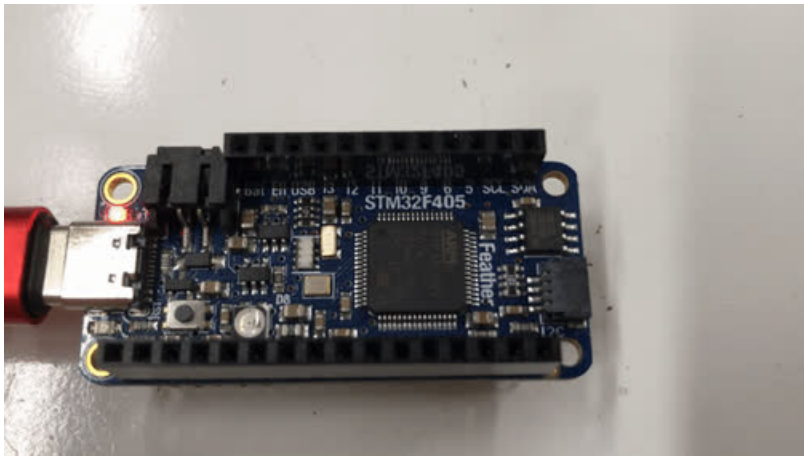
adafruit learning system

Blinking an LED with the Zephyr RTOS

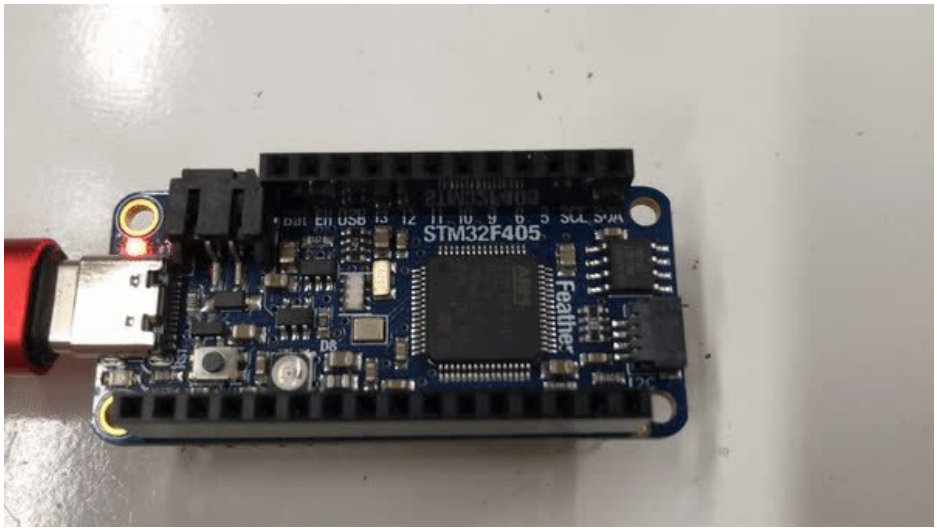
Created by Lucian Copeland



Last updated on 2020-05-07 01:46:21 PM EDT

Overview

There are lots of ways to make programming microcontrollers really easy - CircuitPython, MicroPython, and Arduino are all options to get your project up and running, even as a beginner programmer. But sometimes you don't just need easy - you need *beefy*. When it's time to break out the big guns, you might consider using an RTOS - a Real Time Operating System, sort of a very tiny version of what runs on your desktop or laptop computer, but one that's built for single-chip microcontrollers like those on an Arduino or Feather board.



An RTOS is built to handle chips with lots of features automatically, juggling sensors, buses, screens and buttons without huge messes of custom code to manage them all. Unlike Arduino's `startup` / `loop`, or Circuitpython's `while True:`, an RTOS can run many different operations (called Tasks) in *parallel*, never allowing any one task to fall too far behind. This means an RTOS is great for big, sprawling projects that have a lot of things running at once, or for projects like sensor data collection where one task is so critical that it needs to be constantly serviced.

However, all this capability comes with a cost - RTOSes can be big and complex, since they're usually marketed toward corporate teams or very experienced freelancers. But they don't have to be hard to learn! In this guide, we'll be sticking to the basics - getting an LED up and running in an up-and-coming RTOS, Zephyr, which has been backed by the Linux Foundation, Intel, NXP, and many other powerful microcontroller companies.

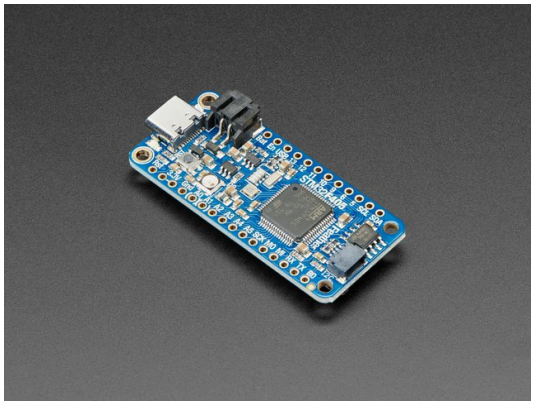
You'll learn how to:

- Install the Zephyr core on Mac OSX or Linux computers
- Install Zephyr's custom management tool, West

- Test your setup with the RTOS's built-in sample projects
- Create your own application folder
- Blink an LED on the Feather STM32F405 Express
- Start learning RTOS concepts for custom projects

Parts

The heart of this project is the Feather STM32F405 Express:



Adafruit Feather STM32F405 Express

OUT OF STOCK

Out Of Stock

Depending on whether you want to work exclusively off of USB, or use a JLink programmer, you may also need one or more of the following parts:

1 x USB C to USB C Cable

A USB C Cable for the Feather STM32F405 for computers with USB C / Thunderbolt ports

Out Of Stock

1 x USB Type A to Type C Cable - approx 1 meter

A USB C Cable for the Feather STM32F405 for computers with traditional USB Type A ports

Add To Cart

1 x SWD 0.05" Pitch Connector - 10 Pin SMT Box Header

A solderable STM box header, required for Jlink programming

Add To Cart

1 x SEGGER J-Link EDU - JTAG/SWD Debugger

The Educational Edition Jlink - use this for hobby or student work

Out Of Stock

1 x SEGGER J-Link BASE - JTAG/SWD Debugger

The professionally licensed Jlink - use this if you intend to do professional work

Out Of Stock

1 x JTAG (2x10 2.54mm) to SWD (2x5 1.27mm) Cable Adapter Board

Jlink SWD adapter board, required for programming the Feather with Jlink

Out Of Stock

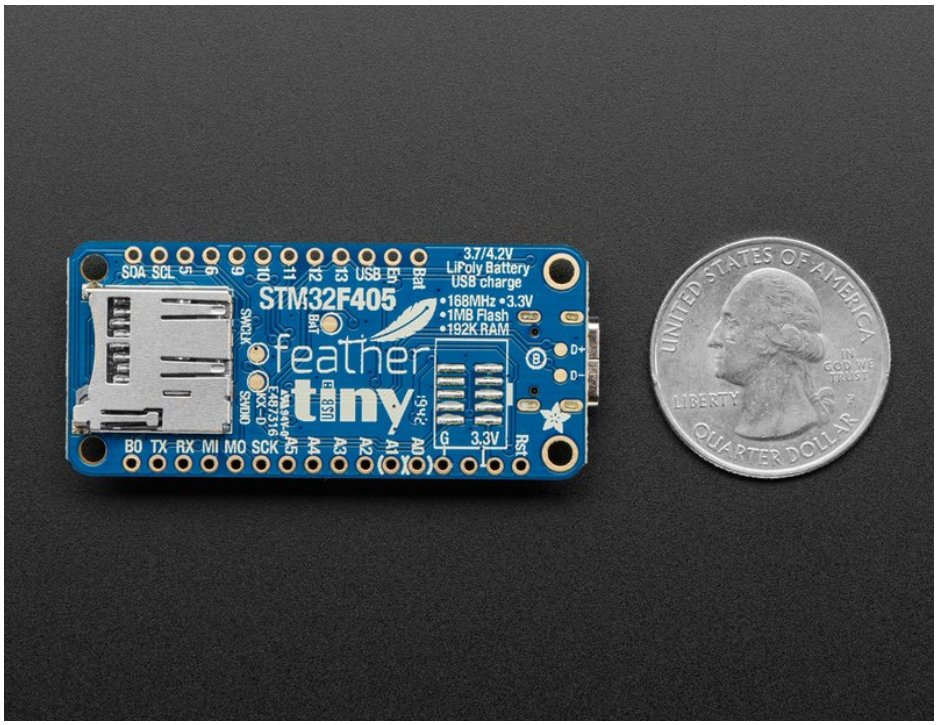
1 x 10-pin 2x5 Socket-Socket 1.27mm IDC (SWD) Cable - 150mm long

Jlink SWD adapter cable, required for programming the Feather with Jlink

Add To Cart

Hardware Setup

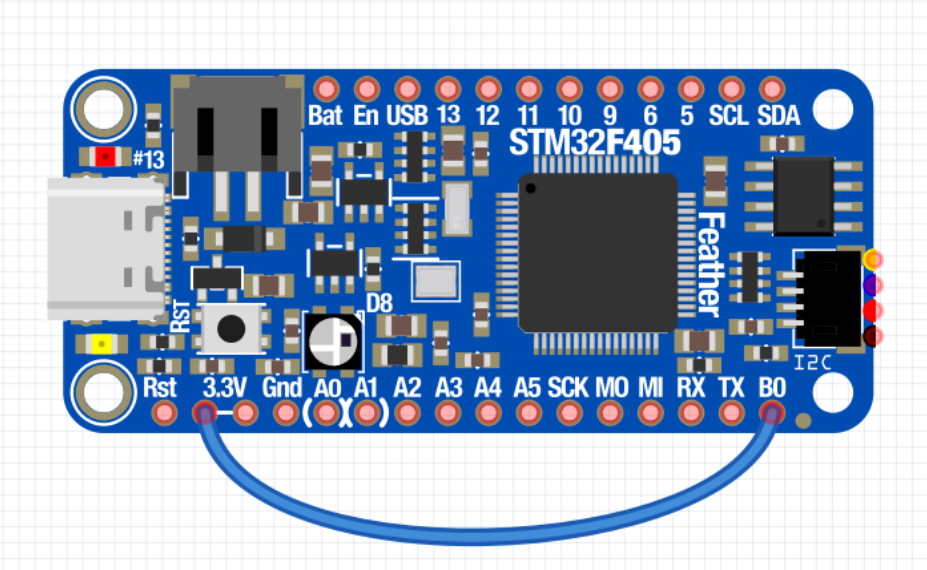
For this project, you won't be making use of any external sensors, so the hardware setup is relatively simple. However, it may be worth going through the primary guide for the STM32F405 Feather Express (https://adafru.it/Jdu) to make sure you've considered what headers to use, what pins are available, etc. etc.



Once you've set up your STM32 Feather the way you want, you'll need to choose which of two programming options you'd like to use - the built in **DFU-bootloader**, or an external **JLink Programmer**. The DFU bootloader is built directly into the Feather, and allows you to skip purchasing a new programmer if you don't have one. The JLink is more expensive, but faster, and it'll allow you to do more advanced debugging if you need to. Both are supported by the Zephyr West tool.

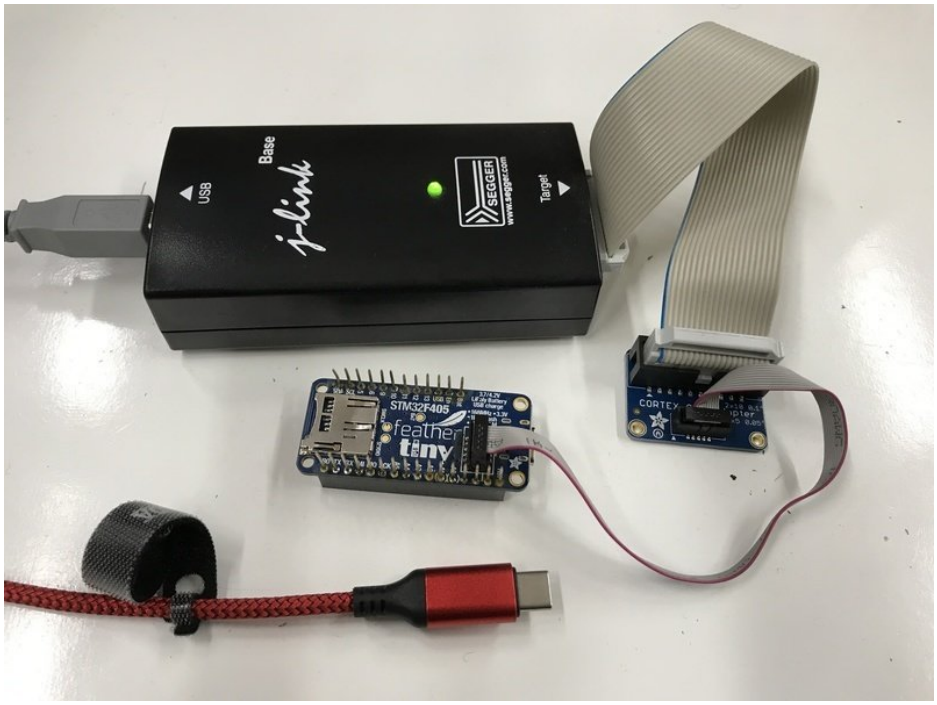
Cheap and easy (DFU Bootloader)

Out of the box, you have the DFU (Device Firmware Upgrade), a programming option over USB that's available by default on all STM32F4 chips. Enabling it is very easy - just connect the Boot0 pin (B0) to 3.3V logic, which you can do by simply connecting a wire from the B0 pin to the 3.3V pin on your feather. Once you do, either power cycling or resetting the STM32 Feather and connecting to USB will put it in bootloader mode, and you can remove the jumper.



Fancy (Jlink)

The Jlink is a great choice if you want to do something like debugging with GDB later. It also doesn't require any added wires, and it's much faster, but requires the purchase of the Jlink Programmer.



To get the Jlink working with the Feather F405 you'll need to first solder on a SWD header on the bottom of the board. Make sure the gap in the SWD header plastic faces the end with the USB C connector.

Then, you can plug in your Jlink using the SWD adapter. Make sure you follow the Jlink instructions to install Jlink Server if you want to use debugging capabilities like GDB later.

Ready to Program!

Once you have one of your two programming options set up, your chip is now ready to be flashed - but you've got nothing to put on it yet! We'll revisit programming the hardware once you've installed all the prerequisite tools and compiled your project.

Installing Zephyr (OSX)

There are three steps to creating a Zephyr RTOS project on your chosen microcontroller:

1. Setting up your development environment (installing prerequisite programs, obtaining Python scripts, setting environment variables, etc)
2. Cloning the Zephyr RTOS source code with the Zephyr multi-purpose tool, West.
3. Creating your own application linked to the Zephyr source, which you can compile and upload to your board.

This page will focus on installing all of the scripts and prerequisites you need, along with some other setup tasks that are usually specific to your host computer. This section is broadly similar (with some added suggestions) to the official Zephyr startup guide (https://adafru.it/Jdv), which you can also check out for more information.

Installing Dependencies

The first thing you'll want to do before installing anything is to ensure your system is up to date: for Mac OSX, this is as simple as navigating to System Preferences from the Apple menu and selecting Software Update. This guide was created using Catalina 10.15.2, though most version differences should hopefully be handled by the package managers you'll be using.

Once you're up to date, you can use Homebrew (https://adafru.it/wPC) to install and manage all the command line software you'll use. If you don't have Homebrew, you can get it by opening your Terminal in your Applications folder and running the following:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once you have Homebrew, you can install software by using the `brew` command. Install the following Zephyr prerequisites by running:

```
brew install cmake ninja gperf ccache dfu-util qemu dtc python3
```

When installing like this, it's possible you may run into some things you've installed already, which is usually ok. If you need to double check the version of a program, you can often do it by running the `--version` option after it, such as `python3 --version`. You can also see all the programs you have installed by running `brew list`.

Python Virtual Environments and West

Zephyr also has a lot of Python3 dependencies you will need to install using the pip3 package manager. However, I'm going to diverge from the official guide here and *strongly* suggest you use a **Python virtual environment** before doing any installations with pip. Often, when doing the installations for different projects, you'll find that one project has a conflicting version requirement with another - a virtual environment system allows you to set up a fresh "virtual" Python installation any time you want, letting you install the right versions of all your prerequisites specifically for that project. It's a good idea for any developer, and is particularly useful for projects with a huge list of Python requirements like Zephyr.

There are a number of virtual environment managers, but I recommend `virtualenv` and `virtualenvwrapper` for simplicity. Install them now by running:

```
sudo pip3 install virtualenv virtualenvwrapper
```

You then need to add the following text to your `~/.bash_profile` on MacOSX. If it doesn't exist already, create it with `sudo nano ~/.bash_profile` and paste in the following after any existing contents:

```
# virtualenv and virtualenvwrapper
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3
source /usr/local/bin/virtualenvwrapper.sh
```

Then, run:

```
source ~/.bash_profile
```

This should display some setup information. If you run into any errors, make sure the location of your python executable is correct by using `which python3`, which will show you the path to your default python. If it's different from `/usr/local/bin/python3`, change that section to whatever `which python3` showed you. It's also possible that `virtualenvwrapper` might need it's location changed as well if the python3 location is different - it'll usually be in the same `*bin/` location as python3.

Note that the location of your python3 installation can be VERY different than what's listed here. This isn't a big deal! There are a lot of ways python3 can get installed, including Homebrew, pre-included mac OSX versions, and versions installed by IDEs and other applications, and they can all put the link in different spots. Your personal bash settings, if you have any, may also impact the location of programs. But as long as you correctly change the environment variables, the specific location shouldn't matter.

Now, you should have `virtualenv` installed! Some of the commands available to you are:

- Create with `mkvirtualenv`
- Activate with `workon`
- Deactivate with `deactivate`
- Remove with `rmvirtualenv`

For now, make a new virtual environment for your Zephyr installs and move to it by using:

```
mkvirtualenv zephyenv
workon zephyenv
```

You're now in a pristine new virtual environment for Python! Anytime you log on or restart your computer, you can return to it by running `workon zephyenv` again. It *only* has Python3 installed, so you don't even have to worry about that pesky default 2.7 install on MacOSX anymore, you can use `python` and `pip` directly and it'll use the Python 3 versions automatically.

Now that that's over with, you can install the Zepyhr multi-purpose tool, West, which will help you download the Zephyr RTOS code and wrap up all your other Python requirements. Get it by running:

```
pip install west
```

Getting the source code and adding packages

Now that you have West installed, you'll be using it to download the Zephyr RTOS source code into a new directory in your home folder called zephyrproject/:

```
cd ~
west init zephyrproject
cd zephyrproject
west update
```

Note that the zephyrproject/ folder is NOT your application folder - it isn't where you're going to put main.c or any of the specific libraries for your application. It's the Zephyr source code, which we'll be linking into new, totally separate application folders that we make later. ZephyrProject refers to the The Zephyr Project™, not your project!

Make sure you give yourself some time to do this. `west update` clones all of the various software libraries and repositories for a big collection of chips - depending on your internet speed, it could take up to an hour.

Once that's done, you'll install the latest python requirements for Zephyr (make sure you're working on the virtualenv you set up before!). From inside the `zephyrproject/` directory, run:

```
pip install -r zephyr/scripts/requirements.txt
```

One of the python prerequisites, hidapi, requires XCode to be installed on Mac OSX. Sadly, it's not optional. You can install Xcode on the Apple Store for free - note that it also can take quite a while to download.

This should show a new, long list of Python modules being installed on your virtual environment.

Installing the Development Toolchain

Now we need the proper build tools! On Linux, this is handled automatically via the Zephyr SDK, but that isn't available on Mac yet, so there's some extra steps. First, get the latest version of the ARM Toolchain, which allows you to compile and link applications like those made in Zephyr. You can download it with Brew by using the following (note the `cask`):

```
brew cask install gcc-arm-embedded
```

You'll now have a host of ARM Toolchain tools installed, which all start with the prefix `arm-none-eabi-*`. Before Zephyr can use them, however, you need to add a few more lines to the end of our `~/.bash_profile` file:

```
#zephyr build vars
export ZEPHYR_TOOLCHAIN_VARIANT=gnuarmemb
export GNUARMEMB_TOOLCHAIN_PATH=/usr/local/
```

Similarly to the virtual environment setup, you can double check that the `GNUARMEMB_TOOLCHAIN_PATH` is correct by running `which arm-none-eabi-gdb`. **Just like `python3`, the location of this can vary based on your system settings** - if it shows somewhere other than `/usr/local/`, make sure to use the new location. When you're done, run:

```
source ~/.bash_profile
```

You can now double check that the environmental variables were set properly by running:

```
echo $ZEPHYR_TOOLCHAIN_VARIANT
```

Once those variables match up, you're all done with the installation process! It's always a lot of work getting a multi-stage development environment like this running, but thankfully you usually only have to do it once. To test your new Zephyr setup, you can use the next section: building a sample program directly out of the Zephyr Project directory using West.

There are three steps to creating a Zephyr RTOS project on a Linux computer:

1. Setting up your development environment (installing prerequisite programs, obtaining python scripts, setting environment variables, etc)
2. Cloning the Zephyr RTOS source code with the Zephyr multi-purpose tool, West.
3. Creating your own application linked to the Zephyr source, which you can compile and upload to your board.

This page will focus on installing all of the scripts and prerequisites you need, along with some other setup tasks that are usually specific to your host computer. This section is broadly similar (with some added suggestions) to the official Zephyr startup guide (https://adafru.it/Jdv), which you can also check out for more information.

Installing Dependencies

The first thing you'll want to do, before installing anything, is to ensure your system is up to date: for Linux, this is handled with the standard terminal commands `sudo apt update` and `sudo apt upgrade`. This guide was created using Ubuntu 18.04 LTS, though most version differences should hopefully be handled by the package managers you'll be using.

Once you're up to date, you can use the `apt` manager to download the prerequisites you need:

```
sudo apt install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-pip python3-setuptools python3-tk python3-wheel xz-utils file \
  make gcc gcc-multilib g++-multilib libsdl2-dev
```

This may skip over some installations if you already have them. Once it is complete, double check that you have the correct version of `cmake` installed:

```
cmake --version
```

If it is not 3.13.1 or above, you'll need to take extra steps to add the Kitware apt repository (https://adafru.it/Jeq), so that you can obtain the correct version of `cmake`. Run the following commands to add the kitware key and repository to `apt`, and then download the correct `cmake` version.

```
wget -O - https://apt.kitware.com/keys/kitware-archive-latest.asc 2>/dev/null | sudo apt-key add -
sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ bionic main'
sudo apt-add-repository 'deb https://apt.kitware.com/ubuntu/ bionic main'
sudo apt update
sudo apt install cmake
```

Python Virtual Environments and West

Zephyr also has a lot of Python 3 dependencies you will need to install using the pip3 package manager. However, I'm going to diverge from the official guide here and *strongly* suggest you use a **python virtual environment** before doing any installations with pip. Often, when doing the installations for different projects, you'll find that one project has a conflicting version requirement with another - a virtual environment system allows you to set up a fresh "virtual" Python installation any time you want, letting you install the right versions of all your prerequisites specifically for that project. It's a good idea for any developer, and is particularly useful for projects with a huge list of python requirements like

Zephyr.

There are a number of virtual environment managers, but I recommend `virtualenv` and `virtualenvwrapper` for simplicity. Install them now by running:

```
sudo pip3 install virtualenv virtualenvwrapper
```

You then need to add the following text to your `~/.bashrc` on Ubuntu. If it doesn't exist already, create it with `sudo nano ~/.bashrc` and paste in the following after any existing contents:

```
# virtualenv and virtualenvwrapper
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
source /usr/local/bin/virtualenvwrapper.sh
```

Then, run:

```
source ~/.bashrc
```

This should display some setup information. If you run into any errors, make sure the location of your python executable is correct by using `which python3`, which will show you the path to your default Python version. If it's different from `/usr/bin/python3`, change that section to whatever `which python3` showed you.

Now, you should have `virtualenv` installed! Some of the commands available to you are:

- Create with `mkvirtualenv`
- Activate with `workon`
- Deactivate with `deactivate`
- Remove with `rmvirtualenv`

For now, make a new virtual environment for your Zephyr installs and move to it by using:

```
mkvirtualenv zephyenv
workon zephyenv
```

You're now in a pristine new virtual environment for Python! Any time you log on or restart your computer, you can return to it by running `workon zephyenv` again. It *only* has Python 3 installed, so you don't have to worry about any other versions of Python you may have, you can use `python` and `pip` directly and it'll use the Python 3 versions automatically.

Now that that's over with, you can install the Zepyhr multi-purpose tool, West, which will help you download the Zephyr RTOS code and wrap up all your other Python requirements. Get it by running:

```
pip install west
```

Getting the source code and adding packages

Now that you have West installed, you'll be using it to download the Zephyr RTOS source code into a new directory in

your home folder called `zephyrproject/`:

```
cd ~
west init zephyrproject
cd zephyrproject
west update
```

Note that the zephyrproject/ folder is NOT your application folder - it isn't where you're going to put main.c or any of the specific libraries for your application. It's the Zephyr source code, which we'll be linking into new, totally separate application folders that we make later. ZephyrProject refers to the The Zephyr Project™, not your project!

Make sure you give yourself some time to do this. `west update` clones all of the various software libraries and repositories for a big collection of chips - depending on your internet speed, it could take up to an hour.

Once that's done, you'll install the latest python requirements for Zephyr (make sure you're working on the virtualenv you set up before!). From inside the `zephyrproject/` directory, run:

```
pip install -r zephyr/scripts/requirements.txt
```

This should show a new, long list of Python modules being installed on your virtual environment.

Installing the Development Toolchain

Now we need the proper build tools! On Linux this is handled automatically via the Zephyr SDK, which has a number of features not available on other operating systems. Download it as a self-extracting binary:

```
cd ~
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.11.1/zephyr-sdk-0.11.1-setup.run
```

Enter the following commands to run the installation binary, add the SDK to your home folder, and set some udev rules required for flashing hardware:

```
chmod +x zephyr-sdk-0.11.1-setup.run
./zephyr-sdk-0.11.1-setup.run -- -d ~/zephyr-sdk-0.11.1
sudo cp ${ZEPHYR_SDK_INSTALL_DIR}/sysroots/x86_64-pokysdk-linux/usr/share/openocd/contrib/60-
openocd.rules /etc/udev/rules.d
sudo udevadm control --reload
```

You'll also want to add some environmental variables to your `~/.bashrc` file to tell West where to find the SDK, so you don't need to remember to set them every time you turn on your computer. Add the following lines to your `~/.bashrc` file, after the lines we added earlier for the `virtualenv`:

```
#zephyr build vars
export ZEPHYR_TOOLCHAIN_VARIANT=zephyr
export ZEPHYR_SDK_INSTALL_DIR=~/zephyr-sdk-0.11.1
```

Add the variables to this session by running:

```
source ~/.bashrc
```

You can now double check that the environmental variables were set properly by running:

```
echo $ZEPHYR_TOOLCHAIN_VARIANT
```

Once those variables match up, you're all done with the installation process! It's always a lot of work getting a multi-stage development environment like this running, but thankfully you usually only have to do it once. To test your new Zephyr setup, you can use the next section: building a sample program directly out of the Zephyr Project directory using West.

Building a Sample Program

To test your development environment and get some validation that everything is ready to work on hardware, you can build one of the Sample Programs that come bundled in with the Zephyr RTOS source installation. Doing this doesn't tell you much about how to set up your own application, but it does at least make sure that when you do have your custom application created, it'll flash to the board correctly.

The Feather STM32F405 Express is built into Zephyr as a supported development board. This means that it's easy to get code built and flashed onto the board in just a few commands, with no board-specific configuration required. To get started, open your terminal and set up the environmental variables for your project:

```
cd ~/zephyrproject/zephyr
source zephyr-env.sh
```

Then, build the most basic example, `blinky`, with the following command:

```
west build -p auto -b adafruit_feather_stm32f405 samples/basic/blinky
```

The `-p auto` parameter will automatically clear out the remains of previous builds, so you can try building some other samples this way later without extra steps.

Once your sample is finished building, you'll need to use one of the two programming options we discussed in the Hardware section.

DFU Bootloader

DFU is the default programming option supported by `west` for the Feather STM32F405. Plug in the STM32 Feather, making sure the Boot0 pin (B0) is connected to 3.3V, and hit the reset button. Then run the following:

```
west flash
```

If for whatever reason you'd rather not use the `west` tool, you can also run the DFU-util command directly on the binary:

```
cd ~/zephyrproject/zephyr/build/zephyr
dfu-util -a 0 --dfuse-address 0x08000000 -D zephyr.bin
```

Jlink

Jlink is also supported as a west flash "runner", but it isn't the default option, so you need a slightly longer command to override it. Connect the Jlink and USB cable to your board, and run:

```
west flash --runner jlink
```

In either case, you should get a blinking LED light on your board if everything is set up correctly.

Creating Your Own Application

Of course, running someone else's code is only useful as a test. The real first step toward creating your own Blinky system is to make your own application directory, separate from the Zephyr source code, and build it. Unlike some other RTOS systems, you don't need to copy all the Zephyr source code into every project you start - `west` will automatically link to your existing `zephyrproject` directory whenever you build.

To get started, create a new directory in your location of choice called `my_blinky`. This can be anywhere, but it's important that the system path to it contains **no spaces**. If you've created a folder named `firmware projects`, for instance, you'll need to change it to `firmware_projects` or nothing will build.

A Zephyr application directory has the following components:

- **CMakeLists.txt**: your build settings configuration file - this tells `west` (really a cmake (https://adafru.it/Jdw) build system under the hood) where to find what it needs to create your firmware. For more advanced projects, it's also used for debug settings, emulation, and other features.
- **prj.conf:** the Kernel configuration file. For most projects, it tells Zephyr whether to include specific features for use in your application code - if you use GPIO, PWM, or a serial monitor, you'll need to enable them in this file first.
- **src/main.c:** your custom application code - where the magic happens! It's advisable to put all of your custom source code in a `src/` directory like this so it doesn't get mixed up with your configuration files.
- **Optional boards/ or soc/ overlay directories:** in some cases, the board or even the specific MCU chip you want to use might not be supported yet in Zephyr. If this is the case, you can create your own definitions (https://adafru.it/KVd) in your custom projects. It's out of the scope of this starting tutorial, but it's a good option to know about.

The Feather STM32F405 Express is built into Zephyr as a supported board. This means that you don't need to code any configuration files or adjust any settings - it's all included already!

You don't need to create the rest of the files from scratch either. Instead, navigate to your `zephyrproject/zephyr/samples/basic/blinky` and copy **CMakeLists.txt**, **prj.conf**, and **src/main.c** to `my_blinky/`. Open **CMakeLists.txt** in your favorite text editor and you'll see the following:

```
# SPDX-License-Identifier: Apache-2.0

cmake_minimum_required(VERSION 3.13.1)
include($ENV{ZEPHYR_BASE}/cmake/app/boilerplate.cmake NO_POLICY_SCOPE)
project(zblinky)

target_sources(app PRIVATE src/main.c)
```

All you have to do is change the `project(projectname)` to `my_blinky`, and your files are ready to compile!

If you opened a new terminal window to do this, note that you'll need to run the following again:

```
workon zephyenv
source ~/zephyrproject/zephyr/zephyr-env.sh
```

This will make sure west is available, and is aware of the environmental variable `$ZEPHYR_BASE` in the `CMakeLists.txt` file above. `$ZEPHYR_BASE` stores the location of the source code in `zephyrproject/` - you can view it by typing `echo $ZEPHYR_BASE` in the terminal if you'd like to double check.

Now that we're set up, building and flashing is very similar to what you did with the sample blinky project:

DFU Bootloader

Plug in the STM32 Feather, making sure the Boot0 pin (B0) is connected to 3.3V, and hit the reset button. Then run:

```
west build -p auto -b adafruit_feather_stm32f405
west flash
```

Jlink

Ensure the board is powered and plugged into the Jlink, then run:

```
west build -p auto -b adafruit_feather_stm32f405
west flash --runner jlink
```

Creating custom code

So far, this hasn't been much different than what you did for the sample blinky project. However, now that you're in our own directory, you can go in and make changes. You can start with a simple change to alter the speed the LED blinks at. Open up `src/main.c` and you'll see the following:

```
#include <zephyr.h>
#include <device.h>
#include <drivers/gpio.h>

#define LED_PORT DT_ALIAS_LED0_GPIOS_CONTROLLER
#define LED  DT_ALIAS_LED0_GPIOS_PIN

/* 1000 msec = 1 sec */
#define SLEEP_TIME 1000

void main(void)
{
 u32_t cnt = 0;
 struct device *dev;

 dev = device_get_binding(LED_PORT);
 /* Set LED pin as output */
 gpio_pin_configure(dev, LED, GPIO_DIR_OUT);

 while (1) {
  /* Set pin to HIGH/LOW every 1 second */
  gpio_pin_write(dev, LED, cnt % 2);
  cnt++;
  k_sleep(SLEEP_TIME);
 }
}
```

Try changing `#define SLEEP_TIME 1000` to `100` instead. Then run the build and flash commands again. The LED should be blinking much faster.

Congratulations! You've taken your first steps into the RTOS rabbit hole. From here, you can explore the other samples, think about how an RTOS could help your own projects, and even start programming your own fully custom projects. I'll offer some tips and links on where to start on the final page.

Next Steps

Often, the toolchain installation phase can be the most frustrating part of a project. Now that you're past it, you're free to begin exploring the real depth of Zephyr and consider how you could make use of an RTOS in practice. Below is a quick overview of some basic concepts to understand, and some links and resources you can follow to explore more on your own.

Basic RTOS vocabulary

- **Threads:**

  The core appeal of an RTOS is that it can run many different objectives at once, called ***Tasks***, using a priority management system called a ***Scheduler***. Tasks can be something like blinking an LED, sending a message over a serial terminal, or handling a motor driver - on an RTOS, the scheduler can juggle many different tasks at once, running each one until another one takes priority or a certain amount of time is exceeded. This makes it simple to handle large amounts of competing tasks in an organized and time-efficient way.

- **Determinism:**

  RTOS systems are desired for their ability to be ***deterministic***, defined as the ability to always complete specific objectives in a known amount of time. User-facing operating systems like Windows, OSX and Linux are not deterministic, allotting time to tasks like memory management or storage at any time, meaning they often experience delays or lags in operation. In many industrial, robotics and sensor applications, delays like these could lead to equipment damage, or even put lives in danger! An RTOS *guarantees* that tasks will be completed on time, making it an essential tool for these high performance fields.

- **Scheduling (pre-emptive and time-slice):**

  Not every scheduler works the same way. A ***pre-emptive scheduler*** handles tasks based on priority - a low priority task like blinking a status LED might happen continuously, until a high-priority message send task or motor control task takes over for a while. High priority tasks can then cede control back to the low priority ones by using a `yield()` or `sleep()` function. Another model is a ***time-slice*** scheduler, which gives each task a certain allotment of time to run per second. Time-slice schedulers are harder to make deterministic (see above) and thus are much less common than pre-emptive schedulers.

- **Semaphores and Mutexes:**

  Often, different threads will need to use the same resource, such as a single I2C bus. But if one attempts to access it while the other is already using it, you could get strange behavior. A ***semaphore*** is a variable that is accessible to both threads, which can convey information about a shared resource. Often, a semaphore is simply a "locked/unlocked" binary variable that states whether a resource is in use - this kind of semaphore is called a ***mutex***.

- **Message Queues:**

  In other cases, longer and more detailed information needs to be distributed between threads. Most RTOS implementations include a ***Message Queue*** system, where threads can post messages to a shared list viewable by other threads. An example could be a system where many different tasks gather sensor data, but they all send it to a single processing task, using a data-carrier Message Queue.

Useful RTOS Samples

- **zephyrproject/zephyr/samples/synchronization**: a great starting point for using threads. Two different threads trade the ability to print "hello world", mediated by Semaphores.
- **zephyrproject/zephyr/samples/philosophers**: a more complex RTOS example, this project follows the classic RTOS "Dining Philosophers" problem, which involves competing threads and many essential concepts.

You can find out more about the classic examples here:
https://docs.zephyrproject.org/latest/samples/classic.html (https://adafru.it/Jdx)

Other useful links

- Wikipedia has an excellent page (https://adafru.it/Jdy) on RTOS concepts, even for new programmers.
- For more advanced application development, including debugging, custom toolchains, and more, the Zephyr documentation has an exhaustive list of instructions (https://adafru.it/Jdz). The docs are always a good place to start if you get stuck.
- Engaging with the Zephyr community:
    - Asking for help tips (https://adafru.it/JdA)
    - The Zephyr github page (https://adafru.it/JdB)
    - Zephyr Slack (https://adafru.it/JdC)
    - Zephyr user mailing list (https://adafru.it/JdD)