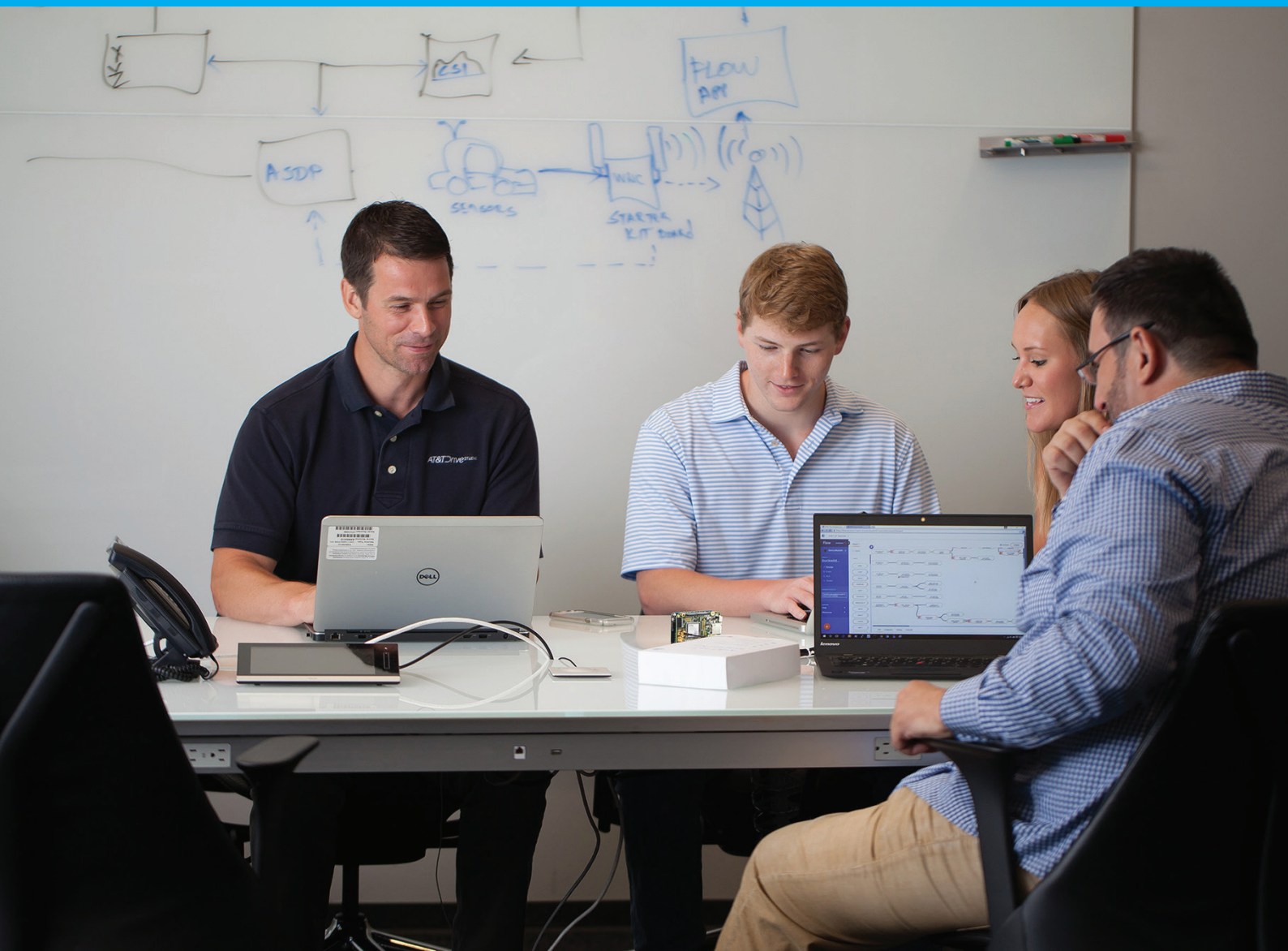
AT&T IoT Starter Kit
Getting Started Guide



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 0.60 (Monday, May 13, 2019)

Table of Contents

Table of Code Listings

License for Code Examples

Preface

Work in Progress

As of April 2019, this User's Guide is a work in progress. It currently includes the first six chapters. With the final two planned for 2Q2019.

The next section highlights the 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.

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 a 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 Input/Output) 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.

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 network. 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 cURL and 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 from 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.

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

---

Chapter 7 - Using the ADC with the Light Sensor

The ADC (analogue to digital converter) is a useful peripheral interface found on the SK2's Linux processor. This port converts real-world analog 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

Hardware

- Windows, Mac, or Ubuntu computer with an available USB 2.0/3.0 port.
  - ***Ubuntu Linux computer*** is required if you want to compile C/C++ code to run on the SK2.

    A virtual Linux computer – for example, using VMware – will allow you to build C/C++ programs if you do not have a Linux computer. (See note at the beginning of Chapter 3.)
  - Windows or Mac computers will work fine if you are only planning on to connect to the SK2 to view files or execute pre-built examples. These operating systems also work well if you're only planning to write Python code.
- AT&T IoT Starter Kit (2nd Generation)

Software

The following list summarizes the software required for building and running code on the SK2.

- **Chapter 2:** Android Debug Bridge (ADB) is required to talk to the SK2 from your computer. Installation instructions are included in Chapter 2.
- **Chapter 3:** The Avnet/WNC SDK must be installed to build C/C++ programs. Additionally, the M18QxIotMonitor program must be installed if you want to rebuild the QuickStart demo. Chapter 3 shows how to install both – along with instructors for their use.
- **Chapter 4:** Python firmware image is required for Python coding in Chapter 4 (and later chapters):
- All Chapters: Command line (i.e. shell) is needed to execute ADB commands and run scripts. Linux (BASH shell) and Mac (Terminal) command-line tools work fine. Some Windows users have experienced problems with their standard command-line tools and may want to install an improved command-line tool, such as:
  - WSL (Windows Subsystem for Linux) https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux
  - CMDER (http://cmder.net/)

Note: The author used the standard command-line utilities for each platform in Chapter 1, then switched to CMDER (on Windows) for subsequent chapters.

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)
- Python M18Qx LTE IoT API Guide (PDF)
- Python M18Qx Peripheral IoT API Guide (PDF)

Software / Repositories

Note that installation & usage instructions are provided in their associated chapters of this user guide.

- Avnet WNC SDK: This repository contains the software development kit (SDK) libraries and documentation for use with the the IoT Starter Kit (2nd generation) (also known as 'SK2').

  https://github.com/Avnet/AvnetWNCSDK

- Avnet IoT Monitor Example: The monitor source code for the (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 IoT Starter Kit (2nd Generation) Python Firmware Image

  https://s3-us-west-1.amazonaws.com/td-marketplace-assets/SDK-01/M18Q2_v12.09.182151_APSS_OE_v01.07.183121.zip

  Warning: Do not install the Python firmware without reading the instructions in Chapter 4.

- AT&T GitHub Repository

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

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

Please refer to Avnet's Cloud Connect Kits page and scroll to review and download the certification documents.

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

Document Locations

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

Page left intentionally blank.

1. Getting Started

The chapter introduces IoT (Internet of Things) and the AT&T IoT Starter Kit (2nd Generation) – nicknamed the "SK2" – and then quickly gets you playing with the QuickStart demonstration program that comes pre-programmed into the kit.

After a brief review of IoT and the kit's hardware, you will need to register 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, whereupon the on-board LEDs indicate the various stages of the demo's execution.

Finally, we'll introduce the support and development resources available for the 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 be useful to cover them all, but here are 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: This is 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 (i.e. “thing”) is capturing data (location, temperature, etc.) and sending it to the cloud (i.e. “Internet”).

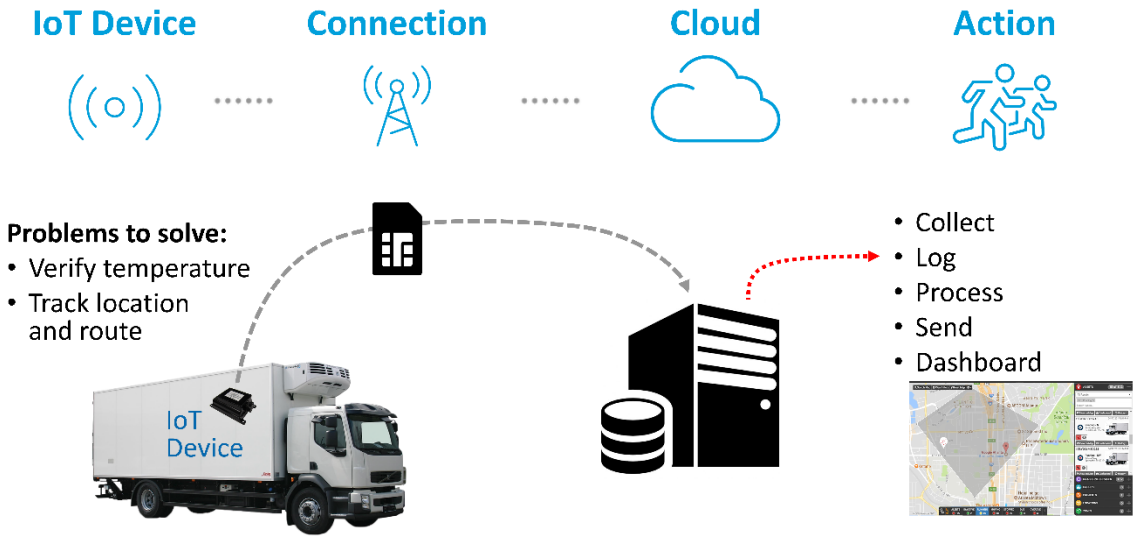


Figure 1.4 – How Does IoT Work?

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 is a popular method for many home or business applications. But a growing number of applications require connectivity over longer distances and with fewer configuration headaches. Cellular connections, as provided in the SK2, makes IoT easier and cheaper for most use cases.
3. **Cloud**: That is, the “Internet”, receives and processes the data sent from the device.
4. **Action**: While it’s true that the Action is generally handled by the Cloud, we’re explicitly noting it 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. The Case for Cellular IoT

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. The SK2: IoT Everywhere

With the AT&T IoT Starter Kit (2nd Generation), developers get the power of AT&Ts LTE network in a small, easy-to-develop module. A one-time configuration with the AT&T certified SIM (subscriber identity module) will ensure your device stays secure and ready to connect anywhere.

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.

The following discussion briefly introduces the top and bottom of the board. For additional technical details regarding the kit's hardware, please refer to Avnet's SK2 Hardware User Guide (PDF).

1.2.2.1. Top

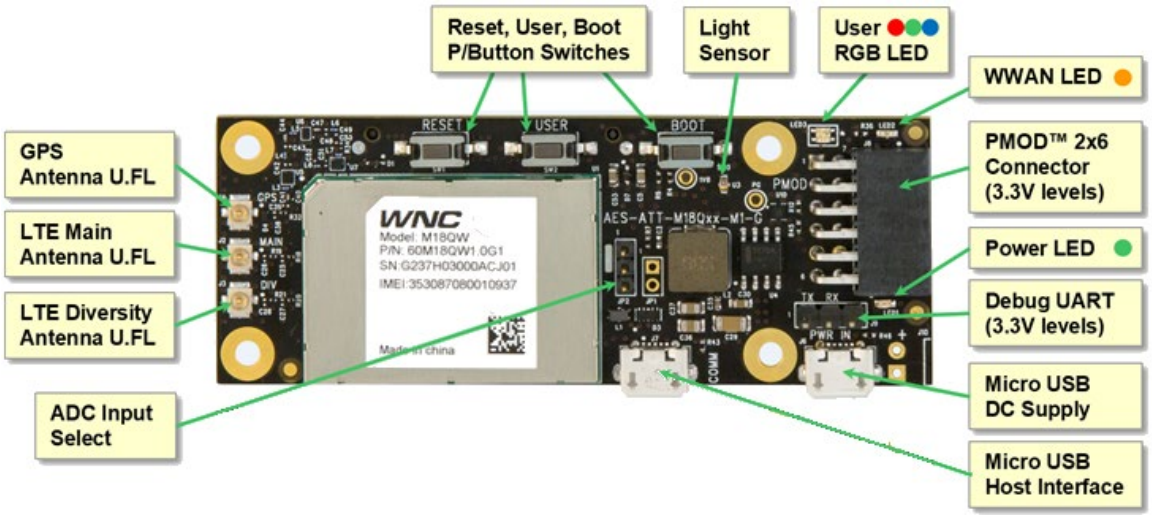


Figure 1.8 – Top of SK2 Module

The WNC modem/processor module sits on the left side of the board; this includes the Linux processor that will be programmed in later chapters.

Starting at the left side of Figure 1.8, other features include:

- **ADC Input Select:** 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.
- **Antenna Mounts (x3):** 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.
- **Three Push Button Switches:** There are three push-button switches on the board.
  - **Reset:** Initiate a reset by pressing and holding the button for greater than 3 seconds.
  - **User:** The USER BUTTON can be accessed by user programs. Programming this button via GPIO (general purpose input/output) is discussed in Chapters 3 and 4.
  - **Boot:** Forces USB boot for reprogramming the firmware. (This is an advanced function that is not covered in this book.)
- **Light Sensor:** The ADC (analog to digital convertor) in the WNC module measures the amount of light hitting this sensor.
- **RGB LED:** The User RGB (red, green, blue) LED can also be controlled via user software (also discussed in Chapters 3 and 4).

- **WWAN LED:** The WWAN (wireless wide-area network) LED is controlled by the LTE modem and provides status regarding the cellular connection. This can also be user programmed using SYSFS, which is discussed in Chapters 2-4.
- **PMOD connector:** This 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. (Digilent provides a handy PMOD reference.)
- **Power LED:** This LED illuminates green when the circuit is correctly powered. It is not user programmable.
- **Debug UART:** This port is dedicated for system debug output (i.e. Linux kernel debug log output) and is currently not programmable.
- **Two Micro USB connectors:**
  - **DC Power Supply:** USB connector requires 5V 2.5A DC power. (The kit includes the correct AC/DC power adaptor).
  - **Host USB:** 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 is discussed Chapter 2.)

1.2.2.2. Bottom

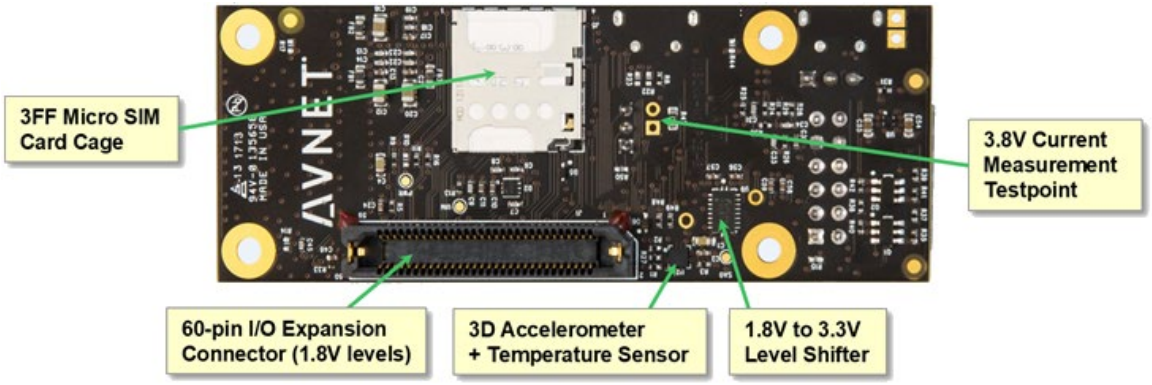


Figure 1.9 – Bottom of SK2 module

Starting from the left and working around the board:

- **3FF Micro SIM Card Cage:** Holds the micro SIM (Subscriber Identification Module) card that comes with the kit. Once registered, the SIM allows communication through AT&T's IoT cellular network. (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 modification, this unpopulated header can be used to measure the current flowing through 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'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).

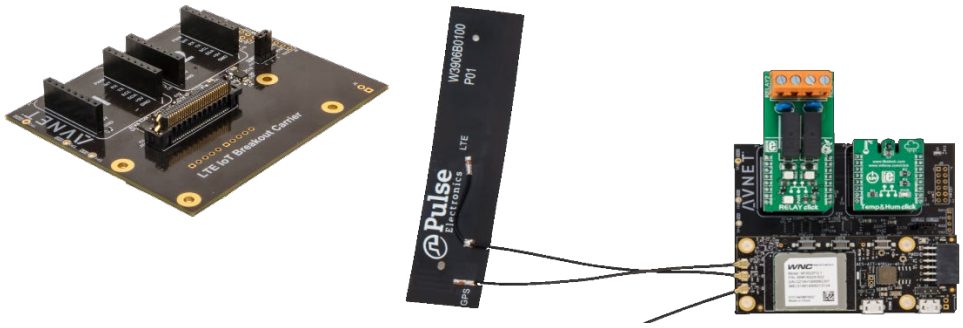


Figure 1.10 Avnet's LTE IoT Breakout Carrier

1.2.2.3. Block Diagram

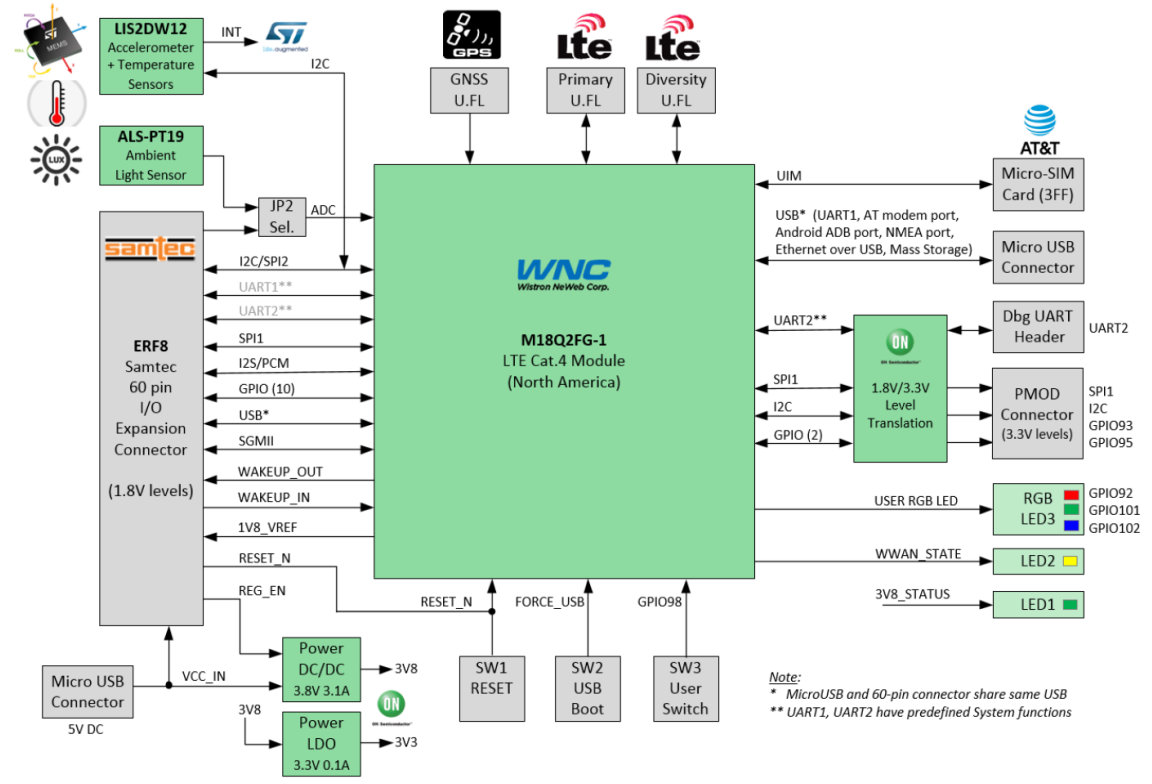


Figure 1.11

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

1.2.3. Software Overview

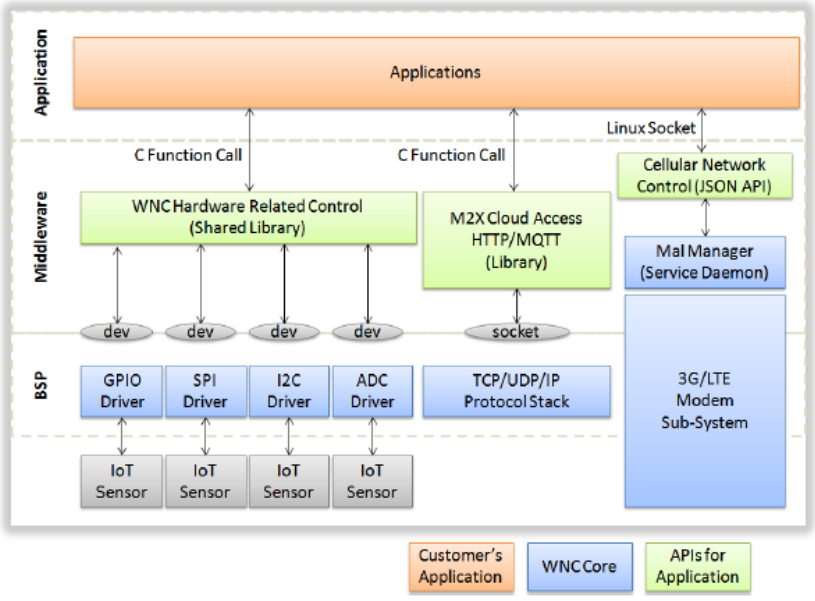
The SK2 uses the WNC M18Q2FG-1 module, whose ARM Cortex-A7 processor is pre-loaded with an OpenEmbedded version of Linux. User application code (scripts, C language, and Python) can run within this Linux environment.

You can program the Linux processor using one or more of three different languages:

1. **Linux shell scripts**: Useful for simple demos and examples.
2. **Python**: The optional AT&T Python environment lets you access the hardware using the popular scripting language. The Python environment is covered in Chapter 4.
3. **C/C++:** You build C applications that run within the Linux environment. In fact, the QuickStart demo - which comes pre-loaded on the SK2 - was written using C/C++. Chapter 3 describes how to setup the tools and build C/C++ programs.

The M18Q2FG-1 Software Development Kit (SDK) provides APIs that simplify access to system resources when programming with C/C++. The SDK consists of two main APIs (Application Programming Interfaces):

- **Cellular Network Control API**: Provides access to the communications services over the LTE cellular radio. This is described in the Avnet M18Qx LTE IoT API Guide and Python M18Qx LTE IoT API Guide.
- **Hardware Related Control API**: 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). This is documented in the Avnet M18Qx Peripheral IoT API Guide and Python M18Qx Peripheral IoT API Guide.



The Python firmware image for the SK2 includes these API by porting them into Python callable functions. Coding with Python is discussed in Chapter 4. (Please refer to the Python versions of the two API guides for specific details.)

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.

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.)

This initial program, called "IoT Monitor", will post readings from the SK2 sensors to the cloud. Linux has been configured to run it automatically upon powering up the SK2. In other words. the *QuickStart* demo is implemented as a C/C++ program called *iot_monitor*.

Further examination of the QuickStart demo includes:

- Chapter 1 (this chapter) examines the QuickStart demo from a user perspective; that is, "how do I run the demo?"
- Chapter 2 describes how the *iot_monitor* runs automatically by Linux when the SK2 boots up – and how you can stop it from running automatically, if you want.
- Chapter 3 examines the demo's the source code (i.e. the C/C++ *iot_monitor* program). It includes instructions for downloading and rebuilding it, after you have installed the Linux C/C++ tools.
- Chapter 5 explores the 'cloud' side of the QuickStart demo by examining the AT&T's M2X service. This includes a description for creating – and then sending – the QuickStart demo's data to your own M2X account.

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

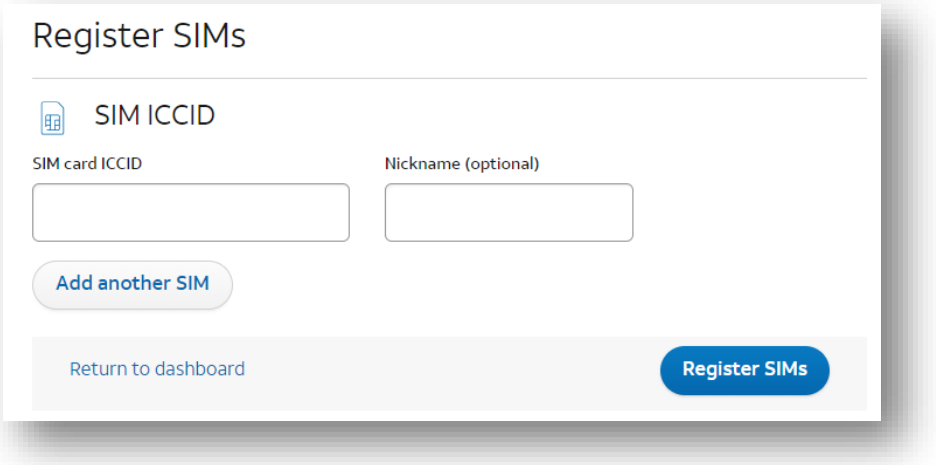
   You have choice of login methods - either using an account that you have previously setup, an AT&T Developer account, or a GitHub account. If you would like to create a new account, click the "Create an account" link near the bottom of the screen.

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:

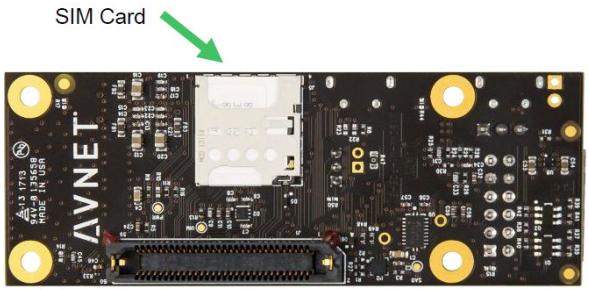


Figure 1.15

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:

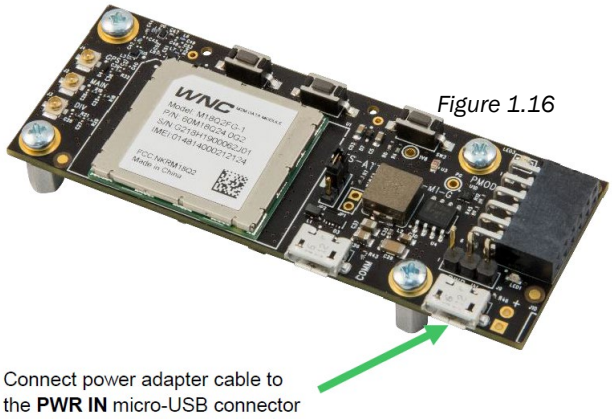


Figure 1.16

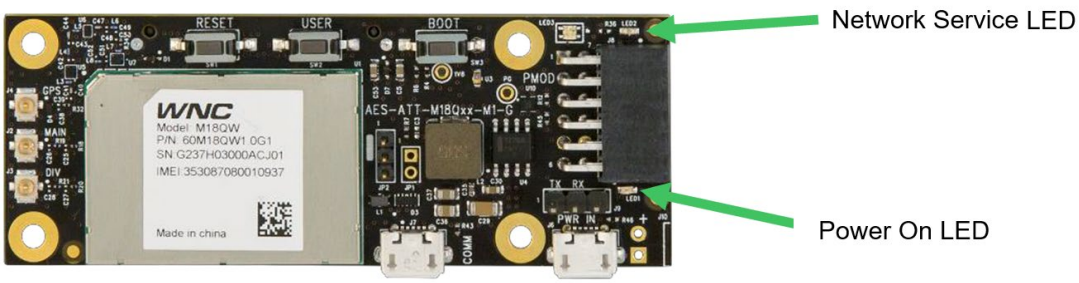
1.3.1.3. Running the QuickStart Demo

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. **Confirm successful power up.**

   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:



Hint: If the power is already on, power-cycle the board by disconnecting and then reconnecting the USB power supply cable.

Figure 1.17

5. **Confirm network connectivity.**

   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 yellow (WWAN) 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.

Use the following QuickStart demo description for completing the next step.

Quick Start Demo Sequence of Events

1. The QuickStart demo starts up with the Red LED lit, indicating that communication has not yet been established with the AT&T service. It becomes green when the service is ready.
2. While the tri-color LED is GREEN, the user may press the USER button to initiate a data transmission.
3. 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.
4. Once the USER push button is released, the tri-color LED will illuminate BLUE and the sensor data will be collected and sent to AT&T's IoT Platform (i.e. the AT&T M2X service). It will stay BLUE until all the data has been sent and acknowledged by M2X. *(Note that the Marketplace Dashboard, described later, pulls the information from M2X. M2X will be described in Chapter 5.)*
5. After the data transmission has 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:

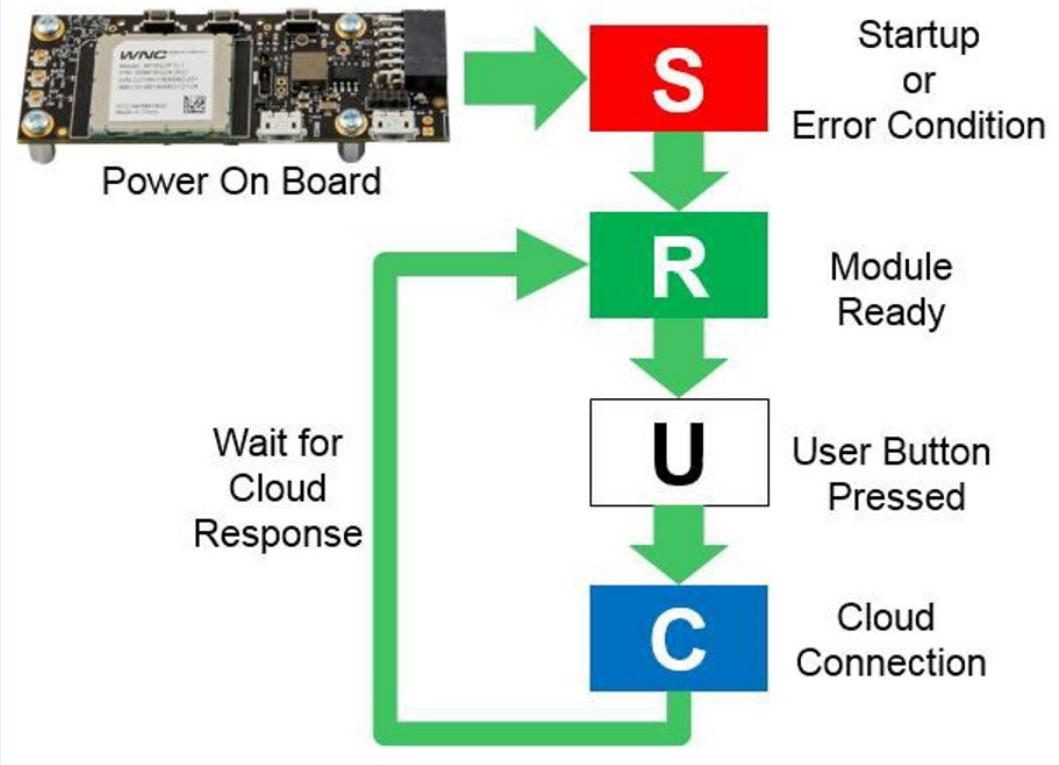


Figure 1.18 – QuickStart Demo Sequence – Tri-Color LED Chart

Stopping the Demo: 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 are 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. *(To permanently stop the demo, please refer to the Linux Boot section in Chapter 2.)*

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).

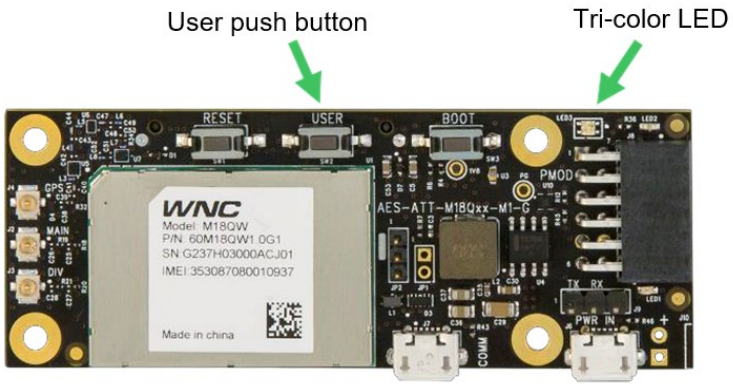


Figure 1.19

Hint:	The QuickStart demo's sequence of events are described on the previous page.

1.3.1.3.1. Sidebar – QuickStart Data Sent to the Cloud

The following sensor data is sent to AT&T M2X 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 at the AT&T M2X service and sends an HTTP response allowing the QuickStart demo to return to the green Ready state.

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.

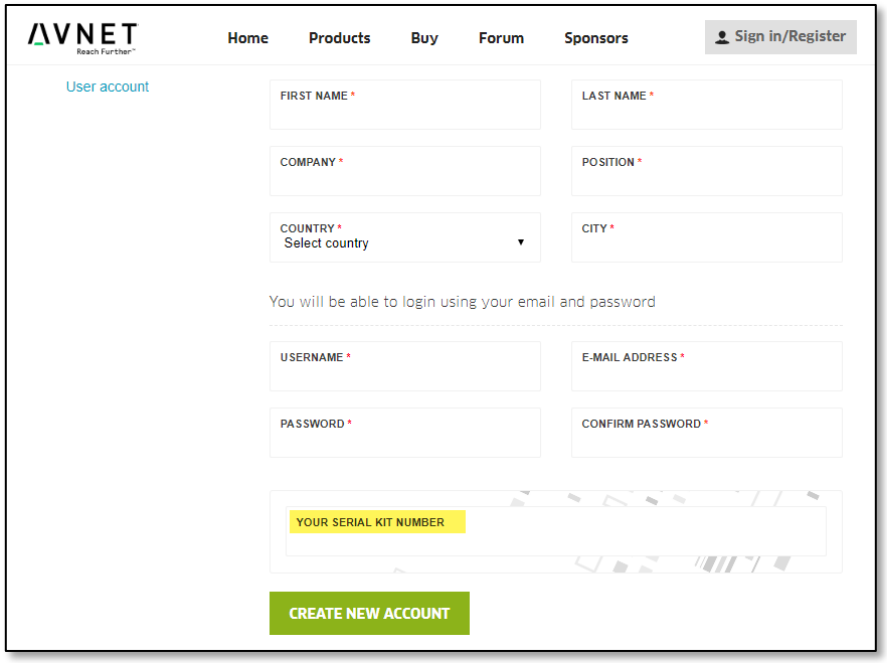


Figure 1.20

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 the preferred 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 (2nd Generation) (i.e. 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 head start with your development. If not, we think you will find this OS, and the kit itself, easy to use 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 does have 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 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.

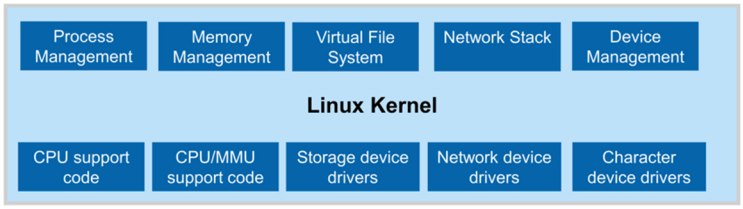


Figure 2.1 - Linux Kernel

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.

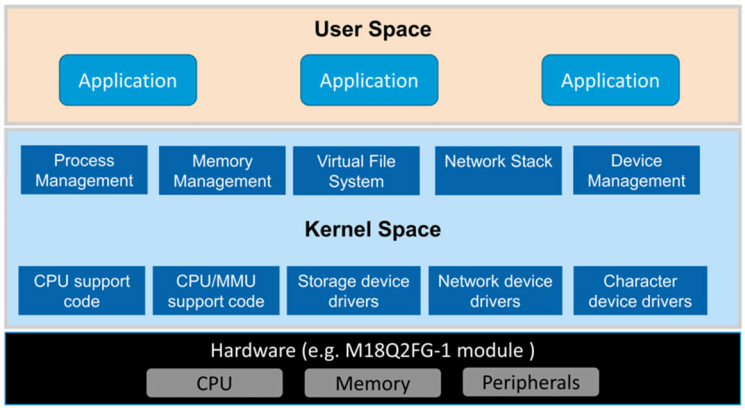


Figure 2.2 User Space vs Kernel Space

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:

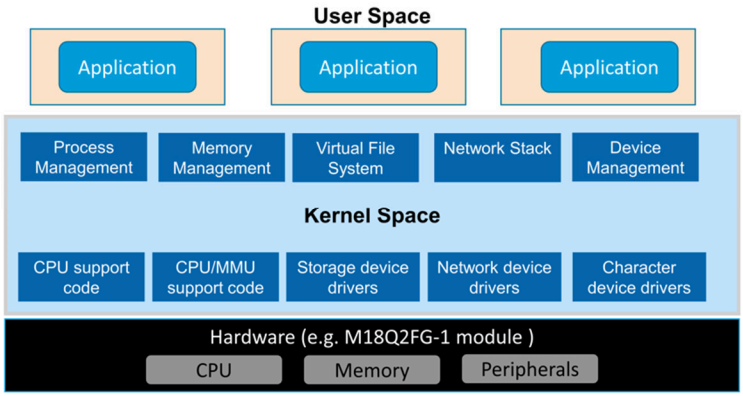


Figure 2.3 Separation of User Space

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.

Our board comes pre-loaded with the Linux distribution. When writing code for the SK2, 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 Hierarchy:

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

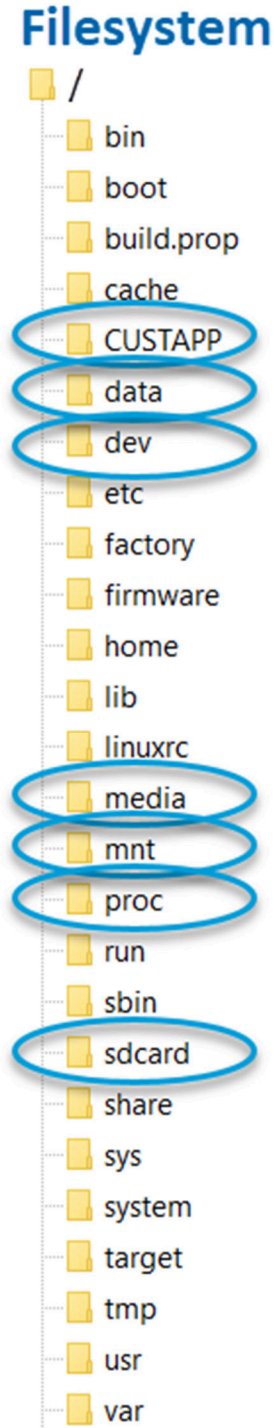


Figure 2.4 Filesystem

2.2.2.1. Virtual Files, Status Info, and Configuration

Extending the concept of the filesystem, Linux uses it for more than just text or binary files. Linux also 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" (small LS) Linux *list* command with the "-ls" (small LS) option. That is, the specific was: `ls -ls`

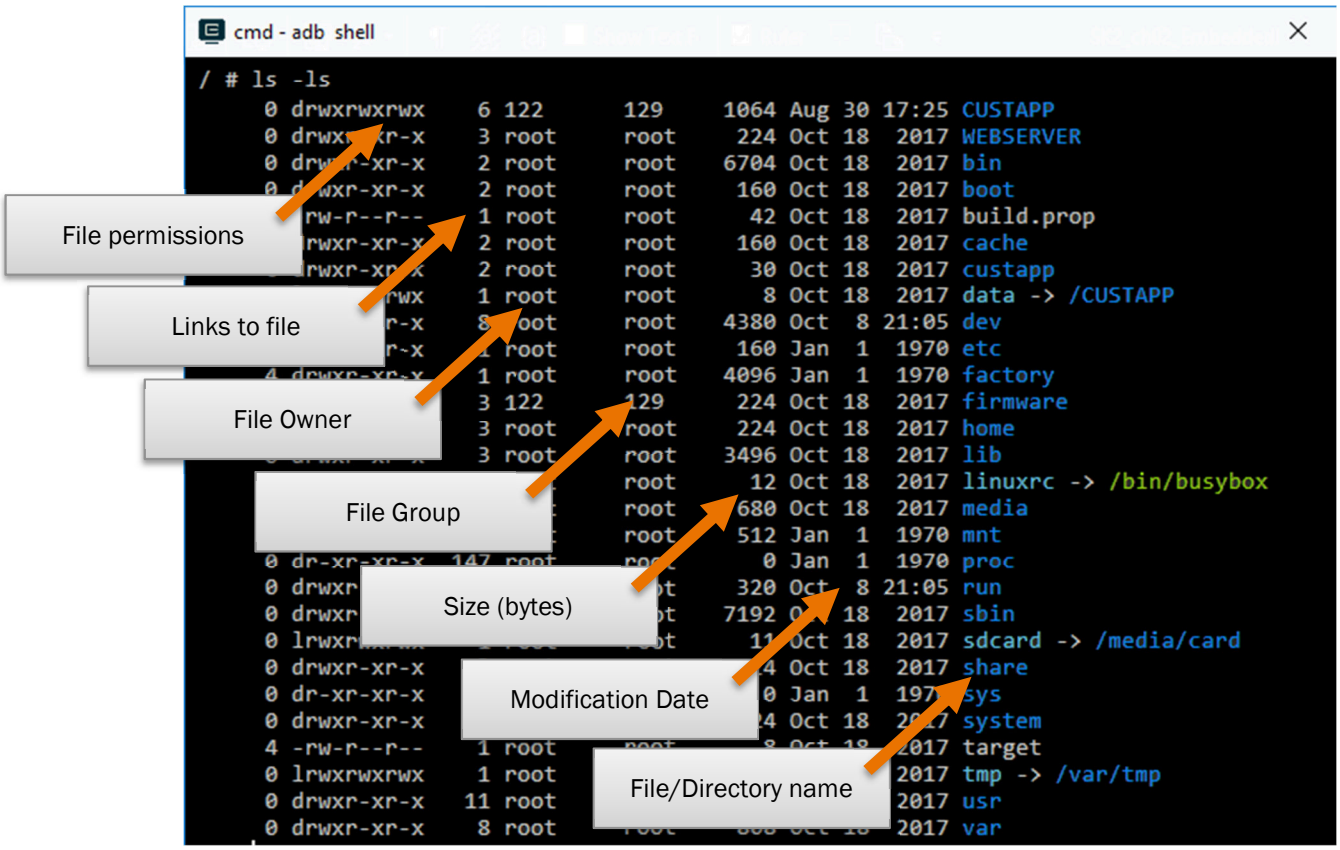


Figure 2.5 Filesystem Listing ("ls -ls")

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/or execute.

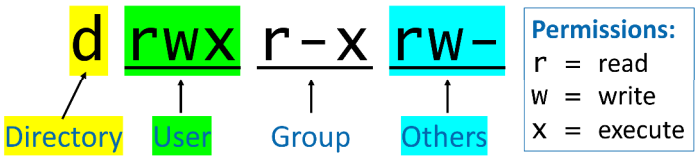


Figure 2.6 - File Permissions Explained

In this example, User can *read, write and execute*; the Group cannot *write*; and Others cannot *execute*.

Here's a handy reference to learn more about file permissions:
https://nitstorm.github.io/blog/understanding-linux-file-permissions

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"). In fact, the procedures in Section 2.52.4.3.3 will require executing `chmod` to enable *execute* permissions.

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

The SK2 relies on BusyBox to provide many of the command line utilities that we use every day.

```
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 2.7 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 execute 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 2.7 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 is an example of a very simple shell script that was used to create *Figure 2.5 Filesystem Listing ("ls -ls")*:

Listing 2.1: Chapter_02/list.sh

```
# 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
```

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

Listing 2.2 Chapter_02/list_dev.sh

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

Let's look at one final example:

Listing 2.3: Chapter_02/list_pipe_cat.sh

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

Later, in Section 2.4.3.3 on page 52, 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

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. **ADB Daemon** (SK2): Resident on your SK2, running in the background, providing development and debug access to authorized users through the COMmunications USB port.
2. **ADB Client** (Computer): The client runs on your development machine, letting you send commands. You can invoke a client from a command-line terminal by issuing ADB commands.
3. **ADB Server** (Computer): The server 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 SK2, 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 authorization key file to gainaccess 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

> **Hint:** If you are using a Debian-based Linux, such as Ubuntu, you can type the following command to download and install ADB (and skip the next step):
>
> ```
> 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 login id)*
   Linux: **`/home/MY_USER_NAME/Desktop/adb`** *(replacing MY_USER_NAME with your login id)*

> **Note:** Linux users can skip this step if ADB was installed 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`** *(replacing MY_USER_NAME with your login id)*
   Mac: **`/Users/MY_USER_NAME/.android`** *(replacing MY_USER_NAME with your login id)*
   Linux: **`/home/MY_USER_NAME/.android`** *(replacing MY_USER_NAME with your login id)*

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:

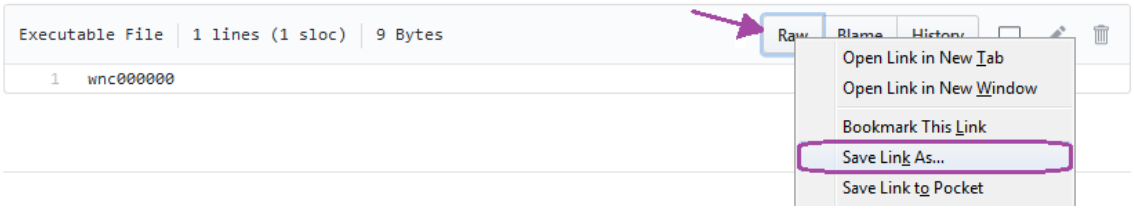
   ```
   wnc000000
   ```

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

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

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

   If downloading from github, we recommend right-clicking on the "Raw" button and select "Save Link As..."



   With this step complete, as an example, a Windows user named "doug" should have the following file on their computer:

   ```
   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 navigate to where you installed ADB (in step 2). For example:

   ```
   cd /home/MY_USER_NAME/Desktop/adb
   ```

   Note: Linux users who installed ADB via **`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: (Precede commands with "`./`")	`./adb devices`
Ubuntu Linux:	`sudo adb devices`

   In response, you should see:

   ```
   WNC_ADB device
   ```

   as shown below:

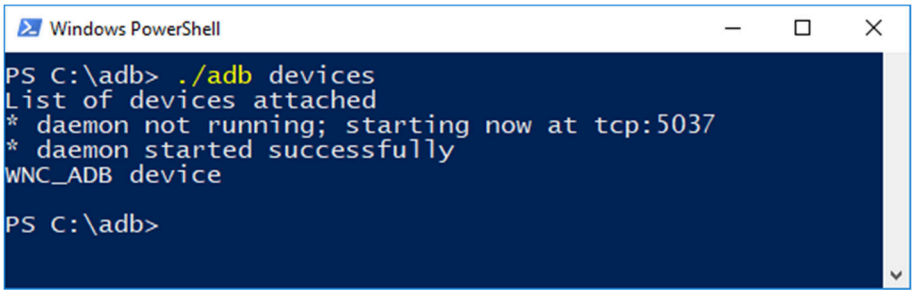


Figure 2.8 adb devices

   You can now run ADB commands (discussed in Section 2.4.3 on page 49) on your device.

Note: Refer to Appendix A5 for ***ADB Troubleshooting Tips***.

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.)
   Note that you can explicitly start the ADB server with the command: `adb start-server`
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 a key doesn’t exist, but it finds “adbkey.pub”, is uses that to create the actual authorization key “adbkey”.
4. Finally, it prints the list of devices to the command line (as seen in Figure 2.8).

2.4.3. ADB Commands

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).

2.4.3.1. 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.

2.4.3.2. Exploring the SK2 Filesystem using "adb shell"

Here are the results after executing the following instructions. (Note that this screen-capture was taken from a Windows computer running the CMDR shell.)

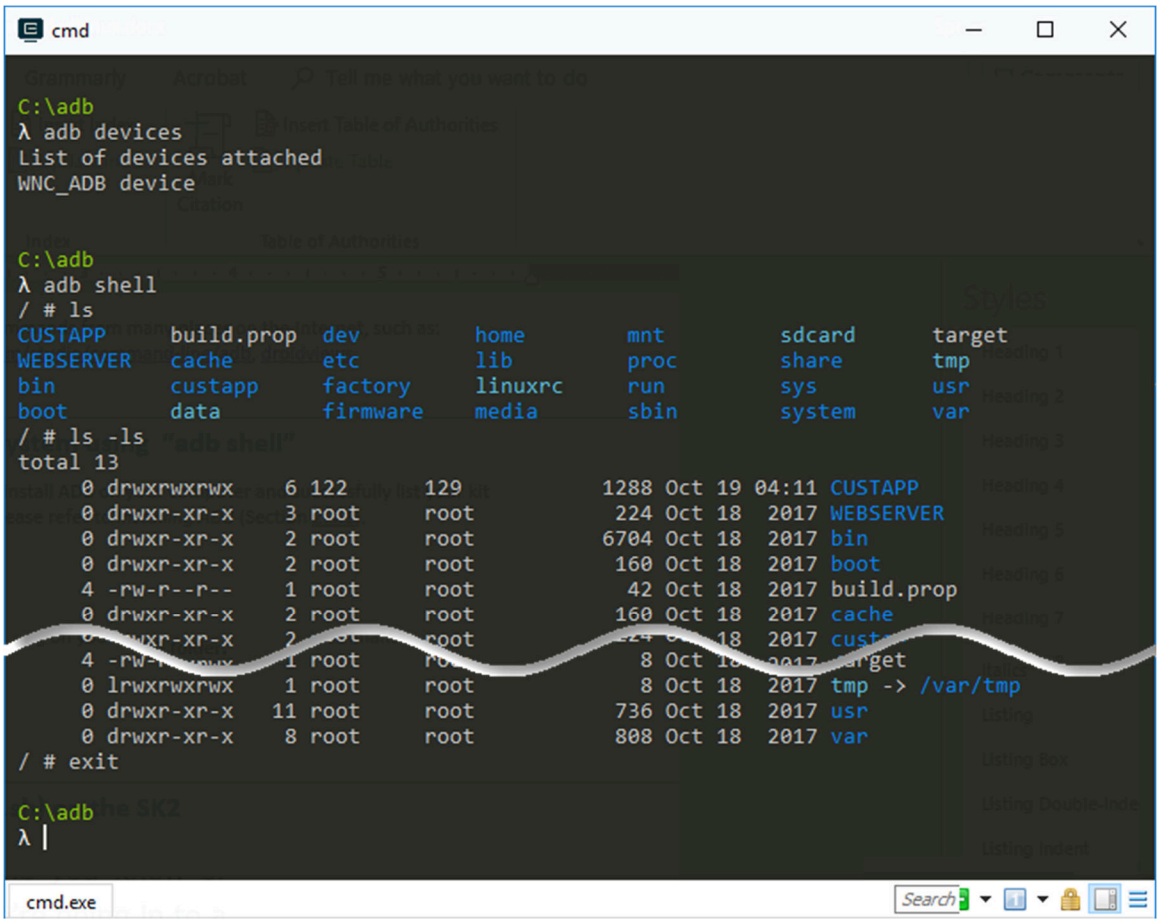


Figure 2.9 Using "adb shell"

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 (Appendix A5).

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 2.9 Using "adb shell"), that the command prompt changes to "#" after starting 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.

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

To demonstrate running shell scripts on your SK2, we will use the `list_dev.sh` script (Listing 2.2) from earlier in the chapter that listed 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's results 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 your 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
   ```

   If not, you will need to verify your ADB installation (Section 2.4.1) and/or do some ADB troubleshooting (Appendix A5).

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

   The ADB push command takes two arguments, the *from* and *to* locations:

   ```
        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   15 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. To prove this to yourself, 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 complete 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.

You can use the `source` command (see 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.

(Notice the "`/dev`" to the left of the "#" prompt.)

13. **Exit the ADB shell.**

```
exit
```

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

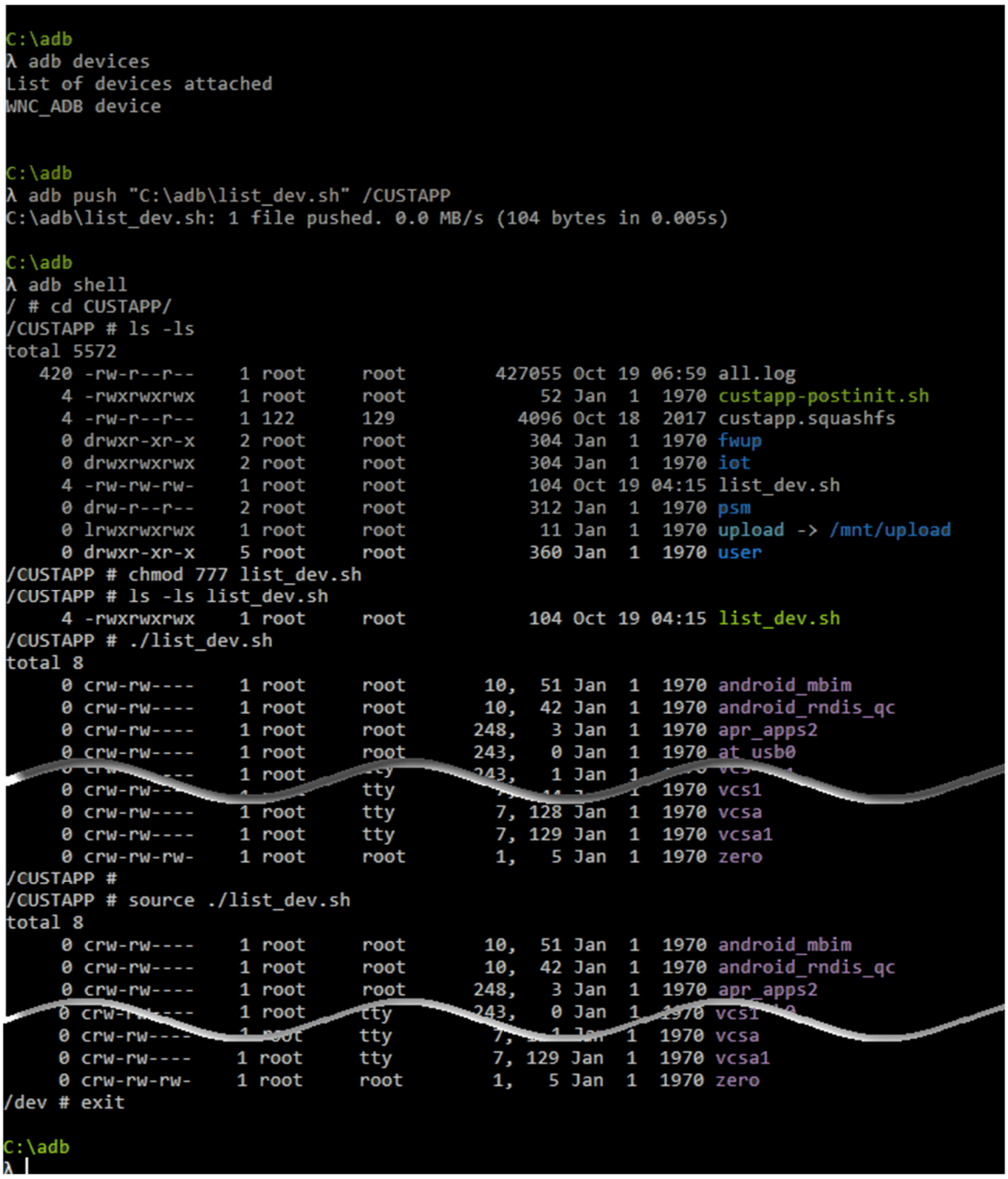


Figure 2.10 Running the list_dev.sh script

2.4.3.4. 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`) 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. **Edit the `list_dev.sh` file.**

   ```
   vi list_lib.sh
   ```

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

   ```
   type the "i" character
   ```

   b) **Change the cd command from "cd /dev" to "cd /lib".**

   Using the arrow keys, move the cursor to the "d" in "/dev".
   Press the "x" (or Delete) key three times (to erase the "dev").
   Press the right arrow key once (to move the cursor to the right of the "/").
   Type "lib".

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

   `<esc>` (i.e., press the Escape key)

   d) **Write the changed file and quit.**

   `:wq` (and press the Enter key)

7. **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.)

If the QuickStart demo is running, you'll need to (temporarily) turn it off by holding the USER Button down for longer than 3 seconds. After doing so, you will see the RGB LED turn off. (Ref: Section 2.6.2)

2.5.1.1. Writing the WWAN LED

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

Figure 2.11 Turning the WWAN LED off/on

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

Listing 2.4: Turn Off the LED

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

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

Listing 2.5: Turn On the LED

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

2.5.1.2. Simple Blink LED Script

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

Listing 2.6: Chapter_02/blink.sh

```
# 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
```

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).

Listing 2.7: Appendix/GPIO/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 2.8: Appendix/GPIO/TurnOffRedLed.sh

```
# 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
```

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?
- Additional 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/C++ program called `iot_monitor`. Its name derived from the fact that it monitors several sensors and then communicates that information over the Internet. (Throughout many chapters in this book we'll explore many facets of the `iot_monitor` program – starting with the C/C++ build tools in the next chapter.)

So how exactly does "iot_monitor" getting started?

1. The simple answer is that Linux always calls the following script file after at then end of its boot sequence:

   ```
   /CUSTAPP/custapp-postinit.sh
   ```

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

   Listing 2.9: 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 2.10: run_demo.sh

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

   Note that the `iot_monitor` command-line arguments are described further in Section 5.3.1.2.

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

Note: This is the simple, short answer. The longer, more involved answer is found in the Appendix Linux Boot Sequence discussion.

2.6.2. Configuring Linux Application Bootstrap

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 (from running automatically)

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 (Ref: Appendix A5).

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 that the command prompt changes to "#" after running the "adb shell" command. (Ref: *Figure 2.9 Using "adb shell"*), 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 "mv" (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.

2.6.2.2. 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.

1. **Reconnect to the ADB shell.**

   ```
   adb shell
   ```

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

   ```
   cd /CUSTAPP
   ```

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

   ```
   ./blink.sh
   ```

   The WWAN LED should blink 5 times.

4. **Create a new copy of the original `custapp-postinit.sh` file (which we renamed in the previous section).**

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

5. **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. After editing it should read:

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

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

   ```
   <esc>
   ```

   d) **Write the changed file and quit.**

   ```
   :wq
   ```

6. **Verify the file is correct.**

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

   Which should print out the contents:

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

7. **Exit the ADB remote shell.**

```
exit
```

8. **Reboot/power-cycle the board to view blink running.**

   Run the following to reboot the board:

```
sudo adb reboot
```

   And, it never hurts to power-cycle the board, too. Once the board reboots, the `blink.sh` script should start running after 10-15 seconds. When it runs, the WWAN LED will blink five times.

2.6.2.3. Autostart the QuickStart Demo

Assuming that you have completed the previous two sections (2.6.2.1 and 2.6.2.2) 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.

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

```
adb shell
cd /CUSTAPP
```

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

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

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

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

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

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

5. **Exit the shell.**

```
exit
```

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

   As an alternative to power-cycling the board, you could restart the SK2 by using this command:

```
adb reboot
```

   How ever you restart the SK2, 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).

Page left intentionally blank

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 nor Mac platforms are supported.

Unfortunately, it is not an uncommon for development tools supporting Linux-based embedded controllers to require that you do your development on a Linux hosted 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 partition 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 Ubuntu 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 Peripheral 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. 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
   ```

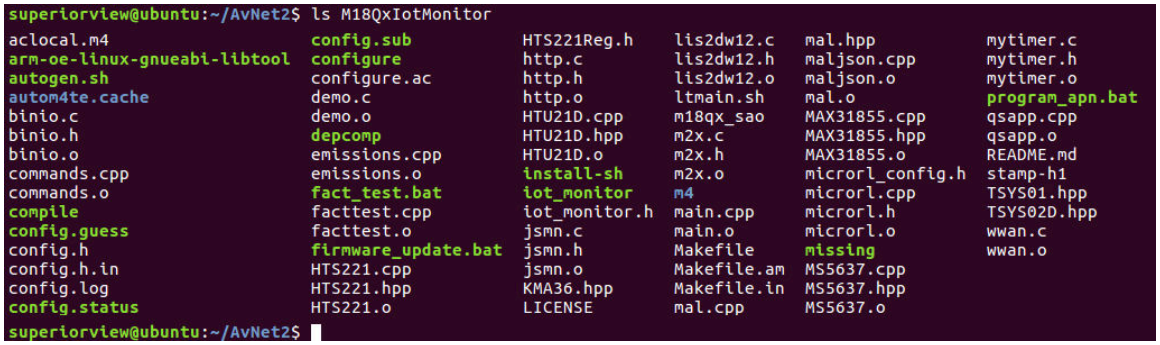


Figure 3.1 - Listing of M18QxIotMonitor

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 iot_monitor Application

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 Ubuntu Linux PC.

You use ADB to push the executable program to the SK2 and place it in the correct location. (This was discussed previously in Chapter 2.)

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

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

2. **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 Appendix A5 for ADB troubleshooting.)

3. **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 before starting to write and debug their own programs.

If your SK2 is running the out-of-box IoT Monitor program (i.e. QuickStart demo), 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 without completing the steps in Section 2.6).

4. **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).

Note: You do need to store your programs in CUSTAPP (or its subdirectories) because CUSTAPP is the only "writeable" directory on the SK2.

---

5. **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.

6. **Change to the CUSTAPP directory**

```
# cd /CUSTAPP
```

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

```
# ./iot_monitor
```

Not only will the program behave as when it was auto-started, 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 begin with the simple “Hello World” program so that we can focus on the steps to build and run a Linux application on the SK2.

Note: The directions in the rest of this chapter assume that the QuickStart demo has been temporarily or permanently disabled – was discussed in Section 3.2.3, Step 3 (on page 70).

3.3.1. Hello World

This program 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. This guide focuses on using the tools as provided by the WNC SDK.

When building your own application programs, 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.)*

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. (Later versions of *_1_autoname.sh* build this file for you.)
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. (Later versions of *_1_autoname.sh* build this file for you.)
Makefile.am	No	Specifies project subdirectories – again, you only change this file if you rename/modify the project's subdirectories. (Later versions of *_1_autoname.sh* build this file for you.)
_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. Install SK2 User Guide Examples

Hint: We recommend using git, but alternatively, you can download them in zip format from: https://github.com/att-iotstarterkits/sk2-Users-Guide/archive/master.zip

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 User Guide examples from GitHub.**

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

   This command creates a new subdirectory and downloads the GitHub project to it.

4. **View the contents of the new `sk2-Users_Guide` directory.**

   ```
   ls -ls sk2-Users-Guide
   ```

3.3.5. Building "Hello"

Here are the steps required to build the "hello" project. (Assuming you downloaded the examples as described in the previous section.)

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

   For example:

   ```
   cd ~/AvNet2/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.

3.3.5.1. .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. Example: Blink LED (File I/O)

Often called the embedded processing version of "Hello World", blinking an LED is one of the fundamental starting points for embedded systems. Once you can blink and LED, you've learned how to build a simple program that can talk to real hardware.

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_with_deinit`.

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 peripherals 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 Peripheral 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. Example: Blink LED (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_with_deinit Example

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

```
1 #include <stdint.h>
2 #include <stdio.h>                                          // Needed for printf()
3 #include <unistd.h>                                         // Needed for usleep()
4 #include <stdlib.h>                                         // Needed for exit()
5 #include <hwlib/hwlib.h>                                    // Needed for WNCSDK GPIO API
6
7 #define LED_RED         GPIO_PIN_92
8 #define LED_GREEN       GPIO_PIN_101
9 #define LED_BLUE        GPIO_PIN_102
10 #define USER_BUTTON     GPIO_PIN_98
11
12 #define SECOND          1000000
13 #define NUM_BLINKS      3
14
15 int main(void) {
16     int i = 0;                                           // loop counter
17     int ret;                                             // return value
18     gpio_handle_t   myGpio;                              // GPIO pin handle
19
20     printf("Hello Gpio\n");
21
22     // Initialize GPIO for Red LED
23     ret = gpio_init( LED_RED, &myGpio );                 // GPIO call to initialize RED LED
24     if (ret != 0) {                                      // gpio_init() returns '0' upon successful c
25         printf("ABORT: Could not initialize Red LED (ret = %d)\n", ret);
26         exit(1);
27     }
28     gpio_dir(myGpio, GPIO_DIR_OUTPUT);                   // Set direction of GPIO pin to 'output'
29
30     // Blink LED NUM_BLINKS times
31     for ( i = 0; i < NUM_BLINKS; i++ )
32     {
33         gpio_write( myGpio, GPIO_LEVEL_HIGH );           // Turn on the Red LED
34         usleep( 1 * SECOND );                            // Wait 1 second
35
36         gpio_write( myGpio, GPIO_LEVEL_LOW );            // Turn off the Red LED
37         usleep( 1 * SECOND );                            // Wait 1 second
38     }
39
40     gpio_deinit( &myGpio );                              // Release the GPIO resource
41
42     return 0;
43 }
```

Notice the 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 5 – you must #include the <hwlib/hwlib.h> library when using the WNC SDK hardware API.
Line 23 – gpio_init() allocates the red GPIO_PIN_98 and assigns the resources to the myGPIO handle.
Line 28 – the pin resource pointed to by the myGPIO handle is configured as an output.
Line 33 and **36** – gpio_write() is used to send a high, then low, level to the pin specified by myGPIO.
Line 40 – deallocates red LED resource (i.e. GPIO_PIN_98).

Hint:	It's important to deallocate hardware resources at the end of a program, such as releasing the LED GPIO pin on line 40 of this program. This topic is discussed further in Appendix A3, under *Deallocating GPIO Resources*.

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.

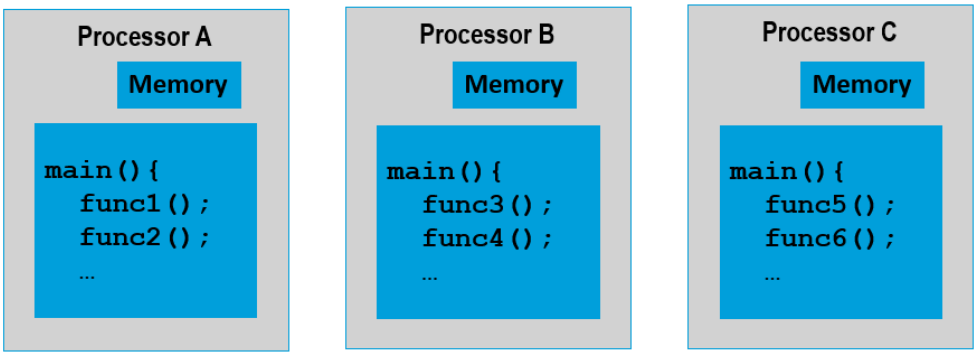


Figure 3.3 - Multi-processing

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

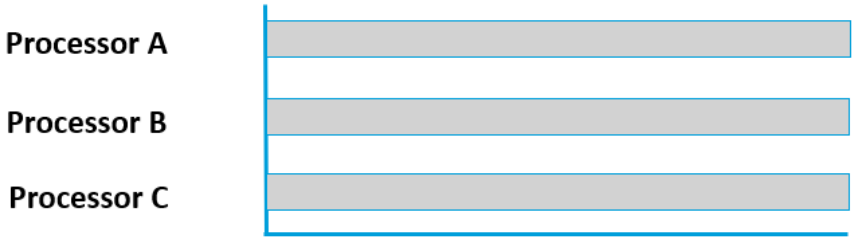


Figure 3.4 - Multi-processing Graph

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.

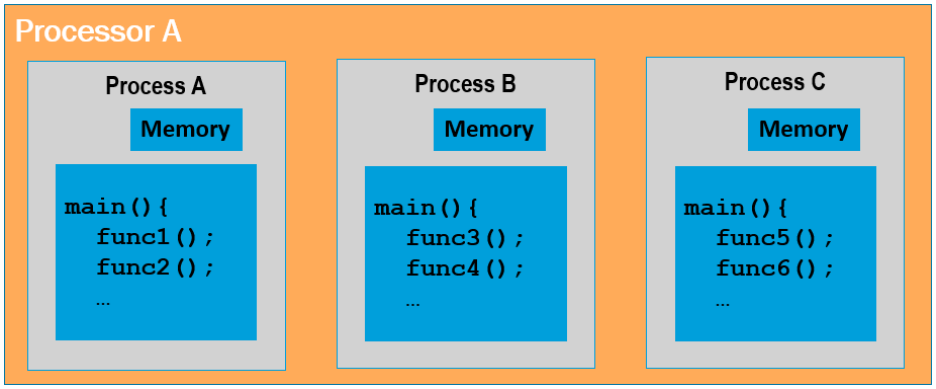


Figure 3.5 - Multiple Processes

Well, let's clarify that last statement. While three different programs can be configured to run simultaneously, 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.

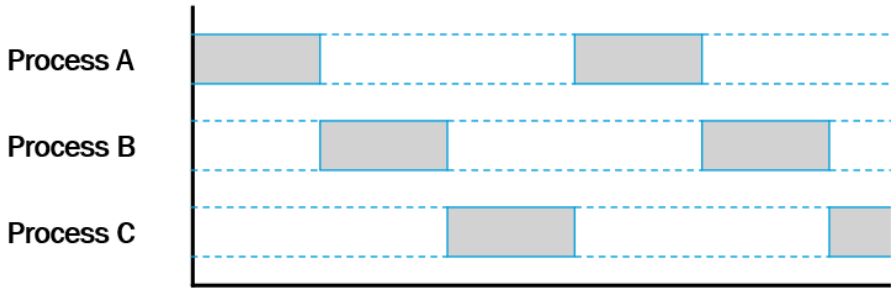


Figure 3.6 - Multiple Processes Graph

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.

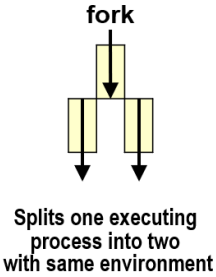


Figure 3.7 - Fork

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.)

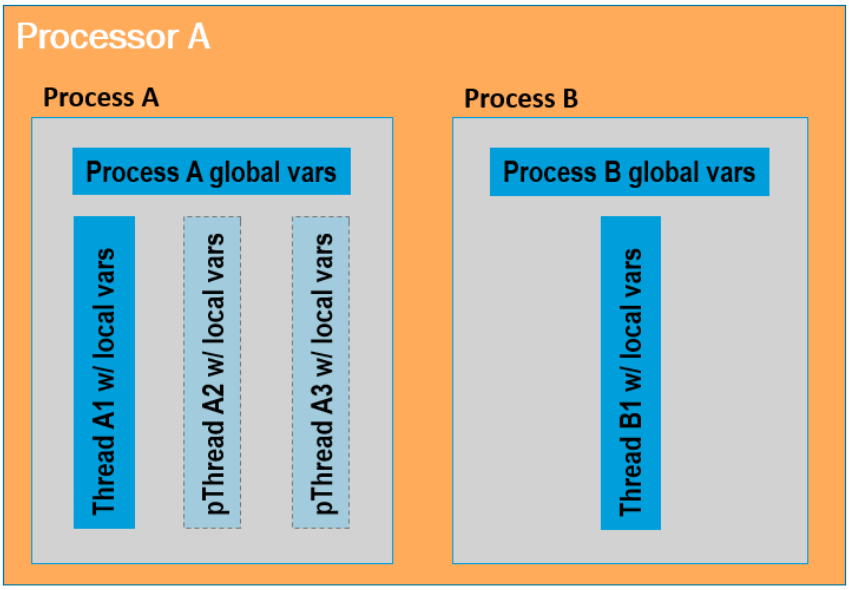


Figure 3.8 - Multiple Threads in a Process

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 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 appropriate unsafe 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:

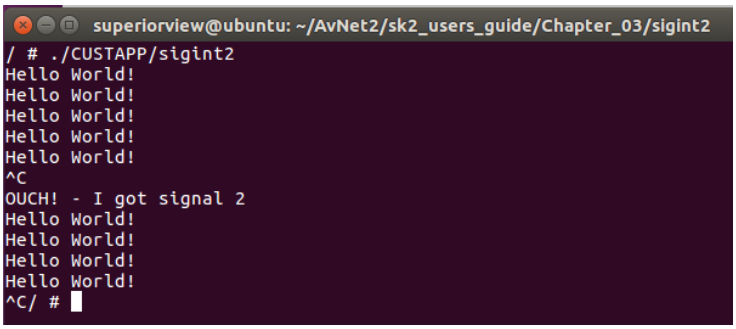


Figure 3.9 - Running sigint2

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:

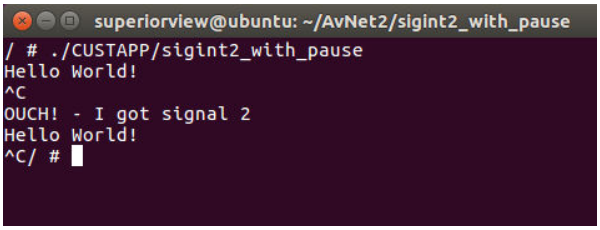


Figure 3.10 - Running sigint2_with_pause

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).

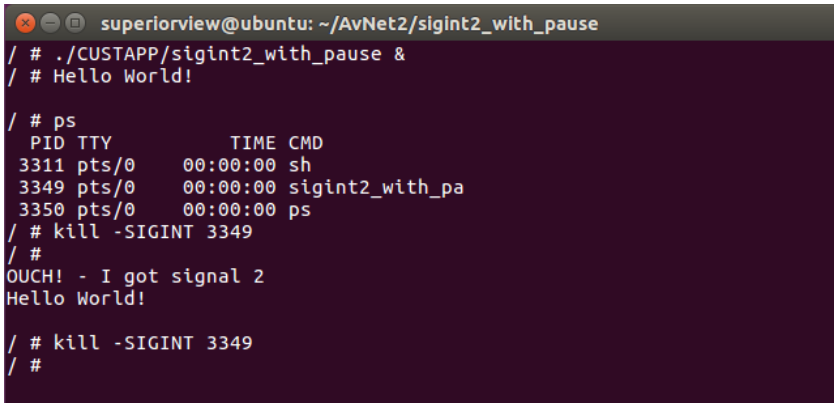


Figure 3.11 - Running sigint2_with_pause in separate process

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. Example: myGpio.c

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

```
    ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0----+----1----+----2----+---
 1  #include <stdint.h>
 2  #include <signal.h>                                                  // Needed for signal()/sigaction()
 3  #include <stdlib.h>                                                  // Needed for exit()
 4  #include <stdio.h>                                                   // Needed for printf()
 5  #include <unistd.h>                                                  // Needed for usleep() and pause()
 6  #include <hwlib/hwlib.h>                                             // Needed for WNCSDK GPIO API
 7  #include "myGpio.h"
 8
 9  #define SECOND      1000000
10  #define NUM_LEDS    3
11
12  int doExit = 1;
13
14  // ****  SIGINT Handler for Control-C ****************************************
15  void my_handler(sig_t s)
16 {
17      doExit = 0;
18      printf("\nCaught signal (%d). Quitting the program...\n",s);
19  }
20
21  // ****  main  ***************************************************************
22  int main(void)
23 {
24      int i = 0;                                                       // Loop counter
25      int r = 0;                                                       // Return value
26
27      printf("\nWelcome to the myGpio example!\n");
28
29      signal (SIGINT, my_handler);                                     // Register SIGINT signal handler
30
31      r = myGpio_init();                                               // Allocate and initialize the GPIO resources
32      if (r)
33          printf("Not all requested GPIO resources were allocated.\n");
34
35      gpio_read(myGpio[3].hndl, &myGpio[3].val);                       // Read and output the value of the User Button
36      printf("Checking on the button's value. It is %d.\n", myGpio[3].val);
37
38      printf("\nLED's will blink in color sequence until User Button is pushed.\n");
39      printf("Alternatively, you can also press Cntrl-C to quit the program.\n");
40      while(doExit)
41      {
42          // Blink all LEDs in sequence
43          for ( i = 0; i < NUM_LEDS; i++ )
44          {
45              gpio_write( myGpio[i].hndl, GPIO_LEVEL_HIGH );           // Turn on LED
46              usleep( 1 * SECOND );                                    // Wait 1 second
47
48              gpio_write( myGpio[i].hndl, GPIO_LEVEL_LOW );            // Turn off LED
49              usleep( 1 * SECOND );                                    // Wait 1 second
50          }
51      }
52
53      myGpio_close();                                                  // Release the GPIO resources
54
55      return 0;
56  }
```

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

```
#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 Peripheral 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 proceeding with the next section.

4.1.2. Setting up the Python IDE

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 Fastboot 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.

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

11. **Change to the directory where you extracted the SK2 Python firmware 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.

    For example:

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

12. **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.

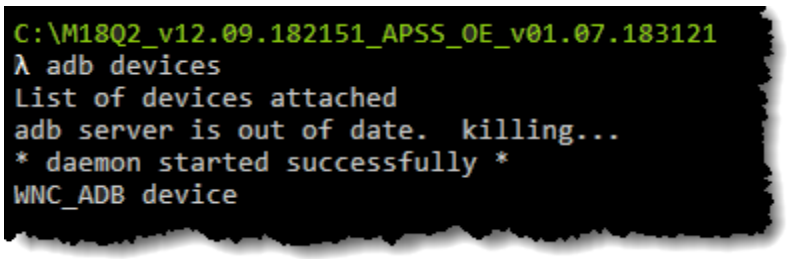


Figure 4.1 - Restarting ADB with "adb devices"

13. **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.

14. **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

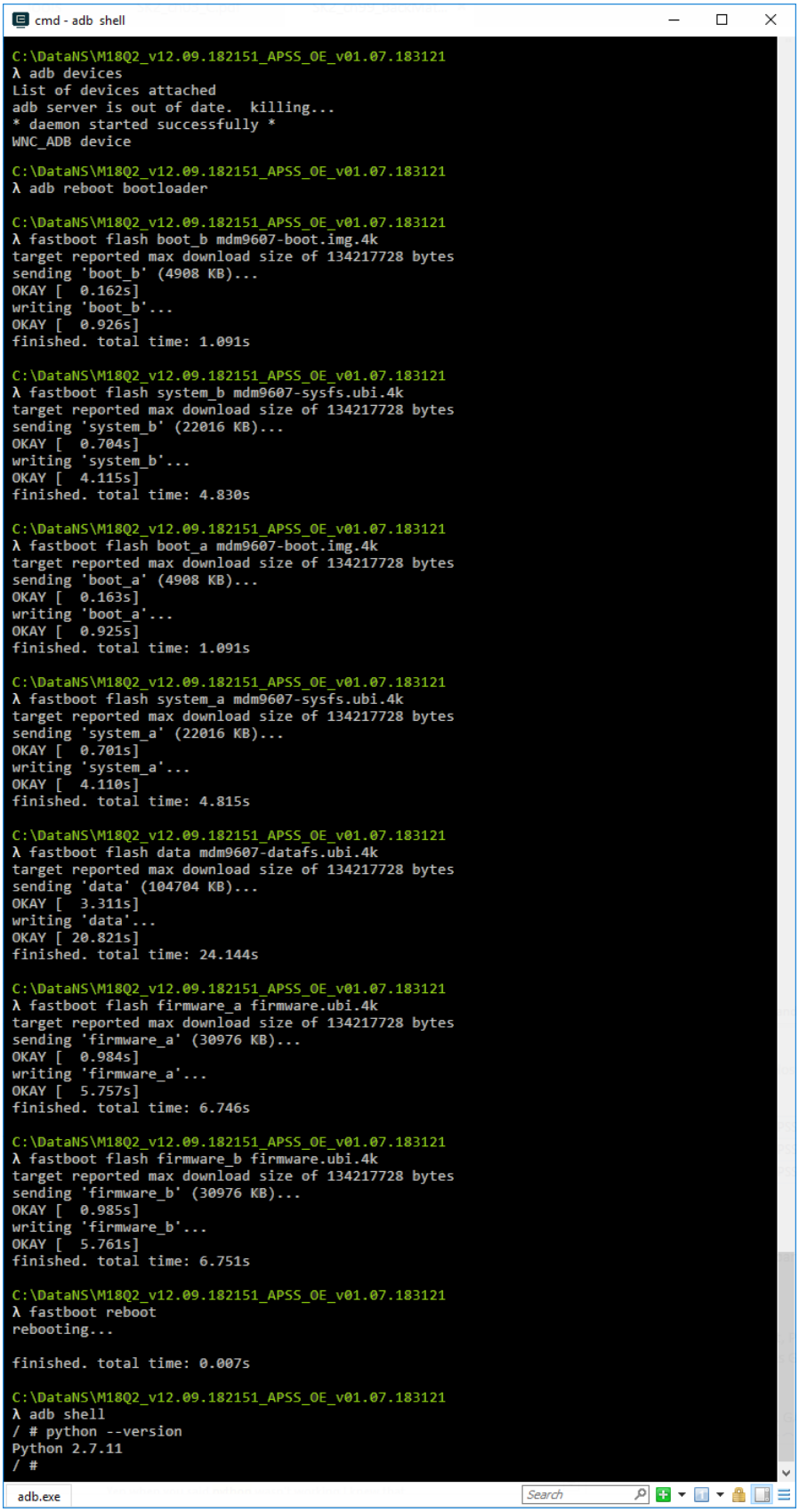


Figure 4.2 - Successful Flashing of Python Firmware

15. Exit the ADB remote session.

```
exit
```

16. Examine your Network settings

```
ipconfig
```

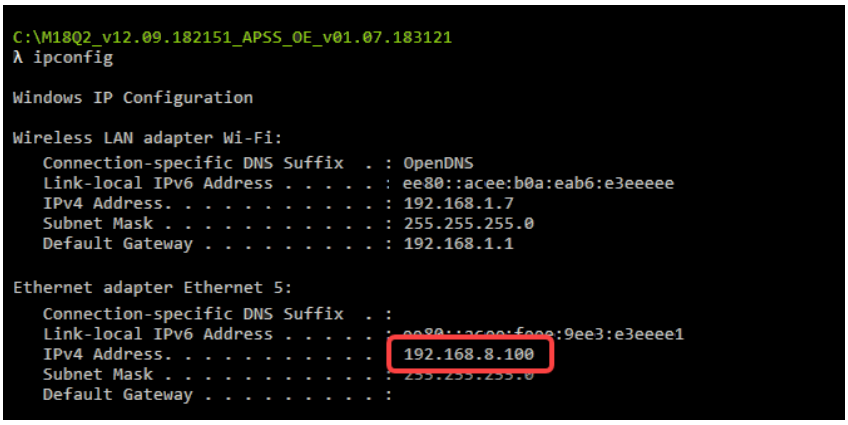


Figure 4.3 - ipconfig

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.)

17. Change back to your original ADB folder.

Leaving the Python image installation folder, return to your normal ADB folder. For example:

```
cd C:\adb
```

18. Run the 'adb devices' command to restart the original ADB server.

```
adb devices
```

Figure 4.4 - Restarting Original ADB Server

19. 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

From an Internet browser, connect to the following address:

```
192.168.8.1
```

Upon a successful connection, you should see the Python IDE overview.

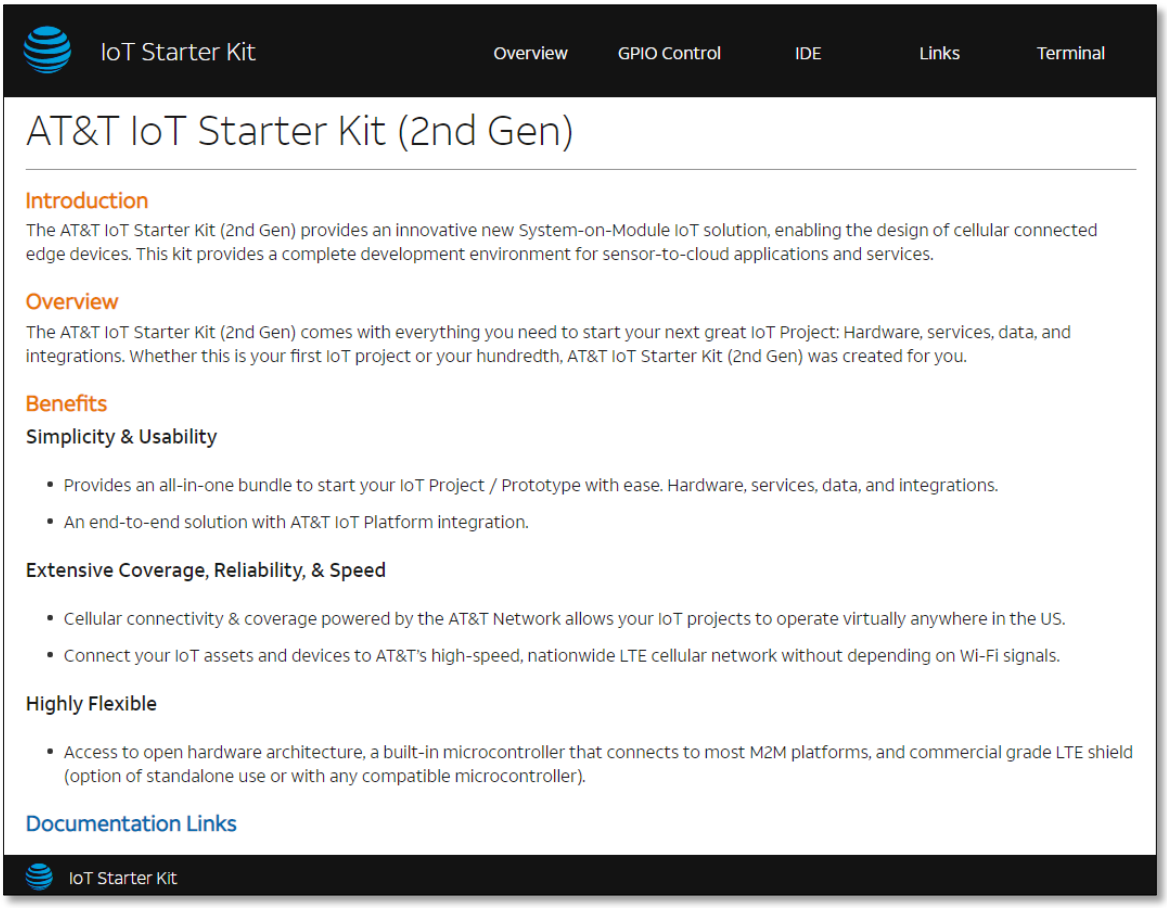


Figure 4.6 - Python IDE (Overview)

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

Click the “GPIO Control” button towards the top of the Python IDE.

```
GPIO Control
```

opens a screen that allows you to control the RGB LED on your connected SK2 as well as reading the ADC value.

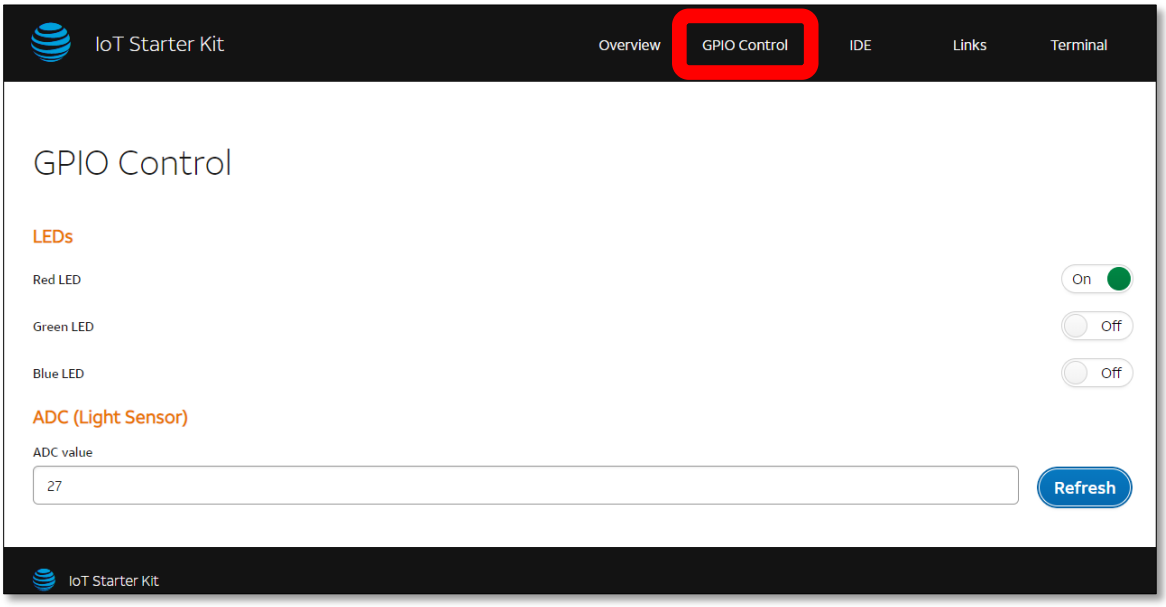


Figure 4.7 - Python IDE (GPIO Control)

You can turn the various LED colors on/off. For example, we turned on the Red LED.

Clicking ‘Refresh’ to read the ADC (analog to digital convertor) causes the IDE to read the ADC peripheral on the WNC module. Since it’s connected to a light sensor on the SK2, reading the ADC gives us a number that represents the amount of light hitting your board.

You can see a difference in light intensity using the SK2. First read 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

Clicking the “IDE” button towards the top right of the Python IDE:

```
IDE
```

opens the main Python IDE screen where you can create, edit, and run Python scripts.

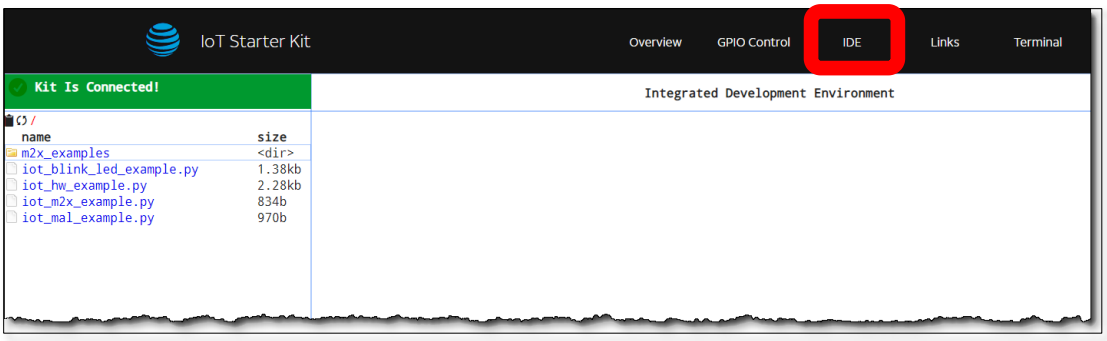


Figure 4.8 Python IDE (IDE)

You can open the `iot_blink_led_example.py` that comes preinstalled with the Python image by double-clicking on it. This opens the script in the editor, allowing you to modify or run the Python script.

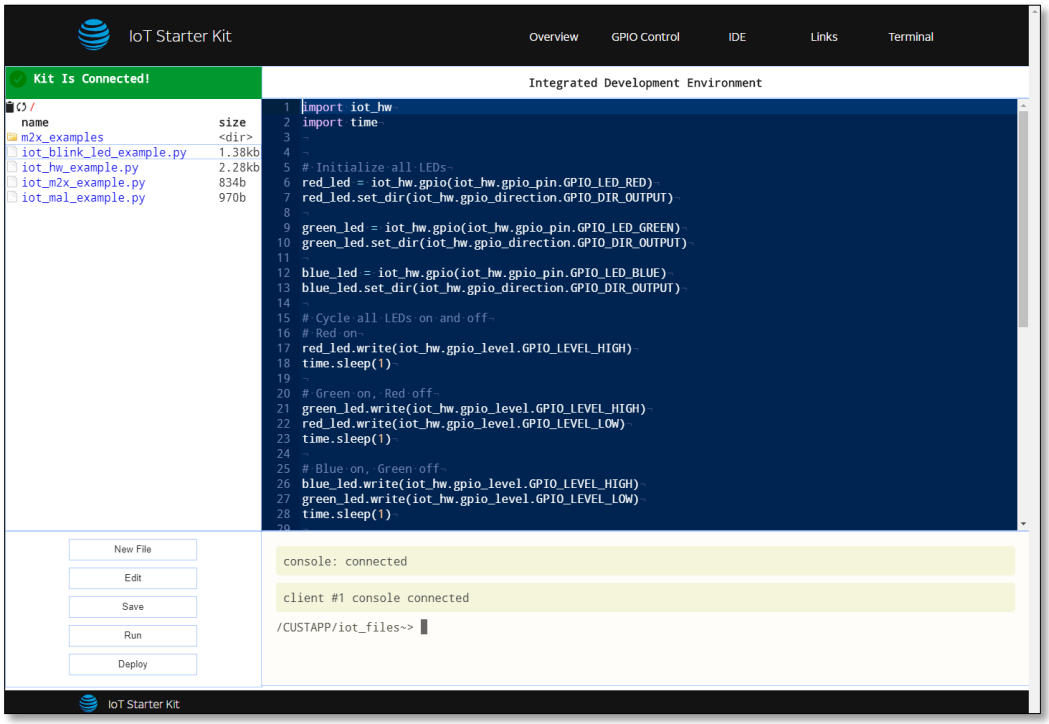


Figure 4.9 - Python IDE (IDE) showing code

Run the script open in the editor by clicking the “Run” button in the lower left-hand corner. The IDE asks you to verify the script you want to run before executing it.

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.

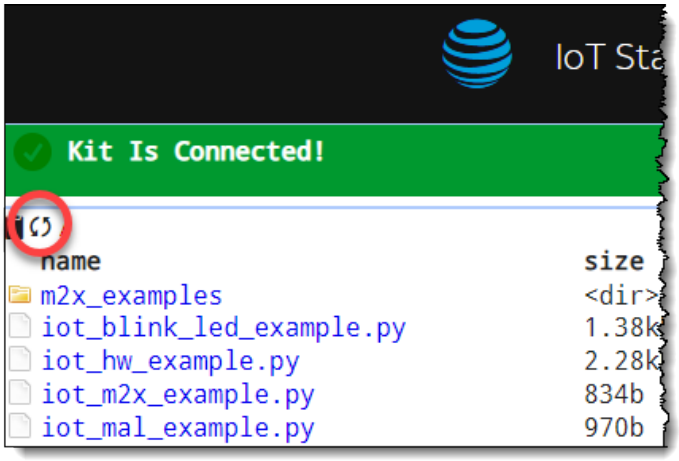


Figure 4.10 - Refresh File Listing

4.2.4. Links

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.

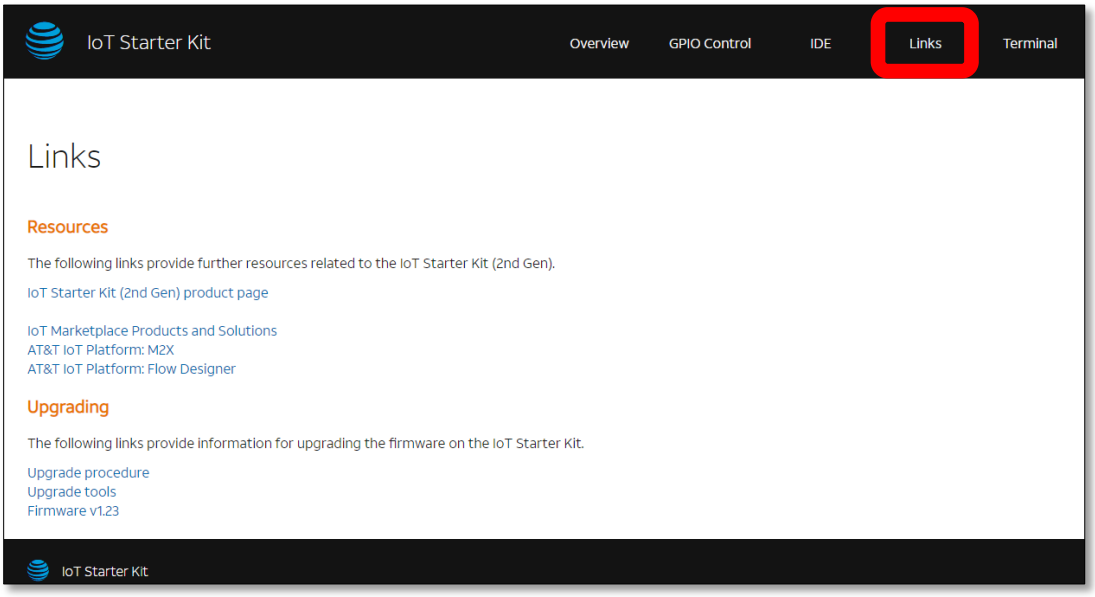


Figure 4.11 - Python IDE (Links)

Note: Some of these links may not be active.

4.2.5. Terminal

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.

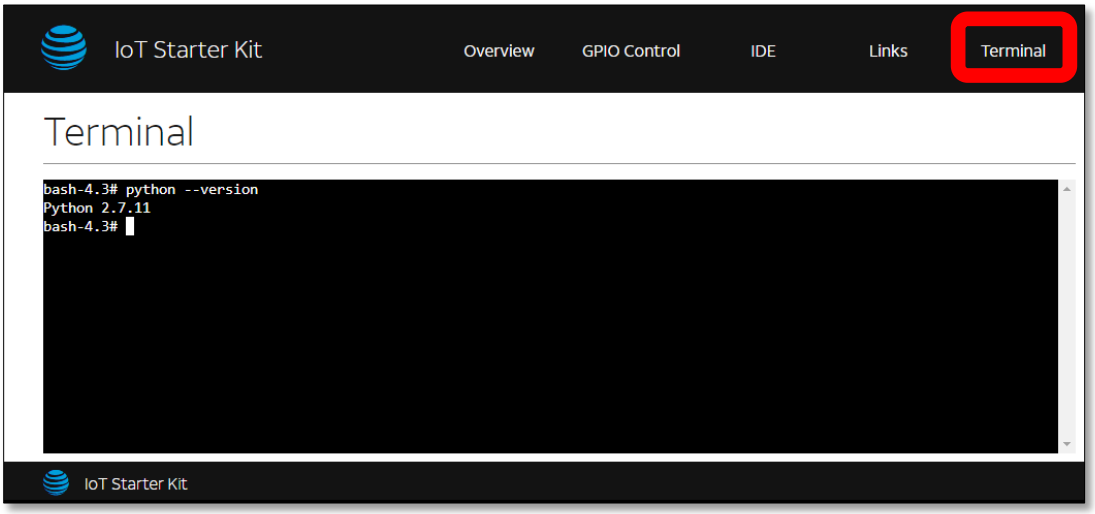


Figure 4.12 - Python IDE (Terminal)

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. Using the 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.

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 4.13 - 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.

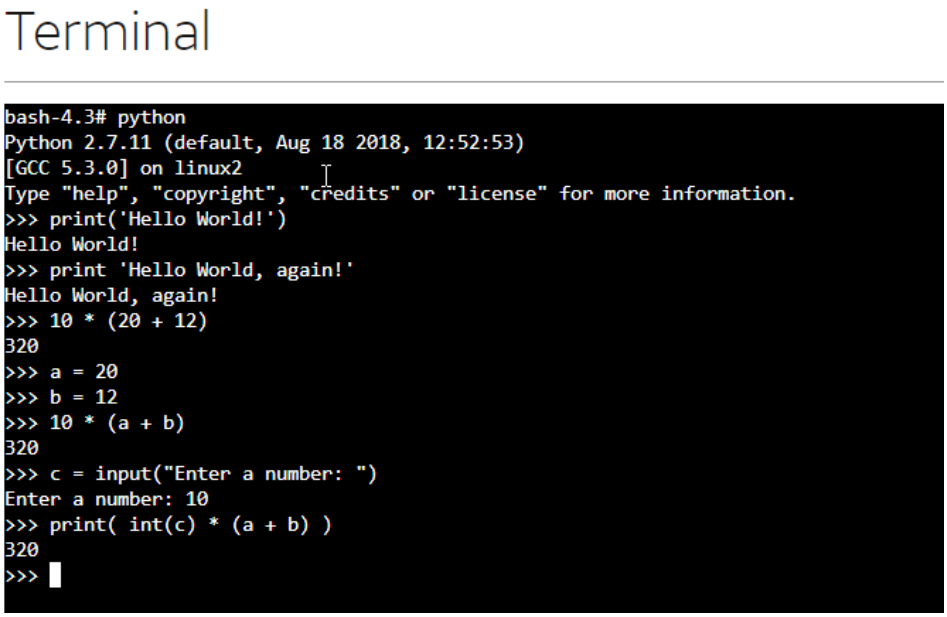


Figure 4.13 - Interactive Python interpreter session

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 113).

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.

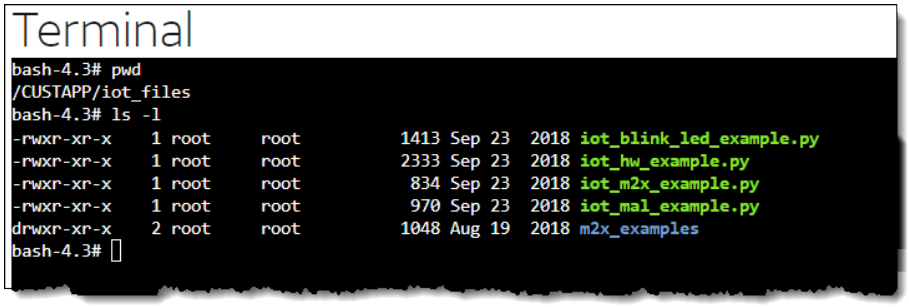


Figure 4.14 - Navigating to 'iot_files'

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. Example: Hello World using Python

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.

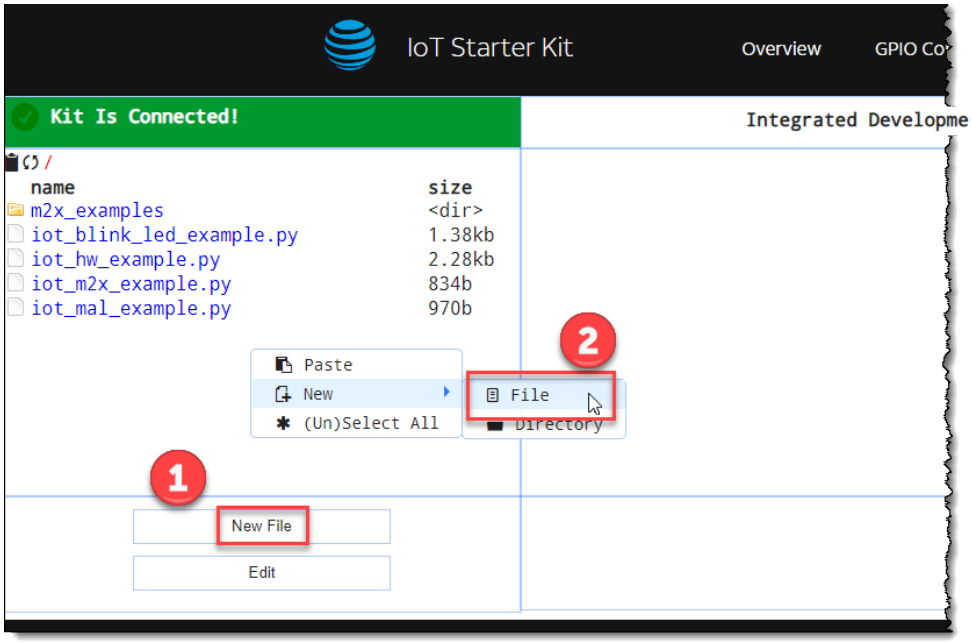


Figure 4.15 - Creating a New Python File

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 4.16 shows the error we received.

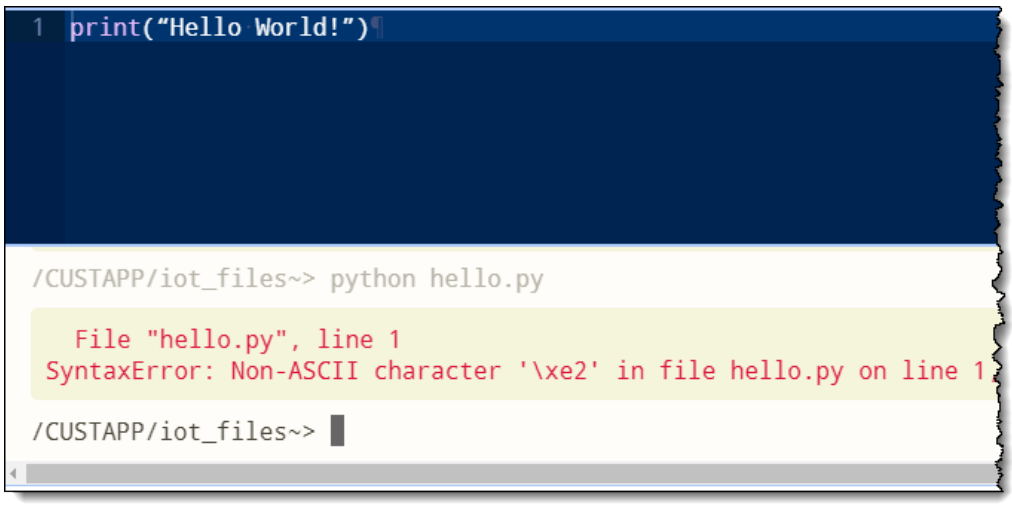


Figure 4.16 - Running 'hello.py' (with error)

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.

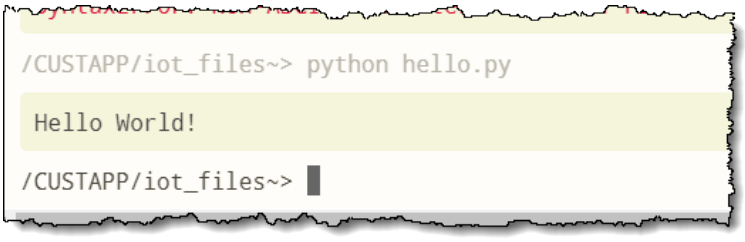


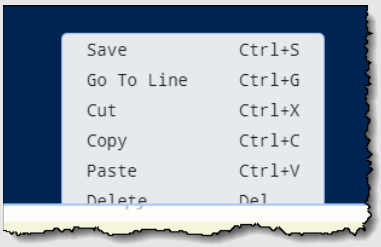
Figure 4.17 - hello.py Success

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. Example: 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 User's 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.py

4.3.4. Example: 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 4.2) 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. Example: 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. Killing a Running Python Program

If you accidentally write a program that won't exit (e.g. an endless loop) you a second terminal and Linux 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

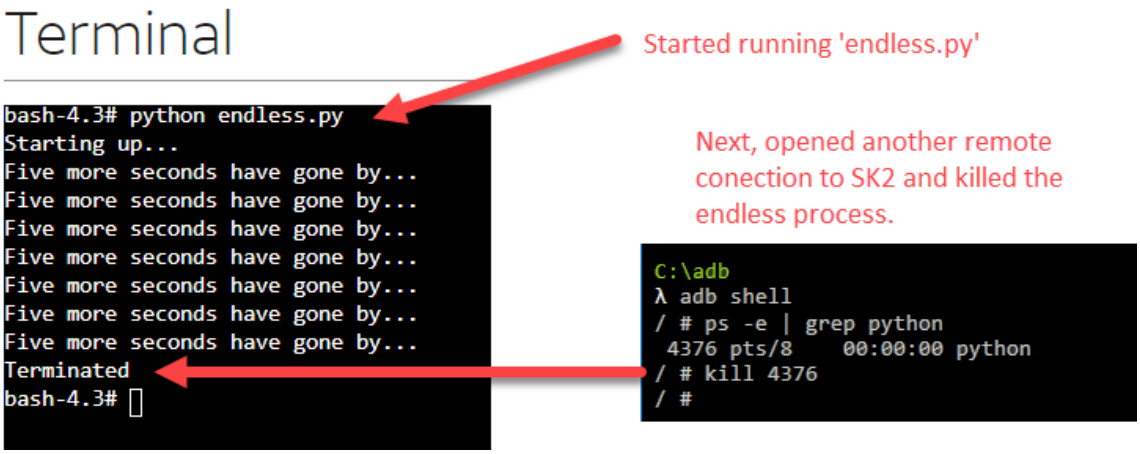


Figure 4.20 - Running and killing endless.py

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. 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. Example: Using WWAN LED Class

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. WWAN Class 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 *Example: 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 4.8) 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 been 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*

5. Wireless Wide Area Networking with LTE

AT&T's 2nd Generation IoT Starter Kit (SK2) provides hardware that allows connecting the module to the Wireless Wide Area Network (WWAN). For the SK2, this puts the "I" into IoT (Internet of Things).

Using both Python and C/C++, this chapter discusses connecting to the Modem Abstraction Layer (MAL) to access the WWAN, as well as status information about the module's radio hardware.

A WWAN connection to the Internet allows connection to AT&T's IoT Cloud Platform services, such as M2X (for managing devices and storing data) and Flow (building custom IoT Cloud applications). Before venturing into these services, though, a simple HTTP example connecting the SK2 to Hookbin.com provides an easy way to confirm our ability to make HTTP connections.

Finally, this chapter also examines sending SMS messages. Since the module does not presently support direct SMS communications, connections via two different 3rd party services – Twilio and IFTTT – are demonstrated.

Topics

Detailed Outline

5.1. Modem Abstraction Layer (MAL)

As described in Chapter 1, the AT&T IoT Starter Kit (2nd Generation) (nicknamed 'SK2) contains both an applications processor – which we can program in C/C++ or Python – as well as an LTE modem to establish and manage cellular network communications over the WWAN.

Applications can access and configure the modem using the MAL library API (application programming interface). After using the MAL to turn on the radio and establish connection to the cellular 3G/4G LTE network, applications are free to communicate over traditional networking protocols, such as HTTP, cURL, and so on.

5.1.1. Connecting to the MAL Using the LTE API

Both Python and C/C++ languages support connections to the Modem Abstraction Layer. Each language's API is documented in its respective manual:

- **Python**: Python_M18Qx_MAL/LTE_IoT_API_Guide
- **C/C++**: *Avnet M18Qx LTE IoT API Guide*

The MAL API consists of four groups of functions:

1. **WWAN**: Control the LTE module and retrieve status information.
2. **Network**: Control the system network.
3. **System**: Provide information about the system (e.g. the module's ICCID).
4. **Location**: Managing GPS location data.

Communicating with the MAL involves sending and receiving JSON packets to the modem. Code examples are provided for each language in this section.

5.1.1.1. Python Example (ex_01_mal_api_example.py)

The Python "iot_mal" module makes it easy connecting to the LTE Modem on the SK2. After importing the module, you can instantiate a Python object for each of the API's function groups. The following example utilizes the "system" API to read the SIM Card's number (i.e. ICCID) from the SK2 hardware.

Listing 5.1: ex_01_mal_api_example.py (Python)

```
1  ################################################################################
2  # file: ex_01_mal_api_example.py
3  #
4  # Connect to the Modem Abstraction Layer and access the SK2 SIM cards ICCID.
5  # MAL provides three API groups: System, Network, and WWAN. This example uses
6  # the "System" API.
7  #
8  # Along with showing access to MAL, this example demonstrates how to get the
9  # data value from the returned dictionary string.
10 #
11 ################################################################################
12 import iot_mal                                          # Import the Modem Abstraction Layer (MAL) API module
13
14 # Accessing MAL to get system information
15 mal_sys = iot_mal.system()                             # Connect to MAL.system
16 raw_iccid = mal_sys.get_iccid()                        # This method reads the ICCID from the SIM card
17
18 print(raw_iccid)                                       # Print the "raw" output for the iccid and
19                                                        # notice that a Unicode Python dictionary is returned
20 # Python formatting examples
21 iccid = raw_iccid.get('iccid')                         # Use .get() to extract iccid value from dictionary var
22 print(iccid)
23
24 print('Your ICCID is: {0}'.format(iccid))             # One of many pythy ways to embed data into output string
25 print('Your ICCID is: {0}'.format(mal_sys.get_iccid().get('iccid')))   # Or do it all in one complex line of code
```

The Python module neatly handles creating and sending the JSON packet to the modem for you. The API function returns a dictionary variable with the results as shown on the first line of the response below.

```
{u'errno': 0, u'errmsg': u'', u'iccid': u'89011703278113077369'}
89011703278113077369
Your ICCID is: 89011703278113077369
Your ICCID is: 89011703278113077369

/CUSTAPP/iot_files>
```

Figure 5.1 - Results of ex_01_connect_to_mal.py

Various Python functions can be used to extract and format the results. This example used the `.get()` function to extract the "iccid" field from the dictionary variable and then formatted it into a pretty output.

5.1.1.2. C/C++ Example (c_01_mal_api_example)

The following example extracts information from the MAL using the C/C++ language. The C/C++ version requires more project code than the Python example, demonstrating that Python is easier to use. Unlike Python, the C/C++ example requires a number of support files to package, send, and receive the MAL commands to and from the modem.

These files were taken from Avnet's "*iot_monitor*" (i.e. QuickStart) example. Using them, our `main.cpp` can start the data service and extract a variety of information from the MAL.

- **`iot_monitor.h:`** Contains definitions for the *iot_monitor* application, which was the basis for the C/C++ MAL example. The simple MAL example does not require all these elements, but to keep things simple, it was copied directly from the Avnet *iot_monitor* example.
- **`mal.cpp/mal.hpp:`** Provides an easy set of C functions for calling the MAL API. Along with getting system information from the modem, there are functions for starting and configuring cellular connections.
- **`jsmn.c/jsmn.h/maljson.cpp:`** Files provided with the *iot_monitor* application for creating, parsing and managing JSON packets when communicating with the MAL API. These files are required by the `mal.cpp` when communicating with the MAL API.

Listing 5.2: c_01_mal_api_example (C/C++)

```
    |----+----1----+----2----+----3----+----4----+----5----+----6----+----7---
 1  #include <stdio.h>
 2  #include "mal.hpp"
 3  #include "iot_monitor.h"
 4  #include <unistd.h>
 5
 6  int c;
 7  sysinfo mySystem;
 8  unsigned int dbg_flag = 0;
 9
10
11  int main(void) {
12
13      json_keyval om[20];
14
15      printf("Hello World\n");
16
17      c = start_data_service();
18
19      while ( c < 0 ) {
20          printf("WAIT: starting WNC Data Module (%d)\n",c);
21          sleep(10);
22          c = start_data_service();
23      }
24
25      printf("\nData service running\n\n");
26
27      do
28          mySystem.model=getModelID(om, sizeof(om));
29      while( mySystem.model == "service is not ready");
30
31      mySystem.firmVer   = getFirmwareVersion(om, sizeof(om));
32      mySystem.appsVer   = getAppsVersion(om, sizeof(om));
33      mySystem.malwarVer = getMALManVer(om, sizeof(om));
34      mySystem.ip        = get_ipAddr(om, sizeof(om));
35      mySystem.iccid     = getICCID(om, sizeof(om));
36      mySystem.imei      = getIMEI(om, sizeof(om));
37      mySystem.imsi      = getIMSI(om, sizeof(om));
38
39      printf("WNC: Module   #    = %s\n", mySystem.model.c_str());
40      printf("WNC: Apps Ver #    = %s\n", mySystem.appsVer.c_str());
41      printf("WNC: Firmware #    = %s\n", mySystem.firmVer.c_str());
42      printf("WNC: MAL Ver. #    = %s\n", mySystem.malwarVer.c_str());
43      printf("WNC: IP Addr. #    = %s\n", mySystem.ip.c_str());
44      printf("SIM: ICCID    #    = %s\n", mySystem.iccid.c_str());
45      printf("SIM: IMEA     #    = %s\n", mySystem.imei.c_str());
46      printf("SIM: IMSI     #    = %s\n\n",mySystem.imsi.c_str());
47
48      return 0;
49  }
50
```

Running this example prints the following results:

```
C:\adb
λ adb shell
* daemon not running; starting now at tcp:5037
* daemon started successfully
/ #
/ # python /CUSTAPP/iot_files/mal_03_get_system_info.py
System Information
------------------
Firmware version: M18Q2_v12.09.182151
IMEI            : 014814000341576
IMSI            : 310170811307736
ICCID           : 89011703278113077369

Connection Information
----------------------
Connection State: Connected
Connection Time : 00:01:25:25
Radio Mode      : 4G LTE
My IP address   : 10.43.197.243
/ #
```

Figure 5.2 – Results from ex_01_mal_api_example

5.1.2. Using WWAN API to Get Network Time

The second example uses a WWAN MAL function to extract the time from the network.

5.1.2.1. Python Example (ex_02_get_network_time)

Using the *iot_mal.wwan()* function, Python can access the time from the WWAN network. As in the previous example, the result is returned in a Python dictionary variable. The following example extracts the time from the variable and adjusts it for Central Standard Time (CST).

Listing 5.3: ex_02_get_network_time.py

```
1  ################################################################################
2  # file: ex_02_get_network_time.py
3  #
4  # Connect to the Modem Abstraction Layer and access the network time.
5  # The network time can be accessed in a couple of different ways; this
6  # example uses the WWAN get_network_time() method.
7  #
8  # Along with showing access to MAL, this example demonstrates how to get the
9  # data value from the returned dictionary string and format it.
10 #
11 ################################################################################
12 import iot_mal                                                  # Import the Modem Abstraction Layer (M
13 from datetime import datetime, timedelta
14
15 CST = -6                                                       # Time difference (hours0 between GMT a
16
17 # Accessing MAL to get system information
18 mal_wwan = iot_mal.wwan()                                      # Connect to MAL.wwan
19 raw_time = mal_wwan.get_network_time()                         # Reads network time over the wireless
20
21 print(raw_time)                                                # Print the "raw" value for the network
22
23 # Python formatting examples
24 net_time = raw_time.get('network_time')                        # Use .get() to extract the time from d
25 print('net_time = {0}'.format(net_time))
26
27 t = datetime.strptime(net_time, '%Y/%m/%d %H:%M:%S-%f,00')     # Convert unicode string to datetime va
28 lt = t + timedelta(hours=CST)
29 print('Local time = {0}'.format(lt))
30
31 # Do it all in one line of code
32 print('Local time = {0}'.format(datetime.strptime(mal_wwan.get_network_time().get('network_time'), '%Y/%m/%d %
```

The results for this example are shown below. Notice that there's a 6-hour time difference between "network time" (GMT) and local time (CST).

```
{u'errno': 0, u'network_time': u'2019/1/4 21:55:05-24,00', u'errmsg': u''}
net_time = 2019/1/4 21:55:05-24,00
Local time = 2019-01-04 15:55:05.240000
Local time = 2019-01-04 15:55:05.240000

/CUSTAPP/iot_files>
```

Figure 5.3 - ex_02_get_network_time.py Output

5.1.2.2. C/C++ Example (c_02_get_network_time)

As discussed in Section 5.1.1, commands sent to the modem (i.e. MAL) must be sent as JSON packets. To simplify coding, `c_01_mal_api_example` (Listing 5.2) reused the `mal.cpp` file from the "iot_monitor" C++ QuickStart demo program, because it already contained a set of functions for accessing data from the MAL. Unfortunately, `mal.cpp` does not contain a function to access the network time.

Example `c_02_get_network_time` works around this limitation by adding a *get_networkTime()* function to `mal.cpp`. While this could have been added to `main.cpp`, we preferred to consolidate all the mal functions in `mal.cpp`.

To make these changes we first added a new function prototype to `mal.hpp`:

Listing 5.4: c_02_get_network_time/src/mal.hpp

```
39  char * getMALManVer(json_keyval *kv, int kvsize);
40  char * get_ipAddr(json_keyval *kv, int kvsize);
41  char * get_networkTime(json_keyval *kv, int kvsize);
42  char * getModelID(json_keyval *kv, int kvsize);
```

Creating the new function was simple. Opening up `mal.cpp`, make a copy of *get_ipAddr()* and then rename it to *get_networkTime()*. The only change required, besides the function name, was the command being sent to the MAL. In this case, the command needed is "get_wwan_network_time". (This was found in the *Avnet LTE API Guide*.)

Listing 5.5: c_02_get_network_time/src/mal.cpp

```
143 char * get_ipAddr(json_keyval *kv, int kvsize) {
144     int  i, k;
145     char rstr[500];
146     char jcmd[] = "{ \"action\" : \"get_wwan_ipv4_network_ip\" }";
147
148     send_mal_command(jcmd, rstr, sizeof(rstr), true);
149     i = parse_maljson (rstr, kv, kvsize);
150     if( i < 0 )  //parse failed
151         return NULL;
152     else if( atoi(kv[1].value) ) // we got an error value back
153         return kv[2].value;      // so return error message
154     else
155         return kv[3].value;
156 }
157
158 char * get_networkTime(json_keyval *kv, int kvsize) {
159     int  i, k;
160     char rstr[500];
161     char jcmd[] = "{ \"action\" : \"get_wwan_network_time\" }";
162
163     send_mal_command(jcmd, rstr, sizeof(rstr), true);
164     i = parse_maljson (rstr, kv, kvsize);
165     if( i < 0 )  //parse failed
166         return NULL;
167     else if( atoi(kv[1].value) ) // we got an error value back
168         return kv[2].value;      // so return error message
169     else
170         return kv[3].value;
171 }
172
```

Since the main() function already referenced `mal.cpp`, the appropriate references were already included. That meant we only needed to add the *get_networkTime()* call to our code to get the time.

Listing 5.6: c_02_get_network_time/src/main.cpp

```
50      printf("SIM: IMSI     #     = %s\n\n",mySystem.imsi.c_str());
51
52      printf("\nNetwork Time = %s\n\n", get_networkTime(om, sizeof(om)));
53
54      return 0;
55  }
```

Thankfully, the arguments for *get_networkTime(om, sizeof(om))* were easy to fill in since they are the same as the *get_ipAddr()* call found earlier in `main.cpp`.

5.1.3. Python MAL Example

The final MAL information example obtains system and connection data.

5.1.3.1. Python Example (ex_03_get_system_info.py)

The third MAL example turns on the modem using the network *set_connection_mode()* function, then retrieves system data as well as information about the connection itself. To enhance the printout, a simple parse_state() function was created in Python to translate the *Connection State* from a numerical value to something more readable.

Listing 5.7: ex_03_get_system_info.py

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9----+----0
################################################################################
# file: ex_03_get_system_info.py
#
# Connect to the Modem Abstraction Layer and access system and connection info
################################################################################
import iot_mal                          # Import the Modem Abstraction Layer (MAL) API module

################################################################################
# function: parse_state(state)
#
# Parses the connection state value and returns descriptive string
################################################################################
def parse_state(state):
    if state == 0:
        ret = 'Disconnected'
    elif state == 1:
        ret = 'Disconnecting'
    elif state == 2:
        ret = 'Connecting'
    elif state == 3:
        ret = 'Connected'
    elif state == 4:
        ret = 'Disconnected, and PIN locked'
    elif state == 5:
        ret = 'Disconnected, and SIM removed'
    return ret

################################################################################
# Main code
################################################################################

# Setup the MAL and data connection
network_handler = iot_mal.network()         # Connect to MAL.network
network_handler.set_connection_mode(
     1,                                     # Mode: 0 - Always, 1 - On-demand
    10,                                     # On-demand Timeout: Disconnect in mins if no access
     2)                                     # Manual Mode: 0 - Disconnect, 1 - Connect (Always/on-de
                                            #              2 - Connect once
system_handler = iot_mal.system()           # Connect to MAL.system

# Print out system information
firm  = system_handler.get_firmware_version().get('version')
imei  = system_handler.get_imei().get('imei')
imsi  = system_handler.get_imsi().get('imsi')
iccid = system_handler.get_iccid().get('iccid')

print('System Information')
print('------------------')
print('Firmware version: ' + firm)
print('IMEI            : ' + imei)
print('IMSI            : ' + imsi)
print('ICCID           : ' + iccid)

# Print out connection information
status = network_handler.get_connection_status()

print('\nConnection Information')
print(  '----------------------')
print('Connection State: ' + parse_state(status.get('state')))
print('Connection Time : ' + status.get('connection_time'))
print('Radio Mode      : ' + str(status.get('radio_mode')) + 'G ' + status.get('data_bearer_tech'))
print('My IP address   : ' + status.get('ip'))

```

The following output was generated by this example:

```
C:\adb
λ adb shell
* daemon not running; starting now at tcp:5037
* daemon started successfully
/ #
/ # python /CUSTAPP/iot_files/mal_03_get_system_info.py
System Information
------------------
Firmware version: M18Q2_v12.09.182151
IMEI            : 014814000341576
IMSI            : 310170811307736
ICCID           : 89011703278113077369

Connection Information
----------------------
Connection State: Connected
Connection Time : 00:01:25:25
Radio Mode      : 4G LTE
My IP address   : 10.43.197.243

/ #
```

Figure 5.4 - ex_03_get_system_info.py Output

5.1.3.2. C/C++ Example (c_03_get_system_info)

This example expands on 5.1.2.2 C/C++ Example (c_02_get_network_time) by adding a call to *WWANStatus()*, which passes WWAN and Network information back for printing to the terminal.

Listing 5.8: Chapter_05\c_03_system_info\src\main.cpp

```
#include <stdio.h>
#include "mal.hpp"
#include "iot_monitor.h"
#include <unistd.h>
#include "wwan_status.hpp"

int c;
sysinfo  mySystem;
wwanInfo myWwan;
netInfo  myNetw;
unsigned int dbg_flag = 0;

int main(void) {

    int ret;
    json_keyval om[20];

    printf("Hello World\n");

    c = start_data_service();

    printf("SIM: ICCID     #    = %s\n", mySystem.iccid.c_str());
    printf("SIM: IMEA      #    = %s\n", mySystem.imei.c_str());
    printf("SIM: IMSI      #    = %s\n\n",mySystem.imsi.c_str());

    printf("\nNetwork Time = %s\n\n", get_networkTime(om, sizeof(

    WWANStatus(&myWwan, &myNetw);

    printf("Radio mode: %s\n\n", myWwan.radioMode.c_str());

    printf("connType %s\n", myNetw.connType.c_str());
    printf("connState %s\n", myNetw.connState.c_str());
    printf("connTime %s\n", myNetw.connTime.c_str());
    printf("provider %s\n", myNetw.provider.c_str());
    printf("radioMode %s\n", myNetw.radioMode.c_str());
    printf("dbTech %s\n", myNetw.dbTech.c_str());
    printf("roamStatus %s\n", myNetw.roamStatus.c_str());
    printf("sigStrength %s\n", myNetw.sigStrength.c_str());
    printf("sigLevel %s\n", myNetw.sigLevel.c_str());
    printf("lte %s\n", myNetw.lte.c_str());
    printf("wcdma %s\n", myNetw.wcdma.c_str());
    printf("ipv6 %s\n\n", myNetw.ipv6.c_str());

    return 0;
```

Rather than code this information directly into `main.cpp`, the new function was added to its own file to make it easier to reuse the code. In fact, you may need to obtain this information more than once if, for example, your program needs to react based on the changing value of a setting, such as *connection time*.

The files defining the new WWANStatus() function include `wwan_status.hpp` and `wwan_status.cpp`. The header (.hpp) file defines two data structures passed into, and updated within, the function. These were defined in a similar way to the *sysinfo* structure found in `iot_monitor.h`.

Listing 5.9: Chapter_05\c_03_system_info\src\wwan_status.hpp

```
#ifndef __WWAN_STATUS_H__
#define __WWAN_STATUS_H__

#ifdef __cplusplus
typedef struct {
  std::string radioMode;
  std::string sigStrength;
  std::string sigLevel;
  std::string radioState;
  std::string cirSwState;
  std::string packSwState;
  std::string regState;
  } wwanInfo;
extern wwanInfo myWwan;
#endif

#ifdef __cplusplus
typedef struct {
    std::string connType;
    std::string connState;
    std::string connTime;
    std::string provider;
    std::string radioMode;
    std::string dbTech;
    std::string roamStatus;
    std::string sigStrength;
    std::string sigLevel;
    std::string lte;
    std::string wcdma;
    std::string ipv6;
  } netInfo;
extern netInfo myNetw;
#endif

#ifdef __cplusplus
extern "C" {
#endif

void WWANStatus(wwanInfo *, netInfo *);

#ifdef __cplusplus
}
#endif

#endif // __WWAN_STATUS_H__
```

The *WWANStatus()* function in the `wwan_status.cpp` file is styled similar to how the *mySystem* structure (based on *sysinfo*) was filled out in main(). Here is a snippet of the file:

Listing 5.10: Chapter_05\c_03_system_info\src\wwan_status.cpp

```
#include <stdio.h>
#include "mal.hpp"
#include "iot_monitor.h"
#include <unistd.h>
#include "wwan_status.hpp"

void WWANStatus(wwanInfo *wwan, netInfo *net)
{
    json_keyval om[20];
    int k;
    int i;

    const char *radio_mode[] = {"No service", "3G", "4G" };
    const char *radio_state[]= {
        "No Registered", "Registered with network",
        "searching", "registration denied", "unknown" };
    const char *xs_state[]   = {"Unknown","Attached","Detached"};
    const char *roaming[]    = {"Home", "Roaming", "Unknown"};

    printf("\nWWAN Status \n");
    printf("============================================================\n");
    printf("               Received: %d key/value pairs:\n", get_wwan_status(om, sizeof(om)));

    wwan->radioMode   = om[3].value;
    wwan->sigStrength = om[4].value;
    wwan->sigLevel    = om[6].value;
    wwan->radioState  = radio_state[atoi(om[7].value)];
    wwan->cirSwState  = xs_state[atoi(om[8].value)];
    wwan->packSwState = xs_state[atoi(om[9].value)];
    wwan->regState    = roaming[atoi(om[16].value)];

    printf("             Radio Mode: %s(%s)\n", radio_mode[atoi(om[3].value)], om[3].value);
    printf("        Signal strength: %s\n", om[4].value);
    printf("           Signal Level: %s\n", om[6].value);
    printf("                  state: %s(%s)\n", radio_state[atoi(om[7].value)], om[7].value);
    printf(" Circuit-switched state: %s(%s)\n", xs_state[atoi(om[8].value)], om[8].value);
    printf("  Packet-switched state: %s(%s)\n", xs_state[atoi(om[9].value)], om[9].value);
    printf("     Registration state: %s(%s)\n\n", roaming[atoi(om[16].value)], om[16].value);

    const char * type[] = {"3G/LTE", "Ethernet", "WiFi" };
    const char * state[] ={ "Disconnected", "Disconnecting", "Connecting", "Connected",
                            "Disconnected, and PIN locked", "Disconnected and SIM removed" };

    printf("Network Connection Status \n");
    printf("============================================================\n");
    printf("          Received: %d key/value pairs:\n", get_connection_status(om, sizeof(om)));

    net->connType    = om[3].value;
```

The calls to the MAL are highlighted in the example above. These two functions, *get_wwan_status()* and *get_connection_status()*, are found in `mal.cpp` where they make calls into the MAL using the following two queries, respectively:

- get_wwan_serving_system_status
- get_network_connection_status

Remember, you can find the C/C++ MAL documented in the Python_M18Qx_MAL_IoT_API_Guide.

5.2. Communicating over HTTP/HTTPS

There are many protocols supporting communication over the Internet, but none quite as ubiquitous as the HyperText Transport Protocol (HTTP). The mainstay of communication with the World Wide Web (WWW), it's also useful sending and receiving data for a while range of uses, such as IoT data transfers.

5.2.1. cURL

This section provides HTTP examples whether you are using Shell Scripts, Python, or C/C++. When communicating from the command-line – or within Shell Scripts – our examples utilize the cURL utility, which is short for "command-line URL".

cURL is defined as: *Command line tool and library for transferring data with URLs.*

We demonstrate making HTTP requests using all three types of coding explored in this book:

Shell Scripts: Uses the cURL utility found in the SK2's Linux distribution.

Python: While Python cURL libraries exist (e.g. PyCurl), these modules are not included in the SK2. Rather, these Python examples use the simple HTTP *Requests* library module, which is installed on the SK2.

C/C++: Uses the cURL library that's already included in the C/C++ tools.

Like HTTP transactions, cURL lets you send data and headers to a URL. Look to Section 5.2.3.1 for an example showing the cURL command-line syntax.

5.2.2. Hookbin.com

To demonstrate HTTP connections, this example communicates with the website Hookbin.com. Upon request, this free site provides a unique URL for each session, which allows you to simply post data to the site and then view those posts via a web browser.

5.2.2.1. Getting Started with Hookbin.com

To get started with Hookbin, go to www.hookbin.com and create a new Hookbin Endpoint (that is, request a Hookbin URL).

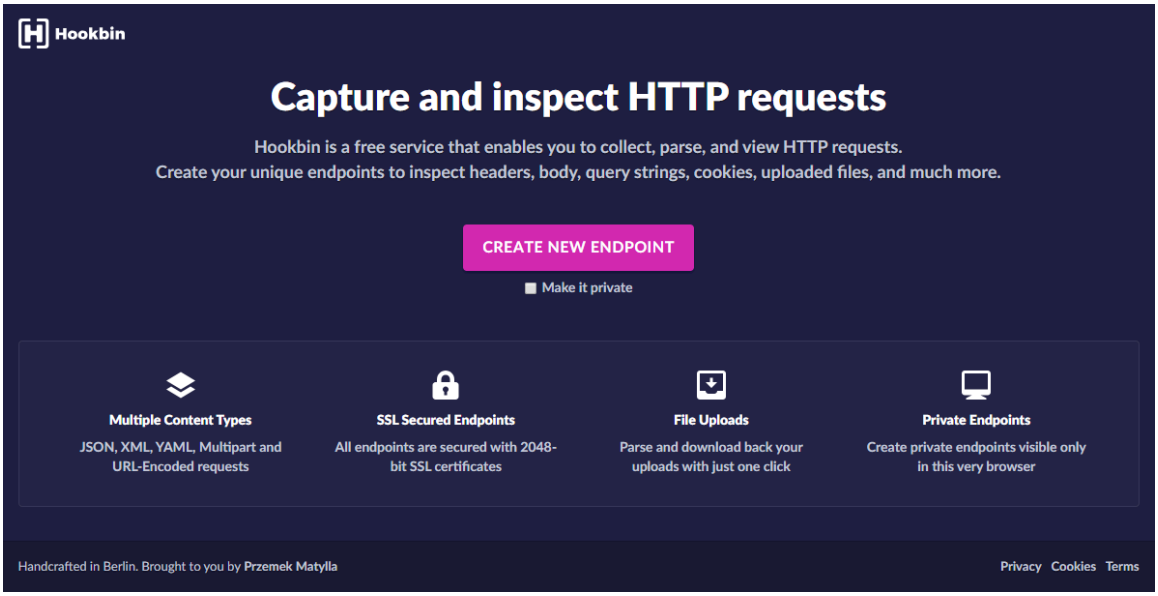


Figure 5.5 - Creating a New Endpoint at Hookbin.com

Here is the URL we received. We've used it within the following examples.

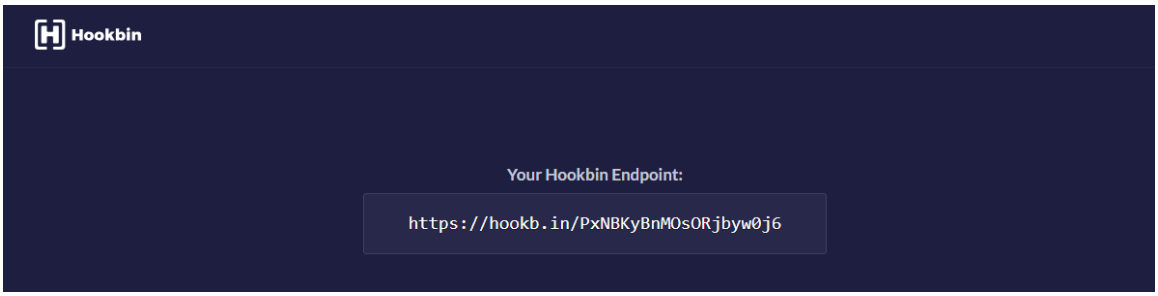


Figure 5.6 - Hookbin Endpoint

5.2.3. Hookbin.com Examples

The following examples – in Shell Script, Python, and C – demonstrate sending the value of the *User Button* (on the SK2) to Hookbin.com.

Note: The following examples depend upon the Hookbin.com endpoint that was established in the previous section (Section 5.2.2.1).

5.2.3.1. Shell-Script Example (sh_04_curl_to_hookbin.sh)

The following Shell Script example borrows heavily from one of the GPIO examples found in the User Guide's GPIO Appendix:

```
Chapter_99_Appendix\GPIO\userButtonLed.sh
```

Instead of modifying the RGB LED, though, this example sends the button's value to Hookbin.com.

Listing 5.11: Chapter_05\sh_04_curl_to_hookbin.sh

```
# sh_04_curl_to_hookbin.sh
#
# This example reads the value of the user button and sends
# that value to Hookbin.com using CURL
#

# Go to the GPIO directory
cd /sys/class/gpio

# Grant user-space access to the USER button

    # USER button
    echo 23 > export
    echo in > gpio23/direction

# Read the value of the USER push button into "val"
val=$(cat gpio23/value)
echo Button = $val

echo 23 > unexport

# Turn on LED
if [ $val -gt 0 ]
then
    # If "up"
    data='{"button":"up"}'
else
    # If "down"
    data='{"button":"down"}'
fi

curl https://hookb.in/PxNBKyBnMOsORjbyw0j6 --header "Content-Type: application/json" --request POST --data $data

```

In brief, the cURL syntax is really just a call to the cURL utility while providing the necessary command-line options. The Hookbin.com example includes four options after the "curl" command:

- **URL endpoint:** For example, Hookbin.com endpoint you established in Section 5.2.2.1
- **HTTP headers:** This example included a content-type header
- **Transaction type:** For example, we did an HTTP 'POST'
- **Data:** The data (i.e. payload) you want to send (in the case of a POST)

Check out the cURL website for more information: https://curl.haxx.se

Running the script results in the following output to the Linux terminal:

```
C:\adb
λ adb push sh_04_curl_to_hookbin.sh /CUSTAPP
sh_04_curl_to_hookbin.sh: 1 file pushed. 0.2 MB/s (658 bytes in 0.004s)

C:\adb
λ adb shell
/ # cd CUSTAPP/
/CUSTAPP # chmod +x sh_04_curl_to_hookbin.sh
/CUSTAPP # ./sh_04_curl_to_hookbin.sh
Button = 0
{"success":true}/CUSTAPP #
```

Figure 5.7 – Running sh_04_curl_to_hookbin.sh

But more importantly, refresh your Hookbin.com page to see the cURL result. The Body in this example shows that our button was being held down.

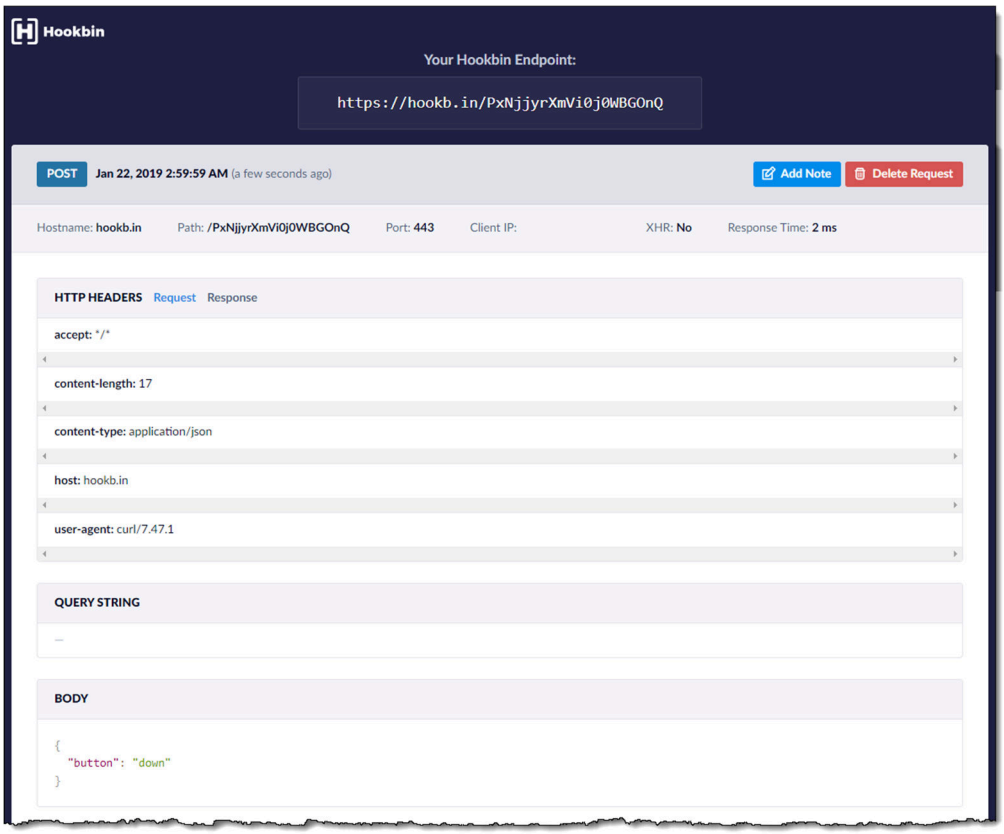


Figure 5.8 – Hookbin.com results for sh_04_curl_to_hookbin.sh

5.2.3.2. Python Example: ex_04_http_to_hookbin.py

The SK2 Python image includes the *simple* HTTP requests library, making it easy to initiate HTTP transfers and allowing you to get or post information to the web. The Hookbin example utilizes the GPIO code from Chapter 4 to read the state of the SK2's USER BUTTON and post that data to Hookbin.com.

Note: While we agree that using "Requests" is simple, it's really the developers who claim the module is *"A simple HTTP library for Python, built for human beings"*.)

Notice the Hookbin URL, that was copied from their website, #defined in the code. This is the URL that we'll post our button state value to.

Listing 5.12: ex_04_http_to_hookbin.py

```
1  ###############################################################################
2  # file: ex_04_http_to_hookbin.py
3  #
4  # This example sends a simple JSON data packet via HTTP to a site
5  # called Hookbin. This free 3rd party site allows you to view HTTP posts,
6  # puts, and gets. To try this example, you'll need to replace the HOOKBIN_URL
7  # address with one you obtain from http://www.hookbin.com
8  #
9  # Using GPIO to read the value of the pushbutton was discussed in Chapter 4.
10 ###############################################################################
11 import iot_mal                                    # Import the Modem Abstraction Layer (MAL) API module
12 import iot_hw                                     # Allows access to SK2 hardware resources
13 import requests                                   # A simple HTTP library for Python, built for human beings
14
15 HOOKBIN_URL = r'https://hookb.in/PxNBKyBnMOsORjbyw0j6'# Replace URL with one you get from Hookbin.com
16
17 #### Setup the MAL and data connection ########################################
18 network_handler = iot_mal.network()               # Connect to MAL.network
19 network_handler.set_connection_mode(
20                                     1,            # Mode: 0 - Always, 1 - On-demand
21                                    10,            # On-demand Timeout: Disconnect in mins if no access
22                                     2)            # Manual Mode: 0 - Disconnect, 1 - Connect (Always/on-dema
23                                                   #              2 - Connect once
24
25 #### Initialize the GPIO connected to the SK2 User Button ####################
26 button = iot_hw.gpio(iot_hw.gpio_pin.GPIO_USER_BUTTON)
27 button.set_dir(iot_hw.gpio_direction.GPIO_DIR_INPUT)
28
29 if button.read():
30     button_state = 'up'
31 else:
32     button_state = 'down'
33
34 #### Use Python 'requests' module to send data over HTTP ######################
35 url = HOOKBIN_URL                                 # Get URL from Hookbin.com (free 3rd party service)
36
37 headers = {                                       # List contains required HTTP headers
38     'Content-Type': 'application/json',
39 }
40
41 data = {                                          # JSON data to be sent
42     "button": button_state
43 }
44
45 r = requests.post(url, headers=headers, json=data)    # Post data to Hookbin
46 print(r.text)                                     # Print the response from Hookbin
47
```

This code example breaks into three sections:

1. Enable the network radio connection.
2. Get the state of the USER Button
3. Configure the “URL”, HTTP Headers, and DATA variables, then call the *requests.post()* function. Like using cURL in the Shell Scripts example, our HTTP post requires the same information. In this case, the “POST” command is part of the function call, while the other elements are passed as parameters.

After executing the example with the USER BUTTON in the up-state, then refreshing the Hookbin.com page, the following POST information appeared.

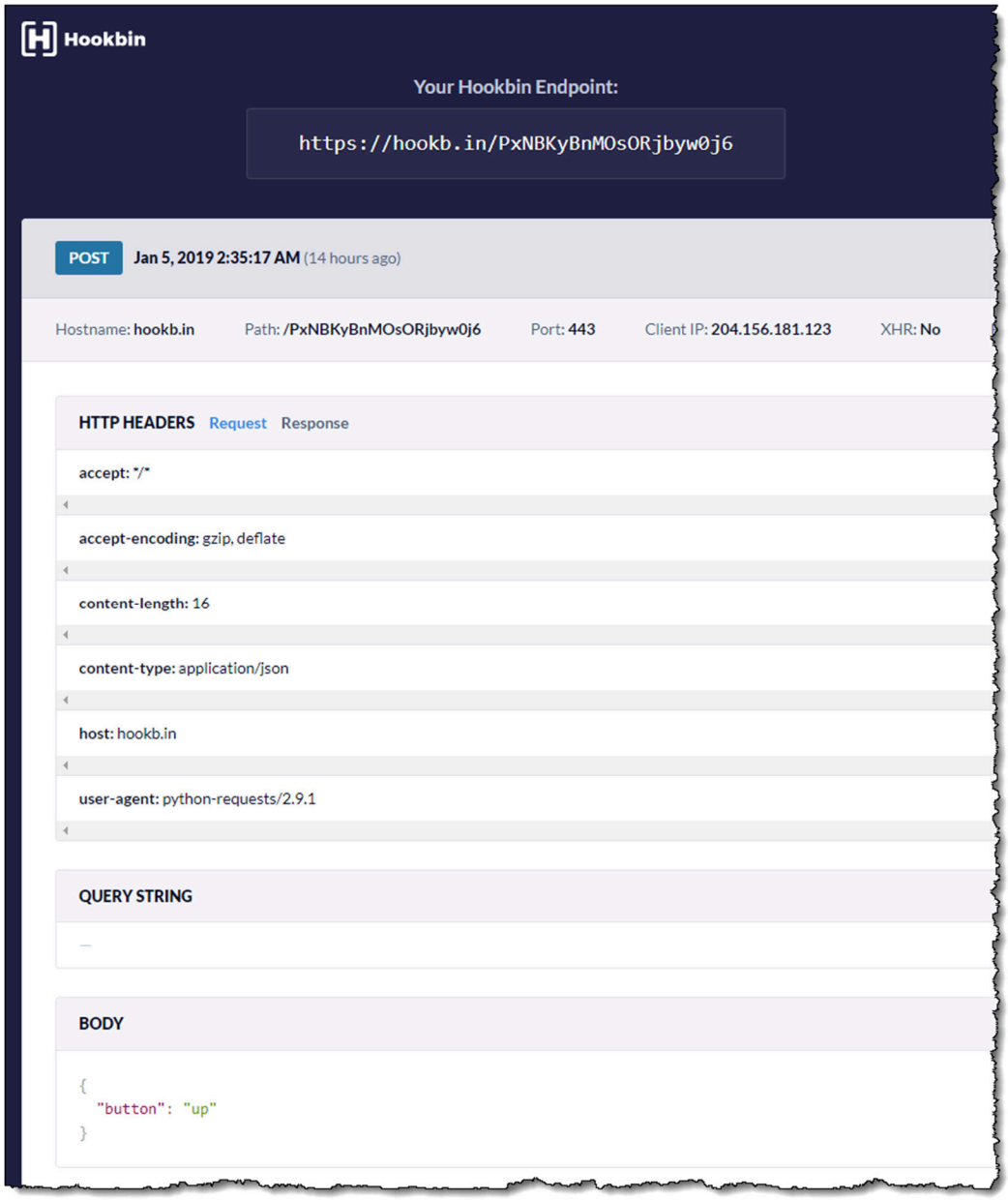


Figure 5.9 - ex_04_http_to_hookbin.py

5.2.3.3. C/C++ Example (c_04_http_to_hookbin)

The C/C++ example builds upon the previous examples in this chapter by sending the module's *Connection Time* to Hookbin.com. (It would have been just as easy sending the button value, but in this case, we used one of the WWAN status values read from the MAL.)

Here's the listing for `main.cpp`:

Listing 5.13: c_04_http_to_hookbin/src/main.cpp

```
1  #include <stdio.h>
2  #include "mal.hpp"
3  #include "iot_monitor.h"
4  #include <unistd.h>
5  #include "wwan_status.hpp"
6  #include "my_curl_post.h"
7
8  #define URL "https://hookb.in/PxNBKyBnMOsORjbyw0j6"
9
10 int c;
11 sysinfo  mySystem;
12 wwanInfo myWwan;
13 netInfo  myNetw;
14 unsigned int dbg_flag = 0;
15 json_keyval wwanStatus[20];
16 json_keyval networkStatus[20];
17
18 int main(void) {
19
20     int ret;
21     char str[80];
22
23     json_keyval om[20];
24
25     printf("Hello World\n");
57     printf("SIM: IMEA     #     = %s\n", mySystem.imei.c_str());
58     printf("SIM: IMSI     #     = %s\n\n",mySystem.imsi.c_str());
59
60     WWANStatus(&myWwan, &myNetw);
61
62     sprintf(str, "Connection Time = %s\n", myNetw.connTime.c_str());
63     ret = curl_post(URL, str);
64
65     printf("\n Program is exiting... \n");
66
67     return 0;
68 }
```

Only four lines were added to `c_03_system_info/src/main.cpp` in order to send the data to Hookbin.com:

1. Include the header file for the curl_post() function.
2. The URL was #defined. (You will need to update this with your own endpoint address.)
3. Formatting the *ConnectionTime* into a string to pass to Hookbin.
4. Sending the data via cURL, passing the URL and string.

While *libcurl* may already be part of the installed software, there is a little more code required to perform a cURL post. This example borrowed the `http-post.c` routine provided by the cURL site. We only changed their code by:

- Converting main() into a function call curl_post()
- Replacing the hardcoded URL and data values with parameters passed to the function

Otherwise, the cURL post example provided a simple way to perform HTTP transactions.

Listing 5.14: c_04_http_to_hookbin/src/my_curl_post.cpp

```
/***************************************************************************
 *                                  _   _ ____  _
 *  Project                     ___| | | |  _ \| |
 *                             / __| | | | |_) | |
 *                            | (__| |_| |  _ <| |___
 *                             \___|\___/|_| \_\_____|
 *
 * Copyright (C) 1998 - 2015, Daniel Stenberg, <daniel@haxx.se>, et al.
 *
 * This software is licensed as described in the file COPYING, which
 * you should have received as part of this distribution. The terms
 * are also available at https://curl.haxx.se/docs/copyright.html.
 *
 * You may opt to use, copy, modify, merge, publish, distribute and/or sell
 * copies of the Software, and permit persons to whom the Software is
 * furnished to do so, under the terms of the COPYING file.
 *
 * This software is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY
 * KIND, either express or implied.
 *
 ***************************************************************************/
/* <DESC>
 * simple HTTP POST using the easy interface
 * </DESC>
 */
#include <stdio.h>
#include <curl/curl.h>

#include "my_curl_post.h"

int curl_post(const char *url, const char *data)
{
    CURL *curl;
    CURLcode res;

    /* In windows, this will init the winsock stuff */
    curl_global_init(CURL_GLOBAL_ALL);

    /* get a curl handle */
    curl = curl_easy_init();

    if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL, url);
        /* Now specify the POST data */
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, data);

        /* Perform the request, res will get the return code */
        res = curl_easy_perform(curl);
        printf("\n");

        /* Check for errors */
        if(res != CURLE_OK)
            fprintf(stderr, "curl_easy_perform() failed: %s\n",
                curl_easy_strerror(res));

        /* always cleanup */
        curl_easy_cleanup(curl);
    }

    curl_global_cleanup();

    return 0;
}
```

Hint:	If you look carefully through the QuickStart's iot_monitor program, you will find they used the same method to post HTTP data.

5.2.3.4. C/C++ Example (c_05_http_put_post_get)

The original QuickStart demo does not provide easy developer access to the HTTP routines used by the example, but with a few modifications they can be used in your own programs.

The `c_05_http_put_post_get` example discards the `my_curl_post.cpp/.hpp` files from the previous example, replacing it with the `http.c/.h` files from the QuickStart *IoT_Monitor* example.

The two big difficulties in using `http.c` are:

- The HTTP post and put functions are static (i.e. not available external to the file).
- There is a bit of preparation coding required before the HTTP functions can be used.

To solve these problems, three new routines were added to http.c/.h:

- *do_http_put()*
- *do_http_post()*
- *do_http_get()*

The *main()* function in the example utilizes each of these for connecting to Hookbin.com, which was explored in the previous example. (These functions are also used in the C/C++ M2X examples in Section 5.3).

Here's a snippet of main.cpp and the results of running the example.

Listing 5.15: Chapter_05/c_05_http_put_post_get/src/main.cpp

```
81      WWANStatus(&myWwan, &myNetw);
82
83      // Create headers for HTTP transaction
84      sprintf(hdr0, "Content-Type: application/json");
85      sprintf(hdr1, "Example: Header Info");
86      h[0] = hdr0;
87      h[1] = hdr1;
88
89      // Create data for HTTP Transaction
90      sprintf(data, "Connection Time = %s\n", myNetw.connTime.c_str());
91
92      // Do HTTP PUT
93      printf("\nNow starting http put...\n");
94      ret = do_http_put(URL, 2, h, data);
95
96      // Do HTTP POST
97      printf("\nNow starting http post...\n");
98      ret = do_http_post(URL, 2, h, data, rbuf);
99      printf("%s\n", rbuf);
100
101     // Do HTTP GET
102     printf("\nNow starting http get...\n");
103     do_http_get(URL, 2, h, data, response, sizeof(response));
104     printf("Response: %s\n", response);
105
106     printf("\nProgram is exiting... \n");
107     return 0;
108 }
```

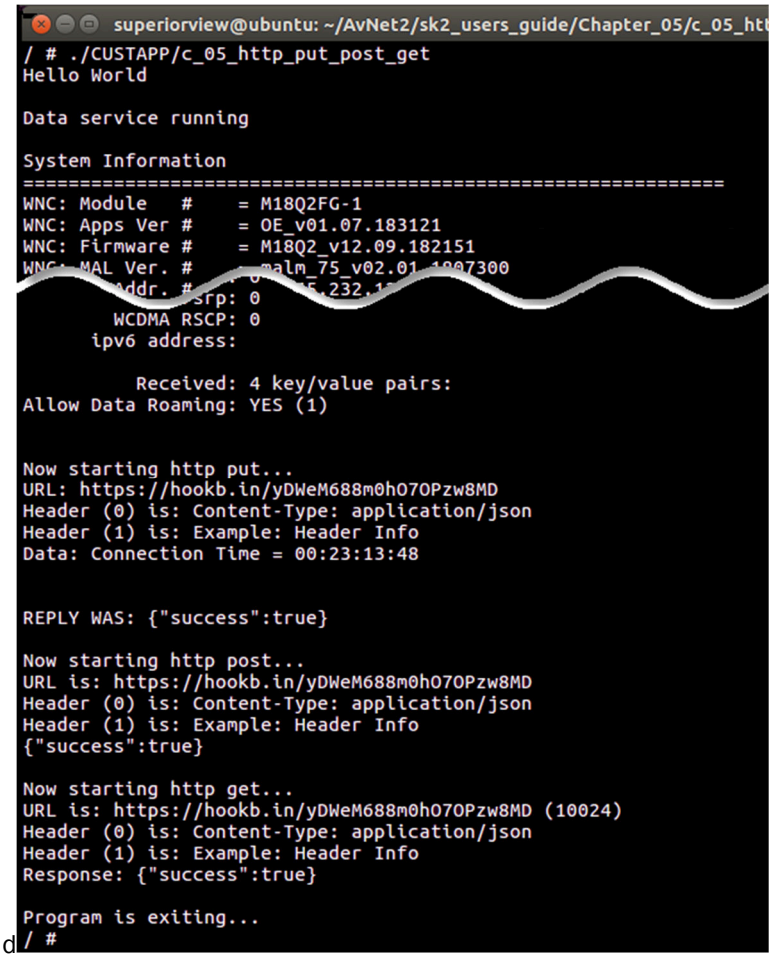


Figure 5.10 - c_05_http_put_post_get output

The results at Hookbin.com are like the previous example, although you will see GET (highlighted below) and PUT results, as well as the POST version seen earlier. The differences aren't remarkable in this context, but it shows we were able to communicate using three HTTP transaction types.

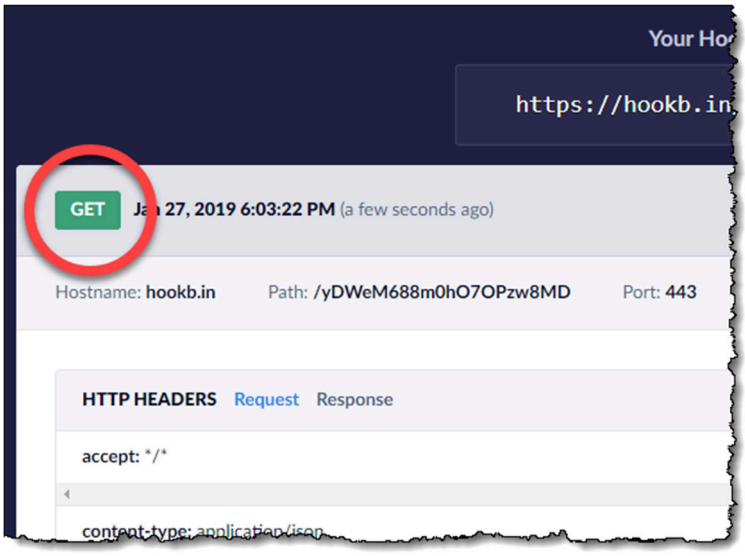


Figure 5.11 - c_05_http_put_post_get Hookbin results

5.3. Connecting to the Cloud: M2X and Flow

AT&T provides two platform tools which make it easy to get your IoT device data into the Cloud:

- **M2X**: Register and manage your devices, while providing storage for your data
- **Flow**: Build custom data flows (i.e. JavaScript programs) to transform and utilize your data

While a thorough investigation of these tools is outside the scope of this book, this section provides a brief explanation for these useful Cloud tools – as well as some code examples to get you started.

M2X

M2X is an excellent resource for IoT developers. It's device centric, which makes it intuitive to use. You can add "devices" to your account in order to represent the devices you have in the field. Within your M2X device you can add data "streams" to represent each of the data items you want to send from your device to the Cloud. For example, if you were creating a thermostat, you might have:

- **M2X Device:** Main Level Thermostat
  - **Stream 1:** Temperature Value
  - **Stream 2**: Humidity Value
  - **Stream 3:** Temperature Setting

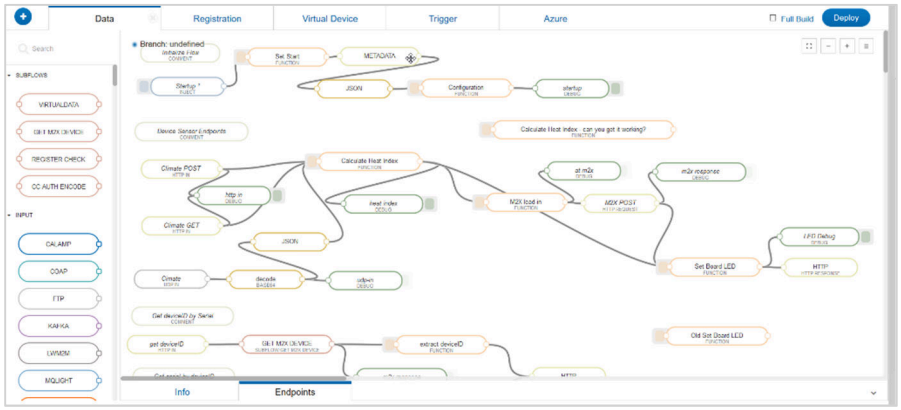
With this set, you could have your actual thermostat hardware send the three data values to your M2X device every minute – or whatever makes sense for your application requirements. (This was just an arbitrary example. We'll see how the SK2 QuickStart uses M2X in the next section.)

While M2X provides a nice graphical interface for letting you create and setup your devices and their associated data streams, this could be tedious if you are making and deploying a fleet of IoT hardware devices. To that end, M2X provides an extensive API letting you do almost everything from your keyboard... or better yet, from the IoT device itself. For example, using the QuickStart demo, your SK2 will create its own M2X device and data streams – and then send data to each stream every time you press the USER BUTTON.

To learn more about M2X, check out: https://m2x.att.com/developer/get-started

Flow

AT&T Flow is based upon Node-Red which describes itself as *"Flow-based programming for the Internet of Things"*. Being both intuitive and easy-to-use, Flow provides you with an excellent way to manage and react to your data in the Cloud.



To get started with Flow, we suggest their introductory tutorial: https://flow.att.io/help/start

5.3.1. M2X QuickStart Demo

We begin examining M2X using the QuickStart demo that ran automatically on your SK2 (when you pulled it out of the box) whenever it was powered up. (Note that it's OK if you already turned off the autorun feature of this demo, we show you how to run it from the command-line.)

As described in Chapter 1, the QuickStart demo collects sensor data from the SK2 and sends it to the Cloud whenever the USER BUTTON is pressed. Until now, most users could only visualize this by watching the RGB LED, as it indicated each of these steps in sequence. But, since the QuickStart demo sends the data to M2X, we can now view the sensor data in the M2X cloud.

5.3.1.1. Obtain a Master M2X API Key

You need to obtain a M2X authorization key in order to send data from your QuickStart app to the M2X service. You can retrieve your key by logging into M2X and accessing your account profile.

Follow these steps to access your M2X API (i.e. authorization) key.

1. **Open M2X in your web browser. (Log in, if necessary).**

   Open the following URL in your browser:

   ```
   m2x.att.com
   ```

   The upper right-corner of the service shows if you're logged in or not. Log into the service if you're not already logged in using your AT&T Marketplace USERID and Password (obtained in Chapter 1).

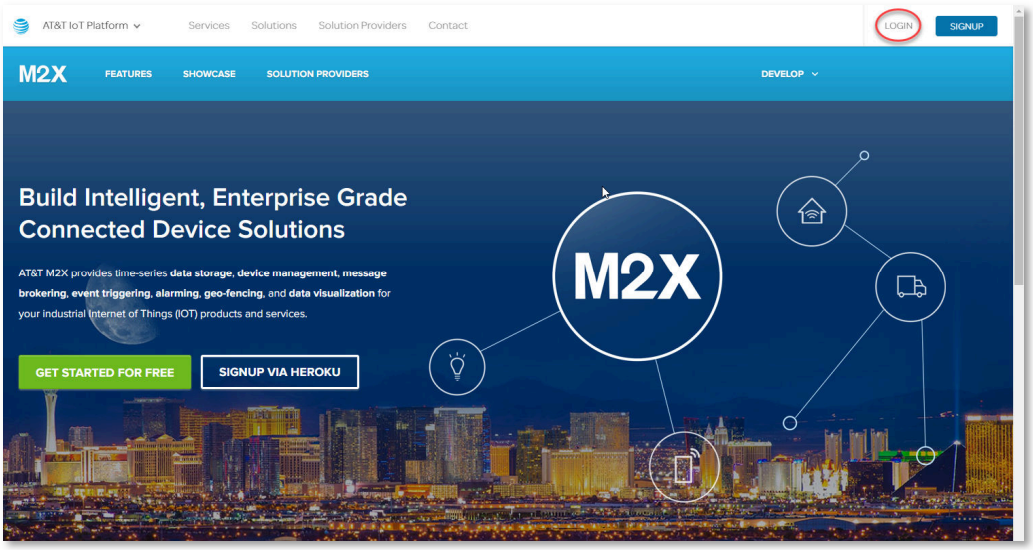


Figure 5.12 - M2X service

2. **Open your Account Settings.**

   In the upper-right corner of the web page, select:

   `Manage → Account Settings`

   as shown below.

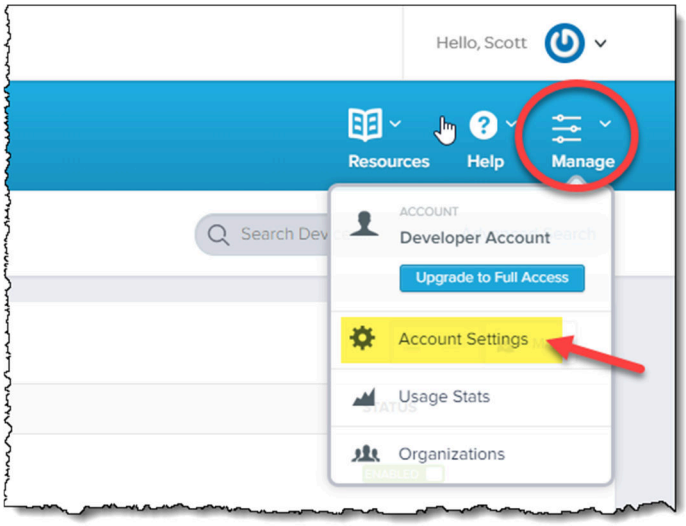


Figure 5.13 - Opening M2X Account Settings

3. **Record your M2X master API Key.**

   From the Account Settings page, record (copy) your Master API Key.

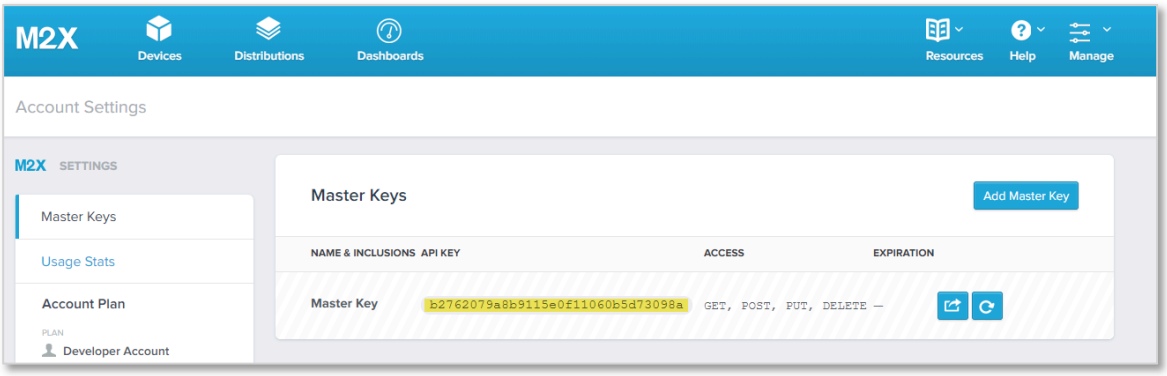


Figure 5.14 - M2X Master Key

Master Key _____

Protect your Master Key since it allows full access to your M2X account. Later on, we'll see that you can create authorization keys for individual devices, but this is the Master Key.

Hint:	If yours becomes public, like in the case above where we shared our in print, you can use the circular arrow button to the right of the key to regenerate a new key.

5.3.1.2. Run the QuickStart Demo with Your Own M2X Key

Run the QuickStart demo while passing your M2X key on the command line. This will direct the output to your M2X account rather than to the Starter Kit AMOC Dashboard.

Note: If you reimaged your SK2 for Python, you will need to rebuild and install the QuickStart demo C/C++ program (named "iot_monitor") before you can continue with the remaining steps in this section. This procedure is described in Section 3.2.

---

4. **Open an ADB remote connection to your SK2.**

```
adb shell
```

5. **Navigate to the `/CUSTAPP/iot` directory.**

   This is the location for the original QuickStart program. You may need to choose a different location if you rebuilt the program and placed it in a different location on your SK2.

6. **Copy the `run_demo.sh` script file.**

   We recommend making a copy of the original `run_demo.sh` script.

```
cp run_demo.sh run_demo.original
```

Hint: If you don't have a `run_demo.sh`, skip to the next step.

---

7. **Edit and save your `run_demo.sh`, adding your M2X API key.**

   We used vi to edit our script:

```
vi run_demo.sh
```

   Edit the file replacing the string after **"-a"** with your M2X key from Section 5.3.1.1.

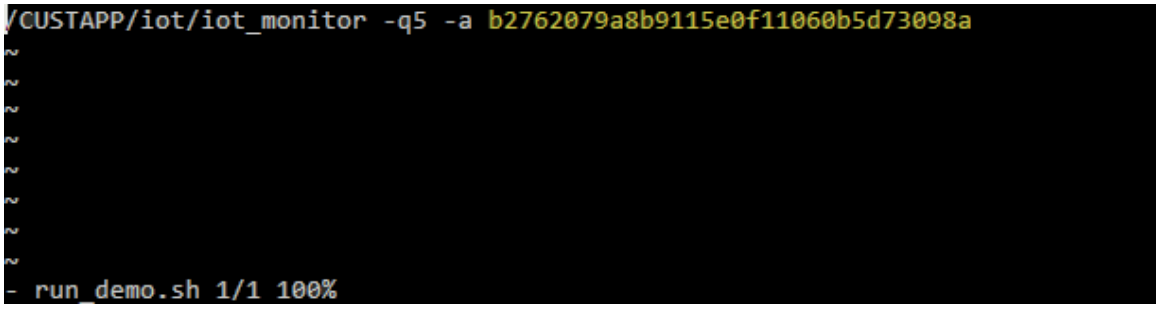


Figure 5.15 – Editing run_demo.sh

Note: If you don't have the `run_demo.sh` file, create the file and specify the correct path to your `iot_monitor` program, and include your M2X API key.

---

8. **Run the QuickStart app using the `run_demo.sh` script.**

```
./run_demo.sh
```

   This is the script that runs automatically when your SK2 is new. We can also run it manually.

9. **Send data to M2X.**

   Once the program initializes and the LED turns Green, push the USER BUTTON to send data to the Cloud.

   After the LED turns green again, go ahead and push it again to send another set of data.

10. **Exit the QuickStart demo. (Although, don't exit the demo if you want to see live data received by M2X.)**

    Following the directions printed to the terminal, you can exit the demo by pressing the USER BUTTON for more than 3 seconds.

Hint:	If you leave the program running and jump to the next section, you can continue to press the USER BUTTON and see new data posted live to M2X.

Here's the terminal display when following these instructions.

```
C:\adb
λ adb shell
/ # cd /CUSTAPP/iot/
/CUSTAPP/iot # cp run_demo.sh run_demo.original
/CUSTAPP/iot # vi run_demo.sh
/CUSTAPP/iot # ./run_demo.sh
-setting API Key to [b2762079a8b9115e0f11060b5d73098a]


Quick Start Application:
-Validating API Key and Device ID...
Create a new device.
-Creating the data streams...
Using API Key = b2762079a8b9115e0f11060b5d73098a, Device Key = 38e11298a0d1ab331bd5174a68b118be
This application will post XYZ data, Temperature data, and Light ADC data to a standard M2X
account.  To access the data, browse to this URL:

       https://api-m2x.att.com/devices/38e11298a0d1ab331bd5174a68b118be

These streams are updated continuously with user a user specfied Delay between postings.
LED colors will display a different colors after each set of sensor data is sent to M2X.

To exit the Quick Start Applicatioin, press the User Button on the Global
LTE IoT Starter Kit for > 3 seconds.

 1. Sending ADC value, TEMP value, XYZ values...All Values sent. (delay 2 seconds)
 2. Sending ADC value, TEMP value, XYZ values...All Values sent. (delay 2 seconds)


Exiting Quick Start Application.
-exiting...
/CUSTAPP/iot #
```

Figure 5.16 – Running the steps from Section 5.3.1.2

5.3.1.3. Viewing QuickStart Data in M2X

Returning to your M2X account, selecting the *Devices* button should reveal a new device named:

```
Global Starter Kit
```

When running the QuickStart demo, it automatically:

a) Added the Global Starter Kit ***device*** to your M2X account.

b) Added data ***streams*** for ADC, TEMP, XVALUE, YVALUE, and ZVALUE

Each time you press the USER BUTTON, the program sends the sensor data to the five streams.

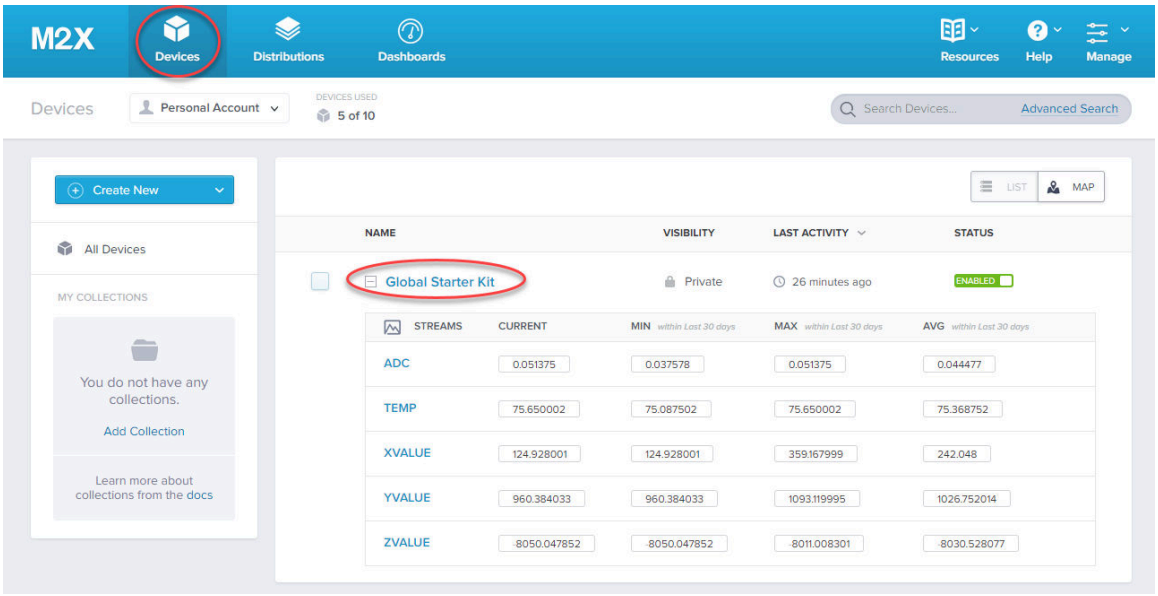


Figure 5.17 – Viewing the Global Starter Kit M2X device

Clicking on the *Global Starter Kit* link (circled above) takes you to an expanded view of your device. This view lets you view a graph of your data – or view the log of data received. It also contains the Device ID, Endpoint, Device Key, and Device Serial # values which allows you to interact with your M2X device using the M2X API.

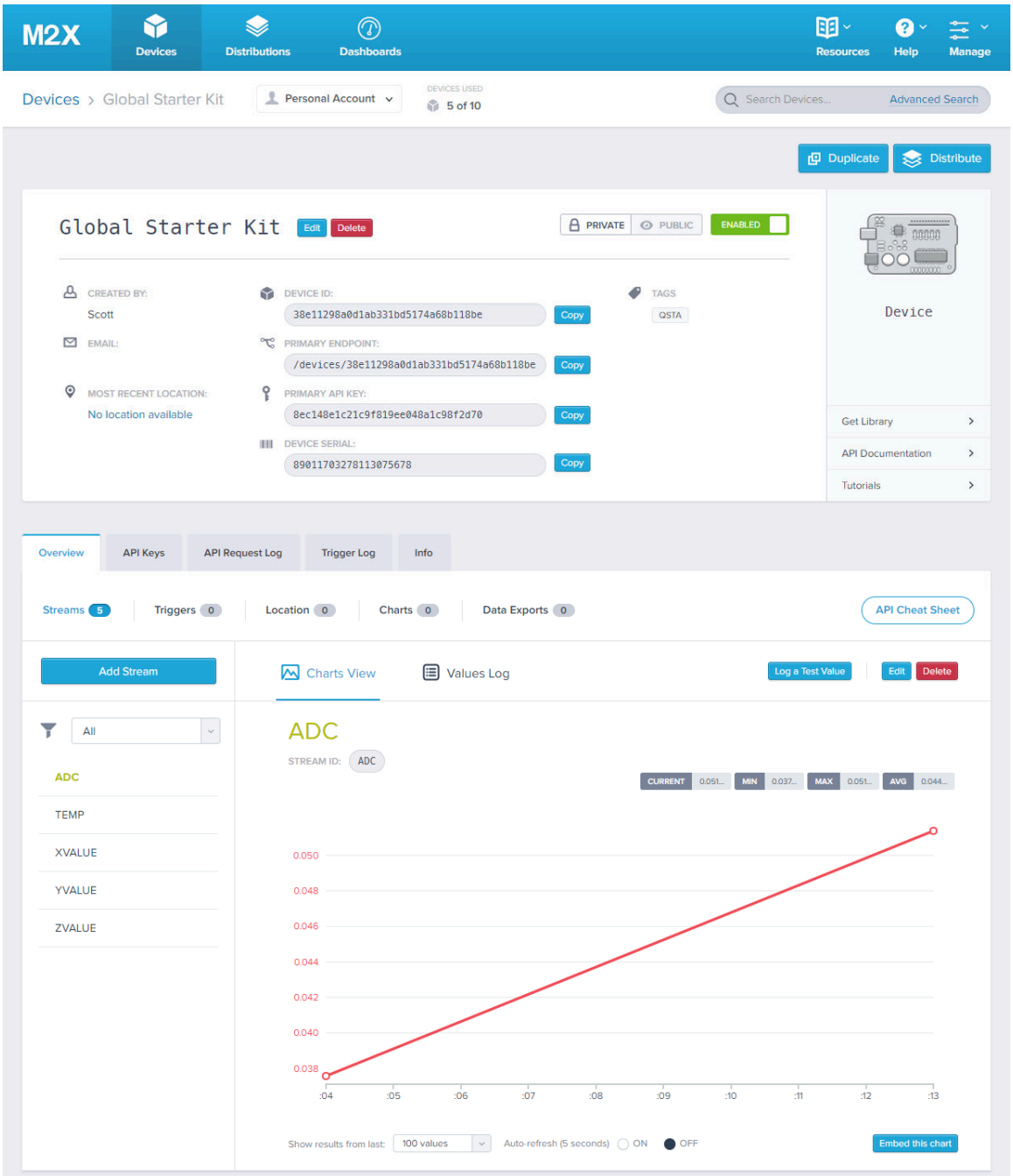


Figure 5.18 – Getting Further Details on Your Device

Hint: Clicking the "API Cheat Sheet" button (middle-right) brings up a series of cURL commands that can be used to interact with your M2X device. These commands can be run from your device – to say, post new data values – or on your computer to retrieve data stored in M2X.

5.3.2. Using M2X

At its most basic, using M2X involves creating one or more devices with streams. While there's a lot more to the service, we will limit the discussion to these features.

If you have already read the preceding parts of Section 5.3, we mentioned that you can interact with M2X in one (or all) of three ways:

- M2X Web-based GUI
- HTTP Rest API (e.g. using cURL)
- Language Specific API libraries

To make your efforts easier, the language specific API's have been ported to the SK2 for Python. We will explore M2X through the example code provided for both Python and C/C++.

5.3.2.1. Using M2X with Python

The SK2 Python IDE includes twelve examples to make it easy to work with M2X.

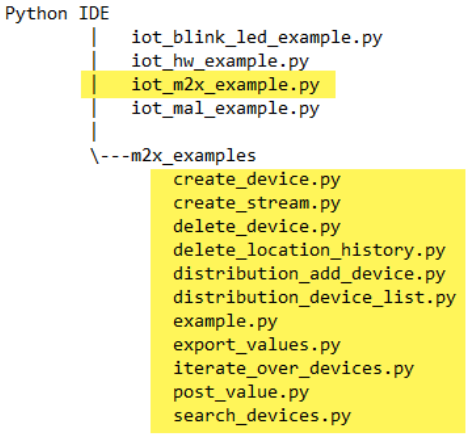


Figure 5.19 – SK2 Python IDE's M2X examples

In this discussion, we examine three of the provided M2X examples in order to help you figure out how to use them with SK2 devices.

5.3.2.1.1. Python Example (iot_m2x_example.py)

This example is designed to send the SK2's *light_sensor* value (discussed in a later chapter) to an M2X device. Before using this example, you need to create an M2X device with a stream named "light_sensor".

Set up M2X

You can create a new device from the M2X Devices dashboard by clicking the "Create New" button and selecting "Device".

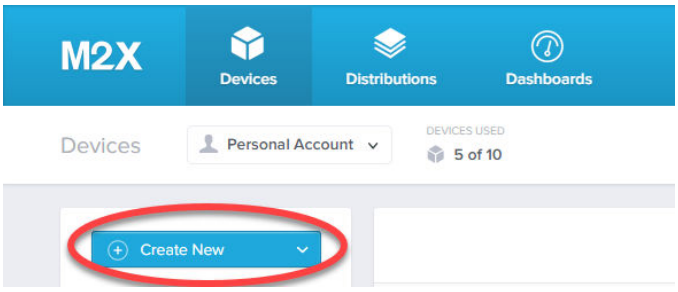


Figure 5.20 – Creating an M2X device

When the create device dialog appears, give it a name – for example, "Test" – and then click the *Create* button at the bottom of the dialog.

Scroll down the page and click on the "Add Stream" button. Set the "Stream ID" to "light_sensor" and click *Save*.

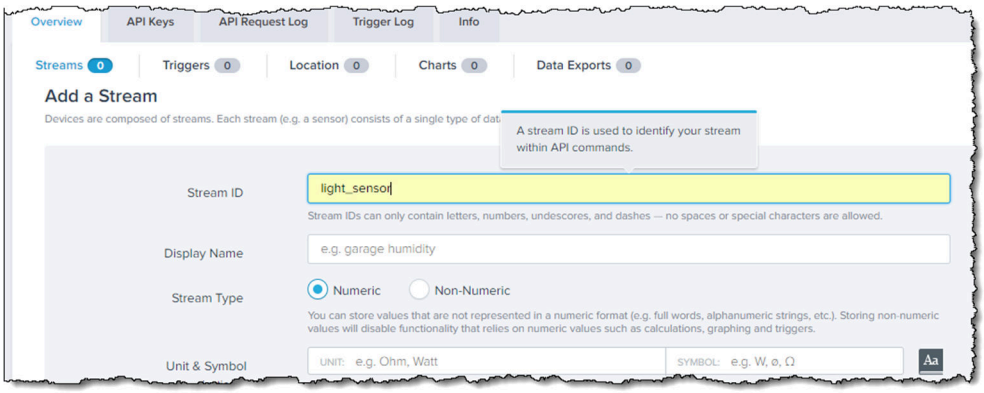


Figure 5.21 – Creating a stream

Edit `iot_m2x_example.py`

For the Python code example to find your M2X device and have the appropriate authorization, you'll need to copy two values from your M2X device into the code example. The values are highlighted below in the M2X GUI as well as in the code listing.

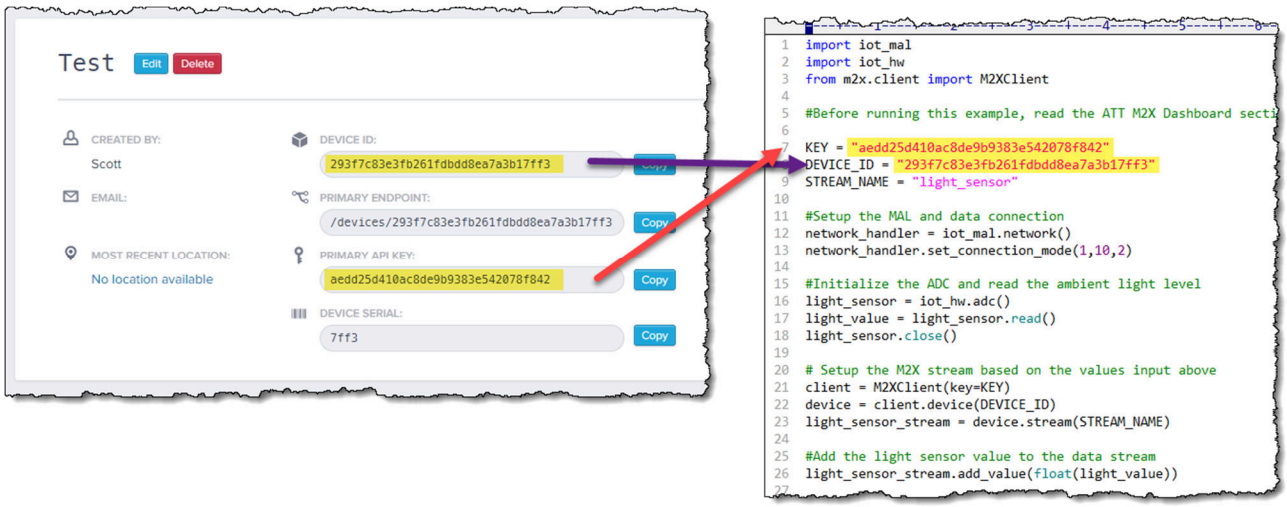


Figure 5.22 – Getting Device ID and Key from M2X "Test" Device Dashboard

Run the Example

Once you have added the two values to the appropriate Python variables, go ahead and run the Python example.

In fact, try running it multiple times while changing the amount of light hitting your SK2 board. Then view the data in M2X device log (or graph).

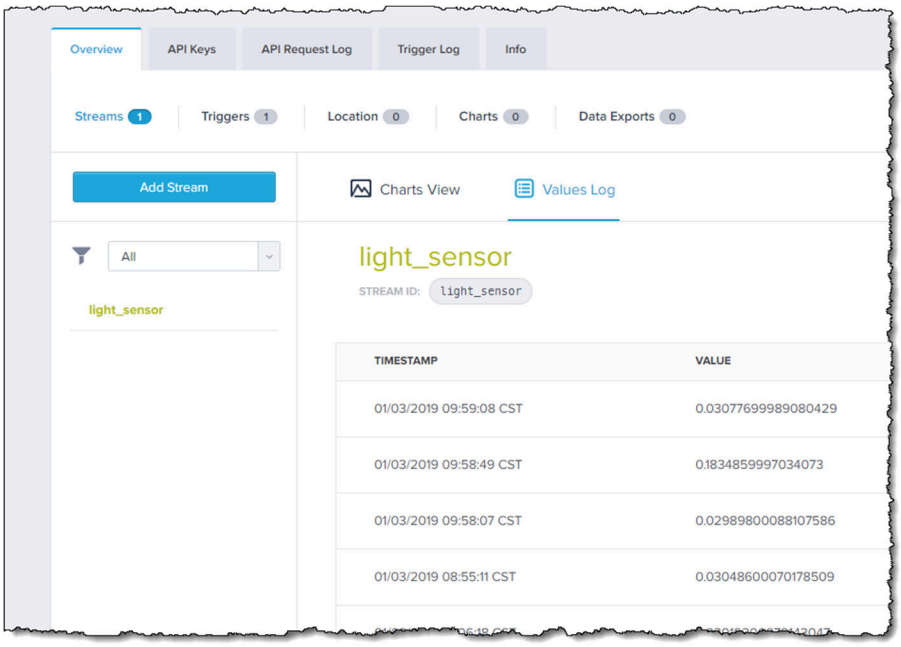


Figure 5.23 – Results after running iot_m2x_example.py multiple times

5.3.2.1.2. Python Examples (m2x examples folder)

Opening the `m2x_examples` directory in the Python IDE lists eleven examples which can be used to manage and control M2X from your SK2. In this section, we'll utilize the two highlighted in the graphic below.

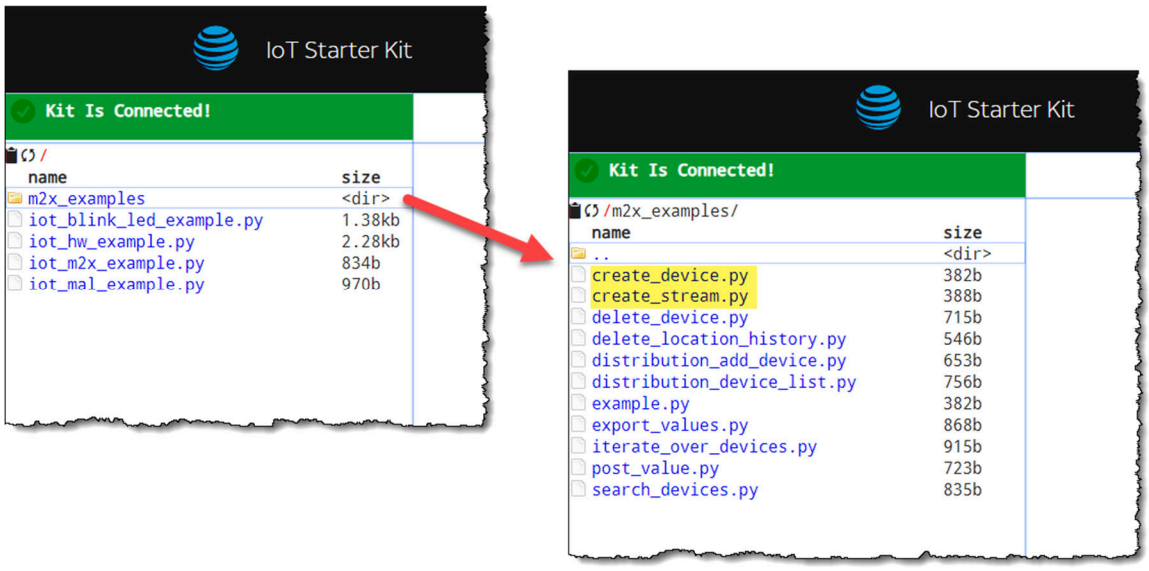


Figure 5.24 – Viewing m2x_examples folder

These examples were designed to be run from the command line – which means that we can call all them as shown below to create a new device and stream.

The comments inside each example roughly describe how the file is to be invoked. For example, `create_device.py` should be called with:

```
API_KEY=<YOUR MASTER API KEY> python example.py
```

Hint: When using the M2X API functions provided as part of the Python SDK, you may find the following resources helpful:

https://m2x.att.com/developer/documentation/v2/device
https://github.com/attm2x/m2x-python/tree/master/m2x/v2
https://github.com/attm2x/m2x-python/blob/master/USAGE.md

The following shell script provides an example for running these two Python scripts as well as posting data to the stream. Originally, we planning to use `post_value.py` to post the data values, but this file is a self-contained example creating a device, stream and then posting data. But that's OK because this allows us to show an example of using the HTTP REST API for interacting with M2X.

Listing 5.16: sh_06_m2x_example.sh

```
1  echo sh_06_m2x_example.sh
2  echo
3  echo This shell script invokes two python scripts from:  /CUSTAPP/iot_files/m2x_examples
4  echo The first Python example creates an M2X device called 'Current Time Example' and the
5  echo second example adds a stream called 'stream_name'.
6  echo
7  echo Since the 'post_value.py' example does more than just posting a value to a stream,
8  echo we chose to use cURL to add two values to the M2X device.
9  #
10 # Note: Your AT&T IoT Starter Kit (2nd generation) must have the Python image installed
11 #       for this example to work
12
13 MY_MASTER_API_KEY=b2762079a8b9115e0f11060b5d73098a      # Replace this value with the Master API Key fo
14
15 echo
16 my_device_id=$(API_KEY=$MY_MASTER_API_KEY python create_device.py)
17 echo The new device key is: $my_device_id
18
19 DEVICE_ID=$my_device_id API_KEY=$MY_MASTER_API_KEY python create_stream.py
20 echo "The 'stream_name' stream was added to the device"
21
22 echo
23 echo "Now the script will add the values '1' and '2' to the stream"
24 echo
25 echo "Adding '1'"
26 echo ---------------------------
27 curl -i -X PUT http://api-m2x.att.com/v2/devices/$my_device_id/streams/stream_name/value -H "X-M2X-KEY:
28
29 echo
30 echo
31 echo "Adding '2'"
32 echo ---------------------------
33 curl -i -X PUT http://api-m2x.att.com/v2/devices/$my_device_id/streams/stream_name/value -H "X-M2X-KEY:
34
```

Running the shell script results in the following output:

```
C:\adb
λ adb push sh_06_m2x_example.sh /CUSTAPP/iot_files/m2x_examples
sh_06_m2x_example.sh: 1 file pushed. 0.4 MB/s (1526 bytes in 0.004s)

C:\adb
λ adb shell
/ # cd /CUSTAPP/iot_files/m2x_examples/
/CUSTAPP/iot_files/m2x_examples # chmod +x sh_06_m2x_example.sh
/CUSTAPP/iot_files/m2x_examples # ./sh_06_m2x_example.sh
This shell script invokes two python scripts from: /CUSTAPP/iot_files/m2x_examples
The first Python example creates an M2X device called Current Time Example and the
second example adds a stream called stream_name.

Since the post_value.py example does more than just posting a value to a stream,
we chose to use cURL to add two values to the M2X device.

The new device key is: 3e80adc5c8e40f32a0d08c9198aa536f
The 'stream_name' stream was added to the device

Now the script will add the values '1' and '2' to the stream

Adding '1'
---------------------------
HTTP/1.1 202 Accepted
Content-Type: application/json; charset=UTF-8
X-M2X-VERSION: v2.112.2
Vary: Accept-Encoding
X-RateLimit-Limit: 10
X-RateLimit-Remaining: 8
X-RateLimit-Reset: 1548462456
Content-Length: 21

{"status":"accepted"}

Adding '2'
---------------------------
HTTP/1.1 202 Accepted
Content-Type: application/json; charset=UTF-8
X-M2X-VERSION: v2.112.2
Vary: Accept-Encoding
X-RateLimit-Limit: 10
X-RateLimit-Remaining: 9
X-RateLimit-Reset: 1548462457
Content-Length: 21

{"status":"accepted"}
/CUSTAPP/iot_files/m2x_examples #
```

Figure 5.25 – sh_06_m2x_example.sh Output

5.3.2.1.3. Python Example (m2x_examples/example.py)

Once you have a device that contains some stream data the `example.py` code will print out information about the device.

Before you try out `example.py` beware of a few caveats:

- The *pprint* module is not installed, therefore you need to comment out "import pprint" and change the *pprint()* function to *printf()*.
- The command-line arguments for this example are different than the other examples. Rather than "API_KEY" this example calls it "KEY". Similarly, "DEVICE_ID" is now "DEVICE".

Here's simple script to try out, making sure you fill in your own DEVICE and KEY (and correct the items from above).

Listing 5.17: ex_08_m2x_print_values.sh

```
# Note: Your AT&T IoT Starter Kit (2nd generation) must have the Python image installed
#       for this example to work

M2X_EXAMPLES_PATH="/CUSTAPP/iot_files/m2x_examples"   # Set location of your m2x_examples dir
MASTER_API_KEY=b2762079a8b9115e0f11060b5d73098a       # Replace the KEY with yours
M2X_DEVICE_ID="293f7c83e3fb261fdbdd8ea7a3b17ff3"      # Need to replace this one, too

DEVICE=$M2X_DEVICE_ID KEY=$MY_MASTER_API_KEY python $M2X_EXAMPLES_PATH/example1.py
```

5.3.2.2. Using M2X with C/C++

The C/C++ M2X library has not been ported over to the SK2, but there are still plenty of ways to communicate with the IoT service.

- Use HTTP transfers – which is what is shown in our code example below.
- Run Linux cURL utility from within your C/C++ programs

The QuickStart demo includes a few M2X functions which are leveraged by our example. In those cases where specific M2X functions are not available – or were too proprietary to the needs of the QuickStart demo – this example borrows from HTTP Get/Put/Post functionality.

5.3.2.2.1. C/C++ Example (c_07_m2x_create_device_stream_value)

Given the following information:

```
#define M2X_API_KEY          "b2762079a8b9115e0f11060b5d73098a"
#define M2X_DEVICE_NAME      "CTEST"
#define M2X_STREAM_NAME      "temp"
```

This example obtains the System, WWAN, and Network status info (same as examples from Section 5.1), then implements the following M2X tasks:

- Given a Master M2X_API_KEY, gets a list of M2X devices for that account.
- Creates a new M2X device for M2X_DEVICE_NAME, if it doesn't already exist for this account.
- Gets the streams available for M2X_DEVICE_NAME.
- Creates a new stream for M2X_STREAM_NAME if it wasn't on the downloaded list of streams.
- Writes the value '2' (arbitrary value) to M2X_STREAM_NAME.

To use this example, you need to:

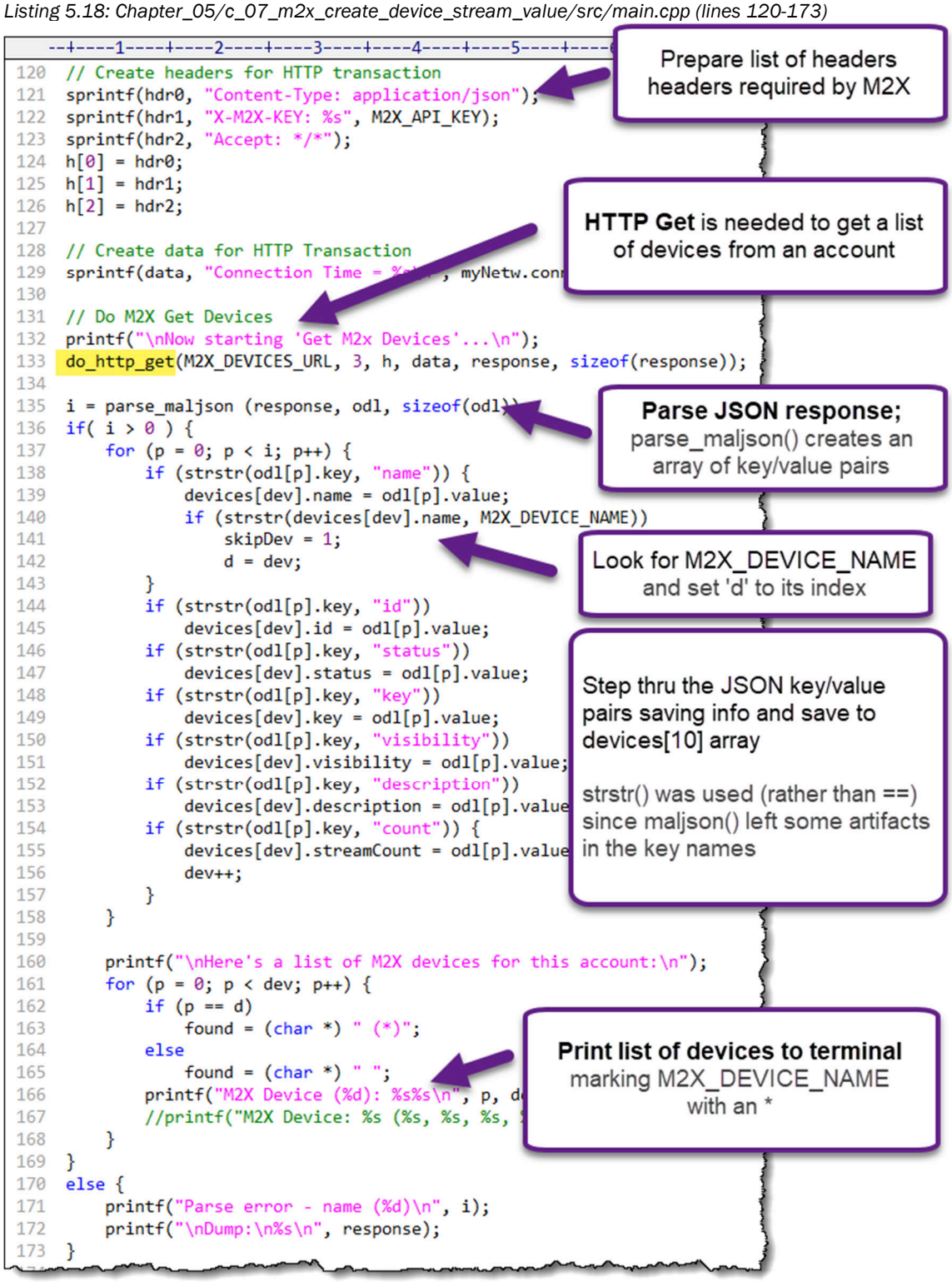
- Create an M2X account. (If you don't already have one.)
- Replace the value for M2X_API_KEY with the value from your account. (Find this key as described in Section 5.3.1.1).

While this example is longer than some of the others we have discussed, it breaks down nicely into manageable parts. The next four subsections describe each of these parts.

5.3.2.2.1.1. Get List of M2X Devices

This example uses HTTP get to access the list of devices for a user's account. The API help for this can be found at https://m2x.att.com/developer/documentation/v2/device#List-Devices.

Listing 5.18: Chapter_05/c_07_m2x_create_device_stream_value/src/main.cpp (lines 120-173)




```
120  // Create headers for HTTP transaction
121  sprintf(hdr0, "Content-Type: application/json");
122  sprintf(hdr1, "X-M2X-KEY: %s", M2X_API_KEY);
123  sprintf(hdr2, "Accept: */*");
124  h[0] = hdr0;
125  h[1] = hdr1;
126  h[2] = hdr2;
127
128  // Create data for HTTP Transaction
129  sprintf(data, "Connection Time = %d", myNetw.con
130
131  // Do M2X Get Devices
132  printf("\nNow starting 'Get M2x Devices'...\n");
133  do_http_get(M2X_DEVICES_URL, 3, h, data, response, sizeof(response));
134
135  i = parse_maljson (response, odl, sizeof(odl));
136  if( i > 0 ) {
137      for (p = 0; p < i; p++) {
138          if (strstr(odl[p].key, "name")) {
139              devices[dev].name = odl[p].value;
140              if (strstr(devices[dev].name, M2X_DEVICE_NAME))
141                  skipDev = 1;
142                  d = dev;
143          }
144          if (strstr(odl[p].key, "id"))
145              devices[dev].id = odl[p].value;
146          if (strstr(odl[p].key, "status"))
147              devices[dev].status = odl[p].value;
148          if (strstr(odl[p].key, "key"))
149              devices[dev].key = odl[p].value;
150          if (strstr(odl[p].key, "visibility"))
151              devices[dev].visibility = odl[p].value;
152          if (strstr(odl[p].key, "description"))
153              devices[dev].description = odl[p].value
154          if (strstr(odl[p].key, "count")) {
155              devices[dev].streamCount = odl[p].value
156              dev++;
157          }
158      }
159
160      printf("\nHere's a list of M2X devices for this account:\n");
161      for (p = 0; p < dev; p++) {
162          if (p == d)
163              found = (char *) " (*)";
164          else
165              found = (char *) " ";
166          printf("M2X Device (%d): %s%s\n", p, d
167          //printf("M2X Device: %s (%s, %s, %s, 
168      }
169  }
170  else {
171      printf("Parse error - name (%d)\n", i);
172      printf("\nDump:\n%s\n", response);
173  }
```

5.3.2.2.1.2. Create M2X Device (if needed)

M2X uses HTTP post to create a new device. This is only done if that device name doesn't already exist in the list we just downloaded. The API help for this can be found at https://m2x.att.com/developer/documentation/v2/device#Create-Device.

Listing 5.19: Chapter_05/c_07_m2x_create_device_stream_value/src/main.cpp (lines 176-210)

```
176  // Do Create Device
177  printf("\nNow starting 'Create M2X Device' (%d)...\n", skipDev);
178
179  if (skipDev) {
180      printf("Device %s already exists, so it won't be created (dev=%d)(d=%d)\n", M2X_DEVICE_NAME, dev, d);
181  }
182  else {
183      devices[dev].name =        (char *) M2X_DEVICE_NAME;
184      devices[dev].visibility =  (char *) M2X_VISIBILITY;
185      devices[dev].description = (char *) M2X_DEVICE_DESCR;
186
187      sprintf(data, "{ \"name\": \"%s\", \"description\": \"%s\", "
188                    "\"visibility\":\"%s\" }",  devices[dev].name, devices[dev].description, devices[dev].visibility);
189      //printf("%s\n", data);
190
191      ret = do_http_post(M2X_DEVICES_URL, 2, h, data, rbuf);
192
193      i = parse_maljson (rbuf, od, sizeof(od));
194      if( i > 0 ) {
195
196          for (p = 0; p < i; p++) {
197              if (strstr(od[p].key, "id")) {
198                  devices[dev].id = od[p].value;
199                  printf("\nNew M2X Device name (id): %s (%s)\n", devices[dev].name, devices[dev].id);
200                  d = dev;
201              }
202          }
203      }
204      else {
205          printf("\nParse error - id\n");
206          printf("%s\n", rbuf);
207      }
208
209      dev++;
```

Skip this task if the device already exists (i.e. skipDev > 0)

Add new device's info to devices[ ] array
Assuming it might be needed later on...

Create 'data' payload with new device information for HTTP post transaction

Do the HTTP post to create the device

Parse the response and print the new device's Name and ID

Since we just added a new device, increment the device count

5.3.2.2.1.3. Get the Device's Stream Names

Like earlier when getting the list of devices, HTTP get is used to download the list of streams available for the specified device. The API help for this can be found at https://m2x.att.com/developer/documentation/v2/device#List-Data-Streams.

Listing 5.20: Chapter_05/c_07_m2x_create_device_stream_value/src/main.cpp (lines 212-248)

```
212  // Get the device's stream names
213  printf("\nNow starting 'Get M2x Streams' (%d)...\n", d);
214
215  memset(url, 0, sizeof(url));
216  sprintf(url, "%s/%s/streams", M2X_DEVICES_URL, devices[d].id);
217  //printf ("URL: %s\n", url);
218
219  do_http_get(url, 3, h, data, response, sizeof(response));
220
221  i = parse_maljson (response, osl, sizeof(osl));
222  if( i > 0 ) {
223      for (p = 0; p < i; p++) {
224          if (strstr(osl[p].key, "display_name")) {
225              //printf("Display name\n");
226          } else if (strstr(osl[p].key, "name")) {
227              streams[str].name = osl[p].value;
228              if (strstr(streams[str].name, M2X_STREAM_NAME))
229                  skipStream = 1;
230                  s = str;
231              }
232              str++;
233          }
234      }
235
236      printf("\nHere's a list of streams (%d) for this device (%s):\n", str, devices[d].name);
237      for (p = 0; p < str; p++) {
238          if (p == s)
239              found = (char *) " (*)";
240          else
241              found = (char *) " ";
242          printf("M2X Stream: %s%s\n", streams[p].name, found);
243      }
244  }
245  else {
246      printf("Parse error - name (%d)\n", i);
247      printf("\nDump:\n%s\n", response);
248  }
```

Build the 'url'
which includes the Device ID

Use HTTP Get to download the list of device streams

Parse the response to find the stream names

Print the list of streams
highlight M2X_STREAM_NAME if it exists

5.3.2.2.1.4. Create the Stream, then Add a Value to It

The SK2 QuickStart demo code already contains generic functions for creating a stream and another for adding a value to it. Like the HTTP get/put/post functions, they are in the http.h/http.c files.

Listing 5.21: Chapter_05/c_07_m2x_create_device_stream_value/src/main.cpp (lines 251-269)

```
251     // Do Create Stream
252     printf("\nNow starting 'Create M2X Stream' (%d)...\n", d);
253
254     if (skipStream) {
255         printf("Stream %s (for device=%s) already exists, so it won't be created (str=%d)(d=%d)(s=%d)\n", M2X_STREAM_N
256     }
257     else {
258         ret = m2x_create_stream( devices[d].id, M2X_API_KEY, M2X_STREAM_
259         printf("Create stream return value: %d\n", ret);
260     }
261
262     // Add Value to the Stream
263     ret = m2x_update_stream_value( devices[d].id, M2X_API_KEY, M2X_STREAM_NAME, "2");  // '2' was chosen arbitrarily
264     printf("Return from update stream v
265
266     printf("\nProgram is exiting... \n");
267     return 0;
268 }
269
```

Don't create the stream, if it already exists

QuickStart iot_monitor demo has m2x_create_stream() function we can use

Similarly the demo includes an m2x_update_stream_value()

5.3.2.2.1.5. Results and Terminal Output

After executing this program successfully, your M2X account will contain a device (named 'CTEST') with a stream named 'temp'. Whether these existed before or not, the value '2' will be added to 'temp'.

Here's a record from running the program on our SK2:

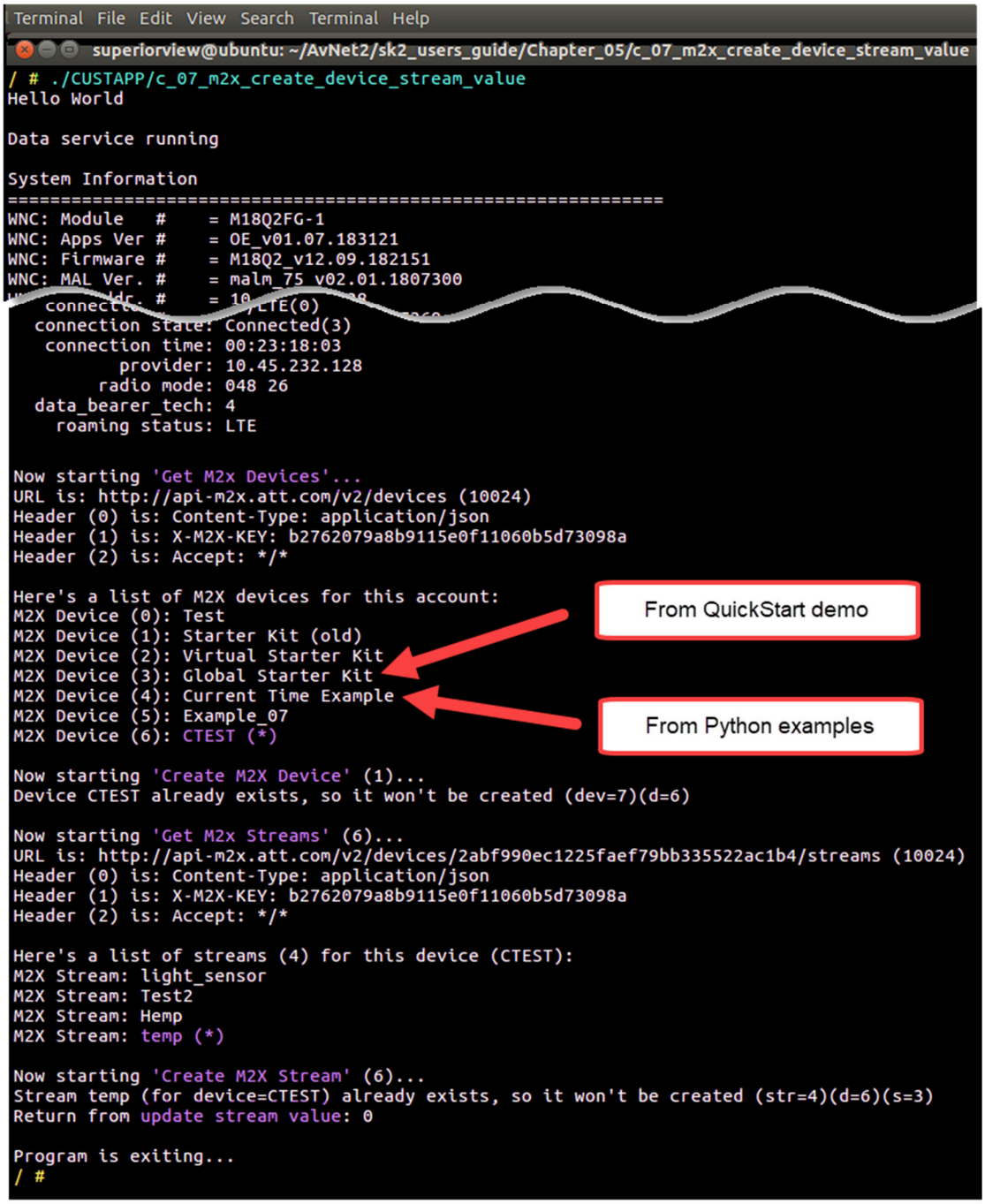


Figure 5.26 – Results from running c_07_m2x_create_device_stream_value

Reading through this output, it appears that we had already run the program at least once before this execution because we see that both the device and stream already existed.

5.3.3. Flow

AT&T Flow, as is its progenitor *Node-Red*, is an excellent rapid-prototyping tool, as well as a production environment.

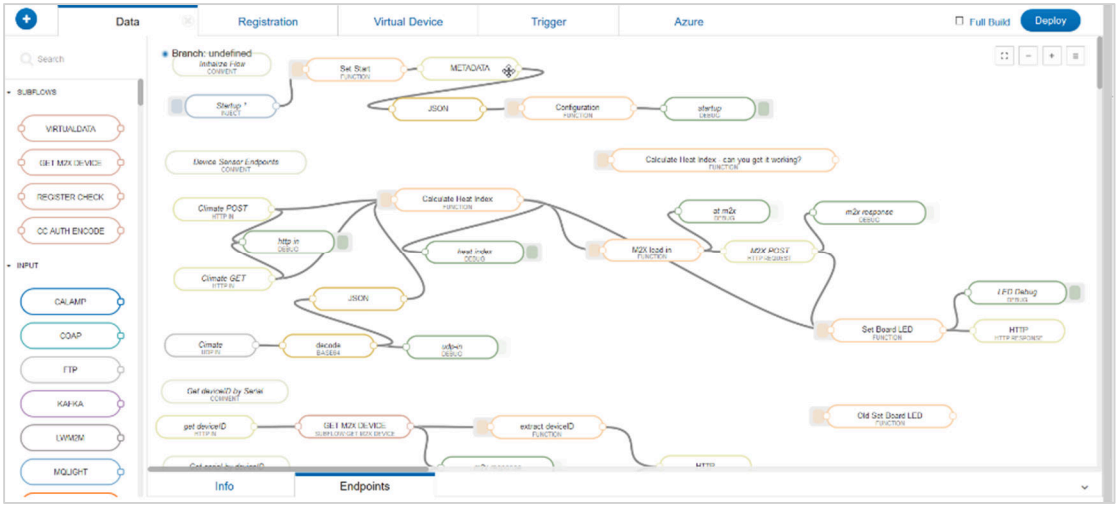


Figure 5.27 – The AT&T Flow Tool

The visceral, drag-and-drop environment is intuitive, making it fun and easy to use. Behind the scenes, once you hit the *Deploy* button, your design begins running in the 'Cloud' without any further effort required. Thankfully, we don't need to worry about the details of server allocation, load balancing, and all that IT stuff which can slow you down on other platforms.

This section only explores the very basic elements of Flow – that is, sending data to Flow and getting a response back. Later, in Section 5.4, we explore how to send SMS messages from Flow.

Hint:	At the time of writing, we had problems opening Flow in Google Chrome. Though, it works fine when using Firefox.

5.3.3.1. Starting up Flow

Head to https://flow.att.io to open Flow. Like M2X, Flow uses the same login settings as your AT&T Marketplace account.

Flow opens to the *Flow Dashboard* where you can see the previous projects you've created.

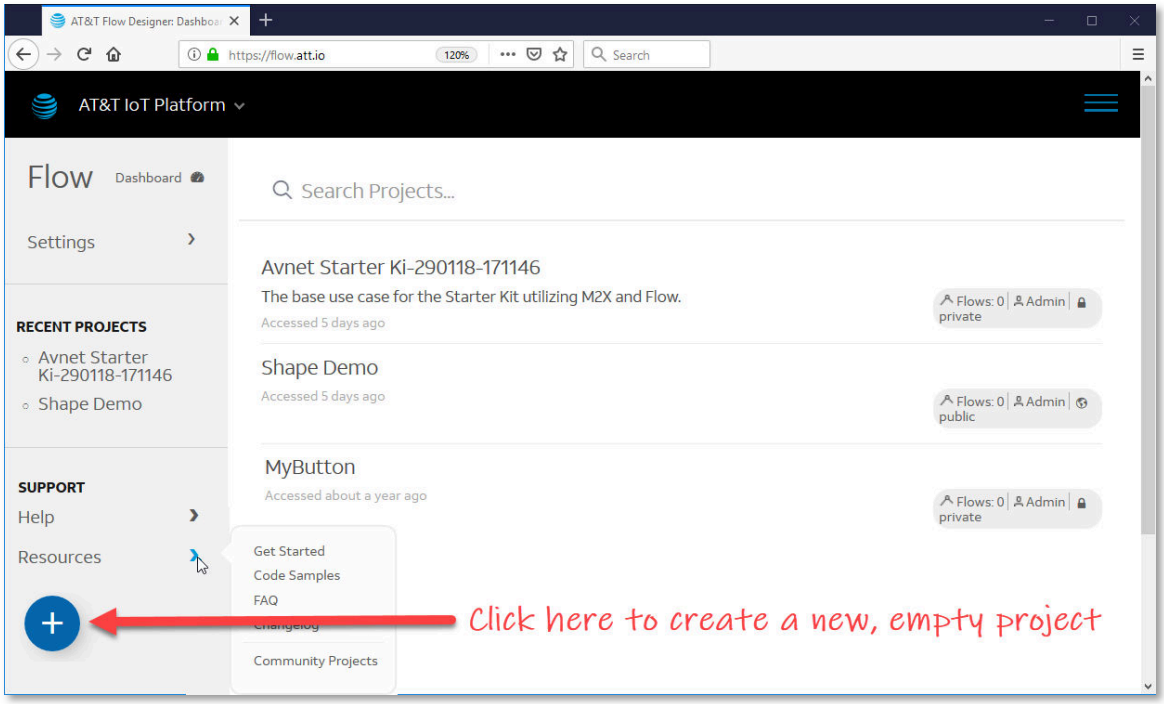


Figure 5.28 – Starting a new Flow

If you're new to flow, you may want to go check out the ***Resources → Get Started*** tutorial.

Additionally, under *Resources* you will also find *Code Examples*, although most of them are not well suited for beginners. You can also access the *FAQ* and *Community Projects* from here. *(The Community Projects may also be difficult for new users. If you are looking for beginner-ware, you may want to do an Internet search for Node-Red beginner projects.)*

To get started with a blank canvas, click the blue **+** button. (We'll do this in the next section.)

5.3.3.2. Flow Example – Timestamp/Debug (ex_09_timestamp_debug_flow.json)

After starting Flow, we'll create a simple, starter project. While simple, we'll build on it with our SK2.

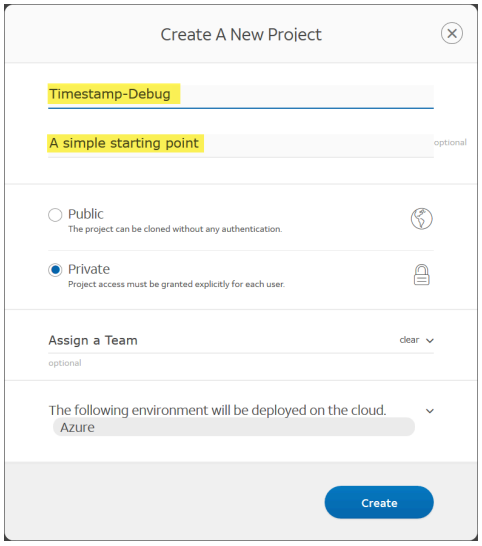
Hint: This Flow example can be imported using the `ex_09_timestamp_debug_flow.json` file found in the SK2 User's Guide GitHub site. Please refer to Section 5.3.3.3.2 for "how to" import Flow projects.

1. **Create a new Flow project.**

   Click the blue + button in the lower-right corner of the Flow Dashboard, as shown in Figure 5.28.

2. **Complete the *New Project* dialog and click *Create*.**

   We recommend calling the project: *Timestamp-Debug*.



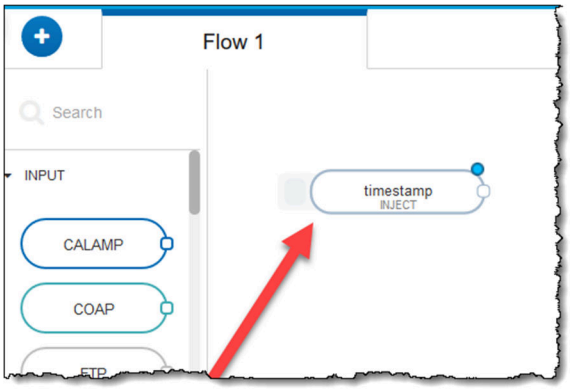
   The Flow canvas should open after the project is created (which may take a few seconds).

Note: While we explain many features about the Flow canvas when though our example, it's outside the scope of this user guide to provide an in-depth explanation of Flow.

3. **Add the *Inject* node to the Flow canvas.**

   The palette to the left of the canvas contains a wide range of nodes (bubbles, functions, or whatever you want to call them). The nodes are grouped – for example, the first ones are *Inputs* and the second group are *Outputs*.

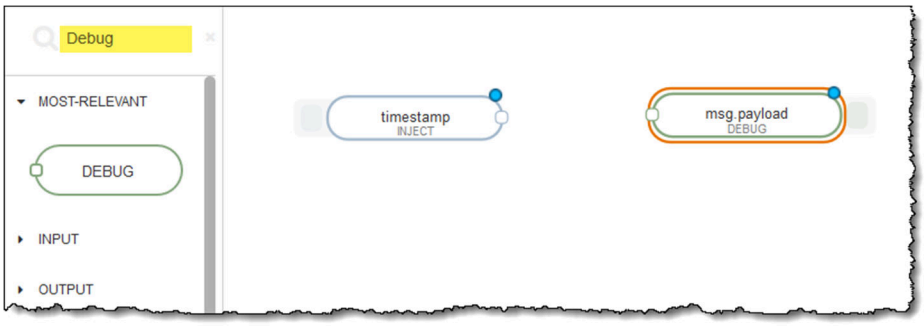
   Drag and drop the ***Inject*** node (from Inputs) to the canvas. (It's about 15 nodes down the list.) After dropping the node, it should say "**timestamp**" with "INJECT".



4. **Add the Debug node to the Flow canvas.**

   Next, let's add the Debug node from the Output group.

Hint:	Type *Debug* into the node search box to filter down the long list of nodes.

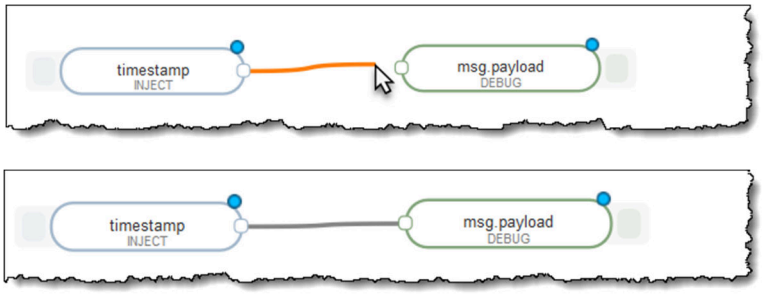


After filtering the list using the *Search* box (as shown above), drag *Debug* onto the canvas. Note that its name will change to "msg.payload" with a small "DEBUG". The node is telling us that it will write the payload of the incoming message (i.e. msg.payload) to the *Debug* window. (Which we'll see shortly.

While you can change the names of the nodes on the canvas, as well as other options, we recommend leaving them as is, for now.

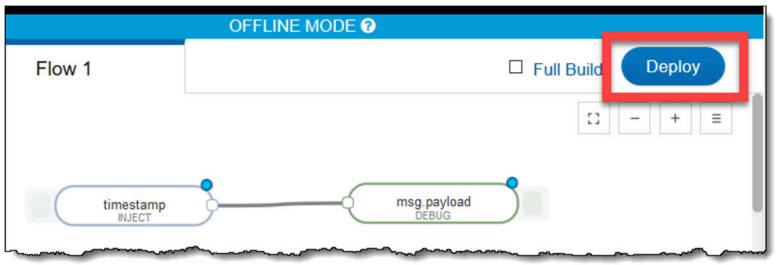
5. **Connect the two nodes – Inject output to Debug input.**

   Start by clicking the white output bubble on the right-side of the Inject node… and then dragging to the white input bubble on the Debug node.



6. **Deploy your design.**

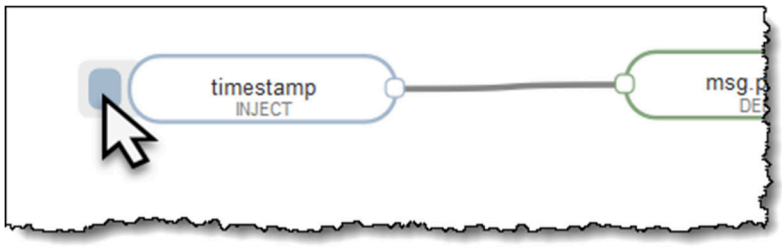
   Click the **Deploy** button in the upper-right corner. This turns your design into a real-world cloud server application.



Note that the first time you deploy a design takes a couple of minutes to complete. Subsequent deploys (after you make changes to your design) go quickly.

7. **Click the Inject button (on the left side of the node).**

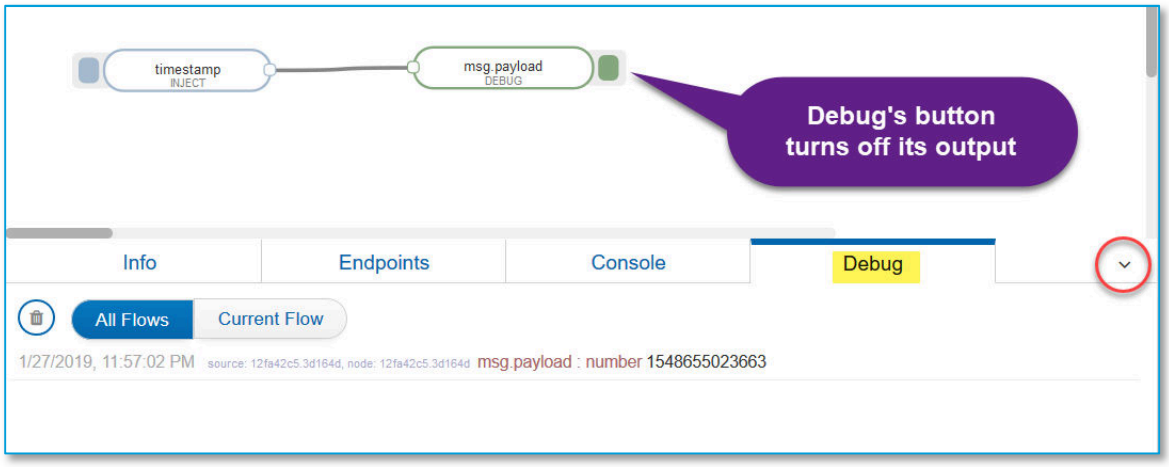
   Clicking this button will inject (i.e. send out) a timestamp value. More specifically, the node will output a JSON message that contains a payload with the current timestamp.



8. **Open the Debug window to view Inject node's output.**

   The *Debug* tab displays the output from Debug nodes (all Debug nodes, if you have more than one in your flow).

   Clicking the arrow (highlighted by the red circle) opens and closes the tabbed windows at the bottom of the display. You may need to click it to view the *Debug* window contents.



   You can toggle a Debug node on/off by clicking the button on its right side. This can be especially helpful if you have multiple Debug nodes in your Flow project. In fact, try clicking the Inject button with the Debug node turned off.

5.3.3.3. Exporting/Importing Flows

We can export a copy of our Flow project.

Even though Flow projects are stored in your account. They can be shared and linked to Git repositories. But sometimes you might want to save a copy to your computer. Or export it so that you can send it to a colleague.

Importing and Exporting are two of the options on the menu under the *Deploy* button.

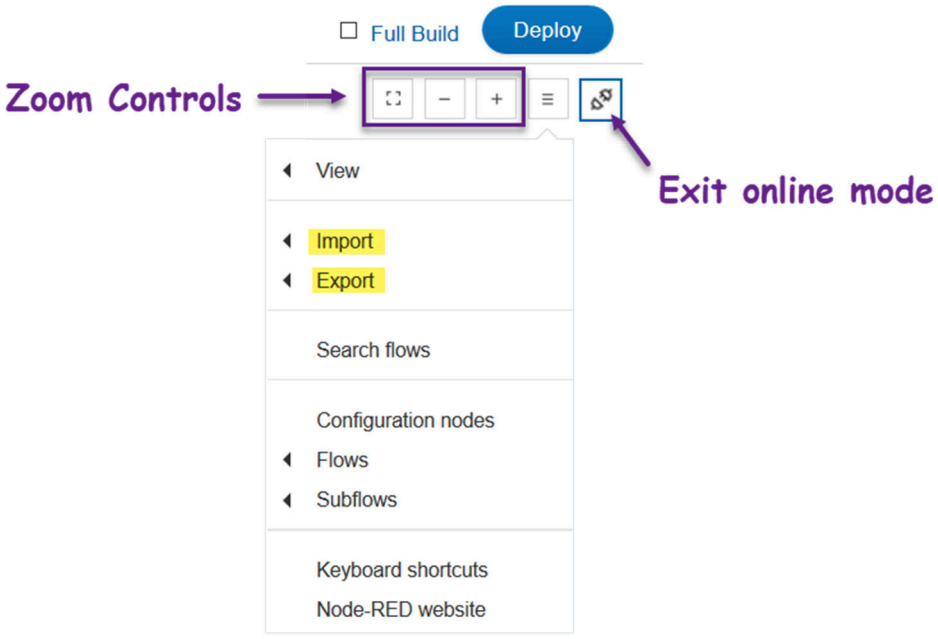


Figure 5.29 – Flow project canvas controls

5.3.3.3.1. Export procedure

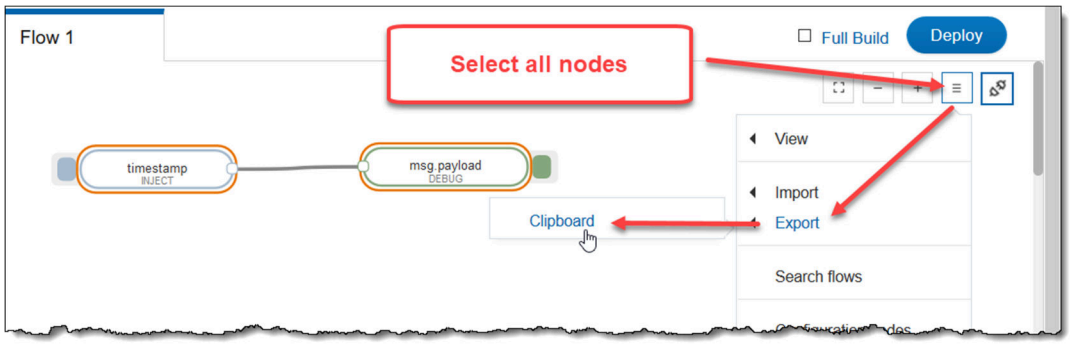
1. **Select all nodes (or at least the ones you want to export).**

   Select the nodes by dragging your mouse over them. Or better yet, using **Control-A**.

2. **Select the Export to Clipboard option.**

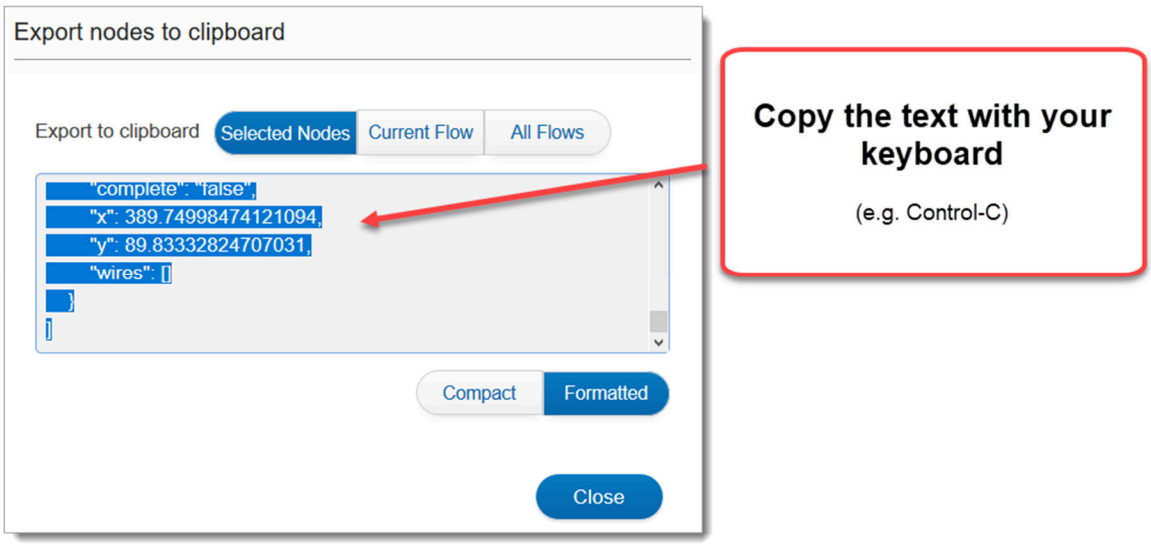
   Click the hamburger menu (i.e. button with three horizontal lines), then *Export*, and then *Clipboard*.

   ```
   Menu → Export → Clipboard
   ```



3. **Copy the selected JSON text using your keyboard. (e.g. Control-C).**

   After you click *Clipboard* in step 2, the following dialog appears.



   Export caveats:

   - At least one node must be selected to enable the Export menu item.
   - Text is not automatically sent to the clipboard.
   - The text must be **selected** in order to be copied.
   - Use either a keyboard shortcut or the right-click context menu to copy the text.
     - It appears like you cannot copy the text with the right-click menu because a red warning sign ⃠ appears at the cursor, but it does let you copy the selected text.
   - Clicking the dialog buttons appeared to de-select the text but clicking the mouse over the text re-selects it.

4. **Paste and save the copied text to a JSON file.**

   You can save the text in any manner you prefer, but we saved our previous example to:

   `ex_09_timestamp_debug_flow.json`

The resulting file looks like:

Listing 5.22: ex_09_timestamp_debug_flow.json

```
[
    {
        "id": "368b3913.e59f1e",
        "type": "inject",
        "z": "7431afd4.02d828",
        "name": "",
        "topic": "",
        "payload": "",
        "payloadType": "date",
        "repeat": "",
        "crontab": "",
        "once": false,
        "x": 142.75,
        "y": 91.83332824707031,
        "wires": [
            [
                "12fa42c5.3d164d"
            ]
        ]
    },
    {
        "id": "12fa42c5.3d164d",
        "type": "debug",
        "z": "7431afd4.02d828",
        "name": "",
        "active": true,
        "console": "false",
        "complete": "false",
        "x": 389.74998474121094,
        "y": 89.83332824707031,
        "wires": []
    }
]
```

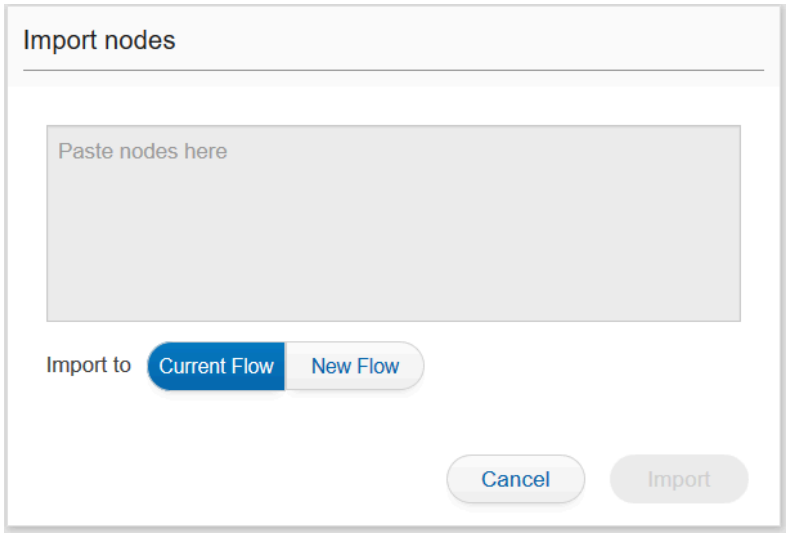
5.3.3.3.2. Import Procedure

Importing nodes is like exporting them.

1. **Using the 'hamburger' menu in the upper-right corner, import from clipboard.**

   ☰ → Import → Clipboard

2. **Paste the JSON node description into the *Import nodes* dialog, then click *Import*.**



3. **Place the resulting nodes by moving your mouse to an appropriate location and then clicking.**

   After clicking the Import button in the previous step, the nodes will appear (grouped together), ready for you to place them by clicking your mouse.

   Move your mouse around until you like the positioning of the imported nodes and then click your mouse.

Try it out

After exporting nodes from a Flow project, create a new Flow project by clicking the blue + button in the lower right-corner. Name the project something, such as "Import Test".

When Flow opens to a new empty project, follow the Import Procedure outlined above. When done, deploy your new flow and then test it out.

After testing is complete, you can delete the Test project. Alternatively, if you want to keep the project we recommend that you set it into offline mode using the button shown in Figure 5.29 (on page 182).

5.3.3.4. Flow Example – HTTP/Debug

The Timestamp/Debug example is a great way to get started with Flow but doesn't exercise the SK2. The HTTP/Debug example takes an input from an HTTP post, which will be sent by the SK2.

This example is broken into four sections:

1. The first part (in Section 5.3.3.4.1) discusses the Flow project.
2. The second part (in Section 0) sends data to Flow using cURL.
3. The third part (in Section 5.3.3.4.3) sends data to Flow using Python.
4. The fourth part (in Section 5.3.3.4.4) sends data to Flow using C/C++.

You only need to complete two sections to run the example. For instance, if you are only interested in programming in Python, you would need to complete #1 and #3.

5.3.3.4.1. Creating the Flow Project (ex_10_flow.json)

Creating the project follows the basic procedure outlined in Section 5.3.3.2. Starting with the Timestamp/Debug project, this one requires adding three additional nodes.

The first two nodes are HTTP Input (HTTP IN) and HTTP Response (HTTP RESPONSE).

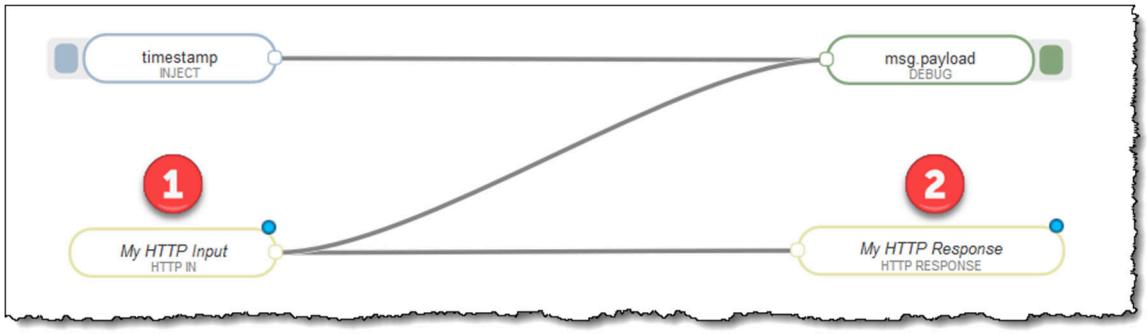


Figure 5.30 – HTTP Input and Response Nodes

When sending an HTTP post to Flow, the HTTP response is not returned to the sender unless you add an HTTP Response node. Depending upon how you code your application posting data using HTTP, it will hang waiting for a response that never arrives. The HTTP RESPONSE node – when configured as shown in Figure 5.30 – will echo back the incoming data to the posting program.

But it might be more fun to do something with the incoming data. By adding a FUNCTION node to the Flow, you can create a Javascript function that will modify the payload being sent to the HTTP RESPONSE node – effectively changing what is being returned to the sender.

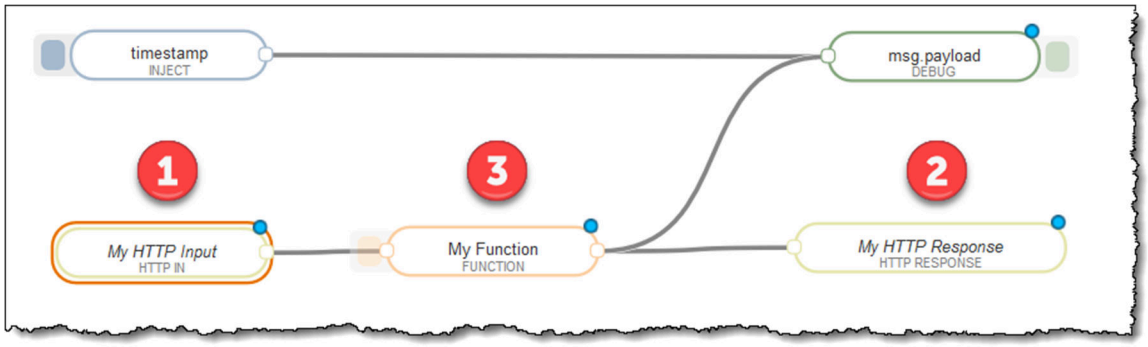


Figure 5.31 – HTTP/Debug Flow Project

Hint: The project described here can be imported using the `ex_10_flow.json` file.

Procedure

Create this Flow project using the following steps.

1. **Re-use or re-create the Timestamp/Debug project from Section Flow Example – Timestamp/Debug (ex_09_timestamp_debug_flow.json)from Section 5.3.3.2.**

   The original project contained two nodes: INJECT, DEBUG.

2. **Add three new nodes to the project.**

   Drag three more nodes onto the project canvas:

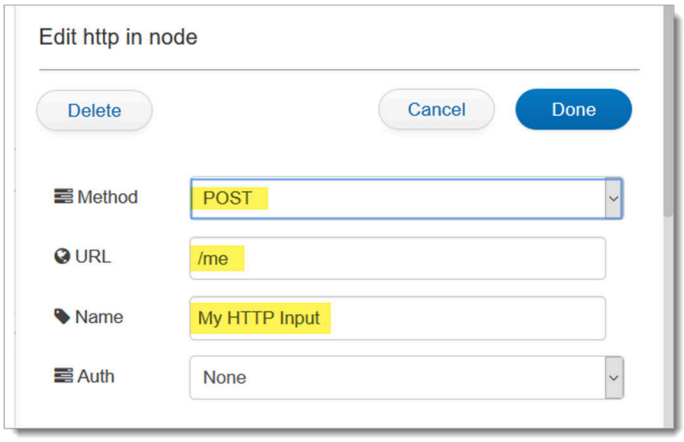
   - HTTP IN
   - HTTP RESPONSE
   - FUNCTION

   Hint: Once again, the search box at the top of the palette makes it easier to find the nodes.

3. **Arrange and connect the nodes as shown in Figure 5.31.**

4. **Configure "HTTP IN" node.**

   Double-click on the HTTP IN node to configure it. Doing so will open a configuration panel:



   Method: Select Post. Any would work for this project, but POST seems appropriate.

   URL: The URL entered here will be appended to your Flow project's Endpoint URL, making the URL for this node unique. This unique endpoint allows us to talk directly to this node.

   Name: Name the node anything you want. This is just displayed in the canvas.

5. **Configure HTTP Response node.**

   The only configuration parameter is its name. (And you don't even have to give it a name.)

6. **Configure the FUNCTION node.**

   The Function node provides a blank canvas, so to speak, for creating a JavaScript function. We'll use this to modify the message payload being sent from HTTP IN to HTTP RESPONSE.

   In our earlier description for this project we stated that we wanted to send a different value, one for the "led" color, in our HTTP RESPONSE – basing the value upon whether the incoming number data was even or odd. The following FUNCTION code implements this:

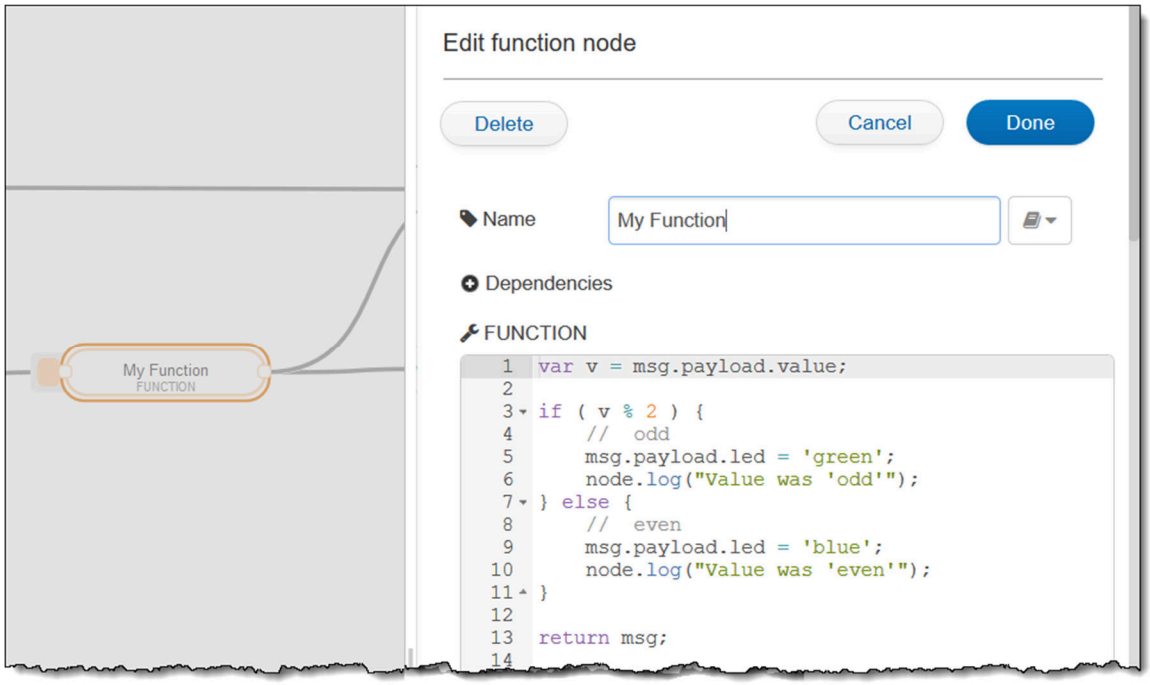
```
var v = msg.payload.value;

if ( v % 2 ) {
    //  odd
    msg.payload.led = 'green';
    node.log("Value was 'odd'");
} else {
    //  even
    msg.payload.led = 'blue';
    node.log("Value was 'even'");
}

return msg;
```

   Most Flow nodes send messages with a payload. If our sending application sends the data "value=56", the HTTP IN adds this to the payload it sends out. The FUNCTION's JavaScript code

picks up this 'value' and then tests whether it's odd/even. Setting an 'led' value to 'green' or 'blue' in the process.

Hint: Setting `msg.payload.led` both adds '`led`' to the message payload as well as sets its value. We suggest that you read more about Node-Red if you'd like to better understand how Flow works.

Here's the dialog we used to configure the FUNCTION node:



7. **Deploy your project.**

   Once the project has been configured, make sure you deploy it.

Hint: If you hadn't noticed before now, the little blue dots in the upper-right corner of nodes, such as in Figure 5.30, indicates that node has been modified since the Flow was deployed.

With the Flow project complete, follow along with one of the next three sections in order to send data to the Flow and see its response.

5.3.3.4.2. cURL Example (ex_10_curl_to_flow.sh)

Sending a value to Flow is straightforward using cURL from a shell script. Here's an example sending the two numbers 25, then 320.

Listing 5.23: ex_10_curl_to_flow.sh

```
    -----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+----9--
 1  echo "ex_10_curl_to_flow.sh"
 2  echo
 3  echo "This shell script sends a number to an AT&T Flow iot project using cURL. "
 4  echo "Flow responds by sending back the number along with a color:"
 5  echo
 6  echo "Green if the number is 'odd'"
 7  echo "Blue if the number is 'even'"
 8
 9  # Replace URL with your Endpoint from AT&T Flow
10  MY_FLOW_URL='https://runm-west.att.io/e9215af2b0eea/8374c4f23615/fa5e6ab1a408526/in/flow/me'
11
12  echo
13  echo
14  echo "Sending an 'odd' number to Flow"
15  echo  --------------------------------
16  curl -X POST $MY_FLOW_URL -d "value=25"
17
18  echo
19  echo
20  echo "Sending an 'even' number to Flow"
21  echo  --------------------------------
22  curl -X POST $MY_FLOW_URL -d "value=320"
23
```

The results from this are shown below.

```
/CUSTAPP/iot_files # ./ex_10_curl_to_flow.sh
ex_10_cur_to_flow.sh

This shell script sends a number to an AT&T Flow iot project using cURL.
Flow responds by sending back the number along with a color:

Green if the number is 'odd'
Blue if the number is 'even'

Sending an 'odd' number to Flow
--------------------------------
{"value":"25","led":"green"}

Sending an 'even' number to Flow
--------------------------------
{"value":"320","led":"blue"}

That's it, we're done!

/CUSTAPP/iot_files #
```

Figure 5.32 – Results from ex_10_curl_to_flow.sh

Let us note that we didn't use the response data from Flow in the shell script to light the LEDs. While eminently possible, we didn't implement this.

5.3.3.4.3. Python Example – Sending Data to Flow

This Python example is like the earlier example: ex_04_http_to_hookbin.py

The modifications from the earlier example include:

1. Changing the URL to point to Flow.
2. Adding in the code (from `leds_blink.py`) to initialize the RGB LED (this code is hidden by the cut-out we used to squeeze the program onto one page).
3. Selecting a random number to send as data.
4. Adding a block to turn on the 'Blue' or 'Green' LED.

Listing 5.24: ex_10_http_to_flow.py

```
15  import iot_mal                                  # Import the Modem Abstraction La
16  import iot_hw                                   # Allows access to SK2 hardware r
17  import requests                                 # A simple HTTP library for Pytho
18  from random import *                            # Module required to generate ran
19
20  # Replace URL with your Endpoint from AT&T Flow
21  MY_FLOW_URL = r'https://runm-west.att.io/e9215af2b0eea/8374c4f23615/fa5e6ab1a408526/in/
22
23  #### Setup the MAL and data connection ##########################################
24  network_handler = iot_mal.network()             # Connect to MAL.network
25  network_handler.set_connection_mode(
26                                      1,          # Mode: 0 - Always, 1 - On-demand
                                       10,          # On-demand Ti        Disconnect i
44  blue_l       iot_hw.gpio_u       _DIR_OUTPU
45  blue_led.write(OFF)
46
47  #### Pick a number (randomly) to send to Flow ###################################
48  value = randint(1, 100)                         # Pick a random number between 1
49  print("\nSending the random number: " + str(value) + "\n")
50
51  #### Use Python 'requests' module to send data over HTTP ########################
52  url = MY_FLOW_URL                               # Get URL from flow.att.io
53
54  headers = {                                     # List contains required HTTP hea
55      #'Content-Type': 'application/json',
56  }
57
58  data = {                                        # Data to be sent
59      "value": value
60  }
61
62  r = requests.post(url, headers=headers, json=data)    # Post data to Flow
63  print("HTTP Response: " + r.text + "\n")              # Print the response from Flow
64
65  #### Turn on the appropriate LED ################################################
66  if 'blue' in r.text:
67      blue_led.write(ON)
68      green_led.write(OFF)
69      print("Setting the RGB LED to 'blue'\n")
70  else:
71      blue_led.write(OFF)
72      green_led.write(ON)
73      print("Setting the RGB LED to 'green'\n")
```

The results from the running the program are shown below. As well, we can confirm that the Blue or Green LED is being lit by the program.

```
/CUSTAPP/iot_files # python ex_10_http_to_flow.py

Sending the random number: 31

HTTP Response: {"value":31,"led":"green"}

Setting the RGB LED to 'green'

/CUSTAPP/iot_files # python ex_10_http_to_flow.py

Sending the random number: 40

HTTP Response: {"value":40,"led":"blue"}

Setting the RGB LED to 'blue'

/CUSTAPP/iot_files #
```

Figure 5.33 –Results from ex_10_http_to_flow.py

5.3.3.4.4. C/C++ Example – Sending Data to Flow

Listing 5.25: c_10_http_to_flow/src/main.cpp

```
142 printf("System Information \n");
143 printf("=============================================================\n");
144 printf("WNC: Module   #    = %s\n", mySystem.model.c_str());
145 printf("WNC: Apps Ver #    = %s\n", mySystem.appsVer.c_str());
146 printf("WNC: Firmware #    = %s\n", mySystem.firmVer.c_str());
147 printf("WNC: MAL Ver. #    = %s\n", mySystem.malwarVer.c_str());
148 printf("WNC: IP Addr. #    = %s\n", mySystem.ip.c_str());
149 printf("SIM: ICCID    #    = %s\n", mySystem.iccid.c_str());
150 printf("SIM: IMEA     #    = %s\n", mySystem.imei.c_str());
151 printf("SIM: IMSI     #    = %s\n\n",mySystem.imsi.c_str());
152
153
154 // Create data to send to Flow
155 value = rand();                                                    // retu
156 printf("Random value = %d\n", value);
157
158 // Create cURL command
159 sprintf(cmd, "curl -X POST %s -d \"value=%d\"", URL, value);
160
161 // Run the cURL comamnd in a subprocess and pipe the response to rbuf
162 printf("\nSending the data to Flow using the following command\ncmd= %s\n", cmd);
163 printf("--------------------\n");
164
165 if ((fp = popen(cmd, "r")) != NULL) {
166     while (fgets(rbuf, BUFSIZ, fp) != NULL)
167
168     printf("--------------------\n");
169     printf("Response = %s\n", rbuf);
170
171     // Turn on correct LED
172     if (strstr(rbuf, "blue")) {
173         printf("\nTurning on Blue LED...\n");
174         LED(GPIO_RED_LED,   OFF, fptr, "Error: red/off");
175         LED(GPIO_GREEN_LED, OFF, fptr, "Error: green/off");
176         LED(GPIO_BLUE_LED,  ON,  fptr, "Error: blue/on");
177     } else if (strstr(rbuf, "green")) {
178         printf("\nTurning on Green LED...\n");
179         LED(GPIO_RED_LED,   OFF, fptr, "Error: red/off");
180         LED(GPIO_GREEN_LED, ON,  fptr, "Error: green/on");
181         LED(GPIO_BLUE_LED,  OFF, fptr, "Error: blue/off");
182     } else {
183         printf("\nError: Turning on Red LED...\n");
184         LED(GPIO_RED_LED,   ON,  fptr, "Error: red/on");
185         LED(GPIO_GREEN_LED, OFF, fptr, "Error: green/off");
186         LED(GPIO_BLUE_LED,  OFF, fptr, "Error: blue/off");
187     }
188 } else {
189     printf("\nError: Turning on Red LED...\n");
190     LED(GPIO_RED_LED,   ON,  fptr, "Error: red/on");
191     LED(GPIO_GREEN_LED, OFF, fptr, "Error: green/off");
192     LED(GPIO_BLUE_LED,  OFF, fptr, "Error: blue/off");
193 }
194
195 // Release Hardware Resources
196 printf("\nReleasing Hardware Resources...\n");
197 UNEXPORT(fptr,GPIO_RED,   "-> Red");
198 UNEXPORT(fptr,GPIO_GREEN, "-> Green");
199 UNEXPORT(fptr,GPIO_BLUE, "-> Blue");
200
201 printf("\nProgram is exiting... \n\n");
202 return 0;
```

```
/CUSTAPP # ./c_10_http_to_flow
Hello World

Allocating Hardware Resources...
  Exported -> Red LED (#38)
  Exported -> Green LED (#21)
  Exported -> Blue LED (#22)

Data service running

System Information
============================================================
WNC: Module   #    = M18Q2FG-1
WNC: Apps Ver #    = OE_v01.07.183121
WNC: Firmware #    = M18Q2_v12.09.182151
WNC: MAL Ver. #    = malm_75_v02.01.1807300
WNC: IP Addr. #    = 10.43.184.167
SIM: ICCID    #    = 89011703278113077369
SIM: IMEA     #    = 014814000341576
SIM: IMSI     #    = 310170811307736

Random value = 2072246913

Sending the data to Flow using the following command
cmd= curl -X POST https://runm-west.att.io/e9215af2b0eea/8374c4f23615/fa5e6ab1a408526/in/flow/me -d "value=2072246913"
--------------------
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    52  100    36  100    16     22      9  0:00:01  0:00:01 --:--:--    32
--------------------
Response = {"value":"2072246913","led":"green"}

Turning on Green LED...

Releasing Hardware Resources...
  Unexported -> Red (#38)
  Unexported -> Green (#21)
  Unexported -> Blue (#22)

Program is exiting...

/CUSTAPP #
```

Figure 5.34 –Results from c_10_http_to_flow/src/main.cpp

5.4. Sending SMS Messages

SMS (short message service) is a popular way to send Alerts and other short pieces of information. It's also an easy way to communicate from your IoT project during development. While the SK2 does not have built-in SMS messaging capabilities, it's easy to get your kit sending messages through a third party service. We highlight two of the most popular – and easiest to work with – in this section.

5.4.1. Twilio

Twilio is one of the leading mobile communications services. After setting up an account with them you can try sending SMS messages right to your phone from your connected SK2.

Twilio is the larger, more professional of the two services we have provided examples for. They've been working in the mobile communications space for a long time. While they are not the only service available today, they make it easy to their services.

5.4.1.1.1. Setting up a Twilio Account

Twilio, as of this writing, still offers free trial accounts that allow you to try out the service. Once you have signed up, you'll need to copy down a few items that will be needed when sending SMS messages.

From the Dashboard you'll need your *Account SID* and *Auth Token*. You can paste these into the shell-script or Python examples that we've provided.

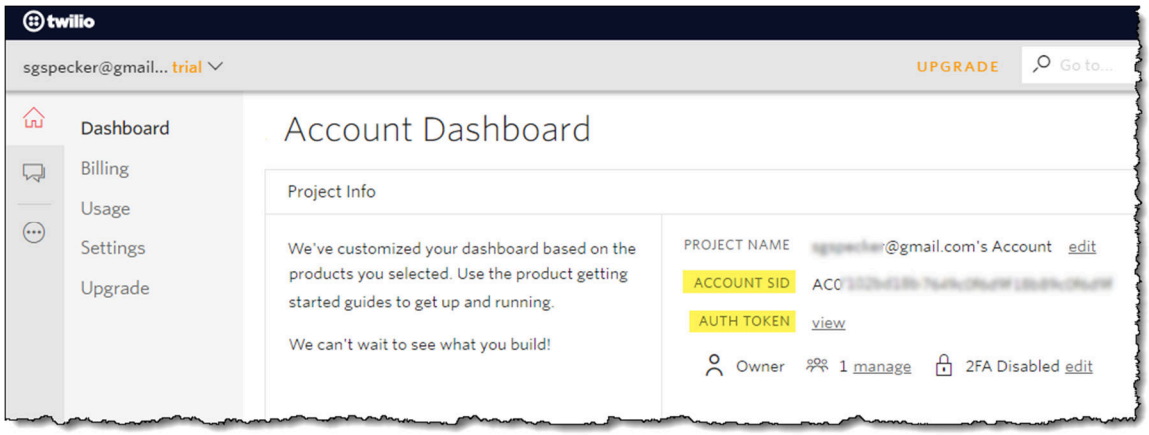


Figure 5.35 –Twilio Dashboard

The third item that you will need is either a Twilio or Verified phone number. You can find out more by visiting their trial account page. *(Note that we used a Twilio number when developing these examples. It was very inexpensive over the two months we needed it.)*

5.4.1.1.2. Flow

AT&T Flow makes it easy to send SMS messages through Twilio, as shown in the following Flow project. This project takes IoT temperature data, coming from M2X (via the HTTP IN node) and processes the data. After processing, the data was posted to HookBin.com and SMS (via Twilio).

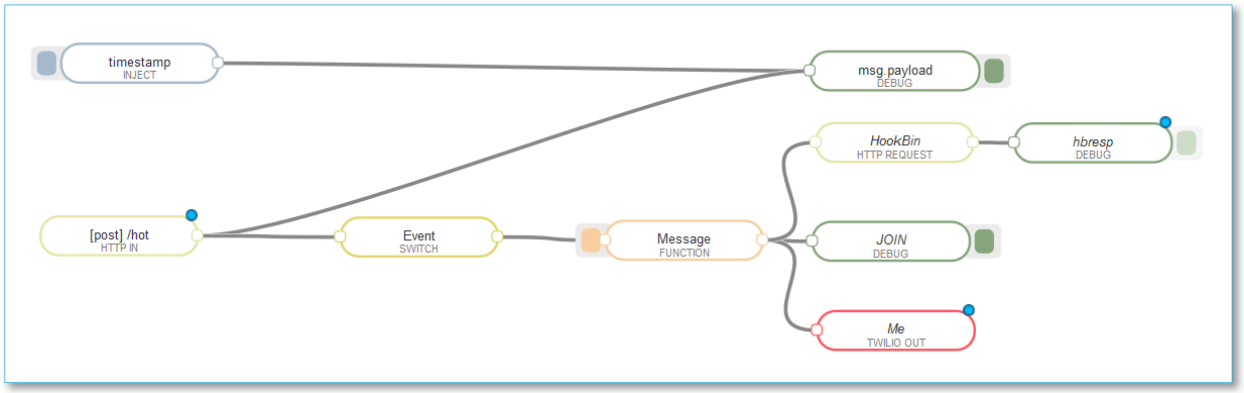


Figure 5.36 –AT&T Flow project with Twilio Node

The red Twilio node can be dragged from the palette and configured using the information from your account. Configuring the Twilio node is a two-step process. Clicking on the *Edit* icon in the dialog on the left takes you to a second dialog where you can enter your Twilio account details.

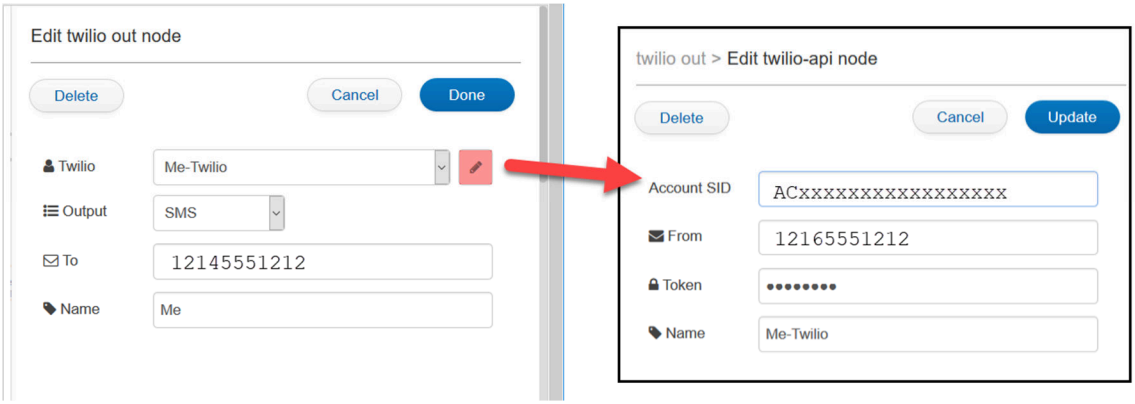


Figure 5.37 –Configuring a Twilio Node in AT&T Flow

5.4.1.1.3. From the SK2 via cURL

The second method uses cURL to send a message to Twilio, who then forwards it to your recipient. Twilio provides an API for development, but their Python SDK has not been ported to the SK2. We couldn't get an HTTP post working either, but thankfully, cURL works well from either Python or a shell-script. (A C/C++ example hasn't been created, but since example c_10_http_to_flow uses cURL, it would be easy to adapt for this purpose.)

The following Python example sends an SMS message using Twilio. You can find the shell-script version of this example in the Chapter_05 directory in the SK2 User Guide Git. The example only requires filling in the four items of information highlighted below.

Listing 5.26: Chapter_05/ex_11_send_twilio_sms.py

```
#############################################################################
# file: ex_11_send_twilio_sms.py
#
# This example sends 'data' to an SMS number using the Twilio.com service.
# Twilio supports sending messages via CURL. Without a native CURL Python
# module on the SK2, the example uses the os.system module to call CURL from
# the Linux command line (i.e. shell out to the command line).
#
# Using GPIO to read the value of the pushbutton was discussed in Chapter 4.
#############################################################################
import iot_mal                                  # Import the Modem Abstraction Layer (MA
import iot_hw                                   # Allows access to SK2 hardware resource
import requests                                 # A simple HTTP library for Python, buil
import os                                       # Needed to call shell commands via 'os

# Replace URL with one you get from your Twilio account
TWILIO_URL  = r'https://api.twilio.com/2010-04-01/Accounts//Messages.json'
TWILIO_SID  = r'ACxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
TWILIO_AUTH = r'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'
SMS_FROM    = r'+12165551212'
SMS_TO      = r'+12145551212'

#### Setup the MAL and data connection ###################################
network_handler = iot_mal.network()             # Connect to MAL.network
network_handler.set_connection_mode(
                                    1,          # Mode: 0 - Always, 1 - On-demand
                                   10,          # On-demand Timeout: Disconnect in mins
                                    2)          # Manual Mode: 0 - Disconnect, 1 - Conne
                                                #              2 - Connect once

#### 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)

if button.read():
    button_state = 'up'
else:
    button_state = 'down'

data = 'Body="button": ' + button_state

curl_url  = " " + TWILIO_URL + "/" + TWILIO_SID + "/Messages.json"
curl_cmd  = " -X POST"
curl_to   = " --data-urlencode 'To=" + SMS_TO + "'"
curl_from = " --data-urlencode 'From=" + SMS_FROM + "'"
curl_data = " --data-urlencode '" + data + "'"
curl_auth = " -u " + TWILIO_SID + ":" + TWILIO_AUTH

curly = 'curl ' + curl_url + curl_cmd + curl_to + curl_from + curl_data + curl_auth
print(curly)

r = os.system(curly)
```

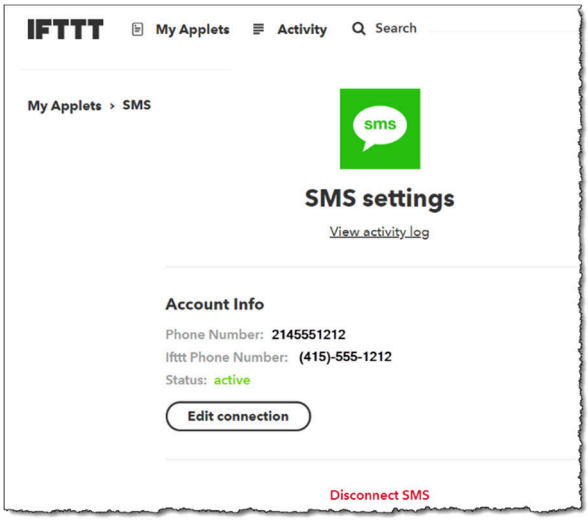
5.4.2. IFTTT

The maker channel of If This Then That (IFTTT) provides another easy way to send yourself SMS messages for free. While not recommended for production systems, this service works fine for maker projects and the like.

Sending SMS messages from your SK2 requires a few steps:

1. **Signing up for a free IFTTT account.**
2. **Adding your SMS (i.e. mobile phone) number to your IFTTT account.**

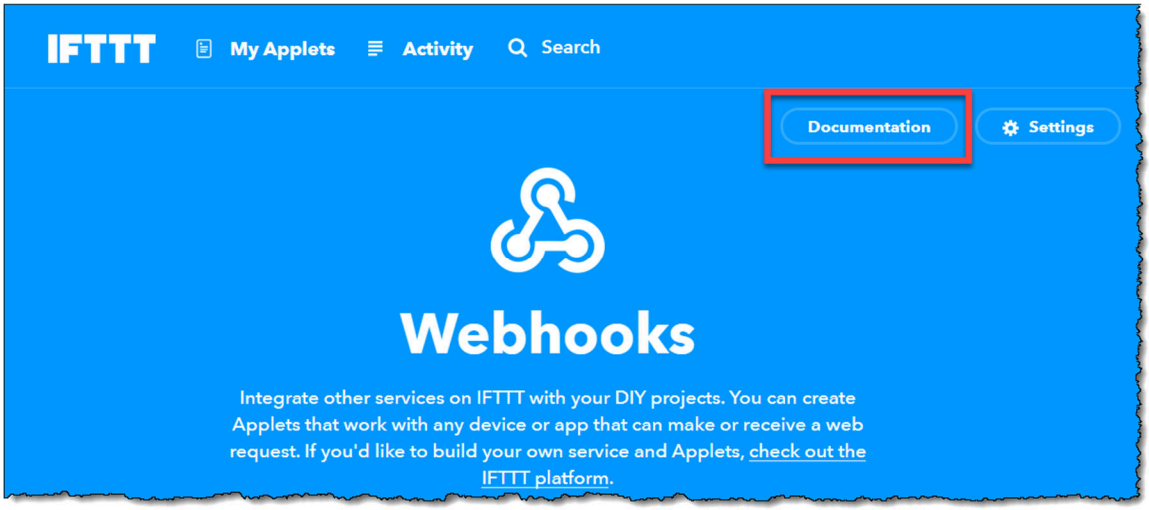
   https://ifttt.com/services/sms/settings



3. **Creating an IFTTT Webhooks applet.**

   https://ifttt.com/maker_webhooks

   Using the maker webhooks you can generate an IFTTT event. These events can be used to trigger any applet, though we just used it to trigger an SMS message.



   Clicking on the Documentation link takes you to a screen that explains how the service works.

Your IFTTT account will generate a unique key, such as that shown below (phi0VVW9_4847GHBSSi). Using this key, you can create custom URLs to signify different maker events. Try filling in the "event" field on the documentation screen, then hitting "Test It".

Page left intentionally blank

6. I2C Communication

I2C (or Inter-Integrated Circuit) is a highly efficient protocol for communicating between one or more devices within an embedded system. Used mainly used for connecting low speed peripherals to a processor, it's become one of the most popular protocols since it conveniently connects multiple environmental sensors – for example, those that measure position, temperature, humidity, light, etc. – to the system processor or MCU.

The AT&T IoT Starter Kit (2nd Generation) – that is, the SK2 – uses I2C to connect the accelerometer/temperature sensor (STM LS2DW12) to the WNC M18Q2FG-1 processor/modem module. Using this protocol, your programs can read the sensor values. Additionally, the I2C pins are brought out to both board's PMOD and Expansion ports.

Topics

6.1. Intro to Serial Communication Ports

Serial communication protocols are popular in embedded systems as they allow the processor to talk to a variety of external devices without requiring too many external pins to implement the port. Typically, serial ports send data "serially" (one value after another) over a single data line – which makes them very efficient (from a pin/package size standpoint).

The SK2 contains three types of serial communication ports. For context, we will briefly describe all three protocols, but only the I2C port is covered in this chapter.

UART: The *Universal Asynchronous Receiver Transmitter* (UART) transmits data over two data pins, one for receive, the other for transmit. The UART is intended for point-to-point (i.e. 1-to-1) communications. The word "asynchronous" in the name means that this port does not transmit data synchronous to a clock, therefore no explicit clock pin is required. The SK2 contains two UART ports, but as of this writing, they are used as Linux debug ports and are not available to users.

SPI: The *Serial Peripheral Interface* (SPI) is a synchronous serial port. By synchronous, we mean that the data is transmitted over the transmit and receive pins in alignment with the port's clock pin. Like the UART, the SPI port communicates point-to-point between two devices. To enhance its usefulness, the SPI port often includes chip select or enable pins which allow some flexibility for talking to multiple devices.

I2C: The *Inter-Integrated Circuit* (I2C) communications port is another synchronous (i.e. clocked) serial port. Using only uses two pins (Clock, Data) this port can communicate between many different devices. This is accomplished by sending out an address before data is read or written. I2C devices in the system use "address" to determine if they are the read/write target for each data transfer. (Unlike SPI, where the data pins are unidirectional, the I2C data pin is bidirectional.)

Note: This chapter only addresses the I2C protocol.

6.2. How I2C Works

From a technical perspective, communication via I2C is complex. For valid communication, the devices on the bus must adhere to a specific protocol. Thankfully, with the advent of hardware I2C ports, which automatically handle the interface protocol details, we only need to concentrate on the data to be sent.

Those implementing I2C connections on their board generally need to know where to find the I2C pins and deal with their open-drain outputs – which is discussed in Section 6.2.2

On the software side, we need to learn two things:

1. How to use the Peripheral API to talk over the I2C bus.

   You will need to understand the I2C Peripheral API functions required for interacting with the I2C bus master controller in the WNC processor module.

2. Details about the device you are communicated with (e.g. the LIS2DW12 sensor).

   Most of your time spent working with I2C ports, though, will be spent on #2, learning about the devices you will talk to – the device I2C address, device configuration, and any specific requirements for sending and receiving data with the device.

6.2.1. Definitions

Before getting into the details of the I2C port, let us define a few terms used throughout this chapter.

Bus

The term “bus” refers to the communication pathway between two (or more) devices in a system. As discussed in the next section, the I2C bus consists of two signals (SCL - clock, SDA - data).

Synchronous

The I2C port sends data synchronous (i.e. in rhythm) to a clock – which is why the I2C bus consists of two signals – one for the clock and the other for data. When data is being transmitted, 1-bit of information is sent across the data line for each cycle on the clock line.

Master/Slave

The I2C bus is a Master/Slave protocol. That is, only one device in the system controls the data flowing across the I2C bus; it is called the bus master. The remaining devices attached to the I2C bus are all listeners (slave devices) which only communicate when directed to by the master controller.

The WNC module on the SK2 only operates as the I2C bus master (and not as a slave).

Address

The I2C bus master uses a 7-bit address to indicate which listener device is to be the target of the read or write operation.

Data

The SK2 I2C port sends 8-bits of data. If a sensor happens to collect 16- or 32-bit data values, then multiple operations are required to move all the data.

Message

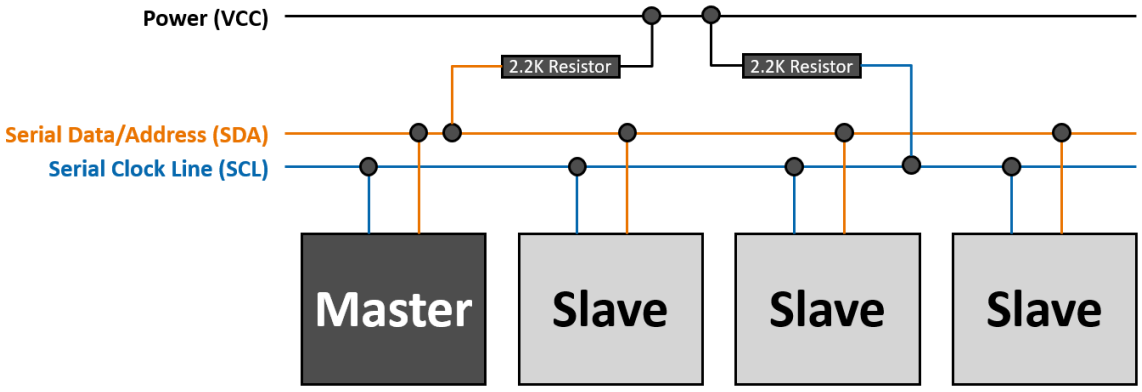
The I2C bus communicates using messages. Each message consists of an *address* frame, a read/write (R/W bit), as well as one or more *data* frames.

6.2.2. Physical Connections

6.2.2.1. I2C Signal Lines

The I2C serial bus consists of two signal lines – SDA and SCL – which are used to communicate between the I2C devices.

- **Serial CLock (SCL):** The clock signal line is always driven by the 'master device'. Pulses on this line correspond synchronously with data (or address) information on the SDA line.
- **Serial DAta (SDA):** This signal line transmits the I2C slave address as well as the data. It is driven by the master when I2C slave address is being transmitted. When transmitting data, either the master or the I2C peripherals drive the line, depending upon who is writing data. (Note that some I2C implementations call this the 'serial data line' and use the SDL abbreviation.)



6.2.2.2. Open-Drain Outputs

How can so many devices be connected to the same pins without causing a conflict? The I/O drivers for both lines (SCL and SDA) are set to an open-drain state when transfers are occurring.

An open-drain output pin is driven by only a single transistor – which drives the pin to only one voltage (generally, to ground). When the output device is off, the pin is left floating (open, or hi-z). Floating pins do not collide with each other, which prevents conflicts between them.

Since there isn't a transistor to pull the pin high, which is needed when sending information across the bus, the SDA and SCL lines must be pulled-up; that is, they must be tied to the power line (VCC) through resistors. (The SK2 uses 2.2K ohm resistors to 'pull-up' the I2C bus lines.)

Note: Open-drain refers to such a circuit implemented in FET technologies because the transistor's drain terminal is connected to the output.

6.2.2.3. I2C Data Transfer Rates

The I2C standard allows for three modes: standard, fast, and high-speed mode. These modes indicate the clock rate (i.e. bus speed) of the I2C port, ranging from 100kHz to 3.4MHz.

The WNC processor module is always the master of its I2C port and drives the clock line (SCL) at the 400 kHz rate.

6.2.3. Signaling Protocol

6.2.3.1. Generic I2C Message

The I2C protocol defines a message in the following way:

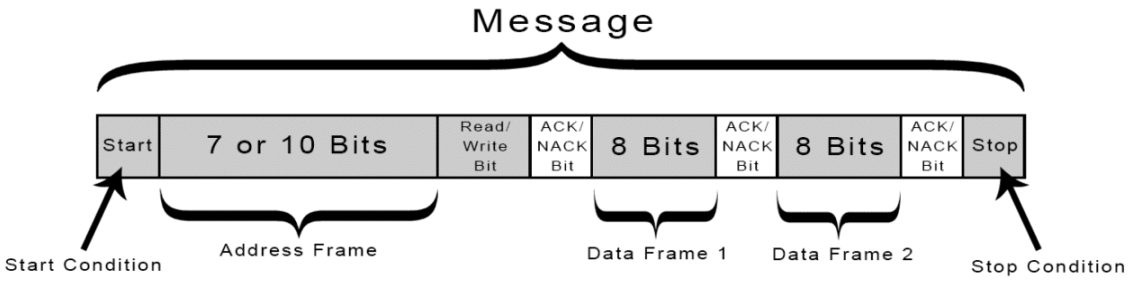


Diagram from: http://www.circuitbasics.com/basics-of-the-i2c-communication-protocol

6.2.3.1.1. Start/Stop Conditions

The I2C bus master indicates a new message by dropping the SDA line before the SCL line. These conditions let the slave devices know when they should be examining the address for a new message.

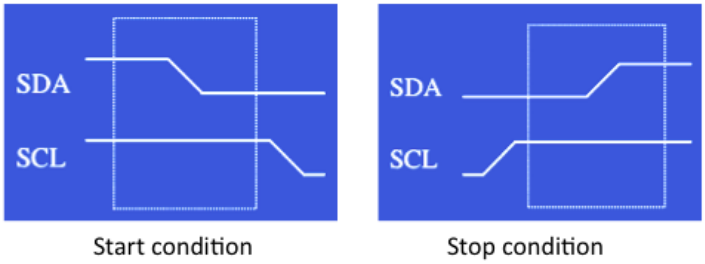


Diagram from: https://microcontrollerslab.com/i2c-bus-communication-protocol-tutorial-applications

6.2.3.1.2. Address Frame + Read/Write Bit + ACK

The address frame tells the target slave device that 'this' message is for them. The SK2 only supports 7-bit addressing. (The I2C standard also allows for 10-bit addressing, but this is not supported.)

Listener (slave) devices will read the 7-bit address (most significant bit first) to determine if they are the target of the message – and then the R/W bit tells it whether to expect a read or write operation.

After the transmission of each frame, the transmitting device will wait for an acknowledgement (ACK) before continuing with the next frame.

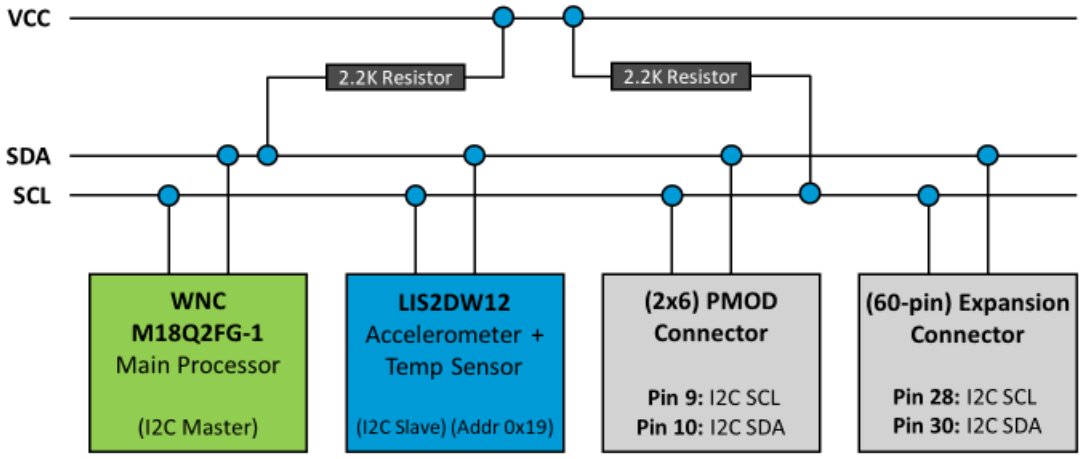
The 9-bits of data are sent for each transfer, wherein 8-bits are sent by the transmitter (MSB to LSB), and the 9th bit is an acknowledgement bit sent by the receiver.

6.2.3.1.3. Data Frame(s) + ACK

Data transfers are implemented in 8-bit frames. If a read was selected, the slave device transmits the data and waits for an ACK from the master device. If a write is being performed, the master sends the data and waits for the ACK from the slave device.

6.3. I2C Implementation on the SK2

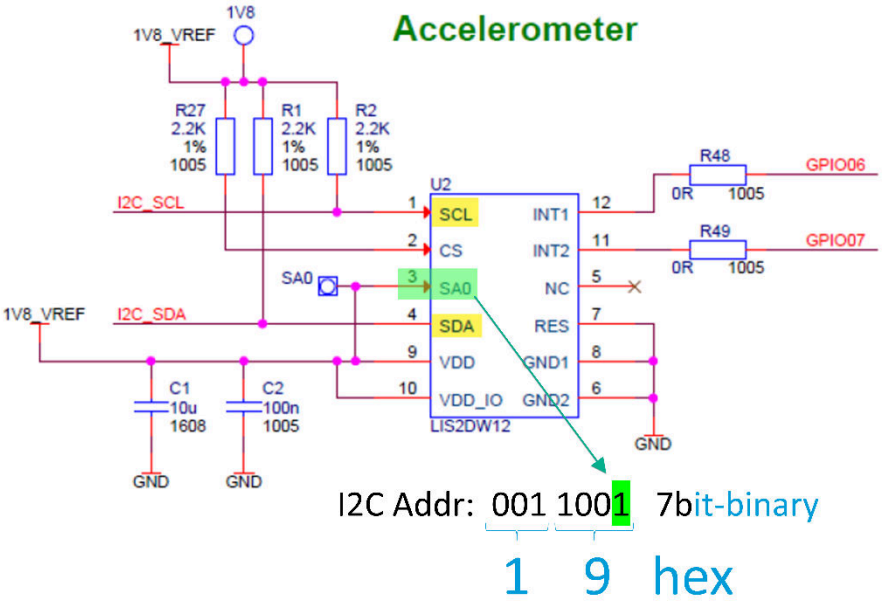
6.3.1. SK2 Hardware



6.3.1.1. LIS2DW12 Sensor

6.3.1.1.1. Connecting to the LIS2DW12 Sensor

From a hardware perspective, we need to understand how to connect to the LIS2DW12 sensor. We can do this by reading the SK2 hardware user's guide. Here is a snippet from the schematic:



This graphic shows the I2C Address as: 0011001 7b, which tells us the address value for the LIS2DW12 is a 7-bit binary number. Translating this to hex format, we find the I2C address for our sensor is `0x19`.

But, where did this address come from? To understand that, we need to read through the LIS2DW12 datasheet.

> The Slave Address (SAD) associated to the LIS2DW12 is 001100xb where the x bit is modified by the SA0/SDO pin in order to modify the device address. If the SA0/SDO pin is connected to the supply voltage, the address is 0011001b, otherwise if the SA0/SDO pin is connected to ground, the address is 0011000b. This solution permits to connect and address two different accelerometers to the same I²C lines.

The address is found on page 28 (for datasheet revision 6 on Nov 2018). The paragraph on slave addressing indicates the address is `001100x`, where `x` is determined by the value applied to the `SA0` pin. While the I2C address could be `0x18` or `0x19`, the SK2 schematic shows that the `SA0` pin is tied to the power line, which means we address this device using `0x19`.

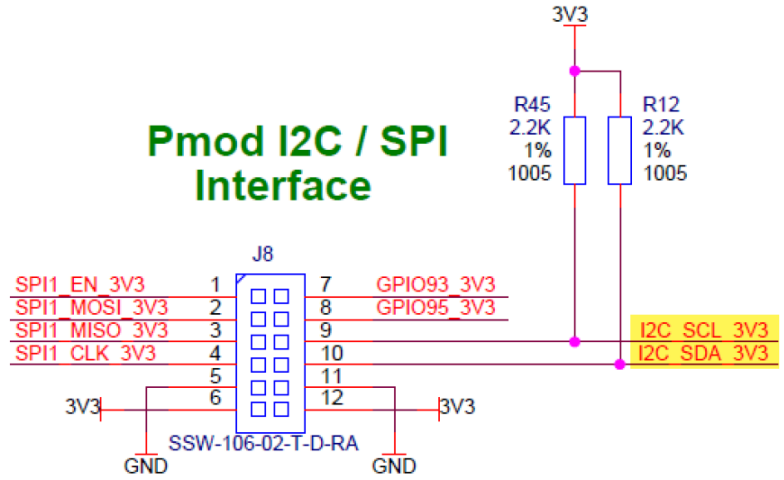
Therefore, you should use the `0x19` address when reading or writing the LIS2DW12 sensor. Here's a snippet from a Python example.

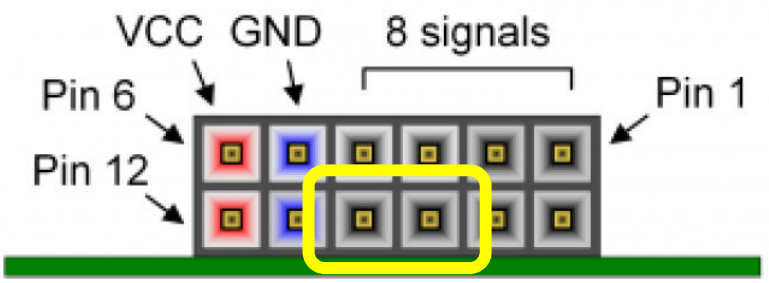
```
slave_address = int("0x19", 16); #0b0011001

i2c_bus.write(slave_address, tx_buffer, 2, iot_hw.i2c_flag.I2C_STOP);
```

6.3.1.2. PMOD Connector

As mentioned in Chapter 1, the PMOD connector allows you to attach additional sensors and hardware to the SK2. PMOD modules that contain sensors (or other hardware) using GPIO, I2C or SPI interfaces are supported.

The SK2 hardware user's guide provides a description for the PMOD interface. Using this, we can find the I2C pins within the PMOD connector. Notice that they're on pins 9 and 10 within the connector. You shouldn't need to know that when connecting a commercial PMOD sensor, but it's required if you decide to build your own board to attach to the PMOD connector.





Hint: Knowing the I2C pins on the PMOD connector is also useful when attaching a logic analyzer to your SK2. Since all I2C devices on the SK2 share a single bus, you can use the logic analyzer to view (and debug) your I2C code as you communicate with the LIS2DW12 sensor. Please refer to Section 6.6 for more information about connecting up a logic analyzer.

6.3.1.3. 60-pin Expansion Connector

The 60-pin expansion connector found on the underside of the SK2 board provides yet another way to access the I2C pins.

Table 4 – J1 Expansion Connector Pinout (Outer Row)

J1 pin	Signal Name	Description	WNC pin	PU /PD	WNC Modem Mode	Ho
2	REG_EN	3V8 Regulator Enable	-	PD	-	
4	ADC_H	ADC ext. input (to JP2)	122	PD	ADC (via JP2)	ADC
6	UART1_CTS	CTS for WNC UART1	80	PD	UART1_CTS	UART
8	UART1_RTS	RTS for WNC UART1	81	PD	UART1_RTS	UART
10	GPIO02	GPIO02	53	PD	GPIO02	GPIO
12	GPIO03	GPIO03	54	PD	GPIO03	GPIO
14	GPIO04	GPIO04	55	PD	GPIO04	GPIO
16	GND	GND	-		-	
18	GPIO95	GPIO95	95	PD	GPIO95	GPIO
20	GPIO94	GPIO94	94	PD	GPIO94	GPIO
22	EPHY_RST_N	EPHY_RST_N	103	PD	-	EPHY
24	GPIO96	GPIO96	96	PD	GPIO96	GPIO
26	GPIO97	GPIO97	97	PD	GPIO97	GPIO
28	I2C_SCL	I2C_SCL	61	PD	I2C_SCL	I2C_S
30	I2C_SDA	I2C_SDA	60	PD	I2C_SDA	I2C_S
32	GPIO93	GPIO93 (output only)	93	NP	GPIO93	GPIO
34	GND	GND	-		-	

6.3.2. Application Programming Interface (API)

Both SK2 software development kits (Python and C/C++) each provide an API for connecting to I2C slave devices. The software interface is rather simple, providing four functions:

Description	Python	C/C++
Initialize the I2C bus	i2c()	i2c_bus_init()
Read	.read()	i2c_read()
Write	.write()	i2c_write()
Release the I2C bus	.close()	i2c_bus_deinit()

The examples provided below demonstrate how to use these API functions.

6.4. Using the LIS2DW12 Sensor via I2C

6.4.1. Description of LIS2DW12

The LIS2DW12 sensor from ST Microelectronics (STM) provides both accelerometer motion data, as well as temperature readings.

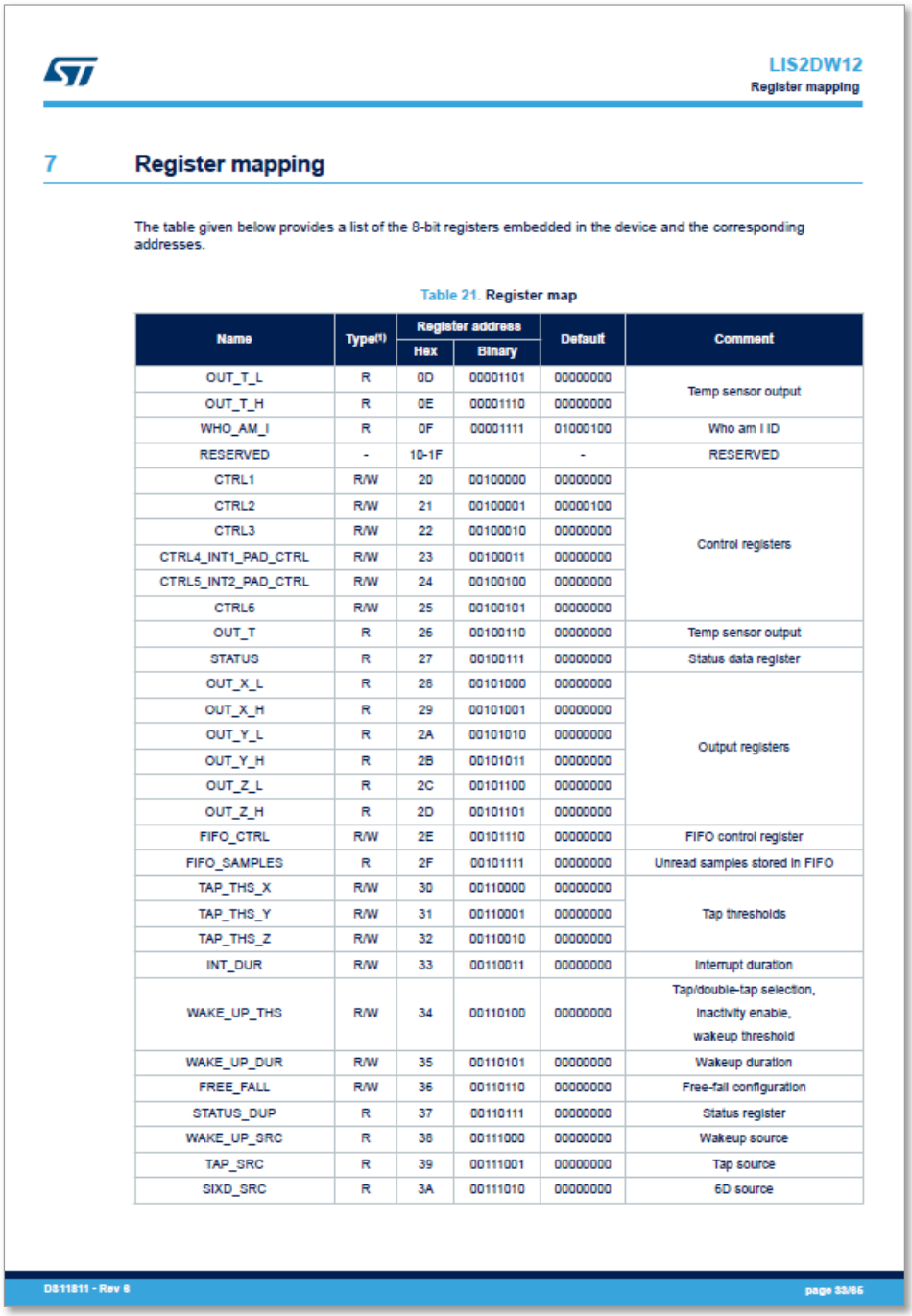
The LIS2DW12 can be used as an I2C slave device. Section 6.3.1.1.1 (above) discussed how the sensor is physically connected to the LIS2DW12; including, how to determine the sensor's slave I2C address.

This section (6.4) discusses how to write software to read and control the LIS2DW12 while providing examples to demonstrate working code.

6.4.2. How To Talk to the LIS2DW12

6.4.2.1. Sensor Device Registers

While some I2C slave devices can be directly read-from or written-to, “talking” to the LIS2DW12 is accomplished by reading and writing to registers within the sensor. The STM datasheet for this device provides a listing of registers found in the LIS2DW12 sensor.



LIS2DW12
Register mapping

7 Register mapping

The table given below provides a list of the 8-bit registers embedded in the device and the corresponding addresses.

Table 21. Register map

Name	Type(1)	Register address		Default	Comment
		Hex	Binary		
OUT_T_L	R	0D	00001101	00000000	Temp sensor output
OUT_T_H	R	0E	00001110	00000000	
WHO_AM_I	R	0F	00001111	01000100	Who am I ID
RESERVED	-	10-1F		-	RESERVED
CTRL1	R/W	20	00100000	00000000	Control registers
CTRL2	R/W	21	00100001	00000100	
CTRL3	R/W	22	00100010	00000000	
CTRL4_INT1_PAD_CTRL	R/W	23	00100011	00000000	
CTRL5_INT2_PAD_CTRL	R/W	24	00100100	00000000	
CTRL6	R/W	25	00100101	00000000	
OUT_T	R	26	00100110	00000000	Temp sensor output
STATUS	R	27	00100111	00000000	Status data register
OUT_X_L	R	28	00101000	00000000	Output registers
OUT_X_H	R	29	00101001	00000000	
OUT_Y_L	R	2A	00101010	00000000	
OUT_Y_H	R	2B	00101011	00000000	
OUT_Z_L	R	2C	00101100	00000000	
OUT_Z_H	R	2D	00101101	00000000	
FIFO_CTRL	R/W	2E	00101110	00000000	FIFO control register
FIFO_SAMPLES	R	2F	00101111	00000000	Unread samples stored in FIFO
TAP_THS_X	R/W	30	00110000	00000000	Tap thresholds
TAP_THS_Y	R/W	31	00110001	00000000	
TAP_THS_Z	R/W	32	00110010	00000000	
INT_DUR	R/W	33	00110011	00000000	Interrupt duration
WAKE_UP_THS	R/W	34	00110100	00000000	Tap/double-tap selection, Inactivity enable, wakeup threshold
WAKE_UP_DUR	R/W	35	00110101	00000000	Wakeup duration
FREE_FALL	R/W	36	00110110	00000000	Free-fall configuration
STATUS_DUP	R	37	00110111	00000000	Status register
WAKE_UP_SRC	R	38	00111000	00000000	Wakeup source
TAP_SRC	R	39	00111001	00000000	Tap source
SIXD_SRC	R	3A	00111010	00000000	6D source

DS11811 - Rev 8 page 33/65

When the I2C device has many registers, your code will need to:

1. Send out a message including the “Slave Address” to select the correct I2C slave device, followed by the register to be read/written.
2. Send out second message that contains, once again, the correct “Device Address” for the I2C slave device, followed by reading (or writing) the actual data values.

Here is an example of what the sequence will look like for an I2C “read” operation:

Figure 1 - Example I2C sequence

6.4.2.2. Repeated Start Bits

Looking back at Figure 1, notice that there are two ”Start” bits. One begins the frame writing the “Register Address”, the other begins the frame reading **“Data”.** This is an example of repeated start bits. Various I2C devices respond differently to repeated start bits. Some always require them, others don’t allow them; always check the devices datasheet to understand which is required.

The LIS2DW12 sensor requires repeated start bits for reads but does not allow them for writes.

6.4.2.2.1. Data Reads

Checking the sensor’s datasheet, we find that from its ‘Table 19’, I2C data reads require repeated start bits, as highlighted below:

Table 19. Transfer when master is receiving (reading) one byte of data from slave

Master	ST	SAD + W		SUB		SR	SAD + R			NMAK	SP
Slave			SAK		SAK			SAK	DATA		

Using the SK2 Python Peripheral API, we naturally get two start bits when reading a value since each API function generates a new start bit. Here’s a code snippet for reading the sensor’s device ID:

```
# Send a request to WHO_AM_I register
myi2c.write(DEV, [LIS2DW12.WHO_AM_I], 1, iot_hw.i2c_flag.I2C_NO_STOP)

# Read data from the WHO_AM_I register
myi2c.read(DEV, lis2dw12_id, 1)
```

As mentioned above, we end up with two start bits because each API call generates a new start bit.

6.4.2.2.2. Data Writes

Conversely, the datasheet's 'Table 17' (shown below) indicates that repeat start bits are not allowed when writing data to the device.

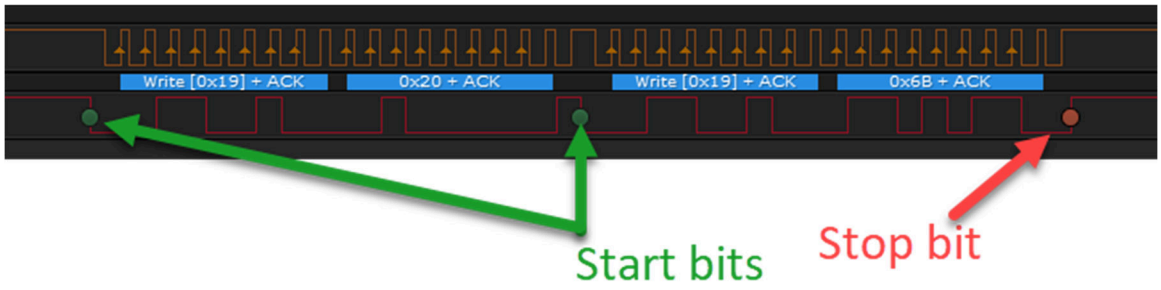
Table 17. **Transfer when master is writing one byte to slave**

Master	ST	SAD + W		SUB		DATA		SP
Slave			SAK		SAK		SAK	

Using two API calls to write data to the device – like we did for reads – fails because this generates two start bits, which are not allowed. Therefore, the following code will not work.

```
# This code fails
# Send a request to the CTRL1 register
myi2c.write(DEV, [LIS2DW12.CTRL1], 1, iot_hw.i2c_flag.I2C_NO_STOP)

# Write data from the CRTL1 register
myi2c.write(DEV, 0x6B, 1, iot_hw.i2c_flag.I2C_NO_STOP)
```

Examining the two messages generated by the preceding code -- using a logic analyzer to trace the I2C bus -- we see repeated start bits:



To execute the write correctly, we recode using a single API write() to transfer both bytes in one call.

```
# This code works!
# Write to the CTRL1 register
myi2c.write(DEV, [LIS2DW12.CTRL1, 0x6B], 2, iot_hw.i2c_flag.I2C_NO_STOP)
```

Note: The read/write code shown here are similar to the examples for reading the temperature from the LIS2DW12 described below.

6.5. Python Examples

The following examples demonstrate how to read data values over the I2C bus from the appropriate registers inside the LIS2DW12 sensor.

6.5.1. Reading the Device ID (i2c_hw_example.py)

Reading the LIS2DW12 sensor's WHO_AM_I register that returns its device ID, which is always 0x44. This is a great way to be assured that you are talking to the LIS2DW12 sensor.

Listing 6.1: i2c_hw_example.py

```
    """
    i2c_hw_example.py

    Taken from the iot_hw_example.py code example, this example toggles two GPIO
    pins before reading from the "WHO_AM_I" register of the LIS2DW12 sensor found
    on the SK2.

    - GPIO flash blue LED and GPIO 95 (found on the PMOD connector)
    - I2C Read of 'WHO_AM_I' register from accelerometer chip

    """
    import iot_hw
    import time

    def test_i2c():
        print('Test I2C')
        dev = 0x19                  # i2c slave address of LIS2DW12 accelerometer
        addr = [0x0f]               # WHO_AM_I register
        data = list()               # Create data array to receive results from "WHO

        myi2c = iot_hw.i2c()        # Initialize (i.e. open) the i2c bus protocol
        myi2c.write(dev, addr, 1, iot_hw.i2c_flag.I2C_NO_STOP)   # Write request to W
        myi2c.read(dev, data, 1)    # Read data from the WHO_AM_I register
        print('  I2C reg {0:#2x} value is: {1:#2x}'.format(addr[0], data[0]))

    def test_leds():
        print('Test LEDs')
        leds = {"Blue":iot_hw.gpio_pin.GPIO_LED_BLUE, "Pmod95":iot_hw.gpio_pin.GPIO_P

        for key in leds:
            myled = iot_hw.gpio(leds[key])
            myled.set_dir(iot_hw.gpio_direction.GPIO_DIR_OUTPUT)
            print('  {} on'.format(key))
            myled.write(iot_hw.gpio_level.GPIO_LEVEL_HIGH)
            time.sleep(1)
            print('  {} off'.format(key))
            myled.write(iot_hw.gpio_level.GPIO_LEVEL_LOW)
            time.sleep(1)
            myled.close()

    def main():
        test_leds()
        test_i2c()

```

Writing this Python program, we started with the SDK's hardware example (`iot_hw_example.py`), removing everything except the I2C and LED portions of the code. Further, we replaced toggling the

green and red LEDs with toggling a GPIO pin on the PMOD connector – which we used to trigger our logic analyzer (shown in Section 6.5.1.2).

6.5.1.1. Examining the I2C code

Let's examine the I2C portion of the code from `i2c_hw_example.py`.

1. We calculated the I2C device address of the LIS2DW12 on the SK2 in Section 6.3.1.1.1.

```
dev = 0x19                    # i2c slave address of LIS2DW12
```

2. The register address of the LIS2DW12 sensor's WHO_AM_I register can be found in the datasheet. This was shown in Section 6.4.2.

```
addr = [0x0f]                 # WHO_AM_I register
```

3. A Python data array was created to receive the results from reading the WHO_AM_I register.

```
data = list()                 # Create data array to receive results
```

4. Using the API, we must initialize the I2C bus before using it.

```
myi2c = iot_hw.i2c()          # Initialize (i.e. open) the i2c bus
```

5. Write the address of the I2C slave device that we want to talk to – in this case, 0x19.

```
myi2c.write(dev, addr, 1, iot_hw.i2c_flag.I2C_NO_STOP)
```

6. Read from the LIS2DW12 sensor's WHO_AM_I register.

   It knows which register to read from based on selecting the register with the previous step's write.

```
myi2c.read(dev, data, 1)      # Read data from the WHO_AM_I register
```

7. Printing out the results:

```
print('I2C reg {0:#2x} value is: {1:#2x}'.format(addr[0], data[0]))
```

   gives us:

```
/CUSTAPP/iot_files # python i2c_hw_example.py
Test LEDs
  Blue on
  Blue off
  Pmod95 on
  Pmod95 off
Test I2C
  I2C reg 0xf value is: 0x44
/CUSTAPP/iot files #
```

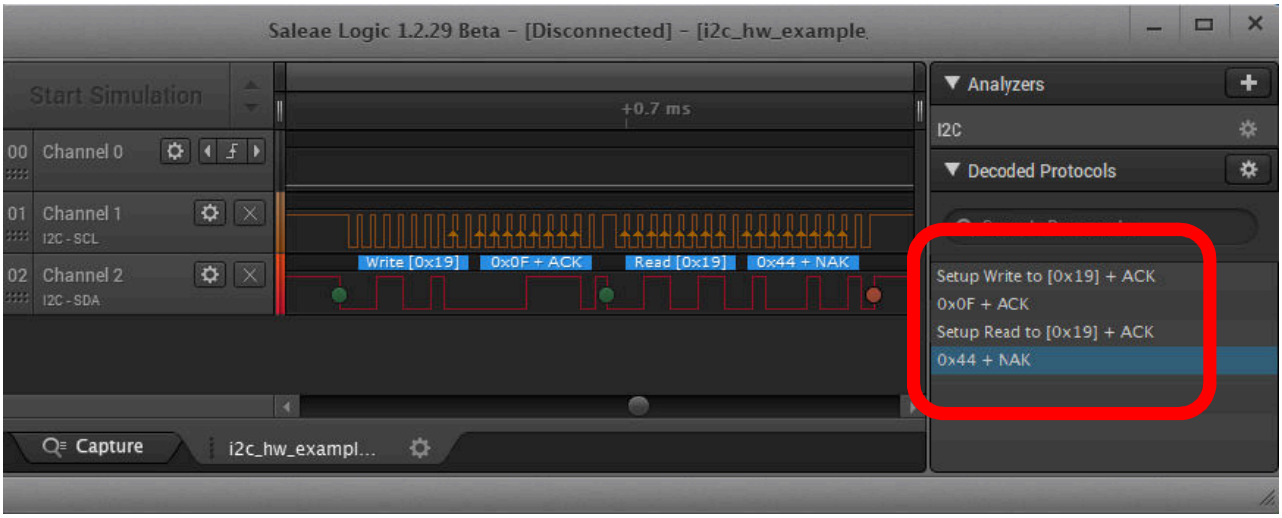
Looking back to page in the datasheet listing all of the sensor's internal registers (shown in Section 6.3.1.1.1), we see that the WHO_AM_I register returns a value 0x44.

Name	Type(1)	Register address		Default	Comment
		Hex	Binary		
OUT_T_L	R	0D	00001101	00000000	
OUT_T_H	R	0E	00001110	00000000	
WHO_AM_I	R	0F	00001111	01000100	Who am I ID

equals 0x44

6.5.1.2. Logic Analyzer Results

Here's a look at the I2C bus transactions for our `i2c_hw_example.py` using a logic analyzer. The analyzer makes it easy for us to see what's happening by decoding the activity on the SCL and SDA pins. In this scope, it's easy to follow that our SK2 code wrote out 0xF to I2C bus address of slave device0x19, then read 0x44 from device address 0x19.



Note: We used the Saleae USB Logic Analyzer for tracing the I2C bus.

6.5.2. Reading the Temperature

Reading the temperature from the LIS2DW12 sensor involves a bit more effort than reading the device ID from the WHO_AM_I register. This process requires configuring control registers and polling for the temperature to become ready.

Here are the basic steps for reading the temperature from the LIS2DW12 sensor:

1. Configuring the CTRL1 register
2. Triggering temperature capture in the CTRL3 register
3. Polling the DRDY_T bit from the STATUS_DUP register
4. Reading the temperature from the 8-bit OUT_T register – or reading the 12-bit temperature resolution from both the OUT_T_H and OUT_T_L registers.

6.5.2.1. Configuring Temperature Read Examples

The configuration code for reading 8-bit or 12-bit temperature values is the same. We'll walk through steps 1-3 (of the 4 we outlined) in this section, after which we'll examine reading both the 8-bit and 12-bit resolutions independently.

6.5.2.1.1. LIS2DW12 CTRL1 Register

There is more than one way to configure the control registers and still be able to read the temperature. The settings you choose will depend upon your systems low-power requirements and whether you plan to read one – or many – values.

Looking at the datasheet, we find the description of CTRL1:

Table 27. **Control register 1**

ODR3	ODR2	ODR1	ODR0	MODE1	MODE0	LP_MODE1	LP_MODE0

In the two code examples provided below, CTRL1 was set to: 0x6B.

- **ODR**[3:1] sets the data rate. We copied Avnet's *iot_monitor* example and set this field for 200Hz.
- **MODE**[1:0} sets the mode selection. We chose the single, on-demand data conversion mode.
- **LP_MODE**[1:0] lets you choose from Low Power mode 1 to 4. This mode defines one of four settings for the sensor's turn-on time. (Again, we copied the setting that Avnet used.)

Here's the snippet of code used to set CTRL1 to 0x6B. (This was the same code shown earlier in Section 6.4.2.2.)

```
51      # Write configuration Control Register 1 (CTRL1)
52      myi2c.write(DEV, [LIS2DW12.CTRL1, 0x6B], 2, iot_hw.i2c_flag.I2C_NO_STOP)
53      print('  -> CTRL1 written - 200Hz, single conversion, LPM4')
```

6.5.2.1.2. LIS2DW12 CTRL3 Register

There are two ways to trigger sensing the temperature. These are controlled by the **SLP_Mode** bits.

Our example sets CRTL3 = 0x3.

Table 33. **Control register 3**

ST2	ST1	PP_OD	LIR	H_LACTIVE	0	SLP_ MODE_SEL	SLP_ MODE_1

CTRL3.SLP_MODE_SEL allows you to trigger sensing based on either (0) external trigger on INT2 pin; or (1) writing to **CTRL3.SLP_MODE_1** bit.

Our example writes 0x3 which configures the sensor for triggering by writing to the **SLP_MODE_1** bit – along with writing to the trigger bit.

Writing to CTRL3 can be accomplished in a similar fashion to configuring CTRL1 in the previous section.

```
55      # Trigger temperature sampling/converstion via CTRL3 register
56      myi2c.write(DEV, [LIS2DW12.CTRL3, 0x03], 2, iot_hw.i2c_flag.I2C_NO_STOP)
57      print('  -> CTRL3 written - Trigger temperature sampling')
```

6.5.2.1.3. LIS2DW12 STATUS_DUP Register

It takes time for the LIS2DW12 sensor to measure the temperature and place the value into the temperature output register(s). We can poll (i.e. check) the DRDY_T bit in the STATUS_DUP register to determine when the temperature output is available.

Table 75. STATUS_DUP register

OVR	DRDY_T	SLEEP_ STATE_IA	DOUBLE_ TAP	SINGLE_ TAP	6D_IA	FF_IA	DRDY

In our program, we do this by reading from the STATUS_DUP register within a for-loop.

```
59      # Wait until temperature is ready and then read temperature (8b and 12b)
60      # Abort if temp not ready after polling ready bit 15 times
61      print('\n  Poll temp status bit (looking for 0x4x)')
62      for x in range(0, 15):
63          myi2c.write(DEV, [LIS2DW12.STATUS_DUP], 1, iot_hw.i2c_flag.I2C_NO_STOP)
64          myi2c.read(DEV, temp_rdy, 1)
65          print('  -> Read STATUS_DUP reg ({0:#2x}): {1:#2x}'.format(LIS2DW12.STATUS_DUP, temp_rdy[x]))
66          r = temp_rdy[x] & 0x40
67          if r > 0:
68              # Get 8-bit temperature value (one I2C read)
```

Using a for-loop allows us to error-out after 15 attempts… though we found the temperature was always ready by the second reading of the STATUS_DUP register.

After reading the STATUS_DUP register, we used an AND mask to set 'r' to the value of the DRDY_T bit. If greater than zero, we proceeded with reading the temperature from the OUT_T register.

6.5.2.2. Example: Reading 8-bit temp (i2c_temp08.py)

The previous sections discussed the code that is common to reading the temperature with either the 8- or 12-bits of resolution. This section – and the following two subsections – show the code specific to reading the temperature with 8-bits of resolution. In addition, we examine the results of running the 8-bit temperature example.

The full code requires a few pages; therefore, it has been included at the end of the chapter. Look for it in Section 6.7.1.

6.5.2.2.1. Reading the 8-bit temperature

Reading the temperature with 8-bit resolution involves reading the single OUT_T register from the LIS2DW12 sensor. Using the raw value read from the sensor, we can then calculate the degrees in Celsius and Fahrenheit.

```
68      # Get 8-bit temperature value (one I2C read)
69      myi2c.write(DEV, [LIS2DW12.OUT_T], 1, iot_hw.i2c_flag.I2C_STOP)
70      myi2c.read(DEV, raw_temp_08, 1)
71      print('\n  The raw 8-bit temp sensor reading is: {0}\n'.format(raw_temp_08[0]))
72
73      # Here's the calculation of the C and F temperatures for the 8-bit resolution
74      tempC08 = raw_temp_08[0] + 25;
75      tempF08 = (tempC08 * 9.0)/5.0 + 32;
76      print('  Temp8 is  = {0} C or {1} F'.format(tempC08, tempF08))
77
```

You can find the full program listed at the end of the chapter – see Section 6.7.1.

6.5.2.2.2. i2c_temp08.py Results

Running the `i2c_temp08.py` program produces the following result. The print statements let us follow along with each step of the process.

```
/CUSTAPP/iot_files # python i2c_temp08.py

Now running file: i2c_temp08.y

Blink PMOD GPIO 95
  Pmod95 on
  Pmod95 off

Getting temperature from LIS2DW12 via I2C
  Initializing LIS2DW12 for temperature reading:
  -> I2C bus initialized
  -> WHO_AM_I = 0x44 (should be 0x44)
  -> CTRL1 written - 200Hz, single conversion, LPM4
  -> CTRL3 written - Trigger temperature sampling

  Poll temp status bit (looking for 0x4x)
  -> Read STATUS_DUP reg (0x37): 0x1
  -> Read STATUS_DUP reg (0x37): 0x41

  The raw 8-bit temp sensor reading is: -1

  Temp8 is  = 24 C or 75.2 F

  i2c bus was closed

/CUSTAPP/iot files #
```

The actual, raw temperature value read from the LIS2DW12 sensor was 0xFF.

Reading this value from the list returned by the API read function gave us “-1”. In other words, the signed hex value was automatically converted to its signed integer representation.

Note: While not a problem when reading a single register, this automatic conversion complicates the reading of 12-bit values, as shown in the next section.

6.5.2.3. Example: Reading 12-bit temp (i2c_temp12.py)

Like the 8-bit example, this example begins with the same "common code" needed to configure the control registers and trigger the temperature conversion – which was discussed in Section 6.5.2.1.

Once configured and triggered, five more steps are needed to complete the 12-bit example:

1. Read the two raw bytes of data that contain the temperature's 12-bit resolution.
2. Translate the raw bytes back into two's-complement
3. Concatenate the two bytes
4. Translate the full 12-bit value back to an integer
5. Convert the 12-bit temperature reading into Celsius and Fahrenheit.

While steps 1 & 5 are similar to those found in the 8-bit example, steps 2-4 are uniquely required when reading 12- or 14-bit values from the LIS2DW12 sensor using Python.

The next five sections will work through each of these steps, one-by-one.

Note: The full code listing for the `i2c_temp12.py` file is included at the end of the chapter. Please refer to Section 6.7.2.

6.5.2.3.1. Reading the 12-bit data

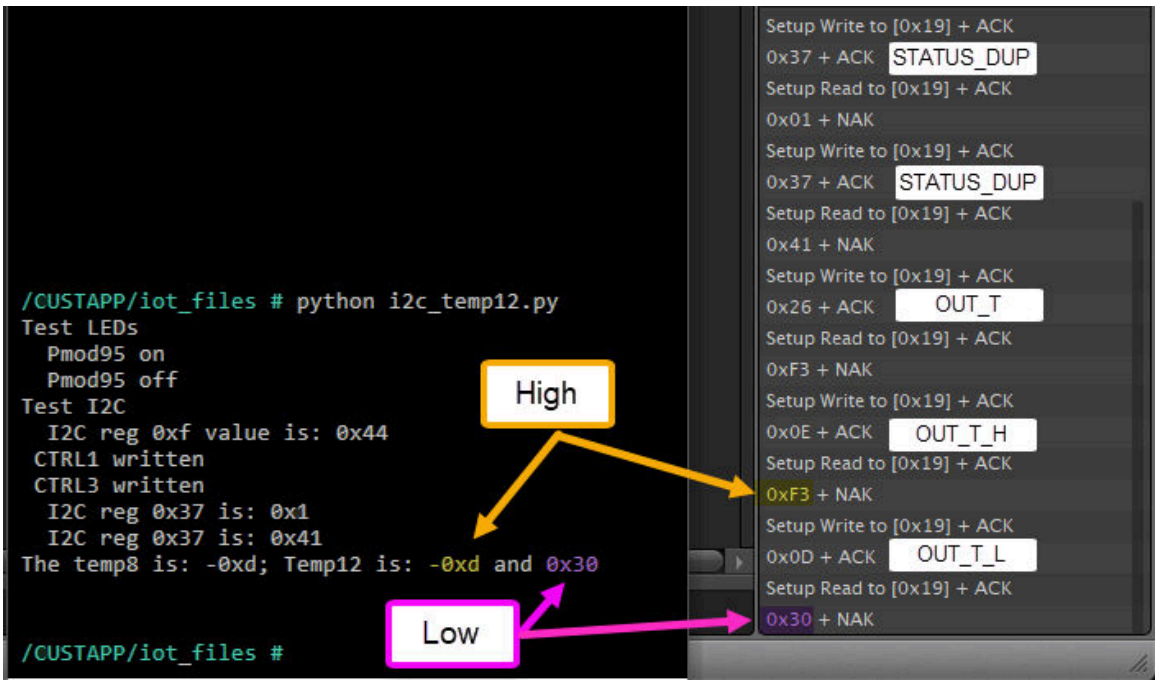
With the configuration complete, the temperature read and ready, the example uses two consecutive I2C read operations to grab both the HIGH and LOW bytes from the 12-bit temperature registers.

```
153     # Get 12-bit temperature data (two I2C reads - high byte, then low)
154     myi2c.write(DEV, [LIS2DW12.OUT_T_H], 1, iot_hw.i2c_flag.I2C_STOP)
155     myi2c.read(DEV, raw_temp_12, 1)
156     myi2c.write(DEV, [LIS2DW12.OUT_T_L], 1, iot_hw.i2c_flag.I2C_STOP)
157     myi2c.read(DEV, raw_temp_12, 1)
158     print('  The raw 12-bit temp sensor reading is: high={0}, low={1}'.format(r
```

6.5.2.3.2. Translating API results back to two's complement

Looking at the results of the 8-bit temperature reading (back in Section 6.5.2.2.2), we noted that the two's complement hex value read from the OUT_T register was automatically converted to a signed integer. While convenient for that case, it complicates the 12-bit example since it prevents us from simply concatenating the HIGH and LOW bytes.

Here's a screen capture showing the values returned from the I2C read() function versus the values seen by the logic analyzer on the I2C bus. In this case the LOW value reads correctly because it's a positive number. But, since the HIGH byte is negative, the value returned by read() is incorrect. Specifically, the negative value reads as -0xD rather than its two's complement value of 0xF3.



Again, the problem described here isn't a concern when the data is only one byte wide but becomes an issue when needing to concatenate two bytes. To resolve the problem, the program contains a simple function which translates the raw number back to a signed two's-complement number.

This function performs a two's complement conversion if the incoming number is negative.

```
53 def val2hex(val, len):
54     max = 2**(len-1)
55 
56     if val < 0:
57         tc = max + val
58         tc = tc + max
59         #print('  ----> Value in: {0}; Hex: {1}'.format(val, hex(tc)))
60         val = tc
61     return val
```

The val2hex() function is called after reading the *raw data* from the sensor. You can see it highlighted below:

```
160       # Call val2hex() for both high/low to get back to twos-complement
161       # Note that Pythonconverts the number incorrectly (which can be val
162       hex_temp_12[HIGH] = val2hex(raw_temp_12[HIGH], 8)
163       hex_temp_12[LOW]  = val2hex(raw_temp_12[LOW],  8)
164       print('  --> The 12-bit hex values: high={0:#2x}, low={1:#2x}'.form
```

6.5.2.3.3. Concatenating upper and lower bytes

After converting the raw data back to two's complement numbers, we must concatenate the HIGH and LOW temperature bytes. One easy way to concatenate the values is to first convert them into strings.

```
166       # We convert the hex numbers to string representations
167       str_temp_12[HIGH] = str(hex(hex_temp_12[HIGH]))[2:].rjust(2, '0')
168       str_temp_12[LOW]  = str(hex(hex_temp_12[LOW]))[2:].rjust(2, '0')
169       #print('  The 12-bit str values: high={0}, low={1}'.format(str_temp_12[HIGH], str_te
170
171       # We can easily concatenate the string versions of the two numbers
172       temp12_string = '0x{0}{1}'.format(str_temp_12[HIGH], str_temp_12[LOW])
173       temp12_string = temp12_string[:5]
174       print('  --> Full 12-bit hex value = {0}'.format(temp12_string))
```

After concatenating the HIGH and LOW bytes in line 172, line 173 slices off the lower four bits.

For example, if the following two values were calculated in lines 167 and 168:

```
167:    str_temp[HIGH] = F3
168:    str_temp[LOW}  = 30
```

The next two lines would evaluate to:

```
172:    temp12_string = 0xF330
173:    temp12_string = 0xF33
```

Where the final `temp12_string` in line 173 represents the 12-bit, two's-complement temperature read from the LIS2DW12 sensor.

6.5.2.3.4. Translating two's complement back to integer

Next let's convert the full 12-bit, two's-complement hex number back to a signed integer using another simple function written for this program.

the 12-bit temperature string value is first converted back into a number before passing it to the u2s() function which converts it back into a signed integer.

```
175
176     # Next we change the full 12-bit string value back to integer and pass to u2s() to get back signed value
177     uTemp12 = int(temp12_string, 16)
178     sTemp12 = u2s(uTemp12, 12)
179     print('  --> Full 12-bit unsigned value: {0}; signed value: {1}\n'.format(uTemp12, sTemp12))
180
```

Looking back to line 82 of our program we find the u2s() function defined:

```
81 def u2s(uVal, len):
82     max = 2**len / 2
83     sVal = uVal
84
85     if uVal > max:
86         sVal = uVal - (2 * max)
87         #print('  ----> unsigned input: {0}; signed output: {1}'.format(uVal, sVal))
88     return sVal
89
```

The u2s() function checks if the input value is larger than the maximum range – based on the radix length. For example, given a 12-bit number, the maximum range is 2^12 divided by 2 (or "2048"). Therefore, if the 12-bit input value is larger than 2048, the function subtracts 4096 to arrive at the negative signed integer value representation.

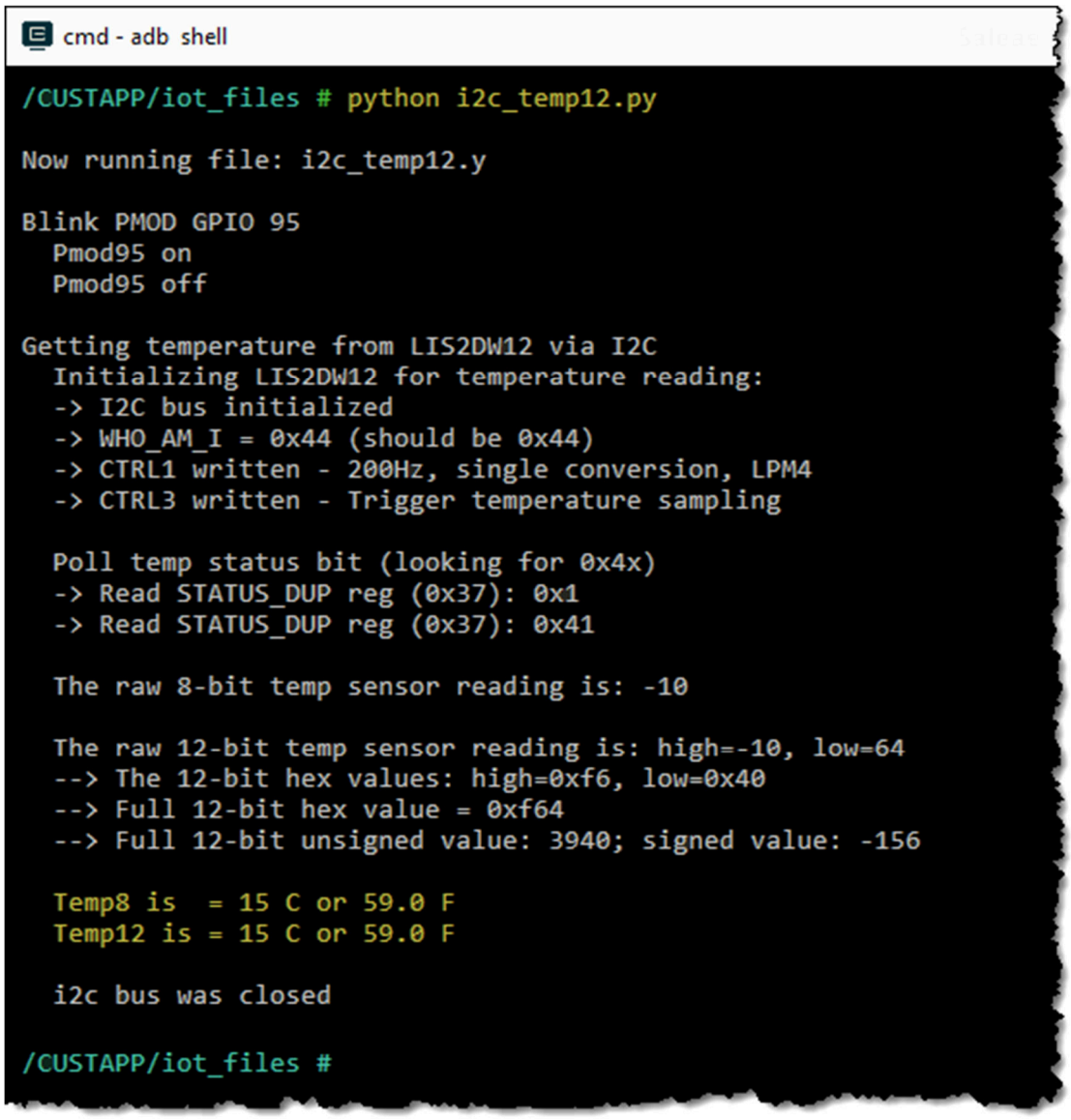
6.5.2.3.5. Converting to Celsius and Fahrenheit

Calculating Celsius and Fahrenheit for 12-bit number is the same as for 8-bit numbers – excepting one additional scaling operation required to account for the extra four bits. Dividing by 16 (2^4-bits) scales the number down to the range required for the Celsius/Fahrenheit calculation.

```
184
185         # Similarly, the calculation of C/F for the 12-bit number
186         # Notice the extra "divide by 16" required to handle the extra 4 bits of precision
187         tempC12 = sTemp12/16 + 25;
188         tempF12 = (tempC12 * 9.0)/5.0 + 32;
189
```

6.5.2.3.6. Results

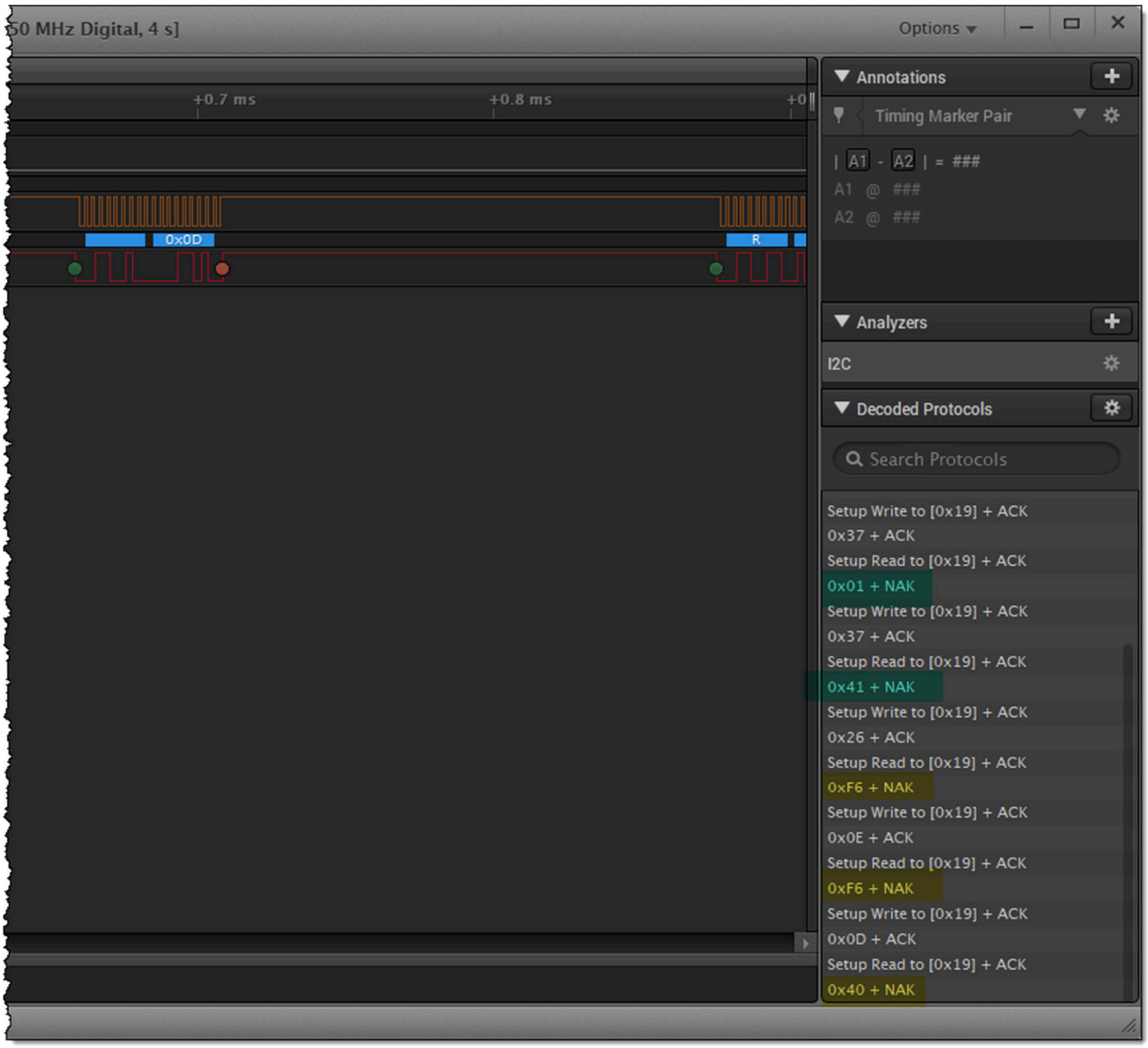
The program's print statements let us follow along while it's running.



Hint:	We found that by setting the SK2 on top of a frozen pork loin we could generate lower temperatures during testing. Thankfully, the SK2 runs cool enough that it didn't cook it!

6.5.2.3.7. Logic Analyzer Results

Using a logic analyzer proved invaluable when writing the 12-bit temperature I2C program. It allowed us to view the 'true' data coming from the sensor, even when the Python's automatic conversion initially obscured the results.



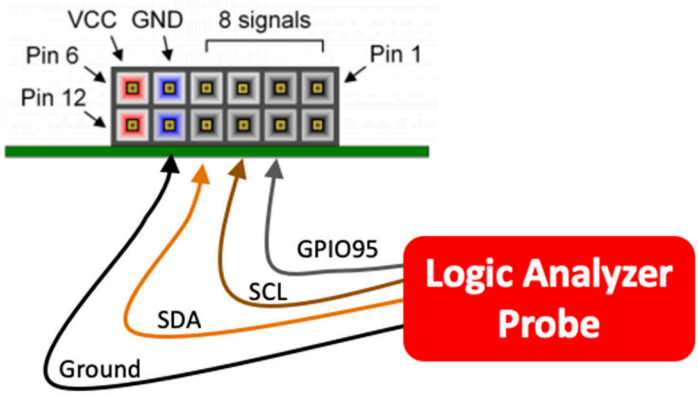
6.6. Tracing I2C Signals with a Logic Analyzer

Since the SK2's I2C bus connects in parallel to all I2C devices and connectors, we can watch I2C messages going between the WNC processor and the LIS2DW12 sensor by connecting a Logic Analyzer to the PMOD I2C pins.

Hint:	You can also use the Expansion Connector's I2C signals to watch I2C messages, but the PMOD connector was easier to access.

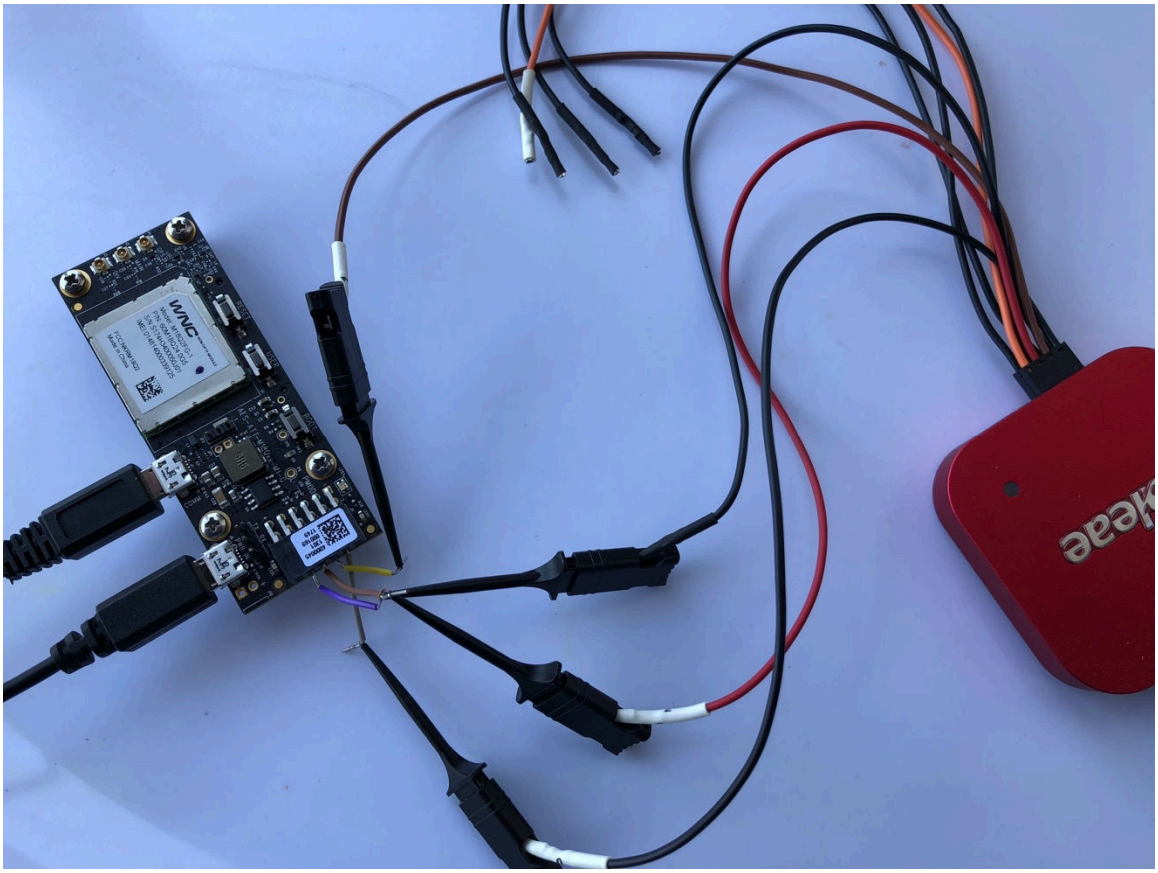
6.6.1. Pin Connections

Creating the examples for this chapter we connected four leads from our logic analyzer to the PMOD connector.



Connecting the GPIO95 lead was not required for tracing the I2C bus, but we used it to trigger the logic analyzer capture.

The following photo demonstrates connecting a Saleae Logic Analyzer to the PMOD connector following the pinout described above. This setup was used to capture and visualize the I2C signals for the examples provided with this chapter.



6.6.2. Logic Analyzer Captures

The logic analyzer allowed us to trace the signals coming from the PMOD connector.

We arbitrarily connected the signals in this manner:

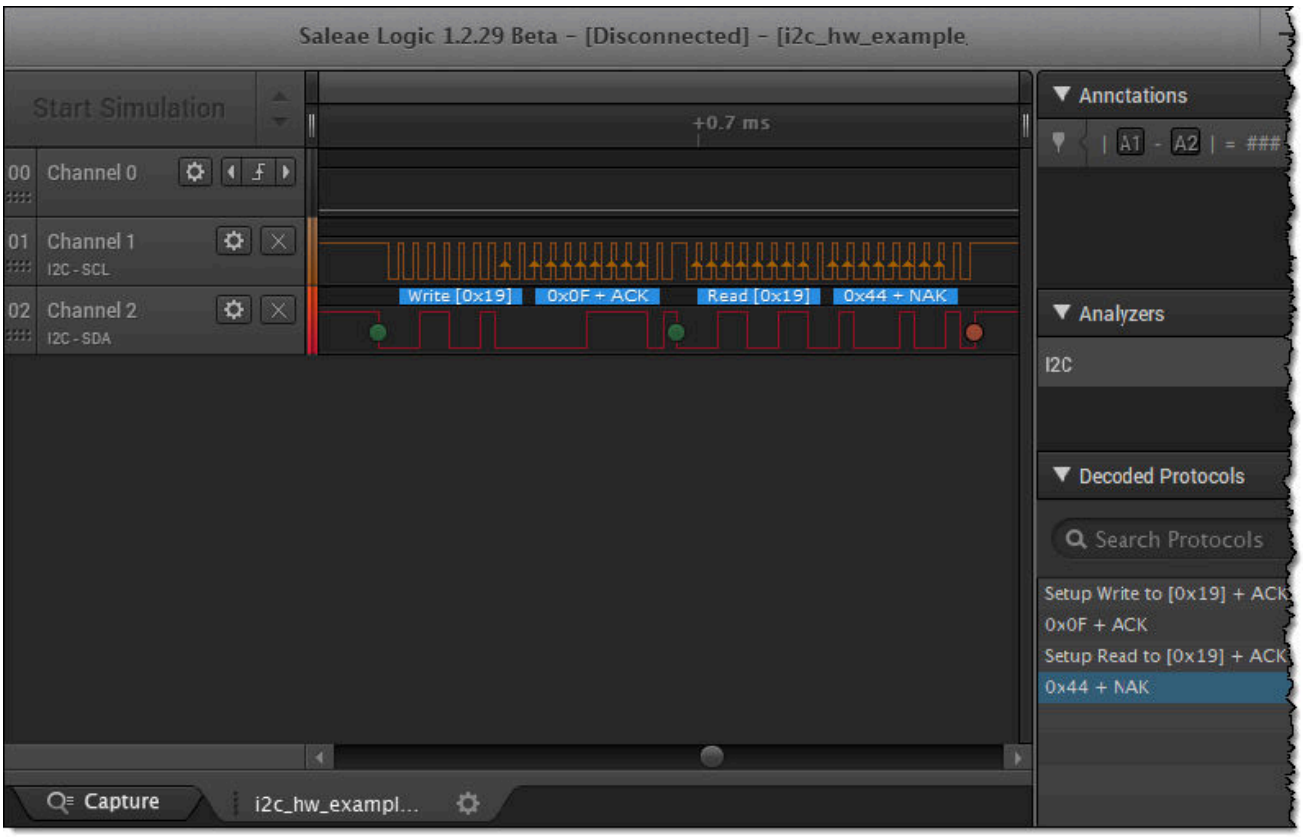
Channel 0: GPIO95

Channel 1: I2C SCL

Channel 2: I2C SDA

Like many logic analyzers, the Saleae brand allowed us to define the SCL and SDA channels so that they could be automatically decoded for us. This allowed the analyzer to show us the values being transferred on the bus in the graphical display, as well as providing a list of transfers in the right-hand column.

Here's an example of a Logic Analyzer capture for the "Who Am I" example (i2c_hw_example.py):



The logic analyzer was key debugging the examples for this chapter. Specifically, it helped in two ways:

- The repeated start bits were easily viewed in the logic analyzer display – and not something anticipated after quickly reading through the API guide. This eventually allowed us to figure out the control registers were not being programmed when writing to them with repeat start bits.
- The 12-bit resolution temperature example (i2c_temp12.py) was not originally giving us readings which matched the 8-bit resolution results. Comparing the data on the I2C bus (viewed through the logic analyzer) helped us understand that the I2C read() function was automatically translating the values – which corrupted our concatenating the HIGH and LOW bytes.

6.7. Full Code Listings

6.7.1. i2c_temp08.py

Listing 6.2: i2c_temp08.py

```
from __future__ import print_function

"""
i2c_temp08.py

Read the 8-bit resolution temperature register from the LIS2DW12 sensor
using the I2C bus.

"""
import iot_hw                           # Required for GPIO and I2C drivers
import time                             # Required for toggling GPIO

class LIS2DW12:                         # Class used to define LIS2DW12 sensor registers & constants
    # LIS2DW12 registers
    WHO_AM_I   = 0x0f                   # LIS2DW12 identification register (always reads as 0x44)
    CTRL1      = 0x20                   # Control Register 1
    CTRL3      = 0x22                   # Control Register 3
    OUT_T      = 0x26                   # 8-bit temperature value register
    STATUS_DUP = 0x37                   # Another status register (this one has temperature status bits)

    # Constants
    ID         = 0x44                   # Value read back from WHO_AM_I register

def Get_Temperature_i2c():
    print('\nGetting temperature from LIS2DW12 via I2C')
    # Define CONSTANTS
    DEV  = 0x19                         # i2c device slave address for LIS2DW12 accelerometer

    # Allocate variables
    myi2c     = 0                       # Object returned from I2C open API
    i2cClosed = 0                       # Return variable from closing i2c bus API

    lis2dw12_id = list()                # List variable for results from "WHO_AM_I" register
    raw_temp_08 = list()                # List variable for reading 8-bit temperature register
    temp_rdy    = list()                # List variable for polling status register


    tempC08 = 0                         # Variables for 8-bit Celsius and Fahrenheit temperature
    tempF08 = 0


    print('  Initializing LIS2DW12 for temperature reading:')
    # Initialize i2c bus
    myi2c = iot_hw.i2c()                # Initialize (i.e. open) the i2c bus protocol
    print('  -> I2C bus initialized')

    # Get device_id from LIS2DW12 WHO_AM_I register (not required,
    # but the first thing we tried when using this sensor
    myi2c.write(DEV, [LIS2DW12.WHO_AM_I], 1, iot_hw.i2c_flag.I2C_NO_STOP)
    myi2c.read(DEV, lis2dw12_id, 1)     # Read data from the WHO_AM_I register
    print('  -> WHO_AM_I = {0:#2x} (should be {1:#2x})'.format(lis2dw12_id[0], LIS2DW12.ID))

    # Write configuration Control Register 1 (CTRL1)
```

```
    myi2c.write(DEV, [LIS2DW12.CTRL1, 0x6B], 2, iot_hw.i2c_flag.I2C_NO_STOP)
    print('  -> CTRL1 written - 200Hz, single conversion, LPM4')

    # Trigger temperature sampling/converstion via CTRL3 register
    myi2c.write(DEV, [LIS2DW12.CTRL3, 0x03], 2, iot_hw.i2c_flag.I2C_NO_STOP)
    print('  -> CTRL3 written - Trigger temperature sampling')

    # Wait until temperature is ready and then read temperature (8b and 12b)
    # Abort if temp not ready after polling ready bit 15 times
    print('\n  Poll temp status bit (looking for 0x4x)')
    for x in range(0, 15):
        myi2c.write(DEV, [LIS2DW12.STATUS_DUP], 1, iot_hw.i2c_flag.I2C_NO_STOP) # Check status of DRDY_T
        myi2c.read(DEV, temp_rdy, 1)    # Read data
        print('  -> Read STATUS_DUP reg ({0:#2x}): {1:#2x}'.format(LIS2DW12.STATUS_DUP, temp_rdy[x]))
        r = temp_rdy[x] & 0x40          # if DRDY_T bit is set, read the temperature
        if r > 0:
            # Get 8-bit temperature value (one I2C read)
            myi2c.write(DEV, [LIS2DW12.OUT_T], 1, iot_hw.i2c_flag.I2C_STOP)     # Send request to OUT_T
            myi2c.read(DEV, raw_temp_08, 1)                                     # Read 8-bit temp data
            print('\n  The raw 8-bit temp sensor reading is: {0}\n'.format(raw_temp_08[0]))

            # Here's the calculation of the C and F temperatures for the 8-bit resolution
            tempC08 = raw_temp_08[0] + 25;
            tempF08 = (tempC08 * 9.0)/5.0 + 32;
            print('  Temp8 is  = {0} C or {1} F'.format(tempC08, tempF08))

            # Break out of the polling loop if we successfully read and printed the temperature
            break

        # If x reaches 14, then we polled 15 times and failed to read the temperature
        if x == 14:
            print('\n(FAIL) Failed to get temperature...')

    i2cClosed = myi2c.close()                                   # Close the i2c bus
    if i2cClosed:
        print('\n  i2c bus was closed')
    else:
        print('\n  FAILED to close i2c bus')
    print('\n')


def Blink_GPIO():
    print('\nBlink PMOD GPIO 95')

    leds      = {"Pmod95":iot_hw.gpio_pin.GPIO_PIN_95}      # List of GPIOs to toggle
    myled     = 0                                           # Object returned from GPIO open API

                    for key in leds:                        # Sequence through all GPIO in 'leds' list
                        myled = iot_hw.gpio(leds[key])
                        myled.set_dir(iot_hw.gpio_direction.GPIO_DIR_OUTPUT)
                        print('  {} on'.format(key))

                        myled.write(iot_hw.gpio_level.GPIO_LEVEL_HIGH)       # Set GPIO high
                        time.sleep(0.5)                                      # Sleep for 1/2 second
                        print('  {} off'.format(key))

                        myled.write(iot_hw.gpio_level.GPIO_LEVEL_LOW)        # Set GPIO low
                        time.sleep(0.5)                                      # Sleep for 1/2 second

                        myled.close()                       # Close the driver for that GPIO key

```

```
114.
115.                 def main():
116.                     print('\nNow running file: i2c_temp08.y')
117.                     Blink_GPIO()
118.                     Get_Temperature_i2c()
119.
120.
121.                 if __name__ == "__main__":
122.                     try:
123.                         main()
124.                     except:
125.                         if False:
126.                             # type, value, tb = sys.exc_info()
127.                             # traceback.print_exc()
128.                             # pdb.post_mortem(tb)
129.                             print('__main__ error')
130.                         else:
131.                             raise
```

6.7.2. i2c_temp12.py

Listing 6.3: i2c_temp12.py

```
1.  from __future__ import print_function
2.
3.  """
4.  i2c_temp12.py
5.
6.  Read the 8-bit resolution and 12-bit resolution temperature
7.  register from the LIS2DW12 sensor using the I2C bus.
8.
9.  """
10.
11. import iot_hw                           # Required for GPIO and I2C drivers
12. import time                             # Required for toggling GPIO
13.
14. class LIS2DW12:                         # Class used to define LIS2DW12 sensor registers & constants
15.     # LIS2DW12 registers
16.     OUT_T_L    = 0x0D                   # Lower 8-bits of 12-bit temp (lower 4-bits always read as 0)
17.     OUT_T_H    = 0x0E                   # Upper 8-bits of 12-bit temperature
18.     WHO_AM_I   = 0x0f                   # LIS2DW12 identification register (always reads as 0x44)
19.     CTRL1      = 0x20                   # Control Register 1
20.     CTRL3      = 0x22                   # Control Register 3
21.     OUT_T      = 0x26                   # 8-bit temperature value register
22.     STATUS_DUP = 0x37                   # Another status register (this one has temperature status bits)
23.
24.     # Constants
25.     ID         = 0x44                   # Value read back from WHO_AM_I register
26.
27. """
28. val2hex()
29.
30. If the input value is negative, its converted to a twos-complement signed
31. hex value. If the number is positive, the input value is returned.
32.
33. Background:
34.     For a given radix (e.g. 8-bit numbers) signed and unsigned numbers provide
35.     the same resolution, although their min/max values change. For example, an
```

```
36.     8-bit   unsigned number range is 0 to 255, while the signed number range
37.     is -128 to 127.
38.
39.     With this in mind, we can convert a negative unsigned number to its
40.     twos-complement equivelent by adding the range maximum (i.e. 128) to the
41.     difference between the input value and the maximum value.
42.
43. Parameters:
44.     val = signed integer
45.     len = number of bits for output value
46.
47. Returns:
48.     twos complement signed hex number
49.
50. Note:
51.     No error handling is implemented in this function
52. """
53. def val2hex(val, len):
54.     max = 2**(len-1)                    # Determin the maximum value based on radix bit length
55.
56.     if val < 0:
57.         tc = max + val                  # Calculate the difference between the input value and max
58.         tc = tc + max                   # Add max to shift range
59.         #print('  ----> Value in: {0}; Hex: {1}'.format(val, hex(tc)))
60.         val = tc
61.     return val
62.
63.
64. """
65. u2s()
66.
67. If the input number is greater than the max value (based on the radix length)
68. then we can assume the number must be negative, in which case we convert it
69. to its signed value representation.
70.
71. Parameters:
72.     uVal = input number
73.     len  = number of radix bits
74.
75. Returns:
76.     twos complement signed hex number
77.
78. Note:
79.     No error handling is implemented in this function
80. """
81. def u2s(uVal, len):
82.     max = 2**len / 2                    # Determin the maximum value based on radix bit length
83.     sVal = uVal                         # Set return value to the input value (in case it's positive)
84.
85.     if uVal > max:                      # Check if number is greater than radix max
86.         sVal = uVal - (2 * max)         # Calculate the negative representation of the number
87.         #print('  ----> unsigned input: {0}; signed output: {1}'.format(uVal, sVal))
88.     return sVal
89.
90.
91. def Get_Temperature_i2c():
92.     print('\nGetting temperature from LIS2DW12 via I2C')
93.     # Define CONSTANTS
94.     DEV  = 0x19                         # i2c device slave address for LIS2DW12 accelerometer
95.     HIGH = 0                            # Index for 'high' byte of 12-bit temperature list
96.     LOW  = 1                            # Index for 'low' byte of 12-bit temperature list
```

```
97.
98.     # Allocate variables
99.     myi2c     = 0                       # Object returned from I2C open API
100.                    i2cClosed = 0            # Return variable from closing i2c bus API
101.
102.                    lis2dw12_id = list()     # List variable for results from "WHO_AM_I" register
103.                    raw_temp_08 = list()     # List variable for reading 8-bit temperature register
104.                    raw_temp_12 = list()     # List variable for reading 12-bit temperature register
105.                    hex_temp_12 = [0, 0]     # Temporary list for holding 12-bit hex values
106.                    str_temp_12 = ['', '']   # Temporary string for holding 12-bit hex-string values
107.                    temp_rdy    = list()     # List variable for polling status register
108.
109.                    x = 0                    # For-loop variable
110.                    r = 0                    # Masked ready bit (is temperature value ready?)
111.                    temp12_string = 0        # String variable holds full 12-bit (concatenated) temp
112.                    uTemp12 = 0              # Temporary variable to hold Unsigned integer conversion
113.                    sTemp12 = 0              # Temporary variable to hold Signed integer conversion
114.
115.                    tempC08 = 0              # Variables for 8-bit Celsius and Fahrenheit temperature
116.                    tempF08 = 0
117.                    tempC12 = 0              # Variables for 12-bit Celsius and Fahrenheit
118.                    tempF12 = 0
119.
120.
121.                    print('  Initializing LIS2DW12 for temperature reading:')
122.                    # Initialize i2c bus
123.                    myi2c = iot_hw.i2c()      # Initialize (i.e. open) the i2c bus protocol
124.                    print('  -> I2C bus initialized')
125.
126.                    # Get device_id from LIS2DW12 WHO_AM_I register
                        # (not required, but the first thing we tried when using this sensor
127.                    myi2c.write(DEV, [LIS2DW12.WHO_AM_I], 1, iot_hw.i2c_flag.I2C_NO_STOP)
128.                    myi2c.read(DEV, lis2dw12_id, 1)
129.                    print('  -> WHO_AM_I = {0:#2x} (should be {1:#2x})'.format
                                                        (lis2dw12_id[0], LIS2DW12.ID))
130.
131.                    # Write configuration Control Register 1 (CTRL1)
132.                    myi2c.write(DEV, [LIS2DW12.CTRL1, 0x6B], 2, iot_hw.i2c_flag.I2C_NO_STOP)
133.                    print('  -> CTRL1 written - 200Hz, single conversion, LPM4')
134.
135.                    # Trigger temperature sampling/converstion via CTRL3 register
136.                    myi2c.write(DEV, [LIS2DW12.CTRL3, 0x03], 2, iot_hw.i2c_flag.I2C_NO_STOP)
137.                    print('  -> CTRL3 written - Trigger temperature sampling')
138.
139.                    # Wait until temperature is ready and then read temperature (8b and 12b)
140.                    # Abort if temp not ready after polling ready bit 15 times
141.                    print('\n  Poll temp status bit (looking for 0x4x)')
142.                    for x in range(0, 15):
143.                        myi2c.write(DEV, [LIS2DW12.STATUS_DUP], 1, iot_hw.i2c_flag.I2C_NO_STOP)
144.                        myi2c.read(DEV, temp_rdy, 1)    # Read data
145.                        print('  -> Read STATUS_DUP reg ({0:#2x}): {1:#2x}'.format
                                                            (LIS2DW12.STATUS_DUP, temp_rdy[x]))
146.                        r = temp_rdy[x] & 0x40      # if DRDY_T bit is set, read the temperature
147.                        if r > 0:
148.                            # Get 8-bit temperature value (one I2C read)
149.                            myi2c.write(DEV, [LIS2DW12.OUT_T], 1, iot_hw.i2c_flag.I2C_STOP)
150.                            myi2c.read(DEV, raw_temp_08, 1)
151.                            print('\n  The raw 8-bit temp sensor reading is: {0}\n'
                                                                    .format(raw_temp_08[0]))
152.
153.                            # Get 12-bit temperature data (two I2C reads - high byte, then low)
```

```
154.                            myi2c.write(DEV, [LIS2DW12.OUT_T_H], 1, iot_hw.i2c_flag.I2C_STOP)
155.                            myi2c.read(DEV, raw_temp_12, 1)
156.                            myi2c.write(DEV, [LIS2DW12.OUT_T_L], 1, iot_hw.i2c_flag.I2C_STOP)
157.                            myi2c.read(DEV, raw_temp_12, 1)
158.                            print('  The raw 12-bit temp sensor reading is: high={0},
                                        low={1}'.format(raw_temp_12[HIGH], raw_temp_12[LOW]))
159.
160.                            # Call val2hex() for both high/low to get back to twos-complement value
161.                            # Note that Pythonconverts the number incorrectly (which can be
                                  validated by viewing i2c bus in logic analyzer)
162.                            hex_temp_12[HIGH] = val2hex(raw_temp_12[HIGH], 8)
163.                            hex_temp_12[LOW]  = val2hex(raw_temp_12[LOW],  8)
164.                            print('  --> The 12-bit hex values: high={0:#2x}, low={1:#2x}'
                                                .format(hex_temp_12[HIGH], hex_temp_12[LOW]))
165.
166.                            # We convert the hex numbers to string representations
167.                            str_temp_12[HIGH] = str(hex(hex_temp_12[HIGH]))[2:].rjust(2, '0')
168.                            str_temp_12[LOW]  = str(hex(hex_temp_12[LOW]))[2:].rjust(2, '0')
169.                            #print('  The 12-bit str values: high={0}, low={1}'.
                                                format(str_temp_12[HIGH], str_temp_12[LOW]))
170.
171.                            # We can easily concatonate the string versions of the two numbers
172.                            temp12_string = '0x{0}{1}'.format(str_temp_12[HIGH], str_temp_12[LOW])
173.                            temp12_string = temp12_string[:5]
174.                            print('  --> Full 12-bit hex value = {0}'.format(temp12_string))
175.
176.                            # Next we change the full 12-bit string value back to
                               # integer and pass to u2s() to get back signed value
177.                            uTemp12 = int(temp12_string, 16)
178.                            sTemp12 = u2s(uTemp12, 12)
179.                            print('  --> Full 12-bit unsigned value: {0}; signed value:
                                                        {1}\n'.format(uTemp12, sTemp12))
180.
181.                            # Here's the calculation for the C and F temp's for the 8-bit resolution
182.                            tempC08 = raw_temp_08[0] + 25;
183.                            tempF08 = (tempC08 * 9.0)/5.0 + 32;
184.
185.                            # Similarly, the calculation of C/F for the 12-bit number
186.                            # Notice the extra "divide by 16" required to handle the
                                  extra 4 bits of precision
187.                            tempC12 = sTemp12/16+ 25;
188.                            tempF12 = (tempC12 * 9.0)/5.0 + 32;
189.
190.                            print('  Temp8 is  = {0} C or {1} F'.format(tempC08, tempF08))
191.                            print('  Temp12 is = {0} C or {1} F'.format(tempC12, tempF12))
192.
193.                            # Break out of the polling loop if we
                               # successfully read and printed the temperature
194.                            break
195.
196.                        # If x reaches 14, then we polled 15 times and failed to read the temperature
197.                        if x == 14:
198.                            print('\n(FAIL) Failed to get temperature...')
199.
200.                    i2cClosed = myi2c.close()                          # Close the i2c bus
201.                    if i2cClosed:
202.                        print('\n  i2c bus was closed')
203.                    else:
204.                        print('\n  FAILED to close i2c bus')
205.                    print('\n')
206.
```

```
207.
208.                 def Blink_GPIO():
209.                     print('\nBlink PMOD GPIO 95')
210.
211.                     leds      = {"Pmod95":iot_hw.gpio_pin.GPIO_PIN_95}   # List of GPIOs to toggle
212.                     myled     = 0
213.
214.                     for key in leds:
215.                         myled = iot_hw.gpio(leds[key])                   # Open each GPIO key
216.                         myled.set_dir(iot_hw.gpio_direction.GPIO_DIR_OUTPUT)
217.                         print('  {} on'.format(key))
218.
219.                         myled.write(iot_hw.gpio_level.GPIO_LEVEL_HIGH)   # Set GPIO high
220.                         time.sleep(0.5)                                  # Sleep for 1/2 second
221.                         print('  {} off'.format(key))
222.
223.                         myled.write(iot_hw.gpio_level.GPIO_LEVEL_LOW)
224.
225.                     # Set GPIO low
226.                         time.sleep(0.5)                          # Sleep for 1/2 second
227.
228.                         myled.close()                            # Close the driver for that GPIO key
229.
230.
231.                 def main():
232.                     print('\nNow running file: i2c_temp12.y')
233.                     Blink_GPIO()
234.                     Get_Temperature_i2c()
235.
236.
237.                 if __name__ == "__main__":
238.                     try:
239.                         main()
240.                     except:
241.                         if False:
242.                             # type, value, tb = sys.exc_info()
243.                             # traceback.print_exc()
244.                             # pdb.post_mortem(tb)
245.                             print('__main__ error')
246.                         else:
247.                             raise
```

Appendix

Topics

A1. 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

A2. 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 (Appendix) 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.

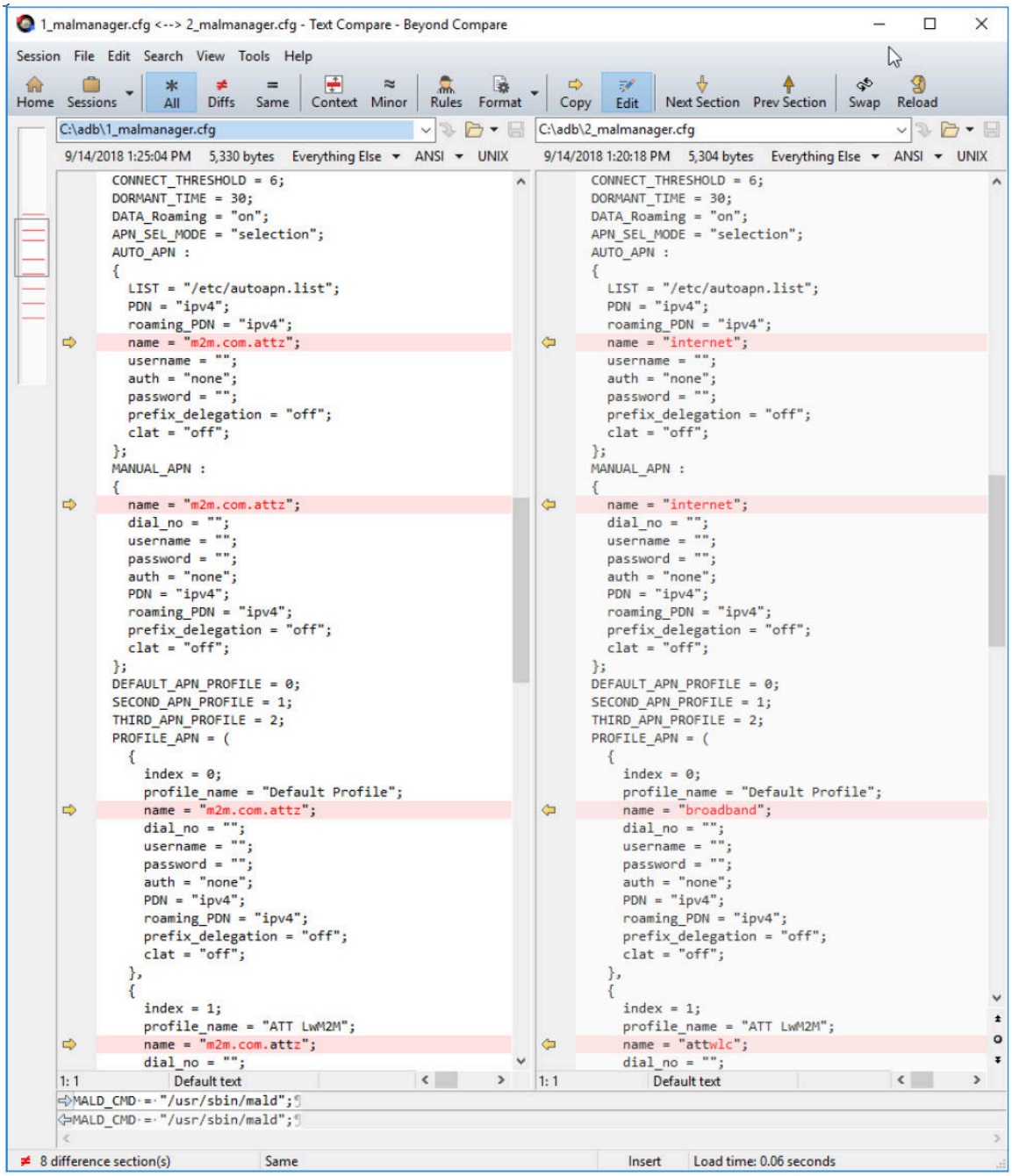


Figure (Appendix) A-1 – Malmanager.cfg comparison

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.**

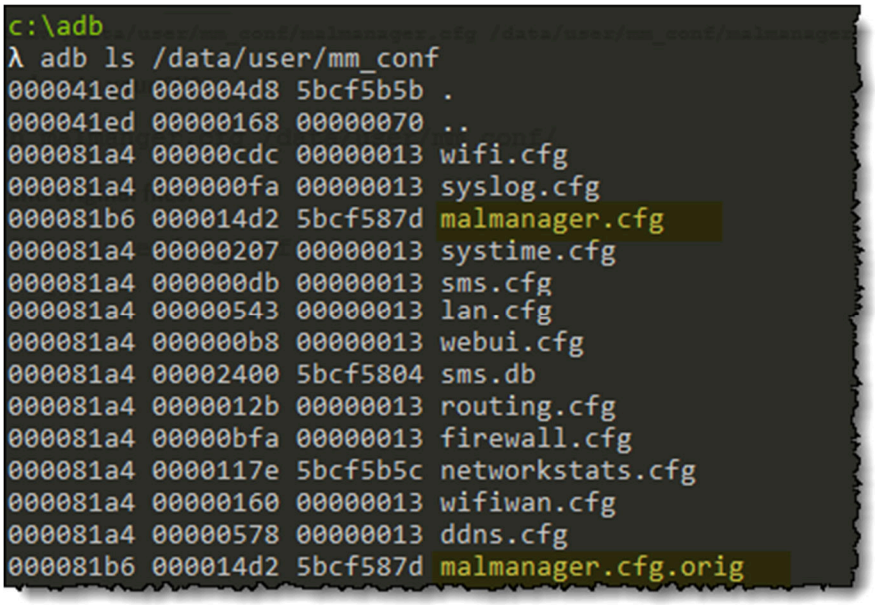
   ```
   adb shell mv /data/user/mm_conf/malmanager.cfg /data/user/mm_conf/malmanager.cfg.orig
   ```

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.**

A3. 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:

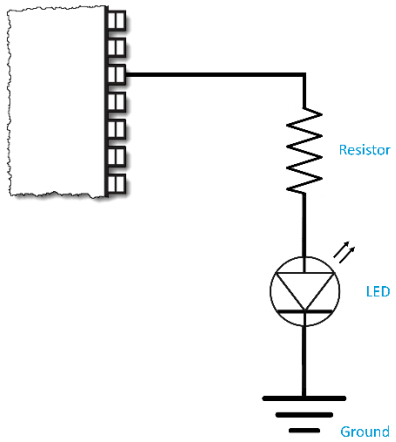


Figure (Appendix) 2 - Generic LED Circuit

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.

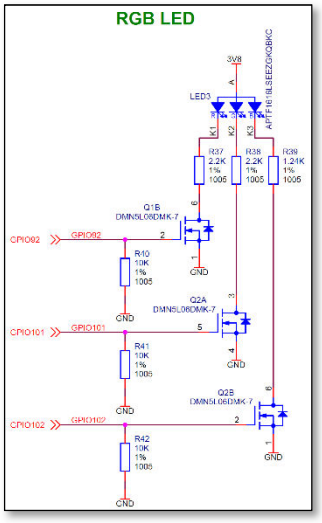


Figure (Appendix) 3 - RGB LED Schematic

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

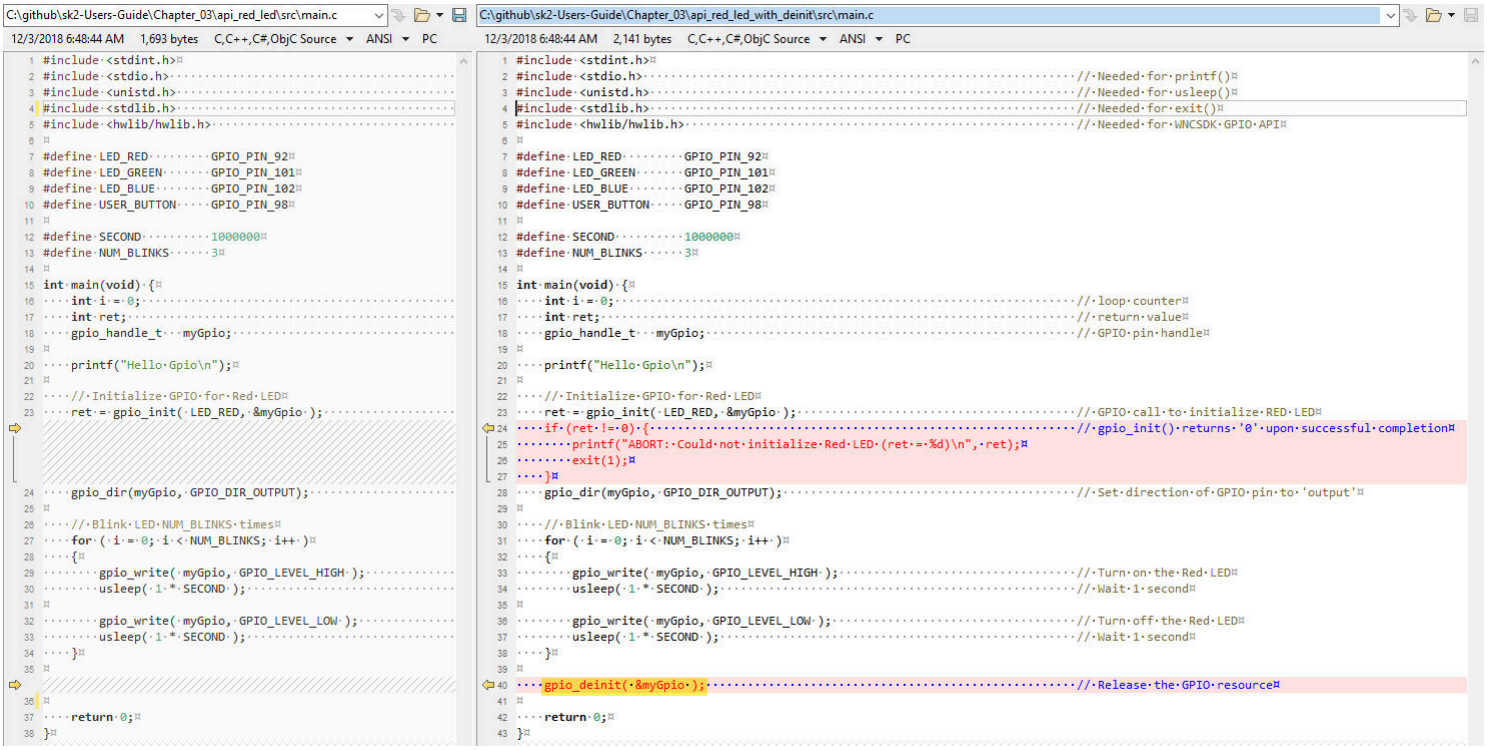
```
# userButtonLed.sh
#
# This example reads the value of the user button and turns
# on either the Green or Red RGB LED depending upon the value
# of the user button
#

# 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

Deallocating GPIO Resources

It likely goes without saying that most programmers realize that they should deallocate any resources they used at the end of their programs. Needless to say, our early blink LED examples did not, which caused problems when running and debugging them. To address this, we added the deinit() function as well as testing for whether we obtained the resource during init(). The differences can be seen in this comparison:



This Appendix discusses what went wrong with the original example as well as our experiences debugging the example.

What's wrong with this example?

The original code (api_red_led/src/main.c) runs, so what's wrong with it?

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 fixed this problem (discussed next) it can still cause problems when debugging buggy code.

We can easily solve the allocation/deallocation problem by inserting the following line of code at the end of the 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
```

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.

A4. 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.

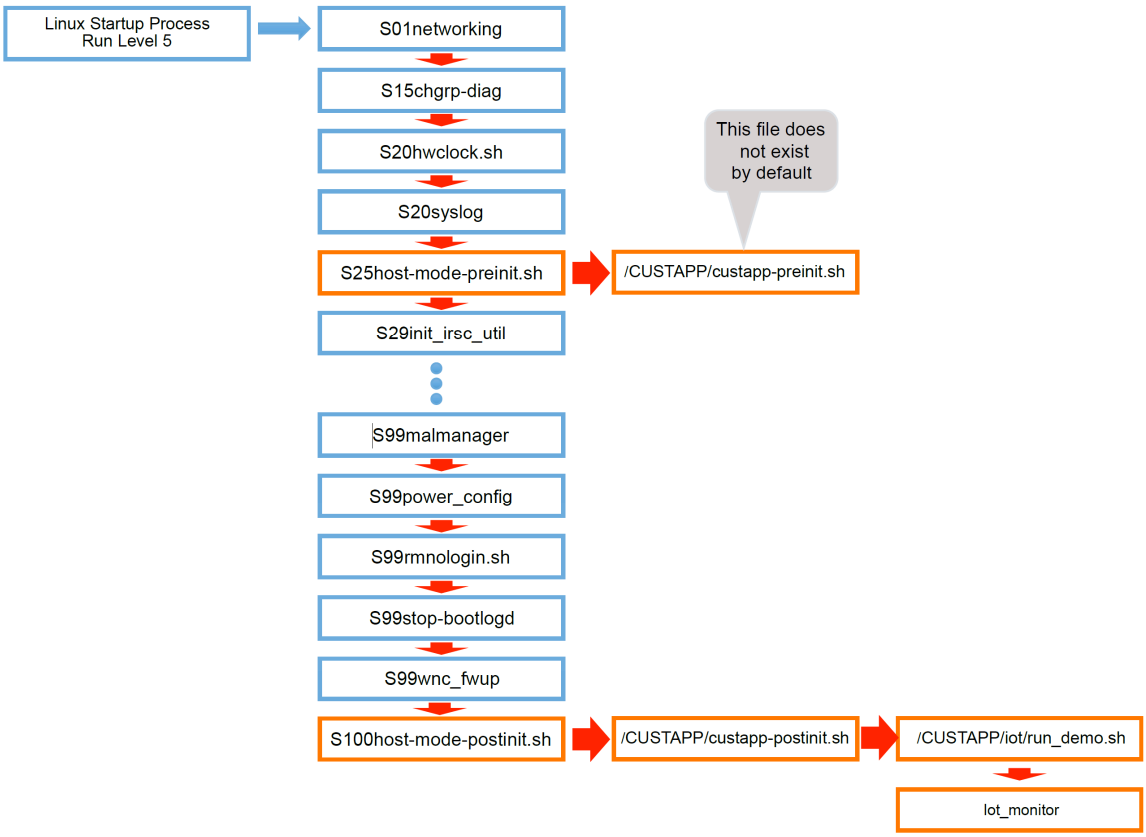


Figure (Appendix) 6 - Linux Boot Sequence - running files from /etc/rc5.d

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.

A5. Troubleshooting “adb devices”

Chapter 2 (Section 2.4) describes how to connect the AT&T IoT Starter Kit (2nd Generation) (i.e. SK2) to your development computer using ADB. This includes both installing ADB as well as the commands that can be issued to start the service, find your device, and connect to it remotely.

Ideally, when you enter the “`adb devices`” command into your terminal, the response will be:

```
WNC_ADB device
```

This appendix topic provides a few troubleshooting suggestions for when this expected response doesn’t occur.

Here are the troubleshooting topics:

A5.1 Cannot find ADB command

For example, when executing the “adb devices” command, we received the following error.

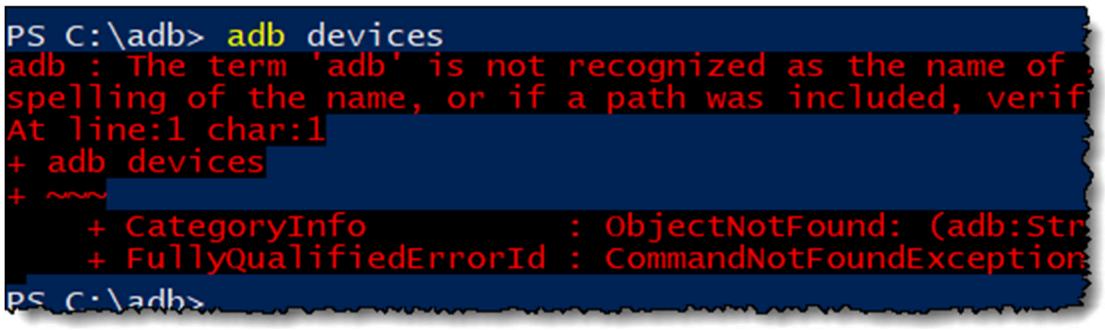


Figure (Appendix) 7 - Mac/Linux/PowerShell needs “./”

There are three reasons the ADB commands might not be recognized by your system.

5. **ADB was not installed** (or improperly installed) on your computer. If this is the case, go review and implement the installation directions in Section 2.4.1.
6. **Not running ADB from the “adb” folder.** When following the ADB installation directions in this guide (and most Internet sites), you need to execute ADB commands from the ADB installation folder. The exception to that in this guide was if you installed ADB with apt-get on an Ubuntu (i.e. Debian) Linux computer. Please refer to the next topic (Section A5.2.)
7. **Precede the ADB command with *./***

   A good rule of thumb would be, “If the ***adb devices*** command doesn’t work on your computer, then try ***./adb devices***.”

   When using Windows PowerShell, Mac, or Linux, you need to precede commands with the ./ characters. Therefore, you should do the same with ADB commands.

   ```
   ./adb devices
   ```

   Note that PowerShell also accepts using the backslash character: `.\adb devices`

A5.2 Execute From ADB Directory

The ADB executable wasn’t found:

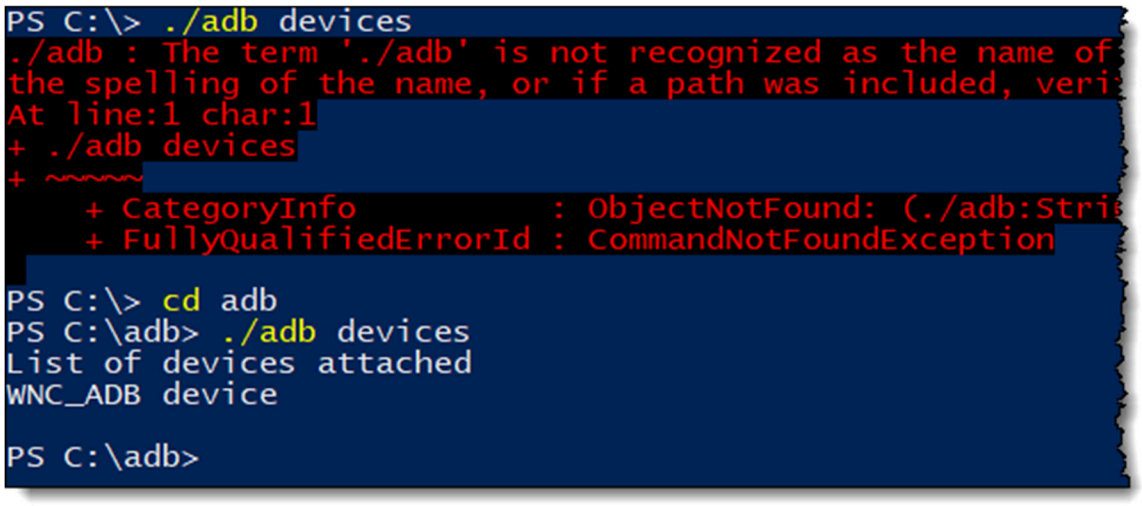


Figure (Appendix) 8 – Execute from the ADB Folder

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
adb devices
```

to get to the proper location before running the “adb devices” command.

A5.3 WNC_ADB unauthorized

When executing the “adb devices”, the command returns the error:

```
WNC_ADB unauthorized
```

Such as in this example:

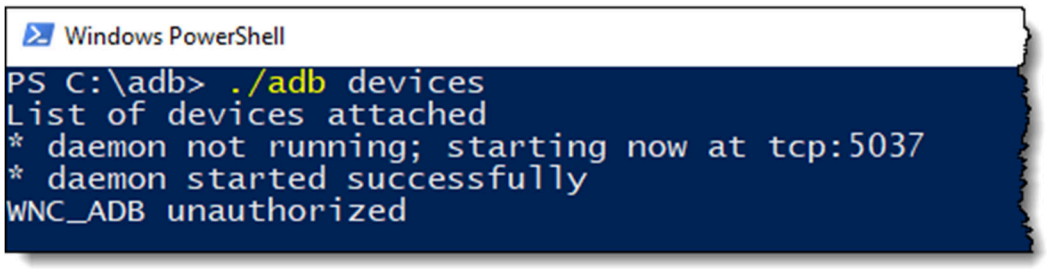


Figure (Appendix) 9 – WNC_ADB unauthorized

In this case, it appears that the ADB server started and found the SK2 (our WNC_ADB) device but did not find the proper credentials to access the device. This error occurs when ADB cannot find the correct authorization key in your `.android` directory.

Examine your `.android` directory. It should now contain two files:

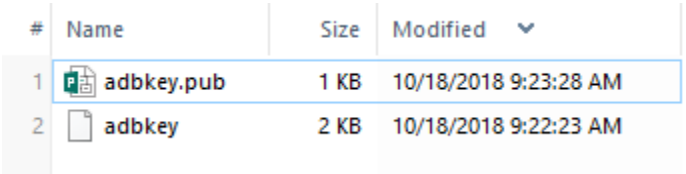


Figure (Appendix) 10 – adb keys

`adbkey` is a private key generated by the ADB Server based upon the `adbkey.pub` file we placed in our `.android` directory, if you followed our ADB Installation directions in Section 2.4.1).

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

Figure (Appendix) 11 - Private key

If our device is unauthorized, then something is likely wrong with this file. Since it is generated from `adbkey.pub`, it is likely that the correct `adbkey.pub` was not present before running “adb devices” or has been corrupted. (For some reason, starting the ADB server sometimes overwrites `adbkey.pub` with an incorrect hash string.)

Procedure to Correct “WNC_ADB 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. You can do this by manually enter the word or by downloading this file from the Avnet GitHub.

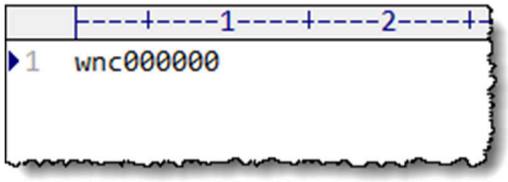


Figure (Appendix) 12 - adbkey.pub

Hint:	The `adbkey.pub` file should contain only this word. It should **not** contain either a **carriage-return** or **line-feed**.

2. Run the following ADB command to stop the adb 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.

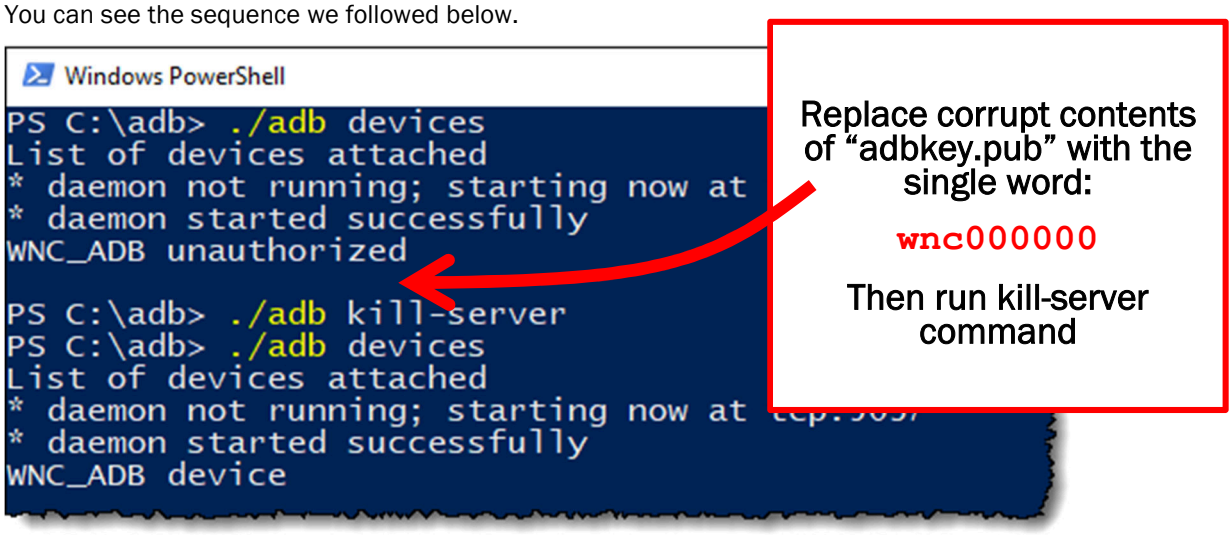


Figure (Appendix) 13 - kill-server and re-running adb devices

If the preceding three steps do not result in “WNC_ADB device”, here’s a few more things to try.

4. Try killing and starting the server several times by repeating this sequence of commands. Each time, confirm that the `adbkey.pub` file is still 9 bytes. If not, correct it as shown above. Then run the “adb devices” command again.

```
adb kill-server        (or ./adb kill-server)
adb start-server       (or ./adb start-server)
adb devices            (or ./adb devices)
```

Did it work?

Does adbkey.pub still only contain 9-bytes?

Try repeating it again.

5. If you are using Linux, try using SUDO.

   Linux users report that starting the server with sudo (i.e. as a root user) can often help to get ADB working. Make sure the ADB server is off, by using kill-server, then try starting the server with sudo.

```
adb kill-server
sudo adb start-server
adb devices
```

Did it work?

Does adbkey.pub still only contain 9-bytes?

Try repeating it again.