 2 - Using Embedded Linux

The SK2 runs a tiny distribution of Linux. If you've used Linux before then you already know most of what's in this chapter. Along with showing you how to connect your personal computer to the kit, this chapter provide a quick primer for running Linux on the SK2.

Chapter 3 - Programming with C

The SK2 can be programmed using the C programming language. In essence, you can write C Linux application to control the SK2 module. The SK2 is supported by two C callable libraries which let you access the on-board peripherals as well as the LTE communications.

This chapter takes you through installing the C programming environment and writing your first C program. Once complete, you should be able to control the on-board LED and read the User Switch by programming the pins attached to those controls via the GPIO (general purpose bit I/O) API (applications programming interface).

Chapter 4 - Programming with Python

The fourth chapter accomplishes the same goals as Chapter 3, but using Python. Therefore, in this chapter we will learn how to install a new Python-based image to the SK2 as well as use it to read the User Switch and toggle the LED.

Note: As described earlier, the following chapters are currently in development

---

Chapter 5 - Connecting to the Cloud with LTE

One of the unique capabilities of the SK2 is its ability to communicate over the AT&T LTE wireless connection. This allows untethered, mobile access to the Internet. In this chapter you will learn how to connect to the Cloud to send and receive data. This chapter leverages the connectivity libraries to support both programming languages (Python and C).

Chapter 6 - Using I2C with the Accelerometer

I²C (pronounced I-squared-C) is a common serial communications port widely used on microcontrollers and processors. Alternatively, it's also called an I2C port, as this is easier to write in simple text files.

This serial port is widely used for communicating with lower-speed peripherals across short-distances within a board. One of the main differences between other types of serial ports (SPI or UART) is that I2C ports support a simple addressing mechanism which allows them to connect multiple devices together across a single port.

Conveniently, the SK2 contains an on-board sensor (the accelerometer) which is connected to our Linux processor by way of the I2C port. This makes it makes it easy to experiment with the I2C port as we learn about how it works.

Chapter 7 - Using the ADC with the Light Sensor

The ADC (analogue to digital converter) is another useful peripheral interface found on the SK2's Linux processor. This port converts real-world analogue signals (i.e. voltages) into digital values (i.e. numbers) that we can use to observe our environment. In this case, the on-board light sensor is connected to our Linux processor through the ADC peripheral interface.

As the light landing on the sensor gets brighter or dimmer, the light sensor outputs a greater or lesser voltage. Our Linux programs (either using C or Python) can read a numerical representation of the voltage via the ADC port.

Chapter 8 - Getting the Location Using GPS

GPS (global positioning system) is another popular sensor built into the SK2. In fact, this one is notable in that the GPS antenna is one of the three wires that need to be connected when first assembling the kit.

If you have a phone, or a GPS unit in your car, then you already know that GPS systems provide location data by reading signals sent by orbiting satellites. We won't get too deep into how GPS works but we'll examine how your programs can retrieve location data from the GPS sensor to use locally or pass along to the Cloud for further processing or tracking.

Downloads, Support and Tools

Downloads

Visit the following site to download the code for this User Guide:

- https://github.com/att-iotstarterkits

Support

Please visit the the AT&T IoT Starter Kit Knowledge Base for support related to this book and/or your kit.

https://att-iotservices.groovehq.com/help_center

Tools Needed for Code Examples

If you're just planning to read this document, all you'll need is your eyes and an inquisitive mind. But if reading turns to doing, and you want to run the examples or write your own code, you'll need a few tools.

Hardware

- Computer with an available USB 2.0/3.0 port.

  A ***Ubuntu Linux computer*** is required if you want to compile C code to run on the SK2.

  Windows or Mac computer will work fine if you only planning on to connect to the SK2 to view files or execute pre-built examples. These operating systems will also be fine if you're only planning to write Python code.

Software

- Android Debug Bridge (ADB) is required to talk to the SK2 from your computer. Installation instructions are included in Chapter 2.
- Python image – required for Python coding in Chapter 4 (and later chapters): https://s3-us-west-1.amazonaws.com/td-marketplace-assets/SDK-01/M18Q2_v12.09.182151_APSS_OE_v01.07.183121.zip
- Command line (i.e. shell) is needed to execute ADB commands and run scripts. Linux (BASH shell) and Mac (Terminal) command-line tools should work fine. Windows users may want to install an improved command-line tool, such as one of these tools:
  - WSL (Windows Subsystem for Linux) https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux
  - CMDR (http://cmder.net/)

Related Documentation

Product Brief

- AT&T IoT Starter Kit 2nd Generation Product Brief (PDF)
- SK2 AT&T IoT Starter Kit Overview Slides (PDF)

Hardware Documentation

- SK2 Hardware User Guide (PDF)
- SK2 SOM Schematic (PDF)
- AES-M18QX LTE SOM Schematic Symbol (Orcad) (ZIP)
- SK2 LTE SOM Bill of Materials (PDF)

Software Guides

- SK2 Quick Start Card (PDF)
- IoT Starter Kit (2nd Generation) Quick Start Guide (PDF)
- Avnet M18Qx LTE IoT API Guide (DOCX)
- Avnet M18Qx Perpherial IoT Guide (DOCX)

Software / Repositories

- Avnet WNC SDK: This repository contains the software development kit (SDK) libraries and documentation for use with the the IoT Starter Kit2.

  https://github.com/Avnet/AvnetWNCSDK
- Avnet IoT Monitor Example: The monitor source code for the IoT Starter Kit 2 (SK2). This is the source code for the Out-of-Box program that is loaded on the SK2 board when delivered from the factory.

  https://github.com/Avnet/M18QxIotMonitor
- AT&T GitHub Repository

  https://github.com/att-iotstarterkits

AT&T IoT Starter Kit (2nd Generation) Videos

- Device Fundamental Tutorial 1: Install the SDK, ADB plus other Tools (MP4)
- Device Fundamental Tutorial 2: Install and Build the IoT_Monitor Reference Design (MP4)
- Device Fundamental Tutorial 3: Running IoT_Monitor and Other Applications (MP4)
- Device Fundamental Tutorial 4: Exploring the IoT_Monitor Application Source Code (MP4)

Certifications

The SK2 module has gained several certifications. Please refer to Avnet's Cloud Connect Kits page and scroll to the "Certifications" heading (toward the bottom of the page) to review and download the certification documents.

http://cloudconnectkits.org/product/lte-starter-kit-2

Document Locations

Most of the SK2 documents can be found at the following locations. Please check these sites for new and/or updated documentation.

AT&T Marketplace

- https://marketplace.att.com/products/att-iot-starter-kit-2nd-gen
- https://marketplace.att.com/quickstart#starterkit-2nd-gen

Avnet Cloud Connect Kits

- http://cloudconnectkits.org/product/lte-starter-kit-2
- http://cloudconnectkits.org/product/global-lte-starter-kit

1. Getting Started

The first chapter introduces you to IoT (Internet of Things) and the AT&T Iot Starter Kit (2nd generation) - nicknamed the "SK2" - and quickly gets you playing with the demonstration program that comes programmed into the kit.

After a brief review of IoT and the kit's hardware, we will take you through registering the SIM (Subscriber Identity Module) card that comes with the SK2 so that your kit can be recognized by AT&T's LTE cellular network.

Once registered, we'll take you through assembling your kit and kicking off the QuickStart demo, by pushing the board's User Button, where upon the on-board LEDs indicate the various stages of the demo's execution.

Once you have successfully run the QuickStart demo, we'll log into the AT&T Marketplace dashboard to view the data sent by your kit every time you click the button while this demo.

Finally, we'll introduce some of the many support and development resources available to you while working with your SK2.

Topics

1.1. What is IoT?

Who hasn't heard of IoT (Internet of Things) these days? Well, maybe my mother doesn't realize that her thermostat, which can be controlled by her iPhone, is an IoT device. But engineers, programmers and makers are all trying to figure out how to leverage the Internet to make their projects more exciting and useful.

There are just too many applications where IoT might useful to cover them all, but here's three examples where you might see it in your daily life:

1. Tracking buses, trains and other transportation - while it's handy to inform riders when to expect the next bus, it also provides useful data for managing operations and expenses.

Figure 1.1

2. Parking - becoming more popular at airports and cities, tracking empty parking spaces allows cloud and mobile apps to direct citizens and clients to available spaces. Once again, though, the aggregate data from these operations also help communities and business better plan for, and utilize, their infrastructure.

Figure 1.2

3. Asset Tracking - is likely the most widely used application for IoT today. Keeping track of trucks, containers, pallets - or just about anything - is an essential requirement for our just-in-time world.

Figure 1.3

1.1.1. How Does IoT Work?

In each of above use-cases, the device (ie “thing”) is capturing data (location, temperature, etc.) and sending it to the cloud (ie “Internet”).

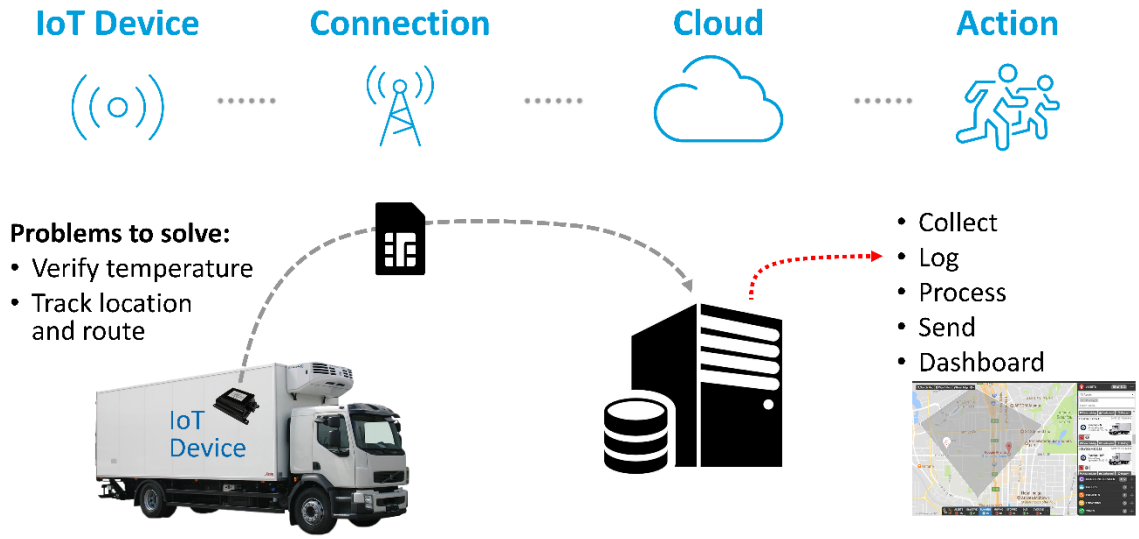


Figure 1.4 – How Does IoT Work?

Breaking it down, we can describe IoT in four parts:

1. **IoT Device** - the device captures sensor data and sends it to the Internet. For decades we have built devices that can read sensors, but only in the past few years have we begun to add the hardware which allows them to talk to networks and the cloud.
2. **Connection** - there are several methods for sending data to the Cloud. For example, until recently, WiFi has been one of the most popular methods for many home or business applications. Although, a growing number of applications require connectivity with greater range and less difficult configuration headaches.
3. **Cloud** - that is, the “Internet”, receives and processes the data sent from the device.
4. **Action** - to be fair, the “Action” is generally handled by the Cloud, but we like to spike it out because this represents the reason for using IoT in the first place. Why capture and send data to the Cloud if you don’t need to trigger alarms, notifications, or analyze the data in the first place?

1.1.2. Connectivity Problem

As stated in the previous section, WiFi has been a popular ingredient for early IoT projects, but it runs into several roadblocks for a growing list of applications. In our earlier examples, WiFi might only be able to send data from truck or container residing in a terminal, but not while in transit.

Similarly, trying to provide WiFi across an entire city, or even just a parking garage, is difficult and expensive.

Finally, WiFi configuration is tedious and difficult. Many early IoT adopters have struggled with the cost of setting up and maintaining WiFi configuration. It's not hard to imagine the value of outfitting every vending machine with an IoT device. Just think of how useful that data might be in managing a vending operation. Then consider how much time you would need to pay your technicians to configure the WiFi settings for each vending machine. Even worse, how about paying them to do this again each time the building or business hosting the machines updates their WiFi passwords?

1.1.3. Why LTE and this Kit?

Cellular LTE communications have revolutionized personal communications. We see this in the phones we carry with us everyday. The nearly ubiquitous coverage and ease of connection make LTE networks ideal.

Even better, their simple, one-time configuration makes LTE easy for users and customers. Our LTE-based devices contain a SIM (subscriber identity module) card which uniquly identifies each device on the cellular network. Configured only once - often right at the factory itself - the cellular device is now enabled throughout the entire LTE network.

Why have IoT devices not used LTE in the past? Because of design, development, certification and monthly costs, application developers have largely ignored the advantages their numerous advantages. But these difficulties are being addressed so that a new wave of applications can be enabled by LTE networks.

Figure 1.5

This is where the AT&T IoT Starter Kits fit into the picture. They give developers access to low-cost, fully certified, small footprint modules that can be used to build LTE connected IoT devices. Use them to host your embedded applications, or to just provide the LTE connectivity to an embedded system you have already developed. Either way, these kits make LTE a reality for IoT applications.

1.2. IoT Starter Kit (SK2)

Figure 1.6

AT&T IoT Starter Kit (2nd Generation) - also known as "SK2" - provides an innovative new System-on-Module IoT solution, enabling the design of cellular connected edge devices, certified for operation in the United States. (An alternative version of the kit "Global LTE IoT Starter Kit" is also available from Avnet to support markets outside of the United States.)

Designed to be used for both prototyping and production, the slim form-factor LTE SK2 board is fully compliant with FCC, PTCRB, and AT&T network certifications, thereby reducing development risk and speeding IoT deployments.

1.2.1. What Comes In the Box

Figure 1.7

1. LTE IoT SOM - LTE System Board (p/n: AES-ATT-M18Q2FG-M1-G).
2. LTE Primary + GPS Antennas - Pulse FPC LTE combo antenna.
3. LTE Secondary Antenna - Pulse FPC LTE antenna.
4. AT&T IoT Starter SIM Card - 3FF Micro-SIM card.
5. AC/DC Power Supply - AC/DC power supply (5V @ 2.5A) plus country/region outlet adapter.
6. USB Cable - for programming and debug.

1.2.2. Hardware Overview

The Starter Kit features a small (79.5 mm x 30 mm) development board built around Wistron NeWeb Corporation (WNC) M18Q2FG-1 LTE Cat-4 modem module. The M18Q2FG-1 module provides cellular modem functionality plus user application code support via a dedicated Arm® Cortex™-A7 processor, thus eliminating the need for an external host processor.

1.2.2.1. Top

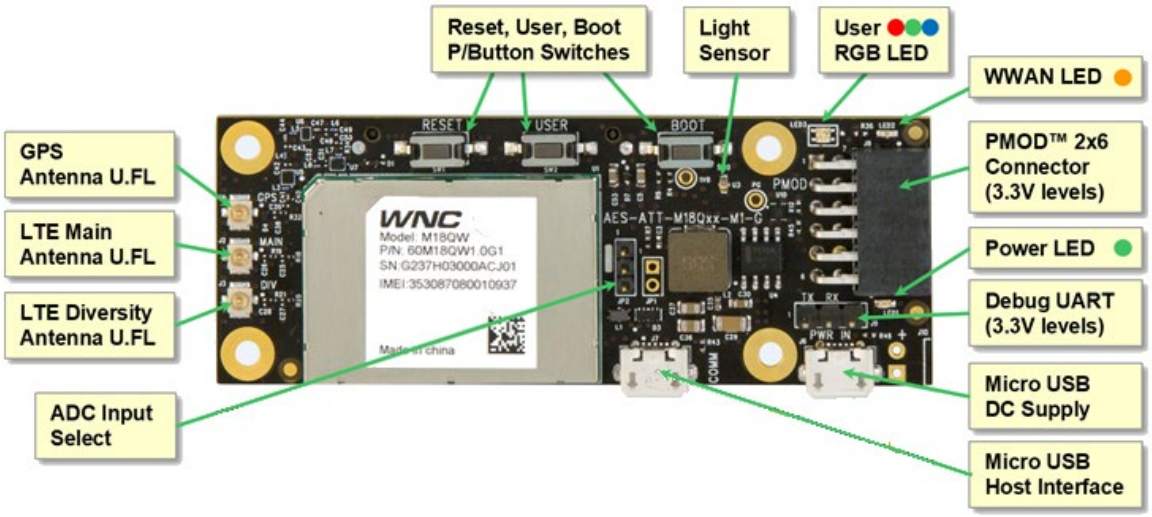


Figure 1.8

Along with the WNC modem/processor module, the top-side of the module contains many components. Starting at the left side of Figure 1.8:

- The ADC (analog to digital convertor) jumper lets you select how the ADC pins from the WNC module are connected within the SK2 board: either to the light sensor or the 60-pin jumper.
- Three antenna connections are found on the left side of the module. This support both the LTE radio (which requires two antennas) as well as the GPS sensor.
- There are three push-button switches on the board - the middle one can be accessed by user programs while the other two are for Rest and Boot. Programming the User Switch via GPIO (general purpose input/output) will be discussed in Chapters 3 and 4.
- The User RGB (red, green, blue) LED can also be controlled via user software (also discussed in Chapters 3 and 4).
- The WWAN (wireless wide-area network) LED is controlled by the LTE modem and provides status regarding the cellular connection.
- The PMOD connector allows you to attach PMOD peripheral boards, making it easy attach sensors or other resources to your kit. There are a wide number of PMOD capable modules that can be purchased separately.
- The Power LED lets you know if power is available (from the Micro USB DC power supply connector).
- The “Debug UART” dedicated for system debug output (ie Linux kernel debug log output).

- There are two Micro USB connectors on the SK2 module:
  - The “DC Supply” USB connector provides DC power to the module. You should use the supply provided in the kit to assure the proper amount of power is available to run the SK2 module.
  - The “Host Interface” USB connector can be connected to your computer, allowing you to modify and control the module via the ADB (Android Debug Bus) protocol. (This will be discussed in the next chapter.)

1.2.2.2. Bottom

The bottom of the SK2 contains a few more items of interest.

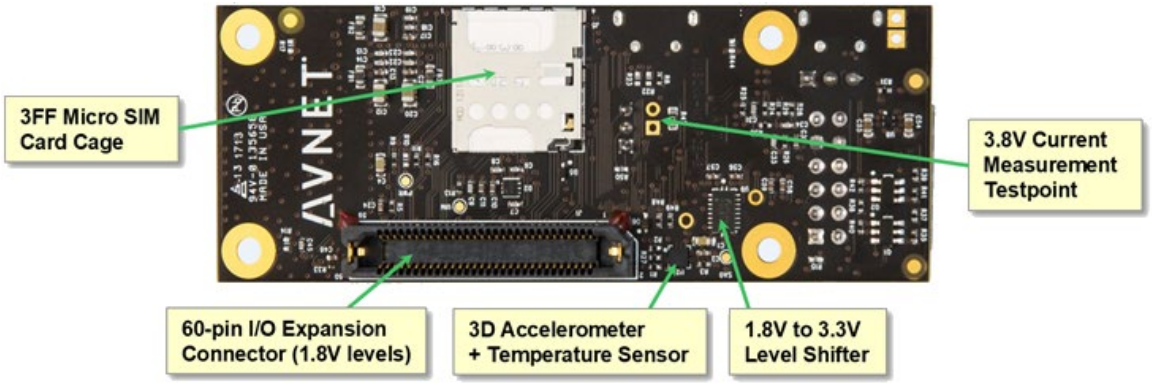


Figure 1.9

Starting from the left and working around the board:

- 3FF Micro SIM Card Cage - is where you will insert the micro SIM (Subscriber Identification Module) card that comes with the kit, once you have registered it with AT&T. (See the SIM registration steps later in this chapter.)
- 3.8V Current Measurement Testpoint - can be used to test the output of the on-board voltage converter. After applying an unregulated 5V supply thru, say, the micro-USB power connector, the on-board converter adapts and regulates the power supply to the board. With a small hardware adaptation, this unpopulated header can be used to measure the current from by this regulated supply.
- 1.8V to 3.3V Level Shifter - handles the voltage differences between the modem processor module pins and the sensor and expansion devices.
- 3D Accelerometer & Temperature Sensor - provides movement and temperature data to the system. (These will be discussed later during the I2c chapter.)
- 60-pin I/O Expansion connector - provides access to many of the processor module’s pins and ports. Use this to add your own hardware to the SK2 system. Alternatively, you can attach the Avnet’s LTE IoT Breakout Carrier as shown below. The breakout makes it easier to access the expansion pins - or allows easy connection of Click modules (as shown below).

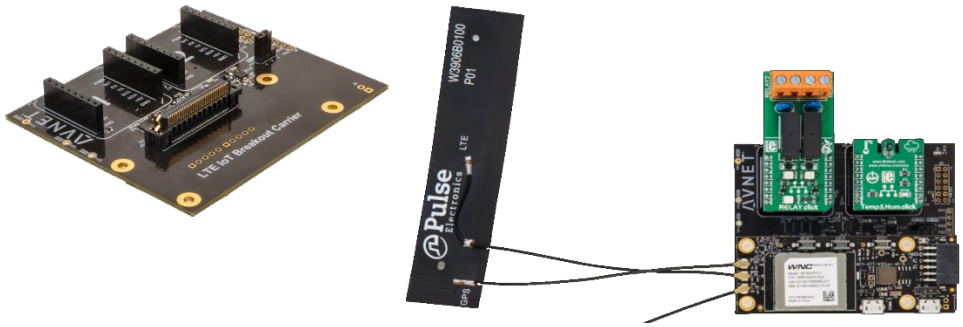


Figure 1.10

1.2.2.3. Block Diagram

The block diagram provides an alternative view of the SK2. From this view, it's easy to see how the kit's components are interconnected. In fact, this view of the system highlights what signals are routed to the PMOD and Expansion connectors.

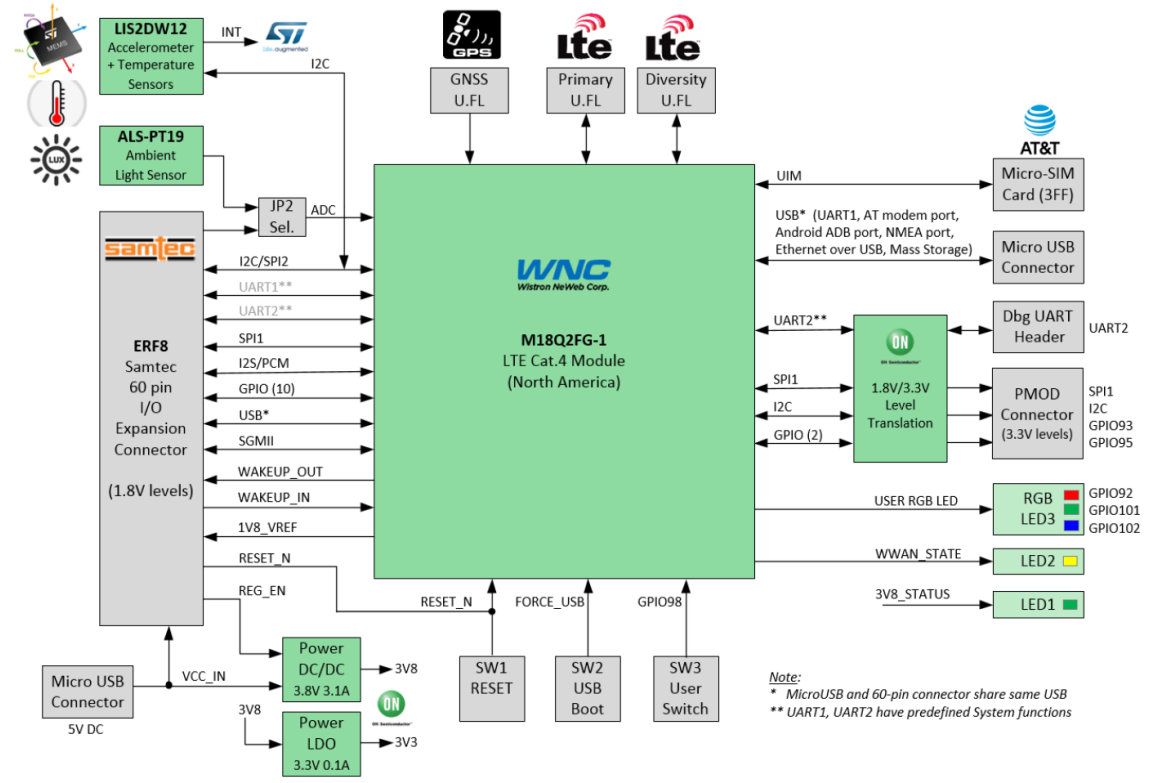


Figure 1.11

Please refer to the SK2 Hardware User's Guide for further details and explanation regarding this board.

1.2.3. Software Overview

From the diagram in the previous section, we can see the WNC M18Q2FG-1 module is the central component of the SK2. It is the 'brains' behind this kit. As mentioned earlier, this module contains an ARM Cortex-A7 processor which runs an OpenEmbedded version of Linux. User application code (scripts, C language, and Python) can run within this Linux environment.

A Software Development Kit (SDK), specific to the M18Q2FG-1 module, provides the necessary API calls allowing your code to access hardware peripherals and system resources. Application code built with the SDK is loaded into the M18Q2FG-1 module through the host USB interface (see the Top of board description), eliminating the need for proprietary debug/emulation hardware.

The SDK consists of two main APIs (Application Programming Interfaces):

- **LTE IoT**: provides access to the communications services over the LTE cellular radio.
- **Peripheral**: provides access to the peripheral ports and pins on the M18Q2FG-1 processor module. In other words, using this API you can access the ports, as well as the sensors that are connected to these ports (e.g. temperature, accelerometer).

You can control these API from one of three different languages:

1. **Linux shell scripts** - useful for simple demos and examples.
2. **Python** - the AT&T Python environment lets you access the hardware using the popular scripting language. The Python environment will be covered in greater detail in a future chapter.
3. **C** - the ever-popular C language is supported; letting you build C applications that run within the Linux environment. In fact, the IoT Monitor demo program - which comes pre-loaded on the SK2 - was written using the C language and accessing the WNC API. A future chapter will describe how to setup and build user applications using the C language.

1.2.4. Cloud Connectivity

AT&T services facilitate Cloud based application development and deployment:

- M2X - a cloud-based, fully managed IoT device management and time-series data storage service for network connected devices.
- Flow Designer - a visual IoT application development and data orchestration environment, with run-time support for complex nonstandard protocol translation, data processing and integrations, to help developers create IoT applications fast.

The SK2's QuickStart demo (i.e. IoT Monitor program) utilizes these services and provides an example for how to get started with them.

1.2.5. Description of QuickStart Demo

When the IoT Starter Kit (2nd Generation) is initially powered-on, a basic user interface will allow you to perform complete Transmit/Receive operations. This initial program will post readings from the sensors located on the IoT Starter Kit module to a pre-configured AT&T Dashboard each time you push the "User" button. You can see this operation progress by following the LED sequence on the board. After registering your kit with the AT&T Dashboard, you will be able to view the data online.

Sensor data from the module includes:

- Motion sensor data provides 3-axis accelerometer data indicating board position
- Temperature sensor
- Ambient light sensor

Note that GPS location data is not enabled in the startup application.

The demonstration runs the "IoT Monitor" application. The code for this can be found on Avnet's GitHub site:

https://github.com/Avnet/M18QxIotMonitor

As this software is pre-loaded onto the module at production, there is no need for you to download or modify it before running the QuickStart out-of-box demo. You can find further explanation, and directions, for this demo in the next section. (Instructions for running the demo can be found in the next section.)

1.2.5.1. IoT Monitor program

To clarify, the initial program that runs automatically upon powering up the SK2 may be called by either of these names:

- QuickStart demo
- IoT Monitor program – The C source-code program used to implement the QuickStart demo

We just point this out so that you won't get confused thinking that there might be more than one startup demo in the SK2. The IoT Monitor source code will be examined later on in this user guide.

1.3. Running the QuickStart Demo

1.3.1.1. Register the AT&T SIM Card

Before we get started with the QuickStart demo, we need to register your kit and its SIM card with the AT&T Marketplace. This provisioning will allow AT&T's network to recognize your kit.

1. **Log onto AT&T's IoT Marketplace:**

   https://marketplace.att.com

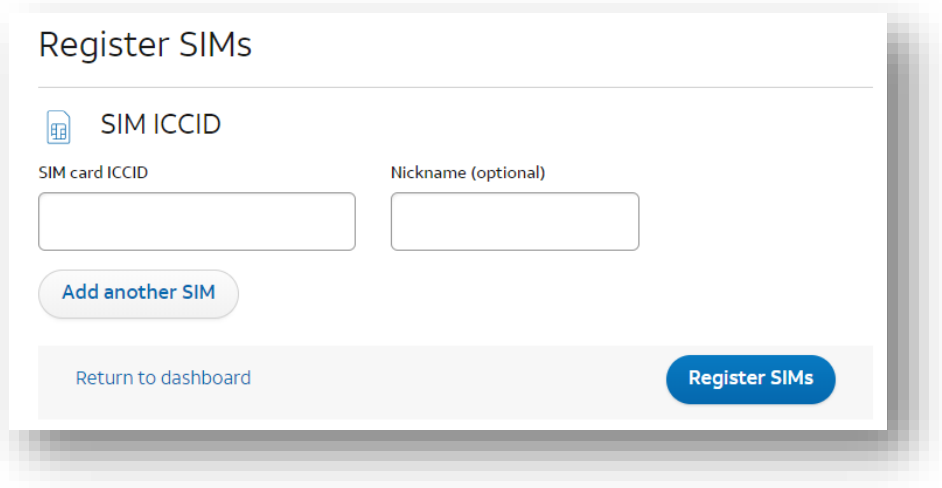
   You have choice of login methods - either using an account that you have previously setup, an AT&T Developer account, or a GitHub account. If you would like to create a new account, click the "Create an account" link near the bottom of the screen.

2. **After you are logged in, navigate to the Register SIMs screen by clicking:**

   `Management > Register SIMs`

3. **Enter the SIM ICCID number located on the SIM card carrier (and the SIM card itself). Then click the "Register SIMs" button.**

   We recommend giving it a nickname, just in case you end up with more than one kit or SIM card and want to tell them apart.



4. **View data about your SIM in the "Dashboard".**

   If you are not taken to the Dashboard, click the "Dashboard" button and you should see your SIM card listed. Click the + button next to your SIM card to view more data about your SIM card.

1.3.1.2. Assembling the SK2

To assemble IoT Starter Kit (2nd Generation) components, there are just a few connection steps required to connect the three main items needed for basic operation.

- Antenna
- SIM card
- Power adapter cable

Directions to assemble the kit include:

1. **Connect the antennas.**

   The flexible antenna arrays should be gently connected to the module antenna terminals as shown by matching each of the labeled antenna cables to the corresponding connector:

Figure 1.13

2. **Install the SIM into the SK2 holder.**

   Carefully pop the AT&T micro-SIM card out from its carrier and install it into SIM card holder:

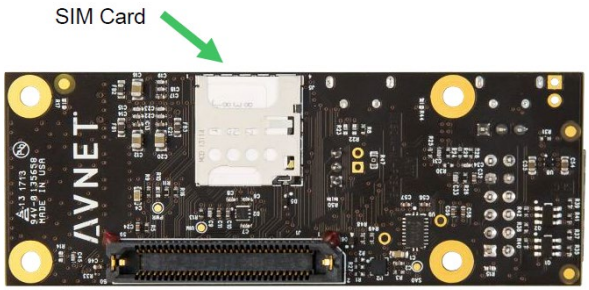


Figure 1.15

Note: Note that you must register the SIM card before it will be recognized by the AT&T network. (These instructions were provided in Section 1.3.1.1.)

3. **Connect the micro-USB power cable.**

   The included power adapter cable should be connected to the micro-USB connector labeled “PWR IN” and the wall adapter plugged into an AC power outlet. This will power on the module:

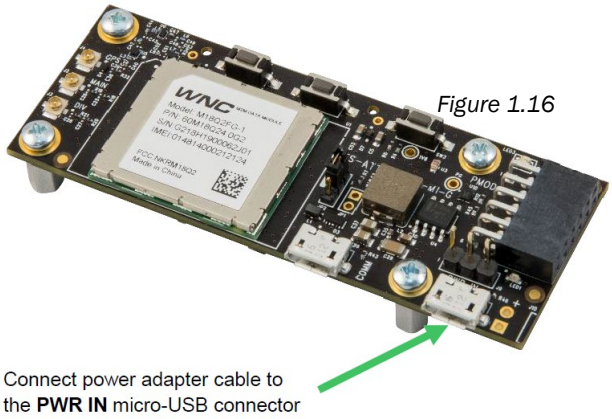


Figure 1.16

1.3.1.3. Running the QuickStart Demo

As stated earlier, the QuickStart demo comes pre-programmed into your kit and should run automatically after power-up.

Note: The next chapter (Chapter 2 – Embedded Linux) will show how to stop this program from running automatically – or how to set your own program or script to run at power-on.

---

4. **Power-on the board. If already on, power-cycle it by disconnecting and then reconnecting power.**

   When power is provided to the IoT module it automatically begins basic operations. The presence of power is indicated by the *Power-On LED* (LED1) illumination:

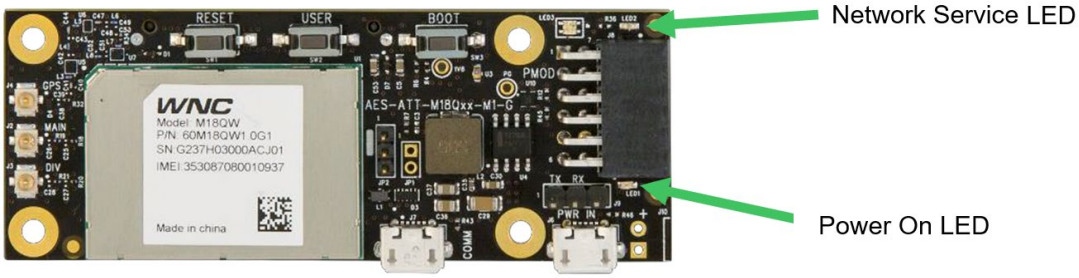


Figure 1.17

5. **Watch for the module to connect to LTE cellular network service.**

   After the module is powered on, the *Network Service LED* (LED2) will begin flashing. This indicates that the module is attempting to register with the AT&T network and establish a connection.

   Once connected to the AT&T network, the Network Service LED will quit flashing and become steady.

Note: If the Network Service LED does not become steady (i.e. it keeps flashing), then it's possible that the APN has been incorrectly configured for your kit. Please refer to the Appendix for more information about the APN setting and how to configure it.

---

6. **When the Tri-color LED turns Green, push the user button once.**

   After network service is obtained, the module's demo will connect to the AT&T M2X service. Once established, the tri-color LED becomes GREEN.

   After network service is established and the module is ready to receive user input, you can press the *USER Push Button* (SW2) to initiate collecting sensor data from the SK2 and sending it to the M2X service.

   Each step in the sequence of events kicked off by pressing the USER button can been seen in the *Tri-color LED* (LED3).

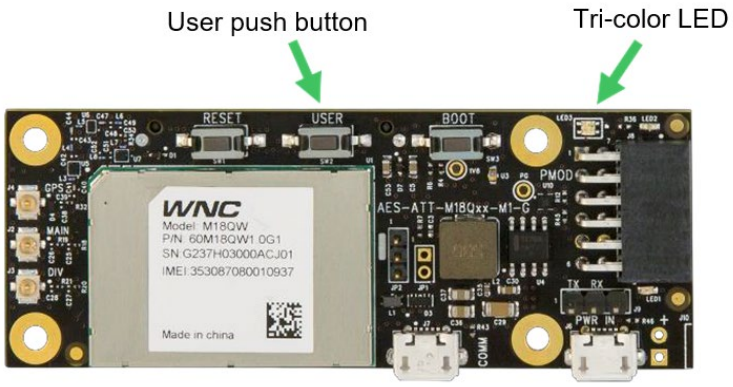


Figure 1.18

The QuickStart program's sequence of events are described in Section 1.3.1.4, on the next page.

1.3.1.4. Quick Start Demo Sequence of Events

While the tri-color LED is GREEN, the user may press the USER button to initiate an M2X data transmission. While the push button is being pressed, the tri-color LED will illuminate white, while the tri-color LED is WHITE no transmission takes place. The WHITE LED only indicates that the module detects that the USER button is being depressed.

Once the USER push button is released, the tri-color LED will illuminate BLUE and the sensor data will be collected and sent to the M2X Dashboard associated with your IoT Marketplace account. It will stay BLUE until all the data has been sent and acknowledged by M2X.

After the data transmission is completed, the tri-color LED will go back to GREEN. This process can be repeated, and the Quick Start demonstration application will continue to follow this execution logic as indicated in the following diagram:

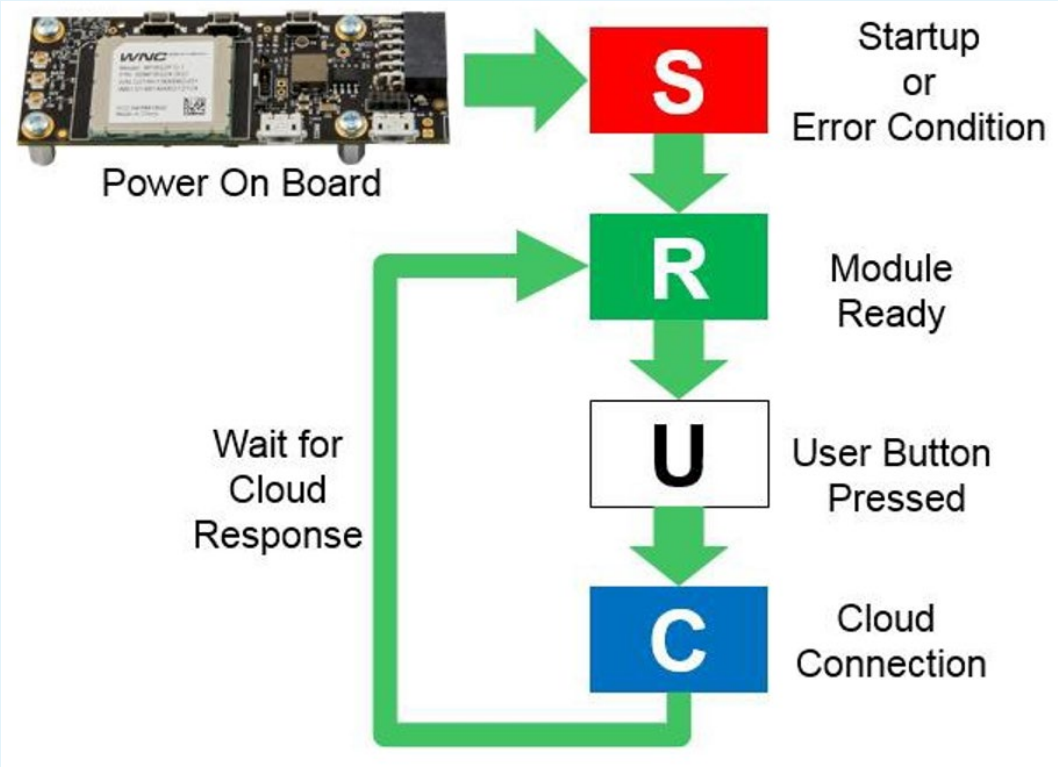


Figure 1.19

If the USER push button is depressed for greater than 3 seconds, it signals that the demonstration program should be terminated, and no further operations performed. At this point to restart the Quick Start demonstration application, you will need to depress the RESET button for longer than 3 seconds (or power-cycle the board) to reset the module.

1.3.1.5. AT&T Marketplace Dashboard

After running the SK2 out-of-box QuickStart demo your data is available in the AT&T Marketplace dashboard. Your kit's serial number needs to be added to your Marketplace account, though, so that your data can be displayed.

7. **Log onto AT&T's IoT Marketplace (unless you are still logged in):**

   ```
   https://marketplace.att.com
   ```

   You have choice of login methods - either using an account that you have previously setup, an AT&T Developer account, or a GitHub account. If you would like to create a new account, click the "Create an account" link near the bottom of the screen.

8. **After you are logged in, navigate to the Marketplace Dashboard registration screen:**

   ```
   marketplace.att.com/amoc/devices/gsk/register
   ```

Note: If the registration form does not look like that shown in Step #3, make sure you are logged into the Marketplace and try clicking on the registration link again.

---

9. **Enter the kit's Serial Number located on your SK2's box.**

   You can find your kit's serial number on the box as shown:

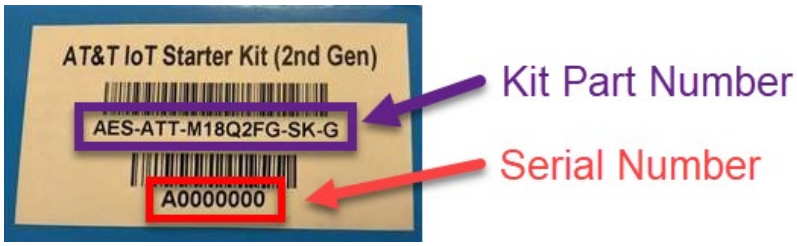


Figure 1.20

Then enter it into the registration dialog:

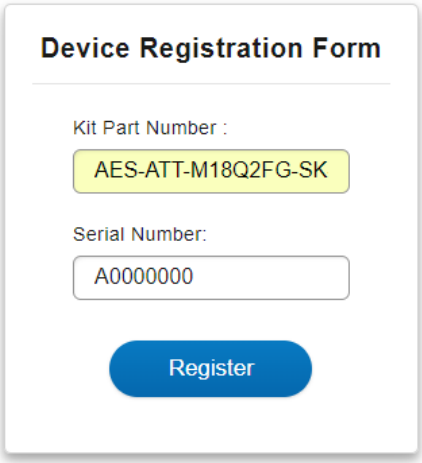


Figure 1.21

10. After clicking “Register”, which closes the dialog, you will find your kit listed on the Data dashboard:

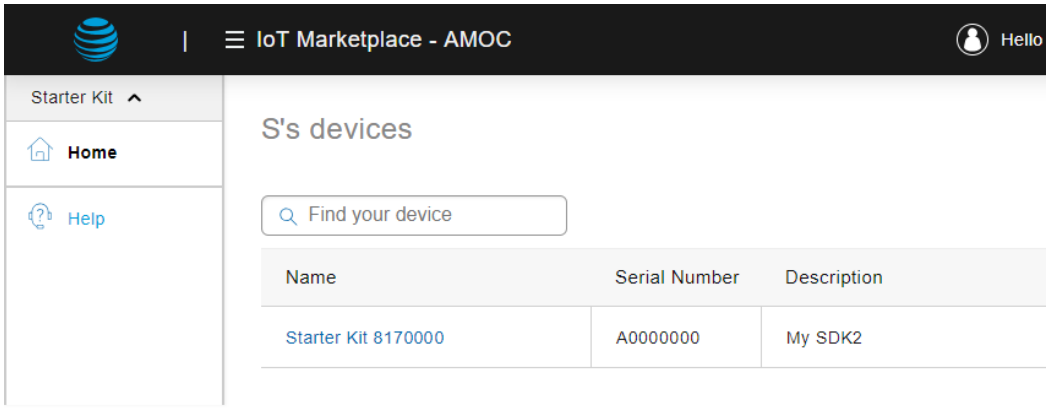


Figure 1.22

11. Click on the Starter Kit’s link (i.e. ‘Name’) and you will find the dashboard’s data displayed for your kit.

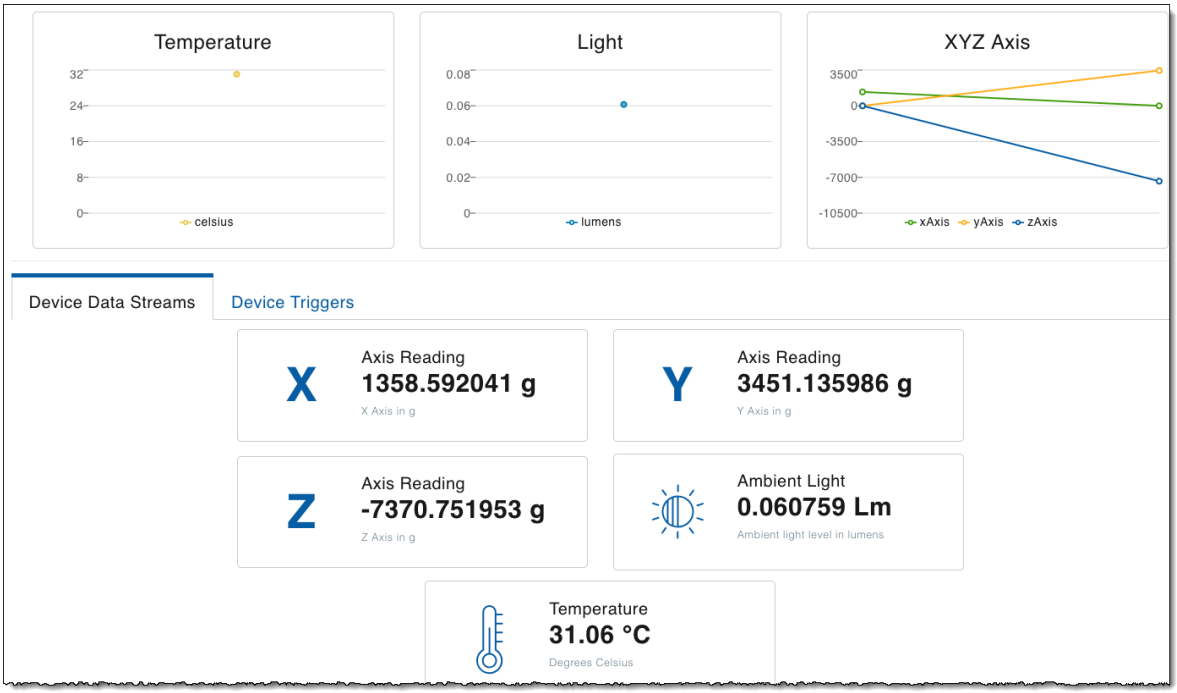


Figure 1.23

1.3.1.5.1. Sidebar – Data Sent to AT&T Dashboard

The following sensor data is sent to your AT&T Marketplace Dashboard each time the user presses the USER key:

3-Axis Acceleration Sensor Data:

- **XVALUE** = X data point from the onboard LIS2DW12 sensor chip
- **YVALUE** = Y data point from the onboard LIS2DW12 sensor chip
- **ZVALUE** = Z data point from the onboard LIS2DW12 sensor chip

Temperature Sensor Data:

- **TEMP** = Temperature data from the onboard LIS2DW12 sensor

Ambient Light Sensor Data:

- **ADC** = Light intensity measurement from the integrated ADC and onboard light sensor

The data is stored within your AT&T Marketplace Dashboard so that you can access this data from your IoT applications later.

1.4. Resources

1.4.1. AT&T Resources

AT&T provides several resources, many of them are listed in the preface of this book. Here are four sites to find support from AT&T for the SK2:

- https://marketplace.att.com/products/att-iot-starter-kit-2nd-gen
- https://marketplace.att.com/quickstart#starterkit-2nd-gen
- https://github.com/att-iotstarterkits
- https://att-iotservices.groovehq.com/help_center

1.4.2. Register with cloudconnectkits.org

Along with AT&T's information website, you can also find resources at Avnet's Cloudconnectkits.org. We recommend creating an account at:

http://cloudconnectkits.org/

During account creating, you will be able to register your kit.

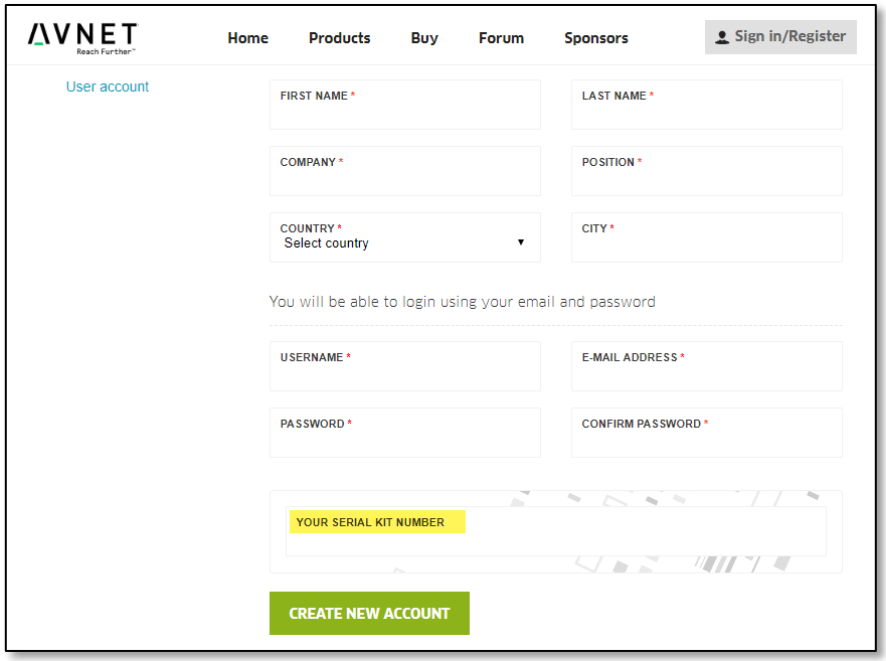


Figure 1.24

1.4.3. What's Next?

In the next chapter, we shall see that Embedded Linux is used as the baseline operating system for the SK2. Due to Linux's ease of use, this makes it easy for us to add, view and manage programs and files on the kit.

Additionally, the Linux command-line interface, accessed via one of the USB ports, makes it easy to control the user LED on the board.

2. Embedded Linux

Most of today's embedded systems run some form of operating system (OS). The OS provides a broad set of services that simplifies programming the device, making it easier – and faster – to deploy new products.

Linux is fast becoming a favorite operating system for embedded devices. Not only does it provide a rich set of services, but it has wide community support and is a favorite among developers.

The AT&T IoT Starter Kit (SK2) is based on an OpenEmbedded distribution of Linux. This environment provides the foundational basis for all interaction and development with this kit. If you are already a Linux master, then you'll have a big headstart with your development. If not, we think you will find this OS, and the kit itself, convenient and fun to program.

Using the SK2 is somewhat akin to using a Raspberry Pi in that they are both small, Linux-based, software-programmable kits. While the IoT Starter Kit doesn't provide quite the wide-range of functionality found in the Raspberry Pi, it has a built-in LTE cellular modem which gives it wide ranging IoT connectivity.

Topics

2.1. Introduction to Linux

Linux, like all operating systems, provides a set of services to the user.

If you have ever used a Linux computer – or another other computer (e.g. Windows, Mac) – you are familiar with many of these user-oriented services. For example, you may have used word processor program to create and edit files, which are stored in the OS's filesystem. Or, you may have used a web browser, which relies on the networking services provided by the operating system.

While many Linux-based "embedded systems" do not generally run big word processor programs, they still rely on the Linux filesystem to manage files – allowing programs to create, edit and delete files as needed.

And like most other operating systems, Linux provides a command-line interface that can be used to view files, execute programs or otherwise interrogate the system. While a command-line is rarely used during the execution of an embedded system's target application (i.e. we don't use a command-line to run our microwave), it is quite handy when developing and debugging software running user software.

While the following diagram is only a generic description, it gives us a summary of the types of services we can expect to find in Linux, such as: memory management, file systems, networking, and device drivers (e.g. serial ports, analog to digital converters). Together, this core group of services is often called the 'kernel' of an operating system, hence the name Linux Kernel.

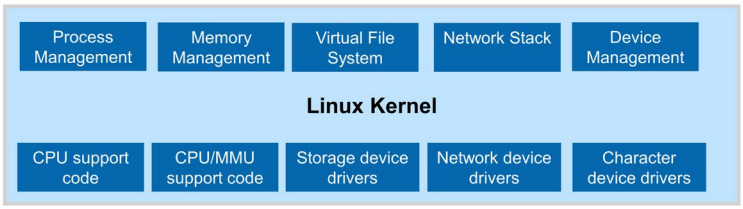


Figure 2.1 - Linux Kernel

During this chapter we hope to provide a brief introduction to embedded Linux, some details about the Linux distribution provided with the SK2, and how to interact with it. In subsequent chapters, we'll dig even further, learning how to write programs that run within the SK2's Linux environment.

2.1.1. Kernel Space vs User Space

Digging a little deeper into Linux, we need to delineate between “kernel” space and “user” space.

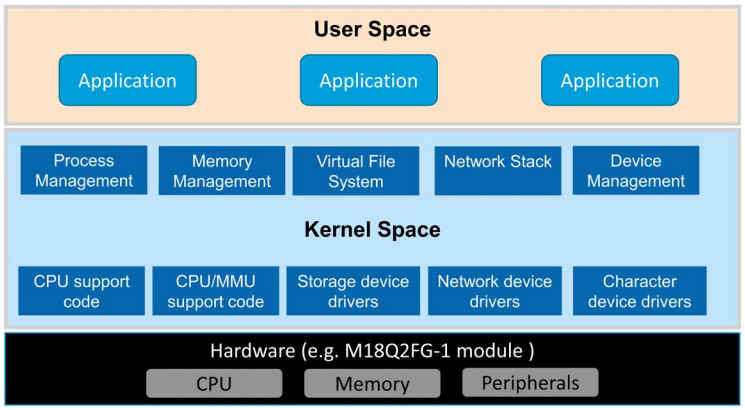


Figure 2.2 User Space vs Kernel Space

2.1.1.1. Kernel Space

In the previous section we described the Linux kernel as all the drivers and networking services provided by the OS. This code typically runs with permissions that allow it to directly interact with the hardware. With the SK2, the hardware manufacturer created the Linux kernel for us to work with their hardware. In other words, WNC adapted generic Linux code to work with the CPU, memory and peripherals found on their M18Q2FG-1 modem module.

2.1.1.2. User Space

In a similar way, we might describe ‘user’ space as all the programs and code that are isolated from the hardware. These programs access standard driver and socket interfaces that are portable across devices.

In fact, if you’ve used an Ubuntu Linux computer before, you might only recognize *user space*, as this is the area where we interact with Linux to run programs and configure our preferences. In an embedded system, we might think of *user space* as the area where we create and run software applications.

2.1.1.3. Protecting the Environment

Notice how Kernel space comes between User space and the hardware? Linux systems are layered in this fashion to create a secure and stable system. User applications are not allowed to talk directly to hardware, rather, they must call upon the services of the Linux kernel to interact with memory and peripherals. Therefore, when writing programs for Linux, we will need to learn how to interact with the Linux kernel – that is, we will need to learn how to call the functions provided by Linux and WNC that will let us access OS resources, such as the peripherals which talk to the sensors on the SK2, as well as the cellular LTE modem.

2.1.1.4. Protecting Users From Each Other

One last note, while we are talking about *protection* and *User space*. Another advantage to enforcing that all resources are accessed via the kernel is that it helps to protect one user from another. We might redraw our previous diagram to look like this:

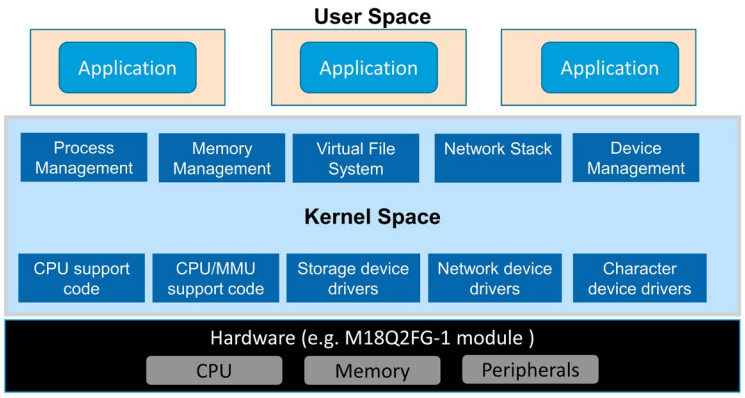


Figure 2.3 Separation of User Space

Linux allows programmers to create application programs which run independently of each other, as shown above. Alternatively, applications can be programmed to run in the same 'space' as each other (like on the previous page). While the choice is yours, as a programmer, how to implement your application(s) code, but we can thank the separation of Kernel space for helping to provide us with these options.

2.2. Linux Distribution

Linux is delivered as a 'distribution'. Generally, a distribution includes:

- Linux kernel – set of core services (as we discussed earlier)
- Filesystem – collection of user space libraries and utilities/applications
- Boot loader – an orderly, sequential means for low level hardware configuration and loading the kernel
- Toolchain – common collection of compiler, linker, libraries, etc.

For personal computers, there are many different examples of Linux distributions; for example, you may have seen: Ubuntu, Red Hat, and Suse. In fact, some of you may have created your own distributions by downloading the Linux source code and putting everything together yourself.

Thankfully, we won't have to build Linux from scratch for the SK2. In this case, our board comes pre-loaded with the Linux distribution. When writing code for the SK2, though, you will need to download the appropriate toolset – choosing whether you want to write your programs in Python or the C language. (Chapters 3 & 4 will describe each of these toolsets – how to download, install, and build programs with them.)

2.2.1. Linux Kernel

We access resources using a combination of common Linux commands (e.g. reading and writing a file or memory) or using a set of device driver libraries provided by WNC (the module vendor). These will be covered in greater detail throughout the rest of this user's guide.

2.2.2. Filesystem

The Linux filesystem provides a hierarchical organization for storing and retrieving files. Like most operating systems, the Linux community has defined a common set of directories for storage. As an example, the description below highlights a few key directories found on the SK2 filesystem (found on the right side of the page):

Filesystem - Unlike Microsoft Windows, Linux only has a single filesystem. In other words, Linux doesn't have a filesystem for each drive or entity. Where Windows users might be used to "C" and "D" drives, Linux only has a single filesystem.

In Linux, the topmost location – that is, the root of the filesystem – is denoted by "/".

We won't describe each folder in the SK2 filesystem, but here's a description for a few of them:

CUSTAPP	• For 'your' application (i.e. customer applications) • It's the only folder in the filesystem that can be written to; all other folders are read only • Contains the QuickStart demo that runs on powerup; this will be discussed further later in this guide
data	• This 'directory' is just a link to '/CUSTAPP'. • Like a shortcut in Windows or Mac /data -> /CUSTAPP
dev	• Common location for listing Linux device drivers
mnt	• Common location to mount other file systems • Additional drives, memory cards, or network locations become part of the one filesystem whenever they are added (i.e. "mounted") to the Linux device. "mnt" is a generic place to place them.
media	• Rather than using mnt for USB drives and CDROM media, some Linux distributions create a "media" where these items are mounted
sdcard	• This 'directory' is just a link to '/media/card'. • Like a shortcut in Windows or Mac /sdcard -> /media/card
proc	• "/proc" is not a real directory, but rather, it's a virtual directory and does not hold physical files. Contained within *proc* are information about processes and other system information. This information is mapped to /proc and mounted at boot time.

Here's a good reference, if you are looking to learn more about the standard Linux Filesystem Heirarchy:

https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard

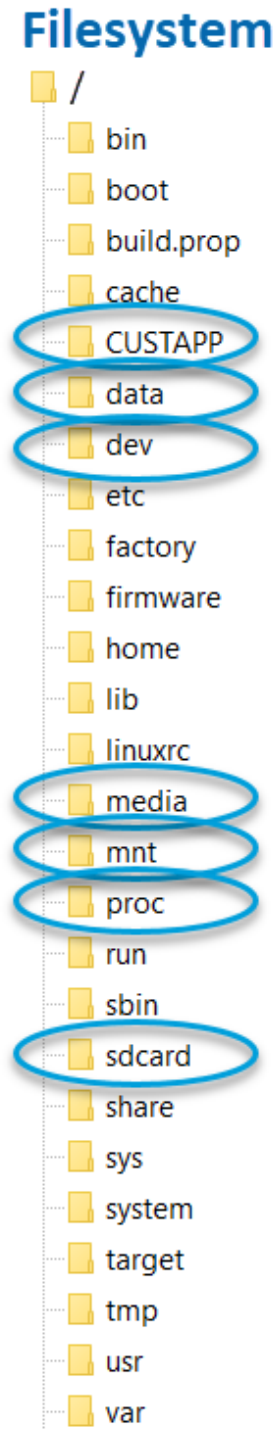


Figure 2.4 Filesystem

2.2.2.1. Virtual Files, Status Info, and Configuration

But, wait, there's more. The filesystem is a fundamental component of Linux. That is, it uses the filesystem for more than just text or binary files.

For one, Linux uses the file reading/writing paradigm to write to peripheral device-drivers. For example, you can write to the user-LED on the SK2 by writing to a virtual file, as we will see later in the chapter.

In a similar fashion, Linux lets users read status information and/or write configuration preferences through its filesystem. (As discussed earlier in the description of the /proc filesystem directory.)

2.2.2.2. Permissions and Owners

The Linux filesystem provides a robust set of file permissions and owners. To access a file, you must have the required permission to read, write, and/or execute it.

Here is another view of the SK2 filesystem. This one was generated using the "ls" Linux listing command.

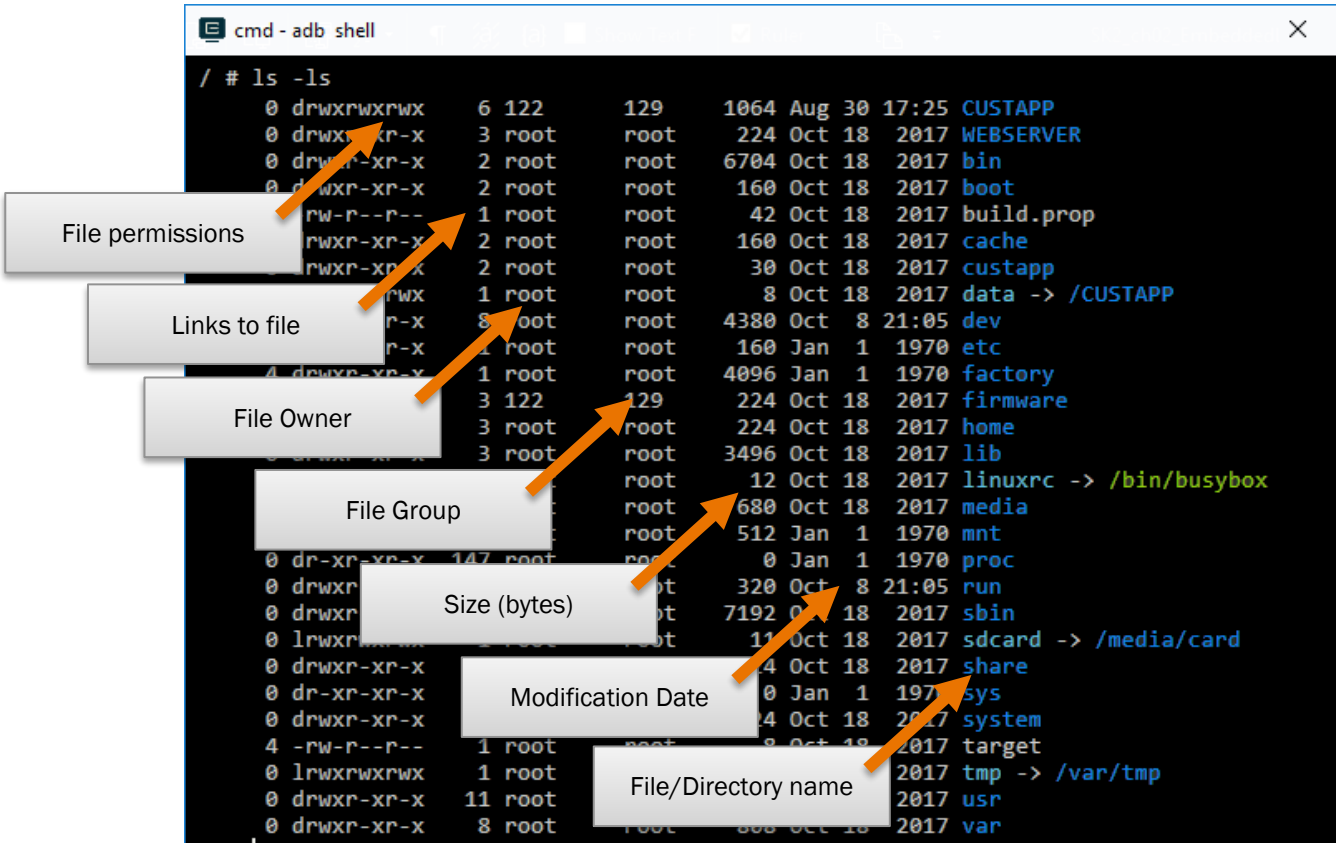


Figure 2.5 Filesystem Listing ("ls -ls")

Note that if the "File Permissions" begins with a "d" then that line represents a directory. The remaining characters indicate if the "owner", "group", and "all users" have permission to read, write, and execute.

The "Linux Commands" section lists several commands useful for listing (e.g. "ls"), reading and modifying the ownership of files, ("chown") as well as viewing and changing the permissions on files ("chmod").

2.2.3. Boot Loader

The SK2 Linux kernel has a pre-defined sequence for getting the module up-and-running. As part of this sequence, you can hook your programs to “auto-execute”. That is, you make your own programs begin running at power-up, just like the Out-of-Box example that ships with the SK2. This will be discussed in further detail during a later part of this Users Guide.

2.2.4. Toolchain

The set of tools used to write software programs to run in a specific version of Linux. In later chapters, we will introduce two toolchains for the SK2. One will focus on writing C programs, while the other will utilize Python.

2.3. Linux Shell

The Linux shell – also known as the Linux command-line – provides a textual interface for issuing commands to Linux and then viewing the results. While most operating systems provide this capability, Linux (and Unix) users, seem to rely on it more than users of other operating systems. With Embedded Linux – where limited memory may not be able to support graphical interfaces – the command-line shell becomes even more important.

The next section highlights several key commands that can be invoked from the Linux shell. Experienced Linux users will likely know these already. For those new to Linux, we provide a short explanation of each. With a little practice – and some Googling – all users should become comfortable with them.

Note: If you are familiar with using the Windows command line, then you should be familiar with the Linux shell. In some cases, both use the same command – for example both use the command "cd" for "changing the active directory". In other cases, they use different commands (e.g. Windows uses "dir" to list a directory, while Linux uses "ls").

Check out the ComputerHope site, which provides a brief comparison of both DOS and Linux command-line shells.

2.3.1. Basic Linux Commands

Linux distributions a common set of programs that can invoke from the Linux shell. For example, if you have ever used Linux – whether Ubuntu or embedded Linux – you will likely have made use of these various command-line programs. From listing the files in a directory ("ls"), to editing files ("vi"), or pinging a network ("ping").

Note: It's OK if you're not familiar with the tools we just mentioned. Those, and many more, will be discussed as needed throughout the rest of this user's guide.

In other words, the commands we run from the Linux shell command are just executable applications that reside within the Linux filesystem.

To minimize the size of the Linux on the SK2, its distribution packs all of these little command-line utility programs into a single executable called BusyBox. This is a common way for embedded Linux systems to include a large set of tools with a very small memory footprint, albeit with some minor tradeoffs in functionality.

2.3.1.1. BusyBox Commands

We mentioned earlier that the SK2, as do many Embedded Linux distributions, relies on BusyBox to provide many of the command line utilities that we use every day. For that reason, BusyBox is often called *"The Swiss Army Knife of Embedded Linux"*.

Here are a couple useful commands that can be used to interrogate BusyBox itself.

```
busybox
```

Entering the 'busybox' at the command line will return the version of BusyBox along with the command-line functions supported with this distribution.

```
/ # busybox
BusyBox v1.24.1 (2017-10-18 19:49:51 CST) multi-call binary.
BusyBox is copyrighted by many authors between 1998-2015.
Licensed under GPLv2. See source distribution for detailed
copyright notices.

Currently defined functions:
        [, [[, acpid, add-shell, addgroup, adduser, adjtimex, arp, arping, ash,
        awk, base64, basename, beep, blkid, blockdev, bootchartd, brctl,
        bunzip2, bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod,
        chown, chpst, chroot, chrt, cksum, clear, cmp, comm, cp, cpio, crond,
        crontab, cryptpw, cttyhack, cut, date, dc, dd, delgroup, deluser,
        depmod, devmem, df, dhcprelay, diff, dirname, dmesg, dnsd,
        dnsdomainname, dos2unix, du, dumpkmap, dumpleases, echo, ed, egrep,
        eject, env, envdir, envuidgid, ether-wake, expand, expr, fakeidentd,
        false, fbset, fbsplash, fdflush, fdformat, fdisk, fgconsole, fgrep,
        find, findfs, flock, fold, free, freeramdisk, fsck, fstrim, fsync,
        ftpd, ftpget, ftpput, fuser, getopt, getty, grep, groups, gunzip, gzip,
        halt, hd, hdparm, head, hexdump, hostid, hostname, httpd, hush,
        hwclock, id, ifconfig, ifdown, ifenslave, ifplugd, ifup, inetd, init,
        insmod, install, ionice, iostat, ip, ipaddr, ipcalc, ipcrm, ipcs,
        iplink, iproute, iprule, iptunnel, kbd_mode, kill, killall, killall5,
        klogd, less, linux32, linux64, linuxrc, ln, loadfont, loadkmap, logger,
        login, logname, logread, losetup, lpd, lpq, lpr, ls, lsattr, lsmod,
        lspci, lsusb, lzcat, lzma, lzop, lzopcat, makedevs, makemime, man,
        md5sum, mdev, mesg, microcom, mkdir, mkdosfs, mkfifo, mkfs.vfat, mknod,
        mkpasswd, mkswap, mktemp, modinfo, modprobe, more, mount, mpstat, mt,
        mv, nameif, nbd-client, nc, netstat, nice, nmeter, nohup, nslookup,
        ntpd, od, passwd, patch, pgrep, pidof, ping, ping6, pipe_progress,
        pivot_root, pkill, pmap, popmaildir, poweroff, powertop, printenv,
        printf, ps, pscan, pwd, raidautorun, rdate, rdev, readahead, readlink,
        readprofile, realpath, reboot, reformime, remove-shell, renice, reset,
        resize, rev, rm, rmdir, rmmod, route, rpm, rpm2cpio, rtcwake,
        run-parts, runsv, runsvdir, rx, script, scriptreplay, sed, sendmail,
        seq, setarch, setconsole, setfont, setkeycodes, setlogcons, setsid,
        setuidgid, sh, sha1sum, sha256sum, sha512sum, showkey, shuf, slattach,
        sleep, smemcap, softlimit, sort, split, start-stop-daemon, stat,
        strings, stty, su, sulogin, sum, sv, svlogd, swapoff, swapon,
        switch_root, sync, sysctl, syslogd, tac, tail, tar, tcpsvd, tee,
        telnet, telnetd, test, tftp, tftpd, time, timeout, top, touch, tr,
        traceroute, traceroute6, true, tty, ttysize, tunctl, udhcpc, udhcpd,
        udpsvd, umount, uname, unexpand, uniq, unix2dos, unlink, unlzma,
        unlzop, unxz, unzip, uptime, users, usleep, uudecode, uuencode,
        vconfig, vi, vlock, volname, watch, watchdog, wc, wget, which, who,
        whoami, xargs, xz, xzcat, yes, zcat, zcip
```

Figure 2.6 BusyBox

2.3.1.2. File Management

- **pwd**: print working directory – simply returns the directory where your command line is currently located; good for answering "where am I" in the filesystem?
- **ls** (small LS): listing – creates a simple listing of the files and subdirectories in the current working directory

  Make this more useful by adding an option, such as, "**ls -ls**" or "**ls -la**". This will list the files vertically and provide the additional information shown in the graphic in Section 2.2.2.2
- **alias**: tells the shell to replace one string with another

  This lets you create a new command from one (or more) other commands; for example:

  ```
  alias ll='ls -la'
  ```

  After entering this command, entering ll (small LL) will execture the "ls -la" command for you.
- **cd**: change directory – changes the current working directory to a new location

  Examples:
  - "**cd /**" sets your command-line session to the root of the filesystem
  - "**cd /CUSTAPP**" makes CUSTAPP your current working directory
- **cp**: copy – copies a file (or files) from one location to another
- **mv**: move – moves a file (or files) from one location to another
  Note that this is how you rename a file in Linux
- **rm**: remove – is how you delete a file or files
- **ln** (small LN): link – link files which is like shortcuts in Windows
  Also popular to create symbolic links, which adds an s option: "**ln -s**".
- **chmod**: change mode – allows you to change the permissions of files and directories
- **chown**: change ownership of files and directories
- **mkdir**: make a new directory
  - **rmdir**: remove directory – note that the directory must be empty before it can be deleted
- **mount**: mount a filesystem to your root (for example, attaching a USB or network drive)
  - Generally, you must create a new empty directory – for example, in the /mnt folder – and then mount the new filesystem into the root filesystem
  - **umount**: unmount – unmounting a filesystem from the root
- **find**: find a file or group of files
- **grep**: global regular expression print – search one or more files for lines that match a regular expression pattern
- **tar**: tape archive - combine a group of files into the *tar* archive format with - or without - compression; the tar command can also be used to modify, extract and manage tar files

2.3.1.3. Viewing, Creating and Editing Text Files

- **touch** – Either creates a new empty file with the specified name or updates the files modification timestamp if the file already exists
- **cat**: catenate – reads data from the specified file(s) and outputs the contents to the command-line
- **vi**: visual editor – a small, simple text editor included with most Linux distributions

  It's not visual (or anything) like Microsoft Word, but it lets us view and modify text files while taking up very little of our valuable memory

2.3.1.4. Program Control

- **ps**: process listing – produces a list of processes running on the Linux system
- **top**: produces a listing of processes sorted by % CPU usage
- **kill**: tell Linux to kill a process using it PID (process ID) number; a PID can be viewed using the ps or top command
- **<ctrl>-c**: stops a process by sending it the SIGINT interrupt signal
- **clear**: clears the Linux shell

2.3.2. Shell Scripting

As discussed earlier, the command-line shell provides a handy and powerful way to work with Linux, such as managing files or executing programs.

Behind the scenes, this *shell* itself is a Linux program that provides the command-line interface and command interpreter. There are a variety of shell programs found in Linux (and Unix) distributions – the most common being one called Bash. Alternatively, the SK2 comes with a similar, smaller shell program called Ash, which is part of the BusyBox toolset. (In fact, you can see this listed in *Figure 2.6 BusyBox*.)

Shell scripting is nothing more than a sequence of command-line (i.e. shell) calls. These sequences can be a simple grouping of one or two commands or complicated sequences that contain logical control statements.

We use the term "scripting", rather than "programming", as these sequences do not need to be explicitly compiled before they are executed.

When grouped together into a single text file, the ".sh" file extension is used to signify a *sh*ell script. Linux doesn't require that we use the .sh extension, but this is common practice in Linux since this makes it easier for us to identify which files are shell scripts.

Listing 2.1: Chapter_02/list.sh is an example of a very simple shell script that was used to create *Figure 2.5 Filesystem Listing ("ls -ls")*:

```
# list.sh
# list the files in the current directory to the command-line
# note that the -ls option creates a vertical listing with more info
ls -ls
```

Listing 2.1: Chapter_02/list.sh

A second example lists the files in the `dev` directory:

```
# list_dev.sh
# change to the /dev directory
# and then print a listing the command line
cd /dev
ls -ls
```

Listing 2.2 list_dev.sh

Let's look at one final example:

```
# list_pipe_cat.sh
#
# - This script writes a listing of /dev directory
#   into the file mylisting.txt in /CUSTAPP
# - The '>' pipes tells the shell to pipe the output
#   into the txt file rather than to the standard output
# - It then prints out the contents of mylisting.txt
#   using the 'cat' command

ls -ls /dev > /CUSTAPP/mylisting.txt
cat /CUSTAPP/mylisting.txt
```

Listing 2.3: Chapter_02/list_pipe_cat.sh

Later, in Section 2.4.4.3 on page 56, we walk you through using shell scripts on your SK2.

Note: **Linux Line Endings**

Be careful when creating shell scripts in Windows for use in Linux on your SK2. Windows uses different characters to signify line-endings than Linux. In some cases, the different characters can return unexpected results.

A couple of guidelines to help you get better results:

- Do not use Windows Notepad to create shell scripts. It does not handle Linux line endings.
- Find and use a good Windows text editor. There are many good ones available. A popular, free editor is Notepad++ (https://notepad-plus-plus.org).

2.4. ADB – Connecting to the SK2

Now that we are familiar with several Linux commands, how do we connect to the SK2 to view the filesystem and utilize these commands?

The SK2 was developed using ADB (Android Debug Bridge) to communicate between your computer and the kit. ADB is a versatile command-line tool that facilitates a variety of device actions, such as pushing and pulling files, as well as providing access to a Unix shell that you can use to run a variety of commands on your device.

It is a client-server program that includes three components:

1. (SK2) An ADB daemon is resident on your SK2, running in the background, providing development and debug access to authorized users through the COMmunications USB port.
2. (Computer) A client, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing ADB commands.
3. (Computer) A server, which manages communication between the client and the daemon. The server runs as a background process on your development machine. Once the server has set up connections to all devices, you can use adb commands to access those devices.

Since the ADB daemon is already installed on your SDK, we only need to install ADB onto our development computers. This is addressed in the next section.

2.4.1. Installing ADB

Since ADB is already installed on your SK2, we only need to install the remaining two components on your development computer; these are both installed with a single installation program.

Installing ADB on your Windows, Mac or Linux computer follows a similar set of steps. After downloading and extracting the ADB executables, you will also need to set up an access key file which will grant you authorization to connect to your SK2.

2.4.1.1. Install ADB Executables

1. **Download the ADB ZIP file for your computer from the Android developers' site.**

   There are separate download links for each OS (Windows, Mac, Linux). Pick the one that matches your development computer.

   https://developer.android.com/studio/releases/platform-tools

---

Note: Debian-based Linux users, such as those using Ubuntu Linux, can type the following command to install ADB:

```
sudo apt-get install adb
```

---

2. **Extract the contents of this ZIP file into an easily accessible folder.**

   Some suggestions include:

   Windows: **`C:\adb`**
   Mac: **`/Users/MY_USER_NAME/Desktop/adb`** *(replacing MY_USER_NAME with your own)*
   Linux: **`/Users/MY_USER_NAME/Desktop/adb`** *(replacing MY_USER_NAME with your own)*

Note: You can skip this step if, as a Linux user, you installed ADB with apt-get.

2.4.1.2. Set up Credentials (i.e. access key)

3. **Create a new folder called ".android" in your user directory.**

   Using your computer's file browser (or command-line), create the new folder as shown:

   Windows: **`C:\Users\MY_USER_NAME\.android`**
   Mac: **`/Users/MY_USER_NAME/.android`**
   Linux: **`/Users/MY_USER_NAME/.android`**

   replacing MY_USER_NAME with your own user account folder.

4. **Add "`adbkey.pub` to your new "`.android`" folder.**

   For the SK2, this is simply a text file named "adbkey.pub" which contains a single word:

   ```
   wnc000000
   ```

   Hint: This file should contain only this word. It should not contain a carriage return or line-feed.

   You can create this file on yourself or download it from github.com:

   https://github.com/Avnet/AvnetWNCSDK/blob/master/adbkey.pub

   For example, in Windows with a user name of "doug", when this step is complete you should have the following in your .android directory:

   ```
   C:\Users\doug\.android\adbkey.pub
   ```

2.4.1.3. Connect to Your SK2

5. **Open a command window in the same directory where you installed the ADB binary.**

   In **Windows**, this can be done by holding *Shift* and *Right-clicking* within the ADB folder then choosing the "open command prompt here" option. (Some Windows 10 users may see "PowerShell" instead of "command prompt".)

   Mac users can open "Terminal" from their Applications folder and navigate to where you installed ADB (in step 2). For example:

   ```
   cd /Users/MY_USER_NAME/Desktop/adb
   ```

   Linux users should open their command-line shell and and navigate to where you installed ADB (in step 2). For example:

   ```
   cd /Users/MY_USER_NAME/Desktop/adb
   ```

   Note: Linux users who installed ADB with apt-get may run adb commands from any location. You do not need to navigate to the adb folder in order to run it.

6. **Connect your SK2 to your computer with a USB cable.**

   Make sure the USB power cable is also connected. It won't work without both USB cables being connected.

7. **In the Command Prompt window, enter the following command to launch the ADB daemon:**

Windows:	`adb devices`
Mac and PowerShell users: (precede commands with "`./`")	`./adb devices`
Ubuntu Linux:	`sudo adb devices`

   In response, you should see:

   ```
   WNC_ADB device
   ```

   As shown below:

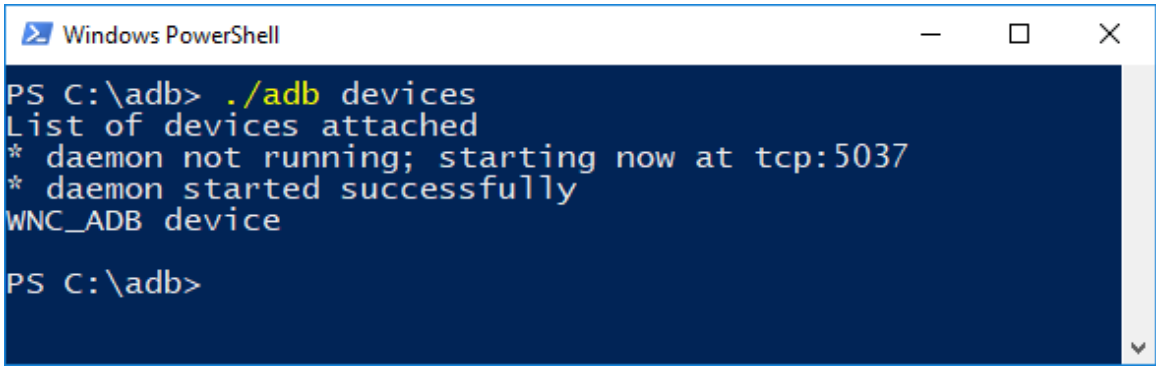


Figure 2.7 adb devices

   You can now run ADB commands on your device – which we'll look at in Section2.4.4 on page 53.

2.4.2. Sidebar - What happens during “adb devices”

When you run “adb devices” a few things happen:

1. ADB Server is started. (This was briefly described in Section 2.4 on page 45.)
2. The ADB Server scans your computer for connected ADB devices.
3. It then verifies if you have the credentials to access the connected ADB devices.
   a) Looks for the required ‘key’ in your .android folder.
   b) Creates the .android folder, if it doesn’t exist.
   c) If it finds the key, is uses that to create the actual authorization key “adbkey”.
4. Finally, it prints the list of devices to the command line (as we saw in the diagram for Step 7 of Section 2.4.1.3).

2.4.3. Troubleshooting “adb devices”

Please refer to this section if you failed to connect to your device in Step 7 of *Installing ADB* (Section 2.4.1). In other words, you can skip this section if executing “adb devices” returns:

```
WNC_ADB device
```

Otherwise, examine some of the problems and solutions listed here.

2.4.3.1. Precede Command with ./

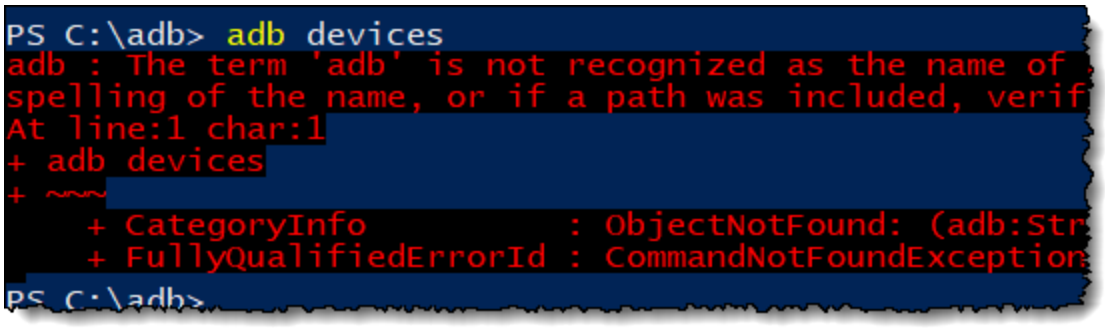


Figure 2.8 Mac/Linux/Powershell needs “./”

When using Windows Powershell, Mac or Linux, you should precede the “adb” command with “./”. (Powershell also accepts “.\”.)

```
./adb devices
```

2.4.3.2. Execute From ADB Directory

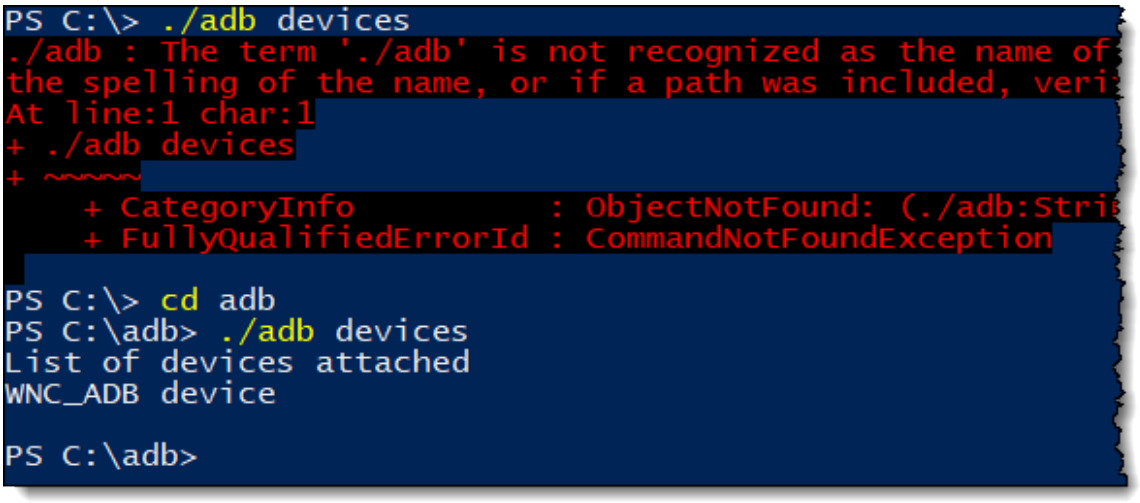


Figure 2.9 Use ADB Folder

The ADB executable wasn't found. Since "./adb" was used correctly for PowerShell, we know that wasn't the issue. But it appears we were trying to run ADB from the "C:/" drive location. Unless you modify your Windows configuration (i.e. PATH variable), you either need to specify the full path to ADB… or simply run ADB from the folder you installed it to.

Change to the ADB folder by using the "cd" command as shown above. Since we installed ADB to the "C:\adb" directory, and our cursor resides at "C:\", we only need to use:

```
cd adb
```

to get to the proper location.

2.4.3.3. WNC_ADB unauthorized

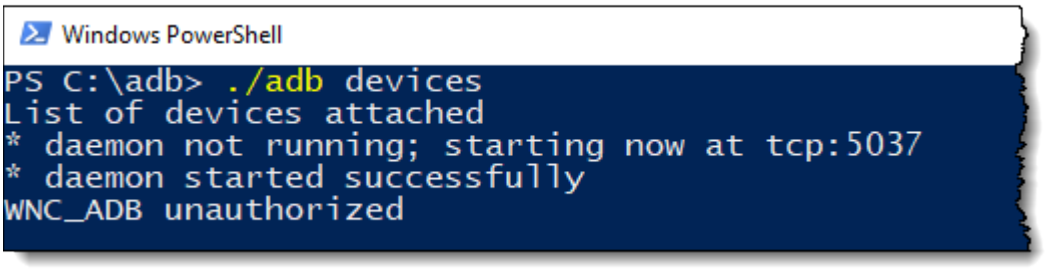


Figure 2.10 WNC_ADB Unauthorized

In this case, it appears that the ADB server started and found our WNC_ADB device but did not find the proper credentials to access the device.

Go and examine your `.android` directory (created in Step 3 on page 47.). It should now contain two files:

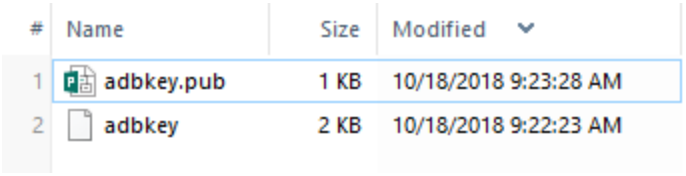


Figure 2.11 adb keys

"`adbkey`" is a private key generated by the ADB Server based upon the "`adbkey.pub`" file we placed in our `.android` directory.

If you examine "`adbkey`" with a text editor, you will notice it looks like:

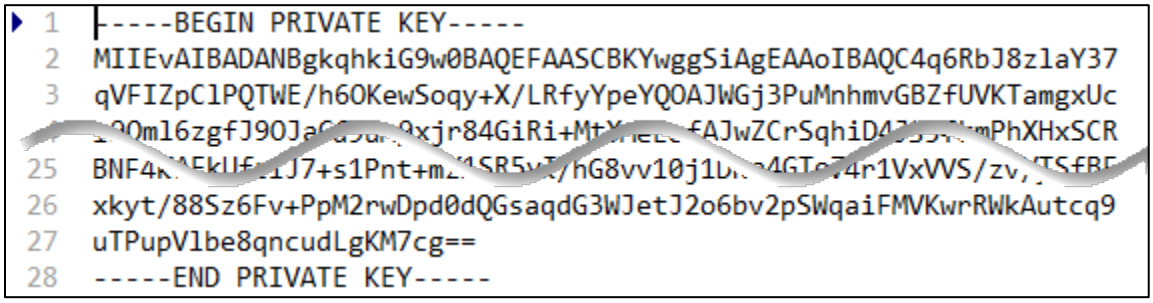


Figure 2.12 Private key

If our device is unauthorized, then something is likely wrong with this file. Since it is generated by "`adbkey.pub`", it is likely that it has been corrupted. For some reason, it gets overwritten with some incorrect hash string.

Procedure to Correct Unauthorized

1. Double-check that "`adbkey.pub`" still contains the single word "`wnc000000`" (without the quotes). if it doesn't, delete the contents of this file and replace it with this word.

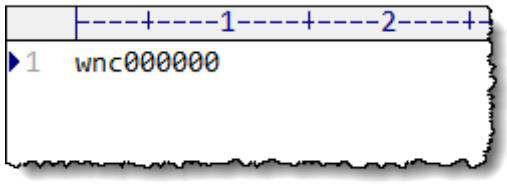


Figure 2.13 adbkey.pub

> **Hint:** This file should contain only this word.
> It should not contain a carriage return or line-feed.

2. Run the following ADB command to stop the server from running.

```
adb kill-server
```

or

```
./adb kill-server
```

3. Run the "adb devices" command again.

You can see the sequence we followed below.

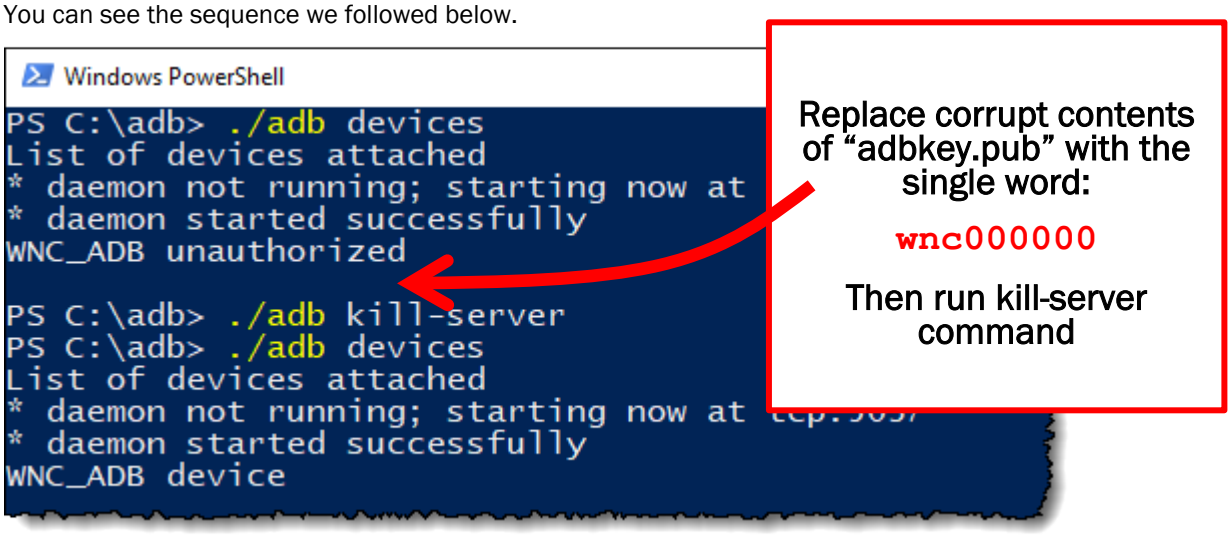


Figure 2.14 kill-server and re-running adb devices

If these three steps do not result in "WNC_ADB device", try working through them again. We have occasionally seen it require these 2 or 3 times for it to work correctly.

2.4.4. ADB Commands

Once ADB is installed on your development computer, there are several ADB commands available for use. On this page we will describe a subset of ADB commands that will be used throughout this user guide.

2.4.4.1. ADB Commands found in this User Guide

ADB Debugging

- adb devices – lists available adb targets
- adb kill-server – stop the adb server
- **adb reboot** – reboots the SK2

File Management

- adb pull – pull a file from the SK2
- adb push – push a file to the SK2

Execute on Device

- **adb shell** - Starts a remote shell on the SK2
- **exit** – exits the remote shell and returns to your computers command-line

Of course, you can find a listing of all the commands from many places on the Internet, such as: adbshell.com, https://developer.android.com/studio/command-line/adb, droidviews.

2.4.4.2. Exploring the SK2 Filesystem using "adb shell"

This section assumes that you were able to install ADB on your computer and successfully list your kit using the "adb devices" command. If not, please refer to *Installing ADB* (Section 2.4.1).

Here is a screen capture from our computer when executed the following instructions. (Note that this example uses the CMDR shell running on Windows.)

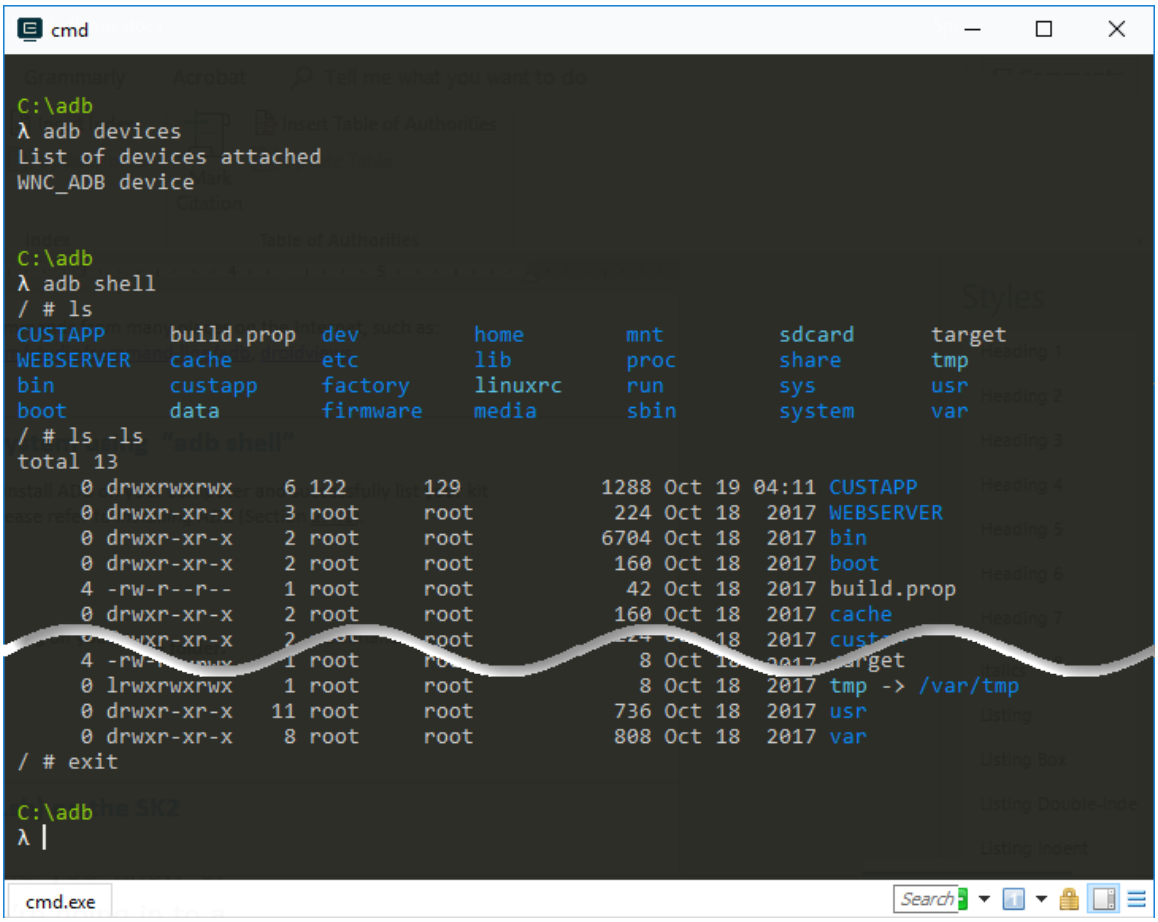


Figure 2.15 Using "adb shell"

1. **Open a command-line shell on your computer in the ADB folder.**

   This was discussed in Step 5 (Section 2.4.1.3) on page 48.

2. **Verify you can connect to your board using ADB.**

   Running this command from the shell you just opened:

   `adb devices` or `./adb devices`

   Should return:

   `WNC_ADB device`

   If it doesn't, then you need to verify your ADB installation (Section 2.4.1) and/or view ADB troubleshooting (Section 2.4.3).

3. **Start a remote session on your SK2 using "adb shell".**

   You can start a remote shell session running on your SK2 using the *adb shell* command.

   ```
   adb shell or ./adb shell
   ```

   Notice, in *Using "adb shell* (Figure 2.15 Using "adb shell"), that the command prompt changes to "#" after running the "adb shell" command. The commands executed from the # shell are running on the SK2 (and not on your computer).

4. **List the SK2 filesystem "`ls`".**

5. **List the SK2 filesystem again using "`ls -ls`".**

6. **Exit the SK2 remote shell.**

   Exit the remote shell using the exit command.

   ```
   exit
   ```

   Notice how the command prompt returns to its original value. This indicates that our command-line is running on our computer again.

2.4.4.3. Running Shell Scripts (.sh) on the SK2

To demonstrat running shell scripts on your SK2, we will use the script (Listing 2.2 list_dev.sh) we examined earlier that lists the contents of the `/dev` directory. We will create (or download) the shell script on our computer and then *push* it to the SK2 and execute it.

Note: See a screen capture of this procedure at the end of the instructions.

---

1. **Create (or download) the `list_dev.sh` file to your adb folder.**

   Since this is such a small file, you may prefer to create it using you're a text editor. If so, create a file called "list_dev.sh" and add the following two lines of code.

   ```
   cd /dev
   ls -ls
   ```

   Note: If using Windows, save the file using Linux (or Unix) line endings.
   (See note on page 44).

   Alternatively, you may want to download this file from the AT&T Starter Kit GitHub site:

   https://github.com/att-iotstarterkits/sk2-Users-Guide

   Hint You can place your shell file in any folder, we only chose the adb folder for convenience.

   ---

2. **Open a command-line shell on your computer, if it isn't already open.**

   This was discussed in Step 5 (Section 2.4.1.3) on page 48.

3. **Verify your SK2 connection using "adb devices".**

   It should return:

   ```
   WNC_ADB device
   ```

   Otherwise you need to verify your ADB installation (Section 2.4.1) and/or view ADB troubleshooting (Section 2.4.3).

4. **Push the shell script to the /CUSTAPP folder.**

   The ADB push command takes two arguments, the from and to location.

   ```
        adb push "C:/adb/list_dev.sh" /CUSTAPP
   or ./adb push "C:/adb/list_dev.sh" /CUSTAPP
   ```

   Modify the path to your `list_dev.sh` file as necessary.

5. **Open a shell session on your SK2.**

   ```
        adb shell
   or ./adb shell
   ```

6. **Change directories, opening /CUSTAPP.**

   ```
   cd /CUSTAPP
   ```

7. **List the /CUSTAPP directory and look at the permissions for your `list_dev.sh` file.**

   ```
   ls -ls
   ```

   You will see the shell file has the following permissions.

   ```
   4 -rw-rw-rw-  1 root  root   104 Oct 19 04:15 list_dev.sh
   ```

   Unfortunately, since the permissions do not include an 'x', you won't be able to execute the script. In fact, you can try to run it, if you'd like.

8. **Modify permissions so that you can execute your shell script.**

   There are several permutations for setting file permissions. By using the "+x" argument with the CHMOD command we simply enable execute permission for our script file. (We recommend that you explore file permission options further on your own.)

   ```
   chmod +x list_dev.sh
   ```

   Hint Handily, Linux will autofill filenames when possible. For example, in this step, typing "chmod +x lis" and hitting the tab key should autofill the full filename.

9. **Check the permissions on the file after chmod.**

   ```
   ls -ls list_dev.sh
   ```

   Notice that the executable flags are set: `-rwxrwxrwx`

10. **Run the `list_dev.sh`.**

    Since we are using the SK2 Linux shell, we must append "`./`" when running executables.

    ```
    ./list_dev.sh
    ```

    The contents of the /dev directory (the SK2 Linux drivers) should be printed to your terminal.

11. **Notice that you remained in the /CUSTAPP directory.**

    Even though the shell script changed to the working directory to `/dev`, the script returned to the /CUSTAPP directory when it finished running.

    This happens because shell scripts are run in their own sub-shell. This is often handy, especially when running multiple sequential scripts.

    Using the "source" command (in the next step) if you want to affect the working directory.

12. **Run your script with the "source" command to have the script affect the working directory.**

    source ./list_dev.sh

    This time, your working directory should be /dev when the script completes.

13. **Exit the ADB shell.**

    ```
    exit
    ```

Here is a recording of our running the list_dev.sh shell script.

```
C:\adb
λ adb devices
List of devices attached
WNC_ADB device


C:\adb
λ adb push "C:\adb\list_dev.sh" /CUSTAPP
C:\adb\list_dev.sh: 1 file pushed. 0.0 MB/s (104 bytes in 0.005s)

C:\adb
λ adb shell
/ # cd CUSTAPP/
/CUSTAPP # ls -ls
total 5572
   420 -rw-r--r--    1 root     root        427055 Oct 19 06:59 all.log
     4 -rwxrwxrwx    1 root     root            52 Jan  1  1970 custapp-postinit.sh
     4 -rw-r--r--    1 122      129           4096 Oct 18  2017 custapp.squashfs
     0 drwxr-xr-x    2 root     root           304 Jan  1  1970 fwup
     0 drwxrwxrwx    2 root     root           304 Jan  1  1970 iot
     4 -rw-rw-rw-    1 root     root           104 Oct 19 04:15 list_dev.sh
     0 drw-r--r--    2 root     root           312 Jan  1  1970 psm
     0 lrwxrwxrwx    1 root     root            11 Jan  1  1970 upload -> /mnt/upload
     0 drwxr-xr-x    5 root     root           360 Jan  1  1970 user
/CUSTAPP # chmod 777 list_dev.sh
/CUSTAPP # ls -ls list_dev.sh
     4 -rwxrwxrwx    1 root     root           104 Oct 19 04:15 list_dev.sh
/CUSTAPP # ./list_dev.sh
total 8
     0 crw-rw----    1 root     root       10,  51 Jan  1  1970 android_mbim
     0 crw-rw----    1 root     root       10,  42 Jan  1  1970 android_rndis_qc
     0 crw-rw----    1 root     root      248,   3 Jan  1  1970 apr_apps2
     0 crw-rw----    1 root     root      243,   0 Jan  1  1970 at_usb0
     0 crw-rw----    1 root     root      243,   1 Jan  1  1970 vcs1
     0 crw-rw----    1 root     tty         7,  [illegible] Jan  1  1970 vcs1[illegible]
     0 crw-rw----    1 root     tty         7, 128 Jan  1  1970 vcsa
     0 crw-rw----    1 root     tty         7, 129 Jan  1  1970 vcsa1
     0 crw-rw-rw-    1 root     root        1,   5 Jan  1  1970 zero
/CUSTAPP #
/CUSTAPP # source ./list_dev.sh
total 8
     0 crw-rw----    1 root     root       10,  51 Jan  1  1970 android_mbim
     0 crw-rw----    1 root     root       10,  42 Jan  1  1970 android_rndis_qc
     0 crw-rw----    1 root     root      248,   3 Jan  1  1970 apr_apps2
     0 crw-rw----    1 root     tty       243,   0 Jan  1  1970 vcs1[illegible]
     0 crw-rw----    1 root     tty         7,  [illegible] Jan  1  1970 vcsa
     0 crw-rw----    1 root     tty         7, 129 Jan  1  1970 vcsa1
     0 crw-rw-rw-    1 root     root        1,   5 Jan  1  1970 zero
/dev # exit

C:\adb
λ
```

Figure 2.16 Running the list_dev.sh script

2.4.4.4. Modifying shell script with vi

You can use the "vi" text editor to create and modify text files with your SK2. This can be handy if you need to make a change to a file already resident on the board.

This section assumes you have created the shell script (list_dev.sh) used in the previous section. Additionally, please refer back to previous sections if you need with the early instructions in this sequence.

1. Open a command-line shell on your computer, if it isn't already open.
2. Verify your SK2 connection using "adb devices".
3. Open a shell session on your SK2.
4. Change directories, opening /CUSTAPP.

```
cd /CUSTAPP
```

5. Copy "list_dev.sh" to "list_lib.sh".

```
cp list_dev.sh list_lib.sh
```

6. Open the **`list_dev.sh`** file for editing.

```
vi list_lib.sh
```

7. Enter insert mode in the vi editor.

```
i
```

8. Change the cd command.

```
From: cd /dev
To:   cd /lib
```

9. Exit the editing mode.

```
<esc>
:
```

10. Write the changes and quit the file.

```
wq
<Return>
```

11. Run the newly edited script.

```
./list_lib.sh
```

If you are used to editing with Microsoft Word, the "vi" editor may take some time to get used to. But, it's a convenient – and very powerful – text editor that resides inside your SK2.

Please search the web for numerous pages describing how to use vi's many options. Here are two to get you started:

- https://www.howtogeek.com/102468/a-beginners-guide-to-editing-text-files-with-vi/
- https://www.cs.colostate.edu/helpdocs/vi.html

2.5. Controlling Hardware Using the Linux Shell

The SK2 hardware can be controlled from the Linux shell. While it isn't very effective to try and control an I2C serial port from the command-line, the LED makes a convenient example.

The following examples demonstrate how to turn the WWAN LED on/off. (Refer to the Hardware Overview in Chapter 1 if you cannot find the WWAN LED.)

2.5.1.1. Writing the WWAN LED

In the filesystem, the WWAN LED is found under the /sys directory. This is another directory, along with /dev, that maps hardware to the Linux filesystem.

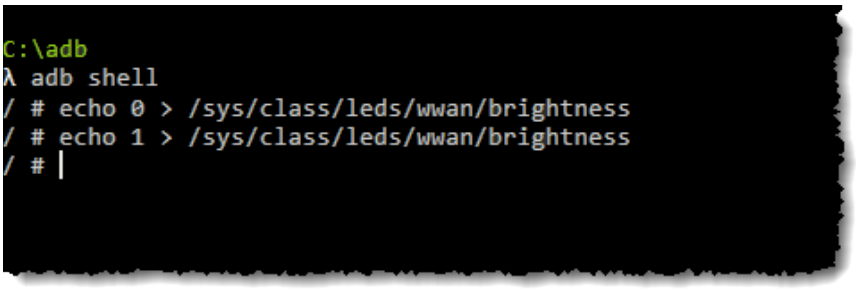


Figure 2.17 Turning the WWAN LED off/on

To turn off the LED, we simply write a "0" to its associated (virtual) file.

```
echo 0 > /sys/class/leds/wwan/brightness
```

Listing 2.4 Turn Off the LED

Similarly, we can turn the LED on by writing a "1" to the same location.

```
echo 1 > /sys/class/leds/wwan/brightness
```

Listing 2.5 Turn On the LED

2.5.1.2. Simple Blink LED Script

Here's a very simple script which blinks the WWAN LED.

```
# blink.sh
#
# This is a simple example for blinking the WWAN LED five times
# - The For loop executes once per character (a thru e)
# - 'sleep 1' causes the cpu to wait 1 second
# - At the end of the script, the LED is turned off
#
for var in a b c d e; do
        echo  0 > /sys/class/leds/wwan/brightness
        sleep 1
        echo  1 > /sys/class/leds/wwan/brightness
        sleep 1
done
echo 0 > /sys/class/leds/wwan/brightness
```

Listing 2.6: Chapter_02/blink.sh

2.5.1.3. Using the RGB LED or USER Button

While the WWAN LED is controlled directly by the WWAN pin on the WNC module, the RGB LED and USER button are controlled by GPIO pins.

Here's a simple example for lighting the Red LED (in the RGB LED).

```
# TurnOnRedLed.sh
#
# This simple example turns on the Red RGB LED
#
cd /sys/class/gpio

echo 38 > export
echo out > gpio38/direction

echo 1 > gpio38/value
```

Listing 2.7: : Appendix/GPIO/TurnOnRedLed.sh

Refer to the Appendix on GPIO for more details concerning:

- What is GPIO?
- How do I use the Linux GPIO driver to access the RGB LED and Button?
- Addtional GPIO shell examples

2.6. Linux Boot Sequence

Linux follows an orderly boot sequence from power-up to user shell access. This sequence of scripts, functions, and tasks prepares the operating system to host users or run programs. While most of these details are not generally of interest – especially on the SK2 since we cannot affect them – the final steps may be useful for us to understand.

In our case, the questions of interest may include:

- How does the SK2 QuickStart demo start running automatically?
- Can I stop the demo from running automatically?
- Once I have written a program of my own, can I start it running automatically?

These are the questions we'll address in this section.

2.6.1. How Does the QuickStart Demo Run Automatically?

The QuickStart demo consists of a C program called `iot_monitor`, since it monitors several sensors and communicates that information over the Internet. (Note that throughout the various chapters in this book we'll explore the various facets of the `iot_monitor` program.)

So how exactly is "iot_monitor" getting started? Let's look at the sequence of events starting at the end of the Linux boot sequence.

1. The simple answer is that Linux always calls the following script file after its done booting up:

   `/CUSTAPP/custapp-postinit.sh`

2. Examining the script file, we find it calls another script called `run_demo.sh`:

   Listing 2.8 - custapp-postinit.sh

   ```
   start-stop-daemon -S -b -x /CUSTAPP/iot/run_demo.sh
   ```

   The start-stop-daemon is used to start system level processes as described by: http://www.man.he.net/man8/start-stop-daemon

3. Finally, when examining the `/CUSTAPP/iot/run_demo.sh`, script we see that it calls the actual `iot_monitor` program (along with a few arguments).

   Listing 2.9 - run_demo.sh

   ```
   iot_monitor -q5 -a a2e26b03f4e77aab23dbc5294b277d69
   ```

Bottom line, you can control what programs autostart on your SK2 by editing (or deleting) the `custapp-postinit.sh` script.

Note: There's the simple, short answer (described here)... and the longer, more involved answer that is in the Appendix Linux Boot Sequence discussion.

2.6.2. Stop the QuickStart Demo from Running Automatically

The QuickStart demo can be turned off by holding the USER Button down for longer than 3 seconds. After doing so, you will see the RGB LED turn off.

Since the iot_monitor program autostarts after power-cycling the SK2, turning off the program using the USER Button is only temporary. To stop the program altogether we need to edit, delete, or rename `the custapp-postinit.sh` script. In this case, let's rename the script and then verify that the QuickStart program doesn't run.

2.6.2.1. Stop the QuickStart Demo

1. **Power-cycle your SK2 and verify the QuickStart demo runs.**

   Unplug, then plug back in the USB power to your SK2. The QuickStart demo should begin running within a minute. You can refer back to Chapter 1 (Running the QuickStart Demo) for more information about the demo.

2. **Open a command-line shell on your computer in the ADB folder.**

   This was discussed in Step 5 (Section 2.4.1.3) on page 48.

3. **Verify you can connect to your board using ADB.**

   Running this command from the shell you just opened:

   ```
   adb devices or ./adb devices
   ```

   Should return:

   ```
   WNC_ADB device
   ```

   If it doesn't, then you need to verify your ADB installation (Section 2.4.1) and/or view ADB troubleshooting (Section 2.4.3).

4. **Start a remote session on your SK2 using "adb shell".**

   You can start a remote shell session running on your SK2 using the *adb shell* command.

   ```
   adb shell or ./adb shell
   ```

   Notice, in *Using "adb shell* (Figure 2.15 Using "adb shell"), that the command prompt changes to "#" after running the "adb shell" command. The commands executed from the # shell are running on the SK2 (and not on your computer).

5. **Navigate to the /CUSTAPP folder.**

   ```
   cd /CUSTAPP
   ```

6. **Rename the custapp-postinit.sh script to something else.**

   In Linux we can use the the "move" command to rename a file. In this case, let's just append ".txt" to the end of the filename.

   ```
   mv custapp-postinit.sh custapp-postinit.sh.txt
   ```

7. **List the directory to verify the name was changed.**

   ```
   ls -l
   ```

8. **Exit the ADB remote shell.**

```
exit
```

9. **Power-cycle your SK2 and watch the LEDs to verify the QuickStart demo doesn't start.**

   If the WWAN and RGB LEDs do not light up within a minute or two, it's a safe bet that the QuickStart demo is not running.

Autostart the Blink Script

If you created and ran the `/CUSTAPP/blink.sh` script in Section 2.5.1.2, let's try running that script automatically. If not, we suggest that you return to that section and create the script and put it into the `/CUSTAPP` directory before doing the following steps in this section.

10. **Reconnect to the ADB shell.**

```
adb shell
```

11. **Change to the /CUSTAPP directory.**

```
cd /CUSTAPP
```

12. **Verify that blink.sh exists and is working.**

```
./blink.sh
```

The WWAN LED should blink 5 times.

13. **Create a new copy the `custapp-postinit.sh` file.**

```
cp custapp-postinit.sh.txt custapp-postinit.sh
```

14. **Edit the `custapp-postinit.sh` file.**

```
vi custapp-postinit.sh
```

a) **Enter Insert/Edit mode.**

```
i
```

b) **Modify the script to be executed:**

The original script runs `/CUSTAPP/iot/rundemo.sh`. Change this to run our `blink.sh` script. Afterward editing it should read:

```
start-stop-daemon -S -b -x /CUSTAPP/blink.sh
```

c) **Exit Insert/Edit mode.**

```
<esc>
```

d) **Quit and save the file.**

```
:wq
```

15. **Verify the file is correct.**

```
cat custapp-postinit.sh
```

Which should print out the contents:

```
start-stop-daemon -S -b -x /CUSTAPP/blink.sh
```

16. **Exit the ADB remote shell.**

```
exit
```

17. **Power-cycle the board to view blink running.**

After 10-15 seconds, the `blink.sh` script runs and blinks the WWAN LED five times.

2.6.2.2. Autostart the QuickStart Demo

Assuming that you have completed the previous two sections (2.6.2.1 and 0) we can return the SK2 to its original state of running the QuickStart demo by renaming (or deleting) our new custapp-postinit.sh file and restoring the original file.

18. **Enter the ADB shell and change to the /CUSTAPP directory.**

```
adb shell
```

19. **Rename your new custapp-postinit.sh file by adding appending blink to the name.**

```
mv custapp-postinit.sh custapp-postinit.sh.blink
```

20. **Restore the original custapp-postinit.sh file.**

```
mv custapp-postinit.sh.txt custapp-postinit.sh
```

21. **Verify the original file was restored by listing the directory and cat'ing the file.**

```
ls -l
cat custapp-postinit.sh
```

22. **Exit the shell.**

```
exit
```

23. **Power-cycle your SK2 to verify the original QuickStart demo runs again.**

    As an alternative, rather than power-cycling the board, you could execute:

```
adb restart
```

    to restart the SK2. However you restart the board, you should see the QuickStart demo begin running within 45-60 seconds, as it did in Chapter 1 (and at the beginning of this Section 2.6.2).

3. Installing and Using C/C++

The SK2 (AT&T IoT Starter Kit - 2nd generation) is supported by a variety of programming tools and languages. As demonstrated in the previous chapter, simply by the fact it's running Linux, the board can be programmed using simple shell scripts. In the next chapter, we explore how you can use the simple, yet powerful Python language to build and run applications. For more demanding applications, though, this chapter discusses how to utilize the C or C++ languages to build and run Linux programs for the SK2.

The chapter begins with an introduction to the WNC Software Development Kit (SDK) and its installation. With the tools installed you can download the source and rebuild the *IoT Monitor* application that comes pre-installed on your SK2.

After building and running the large, production-level IoT application, we pivot to building our own applications. Starting small with *Hello World* provides us an opportunity to examine how to use the GNU Automake toolset to build a simple application. Next, we examine how to program the same GPIO resources – discussed in the previous chapter – using the C language. The chapter ends with a higher-level discussion of building Linux C programs, multi-tasking, and using signals & interrupts.

Prerequisite Knowledge and Tools

This guide assumes that you are generally familiar with the C or C++ language. Language constructs are not explained or taught in this book, although we do cover those topics that deal specifically with using C/C++ on the SK2 running under Linux. Please refer to the "Additional Resources" section at the end of the chapter for more information about building C or C++ programs – or writing such programs running under the Linux operating system.

Additionally, you will need to have Internet access when installing the tools and software during the first few sections of this chapter.

Note: Developing applications for the SK2 using C/C++ is only supported on Linux platforms. Neither Windows or Mac platforms are supported.

Unfortunately, it is not an uncommon for development tools supporting Linux-based embedded controllers to require development on a Linux host platform. As such, if you do not have access to a Ubuntu Linux computer your options include:

- Create a virtual Ubuntu computer using software such as VMware Workstation (Windows), VMware Fusion (Mac), Parallels (Mac), or Virtual Box (Windows and Mac). This essentially allows you to run a Linux computer inside the virtual computer application on your Windows (or Mac) computer.

  The examples provided with this user guide were built and tested using Ubuntu 16.04 LTS running within VMware Workstation 15 virtual machine on a Windows laptop.
- Create a second boot partition on your computer and install Ubuntu to host and run the WNC SDK development tools.
- Program the SK2 using shell scripts (Chapter 2) or Python (Chapter 4).

Topics

3.1. Installing the C/C++ Tools and Software

Note: Once again, the C/C++ toolchain has been developed and tested for the Linux platform. The examples and procedures in this book have been created and tested while running Ubuntu 16.04 LTS.

If you only have a Windows computer, you may find it necessary to create a second boot partion with Ubuntu or create a virtual Ubuntu computer using software such as VMware or Virtual Box. Creating such an environment is outside the scope of this document, but the procedures and examples described in this book have been tested on Ubuntu 16.04 running under VMware Workstation 14 and 15.

As described in Chapter 1, the WNC module on the SK2 contains a single, user-programmable Cortex-A7 processor that runs Linux. This is the processor that runs your software applications. The tools and libraries needed to create your applications programs are found in the WNC Software Development Kit (SDK), as well as natively within your Ubuntu Linux computer. After installing the SDK, your Ubuntu development computer will be able to create ARM programs that will run under Linux on the SK2.

3.1.1. GIT (and ADB)

Before installing the SK2's software development kit (SDK), we need to make sure two other necessary tools are available on your computer: GIT and ADB.

GIT is a popular version control system used to manage software development and deployment. In fact, we will use GIT to download and install the SDK in the next section of this document. Your Ubuntu computer needs to have the GIT tools installed for you to clone and download the tools, as well as Avnet's example IoT application.

If you are working sequentially through this User's Guide, your system should already have ADB installed, which was needed to explore Linux in Chapter 2. We include the installation of ADB here just in case you are skipping around and have not installed it already – and the following won't hurt anything even if it's already been installed.

3.1.1.1. GIT (and ADB) Installation Procedure

The following steps will install both GIT and ADB onto your computer.

1. **Open a command-line terminal window on your Ubuntu computer.**

   This was covered in Section 2.4.1.3. "Connect to Your SK2".

2. **Use apt-get to install GIT and ADB.**

   APT-GET is a tool used for downloading and installing packages from the Internet. The following command will install the tools if they haven't already been installed.

   ```
   sudo apt-get install git adb
   ```

   Prefixing commands with SUDO (superuser do) tells Linux to treat the command as if it's coming from a superuser, who has advanced privileges. Just as with Windows or Mac, Ubuntu Linux requires superuser privileges to install programs.

   When executing SUDO commands, you may be asked to enter your logon password for your Ubuntu computer, after which you should see the tools being installed (if they weren't previously installed).

3.1.2. Software Development Kit (SDK)

The WNC SDK will be installed next. It contains the additional tools and libraries required to build programs for the SK2. As mentioned in the previous section, we'll use GIT to download and install the SDK onto our Linux computer.

3.1.2.1. SDK Installation Procedure

The following steps will download and install the WNC SDK.

1. **Open a command-line terminal window on your Ubuntu computer.**

2. **Navigate to your home directory.**

   ```
   cd ~
   ```

   The "cd" stands for "change directory.
   The tilde "~" is Linux shorthand for the path to your user's home directory.

3. **Make a new directory "AvNet2" for the SDK.**

   While you can do this with the Ubutu file manager, it's just as easy to do it from the command line.

   ```
   mkdir AvNet2
   ```

   where *mkdir* stands for "make directory".

4. **Change to the new *AvNet2* subdirectory.**

   ```
   cd AvNet2
   ```

5. **Clone the WNC SDK from Avnet's GitHub site.**

   On your command line enter:

   ```
   git clone https://github.com/Avnet/AvnetWNCSDK
   ```

   which tells our computer to create a clone of the git repository found at the given URL.

6. **Change to the cloned AvnetWNCSDK directory and view its contents.**

   The clone of the WNCSDK created a new directory inside of AvNet2. Change to that directory and view its contents.

   ```
   cd AvnetWNCSDK
   ls -l       (list command "ls" with the small -L option)
   ```

   You should see the following files in the directory:

   - adbkey.pub
   - adb_usb.ini
   - Avnet M18Qx LTE IoT API Guide.docx
   - Avnet M18Qx Perpherial IoT Guide.docx
   - oecore-x86_64-cortexa7-neon-vfpv4-toolchain-nodistro.0.sh
   - README.md

7. **(Optional) Copy DOCX files and convert to PDF.**

   The API guides are useful references when writing programs. One guide describes the LTE communication library API, while the other describes the Peripheral API.

   Hint: If you find it difficult to open and use Word DOCX files for quick reference books, you may want to create PDF versions of these two guides and place them in a more convenient location on your computer. Alternatively, you may prefer to print them out, if that makes referencing them easier for you.

8. **Install the SDK using the provided shell script.**

   Make sure your command-line cursor is still in the AvnetWNCSDK directory and execute the cloned shell script:

   ```
   sudo ./oecore-x86_64-cortexa7-neon-vfpv4-toolchain-nodistro.0.sh
   ```

   Respond with “y” (for yes) when asked if you want to install the SDK.

   Note: After successful installation, the SDK script tells us that we need to enter the following configuration command each time we open a new shell session where we plan to build code for the SK2. Please make note of this command:

   ```
   . /usr/local/oecore-x86_64/environment-setup-cortexa7-neon-vfpv4-oe-linux-gnueabi
   ```

9. **Run the shell configuration command indicated by the SDK installation script.**

   ```
   . /usr/local/oecore-x86_64/environment-setup-cortexa7-neon-vfpv4-oe-linux-gnueabi
   ```

   This script must be run each time you open a new command-line terminal and want to build code for the SK2. (And, don’t miss the “.” at the beginning of the command.)

3.2. (Re)Build IoT_Monitor example

With the development tools and software installed, let's download the code for the QuickStart demo, rebuild it from source, and download it to the SK2. This allows us to verify that the tools and software were correctly installed.

3.2.1. Clone the IoT_Monitor Source Code

As we learned in the first two chapters, the QuickStart demo – which runs on the SK2 directly after opening the box and powering up the board – is actually a C++ program called "iot_monitor". Like the WNC SDK, we recommend downloading the program source code using GIT.

1. **Open a command-line terminal window on your Ubuntu computer.**

2. **Navigate to your "AvNet2" directory.**

   If you followed the previous procedures in this chapter, you can get there using the following command:

   ```
   cd ~/AvNet2
   ```

3. **Clone the IoT Monitor program from Avnet's GitHub site.**

   ```
   git clone https://github.com/Avnet/M18QxIotMonitor
   ```

   This command will create a new subdirectory "M18QxIotMonitor" and download the contents of the GitHub project into it.

4. **View the contents of the new Iot Monitor program directory.**

   ```
   ls M18QxIotMonitor
   ```

```
superiorview@ubuntu:~/AvNet2$ ls M18QxIotMonitor
aclocal.m4                  config.sub          HTS221Reg.h     lis2dw12.c    mal.hpp          mytimer.c
arm-oe-linux-gnueabi-libtool configure          http.c          lis2dw12.h    maljson.cpp      mytimer.h
autogen.sh                  configure.ac        http.h          lis2dw12.o    maljson.o        mytimer.o
autom4te.cache              demo.c              http.o          ltmain.sh     mal.o            program_apn.bat
binio.c                     demo.o              HTU21D.cpp      m18qx_sao     MAX31855.cpp     qsapp.cpp
binio.h                     depcomp             HTU21D.hpp      m2x.c         MAX31855.hpp     qsapp.o
binio.o                     emissions.cpp       HTU21D.o        m2x.h         MAX31855.o       README.md
commands.cpp                emissions.o         install-sh      m2x.o         microrl_config.h stamp-h1
commands.o                  fact_test.bat       iot_monitor     m4            microrl.cpp      TSYS01.hpp
compile                     facttest.cpp        iot_monitor.h   main.cpp      microrl.h        TSYS02D.hpp
config.guess                facttest.o          jsmn.c          main.o        microrl.o        wwan.c
config.h                    firmware_update.bat jsmn.h          Makefile      missing          wwan.o
config.h.in                 HTS221.cpp          jsmn.o          Makefile.am   MS5637.cpp
config.log                  HTS221.hpp          KMA36.hpp       Makefile.in   MS5637.hpp
config.status               HTS221.o            LICENSE         mal.cpp       MS5637.o
superiorview@ubuntu:~/AvNet2$
```

Figure 3.1 - Listing of M18QxIotMonitor

You will learn about many of these files as you work through this User Guide. You can also watch a video tutorial that briefly describes many of the files in this program directory.

Device Fundamental Tutorial 4 can be found here: https://vimeo.com/247415717

3.2.2. Build the IoT Monitor Program

With the tools, libraries and QuickStart IoT Monitor program source code installed, let's try building the application for ourselves. In all, there are four commands needed to build programs for the SK2:

- ❶ Configure your shell environment (the command provided by SDK installation)
- Run the GNU automake tools
  - ❷ Autogen
  - ❸ Configure
- ❹ Run the makefile

After making sure we're in the correct directory, let's walk through the procedure to see each of these commands in action.

5. **Execute the environment configuration command, if you haven't already done so after opening your terminal command-line window.**

```
. /usr/local/oecore-x86_64/environment-setup-cortexa7-neon-vfpv4-oe-linux-gnueabi
```

6. **Navigate to the "M18QxIotMonitor" project directory.**

```
cd ~/AvNet2/M18QxIotMonitor
```

7. **Run the GNU automake tools to create a makefile for our project.**

   The GNU automake system automatically builds makefiles for your project. This requires running two programs which are found in the project's directory. (These files will be discussed further in the next section.)

   a) Run the "autogen.sh" shell script that's found in the project directory

```
./autogen.sh
```

   b) Run the "configure" automake utility.

```
configure ${CONFIGURE_FLAGS}
```

   Once complete, these two commands will have automatically created a makefile for your program.

8. **Run the makefile to build the IoT Monitor application.**

```
make
```

> **Hint:** While writing and debugging code, you only need to run the "make" command to rebuild the program. The first three commands must be run under these conditions:
>
> • Environment configuration needs to be run each time you start a new shell session.
> • The **`autogen.sh`** and **`configure`** commands only needs to be run if you add, modify, or delete any files associated with your project.

9. **List the directory and verify the application was built.**

   List the directory with the -l (small L) option and check for the "iot_monitor" application. You should see this file in your project directory. Verify that it exists with a time-stamp consistent with your running the "make" command.

```
ls -l
```

3.2.3. Push and Execute the IoT Monitor App You Just Built

After building the Iot Monitor application, we need to push it onto the SK2 before we can run it. Even though both environments are running Linux, the IoT Monitor was built to run on the ARM Cortex-A7 and therefore cannot be tested by running it on your Ubunto Linux PC.

Since the steps required to push and run a Linux program on your SK2 were covered in Chapter 2, we only provide a cursory outline of these steps below. Please refer to Chapter 2 if you need more details.

10. **Make sure you're in the M18QxIotMonitor directory (i.e. where your program file is).**

```
cd ~/AvNet2/M18QxIotMonitor
```

11. **Verify that your SK2 is powered on; plugged into your computer; and that your Ubuntu computer recognizes the device.**

```
Type this          →     adb devices

Response should be →     List of devices attached
                         WNC_ADB device
```

Please refer to Chapter 2 for troubleshooting questions, if needed.

12. **If required, stop the IoT Monitor program that automatically runs when powering up your SK2.**

Chapter 2 discussed how the IoT Monitor program begins running automatically – and how you can prevent this from happening. Most users want to stop the program from automatically running befored starting to write and debugging their own programs.

If your SK2 is running the out-of-box IoT Monitor program, you can temporarily stop it by pressing and holding the USER BUTTON (middle button) on the SK2 for 10 seconds. This will quit the application, but it will auto-start again the next time you power up the board.

13. **Use the ADB *push* command to send your program file to the SK2 "/CUSTAPP" directory.**

```
adb push iot_monitor /CUSTAPP
```

If successful, you will see a response that specifies the size, speed, and time for the transfer. *(Note that your results make differ from those shown here.)*

```
                    93 KB/s (99080 bytes in 1.071s)
```

If you prefer, you can place your program into a subdirectory of CUSTAPP (Customer Applications) as it does not need to be stored at the root of this directory. That said, since CUSTAPP (and its subdirectories) are the only "writeable" directories on the SK2, your programs must be stored somewhere within CUSTAPP.

14. **Open a remote shell to your SK2 board.**

```
adb shell
```

When running the remote shell on the SK2, you should see your command-line prompt change to "#" character.

15. **Change to the CUSTAPP directory**

```
# cd /CUSTAPP
```

16. **Run the program you pushed to the CUSTAPP directory.**

```
# ./iot_monitor
```

Not only will the program behave as when it autostarts, if you are remotely connected via an ADB terminal session, you can see the textual output from the program as well as control it using the program's terminal commands.

3.2.4. Avnet IoT Monitor GitHub and Videos

The IoT Monitor Avnet GitHub page provides a summary of these instructions along with a link to Avnet's Device Foundation videos, which walk through many of the steps listed towards the beginning of this chapter.

GitHub site: https://github.com/Avnet/M18QxIotMonitor

Video Tutorial 1: Setting up the Development Environment (roughly covers Users Guide 3.1)

Video Tutorial 2: Downloading and building the IoT Monitor program (User's Guide 3.2)

Video Tutorial 3: Running the IoT Monitor program

Video Tutorial 4: Brief explanation of Iot Monitor program files

3.3. Create Your First SK2 C/C++ Program

We realize that “Hello World” isn’t a significant C program – especially for embedded systems where standard output terminals don’t usually exist – but starting with a simple C program lets us concentrate on what it takes to build and run a program, rather than dwelling on the details of the C code itself.

3.3.1. Hello World

The “Hello World” program shouldn’t be a surprise if you have ever studied or writen C code. This file simply uses the Standard I/O runtime library to print “Hello World” to the terminal.

Listing 3.1: Chapter_03/hello/src/main.c

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

3.3.2. GNU Automake Build System

As described earlier in the chapter, the WNC SDK uses the GNU Automake tools. These tools simplify the process of building application programs because they automatically build the makefile(s) required to for creating applications. What are makefiles? They are the instructions that tell the toolchain (e.g. compilers, linkers, etc) how to compile and construct your application programs.

When building your own application programs, rather than building and maintaining your own makefiles (which takes lots of time to become proficient), we suggest that you copy and modify the “hello” example, tweaking it to meet your requirements. *(This is what we have done to create the examples provided for the user guide.)*

Unfortunately, it’s outside the scope of this guide to present the details of the GNU make and GNU automake tools; although, there are many such tutorials on the web. Our objective is to show you how to make use of the tools as provided by the WNC SDK.

3.3.3. Starting Project Files

Here is an inventory of the *starter* files needed to build our "hello" project. There will be many more files created after building the application, but these are the only required files when using the build system for the WNC SDK toolchain. *(Actually, the two shell script files highlighted in blue are not required, but we added them for user convenience.)*

```
Chapter_03/hello
             |   autogen.sh
             |   configure.ac
             |   Makefile.am
             |   _1_autoname.sh
             |   _2_build.sh
             |
             /---src
                    main.c
                    Makefile.am
```

Figure 3.2 - Starter files in C/C++

The make system does not require users to put source code into a folder called "src". This is common practice by many GNU automake users, but not a requirement. In fact, the IoT Monitor example discussed earlier in this chapter used a single, flat folder. For the User Guide examples, though, we chose to place our source code into the "src" subdirectory.

Here's a summary of the project files, a brief description, and what you will *need to modify when reusing this example* to build your own applications.

File Name	Need to Edit?	Description
autogen.sh	No	Shell script that runs the automake tools.
configure.ac	No	Configuration script for the 'configure' automake tool. Only need to edit this file if you change the subdirectory structure of your project.
Makefile.am	No	Specifies project subdirectories – again, you only change this file if you rename/modify the project's subdirectories.
_1_autoname.sh	No	This shell script builds a new src/Makefile.am (backing up any current version). It renames the executable program app from the name of the project folder. Additionally, it adds all .c and .cpp files found in the 'src' folder.
_2_build.sh	No	Runs the commands required to build the application and push it to the SK2.
src/main.c	Yes	This is your programs source code. You may also add additional c/c++ files to the 'src' directory. • Your program must contain a "main.c" or "main.cpp" • The main() function must return an 'int'. You app does not require main() to have any arguments.
src/Makefile.am	Maybe	The *_1_autoname.sh* script builds this file for you. For complex programs, though, you may have to manually edit *Makefile.am*.

(While GNU Automake power-users may want to edit all these files and more, most users can get by with these recommendations.)

Note that even a successful execution of the `_1_autoname`.sh script may generate the following errors:

```
ls: cannot access '*.c': No such file or directory
ls: cannot access '*.cpp': No such file or directory
mv: cannot stat 'src/Makefile.am': No such file or directory
```

3.3.4. Building “Hello”

Here are the steps required to build the “hello” project.

1. **Open a command-line terminal window on your Ubuntu computer.**
2. **Navigate to the “hello” directory.**

   For example:

   ```
   cd ~/sk2_users_guide/Chapter_03/hello
   ```

3. **Run the build shell script.**

   ```
   ./_2_build.sh
   ```

   This will follow the build steps outlined in Section 3.2.2. Then the script will push the executable application program, residing in the “src” (`src/hello`), via ADB to the /CUSTAPP folder, if your SK2 is connected.

Hint:	Running the `_2_build.sh` script is convenient, but it always executes each step in the procedure. If you are repeatedly building over-and-over again while debugging code, you can speed up your build time by running “make” and manually pushing your application to the SK2.

3.3.4.1. .gitignore

If you are using GIT for your own development, you likely know that a ".gitignore" file tells GIT which files you do not want to save into your repository. For example, even though running the build tools creates many files, we relied on .gitignore to only retain the absolute minimum files, as presented in Section 3.3.3.

We created our .gitignore file by starting with the recommendation from gitignore.io and adding the files we wanted to ignore that were generated by the GNU automake tools.

Listing 3.2: sample .gitignore

```
# Created by https://www.gitignore.io/api/c++
# Edit at https://www.gitignore.io/?templates=c++

### Backup Files ###
*.bak

### C++ ###
# Prerequisites
*.d
.deps/

# Compiled Object files
*.slo
*.lo
*.o
*.obj

# Precompiled Headers
*.gch
*.pch

# Autogenerated Automake Config Files
autom4te.cache/
m4/
*.m4
arm-oe-linux-gnueabi-libtool
compile
*.guess
config.h
*.in
config.log
config.status
config.sub
configure
depcomp
install-sh
ltmain.sh
Makefile
missing
stamp-h1

## Compiled Static libraries
#*.lai
#*.la
#*.a
#*.lib

# End of https://www.gitignore.io/api/c++
```

3.4. Embedded “Hello World”

The embedded processing community eschews the common definition of “Hello World” that we just examined in the previous section. Rather, they think of “blinking an LED” as their version of the simple “Hello World” program.

Once you can blink and LED, you’ve learned how to build a simple program that can talk to real hardware. That’s a great place to start with any embedded system.

In Chapter 2, we were able to use shell scripts to control pins (connected to LEDs) by writing to Linux GPIO device drivers via virtual files. Since C can read/write files, we can do the same using the C language. (In a later section, we’ll explore using the WNC SDK API to interface to write code for the Linux device drivers.)

3.4.1. Blink LED with File I/O

As described in Chapter 2 – as well as the GPIO section of the Appendix – we can observe and control GPIO pins using the virtual file system provided by the Linux device drivers. The resulting code is very similar to the shell scripts in Chapter 2, only it uses the C standard I/O routines to make it so.

To quickly summarize working with GPIO, your program needs to:

- Export the driver – which allows user space access to the driver
- Set the drivers direction (in/out)
- Write the value (0/1) based on whether you want the LED off/on

The Red LED is connected to GPIO pin 38 on the WNC module. You can find a listing of the LED and USER SWITCH pin numbers in the Appendix section titled “GPIO Pin Numbers – WNC vs Qualcomm”.

Note: When accessing GPIO using file I/O we need to use the *Qualcomm Pin Number* (#38 as shown in the following `fileio_red_led` example). Conversely, when using the WNC SDK API you will need to use the *WNC Pin Number* (#98 as shown later in the `api_red_led` example).

To blink an LED, you must turn it on, then off, while waiting for some time period between the on and off states. We chose to use a 1-second time interval which was accomplished using the usleep(X) function – which sleeps the processor for X microseconds.

To fit the code onto a single page (i.e. the next page), we removed some of the error checking, as well as some printf’s to the terminal we used for debugging the code. You can find the full source code in the project: `Chapter_03/fileio_red_led`.

Listing 3.3: Blinking Red LED with File I/O (Chapter_03/fileio_red_led)

```
#include <stdlib.h>
#include <stdio.h>      // Needed for fileio calls
#include <unistd.h>     // Needed for usleep() calls

#define GPIO_EXPORT     "/sys/class/gpio/export"

#define GPIO_RED        38
#define GPIO_RED_DIR    "/sys/class/gpio/gpio38/direction"
#define GPIO_RED_LED    "/sys/class/gpio/gpio38/value"

#define OFF             0
#define ON              1

#define SECOND          1000000   // One second = 1000000 microseconds

int main(void)
{
    int count;          // Loop counter for blinking
    int max = 4;        // Number of blinks before exiting program
    FILE *fptr;         // File pointer

    // Open GPIO #38 (red) for use by exporting to user space
    fptr = fopen(GPIO_EXPORT,"w");
    fclose(fptr);

    // Set direction for GPIO #38 (red)
    fptr = fopen(GPIO_RED_DIR,"w");
    fclose(fptr);

    // Turn off Red LED
    fptr = fopen(GPIO_RED_LED,"w");
    fclose(fptr);

    // Blink Red LED 'count' times
    for(count = 1; count <= max; ++count)
    {
        fptr = fopen(GPIO_RED_LED,"w");
        fprintf(fptr,"%d", ON);
        fclose(fptr);
        usleep(1 * SECOND);

        fptr = fopen(GPIO_RED_LED,"w");
        printf(fptr,"%d", OFF);
        fclose(fptr);
        usleep(1 * SECOND);
    }

    return 0;
}
```

3.5. Using the SDK's peripheral API

The WNC SDK provides a Peripheral Application Programming Interface (API) library which supports the various peripherals found on the WNC M18Qx module used in the SK2. We can utilize this peripheral interface to initialize and control the peripherals within our C/C++ programs.

The WNC Peripheral API supports the following peripherals:

- I2C
- SPI
- GPIO
- ADC

The GPIO peripheral interface is discussed in this chapter of the User's Guide. The remaining peripheals will be explored in future chapters.

3.5.1. GPIO API Summary

Application programming interfaces (APIs) consist of "data types" and "methods" (i.e. functions). Such is the case with the Peripheral API. The *Avnet M18Qx Perpherial IoT Guide.docx* file, found in the WNC SDK details the typedefs and functions for each peripheral. Turning to the GPIO Interface chapter we find the following elements defined:

Datatype	Example	Description
gpio_pin_t	GPIO_PIN_98	Specifies which GPIO pin should be initialized or deinitialized. Enumeration only supports the GPIO specific pins – and uses the WNC Pin Numbers.
gpio_direction_t	GPIO_DIR_OUTPUT	Specifies the direction of data transfer used by the gpio_dir() function.
gpio_level_t	GPIO_LEVEL_LOW	Is the pin value 0 (low) or 1 (high).
gpio_handle_t	hMyGpio	This value is returned by the gpio_init() function and is an argument to many other gpio_ functions. You can choose any valid C variable name.
gpio_irq_trig_t	GPIO_IRQ_TRIG_RISING	Passed to the gpio_irq_request() function, this argument indicates when an interrupt should be generated: when the signal goes from low→high, high→low, or in both cases.
gpio_irq_callback_fn_t	Any valid C function	This datatype is passed to the gpio_irq_request() function. It indicates which C function should be run in response to a triggered interrupt.

Notes:

- Commonly, definitions using the suffix "_t" are used to indicate data type.
- In many cases, the API guide specifies the valid enumerations (i.e. values) supported by the API. For example, gpio_direction_t allows for GPIO_DIR_OUTPUT and GPIO_DIR_INPUT.
- The API's datatypes support specific requirements needed by the API's function arguments and return values – that is, in those cases where a standard C datatype does not meet the need or may be unclear.
- For those used to programming low-level microcontrollers, you might think of the callback function (e.g. gpio_irq_callback_fn_t) being like an Interrupt Service Routine "ISR".

GPIO API Functions:

Allocating GPIO pin resources – the first three functions are used to allocate and de-allocate the hardware resources to be used by your program.

- gpio_init()
- gpio_deinit()
- gpio_is_inited()

The next three functions are used to configure the GPIO resources once they have been allocated. You always need to indicate the direction (in or out) that you plan to transfer data through the pin. But, you only need to use the two IRQ (interrupt request) functions when the pin is used as an "input" and your program needs the input pin to create an interrupt event when its value changes.

- gpio_dir()
- gpio_irq_request()
- gpio_irq_free()

The final two functions specify the data value of the pin to be transferred externally (when writing) or internally (when reading).

- gpio_write() -- when pin is configured as an OUTPUT
- gpio_read() – when pin is configured as an INPUT

Hint: To review all the details for the GPIO API functions and datatypes, please refer to the *"Avnet M18Qx Peripheral IoT Guide.pdf"* found in your WNC SDK installation or on the Avnet SDK GitHub page.

Note: We won't summarize the other peripheral API in this User's Guide, rather, we'll let you read through the API Guide on your own. We wanted, though, to discuss one of the peripheral's functions and datatypes for users who might not have experienced working with hardware API before the SK2.

3.5.2. Blink LED using the GPIO API

In section 3.4.1, we detailed using the GPIO pin connected to the red LED using the File I/O driver interface. This section looks at the same example but uses the Peripheral API to configure and control the pin. In fact, while the syntax may differ, you'll likely find the general steps in this example are very much like those found in the File I/O example.

3.5.2.1. api_led_red Example

Listing 3.4: Chapter_03/api_led_red/src/main.c

```
1  #include <stdint.h>
2  #include <stdio.h>                              // Needed for printf()
3  #include <unistd.h>                             // Needed for usleep()
4  #include <hwlib/hwlib.h>                        // Needed for WNCSDK GPIO API
5
6  #define LED_RED         GPIO_PIN_92
7  #define LED_GREEN       GPIO_PIN_101
8  #define LED_BLUE        GPIO_PIN_102
9  #define USER_BUTTON     GPIO_PIN_98
10
11 #define SECOND          1000000
12 #define NUM_BLINKS      3
13
14 int main(void) {
15     int i = 0;                                  // loop counter
16     int ret;                                    // return value
17     gpio_handle_t   myGpio;                     // GPIO pin handle
18
19     printf("Hello Gpio\n");
20
21     // Initialize GPIO for Red LED
22     ret = gpio_init( LED_RED, &myGpio );        // GPIO call to initialize RED LED
23     gpio_dir(myGpio, GPIO_DIR_OUTPUT);          // Set direction of GPIO pin to 'output'
24
25     // Blink LED NUM_BLINKS times
26     for ( i = 0; i < NUM_BLINKS; i++ )
27     {
28         gpio_write( myGpio, GPIO_LEVEL_HIGH );  // Turn on the Red LED
29         usleep( 1 * SECOND );                   // Wait 1 second
30
31         gpio_write( myGpio, GPIO_LEVEL_LOW );   // Turn off the Red LED
32         usleep( 1 * SECOND );                   // Wait 1 second
33     }
34
35     return 0;
36 }
```

Notice the four lines of code (#include and three key API functions) that are used to allocate, initialize and control the GPIO pin which is connected to the red LED.

Line 4 – you must #include the <hwlib/hwlib.h> library when using the WNC SDK hardware API.

Line 22 – gpio_init() allocates the red GPIO_PIN_98 and assigns the resources to the myGPIO handle.

Line 23 – the pin resource pointed to by the myGPIO handle is configured as an output.

Line 29 and 31 – gpio_write() is used to send a high, then low, level to the pin specified by myGPIO.

3.5.2.2. What's wrong with this example?

The code runs, so what's wrong with this example?

Try running it a second time (without power-cycling the board) and you see that it fails. This is because we allocated the GPIO pin to our program but did not deallocate it before exiting the program. In fact, even when we fix this problem (discussed next) it can still cause problems when debugging buggy code – which is discussed in Section 3.5.3.

We can easily solve the allocation/deallocation problem by inserting the following line of code at Line 34 in our program:

```
gpio_deinit( &myGpio );
```

This line deallocates the GPIO pin resource once we are done using the pin, but before exiting the program. It's always good programming practice to release any resources – especially shared hardware resources – before exiting your programs.

But wait, there's another problem with this code example… we never checked for a valid return ("ret") from the gpio_init() function. Therefore, this example doesn't even recognize the case where the GPIO pin might have been allocated to a different program. These types of resource collisions can be very difficult to find. It's always good practice to verify function return values and handle the problem 'elegantly' when a resource conflict occurs.

You can find an example that addresses both issues in the User Guide code examples. Look for the following Chapter 3 project:

```
Chapter_03/api_red_led_with_deinit
```

3.5.3. API GPIO Init Problems When Debugging

It's not uncommon to iterate between editing and debugging your code – whether writing in C, shell scripts, or Python. We invariably start out with bugs in our code and slowly fix them all until the program is running smoothly.

What happens when the program fails between initializing a GPIO pin (as in the preceding example) and using it? Since we don't finish executing the program, the GPIO pin doesn't get de-initialized. Thus, unless you're rebooting your SK2 between each iteration, the next call to initialize the same GPIO pin will fail. And, since your new program did not initialize the resource, you cannot even call the gpio_deinit() API function to de-initialize it.

Looking back to the file i/o version of the program, we can guess that the gpio_init() function is – among other things – exporting the pin to user space. Once exported, the gpio_init() function was not written such that it can re-export the pin.

You can simulate this problem by running the `fileio_red_led` program, which exports the red LED pin (but doesn't unexport it), before running the `api_red_led` program. The "api" version of the program fails because it cannot successfully initialize the pin with gpio_init().

```
> ./fileio_red_led
> ./api_red_led        <---  Fails to run
```

You can solve this problem by "unexporting" the pin before calling the `api_red_led` program. Doing this allows the gpio_init() function to successfully allocate the pin. The programs `fileio_export` and `fileio_unexport` allow you to easily export or unexport the GPIO pins, which can be handy when testing and debugging your programs.

```
> ./fileio_red_led
> ./fileio_unexport
> ./api_red_led        <---  Successfully runs
```

Besides debugging, how might this affect you?

1. When your program is running in production, it's not likely that you will run into this problem – unless your program fails and it's restarted without rebooting the board.
2. Your program can borrow from the `fileio_unexport` program, forcing the GPIO pin to be made available by writing to the GPIO pin's unexport virtual file.
3. Forcing a pin to be available (i.e. unexported) in a program may solve 'your' problem… but this may create problems with 'other' programs. Unlike non-Linux embedded systems, Linux allows us to execute many programs simultaneously. This can be a convenient way to build (or add) functionality into your embedded system. But "stealing" resources (as described in #2) will cause one program to run while crashing another.
4. If two programs MUST share a hardware resource, such as a pin, you must handle this in advance. There are many examples discussed across the Internet – and in Universities – for handling resource conflicts. We've even seen cases where one program thread owns the resource and handles requests from many other program threads.

In the end, you will need to plan how – and where – to allocate the resources available to your system.

3.6. Writing C Programs for Linux

Writing C programs for the Linux platform often involves more than just plain C code. While this is the case when writing code for any operating system, Linux provides a rich assortment of capabilities that your programs can leverage.

For the most part, covering the use of Linux services falls outside the goals of this user guide, but we wanted to introduce two main concepts that may be useful when writing programs for the Linux platform: Multi-threading and Events. The latter topic – Events – includes a few example programs since Events are often triggered by hardware peripherals, such as GPIO inputs.

3.6.1. Multi-threading

The terms multi-processing, multi-tasking, and multi-threading are often discussed when using an operating system – especially a high-level O/S such as Linux. There are subtle differences between their definitions, but they all describe multiple things (e.g. programs) executing in parallel. Application programs, as well as users themselves, can start multiple program threads running at the same time by using the operating system's API and commands. Let us introduce a few topics that you may find useful during your code development.

3.6.1.1. Multiple Processors

If your system has multiple processors, it's easy to imagine running more than one program at a time.

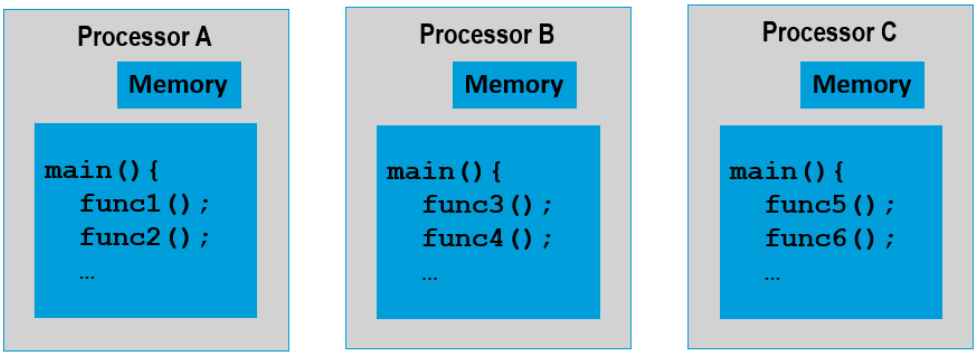


Figure 3.3 - Multi-processing

For example, if you have three processors, each with their own memory, you can execute three different programs concurrently as shown below:

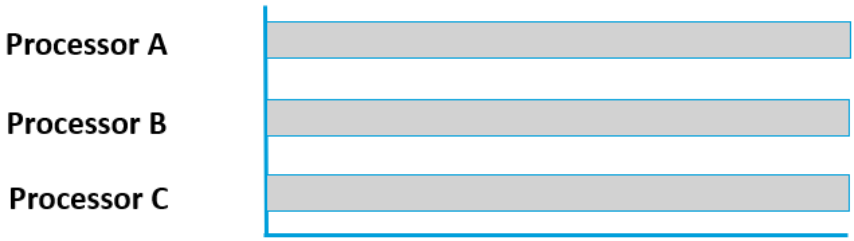


Figure 3.4 - Multi-processing Graph

This type of capability exists within the WNC M18Qx module. It has different processors which handle various tasks, such as LTE communications, or running the Linux platform. We are only allowed to program the single ARM Cortex-A7 apps processor, running Linux, with C/C++ or Python. While the system gains from having multiple processors, your Linux programs can only use the one.

3.6.1.2. Multiple Processes

When you only have one processor, as in Figure 3.5 below, your programs must share the resource. Linux enables this by letting you create multiple processes.

A "process" is defined by its memory and file descriptors. Returning to Figure 3.5, notice how each process has its own memory (which also includes any file descriptors that have been allocated). The memory management hardware in modern application processors, such as the ARM Cortex-A7, firewalls the memory for each process. In other words, the code in "Process A" cannot access the memory from "Process B" or "Process C" and vice-versa. This allows us to robustly run three different programs at the same time.

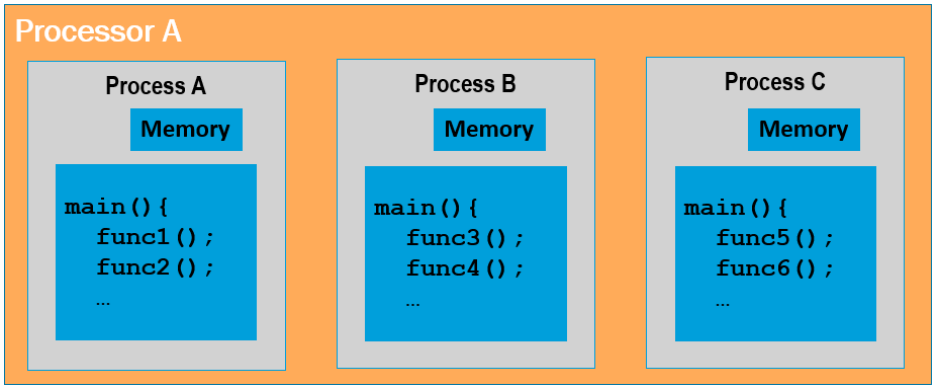


Figure 3.5 - Multiple Processes

Well, let's clarify that last statement. While three different programs can be configured to run simultaneioiusly, with only a single processor, only one program can run at a time. This can be seen in Figure 3.6 where the processes take turns using the processor. The two waiting to run are effectively paused (sometimes called "blocked") waiting for their turn – but at least their memory is protected from the running process.

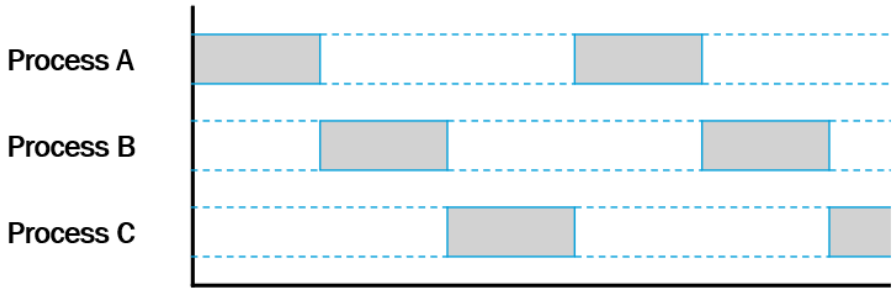


Figure 3.6 - Multiple Processes Graph

3.6.1.2.1. Why create multiple processes?

Why multiple processes? Because it's easier using different programs to solve different problems.

For example, think about the host computer you use. Whether it runs Windows or Linux, you likely have more than one program running at the same time. Think how much easier it is to build a robust spreadsheet program if it doesn't have to include all the features of a word processor. When you start each program, it runs in its own process, sharing your systems resources.

This may also simplify your programming for the SK2. If you have independent activities that need to operate at the same time, it may be easier to write two programs and set them both running when your SK2 boots up.

3.6.1.2.2. How to Create a New Process using Fork

When Linux boots up, it's executing a single process. When a second process is needed, this is usually done by issuing a Linux "fork" command. When a fork is executed, Linux makes a duplicate copy of the current process – including all its memory and file descriptors. At this point, the new, duplicate process can begin running a new program, changing anything or everything in the process without affecting the original process.

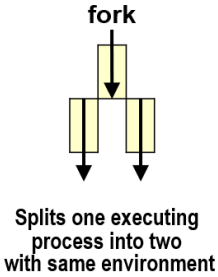


Figure 3.7 - Fork

Forking isn't restricted to C/C++ programming, it can also be done from the Linux command-line. For example, when you invoke a program from the terminal, such as:

```
./blink_led
```

you may have noticed that the terminal session is paused, waiting for the new program to complete. In other words, the *blink_led* program is running in the same process as the terminal. But, Linux actually let's us force the new program to run in its own process. This is done using the "&" symbol:

```
./blink_led &
```

Including the ampersand at the end of the command-line tells Linux to fork the current process and begin running the new program in the new process. In this case, you'll notice that the command-prompt returns immediately, even though the blink program may still be running.

You can implement the same action using fork() in your C/C++ programs – letting one program spawn many different processes. However you invoke fork, it's a handy way to run multiple programs at the same time.

One of the nice things about using Linux for your embedded system is the wide availability of example code. We pulled a `fork.c` example from *Beginning Linux Programming* (see reference at the end of this chapter) and tested it on the SK2. You can find a copy of this generic fork example in the User Guide's code examples: `sk2_users_guide/Chapter_03/fork/src/main.c`

3.6.1.2.3. Useful Commands for Managing Processes

There's a great deal of information available across the internet for learning about and managing Linux processes. We're only introducing a few of the most basic commands that may come in handy.

Command	Description
ps	Lists currently running user processes
ps -e ps ax	Lists all processes
top	Ranks processes in order of CPU usage
kill <*pid*>	Ends a running process • PID is returned by fork() • Get PID using "ps" or "top"
kill <signal> <pid>	Pass a specific signal to a process (Signals are discussed in Section 3.6.2.2)

3.6.1.3. Multi-threading

Linux also includes the ability to create multiple threads of execution within a single process. Since multiple program threads share a processes memory, it lowers the overhead of time and memory versus creating multiple processes. But this is done at the expense of robustness, since one errant thread could wipe out all of the shared memory in the process – in other words, no 'firewall' exists between multiple threads (in a single process) as it does between processes.

Here's a simple diagram showing Process A consisting of three threads of independent execution, while Process B only contains one. (By default, a process always has at least one thread of execution.)

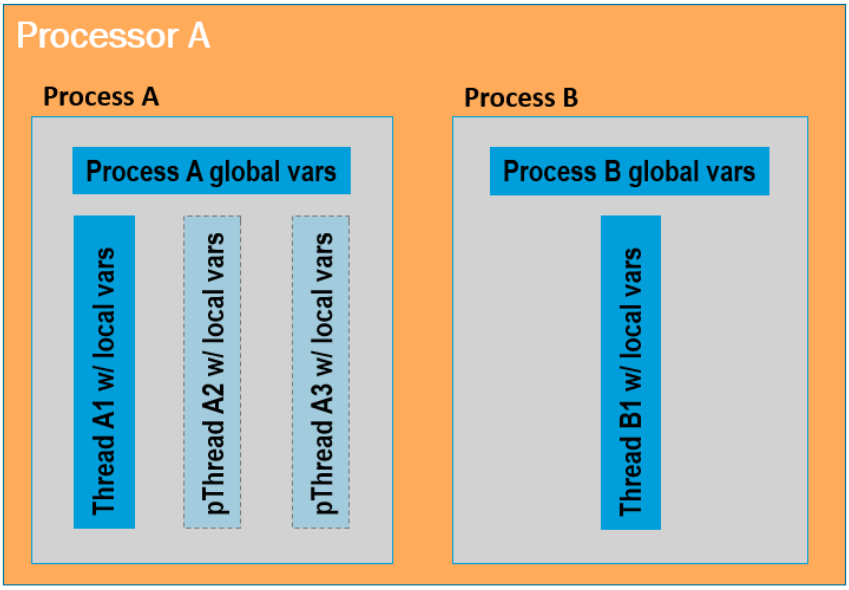


Figure 3.8 - Multiple Threads in a Process

You may have noticed the additional threads in Figure 3.8 are labeled "pThread". This comes from the name of the function (and library) used to create additional threads within a process. Linux generally includes the standard POSIX library (Portable Operating System Interface) which includes routines for creating, managing, and deleting new threads of execution. The `pthread_create()` function can be used to spawn new program threads within in a process.

From a scheduling perspective, Linux sees each thread as an independent unit to schedule when time-slicing between them. In words, the single thread in Process B may only get 25% of the processor time while those in Process A will likely get 25% each. That said, Linux provides multiple ways to tweak and override the scheduling and priority of threads – and that discussion falls outside of what we will address here.

3.6.1.4. Why Should I Use Multiple Threads?

Back in Section 3.6.1.2.1 we suggested that using multiple processes can be useful to simplify programming – creating smaller, simple programs for each independent task is often easier and more robust than creating a single, more complex program.

But, there's another big advantage to using multiple threads and/or processes. When one thread blocks (i.e. is paused) waiting for a resource, such as a serial port or new sensor data, other threads can use that time to execute their code. In fact, embedded systems make great use of this feature.

For example, the main program may spin in a while{} loop – or in a low-power mode – while waiting for Linux Events to announce that new data is available or that a timer has signaled its time to sample a sensor and post the data to the cloud. This leads us to the next section: Linux Events.

3.6.2. Event Handling

Traditional embedded systems are often built around handling various “events”; for example, when data becomes available, a user presses a button, or a specified time interval has expired. When these systems are built with Linux, there are even more events to choose from, such as signals from the operating system.

Unique to the SK2 are the hardware peripherals and their associated API provided by the WNC SDK. In this chapter we introduce the functions and procedures for handling hardware interrupt events generated by the GPIO pins.

Before we deal with the GPIO functions, though, let’s explore a few other common Linux events that are used throughout the user’s guide examples.

3.6.2.1. Sleep

Sleep is one of the many time-based events that can be generated by the Linux O/S. It’s also one of the easiest to use – so easy, in fact, that we’ve already used it in many code examples.

There are a number of “sleep” functions, the most obvious being *sleep()* – which tells the process calling the function to sleep (i.e. suspend execution) for a given number of seconds.

```
#include <unistd.h>

unsigned int sleep (unsigned int seconds);
```

If the *sleep()* function is interrupted by some other event, it returns the number of seconds not slept. Most users ignore the return value, but you can use it to recall the function in order to force the process to sleep for the specified time.

Two other similar functions are:

- `usleep()` – sleeps for a given number of microseconds
- `nanosleep()` – sleeps the process for a given number of nanoseconds

Besides increasing orders of precision, these two functions are slightly different from each other. While the *usleep()* function works the same as *sleep(),* the nanosleep function is found in the <time.h> library and requires a Linux *timespec* structure as its argument.

While our examples wait for seconds, we’ve chosen to use *usleep()* in most cases.

3.6.2.2. Signals

Signals (i.e. software interrupts) are events generated by Linux in response to an internal or external event. Signals occur asynchronously to our software programs – meaning that they could happen at any time, and won't necessarily happen at any specific point in our program code.

A signal can be *raised* (i.e. generated) by a variety of events. For example, when a user generates an interrupt character (e.g. Ctrl-C); an error condition occurs – such as when a process divides by zero; or when one process sends a signal to another.

Applications can be programmed to *catch* and respond to signals. This is done by passing an event handler function to Linux, essentially telling it what code we wish to run in response to a given event… if it occurs. The event handler function is often called a *callback* function because our program is telling Linux to "call back" to our program using a specific function name.

Here's a listing of Signals that Linux supports:

Signal	Description	Signal	Description
SIGABORT	*Process abort	**SIGTERM**	Termination
SIGALRM	Alarm clock	**SIGUSR1**	User-defined signal 1
SIGFPE	*Floating-point exception	**SIGUSR2**	User-defined signal 2
SIGHUP	Hangup	**SIGCHLD**	Child process has stopped or exited
SIGILL	*Illegal instruction	**SIGCONT**	Continue executing, if stopped.
SIGINT	Terminal interrupt	**SIGSTOP**	Stop executing (Can't be caught or ignored)
SIGKILL	Kill (can't be caught or ignored)	**SIGTSTP**	Terminal stop signal
SIGPIPE	Write on a pipe with no reader	**SIGTTIN**	Background process trying to read
SIGQUIT	Terminal quit	**SIGTTOU**	Background process trying to write
SIGSEGV	*Invalid memory segment access		

We'll explore one of the most common signals, SIGINT, the terminal interrupt that's generated when we press the *Control-C* keys while running a program from our terminal command-line.

3.6.2.2.1. Control-C (SIGINT)

SIGINT is a handy, common signal to use when debugging your programs. It makes it easy for you to send a signal to your program from the command line. One simple example, shown here, is to catch this signal and terminate your program.

Listing 3.5: Chapter_03/sigint/src/main.c

```
1  #include <signal.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  void my_handler(sig_t s){
6             printf("\nCaught signal %d\n",s);
7             exit(1);
8
9  }
10
11 int main(int argc,char** argv)
12 {
13    signal (SIGINT,my_handler);
14
15    while(1);
16    return 0;
17
18 }
```

Except for one line of code in main() that deals with the SIGINT signal, this program will loop forever due to the while(1) loop on line 15.

To handle the SIGINT signal event, two lines of code – and the my_handler() function – were added to the program. Looking at main(), let's examine the code required:

Line 1 When using the Linux to handle the SIGINT event generated by Control-C, you must include the <signal.h> header file.

Line 13 The signal() function tells Linux what to do when the SIGINT event occurs. More specifically, this function registers the callback function "my_handler" to the SIGINT signal event.

- Whenever SIGINT occurs while this program is running, Linux will pre-empt your program and run the registered call-back function.
- When my_handler() is called by Linux (after Ctrl-C is pressed), it will print "Caught signal" to the terminal and exit the function.
- If you don't want the program to exit after a signal occurs, don't call exit() inside the signal handler. (We provide an example of this in the GPIO pin Interrupt section.)

Note: Even though our example's signal handler uses printf(), it is not recommended to use use printf() in signal handlers. That said, printf() makes it easy to visualize what is happening during program execution.

In fact, you can refer to the *signal* documentation (or the *Beginning Linux Programming* book listed at the end of the chapter) for a list of functions that are safe to use within a signal handler.

Hint: One recommended technique, if you need to use an unsafe function, is to set a flag during the signal handler and then print the message – or call the appropriateunsafec function – from inside the main program.

3.6.2.2.2. Pause

The *pause()* function is a useful debugging function that puts a process to sleep while waiting for any signal. Upon receiving a signal, it wakes and responds as if the routine as if it had been awake, running the signal handler registered by the program.

We provide two examples to demonstrate this function. The sigint2 does not use pause() while the sigint2_with_pause does. Both functions provide another example for handling the SIGINT.

Listing 3.6: Chapter_03/sigint2/src/main.c

```
// Example taken from "Beginning Linux Programming" 4th Edition
// by Neil Matthew and Richard Stones; Wiley Publishing © 2008
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("\nOUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}

int main()
{
   (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

When running the program, "Hello World!" prints out repeatedly until the user has entered Ctrl-C twice. Notice that the signal handler ouch() redefines the signal's handler, changing it back to the signal's default (SIG_DFL) action for Linux (causing the program to exit). Here's what it looks like in the terminal:

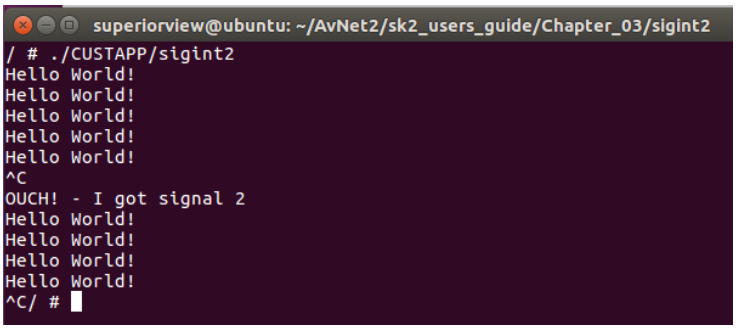


Figure 3.9 - Running sigint2

Modifying the program slightly, we replaced the sleep() command with pause().

Listing 3.7: Chapter_03/sigint2_with_pause/src/main.c

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("\nOUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}

int main()
{
   (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        pause();
    }
}
```

Notice that after swapping out sleep() with pause(), “Hello World!” only prints out once before and after the first Cntrl-C:

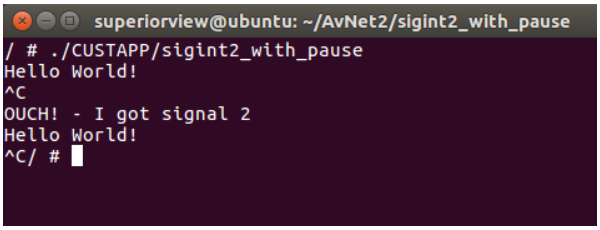


Figure 3.10 - Running sigint2_with_pause

3.6.2.2.3. Sending Signals from the Terminal

In the following figure, we are once again running the program `sigint2_with_pause`, but this time we added "**&**" to run it in a separate process. Notice that after starting the program in Figure 3.11, our terminal returns to allow us to enter new commands – where we then executed the ***ps*** command to see our program running in a new process (with pid = 3349).

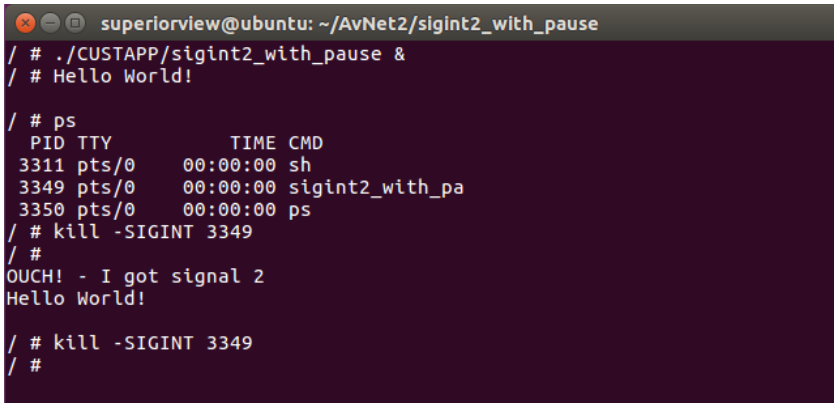


Figure 3.11 - Running sigint2_with_pause in separate process

But now that the program is running in a separate process, it doesn't receive our Control-C commands anymore. How can we send it a signal?

As was mentioned back in Section 3.6.1.2.3, the Linux ***kill*** command can be used to send a signal to a running process. If we just typed:

```
kill 3349
```

our program would simply terminate. But rather, we used ***kill*** to send the SIGINT (i.e. Control-C) signal

```
kill -SIGNINT 3349
```

to the process using ***kill***. Notice how sending it twice caused the same results as was seen in Section 3.6.2.2.2.

3.6.2.2.4. Upgrading to sigaction()

While the *signal*() functionality is long-standing and well adopted by Linux and Unix, *sigaction*() has become the preferred method of handling signals due to its robust flexibility. It takes a little more code to implement, though, which is why we consider this an 'upgrade' to the previous example. Additionally, we followed the earlier advice for moving the *printf*() functions out of the signal handlers.

Listing 3.8: sk2_users_guide/Chapter_03/sigaction/src/main.c

```
#include <signal.h>                     // Needed for sigaction()
#include <stdio.h>                      // Needed for printf()
#include <unistd.h>                     // Needed for sleep()
#include <stdlib.h>                     // Needed for exit()

int flag    = 0;
int lastSig = 0;

void ouch(int sig) {                    // First pass SIGINT handler
    flag    = 1;
    lastSig = sig;
}

void quit(int sig) {                    // Second pass SIGINT handler
    flag    = 3;
}

int main() {
    struct sigaction act;               // 'action' struct for signal

    act.sa_handler = ouch;              // Callback function is 'ouch'
    sigemptyset(&act.sa_mask);          // Don't block any signals
    act.sa_flags = 0;                   // No signal action modifiers

    sigaction(SIGINT, &act, 0);         // Set action for SIGINT signal

    while(1) {                          // Respond to value of 'flag'
        switch(flag) {
            case 0:
            case 2:
                printf("Hello World!\n");
                sleep(1);
                break;
            case 1:
                printf("\nOUCH! - I got signal %d\n", lastSig);
                act.sa_handler = quit;      //Could have set to SIG_DFL
                sigaction(SIGINT, &act, 0); //Set new action for SIGINT
                flag = 2;
                break;
            case 3:
                printf("\nOUCH again. This time it's Goodbye!\n");
                exit(0);
                break;
            default:
                printf("Don't
                exit(1);
        }
    }
}
```

```
superiorview@ubuntu: ~/AvNet2/sigaction
/ # ./CUSTAPP/sigaction
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH again. This time it's Goodbye!
/ #
```

Figure 3.12 - Running sigaction

3.6.2.3. GPIO pin Interrupt

The Linux GPIO pin driver provides an interrupt event whenever an input pin is changed. Your program needs to initialize and configure the GPIO input pin, as well as register the interrupt ("irq") callback function before it can catch the event.

The following code example enables interrupts from the SK2's User Button. Notice the three additional items required to convert an input GPIO pin to an interrupt triggered GPIO pin.

1. Callback function
2. Call IRQ request to initialize the interrupt event
3. Wait for the interrupt request to complete!
4. Free the IRQ resource when it's not needed anymore

Listing 3.9: sk2_users_guide/src/api_button_interrupt

```
#include <stdint.h>
#include <stdio.h>                                  // Needed for printf()
#include <stdlib.h>                                 // Needed for exit()
#include <unistd.h>                                 // Needed for pause()
#include <hwlib/hwlib.h>                            // Needed for WNCSDK GPIO API

#define USER_BUTTON     GPIO_PIN_98

int  ret = 0;                                       // Return value
gpio_handle_t myBtn = 0;                            // Handle for button's GPIO pin
gpio_handle_t myLed = 0;

//*****  GPIO callback routine  ***************************************
int myBtn_irq_callback(gpio_pin_t pin_name, gpio_irq_trig_t direction) {
    if (pin_name == USER_BUTTON)
    {
        printf("Button interrupt received\n");
    }
}

//*****  Main  ********************************************************
int main(void)
{
    printf("\nStarting GPIO Callback Example!\n");
    printf("Please wait while we configure your device...\n");

    // Allocate and configure GPIO pin for User Pushbutton switch input
    ret = gpio_init(USER_BUTTON, &myBtn);
    gpio_dir(myBtn, GPIO_DIR_INPUT);

    // Register callback function for GPIO pin interrupt to trigger on up/down
    gpio_irq_request(myBtn, GPIO_IRQ_TRIG_BOTH, myBtn_irq_callback);

    // Now waiting for interrupt to occur
    printf("Device configuration complete.\n");
    printf("Press and release User Button to trigger interrupt.\n");

    pause();                                        // Wait for signal from user button

    // Release resources and exit
    printf("Releasing GPIO resources...\n");
    gpio_irq_free(myBtn);                           // Free the IRQ push button callback
    gpio_deinit( &myBtn );                          // Release the button's GPIO resource

    return 0;
}
```

Warning

Generating an interrupt before the IRQ request completes triggers a user signal that aborts the program.

Prove this to yourself by pressing the User Button before getting the "Device config complete" message".

The user guide examples archive (or git) contains an additional example that turns on the blue LED when the button is pressed. Look for: `sk2_users_guide/Chapter_03/api_button_irq_led`

3.7. Reuseable myGPIO Example

As the final example for the chapter, we've created a reusable GPIO example. This example contains a generic initialization function that can allocate all the requested GPIO resources. Using the routine requires a few instructions:

- myGpio.c
  - Modify the myGpio structure in myGpio.c for the GPIO pins your application will use.
  - Create or modify the sample myGpio_irq_callback function if you plan to use GPIO interrupts.
- main.c (or where ever you wish to initialize and use GPIO)
  - #include "myGpio.h"
  - Call the myGpio_init() function
  - Call the myGpio_close() function when finished with the GPIO resources

If this doesn't meet everyone's needs, we hope it provides a good example that users can work from.

Listing 3.10: sk2_users_guide/myGpio/src/main.c

```
#include <stdint.h>
#include <signal.h>                                             // Needed for signal()/sigaction()
#include <stdlib.h>                                             // Needed for exit()
#include <stdio.h>                                              // Needed for printf()
#include <unistd.h>                                             // Needed for usleep() and pause()
#include <hwlib/hwlib.h>                                        // Needed for WNCSDK GPIO API
#include "myGpio.h"

#define SECOND      1000000
#define NUM_LEDS    3

int doExit = 1;

// ****  SIGINT Handler for Control-C ****************************************
void my_handler(sig_t s)
{
    doExit = 0;
    printf("\nCaught signal (%d). Quitting the program...\n",s);
}

// ****  main  ***************************************************************
int main(void)
{
    int i = 0;                                                  // Loop counter
    int r = 0;                                                  // Return value

    printf("\nWelcome to the myGpio example!\n");

    signal (SIGINT, my_handler);                                // Register SIGINT signal handler

    r = myGpio_init();                                          // Allocate and initialize the GPIO resources
    if (r)
        printf("Not all requested GPIO resources were allocated.\n");

    gpio_read(myGpio[3].hndl, &myGpio[3].val);                  // Read and output the value of the User Button
    printf("Checking on the button's value. It is %d.\n", myGpio[3].val);

    printf("\nLED's will blink in color sequence until User Button is pushed.\n");
    printf("Alternatively, you can also press Cntrl-C to quit the program.\n");
    while(doExit)
    {
        // Blink all LEDs in sequence
        for ( i = 0; i < NUM_LEDS; i++ )
        {
            gpio_write( myGpio[i].hndl, GPIO_LEVEL_HIGH );      // Turn on LED
            usleep( 1 * SECOND );                               // Wait 1 second

            gpio_write( myGpio[i].hndl, GPIO_LEVEL_LOW );       // Turn off LED
            usleep( 1 * SECOND );                               // Wait 1 second
        }
    }

    myGpio_close();                                             // Release the GPIO resources

    return 0;
}
```

Listing 3.11: sk2_users_guide/myGpio/src/myGpio.c

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0----+----1----+----2----+----3----+---
#include <stdio.h>
#include <stdlib.h>
#include <hwlib/hwlib.h>
#include "myGpio.h"

extern int doExit;

/*****************************************************************************
* myGpio_t
*
* This structure indicates how the myGpio_init() routine should allocate and
* configure the WNC M18Qx GPIO Linux drivers.
*
* hndl: Set to 0; will be initialized in myGpio_init()
* abr:   Used to provide user terminal feedback during myGpio_init()
* nbr:   Number of the GPIO pin being configured
* pin:   WNC SDK enumeration for specified pin
* dir:   Should myGpio_init() configure pin as input or output
* trig:  If using pin as input interrupt, which trigger value enumeration
*        should be configured
* cback: If using pin as input interrupt, what function should be used
*        for callback
* val:   Can be used to hold the pin value when reading pin
* ret:   Return from gpio_init() function when called by myGpio_init();
*        it should be initialized to -1
* irq:   Return from gpio_irq_request() when called by myGpio_init();
*        it should be initialized to -1
*
*****************************************************************************/
myGpio_t myGpio[] = {

//  hndl  abr   nbr   pin            dir               trig                    cback               val  ret  irq  comment
//  ----  ---   ---   ------------   ---------------   ---------------------   -----------------   ---  ---  ---  --------------------
    0,    "R",  92,   GPIO_PIN_92,   GPIO_DIR_OUTPUT,  -1,                     0,                  0,   -1,  -1,  // Red LED
    0,    "G",  101,  GPIO_PIN_101,  GPIO_DIR_OUTPUT,  -1,                     0,                  0,   -1,  -1,  // Green LED
    0,    "B",  102,  GPIO_PIN_102,  GPIO_DIR_OUTPUT,  -1,                     0,                  0,   -1,  -1,  // Blue LED
    0,    "S",  98,   GPIO_PIN_98,   GPIO_DIR_INPUT,   GPIO_IRQ_TRIG_FALLING,  myGpio_irq_callback, 0,   -1,  -1   // User Button Switch
};

#define _MAX_GPIO       (sizeof(gpio)/sizeof(myGpio_t))
#define _MAX_GPIO_PINS  (sizeof(myGpio)/sizeof(myGpio_t))
const int _max_gpio_pins = _MAX_GPIO_PINS;


// ****  Simple example of a GPIO interrupt callback routine  ****************
//
int myGpio_irq_callback(gpio_pin_t pin_name, gpio_irq_trig_t direction)
{
    if (pin_name != myGpio[3].pin)
        return 0;

    doExit = 0;
    printf("\nInterrupt occurred on %d! Now exiting...\n\n", myGpio[3].nbr);
}


// **** myGpio_init() ********************************************************
// Initialize all the binary i/o pins in the system
int myGpio_init(void)
{
    int ret;
    int i     = 0;
    int error = 0;

    // Loop thru myGpio[] allocating required pins
    printf("GPIO allocation (%d pins):\n", _max_gpio_pins);

    for ( i=0; i < _max_gpio_pins; i++ )
    {
        myGpio[i].ret = gpio_init( myGpio[i].pin, &myGpio[i].hndl );            // Call GPIO init API function
        printf("  %s=%d\n", myGpio[i].abr, myGpio[i].ret );

        if (myGpio[i].ret !=0)
        {
            printf("      --> ERROR: GPIO resource unavailable\n");
            error = 1;
        }
    }

    // Loop thru pins setting direction and irq
    printf("\nBeginning configuration of GPIO pins:\n");

    for ( i=0; i < _max_gpio_pins; i++ )
    {
        if (myGpio[i].ret == 0)
        {
            gpio_dir( myGpio[i].hndl, myGpio[i].dir );                          // We're not checking gpio_dir() return value

            if ((myGpio[i].dir = GPIO_DIR_INPUT) && (myGpio[i].trig != -1))
            {
                myGpio[i].irq = gpio_irq_request( myGpio[i].hndl, myGpio[i].trig, myGpio[i].cback );

                if (myGpio[i].irq == 0)
                {
                    printf("IRQ %d enabled\n", myGpio[i].nbr);
                } else
                {
                    printf("ERROR: GPIO %d IRQ unavailable\n\n", myGpio[i].nbr);
                    error = 1;
                }
            }
        } else
        {
            printf("--> Pin configuration skipped for %s (%d) since it failed allocation\n", myGpio[i].abr, myGpio[i].nbr);
        }
    }

    printf("GPIO configuration complete\n\n");
    return error;
}
```

Listing – sk2_users_guide/myGpio/src/myGpio.c (continued)

```
112
113 // **** myGpio_close() ********************************************************
114 // Release the resources allocated by myGpio_init()
115 // Skip any resources that failed during allocation/configuration
116 void myGpio_close(void)
117 {
118     int i = 0;
119
120     printf("Releasing GPIO resources\n\n");
121
122     for ( i=0; i < _max_gpio_pins; i++ )
123     {
124         if (myGpio[i].irq == 0)
125         {
126             gpio_irq_free( myGpio[i].hndl );
127         }
128
129         if (myGpio[i].ret == 0)
130         {
131             gpio_deinit( &myGpio[i].hndl);
132         }
133     }
134 }
```

Listing 3.12: sk2_users_guide/myGpio/src/myGpio.h

```
1
2  #ifndef __MYGPIO_H__
3  #define __MYGPIO_H__
4
5  // **** myGpio_t ****************************************************************
6  typedef struct _myGpio {
7      gpio_handle_t           hndl;       // Set to 0; will be initialized in myGpio_init()
8      char                    abr[2];     // Used to provide user terminal feedback during myGpio_init()
9      int                     nbr;        // Number of the GPIO pin being configured
10     gpio_pin_t              pin;        // WNC SDK enumeration for specified pin
11     gpio_direction_t        dir;        // Should myGpio_init() configure pin as input or output
12     gpio_irq_trig_t         trig;       // If using pin as input interrupt, which trigger value enumeration should be configured
13     gpio_irq_callback_fn_t  cback;      // If using pin as input interrupt, what function should be used for callback
14     gpio_level_t            val;        // Can be used to hold the pin value when reading pin
15     int                     ret;        // Return from gpio_init() function when called by myGpio_init(); it should be initialized to -1
16     int                     irq;        // Return from gpio_irq_request() when called by myGpio_init(); it should be initialized to -1
17 } myGpio_t;
18
19 #ifdef __cplusplus
20 extern "C" {
21 #endif
22
23 extern myGpio_t myGpio[];
24 extern const int _max_gpio_pins;
25
26 int myGpio_init(void);
27 void myGpio_close(void);
28 int myGpio_irq_callback(gpio_pin_t, gpio_irq_trig_t);
29
30 #ifdef __cplusplus
31 }
32 #endif
33
34 #include <sys/types.h>
35 #include <stdint.h>
36 #include <nettle/nettle-stdint.h>
37 #include <hwlib/hwlib.h>
38
39 #endif // __MYGPIO_H__
```

3.8. Where to Go for More Information

Recommended materials for learning more about C/C++ programming for Linux and the SK2.

- Avnet M18Qx Perpherial IoT Guide (DOCX)
- Matthew, Neil, and Richard Stones. *Beginning Linux Programming (4th Edition)*. Wiley, 2011.
- Robert Love. *Linux System Programming: Talking Directly to the Kernel and C Library (2nd Edition)*. O'Reilly Media, 2013
- Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook (1st Edition)*. No Starch Press, 2010.

4. Installing and Using Python

Python 2.7 has been ported to the SK2 to aid in rapid-prototyping – or full-scale development – of applications for the SK2 (AT&T IoT Starter Kit 2nd Generation). After upgrading your SK2 firmware, you will be able to run Python scripts from the Linux command-line. Additionally, the SK2 Python firmware includes a cloud-based IDE for editing, running, and managing python scripts and applications.

This chapter focuses on installing and using the SK2's Python platform. Look back to Chapter 2 for details about the Linux environment and writing shell-scripts. Or Chapter 3 for details about C/C++ environment.

Also note that

Warning!

Installing the Python SK2 Linux firmware will replace the filesystem on your SK2. Please follow the directions carefully – including Section 4.1.1 *"Backup /CUSTAPP"* – to make sure any files you have added to your kit are backed up before installing the Python firmware.

Prerequisite Knowledge and Tools

This guide assumes that you are generally familiar with the Python language. Language constructs are not explained or taught in this book, although it covers a few topics that are part of the Python code examples for the SK2. Please refer to the Additional Resources section at the end of the chapter for more information about Python coding.

Topics

4.1. Installing Python

Installing Python onto your SK2 involves updating the entire firmware image for the development board. Python, and the cloud-based IDE have been baked into the Linux image. While this increases the steps required to install Python – versus installing only a simple Python executable – the result is a complete platform that has been pre-verified to run on the SK2 board.

The SK2 Python firmware downloads as a large ZIP file. After downloading the ZIP file you will unzip the firmware image and run a series of ADB commands in order to update the various elements of the Linux firmware on your kit. When complete, your SK2 will include the Python executable and IDE.

WARNING - *The following Python installation will replace your current SK2 Linux firmware. Any files that have been created or written to the /CUSTAPP directory on your kit will be erased during installation of the new Linux firmware. Please backup any files that you have added to the kit before installing the new firmware – backup instructions are found in the next section.*

4.1.1. Backup /CUSTAPP

The following sections step you through the procedure to update your SK2 with the new Python firmware. These steps entirely replace your Linux firmware – including the /CUSTAPP directory where you have likely created, modified, or uploaded files. Follow the steps in this section to backup – and then verify – that your files have been saved from the SK2 before continuing with installation in the next section.

1. **Open a command-line shell on your computer in your ADB folder and verify that you can connect to your board using "adb devices".**

   This was covered in detail in chapter 2.

2. **Start a remote session on your SK2 using "adb shell".**

   You can start a remote shell session running on your SK2 using the *adb shell* command.

   ```
   adb shell or ./adb shell
   ```

3. **Test if Python is already installed.**

   Python should not be installed, but by trying this step now, we can tell later if our installation was successful.

   ```
   python --version
   ```

   If Python was installed, it would return the version of Python currently available.

4. **Change to the /CUSTAPP directory.**

   ```
   cd /CUSTAPP
   ```

5. **List the read/write directory of the SK2 filesystem.**

   ```
   ls -ls
   ```

   Make a note of any files or directories that you've added to /CUSTAPP.

6. **Backup any files that need to be saved from the SK2.**

   You can individually pull files from the SK2 using the "adb pull" command. Although, the easiest solution may be to TAR (i.e. zip) the files and then pull them all with one command. This is the method we'll demonstrate.

   a) **TAR the files in /CUSTAPP.**

      Use the TAR (tape archive) command to create a compressed, archive of all the files within /CUSTAPP.

      ```
      tar -czvf myCustapp.tar.gz .
      ```

      tar: tape archive command
      -c: Create and archive
      -z: Compress files when added to archive
      -v: Verbose - display progress in the terminal while creating the archive
      -f: Specify the filename of the archive

      You can use any name you want for your archive, although it should end with ".tar.gz". We chose the name "`myCustapp.tar.gz`".

      Finally, the "." at the end of the command specifies all the files in the current directory. The TAR command will recurse any subdirectories and include their files.

Hint:	If you didn't want to include a specific directory in your TAR files – for example, let's say we had a directory named "temp", it could be excluded by adding the following to our example: `tar exclude='temp' -czvf myCustapp.tar.gz .`

   b) **List the /CUSTAPP directory again.**

      The listing should now include your TAR file, for example: `myCustapp.tar.gz`
      If it doesn't, debug the problem with your TAR command and keep trying until you succeed.

   c) **Exit the SK2 remote session.**

      ```
      exit
      ```

      Your command-line terminal should be back in your "adb" folder on your host computer.

   d) **Pull the TAR file from the SK2.**

      ```
      adb pull /CUSTAPP/myCustapp.tar.gz
      ```

      Make sure you include /CUSTAPP in your file's path.

7. **Verify saved files.**

   Open the TAR file on your host computer and verify it contains all the files you wanted to save from step #5.

Hint:	Many utilities allow you to view and extract TAR files. Popular utilities include: WinZip, WinRAR, and 7-zip.

   Make sure that you have saved all the files from your /CUSTAPP directory before proceding with the next section.

4.1.2. Download and Unzip SK2 Python Firmware Image

The SK2 Python firmware image is provided in a large ZIP file. You need to download the ZIP file and extract it to a folder on your host computer.

8. **Download the SK2 Python Image ZIP file.**

   M18Q2_v12.09.182151_APSS_OE_v01.07.183121.zip

9. **Unzip this file onto your computer.**

   After unzipping this file, your computer should contain a new folder named:

   M18Q2_v12.09.182151_APSS_OE_v01.07.183121

   View the files in the archive:

Files we will use for Python: • Blue files support ADB and Fasboot installation • Remaining files make up the Python firmware image	• adb.exe • AdbWinApi.dll • AdbWinUsbApi.dll • fastboot.exe • firmware.ubi.4k • fw.bin.full.4k • mdm9607-boot.img.4k • mdm9607-datafs.ubi.4k • mdm9607-sysfs.ubi.4k
Files that are not needed from the Python image ZIP file:	• README - Usage of WNC_Dloader_SB3.0_Support_MDM9x07.txt • appsboot.mbn • cmd.bat • rpm.mbn • sbl1.mbn • tz.mbn • tzbsp_no_xpu.mbn • WNC_Dloader_SB3.0_Support_MDM9x07.exe • ENPRG9x07.mbn • factory.yaffs2.2k.ESMT • factory.yaffs2.2k.ETRON • factory.yaffs2.4k • firmware.ubi.2k • fw.bin.full.2k • M18_wnccm_fastboot_2k.txt • M18_wnccm_fastboot_2k_ESMT.txt • M18_wnccm_fastboot_4k.txt • mdm9607-boot.img.2k • M18_download_4k.bat • mdm9607-datafs.ubi.2k • mdm9607-sysfs.ubi.2k • NPRG9x07.mbn • partition.mbn.2k • partition.mbn.4k

Note that you do not need to delete any of the “unnecessary” files listed above. We just wanted to state that these files would not be needed in case you wondered why they weren’t used in the following procedure.

4.1.3. Install SK2 Python Image Files

To install the new firmware image, you will need to switch to the new directory created by unzipping the SK2 Python Image ZIP – that is, the directory containing the files we just examined. Along with ADB, the following procedure makes use of the "fastboot" Android executable to write the image files to their appropriate place in the SK2's filesystem.

10. **Open a command-line shell on your computer, if it isn't already running.**

    Change to the directory where you extracted the SK2 Python firmware image files. For example:

    ```
    C:\M18Q2_v12.09.182151_APSS_OE_v01.07.183121
    ```

11. **Verify that you can connect via ADB using the ADB in the Python image directory.**

    ```
    adb devices
    ```

    You will likely see a notice stating that the ADB daemon is not running and that it has been started. Since we are running a new version of ADB from the Python image directory, ADB will stop the currently running server and restart it using the version of ADB in our new directory.

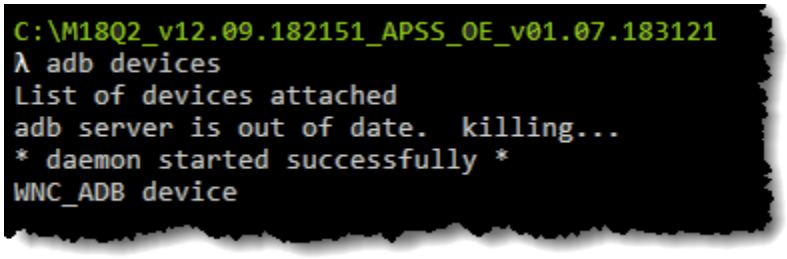


Figure 4.1 - Restarting ADB with "adb devices"

12. **Execute the following series of commands to write (i.e. 'flash') the Python image files to your SK2.**

    The following steps reboot the device into bootloader mode and then use 'fastboot' to write the images to their respective locations on the SK2.

    ```
    adb reboot bootloader
    fastboot flash boot_b mdm9607-boot.img.4k
    fastboot flash system_b mdm9607-sysfs.ubi.4k
    fastboot flash boot_a mdm9607-boot.img.4k
    fastboot flash system_a mdm9607-sysfs.ubi.4k
    fastboot flash data mdm9607-datafs.ubi.4k
    fastboot flash firmware_a firmware.ubi.4k
    fastboot flash firmware_b firmware.ubi.4k
    fastboot reboot
    ```

    Copy/Paste the commands from this page.

    Or, you can copy/paste them from the git file:

    Chapter_04\py_install_cmds.txt

Hint: View a successful execution of these commands on the next page.

13. **When complete, open an ADB shell and check what version of Python is now installed.**

    ```
    adb shell
    python --version
    ```

    Not only will your python version command return a version – rather than an error – you can also see quite a few new files and directories by listing the /CUSTAPP directory.

A successful execution of the Python installation looks like →

It only took a few minutes to complete the entire sequence

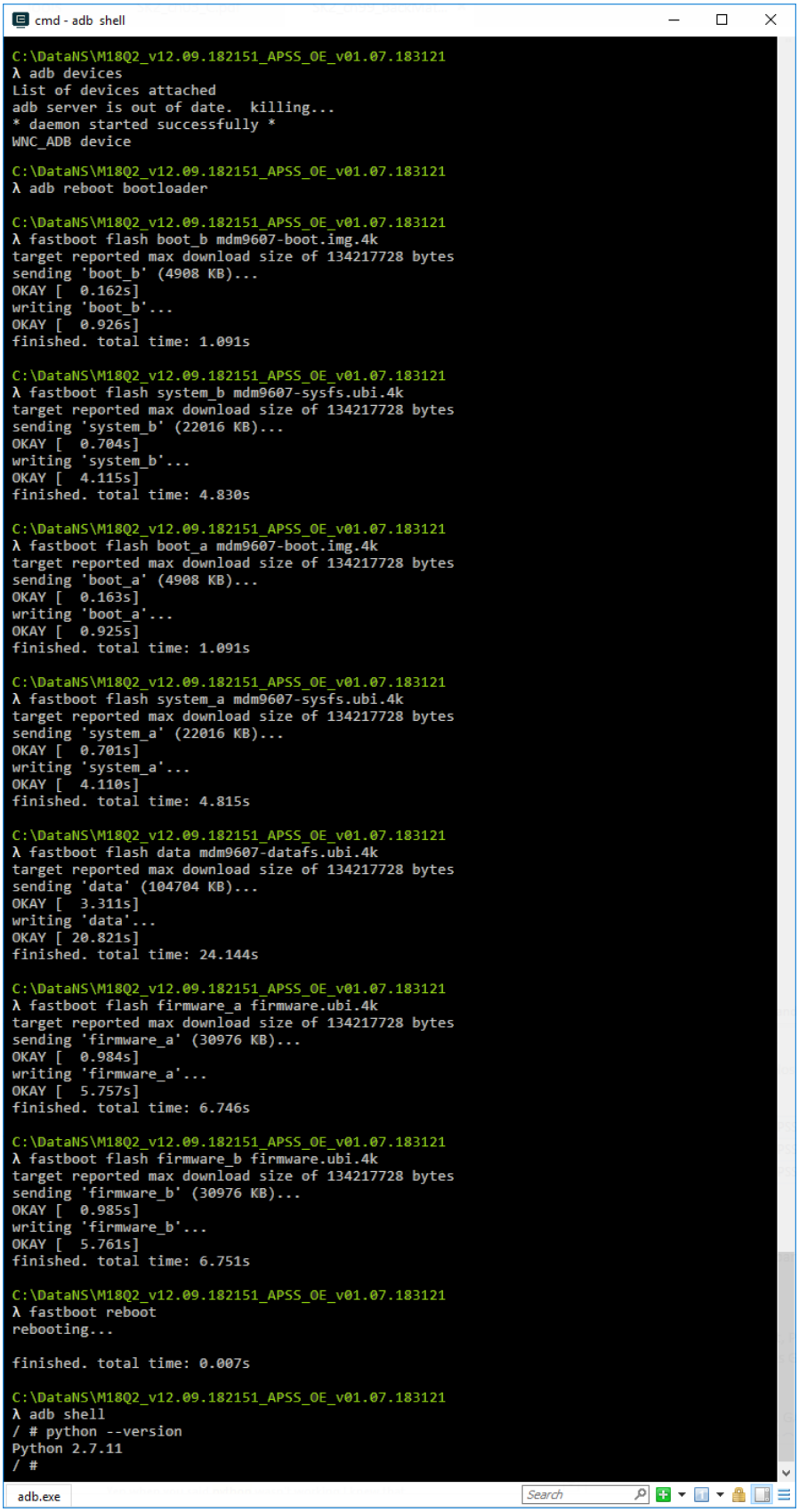


Figure 4.2 - Successful Flashing of Python Firmware

14. Exit the ADB remote session.

```
exit
```

15. Examine your Network settings

```
ipconfig
```

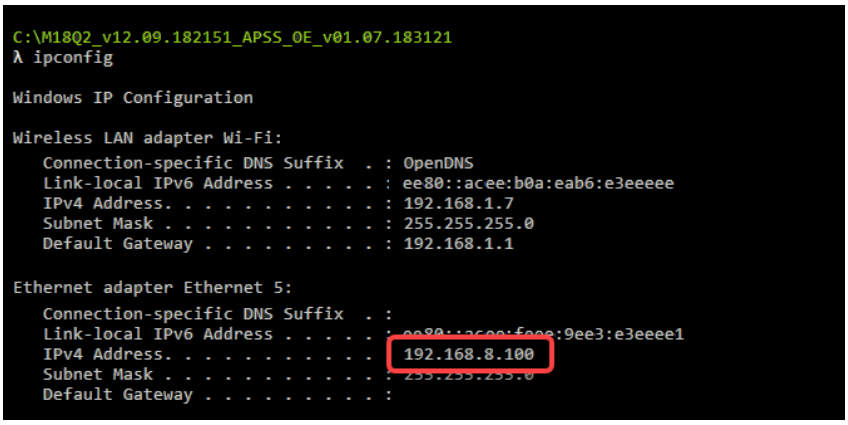


Figure 4.3 - ipconfig

Notice that along with our wireless LAN adapter, we now have an additional Ethernet connection with an IPv4 address of `192.168.8.100`. (This is important for the next section.)

16. Change back to your original ADB folder.

Leaving the Python image installation folder, return to your normal ADB folder. For example:

```
cd C:\adb
```

17. Run the 'adb devices' command to restart the original ADB server.

```
adb devices
```

Figure 4.4 - Restarting Original ADB Server

18. Open an ADB shell and try verifying Python again.

```
adb shell
# python --version
```

Figure 4.5 - Verifying Python

4.2. Connect to Python IDE

After the Python firmware has been installed on your SK2, you should be able to run Python scripts from the SK2 command-line (from within an ADB remote session). Additionally, you can open and use the Python IDE that was installed to your SK2 – which is what we will experiment with first.

4.2.1. Overview

1. **Open an Internet browser (e.g. Chrome) and connect to 192.168.8.1.**

   From an Internet browser, connect to the following address:

   ```
   192.168.8.1
   ```

   Upon a successful connection, you should see the Python IDE overview.

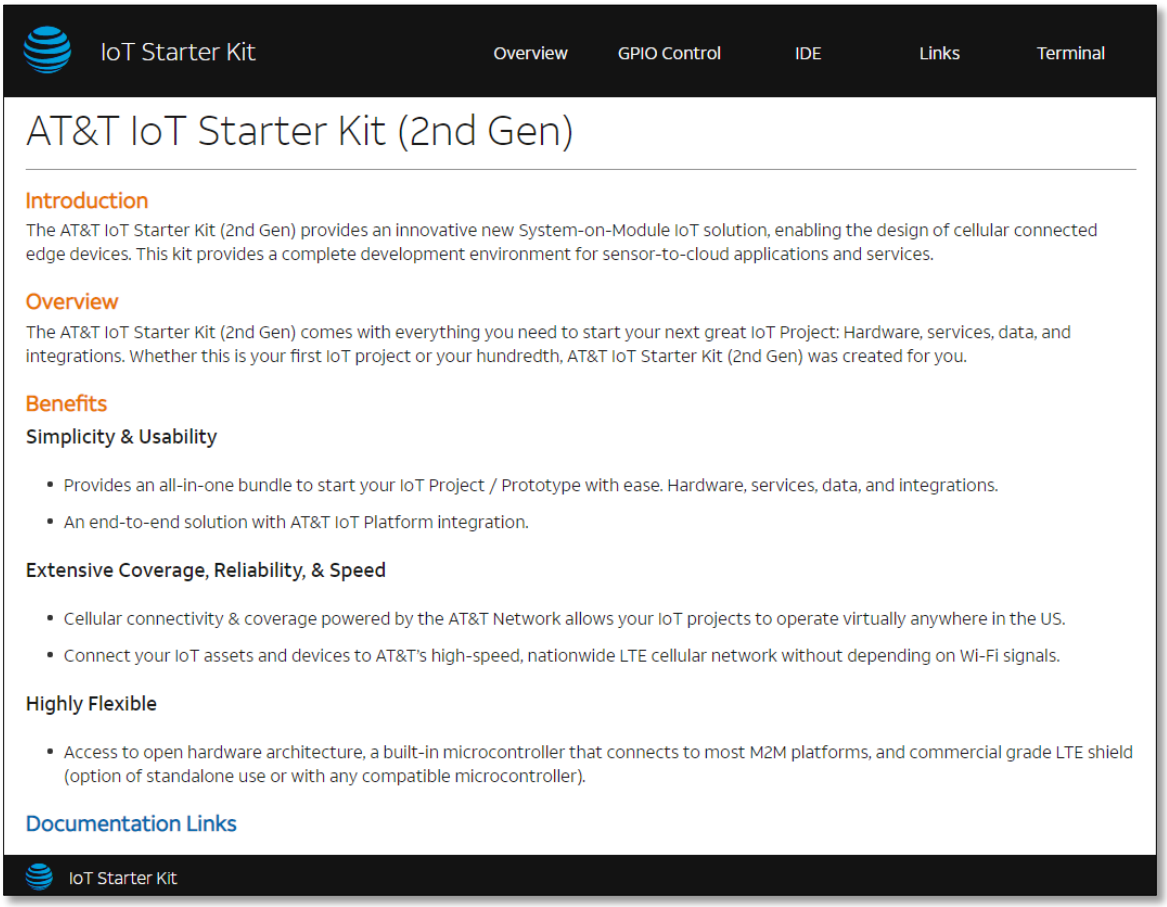


Figure 4.6 - Python IDE (Overview)

The overview provides a short introduction to the SK2 and its advantages. Next, let's explore the main facets of the Python IDE.

4.2.2. GPIO Control

2. **Click the "GPIO Control" button towards the top of the Python IDE.**

   Clicking the button:

   ```
   GPIO Control
   ```

   opens a screen that allows you to control the RGB LED on your connected SK2 as well as reading the ADC value.

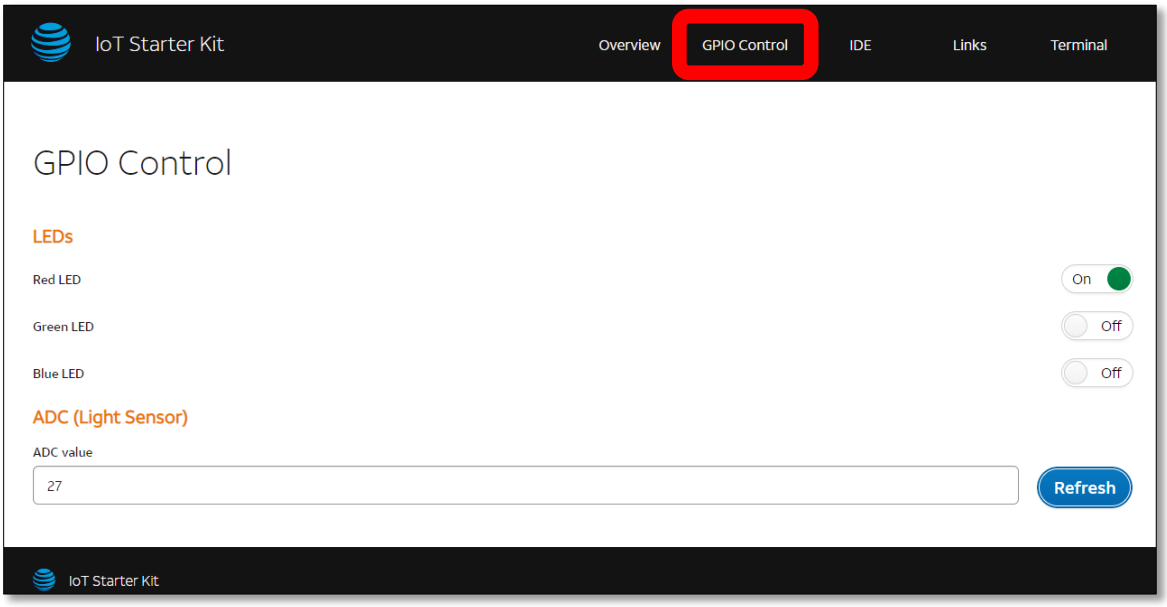


Figure 4.7 - Python IDE (GPIO Control)

3. **Try turning the various LED colors on/off.**

   In the above diagram, we have turned on the Red LED.

4. **Click 'Refresh' to read the ADC (analog to digital convertor).**

   The ADC peripheral on the WNC module is connected to a light sensor on the SK2. Reading the ADC gives us a number that represents the amount of light hitting your board.

   Try reading the ADC by clicking 'Refresh'. Then change the amount of light hitting the board – say, by shining a flashlight on it – and clicking 'Refresh' again. Here's an example of our readings:

Room lighting	27
Flashlight	3521

4.2.3. IDE

5. **Click the "IDE" button towards the top right of the Python IDE.**

   Clicking the button:

   `IDE`

   opens the main Python IDE screen where you can create, edit, and run Python scripts.

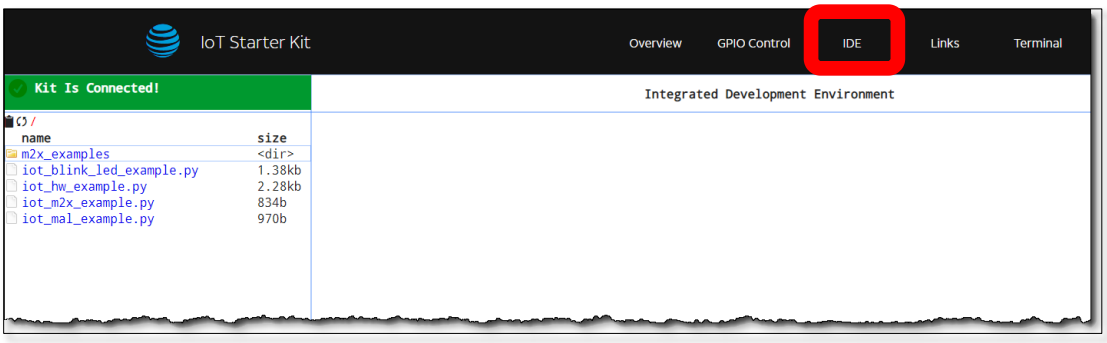


Figure 4.8 Python IDE (IDE)

6. **Open the `iot_blink_led_example.py` that comes preinstalled with the Python image.**

   Double-click on the script to open it in the editor, allowing you to modify or run the Python script.

   `iot_blink_led_example.py`

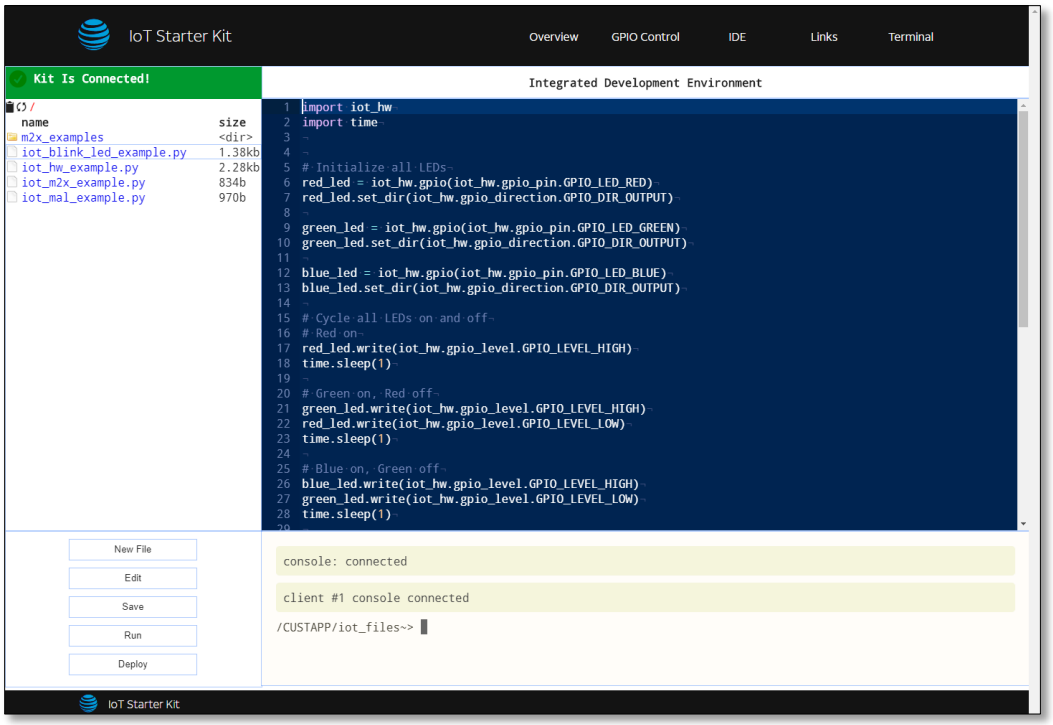


Figure 4.9 - Python IDE (IDE) showing code

7. **Run the open script by clicking the "Run" button in the lower left-hand corner.**

   When the IDE verifies the script you want to run, click "OK".

Refreshing IDE File Listing

Note that if you modify the files in the Python IDE directory while it is open, the IDE may not see the new files or folders. You can refresh the file listing by clicking the “Refresh” button or using Ctrl-R.

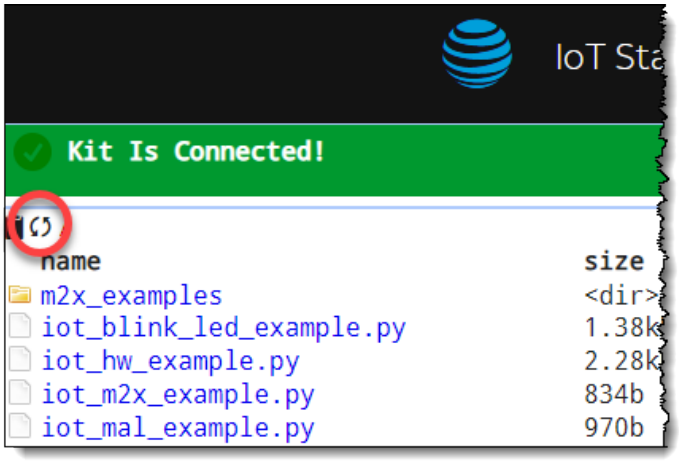


Figure 4.10 - Refresh File Listing

4.2.4. Links

8. **Click “Links” to open the IDE’s links page.**

   This page provides a few links connecting you to SK2 resources, such as the product page.

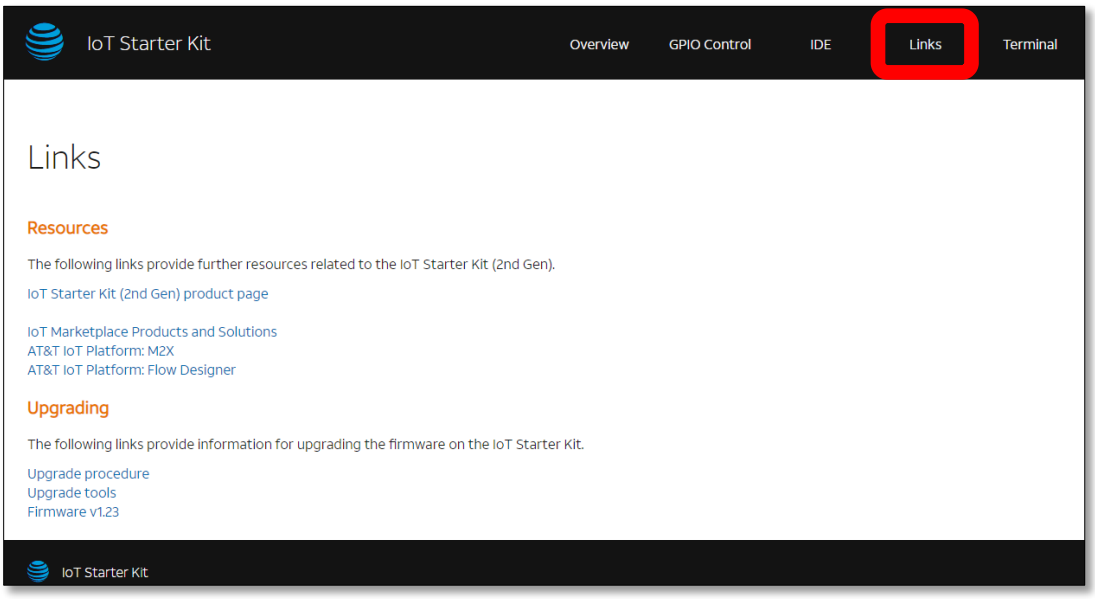


Figure 4.11 - Python IDE (Links)

Note: Some of these links may not be active.

4.2.5. Terminal

9. **Click "Terminal" to open a BASH command-line terminal.**

   This page allows you to connect to your SK2 and run command-line functions. It provides an alternate to the ADB connection that we have used in other parts of this user guide.

   For example, we executed the Python version command that was used earlier in this chapter.

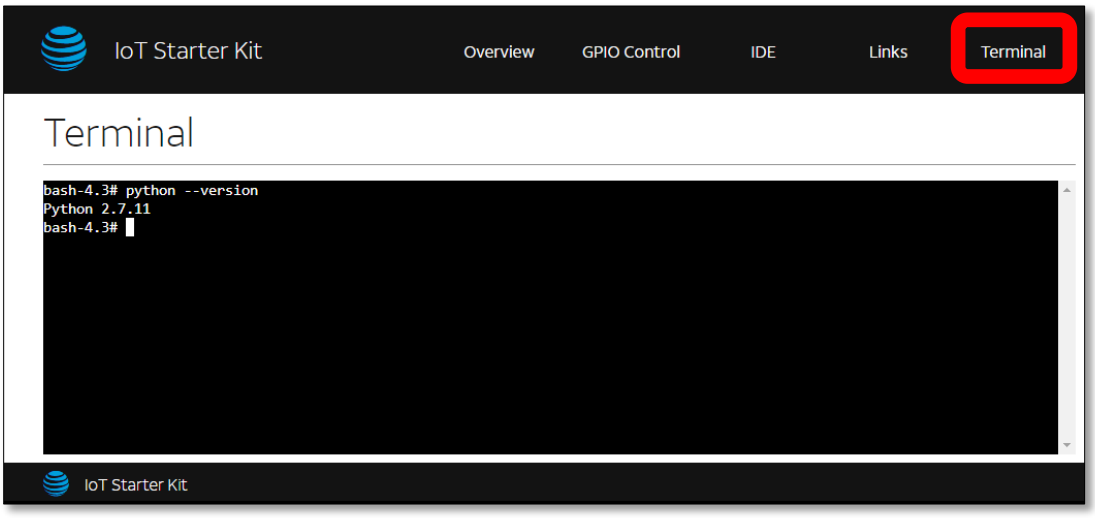


Figure 4.12 - Python IDE (Terminal)

4.3. Getting Started with Python

Python is high-level, general purpose programming language that's easy and fun to program. In fact, it was named after the British television comedy *"Monty Python's Flying Circus"*, so how much more fun does it get than this?

Here are some key distinguishing features from *"Python in Easy Steps"* (Mike McGrath) that do well in describing the Python language. (See "Additional References" at the end of the chapter for a reference to his book.)

- **Python is free** – is open source distributable software.
- **Python is easy to learn** – has a simple language syntax.
- **Python is easy to read** – is uncluttered by punctuation.
- **Python is easy to maintain** – is modular for simplicity.
- **Python is "batteries included"** – provides a large standard library for easy integration into your own programs.
- **Python is interactive** – has a terminal for debugging and testing snippets of code.
- **Python is portable** – runs on a wide variety of hardware platforms and has the same interface on all platforms.
- **Python is interpreted** – there is no compilation required.
- **Python is high-level** – has automatic memory management.

Like with shell-scripts and C, we'll begin using Python with the simple "Hello World" program, after which we'll quickly focus on how to access the SK2 hardware using the Python language.

4.3.1. Running Python from the Command-Line

Running Python from the command-line is different than you might have seen with other languages, such as C/C++ which was covered in the previous chapter. While there are utilities that can convert Python scripts into executable files, generally Python scripts (i.e. Python .py files) are executed by the Python interpreter. But before we explore executing Python files, let's examine the Python command interpreter.

4.3.1.1. Python Interactive Interpreter

While you are not likely to use the Python interpreter in your normal SK2 applications, it can be handy if you're starting out with Python and want to execute commands one-by-one. Let's look at this first.

1. **Open a SK2 remote command-line session.**

   There are now two easy ways to do this. With the SK2 powered-on and connected to your computer:

   - You can execute "adb shell" from your host computers terminal, as we've been doing since Chapter 2.
   - Open the Python IDE's "Terminal" window, as was discussed in the previous section of this guide.

   It doesn't matter which method you choose, either will let you run Python.

2. **Start the Python interpreter.**

   Typing the command "python" without any arguments starts the python interpreter.

   ```
   python
   ```

   Hint: Note that you can see an example invocation of the commands from section 4.3.1.1 in *Figure 13 - Interactive Python interpreter session*.

3. **Enter the Python syntax to print 'Hello World' to the terminal output.**

   ```
   print('Hello World!')
   ```

   Python doesn't care if you use single or double quotes, as long as they're a matched pair. Also, while Python 2 doesn't require parenthesis for print statements, Python 3 does, so we recommend getting used to it right from the beginning.

4. **Execute a Python expression with the interpreter.**

   Enter the following expression:

   ```
   10 * (20 + 12)
   ```

   and see it return the result.

5. **You can even use variables.**

   Try entering the following three lines, hitting return after entering each one:

   ```
   a = 20
   b = 12
   10 * (a + b)
   ```

   Should return the same number we saw earlier.

6. **How about getting information from the command-line user (i.e. yourself).**

   Try the following two lines:

```
c = input("Enter a number: ")
10
print( int(c) * (a + b) )
```

   The input function reads characters from the command line, in this case, assigning the value to the variable 'c'. We then printed the result of an expression – though we needed to cast the character value in 'c' as an integer before we could mathematically combine it with 'a' and 'b'.

   Here are the commands from this section while using the Python IDE Terminal.

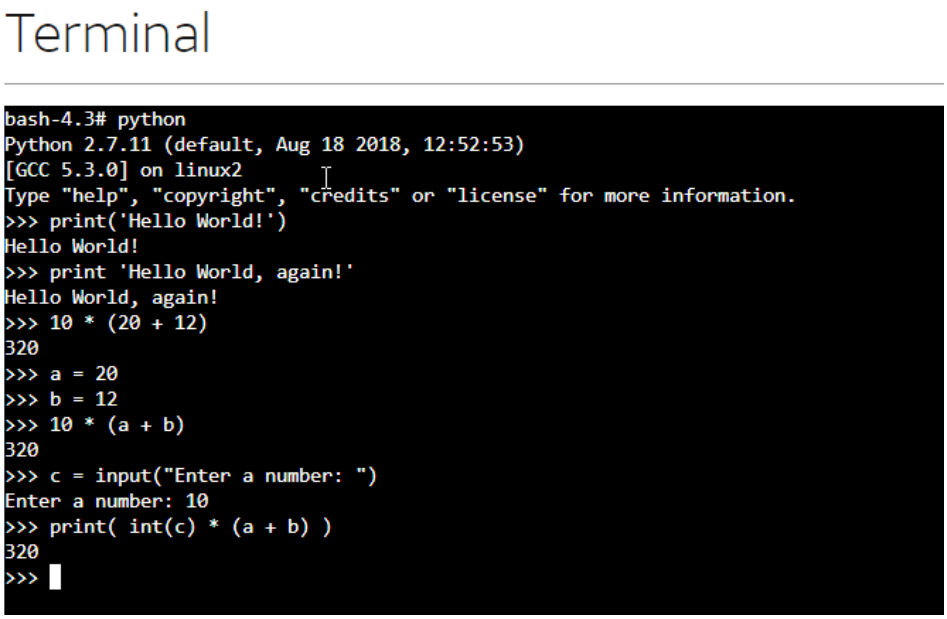


Figure 4.13 - Interactive Python interpreter session

7. **Exit the Python interpreter.**

```
exit()
```

4.3.1.2. Running Python scripts

More likely you will want to write your Python code into a file and run it. Before we write and execute our own code, let's simply try running one of the examples that's included within the Python image.

1. **Open a SK2 remote command-line session.**

   Use either of the methods discussed in Section 4.3.1.1, Step 1 (on page 119).

2. **Open the /CUSTAPP/iot_files directory on your SK2.**

   If you are using the Python IDE Terminal, you should already be at this location. If using the "adb shell" method, you'll need to change directories to reach this location.

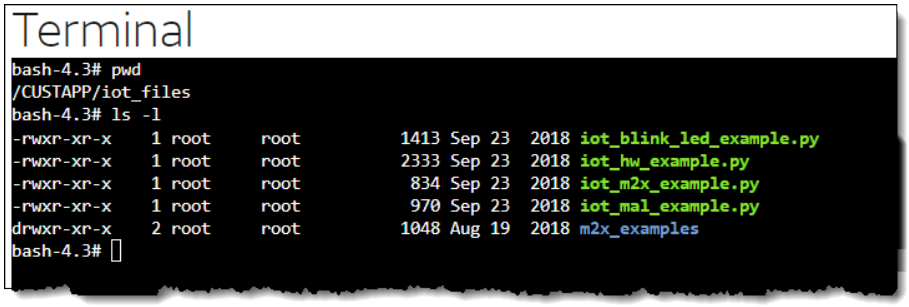


Figure 4.14 - Navigating to 'iot_files'

3. **Run the `iot_blink_led_example.py` python script.**

   This is the same file we executed earlier, back in Section 4.2.3. To execute it from the command line, you need to include "python" along with the file name.

   ```
   python iot_blink_led_example.py
   ```

   As the script it running, you should see the RGB LED blink a variety of colors.

4.3.2. Hello World, the Python Way

Writing Python can be accomplished with any text editor, then executing the code as was examined in the previous section. The SK2 makes the task easier, though, by providing a Python aware text editor in the IDE view. Let's create our first Python file using the IDE.

1. **Open the Python IDE to the 'IDE' pane.**

2. **Create a new Python file named "hello.py".**

   There are two ways to create a new file in the IDE:

   a) Select the "New File" button.

   b) Right-click in the editor, then choose: *New* → *File*.

   In either case, enter the name "hello.py" and hit OK.

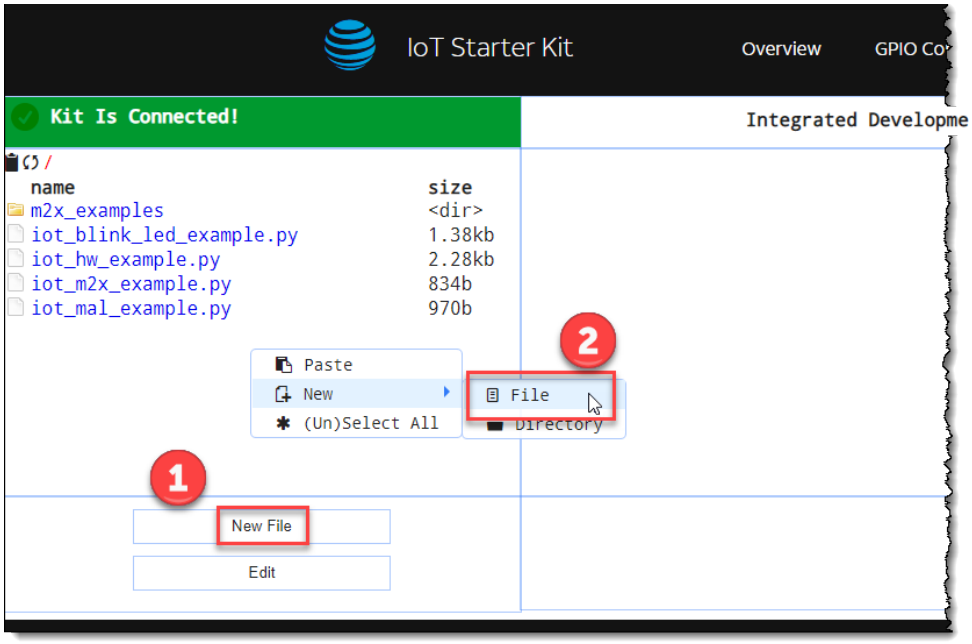


Figure 4.15 - Creating a New Python File

Hint: Notice that the Right-click menu also allows you to create a new "Directory".

Both tasks – creating a new file or directory – can be done from the command-line. There isn't anything fancy being done by the IDE, it just provides a simple way to create these objects without having to leave the IDE.

3. **Open "hello.py" in the IDE's editor.**

   This can also be done two different ways:

   a) Double-click on the 'hello.py' filename.

   b) Right-click on the filename and choose "Edit".

4. **Enter the code for Hello World!**

   Simply add the following code which we saw earlier in the chapter:

```
print("Hello World!")
```

5. **Save and run your hello.py script.**

   Clicking the two buttons in the lower-left corner will save and execute your script.

   ```
   Save
   Run
   ```

   Upon clicking run, you'll be asked to confirm the command that will be executed. While we could modify it before clicking OK, we don't need to do that for this example.

   ```
   Click OK
   ```

   Hopefully your program ran fine. Ours didn't. Figure 16 shows the error we received.

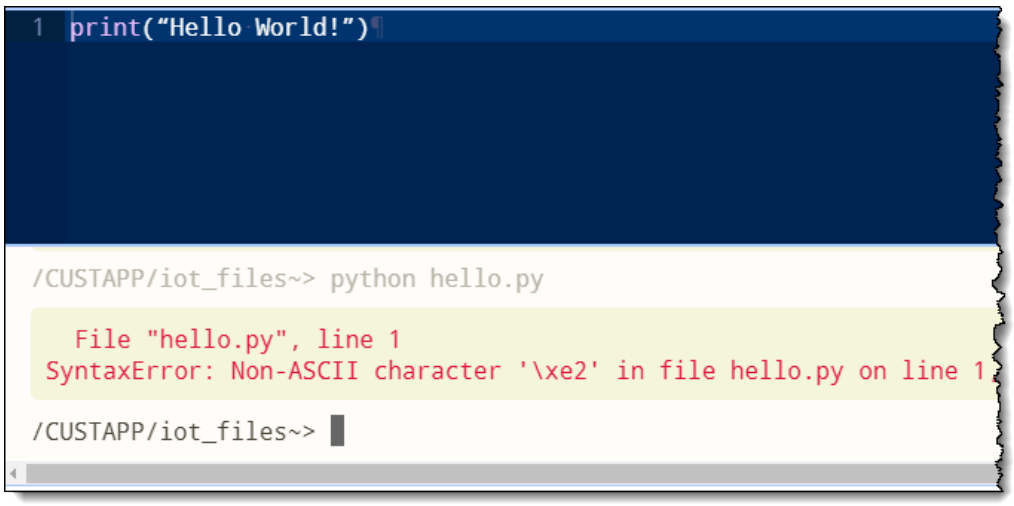


Figure 4.16 - Running 'hello.py' (with error)

It's difficult to see the error. It's the same code we used before with the interactive interpreter, so why is it failing now? Well, we were lazy and copied the code from the user's guide. Unfortunately, Microsoft Word has a habit of replacing quotes with fancy curly versions… which apparently the Python interpreter doesn't like. Replacing the quotes in our file fixed the problem.

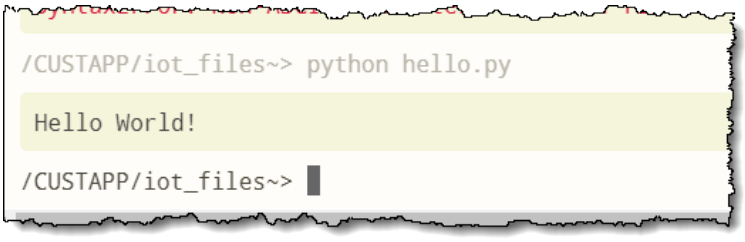


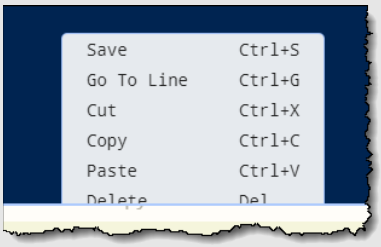
Figure 4.17 - hello.py Success

6. **Close the file.**

```
Right-click and select 'Close'
```

Note that while the Right-click menu indicates that hitting the 'Esc' key will close the file, it didn't work for us.

Hint: Right-clicking too close to the bottom of the editor hides part of the context menu and you might therefore be unable to see the lower options, such as 'Close'.



Make sure you Right-click high enough on the screen to see the entire list.

4.3.3. Cheer for 'Hello World'

To demonstrate another relatively simple program, we expanded on the Hello World script by adding a for loop. The code is in Listing X (and can be downloaded from the Users Guide Git).

Listing 4.1: Chapter_04/hello_cheer.py

```
# hello_cheer.py

msg = 'Hello World'

for c in msg:
    print("Give me a '" + c + "'")

print("\nWhat does it spell? \n " + msg)
```

For those that are new to Python, note that the language is easy to read. Also note that white space is important. The "contents' of the for loop are defined by white space; notice that the print() statement in the loop is indented and will be repeated, while the second print() only executes once. (By the way, the "\n" characters tell Python to insert a line-feed, similar to how they work in C/C++.)

```
/CUSTAPP/iot_files # python hello_cheer.py
Give me a 'H'
Give me a 'e'
Give me a 'l'
Give me a 'l'
Give me a 'o'
Give me a ' '
Give me a 'W'
Give me a 'o'
Give me a 'r'
Give me a 'l'
Give me a 'd'

What does it spell?
 Hello World
/CUSTAPP/iot_files #
```

Figure 4.18 - Running hello_cheer.pycd

4.3.4. Blinking the Red LED

Like shell-scripts and C/C++, blinking an LED is a good way to begin working with hardware. It's easy to see the results by verifying if the LED turns on or not.

The `red_led_blink.py` example (Listing 2) was easy to create because we could borrow directly from the `iot_blink_led_example.py` that came with the Python firmware image.

Listing 4.2: Chapter_04/red_led_blink.py

```
import time                    # Needed for time.sleep() function
import iot_hw                  # Allows access to SK2 hardware resources

# Define 'ON' and 'OFF'
ON  = iot_hw.gpio_level.GPIO_LEVEL_HIGH
OFF = iot_hw.gpio_level.GPIO_LEVEL_LOW

# Allocate and configure GPIO as output for Red LED
red_led = iot_hw.gpio(iot_hw.gpio_pin.GPIO_LED_RED)
red_led.set_dir(iot_hw.gpio_direction.GPIO_DIR_OUTPUT)

# Red on
red_led.write(ON)
time.sleep(1)

# Red off
red_led.write(OFF)

# Release Red LED resource
red_led.close()
```

Executing the Python script will blink the Red LED once. Here are some notes about this example:

import time	Like C, 'time' is a standard library which provides a simple sleep() function
import iot_hw	Again, like C/C++, this imports the library API which allows us to use the SK2 hardware resources
ON, OFF	These variables were not required; alternatively, the code could have used the longer values (i.e. iot_hw.gpio_level.GPIO_LEVEL_LOW) in the write() function
iot_hw.gpio()	Allows the program to allocate GPIO resources; you must import iot_hw in order to use this function; setting the variable with iot_hw.gpio_pin.GPIO_LED_RED selects the color red – i.e. it tells the library which pin needs to be used
iot_hw.set_dir()	Configures the red_led object's direction, which was allocated by iot_hw.gpio()
red_led.**write()**	The write() function, provided by the iot_hw module sets the value of the GPIO pin

Please examine the `iot_blink_led_example.py` to see another example utilizing the RGB LEDs.

4.3.5. Reading the User Button

Reading the User Button on the SK2 is similar to using the LEDs except for two items:

- The Button needs to be set as an input (iot_hw.gpio_direction.GPIO_DIR_INPUT)
- You read() the button rather than write() to it.

The following example simply reads from the button and prints out its value with a string.

Listing 4.3 - Chapter_04/button_read.py

```
# button_read.py
#
# This script reads the button and prints whether
# the button is up or down

import iot_hw

ON  = iot_hw.gpio_level.GPIO_LEVEL_HIGH
OFF = iot_hw.gpio_level.GPIO_LEVEL_LOW

# Initialize the GPIO connected to the SK2 User Button
button = iot_hw.gpio(iot_hw.gpio_pin.GPIO_USER_BUTTON)
button.set_dir(iot_hw.gpio_direction.GPIO_DIR_INPUT)

val = button.read()

if val:
    print("The button is up (" + str(val) + ")")
else:
    print("The button is down (" + str(val) + ")")
```

Running the example twice – first with the button up, and then down – gives the following output.

```
/CUSTAPP/iot_files/Chapter_04> python button_read.py
The button is up (1)
/CUSTAPP/iot_files/Chapter_04> python button_read.py
The button is down (0)
```

Figure 4.19 - Running the button_read.py script twice

Note: Configuring the button as an interrupt is explored in Section 4.5.

4.3.6. Accessing Additional GPIO Pins

The *Python Peripheral IoT API Guide* provides details of all the peripheral API functionality provided by the iot_hw Python module.

The API guide also includes details for setting up all the GPIO pins including the RGB LEDs and the User Button.

4.3.7. Handy way to Kill Python Program

If you accidentally write a program that won't exit (e.g. an endless loop) you a second terminal and Linxu to kill it. This is done by getting the PID (process ID number) assigned to your program by Linux, then using the Kill command as discussed in Chapters 2 and 3.

```
ps -e | grep python
kill <pid>
```

Here's a simple code example for an endless loop:

Listing 4.4: endless.py

```
import time

print("Starting up...")
while 1:
    time.sleep(5)
    print("Five more seconds have gone by...")
```

Running the `endless.py` example in the Terminal (below left), the program will run forever.

By starting a second remote terminal session on the SK2 (below right)

- Run the "ps" command to list the active Linux processes and pipe the results into grep to only show those named 'python'
- Using the PID (process ID), which in this case was 4376, you can tell Linux to kill the process

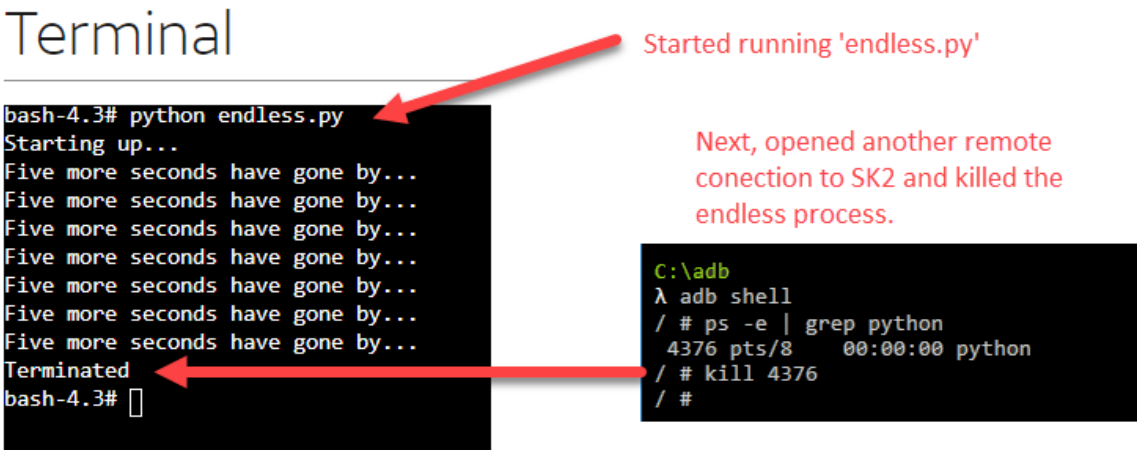


Figure 4.20 - Running and killing endless.py

4.4. WWAN LED Class

The Python peripheral API (iot_hw module) does not include support for the WWAN LED. The User Guide software examples include a Python class which allows easy use of the WWAN LED. The class includes the following methods: on(), off(), toggle(), value(), read().

4.4.1. First, a little about Python Functions and Classes

4.4.1.1. Python Functions

You can create functions in Python using the "def" (define) keyword. The following example defines and then uses a function called my_func().

```
# my_func example

def my_func(a):
    print(a)

# Using the function my_func()
my_func('Print me!')
```

Which outputs:

```
Print me!
```

Notice that, like the for loop examined earlier, the body of the function is defined by the white space, either using tabs or spaces.

Just like using C, or other extensible languages, you can "import" functions from other .py files. For example, as long as we had added the proper import statement, we could have defined my_func() in a separate .py file, away from where it was used.

4.4.1.2. Python Classes

Like other high-level languages, such as C++, Python classes can contain *data* and *methods* (i.e. functions). The data and functions are coded like they are outside of classes with two distinct differences:

- Class elements must follow a “class” statement.
- Class elements must be indented using white-space.

For example, a class called “my_class” might look like:

```
class my_class:
    my_class_variable = 0

    def __init__(self):
        # add code here to instantiate the class object

    def print_me(self, a):
        print(a)
```

You could then use the class in your code by:

```
x = my_class()
x.print_me(“Hello”)
```

We don’t include this example in our code listings since the next section examines a real-world example class. The goal here was to provide a simple description for Python classes.

4.4.2. The WWAN LED Class Code Example

The following code makes it easy to access and control the WWAN LED from your Python scripts, whether you want to turn the LED on or off. You can also toggle the LED and read its value.

Listing 4.5: Chapter_04/wwan_class.py

```
################################################################################
# wwan_class.py
# Created 2018.12.09
#
# WWAN LED routine uses os and subprocess to call shell commands to read and
# modify the WWAN device driver file descriptor. It was found that Python
# file I/O was an unreliable method for managing the WWAN LED.
#
################################################################################
import os
import time
import subprocess

WWAN_LED = r'/sys/class/leds/wwan/brightness'

OFF = '0'
ON  = '1'

COUNT = 1

class wwan:
    value = '0'

    def __init__(self):
        self.value = self.read()

    def on(self):
        self.value = '1'
        os.system("echo 1 > " + WWAN_LED)

    def off(self):
        self.value = '0'
        os.system("echo 0 > " + WWAN_LED)

    def toggle(self):
        i = int(self.value) ^ 1
        self.value = str(i)
        msg = "echo " + self.value + " > " + WWAN_LED
        os.system(msg)
        return self.value

    def value(self):
        return self.value

    def read(self):
        p = subprocess.Popen(["cat", WWAN_LED], stdout=subprocess.PIPE)
        s = p.communicate()
        r = s[0].strip()
        return r
```

The WWAN LED is turned on/off using file I/O as was described in the Linux chapter. Using the value() method returns the value tracked in the class itself. Whereas, the read() method actually goes out and reads the value from filesys.

4.4.3. Importing & Using the WWAN LED Class

Using the WWAN class by following the steps required for most object-oriented languages. You must follow two steps before you can use it:

- Import the class
- Instantiate the class object

At which point you are free to use it in your code.

Listing 4.6: Chapter_04/wwan_blink.py

```
#¬wwan_blink.py¬
#¬
#¬This code provides a simple example for¬
#¬instantiating and using the 'wwan' class¬
¬
import time¬
from wwan_class import wwan¬
¬
COUNT = 4¬
¬
print("Starting wwan_blink.py...")¬
w = wwan()¬
w.off()¬
¬
for i in range(0, COUNT*2):¬
    v = w.toggle()¬
    time.sleep(1)¬
```

In this example, after we imported the “wwan” class from the wwan_class python file, we were able to instantiate our local class object by setting our variable “w” equal to the wwan() class. From that point on we used the off() and toggle() methods in order to blink the WWAN LED 4 times.

4.4.4. Why does WWAN Class use OS and Subprocessing?

We planned to write a simple example that showed accessing WWAN and the RGB LEDs using simple File I/O function calls like the examples in the C/C++ chapter. This proved to be more difficult in Python than expected. While performing file reads/writes in Python is pretty easy, there were timing issues when talking to the virtual GPIO and WWAN drivers.

With GPIO resources being well handled by the Python Peripheral Driver API (i.e. iot_hw module), the File I/O examples for the LEDs and User Button were skipped. Unfortunately, since the API does not provide support for the WWAN LED, this needed to be handled with a File I/O solution.

Sending commands (i.e. sending 0 or 1) to the WWAN LED is less problematic than GPIO since this resource does not need to be exported before use. That said, timing issues still arose, especially when trying to blink the WWAN LED quickly. The simple answer was to call shell commands from Python using the “os.system()” call. Examples for this can be seen in the following WWAN class methods: on(), off(), and toggle().

Reading the WWAN LED value was tricky. The Linux ‘cat’ function would read the WWAN virtual file from sysfs, but subprocessing was needed to return the value from ‘cat’ back to Python.

Subprocessing let your Python program spawn new applications. It also has a mechanism for returning results back to your program. You can see an example of this in the WWAN read() method.

4.5. GPIO Interrupts in Python

Along with simply reading a GPIO input value, the Python Peripheral API also supports interrupts. This support is similar to the C/C++ Peripheral API support which was discussed in Chapter 3.

Using GPIO pins to generate interrupts requires three steps:

1. Allocate the GPIO pin.
2. Configure the GPIO pin as an input.
3. Set the GPIO input as an interrupt (i.e. irq) specifying:
   a) Whether interrupts should be *triggered* when the pin's value rises, falls, or in both cases.
   b) What *callback* function should run when a configured interrupt occurs.

The first two steps were explored in Section 4.3.5 *Reading the User Button*. Interrupts require one more function call to register the *trigger* and *callback* values.

Examine the following example which uses the SK2 User Button to trigger an interrupt.

Listing 4.7: Chapter_04/button_interrupt.py

```
#############################################################################
# button_interrupt.py
# 2018.12.07
#
# Configure SK2 User Button to generate interrupt, then sleep for 30 seconds.
# If button is pushed within 30 seconds, light the Blue LED and then exit the
# program. If not pushed, the program will still exit after the 30 second wait.
# Note: If turned on, Blue LED will stay on after program exits.
#
# This example was adapted from the sample code provided in the Python
# Peripheral API Guide.
#############################################################################
import iot_hw                                   # Needed for GPIO API to use SK2 LE
import time                                     # Needed for time.sleep() function

# A callback function that will turn on the blue LED
def callback_function(gpio_pin, trigger):
    print("Interrupt occured...")
    led = iot_hw.gpio(iot_hw.gpio_pin.GPIO_LED_BLUE)      # Allocate Blue LED GPIO pin
    led.set_dir(iot_hw.gpio_direction.GPIO_DIR_OUTPUT)    # Set Blue LED GPIO pin to "output"
    led.write(iot_hw.gpio_level.GPIO_LEVEL_HIGH)          # Turn on Blue LED
    led.close()
    return 0

print("Starting up...")
button = iot_hw.gpio(iot_hw.gpio_pin.GPIO_PIN_98)         # Allocate User Button GPIO pin
button.set_dir(iot_hw.gpio_direction.GPIO_DIR_INPUT)      # Configure User BUtton as "input"
button.interrupt_request(
    iot_hw.gpio_irq_trig.GPIO_IRQ_TRIG_FALLING,           # Trigger interrupt when pressing
    callback_function)                                    # Run callback_function() after the

print("Ready to sleep...")
time.sleep(30)                                  # Sleep for 30 seconds
# Press the button here, during the 30 second wait

print("Freeing resources...")
button.interrupt_free()                         # Turn off button interrupt and fre
button.close()                                  # Free up User Button's GPIO pin
```

Running the button_interrupt.py function results in the following:

```
/CUSTAPP/iot_files/Chapter_04> python button_interrupt.py

Starting up...
Ready to sleep...
Interrupt occured...
Freeing resources...
```

Figure 4.21 - Running button_interrupt.py

Note that if the User Button was not pushed within the 30 second sleep time, the resulting output would not include “Interrupt occurred…”.

4.5.1. Fancier Button Interrupt Example

The following example (Here’s the fancy code example. Notice below how the interrupt’s callback function ‘trigger’ argument can be used to determine whether the button was pushed or released.

Listing 8) combines several elements discussed in this chapter:

- RGB LED using the Python Peripheral API
- User Button configured as an interrupt
- Interrupt callback function detecting which edge (rising, falling) and acting accordingly
- Using Python time.time() to get the current time – which was used to determine if the button had be held for longer than 3 seconds
- WWAN LED class used to signify script is running
- ‘global’ keyword allows use of global variables within a function

Running the example produces the following output:

```
/CUSTAPP/iot_files/Chapter_04 # python button_interrupt_fancy.py

Starting up... please wait
Ready... feel free to press button

--> Falling interrupt occured...
<-- Rising interrupt occured...
Button was held down for 2.962386 seconds

--> Falling interrupt occured...
<-- Rising interrupt occured...
Exiting... because button was held down for more than 3 seconds (5.499417 to be exact)

Freeing resources...

/CUSTAPP/iot_files/Chapter_04 #
```

Figure 4.22 - Running button_interrupt_fancy.py

Here's the fancy code example. Notice below how the interrupt's callback function 'trigger' argument can be used to determine whether the button was pushed or released.

Listing 4.8: Chapter_04/button_interrupt_fancy.py

```
################################################################################
#  button_interrupt_fancy.py
# 2018.12.17
#
# After initialization, the routine will continue to blink the WWAN LED
# until the User Button is pressed and held for 3 or more seconds.
#
# Each time the User Button is pressed, the Blue LED is toggled on or off.
################################################################################
# Import modules
import iot_hw                                           # Needed for GPIO API to use SK2 LEDs and Button
import time                                             # Needed for time.sleep() function
from wwan_class import wwan                             # Needed for WWAN LED class

# Global variables
loop = True                                             # Used for main while loop
blue = 1                                                # Used when toggling Blue LED
button_pressed = 0                                      # Keeps track of time button is pressed between interrupt callbacks

# A callback function that will turn on the blue LED
def my_callback_fcn(gpio_pin, trigger):
    global loop, blue, button_pressed                   # Give function access to these global variables

    if trigger == 0:                                    # If pushing the button down
        print("--> Falling interrupt occured..." )
        button_pressed = time.time()                    # Store current time to 'button_pressed'

        blue ^= 1
        if blue == 0:
            led.write(iot_hw.gpio_level.GPIO_LEVEL_HIGH)    # Turn on Blue LED
        else:
            led.write(iot_hw.gpio_level.GPIO_LEVEL_LOW)     # Turn off Blue LED

    else:                                               # If letting up on the button
        print("<-- Rising interrupt occured...")
        button_held = time.time() - button_pressed      # Calculate how long button was held

        if button_held >= 3:
            loop = False                                # Change while loop's exit condition since button was held long enough
            print("Exiting... because button was held down for more than 3 seconds (" + str(button_held) + " to be exact)\n")
        else:
            print("Button was held down for " + str(button_held) + " seconds\n")

    return 0

# Main code starts here
print("\nStarting up... please wait")
led = iot_hw.gpio(iot_hw.gpio_pin.GPIO_LED_BLUE)        # Allocate Blue LED GPIO pin
led.set_dir(iot_hw.gpio_direction.GPIO_DIR_OUTPUT)      # Set Blue LED GPIO pin to "output"
led.write(iot_hw.gpio_level.GPIO_LEVEL_LOW)             # Turn off Blue LED

button = iot_hw.gpio(iot_hw.gpio_pin.GPIO_PIN_98)       # Allocate User Button GPIO pin
button.set_dir(iot_hw.gpio_direction.GPIO_DIR_INPUT)    # Configure User BUtton as "input"
button.interrupt_request(
    iot_hw.gpio_irq_trig.GPIO_IRQ_TRIG_BOTH,            # Trigger interrupt when pressing down on button
    my_callback_fcn)                                    # Run my_callback_fcn() after the interrupt is generated

w = wwan()                                              # Create instances of WWAN LED class and set equal to 'w'
w.off()                                                 # Turn off WWAN LED

print("Ready... feel free to press button\n")
while loop == True:                                     # Loop until 'loop' variable is changed by the interrupt routine
    w.toggle()                                          # Toggle WWAN LED
    time.sleep(1)                                       # Sleep for 1 second

# Don't access this code unless button is pressed and held for 3 or more seconds
print("Freeing resources...\n")
button.interrupt_free()                                 # Turn off button interrupt and free IRQ resources
button.close()                                          # Free up User Button's GPIO pin
led.close()                                             # Free up Blue LED
w.off()                                                 # Make sure the WWAN LED is off
```

4.6. Running Python Scripts at Boot

The Linux boot sequence was discussed towards the end of Chapter 2. But here's the gist of that discussion: In order to autostart an application when powering on the SK2, you need to add an entry into the following script:

```
/CUSTAPP/custapp-postinit.sh
```

In the following example, we added a shell script to `custapp-postinit.sh`. We viewed this file by executing the Linux 'cat' command.

Using *cat* to view `myCode.sh`, we see that it runs a simple blink example for the Green LED and then executes the fancy interrupt example discussed in the previous section of this chapter.

Figure 4.23 - Viewing custapp-postinit.sh and myCode.sh

4.7. Additional References

There are a great many Python tutorials available on the Internet. Additionally, there are a great number of Python books. Here are a couple that we found useful:

- Mike McGrath. *Python In easy steps (2nd Edition)*. In Easy Steps Limited, 2018.
- Magnus Lie Hetland. *Beginning Python: From Novice to Professional (3rd Edition).* Apress, 2017.

Please note that two generations are in active use and both supported by the community: Python 2 and Python 3. The SK2 supports Python 2, so keep this in mind as you reference the books or Internet. There aren't a lot of differences between the two, but it good to keep this in mind when researching and referencing Python information.

Along with the Python documentation, you may need to reference the API document:

- *SK2 Python Peripheral API Guide* (link not available at the time of publishing)

Appendix

Topics

Glossary

APN (Access Point Name)

The APN is the endpoint where cellular communications enter a cellular carrier's mobile network.

Or, as Wikipedia says, *an **Access Point Name** (**APN**) is the name of a gateway between a GSM, GPRS, 3G or 4G mobile network and another computer network, frequently the public Internet.*

The APN's network must be provisioned to recognize each SIM card that wants communicate with it. Based on this, most cellular phone SIM cards will not be able to communicate with IoT APN's and vice versa.

Embedded System

Embedded Systems are small systems that generally provide a static set of functionalities aimed at addressing a problem. Think, for example, that your microwave oven cooks food, but that's about it. Similarly, your thermostat controls your heating and air conditioning, but is limited to that functionality, too.

EOL (End of Line)

The end-of-line character(s) – also known as "newline" – tell software reading a document that a line of text has ended. While this is a relatively straightforward concept, Mac/Linux/Unix and Windows use different characters (or sets of characters) to signify EOL.

- Mac / Linux / Unix use a line-feed (LF) to signify the end of a line.
- Windows uses line-feed (LF) and a carriage-return (CR) to end a line.

GPIO (General Purpose Input/Output)

Processors (e.g. microprocessors, microcontrollers) have various ways to communicate with external devices, the most basic of which is GPIO. GPIO generally refers to talking across a single pin of the device. With a single pin the device can receive or send (i.e. the input and output in GPIO) an "On" or "Off" value using a higher or lower voltage, respectively. Programmers describe this On/Off communication using a "bit" (binary digit) which can have the value "1" or "0". In light of this, it's easy to understand why many users commonly define GPIO as "General Purpose Bit I/O".

LED (Light Emitting Diode)

A light-emitting diode is a semiconductor light source. It's a diode that emits light a suitable current is applied to it. LEDs can be created in a variety of colors.

Microcontroller (MCU) / Microprocessor (MPU)

The terms *processor*, *microprocessor*, and *microcontroller* are used fairly interchangeably nowadays. They generally refer to an integrated circuit (i.e. semiconductor) that can be programmed with software to implement a variety of tasks.

The *microprocessor* originally consisted of a block called the *central processing unit* (CPU) which handled the execution of software mainly consisting of logical, arithmetic, and control instructions. The term "micro" processor referred to their small size as compared to the original computers (which were often room-size). While *processor* is a general term for "processing" things, it has also become an abbreviated way to reference microprocessors. Likewise, the acronym "MPU" is used for a long list of terms, one of which is "microprocessor".

The term *microcontroller* (MCU) was created in 1971 when engineers from Texas Instruments put all the building blocks of a computer onto a single integrated semiconductor. You can think of MCU's as a superset of the microprocessor. Along with the CPU (found in the microprocessor), they added memory and input/output peripherals.

Over the years both types of processors have become more and more integrated with the addition of different types of memory and peripherals. In fact, it can difficult to tell them apart in todays processor market. Vendors still use the term microcontroller, though, to refer to their processor chips which include Read Only Memory (ROM) – such as Flash EEPROM (electrically eraseable programmable read only memory). Devices use ROM memory to store software instructions, thus making them stand alone computer devices. Microprocessors, on the other hand, require an external ROM-like memory chip to store their instructions, which are usually loaded into fast RAM (read-write memory) at boot time (i.e. startup). In these days of highly integrated circuits, microprocessors still use an external memory chip because inexpensive read-only memory circuits are not fast enough to keep up with the high clock rates of MPU's.

Operating System (OS or O/S)

An operating system is system software that manages computer – or embedded system – hardware and software resources and provides common services for computer programs.

The OS is often thought of the support infrastructure for the program that users what to run. For example, a user may want to run a want to run a word processor. The OS does not include a program such as this, but it provides underlaying software support for such programs.

Similarly, looking at an embedded system example – say, an Internet connected thermostat. The operating system would not provide the logic to control the heating and ventilation (HVAC) unit, but it would provide services to the embedded software that runs the HVAC unit.

SIM (Subscriber Identity Module)

A subscriber identity module or subscriber identification module, widely known as a SIM card, is an integrated circuit that is intended to securely store the international mobile subscriber identity number and its related key. These are used to identify and authenticate subscribers on mobile telephony devices

What is, and how do you configure, the APN?

What is "APN"?

According to Wikipedia:

> *An **Access Point Name** (**APN**) is the name of a gateway between a GSM, GPRS, 3G or 4G mobile network and another computer network, frequently the public Internet.*

In other words, the APN is the endpoint for the cellular network that your IoT Starter Kit connects to. This value is not embedded into the SIM card; therefore, you may need to configure this if it's not connecting to the network properly.

Note, AT&T cellular phones (as well as other vendors) use different APN addresses. As such, you cannot generally use a SIM card provisioned for a cellular phone in your AT&T IoT Starter Kit. You must use the SIM card that comes with the Starter Kit or purchase a new IoT SIM from the AT&T Marketplace. (Note that SIM cards are defined in Chapter 1.)

Starter Kit APN value

The APN for our Starter Kit:

```
m2m.com.attz
```

Your board should be pre-configured with this value. If not, you will need to modify your board's configuration before it can connect with the network.

Prerequisites

The following procedure requires:

- ADB connection to your starter kit. (Chapter 2 - ADB)
- Rudimetary knowledge of Linux filesystem. (Chapter 2 - Filesystems)

View APN

The APN for your SK2 is stored on your kit in the malmanager.cfg file. If you are comfortable with editing using the "vi" editor, you could connect to your SK2 (using "ADB shell") and directly view this file. Alternatively, you can pull this file over to your computer and view it with any text editor.

The following command will "pull" a copy of the malmanage configuration file from your SK2 filesystem to the "adb" folder on your computer.

```
adb pull /data/user/mm_conf/malmanager.cfg C:\adb\
```

You don't have to use the "adb" folder, but that is where we copied it to on our Windows computer.

Once the file is on your computer, you can search for "name" to see if it's set to the correct value (i.e. m2m.com.attz). You can see a picture of the "name" in the diagram below.

For the comparison in Figure A-1, we copied the `malmanger.cfg` from both a working SK2 (left) and a non-working SK2 (right). Notice how the APN in the working system includes "`m2m.com.attz`", while the non-working system does not.

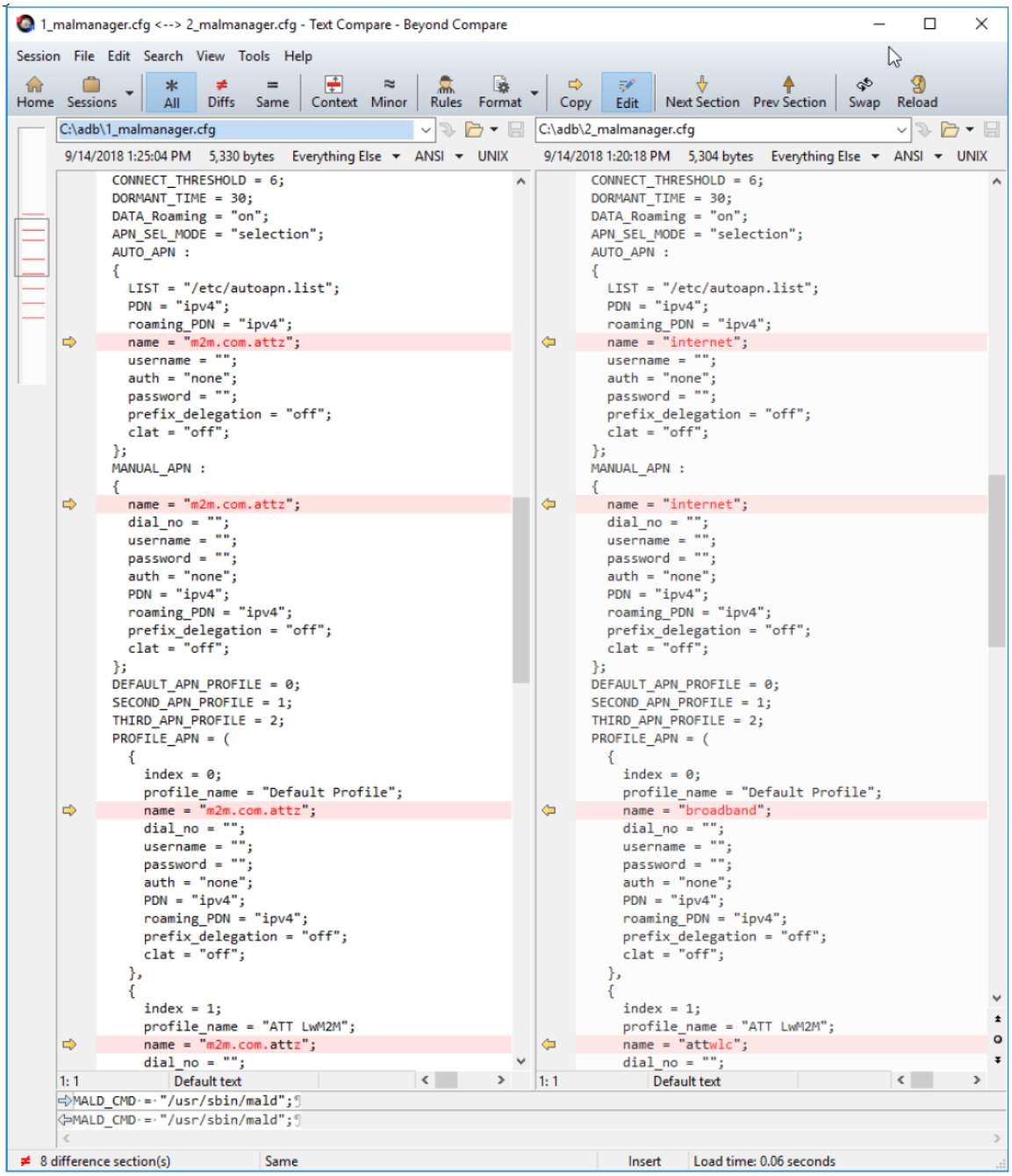


Figure (Appendix) A-1 – Malmanager.cfg comparison

Modify APN

To modify the APN for your SK2, you need to edit malmanager.cfg to include the proper APN value specified by your cellular carrier. For the SK2, that means using “m2m.com.attz”.

Edit in Place

You can edit your APN “in place” by connecting to the SK2 and changing the APN.

1. **Verify that your computer is connected to the SK2.**

   ```
   ADB devices
   ```

2. **Open a remote session on your SK2.**

   ```
   ADB shell
   ```

3. **Change to the mm_conf directory.**

   ```
   cd /data/user/mm_conf
   ```

4. **Edit malmanager.cfg with vi.**

   ```
   vi malmanager.cfg
   ```

5. **Change the APN, as required using “vi”.**

   See Chapter 2 - vi for more info.

6. **Power-cycle your SK2.**

Edit on your Computer

Alternatively, you can pull the malmanager.cfg file over to your computer. Edit it and then push it back onto the SK2.

1. **Verify that your computer is connected to the SK2.**

   ```
   adb devices
   ```

2. **Pull the malmanager.cfg file onto your computer. (We chose to place it into our "adb" folder.)**

   ```
   adb pull /data/user/mm_conf/malmanager.cfg C:\adb\
   ```

   Replace C:\adb\ as needed for your OS and preference.

3. **Change the APN, as required using your favorite text editor.**

   Windows users, make sure you save the file using Unix/Linux/Mac line-endings.

4. **Rename the original file.**

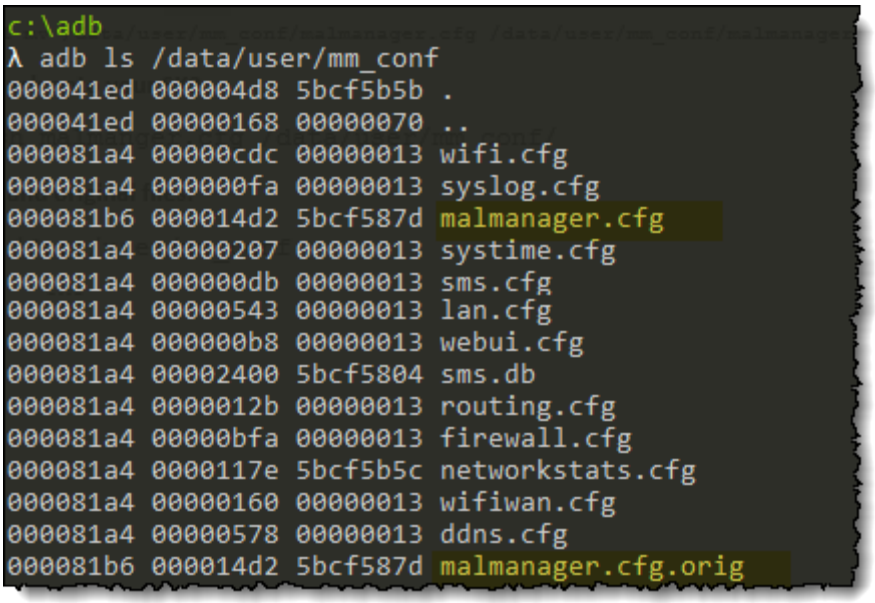
   ```
   adb shell mv /data/user/mm_conf/malmanager.cfg /data/user/mm_conf/malmanager.cfg.orig
   ```

5. **Push the file back onto your SK2.**

   ```
   adb push malmanger.cfg /data/user/mm_conf/
   ```

6. **View the new and original files.**

   ```
   adb ls /data/user/mm_conf
   ```



7. **Power-cycle your SK2 to utilize the new APN.**

General Purpose Bit I/O (GPIO)

What is GPIO?

From the glossary we find GPIO defined as:

> Processors (e.g. microprocessors, microcontrollers) have various ways to communicate with external devices, the most basic of which is GPIO. GPIO generally refers to talking across a single pin of the device. With a single pin the device can receive or send (i.e. the input and output in GPIO) an "On" or "Off" value using a higher or lower voltage, respectively. Programmers describe this On/Off communication using a "bit" (binary digit) which can have the value "1" or "0". In light of this, it's easy to understand why many users commonly define GPIO as "General Purpose Bit I/O".

In other words, GPIO allows us to send or receive a single bit of information at a time through one of the GPIO pins on a processor. While multiple GPIO pins on a device could be grouped together to send larger numerical values than 1 or 0, a single bit of information is actually quite useful in embedded systems. With a single bit we can control lights (e.g. LED) or read the value of a switch. Similarly, we could use it to control whether a motor – or just about anything – is on or off. Of course, there is a limited amount of power that can be supplied by a microcontroller's pin, but the GPIO pin can be used to control a relay or driver that can supply a much greater amount of power to a circuit.

Looking at at a simple circuit connecting an LED to a GPIO pin we see:

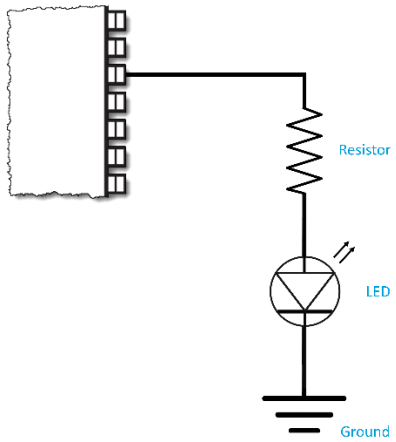


Figure (Appendix) 2 - Generic LED Circuit

In this example, the GPIO pin is supplying the voltage and current which travels through a resistor, the LED, and then into ground. When the pin is set to low (i.e. "0"), there is little to no voltage difference between the pin and ground and therefore the LED is off. Conversely, when the pin is set high (i.e. "1"), the voltage drop between the pin and ground is large enough to allow the LED to turn on.

Note: If you were building this circuit, you would choose a resistor value that allows the full (or desired) brightness of the LED without burning it out.

The LED circuits on the SK2 board are a bit fancier than what we have shown here, but the premise still holds. By setting the GPIO pin on the processor module to "1" or "0" we can turn the associated LED on or off.

SK2 GPIO Pins for LEDs and Pushbuttons

As shown in Chapter 1 (Hardware Overview), the SK2 provides a few user-controllable items connected to GPIO lines on the WNC module. They include:

- WWAN LED – which uses 1 GPIO pin.
- RGB LED – which uses 3 GPIO pins to control the Red, Green, and Blue individual colors.
- User push-button switch – which uses a GPIO pin to input the physical state of the switch.

The SK2 schematic shows how the GPIO lines are connected to each user component. For example, here's a clip of the schematic showing the RGB LED.

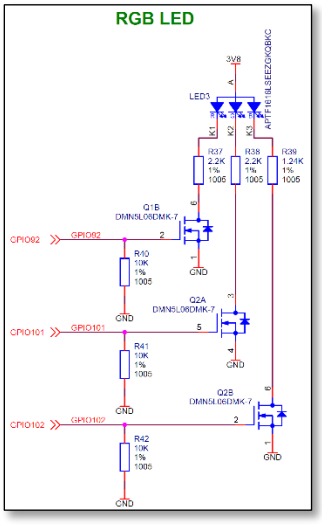


Figure (Appendix) 3 - RGB LED Schematic

From this diagram, we can see that the following GPIO pins are connected to the RGB LED colors:

GPIO92	Red LED
GPIO101	Green LED
GPIO102	Blue LED

Therefore, we need to set the values on these three pins to turn the associated LEDs on/off, which is discussed in Chapter 2 where we demonstrate controlling LEDs from the Linux Shell. In fact, one of the examples takes this a step further by showing how to blink an LED, turning it off/on once per second.

By speeding up the blinking to a rate fast enough that the human eye doesn't perceive blinking anymore, you could effectively control the brightness of each LED. And with the RGB LED, by similarly controlling all three LEDs and their brightnesses, it's possible to create the effect of generating a wide array of colors.

GPIO Pin Numbers – WNC vs Qualcomm

As discussed earlier in this document, the IoT Starter Kit (2nd generation) board includes the WNC M18Q2FG-1 LTE module. If we could look inside the WNC module, we would find it contains a Qualcomm processor, which is at the heart of the SK2.

Figure (Appendix) 4 - SK2 Nested Processor Modules

The GPIO pins from the Qualcomm processor are routed to pins on the WNC module… which are then routed to the various components on the SK2, such as the LEDs. As often happens when building systems up from various components, WNC pin numbers don't exactly match with those coming from the Qualcomm processor. Here's a table that describes the GPIO pin numbers we are concerned with:

SK2 Component	WNC Pin Number	Qualcomm Pin Number
USER Button	GPIO98	GPIO_MDM_GPIO23
RGB – Red LED	GPIO92	GPIO_MDM_GPIO38
RGB – Green LED	GPIO101	GPIO_MDM_GPIO21
RGB – Blue LED	GPIO102	GPIO_MDM_GPIO22

Using this table as a key, we can figure out how to design the hardware or software. Since the SK2 hardware designers were building the board using the WNC module, their design schematics show the WNC GPIO pin numbers (e.g. Red LED connected to GPIO92).

On the other hand, since software actually runs on the Qualcomm processor inside the WNC module, software should be written using the Qualcomm pin number. For example, from the Linux shell we can use the Linux "echo" command to write a "1" to Pin 38 to turn on the Red LED. (We explain what "/value" means in the next section.)

```
echo 1 > gpio38/value
```

Note that code examples for controlling LED's using Python and C will be described in Chapters 3 and 4.

Note: The WWAN LED is driven by the WWAN pin on the WNC module. As such, it does not connect to a general purpose GPIO pin which is why it was not listed in the WNC vs Qualcomm table.

Linux GPIO Drivers

By choosing Linux for the SK2, its developers were granted access to the various services in the O/S. In this case, it means they were able to utilize the Linux GPIO driver architecture. But it's not only the developers who benefit by using standard Linux drivers, SK2 users also get to use a common, standard driver API for GPIO. If you're experienced with Linux, the following should be familiar to you; if you're new to Linux, you'll not only learn how to use the GPIO driver for the SK2, but this knowledge will likely by useful if you ever use another Linux based system.

GPIO Driver Architecture

As discussed in Chapter 2 (Kernel Space vs User Space), the Linux Kernel *owns* all the hardware resources. Users must request access – or call Kernel functions – to gain access to these resources. Such is the case with GPIO. The Linux GPIO driver architecture provides a standard methodology for accessing and reading/writing the available GPIO pins in the system.

When a User request is granted by the Linux Kernel, it will become available under the Sysfs ("system file system") directory in the Linux filesystem. Specifically, it will be found in:

```
/sys/class/gpio
```

Also mentioned back in Chapter 2 (Virtual Files), that while its obvious that a GPIO pin is not a real file, upon request, Linux will create a virtual file interface for each available GPIO pin that you can read or write. For example, writing a "1" to this virtual file will set the associated GPIO pin "on" (i.e. high voltage).

GPIO Control Interface

The Linux GPIO driver provides two functions for controlling User access to a GPIO pin:

`export` Request kernel to export control of GPIO pin to userspace

`unexport` "Return" control of GPIO pin back to kernel

Note that once you "export" a pin, unless you export it, it will be available until you power-down or reset the device. In other words, it's common to "export" the pins you need during your programs system initialization code.

GPIO Signal Interface

Once exported, access to GPIO pins are found along the `/sys/class/gpio` path. For example, gpio38 would be found in the filesystem at:

```
/sys/class/gpio/gpio38
```

Even further, working with each GPIO is structured around a standard set of read/write attributes.

`direction` Defines which direction ("in" or "out") should be assigned to a given pin.

Note that the the SK2 hardware is fixed for the GPIO pins being discussed; in other words, you cannot change how the hardware works by using this signal, rather, you must apply the value as assigned by the hardware (i.e. "out" for LED, "in" for button).

`value` Represents the "value" of the pin. For *input* direction, reads as either a 0 (low) or 1 (high). When used as an *output*, writing a 0 sets the pin "low", while any non-zero value sets it "high".

LED Examples

Considering the previous discussions for the SK2 GPIO assignments and the Linux GPIO driver architecture, the following shell code example turns “on” the RED.

Listing (Appendix) 1 - # TurnOnRedLed.sh

```
# TurnOnRedLed.sh
#
# This simple example turns on the Red RGB LED
#
cd /sys/class/gpio

echo 38 > export
echo out > gpio38/direction

echo 1 > gpio38/value
```

Listing (Appendix) 2 - Turn off Red LED

```
# TurnOffRedLed.sh
#
# This simple example turns off the Red RGB LED
#
cd /sys/class/gpio

echo 38 > export
echo out > gpio38/direction

echo 0 > gpio38/value
```

Like the *export* command, the direction persists until the system is reset or powered off. Thus it can also be set once in your program’s initialization code.

The following *blink* example simply blinks the Blue LED five times.

Listing (Appendix) 3 - Blink the Blue LED five times

```
# blinkBlue.sh
#
# This is a simple example for blinking the Blue RGB LED five times
# - The For loop executes once per character (a thru e)
# - 'sleep 1' causes the cpu to wait 1 second
# - At the end of the script, the LED is turned off
#
cd /sys/class/gpio

echo 22 > export
echo out > gpio22/direction

for var in a b c d e; do
    echo  0 > gpio22/value
    sleep 1
    echo  1 > gpio22/value
    sleep 1
done
echo 0 > gpio22/value
```

USER Button Examples

The USER button on the SK2 is like the LED example except that we set it up as an *input* and read the value from the virtual file.

Hint: The SK2 button hardware was designed so that the button is "up" by default and therefore reads as "1". When pressed down, the button will read as "0".

The first button example makes use of the Linux "cat" command. You may remember that this will read the contents of a file and write it to the standard output (i.e. your ADB shell terminal – refer to Chapter 2 for a listing of common Linux shell commands as well as how to use the ADB shell.)

The example also sleeps for 3 seconds after reading the switch, letting you change the state of the button – that is, pressing it down – before reading the button again.

Listing (Appendix) 4 - Reading USER Button

```
# userButtonCat.sh
#
# This is a simple example reads the value of the user button
# and outputs the value (using 'cat') to the command line.
# It then waits 3 seconds and does it again#
cd /sys/class/gpio

echo 23 > export
echo in > gpio23/direction
cat gpio23/value

sleep 3
# now hold-down the USER button and press up-arrow to repeat previous command
cat gpio23/value
```

The second button example reads the value of the button and turns on either the Red or Green LED.

Listing (Appendix) 5 - USER Button Lights Green or Red LED

```
# userButtonLed.sh
#
# This example reads the value of the user button and turns
# on either the Green or Red RGB LED depending upon the value
# of the user button
#

# Go to the GPIO directory
cd /sys/class/gpio

# Grant user-space access to all 3 LEDs and the USER button

    # USER button
    echo 23 > export
    echo in > gpio23/direction

    # Green LED
    echo 21 > export
    echo out > gpio21/direction

    # Blue LED
    echo 22 > export
    echo out > gpio22/direction

    # Red LED
    echo 38 > export
    echo out > gpio38/direction

# Turn off all three LEDs
echo 0 > gpio38/value
echo 0 > gpio22/value
echo 0 > gpio21/value

# Read the value of the USER push button into "val"
val=$(cat gpio23/value)
echo $val

# Turn on LED
if [ $val -gt 0 ]
then
    # If "up" turn on Green LED
    echo 1 > gpio21/value
else
    # If "down" turn on Red LED
    echo 1 > gpio38/value
fi
```

Further Reading

For further details, you may want to refer to:
https://www.kernel.org/doc/Documentation/gpio/sysfs.txt

More Details about the Linux Boot Sequence

Chapter 2 (Linux Boot Sequence) provided the simple answer to "How does the QuickStart boot automatically run at startup?" The QuickStart program (called iot_monitor) is started by the script

```
/CUSTAPP/custapp-postinit.sh
```

which is always run by Linux once the boot process has completed. This chapter provides further details for how this script gets run.

The following discussion is only background information, though, since the scripts leading up to `custapp-postinit.sh` reside in read-only memory on the SK2.

Getting to custapp-postinit.sh

We won't start at the beginning of the Linux boot sequence, but rather at the at the point where it runs a set of scripts found in the "/etc/rc5.d" directory.

```
/ # ls -l /etc/rc5.d/
lrwxrwxrwx    1 root     root            20 Oct 18  2017 S01networking
lrwxrwxrwx    1 root     root            20 Oct 18  2017 S15chgrp-diag
lrwxrwxrwx    1 root     root            20 Oct 18  2017 S20hwclock.sh
lrwxrwxrwx    1 root     root            16 Oct 18  2017 S20syslog
lrwxrwxrwx    1 root     root            30 Oct 18  2017 S25host-mode-preinit.sh
lrwxrwxrwx    1 root     root            24 Oct 18  2017 S29init_irsc_util
lrwxrwxrwx    1 root     root            17 Oct 18  2017 S30mssboot
lrwxrwxrwx    1 root     root            15 Oct 18  2017 S40qmuxd
lrwxrwxrwx    1 root     root            24 Oct 18  2017 S40thermal-engine
lrwxrwxrwx    1 root     root            17 Oct 18  2017 S45netmgrd
lrwxrwxrwx    1 root     root            29 Oct 18  2017 S45qmi_shutdown_modemd
lrwxrwxrwx    1 root     root            29 Oct 18  2017 S55reset_reboot_cookie
lrwxrwxrwx    1 root     root            33 Oct 18  2017 S90start_subsystem_ramdump
lrwxrwxrwx    1 root     root            30 Oct 18  2017 S91start_shortcut_fe_le
lrwxrwxrwx    1 root     root            19 Oct 18  2017 S97data-init
lrwxrwxrwx    1 root     root            15 Oct 18  2017 S97qrngd
lrwxrwxrwx    1 root     root            21 Oct 18  2017 S98misc-daemon
lrwxrwxrwx    1 root     root            28 Oct 18  2017 S999wnc-init-dlmode.sh
lrwxrwxrwx    1 root     root            20 Oct 18  2017 S99malmanager
lrwxrwxrwx    1 root     root            22 Oct 18  2017 S99power_config
lrwxrwxrwx    1 root     root            22 Oct 18  2017 S99rmnologin.sh
lrwxrwxrwx    1 root     root            23 Oct 18  2017 S99stop-bootlogd
lrwxrwxrwx    1 root     root            18 Oct 18  2017 S99wnc_fwup
lrwxrwxrwx    1 root     root            31 Oct 18  2017 S100host-mode-postinit.sh
```

Figure (Appendix) 5 - Listing /etc/rc5.d

Ideally a very skilled system administrator can modify these scripts files to change the system options and programs that get started at boot time. (But as we said above, on the SK2 these files reside in read-only memory and cannot be changed.)

During the boot process, Linux executes these files in numerical order. “S01” is executed first, followed by “S15”, then “S20”, and so on. The following diagram highlights this sequence.

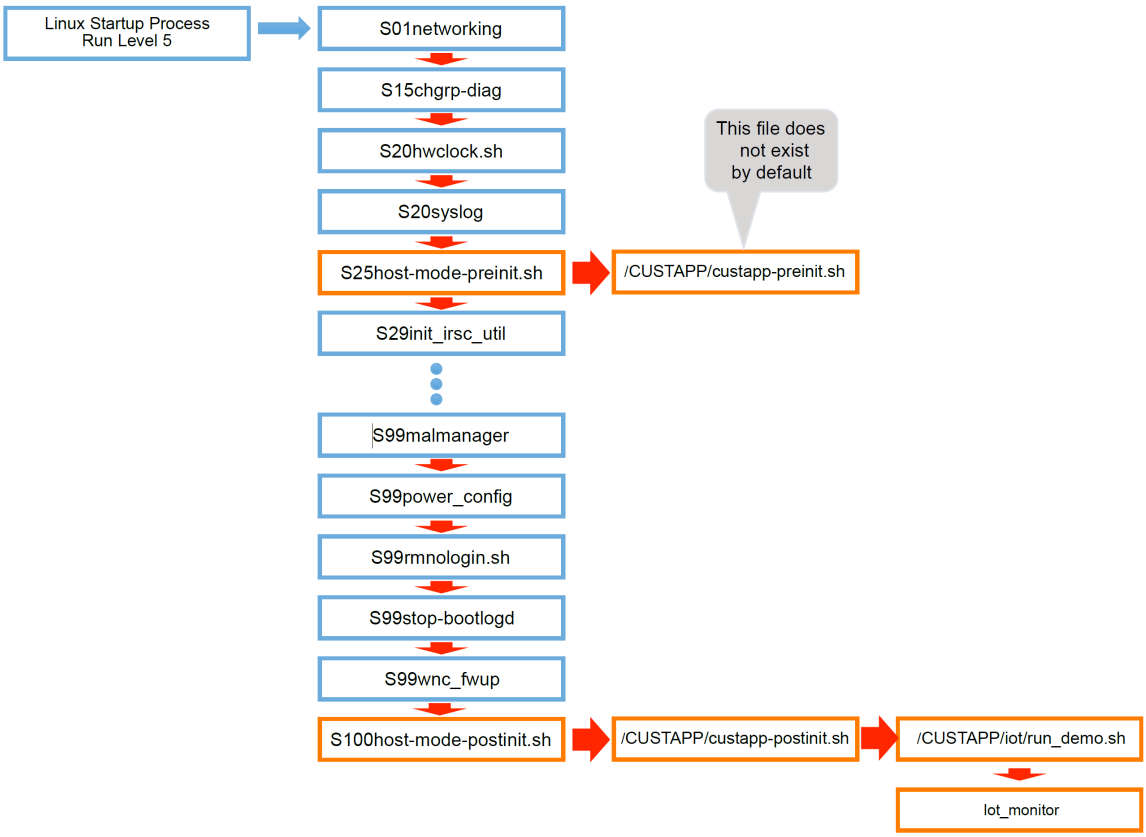


Figure (Appendix) 6 - Linux Boot Sequence - running files from /etc/rc5.d

Note that `S25host-mode-preinit.sh` executes 4th – or better put, very early in the boot process – while `S100host-mode-postinit.sh` executes last or very late in the process.

Viewing the contents of `/etc/rc5.d/S25host-mode-preinit.sh` we see:

Listing (Appendix) 6 - /etc/rc5.d/S25host-mode-preinit.sh

```
#!/bin/sh
if [ -e /data/custapp-preinit.sh ]; then
chmod +x /data/custapp-preinit.sh
. /data/custapp-preinit.sh
fi
```

Looking at this file, we see that the `S25host-mode-preinit.sh` file calls `/data/custapp-preinit.sh`, and this is started early in the process.

From Chapter 2, remember `/data` is a link to `/CUSTAPP`. Examining that directory:

```
/CUSTAPP # ls -la
total 1144
drwxrwxrwx 6 122 129 1128 Feb 8 20:09 .
drwxr-xr-x 24 root root 2048 Feb 7 20:34 ..
-rw-r--r-- 1 root root 170378 Feb 8 20:38 all.log
-rwxrwxrwx 1 root root 0 Feb 7 19:51 custapp-postinit.sh
drwxr-xr-x 2 root root 304 Jan 1 1970 fwup
drwxrwxrwx 2 root root 304 Feb 6 22:39 iot
drw-r--r-- 2 root root 312 Jan 3 1970 psm
lrwxrwxrwx 1 root root 11 Jan 1 1970 upload -> /mnt/upload
drwxr-xr-x 5 root root 592 Jan 2 1970 user
```

we find there is not a file called `custapp-preinit.sh`. That is okay, though, since the script tests for the existence of the file before executing it (using the "-e" option).

Further down the list of /etc/rc5.d files is a second script `S100host-mode-postinit.sh` that executes after all the board "services" are set up and running. This script:

Listing (Appendix) 7 - /etc/rc5.d/S100host-mode-postinit.sh

```
#!/bin/sh
if [ -e /data/custapp-postinit.sh ]; then
chmod +x /data/custapp-postinit.sh
. /data/custapp-postinit.sh
fi
```

also tests for and executes a program in the `/data` (i.e. `/CUSTAPP`) directory called `custapp-postinit.sh`. This is the same file we discussed in Chapter 2 (How Does the QuickStart Run). Examining it we find:

Listing (Appendix) 8 - /CUSTAPP/custapp-postinit.sh

```
start-stop-daemon -S -b -x /CUSTAPP/iot/run_demo.sh
```

Following the chain to `/CUSTAPP/iot/run_demo.sh` file, we find that it finally points to the QuickStart demo executable, called `iot_monitor`.

Listing (Appendix) 9 - run_demo.sh

```
iot_monitor -q5 -a a2e26b03f4e77aab23dbc5294b277d69
```

That's quite a long process to start the "iot_monitor" QuickStart demo, but this chain of events provides a great deal of flexibility for the Linux operating system. In our case, we can create/edit the two init scripts in the /CUSTAPP directory to kick off programs early or late in the boot process, as needed:

```
custapp-preinit.sh
custapp-postinit.sh
```

This process was explored, for `custapp-postinit.sh`, in Chapter 2.

Page left blank.