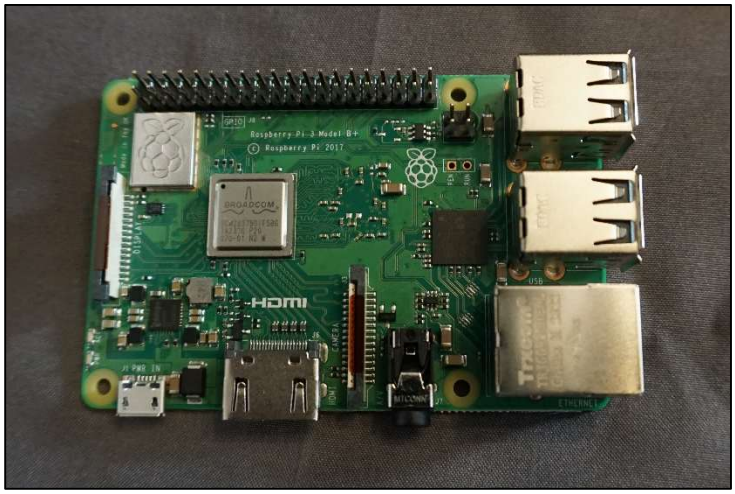
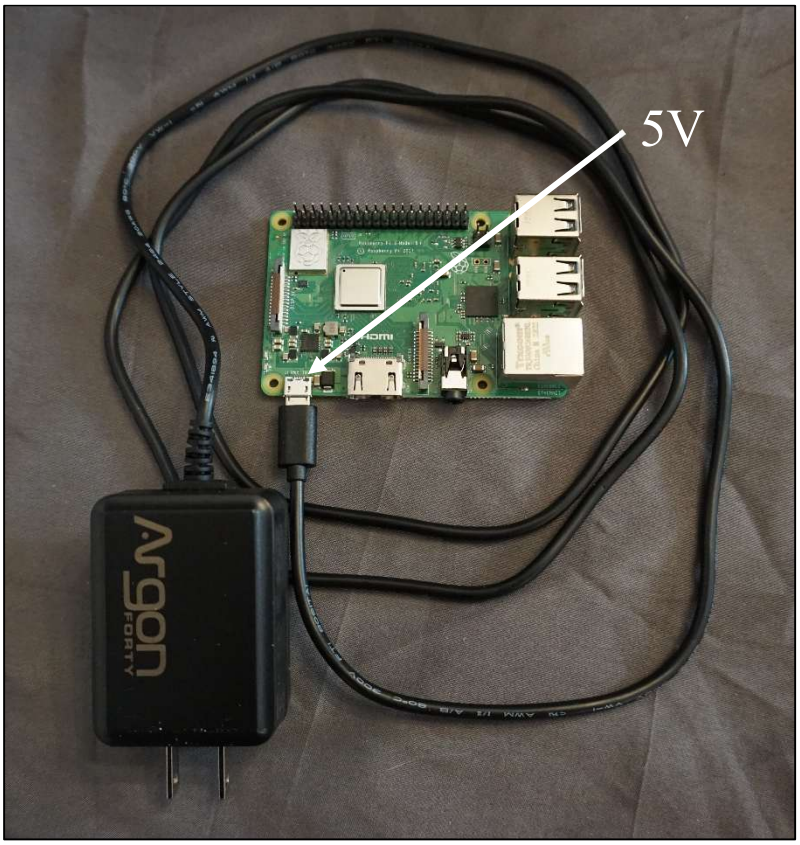
How to set up a Raspberry Pi 3B+ motherboard

I have a specific application in mind which does not require the features of the more recent (and more expensive) Raspberry Pi 4 or Pi 5. A Pi 3B+ will do. When you open the box, what you see is this.



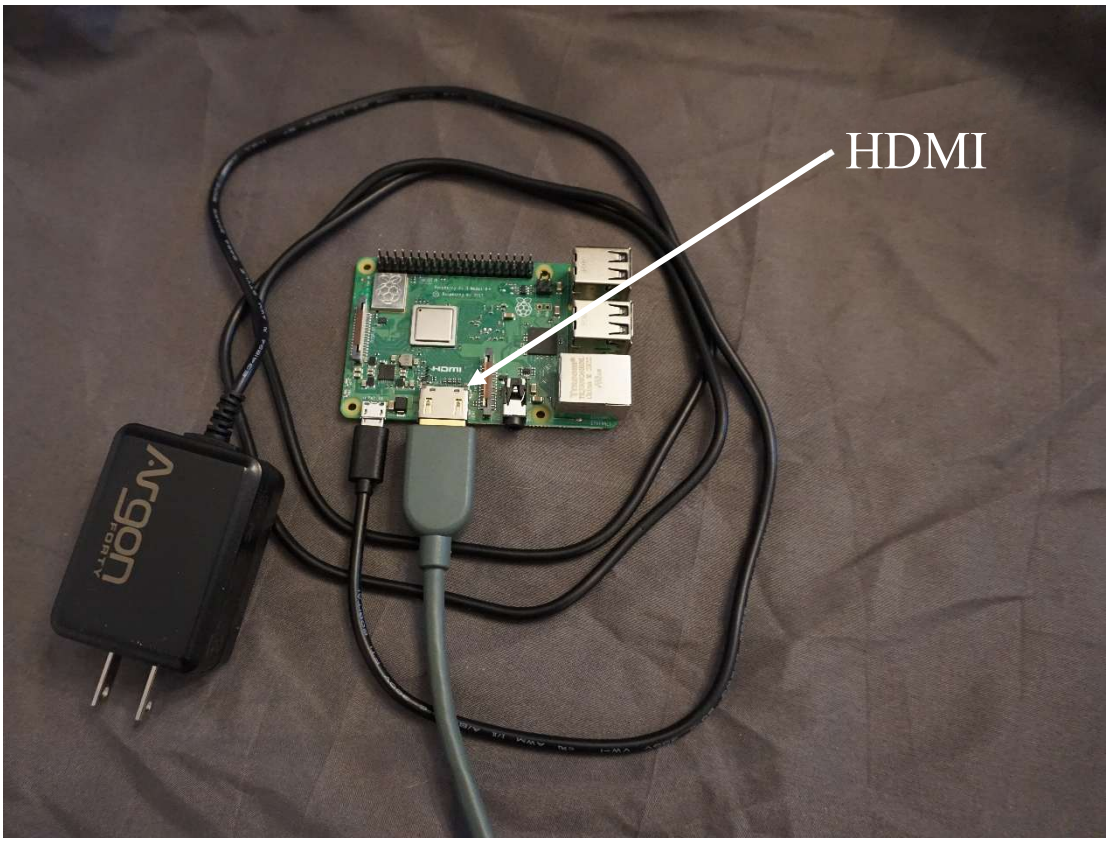
It's a small printed circuit board (PCB) with, on the bottom side, quite a nice computer chip. All of the components mounted around the perimeter of the motherboard are merely connectors allowing other devices to be hooked up to the computer.

The first thing to be hooked up is a power supply. A 5-Volt wall wart rated at 2.5 Amperes will be adequate. It is plugged into the micro-USB jack located in the lower left-hand corner of the PCB as shown in the following photograph.



Note that the motherboard does not have an on-off switch. The wall wart must be plugged and unplugged as needed to turn the device on and off or the 120VAC outlet must have its own power switch.

The next external device needed is a monitor. The motherboard talks to a monitor through an HDMI jack located along the bottom edge. I had an old computer monitor which used HDMI. Fortunately, an old signal cable with HDMI plugs on both ends had been left in place. I repurposed both of them for this project. With the HDMI cable now plugged in, the motherboard looks like this.

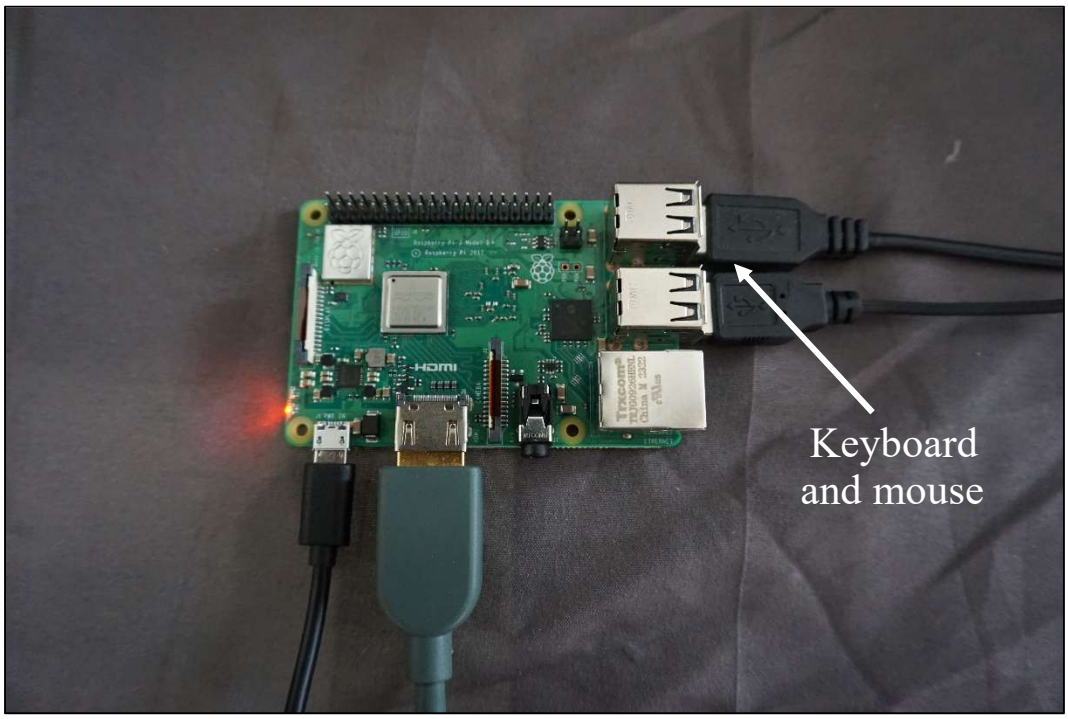


We also need a keyboard. I went overboard and also added a mouse. Although many applications – like Linux -- can be run without a mouse, I must have been a cat in a previous life and can't live without one. Fortunately, I also have some old keyboards and mice sitting around and waiting for reuse. Note, though, they can't be too old – the PS/2 plugs with round purple and green housings simply won't do. The keyboard and mouse have to have USB plugs to connect directly to the Pi motherboard.

Here, we have some choice. The Raspberry Pi 3B+ has four USB ports. They are located in the upper right-hand corner of the PCB. There are two locations with a

two-storey USB plug at each location. All four ports are USB 2.0. USB 3.0 devices will work if plugged in here, but at the slower USB 2.0 speed.

With the keyboard and mouse now plugged in, and the wall wart plugged in and supplying power, the motherboard now looks like this. Note the red LED near the bottom corner of the left-hand side. The steady red glow shows that the motherboard is receiving adequate power. Flashing red signifies a power problem.



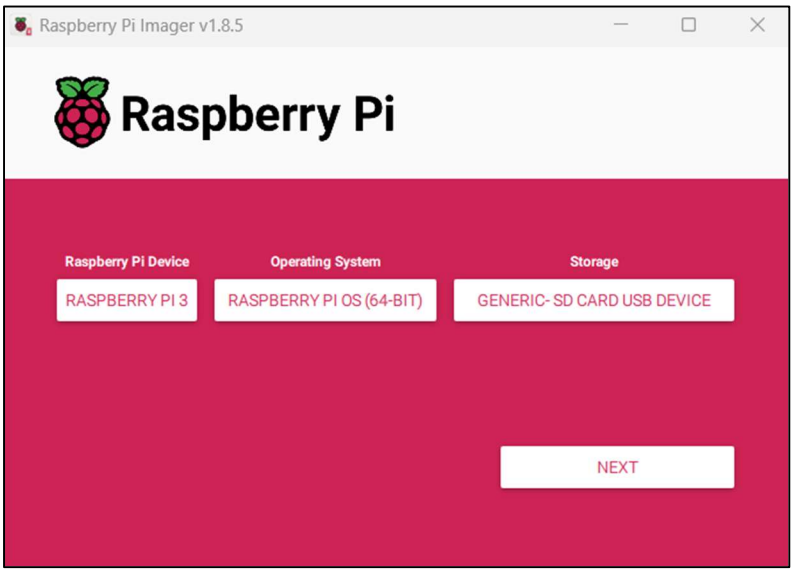
With the peripherals now attached and power applied, the monitor displays … nothing. That is because there is no program code stored anywhere on the motherboard. There is no BIOS (built-in operating system) and no POST (power-on self-test) is executed.

The Raspberry Pi 3B+ is configured so that, on first use, there is only one way to provide an operating system and that is by means of a micro-SD card. Once the motherboard is up and running, it can then be reconfigured to boot up from a USB stick, but that option is not available at first use.

So, the next step is to write a suitable operating system onto a micro-SD card. A separate functioning computer must be used to do this. Since my work-a-day laptop does not have a slot for SD cards, I had to buy an SD-to-USB memory card reader as well as a blank SD card. The reader looks like a standard USB stick but has a slot along one edge which accepts SD cards. The setup I used looks like this.

The micro-SD card plugs into the adapter, which then plugs into the reader, which in turn plugs into a USB port on the side of the laptop.

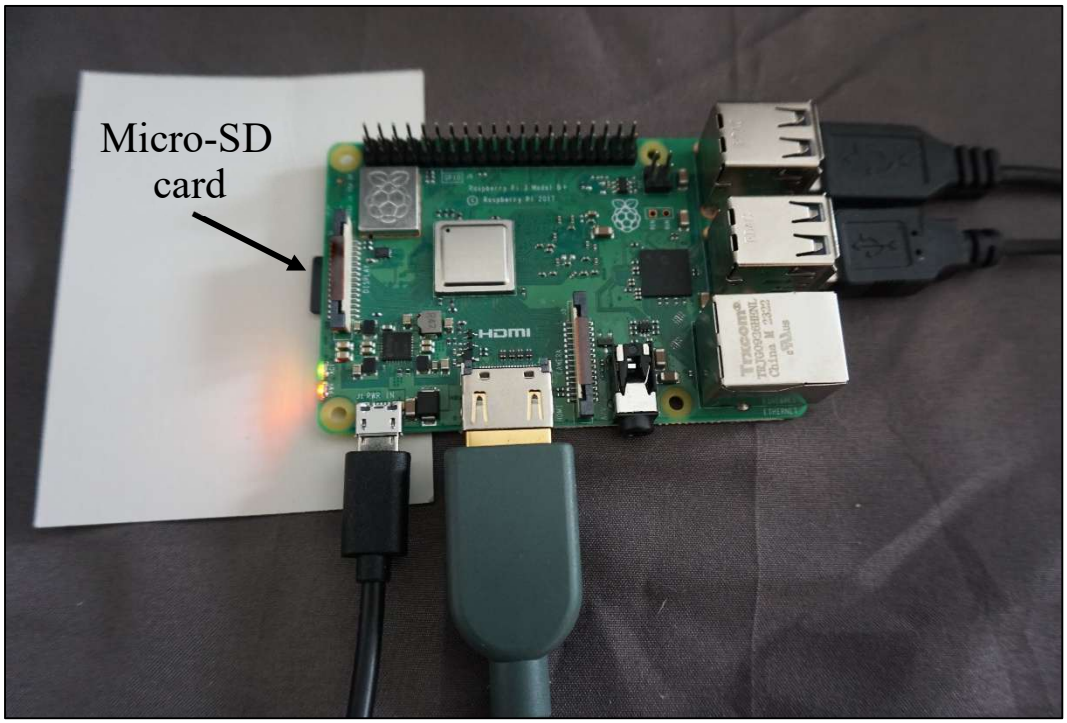
Download a program called “Raspberry Pi Imager” from the Raspberry homepage www.raspberrypi.com. “Raspberry Pi Imager” is a Windows executable program (there is a Mac equivalent, too) named “imager_1.8.5.exe”. When you run the application, it will display the following screen.



The user makes three selections: the motherboard model, the desired operating system and the destination storage device. After the selections have been made, clicking the Next button will download the operating system and write it onto the micro-SD card.

When selecting the desired operating system in the previous paragraph, three choices are offered. Two of the choices are 32-bit versions; the recommended choice is the newer 64-bit version. All three are variants of the Linux operating system known as "Debian Bookworm". Linux is ~~known~~ infamous for the constant slew of updates users must endure – Bookworm is the 12th revision of Debian, which itself is a version of Linux which "prioritizes stability and customization". In any event, the official supported operating system that has just been written onto the micro-SD card is called "Raspberry Pi OS".

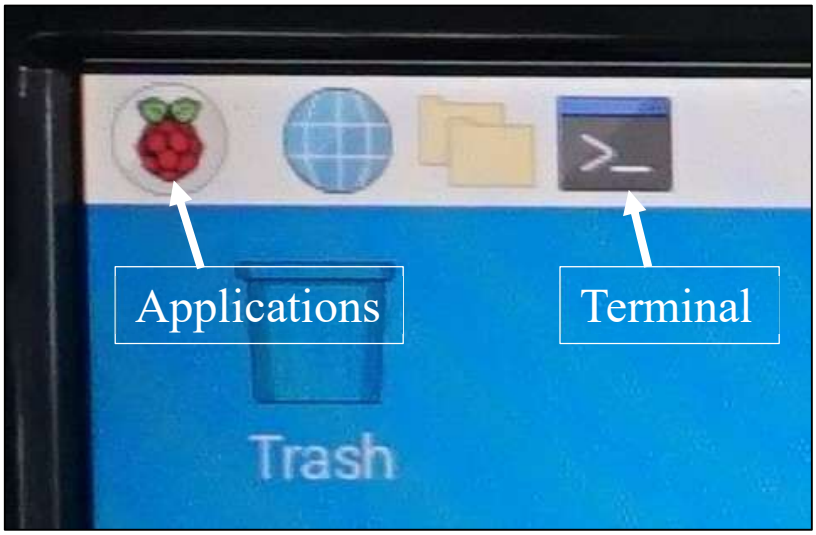
The micro-SD card is inserted into the connector mounted on the bottom side of the motherboard halfway along the left-hand edge. As always, the motherboard should be unpowered when new hardware is connected. When the wall wart is plugged back in, the Pi looks like this. The green LED next to the red one at the lower left flashes when data is being read from or written to the micro-SD card.



This time around, the system boots up properly. Since this is the first-time use, the user is prompted to enter a username and password. Then, much like a new Windows installation, the user is prompted to: (i) specify the language, country and

time zone, (ii) select from a list of detected WiFi networks, and (iii) select either Chromium or Firefox as the default web browser. When all is said and done, the monitor displays the following start-up screen.

Debian presents itself to the user much like traditional Windows, including the icons in the upper left-hand corner, which I have enlarged here.



Clicking on the raspberry icon exposes a drop-down list of the application programs and accessories available, which includes a text editor. Clicking on the terminal icon opens a terminal window with a command prompt, which is the traditional means of communicating with Linux. Users who are comfortable with Linux can take things from here.

Programming in C

Before I leave the set-up process, I want to make sure that I can use this computer to carry out some basic tasks. One such task is to run a C-program. The "Raspberry Pi OS" installed on the Raspberry Pi 3B+ motherboard includes GCC (GNU Compiler Collection), so it is possible to program in C or C++ without further ado. As always, the programming cycle consists of three steps. The desired program must be written, then compiled into an executable, and then run.

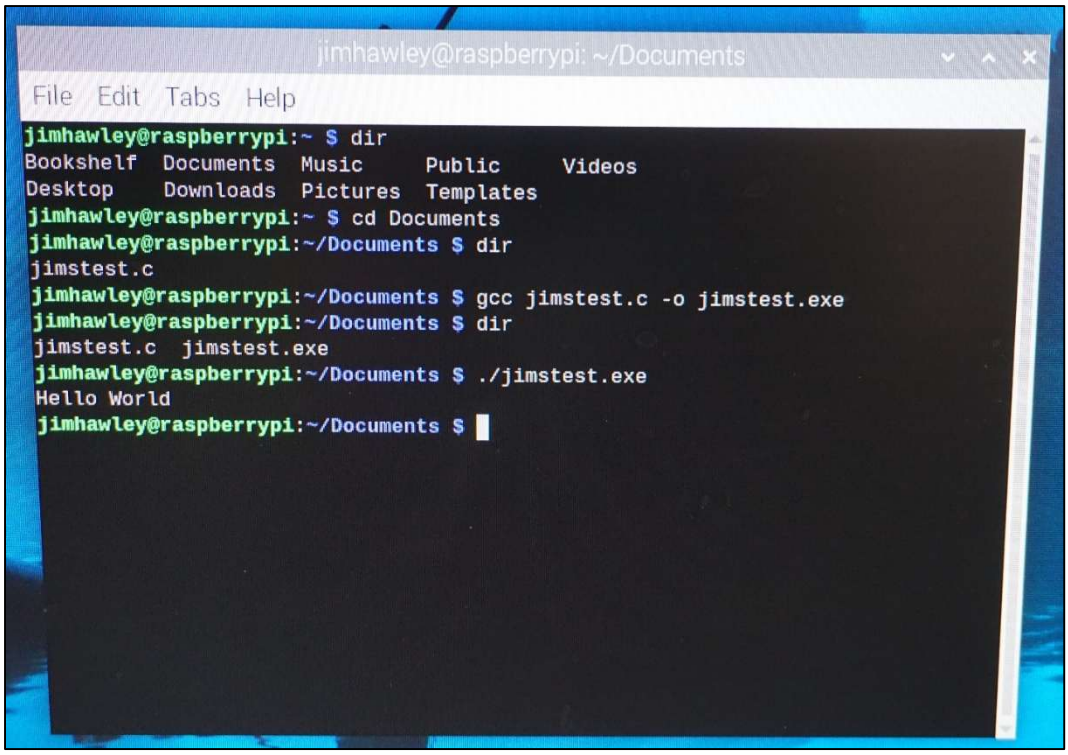
As I said above, a simple text editor can be started by clicking down through the raspberry icon. Once in the text editor, I typed the following text into a new blank document.

```
// Raspberry Pi 3B+ Output test
#include <stdio.h>
int main()
{
     printf("Hello World\n");
     return 0;
}
```

I saved this text with the filename `jimstest.c`. Note that Linux does not recognize extensions so it treats the "`.c`" as simply the last two characters of the file's name. When I saved this file, I saved it in a folder named `Documents`, which the text editor prompted as the destination directory. Note that here, and in quite a few other circumstances, Debian tries to act in the same way that Windows does.

After saving the file, I looked around to see where it had actually been saved. The exact path name is `/home/jimhawley/Documents/jimstest.c`. Just to clarify, the `jimhawley` part of the path is the username I gave Pi to use during log-in.

The next step is to compile this little program. The following screenshot shows the sequence I used to compile and then run this little "Hello World" program. Clicking on the Terminal icon, as I described earlier, opens a terminal window with the command prompt `jimhawley@raspberrypi:~ $`.



The first `dir` command lists all folders and files in the directory, which includes the `Documents` directory I mentioned above. The `cd` command changes the active directory to the one named `Documents`. A second `dir` command lists all folders and files in that directory. At this time, it contains only the `jimstest.c` file I created using the text editor.

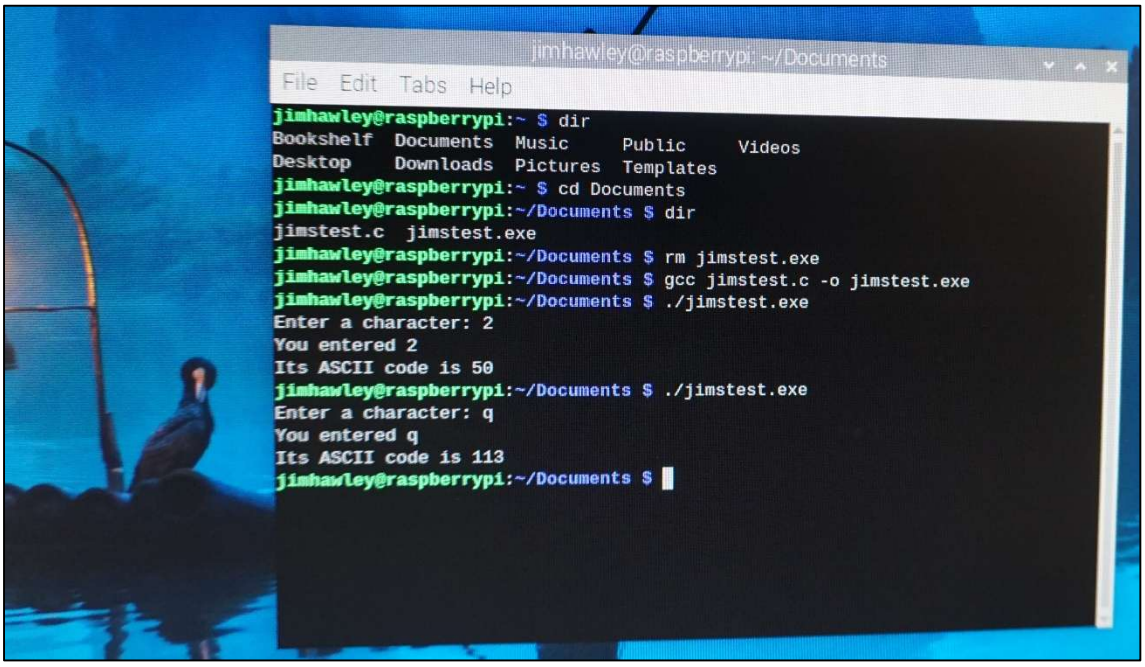
The command `gcc jimstest.c -o jimstest.exe` compiles the C-program and saves the compiler's output in the file named `jimstest.exe`. Once again, the `.exe` extension is solely for my benefit, not Debian's. A third `dir` command confirms that the `Documents` directory now includes the new executable file. It is run using the command `./jimstest.exe`.

Running the program prints the words Hello World on the screen. Note that the suffix `\n` in the C-program is the code for carriage-return line-feed, which causes the cursor to jump to the start of the next line on the screen.

That's good. We now have a way to output information. Now let's revise the program to demonstrate how to read characters typed on the keyboard. The next program prompts the user to type in a character and then displays the character and its ASCII number on the monitor.

```
// Raspberry Pi 3B+ Input test
#include <stdio.h>
int main()
{
     // Define a variable for the character
     char MyChar;
     // Prompt and read the keyboard
     printf("Enter a character: ");
     MyChar = getchar();
     // Print the results
     printf("You entered %c\n", MyChar);
     printf("Its ASCII code is %d\n", MyChar");
     return 0;
}
```

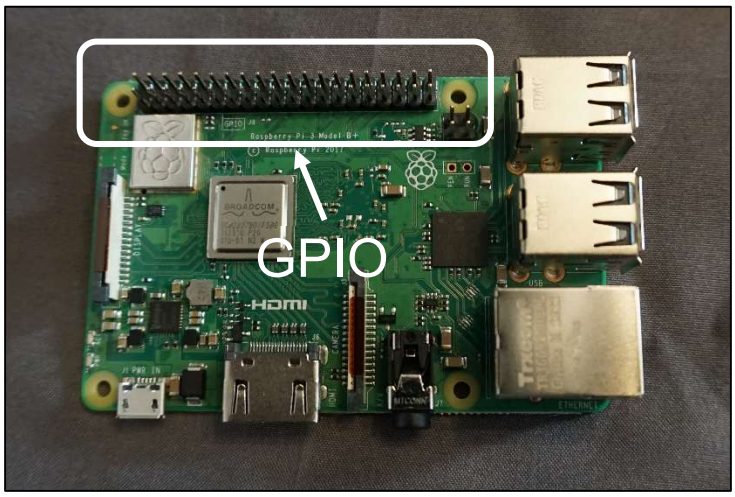
The sequence of commands I entered into a new terminal window are shown here.



I used the `rm jimstest.exe` command to delete the earlier Hello World executable. I then compiled the new `jimstest.c` program using the `gcc` command. I ran the new program twice. The first time, I entered number "2" on the keyboard. Its ASCII code (in base ten) is 50. The second time through, I entered a small letter "q", whose ASCII code (in base ten) is 113.

Toggling an LED using the keyboard

Good – we now have some basic I/O skills. Another basic task is hardware control. The motherboard has a physical interface we can use to control external hardware. It consists of a set of pins called the GPIO (general purpose input-output) located along the top edge of the PCB. We have already seen this connector in some of the previous photographs.

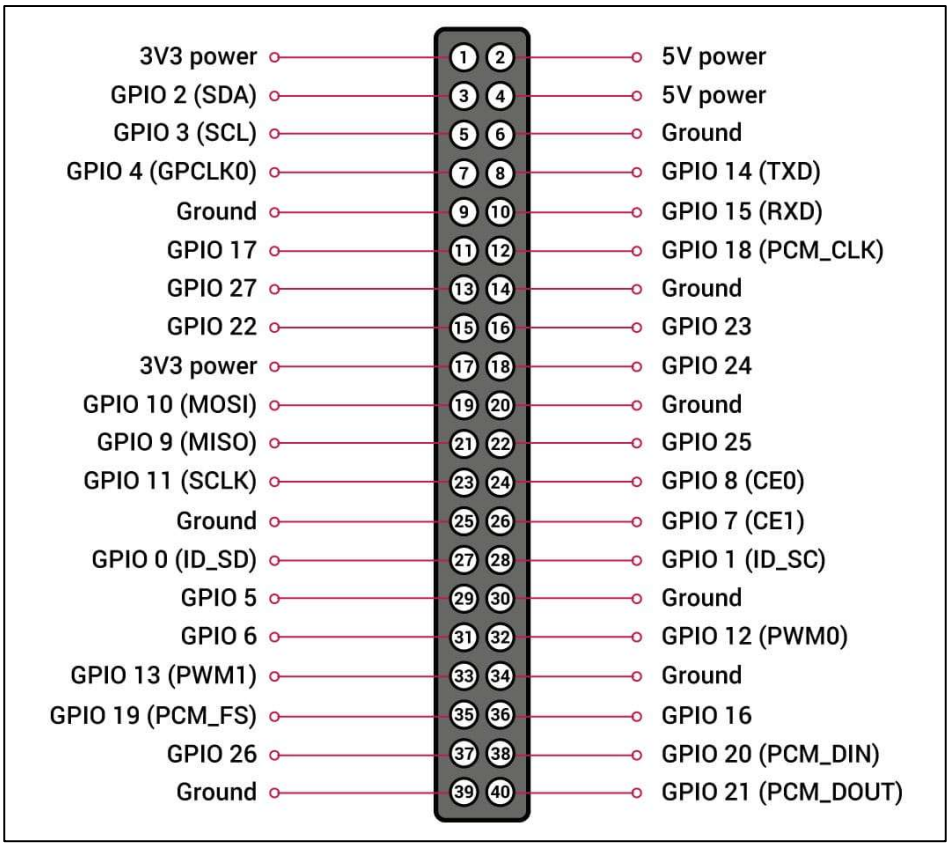


The are two rows with 20 pins in each row. Technically, this connector is called a J8 header strip. The diagram at the top of the next page is called a "pinout" – it lists the name and purpose of each pin. Examples include:

Pin #2 is described as "5.0V Power". It is wired directly to the positive side of the motherboard's 5.0V power supply. Similarly, pin #6 is described as "Ground" and is wired directly to the negative side of the power supply. When the Pi is up and running, an external device can be powered by attaching its power leads to these two pins. For convenience when wiring up multiple external devices, several different pins carry 5.0V and Ground.

Pins #8 (TXD) and #10 (RXD) are the transmit and receive lines used for RS232 communication. For the user's convenience, the Pi has a built-in RS232 module which uses these two pins. Similarly, pin #3 (SDA) and #5 (SCL) are the serial data and serial clock lines for I2C communication. I2C stands for "inter-integrated circuit") and is a common protocol used to allow one integrated circuit chip (IC) to talk directly to another.

Pins #32 (PWM0) and #33 (PWM1) are the signal outputs from the two separate pulse-width modulation units which are built in to the Pi.



Many of the pins are named “GPIO xx”. They are general-purpose lines which can be individually controlled by the motherboard’s computer. Some of them, like the TXD and RXD pins, are dual-purpose. If the user’s program is running an RS232 application, then they will be used for that. But, if the user is not running an RS232 application, then these pins are available for any other purpose.

To use a GPIO pin for hardware control, it must be “configured”. The Pi needs to know whether the pin will be used for “output”, where the voltage the Pi places on the pin will be read by the external hardware, or whether the pin will be used for “input”, where it is the external hardware that controls the voltage on the pin.

Configurating a pin also means setting its electrical characteristics. For example, a pin to be used for output can be configured as “push-pull” or as “open collector”. In the former configuration, the output circuit of the pin is able to source and sink current. That means the voltage on the pin is set high (5V) or held low (0V). In the latter configuration, the output “signals” consist of either: (i) pulling the pin’s voltage down to ground, or (ii) disconnecting the pin in such a way that its voltage floats up or down to a level determined by the external hardware.

To avoid confusion, note that the numbers in the logical names of the “GPIO xx” pins are not the same as their physical locations (1-40) on the J8 header.

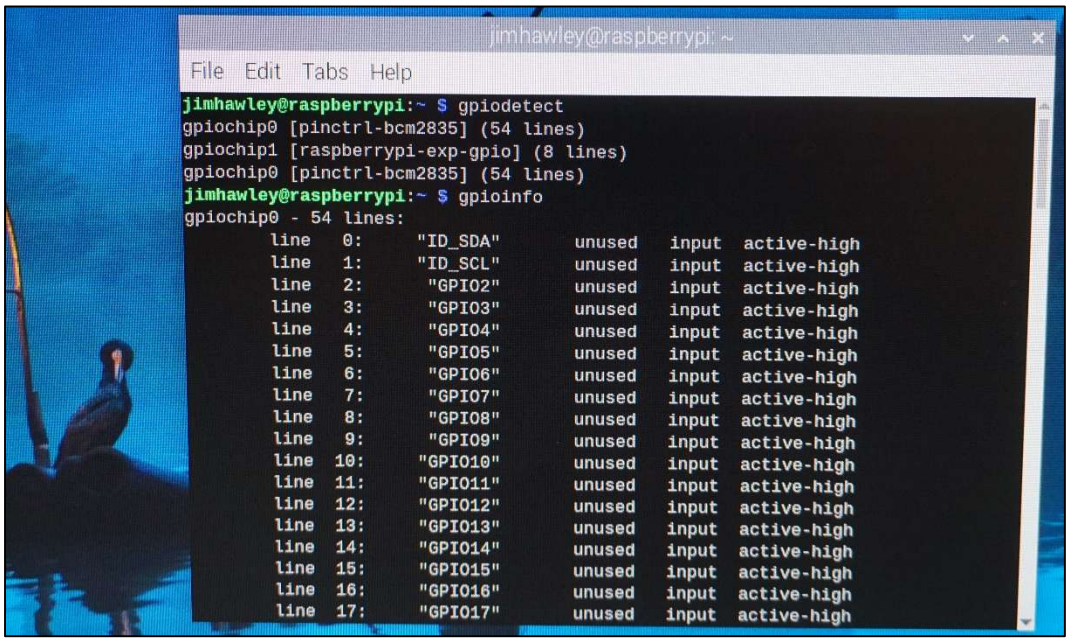
It turns out that the operating system we have downloaded and are using does not contain the functions and programs which are needed to control the GPIO pins. The library which contains them must be installed explicitly. Typing the following command into a new terminal window will download and install the `gpiod` developers’ library.

```
sudo apt install libgpiod-dev
```

To use this library, we need to know the name of the J8 connector. We get this by typing in the following two commands into a terminal window.

```
gpiodetect
gpioinfo
```

The first command lists all the GPIO devices which can be found. The second command prints out the current configuration of the pins on each device. The following photograph shows the first page of the results.

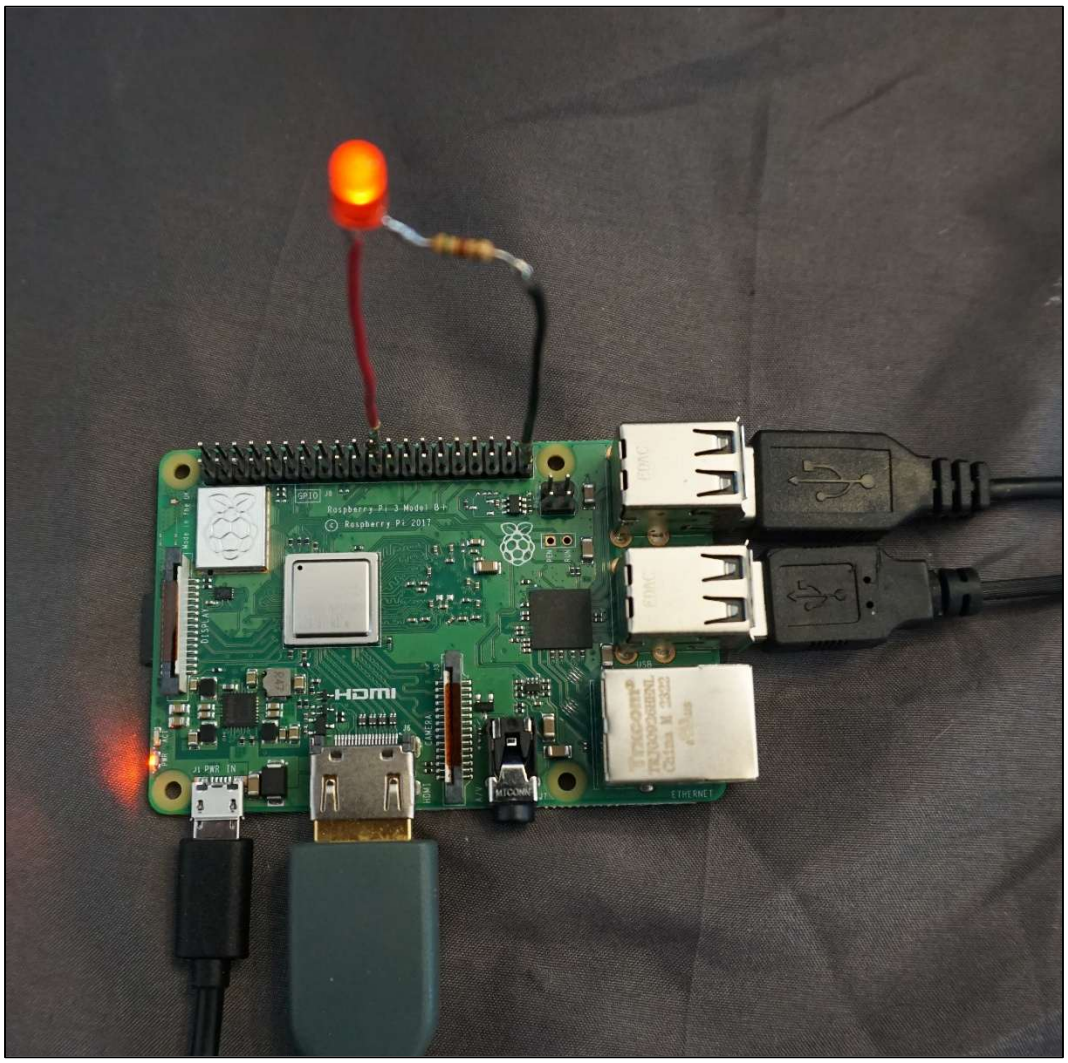


Among other things, we now know that our J8 has the logical name `gpiochip0`.

Oops, I forgot something. We need some kind of “external hardware” if we are to show our prowess. I soldered a 510Ω resistor in series with a standard red LED. It

will glow nicely when 5V is applied (with the right polarity) over them. I also soldered individual crimp header connectors onto the tag ends which slip easily onto the pins of the J8.

I attached the positive wire of the LED-plus-resistor to physical pin #22 of the J8 and the negative wire to physical pin #39 (Ground). The following photograph shows the setup.



The LED is glowing because I have already turned it on. The following two terminal commands turn the LED on and off, respectively.

```
gpioset gpiochip0 25=1
gpioset gpiochip0 25=0
```

Note that physical pin #22 has the logical name `GPIO 25` and that, as always, "1" is High (5V) and "0" is Low (Ground).

These two `gpioset` commands look deceptively simple. Using the default settings, both commands operate as I've just described. But there are many options – none of which were needed here -- to change the settings and configuration.

Controlling hardware in a C-program

It is important that we can control hardware, not just from the keyboard, but from with a program. I used the following little program.

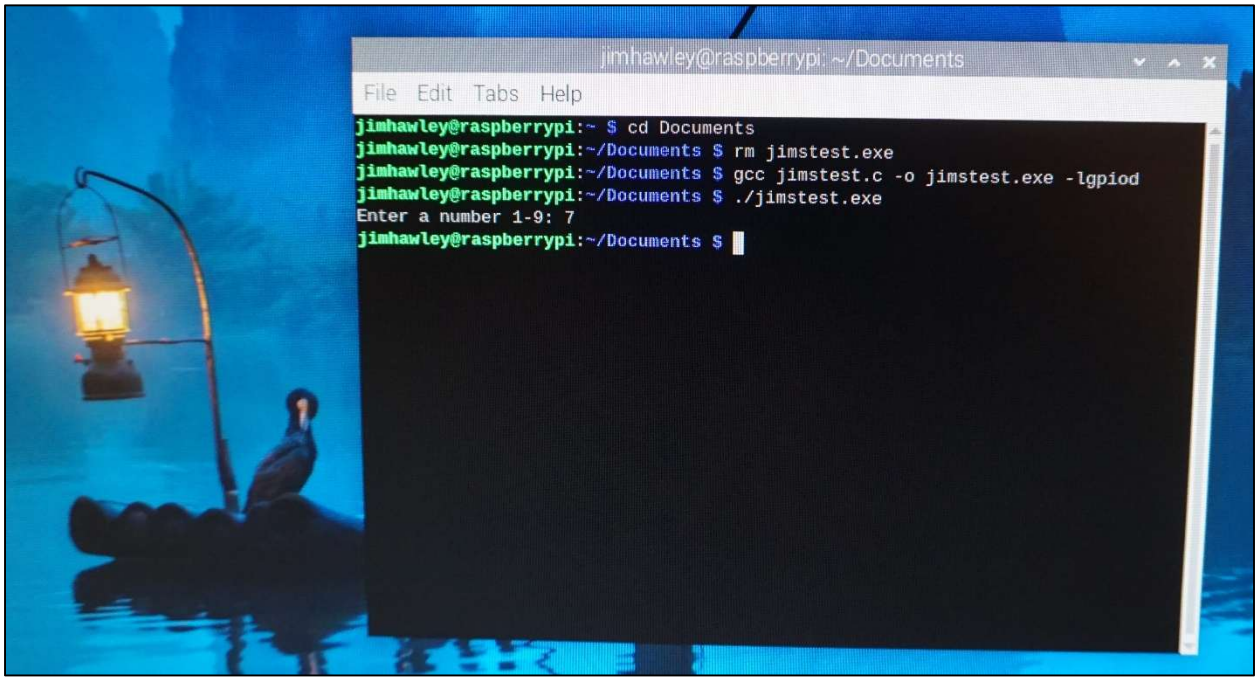
```
// Raspberry Pi 3B+ Hardware test
#include <gpiod.h>
#include <stdio.h>
#include <time.h>
int main()
{
    // Declare the GPIO chip
    MyChip = gpiod_chip_open_by_name("gpiochip0");
    // Declare the line to physical pin #22
    MyLED = gpiod_chip_get_line(MyChip, 25);
    // Open the line for output
    gpiod_line_request_output(MyLED, "dummy", 0);
    // Prompt and read the keyboard
    printf("Enter a number 1-9: ");
    int MyCounter = getchar() - '0';
    // Define a specification for one-half second
    struct timespec MyDuration = { 0 };
    MyDuration.tv_nsec = 500000000;
    // Blink the LED MyCounter times
    for (int I=1; I<=MyCounter; I=I+1)
    {
        // Turn the LED on for one-half second
        gpiod_line_set_value(MyLED, 1);
        nanosleep(&MyDuration, NULL);
        // Turn the LED off for one-half second
        gpiod_line_set_value(MyLED, 0);
        nanosleep(&MyDuration, NULL);
    }
```

```
        // Release the line and the chip
        gpiod_line_release(MyLED);
        gpiod_chip_close(MyChip);
        release 0;
    }
```

There are only a couple of things which aren't pretty straightforward once one has figures out some of the syntax tricks. One is the need for a name – I used the name "dummy" – when opening a pin. The manual suggests that this entity is the "customer" who somehow now owns this pin. Why is that necessary? Dunno.

The second thing is all the business of defining a time specification in order to get a one-second delay. In the old days, one could simply call the function `msleep(500)` which would cause execution to pause for 500 milliseconds or the function `usleep(500000)` which would wait for 500,000 microseconds (also equal to one-half second). But those handy functions have been deprecated and replaced by a nanosecond timer. (Of course, the Pi's motherboard can't begin to keep track of time on a nanosecond basis, but who's counting?)

The following screenshot shows how this program is compiled and run.



The compilation command must be altered a little bit in order to ensure that the compiler knows that it will need to refer to the `gpiod` library. That requires adding the `-l` option to the end of the command line. The full command is:

```
gcc jimstest.c -o jimstest.exe -lgpiod
```

When the program is run, it prompts the user to enter an integer in the range 1-9. The program then flashes the LED that number of times at a frequency on one Hertz. (The program does not do any error checking, so I don't know what might happen if a character was entered or a non-alphanumeric key was pressed.)

Remote control of the Raspberry Pi

The internet tells me there are three ways to control a Raspberry Pi from a remote computer. Running in "headless mode" like that can be convenient since the Pi need not be connected to a monitor, keyboard or mouse. The internet tells me there are three ways to do this: (i) "Raspberry Pi Connect", (ii) "Virtual Network Computing" (VNC) and (iii) "Secure Shell" (SSH).

I learned right away that "Raspberry Pi Connect" only runs on version Pi 4 and later.

Previous versions of Pi, including 3B+, come with "RealVNC" already installed. That sure is handy but – most recent news – "RealVNC" is now a paid subscription. There is a free VNC called "TightVNC" but it needs Linux on both ends, on the Pi and on the controlling PC.

So, we come to "Secure Shell". The Pi 3B+ comes with it already installed but not enabled. Enabling it is not too difficult. Drill down through the Raspberry icon as follows.

```
Raspberry icon > Preferences > Configuration > Interfaces
```

Once there, slide the SSH switch and click OK. We're almost there. We want remote control to be possible as soon as the Pi boots up. That requires that the `ssh` executable be mentioned somewhere in the `boot` folder, a glitch which is dealt with by typing

```
sudo touch /boot/ssh
```

in a terminal window. That completes the installation of the Secure Shell server on the Pi. Next, we need to install the Secure Shell viewer on a PC. I did a lot of browsing, twisting, turning and tinkering, so much so that I cannot recall the many and varied things I tried. Eventually, I arrived in paradise and found I could open

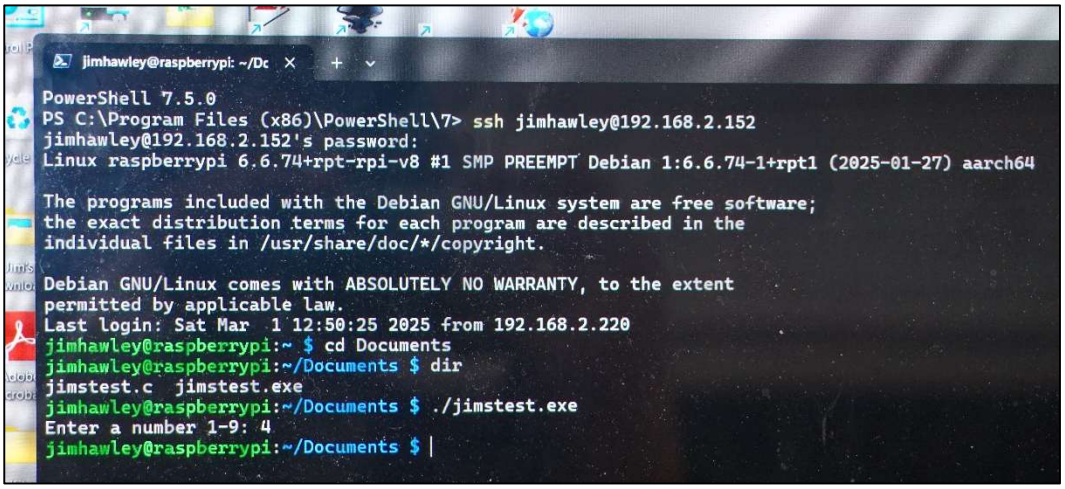
an “elevated Power Shell” on my laptop. More labour followed, and then joy when a folder named `C:\Windows\System32\OpenShell` appeared. Inside that folder was the magic executable `ssh.exe`.

The following photograph shows the screen of my laptop PC. A terminal window is opened using the “elevated Power Shell”. Typing the following command starts the session:

```
ssh jimhawley@192.168.2.152
```

`jimhawley` is, of course, the name I gave the Pi when I first set it up. The IP address is the Pi’s address on my home WiFi network. The password is required in the next line. After a short header, the following prompt appears:

```
jimhawley@raspberrypi:~ $
```



From this point onwards, the laptop PC is in control of a terminal window on the Pi. The photograph shows that I ran my LED blinking C-program. I can confirm that the LED did flash four times, as expected.

Well, that’s enough familiarization for now. I have some idea of what the Raspberry Pi is and what it can do. Next step: try to set up an ADS-B aircraft receiver so that the Pi can be run as a base station in the FlightRadar24 network.

Jim Hawley
February 2025