Python Tutorial and Application HandBook

MPO-2000 Series

GW INSTEK

Preface

This manual is a Python basic tutorial document that we have written for beginners, with the aim of helping users quickly learn and master the basics of Python programming. Through the examples provided in this manual, you can learn how to control various built-in hardware devices and external equipment in the MPO-2000 series, thereby applying Python to practical automation control and automated testing projects.

We would like to express special thanks to Guido van Rossum, the original designer of the Python language, who first released Python 0.9.0 in 1991. We would also like to thank Damien P. George, who was inspired by the Python language and implemented MicroPython, which can run on embedded systems. MicroPython made its first appearance in a Kickstarter crowdfunding campaign in 2013. And thanks to the contributions of the Python and MicroPython communities, based on their achievements and the efforts of our engineers, we are able to provide users with the ability to directly execute MicroPython application scripts on MPO-2000.

MicroPython is a subset of the Python 3 scripting language designed for microcontrollers and embedded systems. Like Python, it does not require programs to be compiled in advance on a PC and can be executed directly by an interpreter. Compared to C or other languages, using MicroPython allows users to develop software more efficiently and with higher readability and maintainability. Although MicroPython is limited by system resources and only

implements some data types and module functions of Python 3, we believe it is fully capable of being applied to small-scale automated test systems, and we will refer to it as Python in the following description.

To enable Python application scripts in the MPO-2000 series to have a richer graphical user interface, we have ported the LVGL (Light and Versatile Graphics Library) library into the system, allowing Python script developers to provide more aesthetic and user-friendly user interfaces when designing automated measurement functions. Due to the limited documentation available for LVGL library use on the internet, you may encounter more issues during development. We recommend that this be done by professional engineers.

We hope that this manual will lead users to understand the basics of Python script design in the shortest time possible, so that users can control local hardware and external devices through the examples we provide, understand the applications and advantages of Python, and apply what they have learned to practical testing projects.

We recommend that users further their learning by reading more books and examples about Python. Python is a very popular scripting language with abundant resources and communities, and you can further enhance your programming skills by learning Python extensively.

Sincerely,
The Oscilloscope Development Team

* Python is a trademark of the Python Software Foundation (PSF).
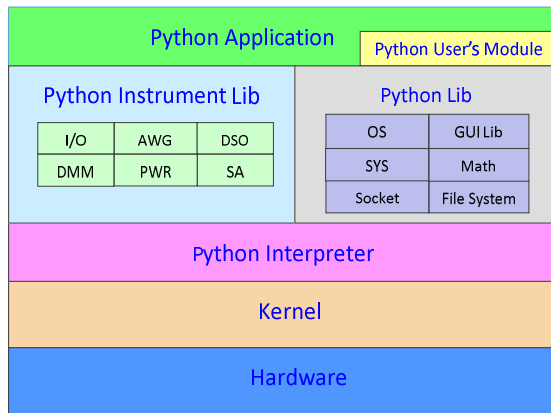
Table of Contents

System Operation and Architecture

In a typical scenario, after booting up, MPO-2000 is controlled by the main control software to operate various functions of the oscilloscope. Users operate the oscilloscope's functions through the on-screen menus, panel buttons, and knobs. Python scripts start running when the user clicks on the pre-installed Python APP by selecting the 'µPy/Exit' button. At this point, the main control software invokes the Python interpreter to execute the .py file in the script's directory. Users can also directly run Python scripts written by themselves from the file utility by selecting internal flash disks or USB drives containing .py files. For the main control software, the Python script run by the interpreter is an independently running software. The communication of commands and data between the main control software and Python scripts is achieved through the socket communication protocol, and the transmitted commands use the SCPI commands commonly used in remote control. Therefore, before controlling various functions of the oscilloscope, Python scripts must establish a socket connection. Once the socket connection is successfully established, the transmission of various commands can begin. The following diagram illustrates the Python system architecture of MPO-2000.

Python Application

Python User's Module

Python Instrument Lib

I/O	AWG	DSO
DMM	PWR	SA

Python Lib

OS	GUI Lib
SYS	Math
Socket	File System

Python Interpreter

Kernel

Hardware

Software block diagram

MPO-2000 integrates not only the functions of an oscilloscope but also various other devices such as an arbitrary waveform generator, a spectrum analyzer, a digital multimeter, and a power supply unit. We also provide a basic Python Library for greater user convenience. Among these devices, the oscilloscope, arbitrary waveform generator, digital multimeter, and power supply unit operate independently, and the settings in the Python Library for each device do not interfere with each other. However, since our spectrum analyzer operates by processing time-domain sampled data using FFT (Fast Fourier Transform), there may be conflicts and anomalies if you wish to control both the oscilloscope and the spectrum analyzer simultaneously using the Python Library. Special attention is needed in such cases.

MPO-2000 is designed to meet the requirements of small-scale automated testing, particularly suitable for simple and repetitive measurements and control. Under the control of Python scripts, measurement data can be directly evaluated on the local machine, eliminating the need to transfer waveform data or measurement data back to the host PC for evaluation. This simplifies the test system architecture and also offers energy savings and reduced carbon emissions. For some simple testing applications, only one MPO-2000 unit, along with the necessary probes and test fixtures, is required.

MPO-2000 also provides a Python GUI library, allowing users to design their own graphical information displays on the machine's screen, including custom menus, measurement data, statistical charts, I-V characteristic curves, and more.

When the built-in devices are insufficient to meet testing requirements, such as insufficient DC power voltage or power, MPO-2000 can cooperate with external devices controlled by Python scripts via a USB host port, controlling external devices like the GW Instek PSW, PFR, and PPX power supply series. It's important to note that the connected devices must use the USB CDC-ACM protocol to communicate commands and data with MPO-2000. In addition to the USB interface, MPO-2000 can also control external devices for collaborative testing through network interfaces using the socket protocol.

The diagram below illustrates an example of MPO-2000 controlling external devices for collaborative testing and uploading measurement data to a server or the cloud.
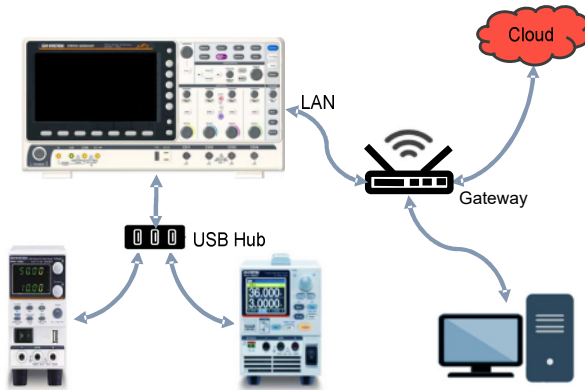
Collaborative testing example

Under the control of Python scripts, the components of the graphics library in MPO-2000 also support data input from USB keyboards, mouse, and barcode scanners, making the human-machine interface operation of the testing system more convenient.

Differences Between Basic and Professional Versions

The MPO-2000 series is categorized into two versions, Basic and Professional, based on functional specifications. Apart from differences in bandwidth, the model code suffix 'B' represents the Basic version, which has fewer system resources available for Python script execution. The 'P' suffix represents the Professional version, which offers more system resources for Python script execution. Python APP provided by MPO-2000 can be executed directly on both Basic and Professional models without being restricted by the three functional differences described below. For the individual execution of Python APP, some may require suitable circuit boards or external devices, as described in the documentation for each Python APP. Users can also copy the source code of Python APP from the menu, modify it as needed, and save it to the internal disk for execution. Scripts modified from Python APP will be subject to the three functional differences described below when executed on Basic models. The following are the four functional differences between the Basic and Professional versions:

Python Script Memory Usage

The Basic version of Python scripts uses a limited amount of memory and is suitable for smaller-scale control software. For applications that require waveform data capture and processing, it is recommended to set the sampling points to 1000 points. Otherwise, you may encounter insufficient memory situations.

The Professional version of Python scripts, on the other hand, can utilize more memory and handle more complex control software. For cases requiring waveform data capture and processing, it is advisable to set the sampling points to a maximum of 100,000 points.

The actual available memory space may vary depending on user-defined variables and the size of the software. Users should make use of the *gc* module to release unused memory and avoid variables used within functions continuously occupying additional memory.

External USB Device Support

The Professional version of the device, under the control of Python scripts, can control external devices with USB CDC-ACM protocol support through the front-panel USB host port. The Basic version does not support this feature.

Currently, the USB CDC-ACM devices supported by MPO-2000 include our PSW series, PFR series, and PPX series power devices. If you have requirements for other products control, please consult with us first. Other devices with USB interfaces not produced by GW Instek are not supported because their driver mechanisms may differ from those of our products.

In conjunction with the use of the Professional version's graphical interface, we also provide support for USB HID protocol keyboards, mouse, and barcode scanners. Users can use the mouse for graphical interface operations and can also input text messages in automated test programs, facilitating the recording and archiving of the model, serial number, and measurement data of the DUT (Device Under Test).

Python Graphics Library Support

To enable richer graphical user interfaces for Python application scripts on the MPO-2000 series, we have ported the LVGL (Light and Versatile Graphics Library) to the system. LVGL is a MIT-licensed open-source project available on the GitHub website. If users need to design their own graphical user interfaces, charts, or modify Python apps that utilize the LVGL library, they must do so on the Professional version of the MPO-2202P/2204P.

Packaging Python Scripts into a Python App

In addition, the Professional version also includes the capability to package Python scripts into Python APP installation file(*.xpy*). Advanced users can package the relevant files of their designed Python scripts and provide them for installation on specific machines. When packaging, users must prepare a list of MAC addresses and serial numbers for the machines they intend to install on. Only machines appearing on the list are allowed to install this Python APP. Considering the intellectual property rights of third-party APP developers, the source code of the installed Python APP is not available for copying and downloading, and users cannot access the Python script source code on the machine. This serves the purpose of protecting intellectual property rights.

Users who have purchased the Basic version of the MPO-2102B/2104B can upgrade to the Professional version to enable graphical user interface Python script development.

Python Basic

In this chapter, we will introduce the basic syntax of Python, allowing beginners to quickly get started.

When writing Python scripts, it's important to note that our system only supports ASCII code. If your code contains multi-byte characters such as UTF-8, UTF-16, BIG5, or GB2312, it may lead to display errors in our provided online editor or runtime errors.

Coding Style Guides

The naming style for Python script is recommended to follow the PEP 8 naming conventions. Here are some basic naming rules for Python script:

- Use four spaces for indentation; do not use tabs.
- Names of variables and functions should be concise and descriptive. Use meaningful words and phrases to describe the purpose of variables and functions whenever possible.
- Variable names should be in lowercase, and words can be separated by underscores, e.g., "my_variable".
- Function names should be in lowercase, and words can be separated by underscores, e.g., "my_function".
- Class names should use CamelCase, where the initial letter of each word is capitalized, e.g., "MyClass".
- Constant names should be in all uppercase, and words can be separated by underscores, e.g., "MY_CONSTANT".
- Module names should be in lowercase, and words can be separated by underscores, e.g., "my_module.py"

In addition, consider the following points:

- Avoid using abbreviations and single characters as variable and function names unless they are very common and widely understood.
- Avoid using variable or function names that are the same as built-in function and class names.
- Follow the principle of naming consistency. Use similar naming styles throughout the codebase to make the script more readable and maintainable.

Comments

In Python script, you may need to add some comments to explain your script and improve its readability; they will not be executed by the interpreter. Comments are used to provide explanations about the script, making the script easy to maintain in the future.

Here are the ways to use comments in Python:

Single-line Comment

You can add explanatory text in your script using the "**#**" symbol as the beginning of a comment, extending until the end of that line. Doing this allows you to add explanations within your script without affecting the script's execution.

For example:

Python code

```
# This is a single-line comment
x = 5  # This is a code line with a comment
```

Multi-line Comment

You can also use three single quotes ''' or double quotes """ to enclose a block of description text. Doing so allows you to add multi-line comments in your code without affecting its execution.

For example:

Python code

```
'''
This is a multi-line comment,
It can be used to explain how the code works
'''
x = 5  # This is a code line with a comment
```

Good comments can improve code readability, making your code easier to understand and maintain.

Variables

Variables are containers in Python scripts used to store data. We can use the equal sign to assign values to variables.

For example:

```
x = 5
y = "Hello"
```

The above code defines two variables, x and y. The value of x is 5, and the value of y is the string "Hello". Variable names can use letters, numbers, and underscores, but variable names cannot start with a number.

Data Types

Each variable in a Python script has its corresponding data type. Common data types include integers, floating-point numbers, strings, booleans, tuples, lists, and dictionaries.

You can use the *type()* function to query the data type of a variable.

For example:

Python code

```
X = 5
print(type(x))    #Output:<class 'int'>

y = 3.14159
print(type(y))    #Output:<class 'float'>

z = "Hello"
print(type(z))    #Output:<class 'str'>

w = True
print(type(w))    #Output:<class 'bool'>
```

Integer

Integer is a fundamental data type. You can define integers using the following methods:

Python code
```
x = 10
y = -5
```

Integers support basic mathematical operations such as addition, subtraction, multiplication, division, modulo (remainder), and exponentiation:

Python code
```
x = 10 + 5   # x will be set to 15
y = 10 - 5   # y will be set to 5
z = 10 * 5   # z will be set to 50
w = 10 / 5   # w will be set to 2.0 (floating point number)
r = 10 % 3   # r will be set to 1
p = 10 ** 3  # p will be set to 1000
```

Floating Point

Floating-point is another fundamental data type used to represent real numbers. Floating-point numbers can be defined as follows:

Python code
```
x = 3.14159
y = -0.25
```

Floating-point numbers support the same mathematical operations as integers, as well as some additional operations such as rounding, flooring, and taking absolute values:

```
x = 3.14159
y = round(x, 2)  # Round to 2 decimal places
z = int(x)       # Get the integer part
w = abs(x)       # Get the absolute value
```

String

A string is another fundamental data type used to represent text. You can define a string using the following methods:

Python code
```
x = "Hello, world!"
y = 'This is a string.'
```

We can perform many operations on strings, such as concatenation, splitting, and indexing:

Python code
```
x = "Hello, "
y = "World!"
z = x + " " + y   # z will be set to "Hello world"
w = x[0]          # w will be set to "H"
v = x[0:5]        # v will be set to "Hello"
```

Boolean

A boolean value is a data type that can only have two possible values: *True* or *False*. They can be defined as follows:

Python code
```
x = True
y = False
```

Boolean values are commonly used in conditional statements to control the flow of a program.

Python code

```
x = 5
if x > 3:
        print("x is greater than 3")
```

Tuple

Tuple is an immutable sequence used to store an ordered set of values. Unlike a list, the elements of a tuple cannot be modified. Here's a simple example of using a tuple:

Python code

```
# Create a tuple
my_tuple = (1, 2, 3, "hello", "world")

# Access elements in the tuple
print(my_tuple[0])  # 1
print(my_tuple[3])  # "hello"
# Cannot modify elements in the tuple
my_tuple[0] = 4
```

In the example above, we use parentheses () to create a tuple containing integers and strings, and store it in the variable *my_tuple*. Then, we use indexing to access elements in *my_tuple*. However, because the elements of a tuple cannot be modified, attempting to use indexing to change the value of an element will result in an error at the last line of this script.

Tuples are similar to lists and also have some useful built-in functions, such as:

len(): Returns the number of elements in the tuple.
count(): Returns the number of times a specified element appears in the tuple.
index(): Returns the index of the first occurrence of a specified element in the tuple.

Python code
```
# Create a tuple
fruits = ('apple', 'orange', 'banana', 'grape', 'orange')

# Use index() method to find the index position of 'banana' in the tuple
print(fruits.index('banana'))  # Output: Index position: 2

# Get the length of the tuple using len()
print(len(fruits))  # Output: Number of fruits: 5

# Count the occurrences of the element 'orange' in the tuple using count()
print(fruits.count('orange')) # Output: Number of occurrences of 'orange': 2
```

Since tuples are immutable, they are typically used to store a fixed number of values. When deciding whether to use a list or a tuple, consider whether your program needs to modify elements within the sequence. If modification is required, use a list; otherwise, use a tuple.

List

Lists are a mutable data type used to store multiple items. You can define a list using the following methods:

```
x = [1, 2, 3]
y = ['apple', 'banana', 'orange']
```

Lists are mutable, which means items can be added, removed, or modified within a list. You can use indices to access items in a list:

```
x = [1, 2, 3]
print(x[0])    # will output 1
x[0] = 4       # change the first item to 4
print(x[0])    # will output 4
```

Lists support many operations, such as adding, deleting, and sorting:

```
x = [1, 2, 3]
x.append(4)    # add 4 to the end of the list
print(x)       # will output [1, 2, 3, 4]
x.remove(2)    # remove the item with value 2
print(x)       # will output [1, 3, 4]
```

Dictionary

A dictionary is a data type of key-value pairs used to store multiple items, and it can be defined using the following way:

Python code
```
x = {'name': 'Alice', 'age': 30, 'gender': 'female'}
y = {'apple': 1, 'banana': 2, 'orange': 3}
```

Each item in the dictionary has a key and an associated value, and you can use the key to retrieve the value from the dictionary:

Python code
```
x = {'name': 'Alice', 'age': 30, 'gender': 'female'}
print(x['name'])   # will output "Alice"
print(x['age'])     # will output 30
```

The dictionary supports various operations, such as adding, deleting, and modifying items:

Python code
```
x = {'name': 'Alice', 'age': 30}
x['gender'] = 'female' # add a new key-value pair
print(x)        # will output {'name': 'Alice', 'age': 30, 'gender': 'female'}
del x['age']    # remove the item with key 'age'
print(x)        # will output {'name': 'Alice', 'gender': 'female'}
```

Besides, dictionaries also have some useful built-in functions, such as:

len(): Returns the number of key-value pairs in the dictionary.

keys(): Returns all the keys in the dictionary.

values(): Returns all the values in the dictionary.

items(): Returns all the key-value pairs in the dictionary as tuples.

When using dictionaries, it's important to note that keys must be unique. If the same key is used multiple times, the last value assigned to it will replace the previous value. If you need to maintain order in a dictionary, you can use the OrderedDict class from the collections module.

Dictionaries are widely used in Python programming, especially when dealing with complex data structures and algorithms. This chapter introduces the fundamental data types supported in Python, including integers, floats, booleans, strings, tuples, lists, and dictionaries. Understanding these data types is crucial for learning Python programming. Once you are familiar with these fundamental data types, you can begin to explore more advanced topics such as functions, conditional statements, loops, and more.

Array

The Python *array* module provides a numeric sequence called an array, which is similar to Python lists but is limited to storing data of a fixed type, such as integers or floating-point numbers. Using the *array* module can save memory and improve script execution speed.

Here is an example of using the *array* module:

Python code

```
import array

# Create an array of integers
my_array = array.array('i', [1, 2, 3, 4, 5])

# Access elements in the array
print(my_array[0])  # 1
print(my_array[1])  # 2

# Modify elements in the array
my_array[0] = 6

# Iterate over elements in the array
for element in my_array:
    print(element)
```

In the above example, we create an array containing integers using the *array.array()* function and store it in the *my_array* variable. '*i*' is a character code representing data of type int. Next, we access elements in *my_array* using indexing and modify them in the same way. Finally, we iterate through the elements in *my_array* using a *for* loop.

The *array* module also provides other functions and methods for manipulating arrays, such as:

append() ：Adds an element to the end of the array.
extend() ：Appends all elements from another array to the end of the current array.

When using the *array* module, make sure that the data type you choose matches the type of data your application needs to store. If you need to store data of different types, consider using Python lists instead of arrays.

Operators

Python scripts can perform various operations using a variety of operators, including arithmetic operators, comparison operators, and logical operators, among others. These operators can assist you in performing mathematical calculations, logical evaluations, and more.

Arithmetic Operators

When performing basic mathematical operations, we can use the following arithmetic operators:

- Addition: '+'
- Subtraction: '-'
- Multiplication: '*'
- Division: '/'
- Modulus: '%'
- Exponentiation: '**'

Here are some simple examples:

Python code
```
x = 5
y = 3
print(x + y)    # Output:8
print(x - y)    # Output:2
print(x * y)    # Output:15
print(x / y)     # Output:1.6666666666666667
print(x % y)    # Output:2
print(x ** y)    # Output:125
```

Comparison Operators

When we need to compare two values, we can use comparison operators and obtain a Boolean value (True or False). We can use the 'if' statement to handle the two possible comparison outcomes. Here are the comparison operators in Python:

- Equal: '=='
- Not Equal: '!='
- Less Than: '<'
- Greater Than: '>'
- Less Than or Equal To: '<='
- Greater Than or Equal To: '>='

Here are some simple examples:

Python code
```
points = 85
if points <= 60:
    print("You need to work harder!")
else:
    print("You're doing great!")
```

Logical Operators

When we need to perform logical evaluations in a Python script, we can use logical operators in conjunction with the *if* statement. Here are three common logical operators:

- 'and' operator
- 'or' operator
- 'not' operator

Here are some simple examples:

Python code
```
x = 5
y = 25

if (x > 10) or (y < 20):
    print("At least one condition is true.")
else:
    print("Both conditions are false.")
```

Control Flow Statements

We can use control flow statements to control the execution flow of a script. Here are common control flow statements in Python:

If Statement

The *if* statement can be used to determine whether a condition is met. If the condition is met, the corresponding scripts is executed. For example:

Python code
```
x = 5
if x > 0:
    print("x is greater than 0")
else:
    print("x is not greater than 0")
```

The *else* part is optional, and you can omit it as needed. If *else* is omitted, the program will continue executing the next statement when the condition is false.

We can also use the *elif* statement to handle multiple different conditional checks. For example:

Python code
```
x = 5
if x > 0:
    print("x is greater than 0")
elif x == 0:
    print("x is equal to 0")
else:
    print("x is less than 0")
```

For Loop Statements

A for loop is a common looping structure in Python that allows you to repeatedly execute the same script code. It can also be used to iterate through a sequence of lists, tuples, strings, or other iterable objects. Here is an explanation of how the for loop works:

for **variable** in **iterable**:
    Write the code to be executed here

variable: During each iteration of the loop, an element from the iterable object is assigned to this variable. You can choose any name for this variable, typically related to the elements, such as "item" or "element."

iterable: This is an object that contains multiple elements, such as a list, tuple, string, etc. The for loop will iterate through each element of this object and assign it to the above variable.

Write the code to be executed here: This code will be executed once for each element in the iterable object.

Here is a simple for loop that demonstrates how to use it to display each element in a list:

```
fruits = ['apple', 'banana', 'cherry']

for x in fruits:
        print(x)
```

```
apple
banana
cherry
```

In this example, the variable *x* iterates through each element in the list *fruits* sequentially, and outputs the current element in each iteration.

We can also use the *range()* function to generate a sequence of numbers and then iterate through each number in this sequence using a *for* loop.

Here's an example demonstrating how to use the *range()* function and a *for* loop to display integers from 1 to 5.

```
for x in range(1, 6):
        print(x)
```

```
1
2
3
4
5
```

In this example, *range(1, 6)* generates a sequence from 1 to 5. The *for* loop iterates through each number in this sequence one by one and outputs it to the screen in each iteration.

While Statement

The *while* loop is used to repeatedly execute a block of code as long as a specified condition is true. When the condition is no longer met, the loop stops executing. Here is an explanation of how the *while* loop is used:

```
while condition:
    Code to be executed within the loop
```

Condition: This is a Boolean expression. The loop continues to execute as long as the value of this expression is True. When the condition's value becomes False, the loop stops.

Code to be executed within the loop: This is the code block that you want to run each time the loop executes.

Python code
```
i = 0
while i < 5:
    print(i)
    i += 1
```

Please note that, to prevent infinite loops, designers must ensure that the code within the loop will eventually result in the condition becoming False so that the loop can terminate. If the condition never becomes False, the loop will run indefinitely.

Functions

What is a Function?

Functions are encapsulated and reusable script blocks. You can think of functions as tools that can take input (parameters), perform certain operations, and then output results (return values).

Functions are very commonly used in script design; they allow us to organize scripts better, making the entire script easier to read, maintain, and expand.

Syntax and Usage of Functions

In Python, the syntax of functions is very simple. Here is an example:

```
def add(a, b):
    c = a + b
    return c
```

Here, ***def*** is a keyword, indicating that you are defining a new function. ***add*** is the name of the function, and you can choose a meaningful name yourself. ***a*** and ***b*** are parameters of the function; they are the inputs to the function. The script block inside the function can operate on these parameters. Finally, the ***return*** keyword indicates the output of the function, which is its return value ***c***.

Once you have defined a function, you can use it. Here's an example of calling *add*:

```python
result = add(2, 3)
print(result)  # 5
```

Here, we passed 2 and 3 as arguments to the *add* function, stored the return value of 5 in *result*, and printed it out.

Functions can also have no parameters or return values. For example:

```python
def say_hello():
    print("Hello!")

say_hello()  # Call the Function
```

In this example, the *say_hello* function has no parameters and no return value; it simply prints a "Hello!" string.

Using Global Variables in Functions

In a function, you can use global variables, which are variables defined outside the function. Here's an example:

```python
x = 10

def add():
    global x    # Set x as a global variable
    x += 1
    return x
print(add())  # 11
print(x)        # 11
```

In this example, we have defined a global variable *x* and then modified it within the *add* function. To use a global variable, you need to declare it in a function using the *global* keyword.

Lambda Function

In addition to regular functions, Python also supports *lambda* functions. Lambda functions are a type of simple anonymous function, typically used for operations like simple calculations and sorting. Here is an example:

Python code
```
add = lambda x, y: x + y
result = add(2, 3)
print(result)  # 5
```

In this example, we have defined a *lambda* function using the *lambda* keyword. This function takes two parameters, *x* and *y*, and returns their sum. We have assigned this *lambda* function to the variable *add* and then called it just like a regular function.

Class

A *class* is a blueprint or template in Python that allows you to encapsulate related data and behavior into a single object. A class contains attributes and methods that define the behavior of objects created from that class. Using class can help structure script code and promote reusability.

Creating a Class

We can use the keyword *class* to define a class. A class consists of attributes, which are variables of the class, and methods, which are functions of the class.

Here is a simple example of a class definition:

Python code
```python
class MyClass:
    def __init__(self, value):
        self.x = value

    def get_x(self):
        return self.x

    def set_x(self, value):
        self.x =value
```

In the code above, we have defined a class called *MyClass*. This class has an attribute x and two methods, *get_x* and *set_x*. The first parameter of methods is always *self*, which refers to the instance of the class itself. When we call the class to create a usable object, we don't need to pass an actual *self* parameter; Python handles it automatically for you. In this example, we have also defined a

special method called **_init_**, known as the constructor function. This method is used to initialize object attributes when creating an object. In this case, the **_init_** method sets the input parameter *value* as the initial value for the attribute x.

Creating an Object

Once we define a *class*, we can create objects of that class in the script for practical use. Here is a simple example of creating objects using a *class*:

Python code
```
my_object = MyClass(10)
```

The above code creates an object of *MyClass* named *my_object* and sets its attribute x to 10.

Accessing Attributes

Once we have created an object of a *class*, we can use the dot operator (**.**) to access the properties of this object. Here are examples of accessing object properties:

Python code
```
my_object = MyClass(10)
print(my_object.x)  # Output 10
```

In the above code, we created an object of *MyClass* named *my_object* and set its attribute x to 10. Then, we used the dot operator to access this attribute.

Using Methods

We can use the dot operator to invoke methods of an object. Here are examples of invoking object methods:

```python
my_object = MyClass(10)
print(my_object.get_x())  # Output 10

my_object.set_x(20)
print(my_object.get_x())  # Output 20
```

In the above code, we created an object of *MyClass* and initialized its attribute *x* to 10. Then, we called the *get_x* method and printed its output value. Next, we called the *set_x* method to set the object's attribute *x* to 20. Finally, we called the *get_x* method again and printed its output value.

Inheritance

In Python, you can use inheritance to create new classes that inherit the attributes and methods of a parent *class*. Continuing from the previous **MyClass** class, here is a simple example of inheritance:

```python
class MyChildClass(MyClass):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y

    def get_y(self):
        return self.y

    def set_y(self, y):
        self.y = y
```

In the above code, we have defined a subclass named *MyChildClass* which inherits all the attributes and methods from *MyClass* additionally, the subclass has a new attribute *y* and two new methods *get_y* and *set_y*.

In the constructor of the subclass, we use the *super()* function to call the parent class's *__init__* method to set the parent class's attribute x. Then, we set the subclass's attribute *y*.

In Python, classes are used to organize code. Classes consist of attributes and methods and can be used to achieve code structuring and reusability. The dot operator is used to access the attributes and methods of objects. Furthermore, inheritance can be used to create new classes that inherit the attributes and methods of the parent class, while also allowing the addition of new attributes and methods.

Print

We often use Python's *print()* to display text messages or variable contents, helping with script debugging. When debugging with the MPO-2000's *WebREPL* feature, the messages printed by *print()* will appear on the connected browser's *WebREPL* page.

We can use the following symbols to format the output of *print()*:

%s - string

For example

Python code
```
name = "John"
print("My name is %s" % name)
```

Output
```
My name is John
```

%d - integer

For example

Python code
```
age = 25
print("My age is %d years old" % age)
```

Output
```
My age is 25 years old
```

%f – float

For example

```
height = 1.75
print("My height is %.2f meters" % height)
```

My height is 1.75 meters

%x - hexadecimal integer

For example

```
number = 255
print("The hexadecimal representation of this number is %x" % number)
```

```
The hexadecimal representation of this number is ff
```

%o - octal integer

For example

```
number = 255
print("The octal representation of this number is %o" % number)
```

```
The octal representation of this number is 377
```

These formatting symbols can be used in combination with other parameters of the *print()* function, as shown in the following example:

```python
name = "John"
age = 25
height = 1.75
print("My name is %s, my age is %d years old, and my height is %.2f
meters." % (name, age, height))
```

```
My name is John, my age is 25 years old, and my height is 1.75 meters.
```

Module

The modules in Python are a way of organizing script code, allowing you to group related classes, functions, and variables within a single file for more efficient code management and reusability. The main purpose of modules is to provide a means of modularizing script code, making it easier to maintain and extend. Here are the steps for creating and using modules in Python:

1. Create a new file with a .py extension, for example, my_module.py.
2. Define the script code for the functions, classes, variables, etc., that you want within the file.
3. In the file where you need to use that module, use the import keyword to import the module.

For example, create a function named *my_function* in *my_module.py*:

Python code
```
def my_function():
        print("Hello from my_function!")
```

Load *my_module* in another script file that requires the use of *my_function*:

Python code
```
import my_module
my_module.my_function()  # Call my_function
```

Output
```
Hello from my_function!
```

Import

We can use *import* to bring external libraries, modules, and classes into the program so that we can utilize the functions, classes, and methods they provide.

Using the import statement

Using the *import* statement allows you to import external modules and libraries in Python. By using the dot operator, you can also load specific modules and classes from a library. Here is a simple example demonstrating how to use the *import* statement in Python:

Python code
```
import time

while True:
    print("Hello, World!")
    time.sleep(1)
```

In this example, we use the *import* statement to import the *time* library. After printing 'Hello, World!' each time, we use the *time.sleep()* function to pause the script for 1 second before continuing.

Loading Modules and Classes from a Library

A library typically contains multiple modules and classes. When using the *import* statement to import a library, it is possible to load specific classes or functions from the library without importing the entire module. Here is a simple example demonstrating how to import the *sqrt* function from the *math* module in Python:

Python code
```
from math import sqrt

number = 16
square_root = sqrt(number)
print("The square root of", number, "is", square_root)
```

File

Regarding file access, the standard method for accessing files is by using the built-in function *open()*. The default behavior of this function is for reading mode, while for writing mode, you need to provide the second parameter '*w*'. In binary reading mode, the second parameter is '*rb*', and in binary writing mode, it is '*wb*'.

By specifying different paths, we can access files stored in different locations. Here are several paths for accessing files:

- /mnt/disk : Stored in the internal flash memory of the machine. Due to limitations in storage space and the number of write cycles of flash memory cells, we do not recommend storing a large amount of data or frequently written files here.
- /mnt/usb : Stored on an external USB storage device. Before using it, please ensure that the USB storage device is properly connected, and the file system is recommended to be in FAT32 format.
- /tmp/remote : Stored on remote disks. Before using it, please ensure that the network cable is properly connected, remote PC sharing settings are configured, and local network and related settings (APP -> Mount Remote Disk) are completed.

Write to a File

Here, creating a file under /mnt/disk is used as an example.

Handling files in different paths is similar.

Python code
```
f = open('/mnt/disk/data.txt', 'w')
f.write('some data')
# The "9" in console is the number of bytes that were written with the write() method.
f.close()
```

Read File

Continuing from the previous file writing operation, the file reading operation under /mnt/disk is as follows.

Python code
```
f = open('/mnt/disk/data.txt')
f.read()
# The "some data" in console is the file contents .
f.close()
```

Try... Except

When writing Python scripts, you may encounter some errors, such as when you attempt to execute a function that doesn't exist, manipulate an uninitialized variable, or encounter syntax errors, and so on. These errors will cause your script to stop executing and often result in an error message being printed to the console. To prevent your script from crashing, you can use the *try-except* statement to catch and handle these errors. In Python script design, error handling is crucial, as it is a key factor in whether the script can run stably.

When working with file handling, you may encounter situations like files not existing or read errors. When performing network operations, you might face scenarios such as failed network requests, server connection issues, or timeouts. When taking user input, invalid data from the user can lead to subsequent script processing errors. During mathematical operations, division by zero might occur. Improper data input during type conversion can also result in processing errors. All of these can lead to errors and crashes during script execution. Therefore, whether a Python script can run stably is closely related to how well the script designer handles exceptions.

Here's a simple example to illustrate how to use the *try-except* statement:

Python code
```
try:
    # code that might raise an error goes here
    x = int("hello") # this will raise a ValueError error
except ValueError:
    # handle the ValueError error here
    print("Invalid input. Please enter a valid number.")
```

In the above script code, we use *try-except* to catch the *ValueError* error that may occur when we attempt to convert the string "hello" into an integer. If a *ValueError* error occurs, the script code inside the *except* block will be executed.

You can also use multiple *except* blocks to handle different types of errors:

Python code
```
try:
    # code that might raise an error goes here
    x = 5 / 0 # this will raise a ZeroDivisionError error
except ZeroDivisionError:
    # handle the ZeroDivisionError error here
    print("Division by zero is not allowed.")
except ValueError:
    # handle the ValueError error here
    print("Invalid input. Please enter a valid number.")
```

In the above script, we attempt to divide 5 by 0, which results in a *ZeroDivisionError*. We use a *try-except* statement to catch this error, handle it in the *except* block, and output a message.

The basic concept of the *try-except* statement is as follows: try to execute the script code that may produce an error, and if an error occurs, jump to the corresponding except block where the error is handled. If the script runs without any issues, it won't jump to the *except* block.

You can also use a generic *except* block to handle errors of all types:

Python code
```
try:
    # code that might raise an error goes here
    x = int("hello") # this will raise a ValueError error
except:
    # handle all types of errors here
print("An error occurred in the program.")
```

In the above script, we used a generic *except* block to handle all types of errors. If any type of error occurs, it will jump to this *except* block.

Finally, you can also use an *else* block within the *try-except* statement. This *else* block executes after the script inside the *try* statement runs normally and only if there are no errors. For example:

Python code
```
try:
    # code that might raise an error goes here
    x = 5 / 2 # this will not raise an error
except ZeroDivisionError:
    # handle the ZeroDivisionError error here
    print("Division by zero is not allowed.")
else:
    # code that will be executed if no errors occur
    print("The program ran successfully.")
```

In the script above, we used an *else* block to determine if the script ran successfully. Since no errors occurred, the script within the *else* block gets executed.

Using a *try-except* statement can help us catch and handle any errors that may occur within the script, thereby preventing the software from crashing. When using a *try-except* statement, you can use different *except* block as needed to handle different types of errors, or you can use a generic *except* block to handle all types of errors. Additionally, you can also use an *else* block to determine if the script ran successfully.

Garbage Collection

The *gc* module is used to reclaim memory that is no longer in use, and it can improve the efficiency of script execution. Below is an explanation of how to use the *gc* module:

1. Importing the Module
Before using the *gc* module, you need to import it. You can import the *gc* module into your program using the following code:

Python code
```
import gc
```

2. Manual Memory Recycling
If you wish to manually recycle memory in certain situations, you can use the following code:

Python code
```
gc.collect()
```

3. Checking Memory Usage
The *gc* module provides *mem_free()* and *mem_alloc()* functions to inquire about the current memory usage and available memory space. Specifically, the *mem_free()* function returns the number of bytes of currently available RAM (Random Access Memory) space. The *mem_alloc()* function returns the number of bytes of allocated RAM space. You can use these functions in your Python code to query the current RAM usage.

Here is an example code that utilizes the *gc* module to display available and allocated RAM space:

```python
import gc
import gds_info

# Clear the RAM space
gc.collect()

# Query the current available RAM space
free_mem = gc.mem_free()
print("1. Free memory: {} bytes".format(free_mem))

# Query the allocated RAM space
alloc_mem = gc.mem_alloc()
print("alloc_mem: %d"%gc.mem_alloc())

# Create a list to allocate RAM space
s=[0]*100

# Query the allocated RAM space
alloc_mem = gc.mem_alloc()
print("alloc_mem: %d"%gc.mem_alloc())
```

Output *(Professional version)*
```
1. Free memory: 20424048 bytes
alloc_mem: 67248
alloc_mem: 67760
```

Output *(Basic version)*
```
1. Free memory: 954800 bytes
alloc_mem: 69808
alloc_mem: 70320
```

This example code demonstrates how to use the *gc* module to query available and allocated RAM space.

4. Enable/Disable Garbage Collection

You can use the following script to enable/disable garbage collection:

Python code
```
gc.enable()
gc.disable()
```

Among them, *gc.enable()* is used to enable garbage collection, and *gc.disable()* is used to disable garbage collection.

Common Errors

The following are some common Python error messages and their explanations. If you understand the meanings of these error messages, you can quickly find the corresponding solutions to resolve issues.

1. **ArithmeticError**

   This error message indicates a general arithmetic exception error, which includes anomalies in mathematical operations, such as division by zero.

2. **AssertionError**

   This error message indicates that an assertion in your script has failed. Please check the content of your script to ensure that assertions are correct.

3. **AttributeError**

   This error message indicates that you are trying to access an attribute or method of an object that does not exist. Please check the object you are trying to access and ensure that the desired attribute or method exists.

4. **EOFError**

   This error message indicates an unexpected end-of-file encounter. This typically happens when attempting to read data from a file but reaching the end of the file.

5. **Exception**

   This error message signifies that an error has occurred; it is a general error message used to confirm the occurrence of any type of error.

6. **ImportError**

This error message indicates that you are trying to import a module or package that does not exist. Please verify the correctness of the module name you are trying to import or check your module path.

7. **IndentationError**

This error message indicates that there is an indentation error in your script. Please ensure proper indentation according to Python syntax.

8. **IndexError**

This error message indicates an attempt to access an index that does not exist in a list or array. Please check your index range and ensure that the index value is within a reasonable range.

9. **KeyError**

This error message indicates an attempt to access a key in a dictionary that does not exist. Please check your dictionary and verify that the key you are trying to access exists.

10. **KeyboardInterrupt**

This error message indicates that you have pressed the [Ctrl] + [C] key combination during script execution, forcefully terminating the script. Please ensure that your [Ctrl] + [C] action is intentional and handle this interruption signal in your script when necessary.

11. **LookupError**

This error message indicates a failed lookup operation.

12. **MemoryError**

This error message indicates a memory shortage issue during script execution. Please reduce memory usage in your script or check for issues like infinite loops or recursive functions.

13. **NameError**

This error message indicates the usage of an undefined variable. Please define the variable in your script.

14. **NotImplementedError**

This error message indicates an attempt to use a feature or method that has not yet been implemented. It is often used by developers in classes to signify that a feature is not yet implemented but may be in the future.

15. **OSError**

This error message indicates an error related to the operating system. Please check your script and find a solution for the operating system error.

16. **RuntimeError**

This error message indicates a runtime issue in your script. For example, a script using a recursive function may encounter a stack overflow during runtime.

17. **StopIteration**

StopIteration is typically used to mark the end flag of an iterable object that has been fully traversed.

18. **SyntaxError**

This error message indicates that your code has a syntax error. Please check for errors in your code and make corrections.

19. **TypeError**

This error message indicates an attempt to use an object in an unsupported way, such as trying to concatenate a string with a number. Please check the data type of the object and use it appropriately.

20. **ValueError**

This error message indicates an attempt to use a value that is invalid for a specific operation, such as passing an empty string to the *int()* function. Ensure that the value you are using is valid for the operation.

21. **ZeroDivisionError**

This error message indicates that you are attempting to perform a division by zero on a number.

Oscilloscope Library

We have created Python libraries for various built-in features on the MPO-2000, making it convenient for users to call. This chapter will introduce how to use these Python libraries to control this multifunctional oscilloscope, including the basic Python control methods for the oscilloscope, arbitrary waveform generator, spectrum analyzer, digital multimeter, and DC power supply, as well as the basic Python control methods for the GO-NOGO output pin.

Basic Oscilloscope Operations with Python

We must first load the *gds_info* module and establish a connection before we can begin controlling the oscilloscope. The following example will demonstrate how to establish an oscilloscope connection and configure its parameters such as channels, horizontal settings, trigger, and others.

```
# Import the DSO module and open a socket connection.
import gds_info as gds
dso=gds.Dso()
dso.connect()

# Set to the default settings.
dso.default()

# Turn on CH1
if not dso.channel.is_on(ch=1):
      dso.channel.set_on(ch=1)

# Set the probe ratio, probe type, vertical scale, vertical position of CH1.
dso.channel.set_probe_ratio(ch=1,ratio=1)
dso.channel.set_probe_type(ch=1,type='VOLTAGE')
dso.channel.set_scale(ch=1,scale=0.1)
dso.channel.set_pos(ch=1,vpos=-0.0)

# Set the horizontal scale.
dso.timebase.set_timebase(hdiv=1e-05)

# Set the trigger mode, trigger level.
dso.trigger.set_mode(mode='AUTo')
dso.trigger.set_level(value=0.0)

# Turn off CH1
if dso.channel.is_on(ch=1):
      dso.channel.set_off(ch=1)

# Close the socket connection
dso. close ()
```

Controlling the Built-In Spectrum Analyzer

The following example will demonstrate how to control a spectrum analyzer, configure parameters such as the source and frequency, and set the frequency range, and other parameters.

Python code
```
# Import the DSO module and open a socket connection.
import gds_info as gds
dso=gds.Dso()
dso.connect()

# Activate the spectrum mode.
if not dso.sa.is_spectrum_mode():
   dso.sa.set_spectrum_mode('ON')

# Activate the input source.
if not dso.sa.get_state(_id=1):
   dso.sa.set_state(state='ON',_id=1)

# Activate the spectrum trace.
if not dso.sa.is_spectrum_trace(trace_type='NORMAL',_id=1):
   dso.sa.set_spectrum_trace(trace_type='NORMAL',state='ON',_id=1)

# set the center frequency to 25 MHz.
dso.sa.set_freq(freq=25e6,_id=1)
# get the center frequency.
dso.sa.get_freq(_id=1)

# set the span frequency to 25 MHz.
dso.sa.set_span(freq=25e6,_id=1)
# get the span frequency.
dso.sa.get_span(_id=1)

# set the start frequency to 12.5 MHz.
dso.sa.set_start(freq=12.5e6,_id=1)
# get the start frequency.
dso.sa.get_start(_id=1)
```

```
# set the stop frequency to 37.5 MHz.
dso.sa.set_stop(freq=37.5e6,_id=1)
# get the stop frequency.
dso.sa.get_stop(_id=1)

# Set the RBW value in the manual mode.
dso.sa.set_RBW_Manual(rbw=2.5e4, _id=1)

# Set the "Span:RBW " in the auto-RBW mode.
dso.sa.set_Span2RBW_Ratio(ratio='RATIO_1K',_id=1)

# Set the window type.
dso.sa.set_window(window=2,_id=1)

# Set the vertical scale and unit.
dso.sa.set_scale(scale=2.0e1,unit=0,_id=1)

# Set the zero level position.
dso.sa.set_position(position=3.0e0,_id=1)

# Deactivate the spectrum mode.
if dso.sa.is_spectrum_mode():
    dso.sa.set_spectrum_mode('OFF')
```

Controlling the Built-In AWG

The following example will demonstrate how to control an arbitrary waveform generator and configure parameters such as waveform, frequency, amplitude, and more.

Python code
```
# Import the DSO module and open a socket connection.
import gds_info as gds
dso=gds.Dso()
dso.connect()

# Activate the AWG(channel 1), and set waveform, frequency,
amplitude, offset at the same time.
if not dso.awg.is_on(ch=1):
   dso.awg.set_on(ch=1,wave='SINE',freq=1e5,amp=2.5e-1,offset=0e0)

# Set the load to HighZ.
dso.awg.set_load_highz(ch=1)

# Set the load to 50 ohm.
dso.awg.set_load_50ohm(ch=1)

# Set the phase to 0.0.
dso.awg.set_phase(ch=1,value=0.0)

# Deactivate the AWG(channel 1).
if dso.awg.is_on(ch=1):
   dso.awg.set_off(ch=1)
```

Controlling the Built-In DMM

The following examples will demonstrate how to control a digital multimeter and configure various measurement modes.

Python code
```
# Import the DSO module and open a socket connection.
import gds_info as gds
dso=gds.Dso()
dso.connect()

# Activate the DMM.
if not dso.dmm.is_on():
   dso.dmm.set_on()

# Set DMM's mode as ACV.
dso.dmm.set_mode_ACV(range='AUTO')
# Get the DMM's measurement result.
dso.dmm.get_value()

# Set DMM's mode as DCA.
dso.dmm.set_mode_DCA()
# Get the DMM's measurement result.
dso.dmm.get_value()

# Set DMM's mode as temperature.
dso.dmm.set_mode_temperature(type='TYPEK', units='C', sim=23)
# Get the DMM's measurement result.
dso.dmm.get_value()

# Deactivate the DMM.
if dso.dmm.is_on():
    dso.dmm.set_off()
```

Controlling the Built-In DC Power Units

The following examples will demonstrate how to control and configure voltage and other parameters.

Python code
```
# Import the DSO module and open a socket connection.
import gds_info as gds
dso=gds.Dso()
dso.connect()

# Set 5V on channel 1.
dso.power.set_voltage(ch=1, volt=5)

# Turn on the power supply.
if not dso.power.is_on():
   dso.power.set_on()

# Reconfigure the power supply when OCP occurred on channel 1.
if (dso.power.check_ocp(1)):
   dso.power.clear_ocp(1)

# Turn off the power supply.
if dso.power.is_on():
    dso.power.set_off()
```

Control Method of GO-NOGO Output Pin

The Go/NoGo function of MPO-2000 features an open-collector output pin, which can serve as a status indicator for notifying external circuits when a NoGo violation event occurs. When using it, this output pin should be connected in series with a pull-up resistor to an external voltage source. The voltage level of the external voltage source must consider the voltage range that the connected external circuit IO pin can withstand.

We also provide Python script control for this output pin. With the premise that the Go/NoGo output pin is connected to an external voltage source through a pull-up resistor, you can use **dso.gonogo.output_on()** to set the collector electrode to a conducting state to ground (the Go/NoGo output pin will be at a low voltage). You can use **dso.gonogo.output_off()** to set the collector electrode to a non-conducting state to ground (the Go/NoGo output pin will be at a high voltage from the external voltage source). You can also use **dso.gonogo.is_output_on()** to query the current configuration status. Please refer to the following example.

<mark>Python code</mark>

```
# Import the DSO module and open a socket connection.
import gds_info as gds
dso=gds.Dso()
dso.connect()

# Turn on the output.
if not dso.gonogo.is_output_on():
    dso.gonogo.output_on()

# Turn off the output.
if dso.gonogo.is_output_on():
    dso.gonogo.output_off()
```

Control of Connected External Devices

This oscilloscope can control external devices through a network interface via socket connection, such as GWInstek's PSW、PFR、PPX、PEL and other series of devices. It can also communicate with the aforementioned devices via the USB host interface, which supports the USB CDC-ACM protocol. When communicating with external devices, it's important to note that the transmission of commands and data may take some time. It's advisable to add brief delays between commands to ensure that the external devices receive the commands and execute the corresponding actions, which can enhance the stability of the system.

Control of connected external devices is achieved based on SCPI commands. Currently, only common commands for some devices are provided. Users can refer to the Programming Manual of their devices to find SCPI commands for various functions. In cases where Python modules do not provide specific commands, users can use the *write()* function to send SCPI commands to the device, and the *query()* function can be used to read the responses to inquiries.

Simple Method for Connecting External USB Devices

Using the PSW Module

The use of the *psw* module enables easy connection to power supplies supporting the USB CDC-ACM protocol, and this module is compatible with devices from the PSW、PFR and PPX series.

Python code
```python
import psw
PSW_SN = 'GEW192100'# Device serial number

if __name__ == '__main__':
    inst = psw.Psw()
    inst.connect(PSW_SN)
    print(inst.idn())

    inst.set_on()
    print(inst.is_on())
    inst.set_off()

    inst.close()
```

Output
```
GW-INSTEK,PSW160-14.4,GEW192100,02.53.20220419
True
```

Control External Devices with SCPI Commands

The *psw* module does not implement control commands for all functions. Users can control external devices by sending SCPI commands. Please refer to the Programming Manual of the respective device for a list of SCPI commands for each function.

Python code
```
import psw
PSW_SN = 'GEW192100'  # Device serial number

if __name__ == '__main__':
    inst = psw.Psw()
    inst.connect(PSW_SN)
    print(inst.query('*idn?'))

    inst.write(':OUTP ON')
    print(inst.query('OUTP?'))
    inst.write(':OUTP OFF')

    inst.close()
```

Output
```
GW-INSTEK,PSW160-14.4,GEW192100,02.53.20220419
1
```

Further Learning with the Serial Module

Besides using the *psw* module, users can directly connect external
USB CDC-ACM devices through the *serial* module, but we need to
know the model and serial number of the connected USB device in
advance. The same method can also be used to control other devices
that support the USB CDC-ACM protocol.

Get External Device Model and Serial Number

If connected using the USB CDC-ACM protocol to an external
device, when the local underlying driver is operating correctly, the
corresponding device filename inside the machine will be ttyACM0
(if there are more than one devices, the numbers will increment
sequentially, such as ttyACM0, ttyACM1, ttyACM2...).

Python code
```
Import serial
if __name__=='__main__':
    ttystr = '/dev/ttyACM0'  # External USB device location
    try:
        acm = serial.Serial(ttystr, baudrate=115200, timeout=3)
        acm.write('*idn?\n')
        str1 = acm.read(100).decode().split(',')
        print(str1)
    except:
        print('Serial Connection Error!')
        sys.exit()
    sys.exit()
```

Output
'GW-INSTEK', 'PSW160-14.4', 'GEW192100', '02.53.20220419\n']

The 'PSW160-14.4' is the device model, and 'GEW192100' is the
device serial number.

If you are connecting an external device with an RS232 interface using an RS232 to USB cable, please note that the baudrate parameter in the script should be set to match the baud rate setting of the RS232 interface on the device. The device filename should be modified to 'ttyUSB0'.

Python code
```
import serial
if __name__=='__main__':
   ttystr = '/dev/ttyUSB0'  # External USB device location
   try:
     usb = serial.Serial(ttystr, baudrate=115200, timeout=3)
     usb.write('*idn?\n')
     str1 = usb.read(100).decode().split(',')
     print(str1)
   except:
     print('Serial Connection Error!')
     sys.exit()
   sys.exit()
```

Output
```
['GWInstek', 'GDM8261A', 'EN121422', '1.01\r\n']
```

The 'GDM8261A' is the device model, and 'EN121422' is the device serial number.

Connect to External Devices via USB CDC-ACM Protocol

The following diagram illustrates the connection

After reading the device model, you can connect to PSW using the following methods.

Python code
```
import serial
PSW_NAME = 'PSW160-14.4'# Device name
ACM_MAX = '9'  # Search device from ACM0 to ACM_MAX

def ACM_Connect(device_name, device_sn=False):
   global acm
   numACMMAX = int(ACM_MAX)
   numACM = 0
   while(True):
      if numACM > numACMMAX:
         print('%s not found!' % (device_name))
         sys.exit()
      ttystr = '/dev/ttyACM' + str(numACM)
      print('Searching...',ttystr)
      try:
         acm = serial.Serial(ttystr, baudrate=115200, timeout=3)
      except:
         print('Serial Connection Error!')
         sys.exit()

      acm.write('*idn?\n')
      str1 = acm.read(100).decode().split(',')
      if len(str1) > 1:
         if device_sn != False:
            if str1[1] == device_name and str1[2] == device_sn:
```

```
                print('Connected with ', str1[1], str1)
                break
          else:
             if str1[1] == device_name:
                print('Connected with ', str1[1], str1)
                break
       numACM += 1
    return acm

  if __name__ == '__main__':
    ACM_MAX = '9'# Search device from ACM0 to ACM_MAX
    device = ACM_Connect(PSW_NAME) # Connected with external device
    device.write('*idn?\n')
    str = device.read(100).decode().split(',')
    print(str[1])
```

Output

```
 Searching... /dev/ttyACM0
 Connected with PSW160-14.4 ['GW-INSTEK', 'PSW160-14.4',
 'GEW192100', '02.53.20220419\n']
 PSW160-14.4
```

Connected to External Devices via the RS232 Interface.

As mentioned in the previous section, devices connected through an RS232 interface can be used with an RS232 to USB cable. Please ensure that the baud rate parameter in the script is set to match the RS232 interface baud rate of the device. Also, modify the device filename to 'ttyUSB0'. Considering the connection of multiple RS232 devices, here we have a more comprehensive script example, and the following diagram illustrates the connection:

Python code
```python
import serial
USB_MAX = '9'  # Search device from USB0 to USB_MAX
GDM_NAME = 'GDM8261A'  # Device name

def USB_Connect(device_name, device_sn=False):
    global usb
    numUSBMAX = int(USB_MAX)
    numUSB = 0
    while(True):
        if numUSB > numUSBMAX:
            print('%s not found!' % (device_name))
            sys.exit()
        ttystr = '/dev/ttyUSB' + str(numUSB)
        print('Searching...',ttystr)
        try:
            usb = serial.Serial(ttystr, baudrate=115200, timeout=3)
        except:
            print('Serial Connection Error!')
            sys.exit()
```

```
        usb.write('*idn?\n')
        str1 = usb.read(100).decode().split(',')
        if len(str1) > 1:
            if device_sn != False:
                if str1[1] == device_name and str1[2] == device_sn:
                    print('Connect to', str1[1], str1)
                    break
            else:
                if str1[1] == device_name:
                    print('Connect to', str1[1], str1)
                    break
        numUSB += 1
    return usb

if __name__ == '__main__':
    device = USB_Connect(GDM_NAME) #Connect to Instrument
    device.write('*idn?\n')
    str = device.read(100).decode().split(',')
    print(str[1])
```

Output

```
Searching... /dev/ttyUSB0
Connect to GDM8261A ['GWInstek', 'GDM8261A', 'EN121422', '1.01\r\n']
GDM8261A
```

Connecting Multiple External Devices of Different Models

The following diagram illustrates the connection.

To connect multiple external devices through a USB hub, it is necessary to read the model of each device before proceeding with the connection. When attempting to connect multiple USB devices, it is also important to note that both the USB hub and each USB device require power from the oscilloscope's USB host controller. If the power supplied from the USB host controller reaches a critical level, it may lead to unstable data transmission in the USB connection.

Python code

```
import serial
USB_MAX = '9'  # Search device from USB0 to USB_MAX
GDM_NAME = 'GDM8261A'# Device name
ACM_MAX = '9'  # Search device from ACM0 to ACM_MAX
PSW_NAME = 'PSW160-14.4'  # Device name

if __name__ == '__main__':
   GDM = USB_Connect(GDM_NAME)  #Connect to Instrument
   PSW = ACM_Connect(PSW_NAME)  #Connect to Instrument

   GDM.write('*idn?\n')
   str = GDM.read(100).decode().split(',')
   GDM_str = str[1]
```

```
PSW.write('*idn?\n')
str = PSW.read(100).decode().split(',')
PSW_str = str[1]

print(GDM_str, PSW_str)
```

Output
Searching... /dev/ttyUSB0
Connect to GDM8261A ['GWInstek', 'GDM8261A', 'EN121422', '1.01\r\n']
Searching... /dev/ttyACM0
Connect to PSW160-14.4 ['GW-INSTEK', 'PSW160-14.4', 'GEW192100',
'02.53.20220419\n']
GDM8261A PSW160-14.4

USB_Connect() function please refer to page 81

ACM_Connect() function please refer to Page 79

Connecting Multiple External Devices of the Same Model

The following diagram illustrates the connection.
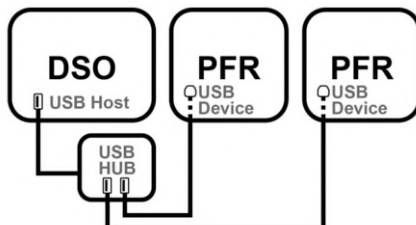
If multiple devices of the same model are connected, it is also necessary to determine the serial number of the machine to ensure that voltage, current, and other configuration commands are sent to the correct USB device. This avoids confusion in controlling the target and prevents any power supply errors that could lead to damage to the test object.

Python code

```
import serial
ACM_MAX = '9'  # Search device from ACM0 to ACM_MAX
PFR_NAME = 'PFR-100M'  # Device name
PFR1_SN = 'GER200751'  # Device serial number
PFR2_SN = 'GER200718'  # Device serial number

if __name__ == '__main__':
    PFR1 = ACM_Connect(PFR_NAME, PFR1_SN)  #Connect to Instrument
    PFR2 = ACM_Connect(PFR_NAME, PFR2_SN)  #Connect to Instrument

    PFR1.write('*idn?\n')
    str = PFR1.read(100).decode().split(',')
    PFR1_str = str[1]+' '+str[2]

    PFR2.write('*idn?\n')
    str = PFR2.read(100).decode().split(',')
    PFR2_str = str[1]+' '+str[2]
```

```
print('1. '+PFR1_str)
print('2. '+PFR2_str)
```

Output

```
Searching... /dev/ttyACM0
Searching... /dev/ttyACM1
Connect to PFR-100M ['GW-INSTEK', 'PFR-100M', 'GER200751',
'01.32.20221031\n']
Searching... /dev/ttyACM0
Connect to PFR-100M ['GW-INSTEK', 'PFR-100M', 'GER200718',
'01.32.20221031\n']
1. PFR-100M GER200751
2. PFR-100M GER200718
```

ACM_Connect() function please refer to page 79

Graphical User Interface in Python

Introduction to LVGL

LVGL (Light and Versatile Graphics Library) is a free and open-source graphics library licensed under the MIT license. It provides a variety of user interface components for easy interface design, such as buttons, labels, images, lists, charts, text areas, and more. It also offers various event handling mechanisms, including mouse clicks and drag-and-drop processing, keyboard input, and more. This makes it convenient for users to create intuitive graphical interfaces, and it can run on various embedded platforms. We have ported LVGL to the MPO-2000 system, allowing users to construct their own test charts and user interfaces under the control of Python scripts. Please note that graphical user interface script development is only available on MPO-2000P, and it is not possible to develop this type of script on the Basic version.

Several key core elements of LVGL include:

1. **Objects**
   LVGL provides various objects such as buttons, labels, lists, input fields, and more. These objects can be placed on the screen and interacted with by users. Each object has its own attributes and methods that can be customized as needed.

2. **Graphic Styles**
   Graphic styles define the appearance and behavior of graphic elements, including colors, fonts, borders, and more. Developers can easily change their visual effects.

3. **Event Handling**

   LVGL provides an event handling mechanism. When users interact with objects, such as pressing a button, the corresponding events are triggered. Developers can write event handlers to respond to these events.

Regarding Python script design for MPO-2000, invoking the LVGL library falls under advanced usage. Users with general scripting skills can practice by attempting to modify the sample scripts we provide, extensively reading relevant Python literature, and conducting practical exercises to enhance their design abilities. For the actual construction of small-scale automated testing systems, it is recommended to involve skilled software engineers with expertise in Python. Using the LVGL library involves a certain level of complexity, and the graphical interface consumes a significant amount of memory. To ensure long-term stability and prevent memory leaks during script execution, it is essential to avoid continuous memory leaks that can lead to errors or crashes.

LVGL Basic Examples

Next, we will illustrate how to use LVGL for graphical interface scripting by providing a few simple examples that demonstrate the presentation of commonly used text, images, and charts.

Please note that, after calling the GUI library to draw graphs or text, you need to allow the system a brief moment to complete the screen update. This is important for short scripts that only draw once and then exit, where you can add a time.sleep(0.1) delay after the script drawing. Otherwise, it's possible that the screen updates halfway and the script stops running.

LVGL Initialization

Before performing any drawing operations, it is necessary to initialize LVGL and the display screen. This includes initializing LVGL, the framebuffer, setting up the display buffer, and registering the display driver.

Subsequent examples will omit this initialization segment!

Python code

```python
import lvgl as lv
import fb

if __name__ == '__main__':

###############################
#          GUI Initialization          #
###############################

    # Initialize LVGL and Framebuffer
```

```
lv.init()
fb.init()

# Set screen width and height
screen_width = 800
screen_height = 480

# Create the display buffer
disp_buf = lv.disp_draw_buf_t()

# Set the size of the display buffer
buf = bytes(screen_width*screen_height*3)

# Initialize the display buffer in 32-bit units
disp_buf.init(buf, None, len(buf)//4)

# Register FB display driver
disp_drv = lv.disp_drv_t()
disp_drv.init()
disp_drv.draw_buf = disp_buf
disp_drv.flush_cb = fb.flush
disp_drv.hor_res = screen_width
disp_drv.ver_res = screen_height
disp_drv.register()
```

⚠️ Note: If there are any display anomalies, you can try connecting to the oscilloscope before initiating the drawing.

Python code
```
import gds_info as gds
dso = gds.Dso()
dso.connect()
```

Text Display

In this example, we will demonstrate how to display the text string 'Hello World' on the screen.

(LVGL_HelloWorld.py)
Python code

```python
import lvgl as lv
import fb
import time

if __name__ == '__main__':

    ################################
    #          GUI Initialization          #
    ################################

    # Please refer to LVGL Initialization
    ################################
    #              Hello World              #
    ################################

    # Create a label object
    label = lv.label(lv.scr_act())

    # Set the position of the label
    label.set_pos(360, 220)

    # Set the text of the label
    label.set_text('Hello World')
    time.sleep(0.1)
```

Text Rotation
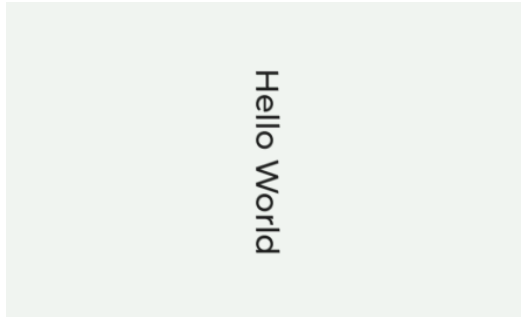
In this example, we will demonstrate how to rotate a string and set its rotation angle.

(LVGL_TextRotation.py)
Python code
```
import lvgl as lv
import fb
import time

if __name__ == '__main__':

    ################################
    #              GUI Initialization           #
    ################################
    # Please refer to LVGL Initialization

    ################################
    #              Set the Text                  #
    ################################

    # Create a label object
    label = lv.label(lv.scr_act())

    # Set the position of the label
    label.set_pos(400, 200)
```

```
# Set the text of the label
label.set_text('Hello World')

#################################
#             Text Rotation              #
#################################

# Set rotation angle to 90 degree
label.set_style_transform_angle(900, lv.PART.MAIN)
time.sleep(0.1)
```

Style Configuration
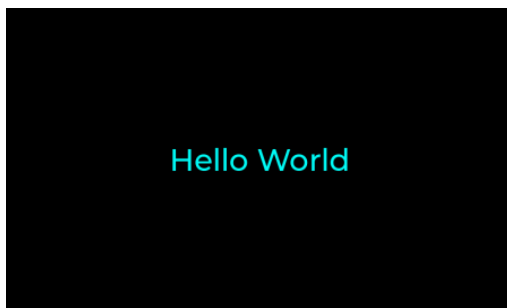
In this example, we will demonstrate how to change the background and text colors. All color settings should be entered in hexadecimal representation (RGB color codes). You can also import the *dso_colors* module to use the basic colors we provide, for instance:

Python code
```
import dso_colors as color
style.set_bg_color(color.BLACK)
```

(LVGL_Style.py)
Python code
```
import lvgl as lv
import fb
import time

if __name__ == '__main__':

###############################
#          GUI Initialization           #
###############################

# Please refer to LVGL Initialization
###############################
#            Create a style              #
###############################
```

```
# Get the active screen object
obj = lv.scr_act()

# Create a style for the screen
style = lv.style_t()
style.init()
obj.add_style(style, 0)

# Set the background color of the style
style.set_bg_color(lv.color_hex(0x000000))

################################
#            Create a label            #
################################


# Create a label object
label = lv.label(lv.scr_act())

# Set the position of the label
label.set_pos(360, 220)

# Set the text color of the label
label.set_style_text_color(lv.color_hex(0x00FFFF), lv.STATE.DEFAULT)

# Set the text of the label
label.set_text('Hello World')
time.sleep(0.1)
```

Displaying PNG Images
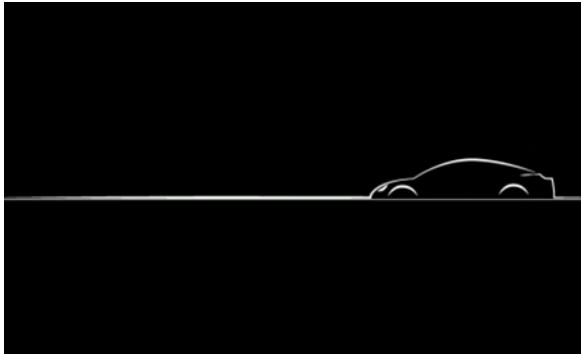
In this example, we will demonstrate how to read a PNG file and display it on the screen.

(LVGL_Image.py)
Python code

```python
import lvgl as lv
import fb
import os
import sys

if __name__ == '__main__':

    ################################
    #          GUI Initialization         #
    ################################
    # Please refer to LVGL Initialization

    ################################
    #          Displaying Image           #
    ################################

    # Set the background color
    obj = lv.scr_act()
    style = lv.style_t()
    style.init()
```

```
obj.add_style(style, 0)
style.set_bg_color(lv.color_hex(0x000000))

# Set the current working directory to the directory where the script is located
os.chdir(sys.path[0])

# Read the image file and display it on the screen
with open(LVGL_Image.png','rb') as f:
  # Read the PNG data from the file
  png_data = f.read()

# Create an image descriptor
png_img_dsc = lv.img_dsc_t({
   'data_size': len(png_data),
   'data': png_data
})

# Create an image object and set the image
img = lv.img(lv.scr_act())
img.align(lv.ALIGN.CENTER, 0, 0)
img.set_src(png_img_dsc)

# Process LVGL tasks
while True:
   lv.task_handler()
```

Simple Line Chart
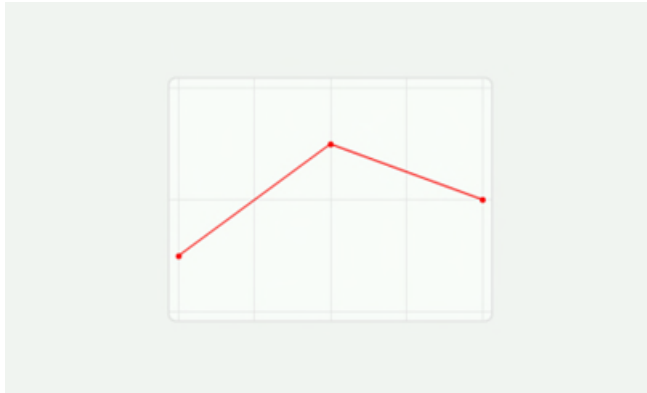
Due to the increased memory consumption when invoking the graphics library for drawing, currently, the system can draw a maximum of 50,000 data points in a line chart. If the data volume exceeds this limit, it can be drawn by selecting specific data points or averaging multiple sampling points before plotting. In this example, we will demonstrate how to draw a simple line chart.

(LVGL_Chart.py)

Python code

```
import lvgl as lv
import fb
import time

if __name__ == '__main__':

    ################################
    #            GUI Initialization            #
    ################################
    # Please refer to LVGL Initialization

    ################################
    #              Draw a Chart              #
    ################################

    # Create a chart
    chart = lv.chart(lv.scr_act())
    chart.set_size(400,300)
    chart.align(lv.ALIGN.CENTER,0,0)
    chart.set_type(lv.chart.TYPE.LINE)

    # Add data series
    ser=chart.add_series(lv.color_hex(0xFF0000), lv.chart.AXIS.PRIMARY_Y)

    # Set the number of points, default is 10
    chart.set_point_count(3)

    # Set points on ser
    ser.y_points = [25, 75, 50]

    # Update the chart
    chart.refresh()
    time.sleep(0.1)
```

Line Charts and Scales
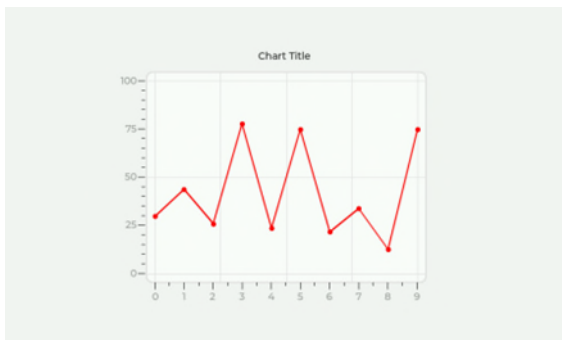
In this example, we will demonstrate how to add scales and titles to a line chart, and explain the parameter definitions of the scale functions.

(LVGL_ChartTick.py)
Python code

```
import lvgl as lv
import fb
import time

if __name__ == '__main__':

    ###############################
    #          GUI Initialization          #
    ###############################

    # Please refer to LVGL Initialization
    ###############################
    #            Create a Chart             #
    ###############################

    # create a chart and set points
    chart = lv.chart(lv.scr_act())
    chart.set_size(400,300)
    chart.align(lv.ALIGN.CENTER,0,0)
    chart.set_type(lv.chart.TYPE.LINE)
    ser=chart.add_series(lv.color_hex(0xFF0000), lv.chart.AXIS.PRIMARY_Y)
```

```python
ser.y_points = [30, 44, 26, 78, 24, 75, 22, 34, 13, 75]
################################
#           Set the ticks and texts           #
################################

chart.set_axis_tick(
    lv.chart.AXIS.PRIMARY_X,  #Choose to set the x or y-axis
    10,  # Set the length of the major tick
    5,   # Set the length of the minor tick
    10,  # Set the total number of major ticks on the x-axis
    2,  # Set the number of intervals between major tick on the x-axis
    True, # Set whether to display tick labels
    50  # Set the length of the tick label
)

chart.set_axis_tick(
    lv.chart.AXIS.PRIMARY_Y,  #Choose to set the x or y-axis
    10,  # Set the length of the major tick
    5,  # Set the length of the minor tick
    5,  # Set the total number of major ticks on the y-axis
    5,  # Set the number of intervals between major tick on the y-axis
    True, # Set whether to display tick labels
    50  # Set the length of the tick label
)

# Update the chart
chart.refresh()

################################
#               Chart title               #
################################

    # Create a label object
    label = lv.label(lv.scr_act())

    # Set the position of the label
     label.set_pos(360, 60)

    # Set the text of the label
    label.set_text('Chart Title')
    time.sleep(0.1)
```

Text Area
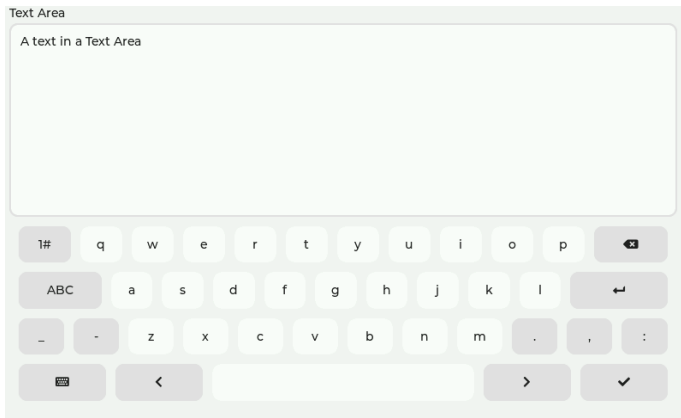
In this example, we will demonstrate how to create a text area and add text to it by clicking on a virtual keyboard with a USB mouse or using a USB keyboard.

(LVGL_TextArea.py)
<mark>Python code</mark>

```
import lvgl as lv
import fb

if __name__ == '__main__':

    ###############################
    #         GUI Initialization          #
    ###############################
    # Please refer to LVGL Initialization

    ###############################
    #              Text Area               #
    ###############################

    LV_HOR_RES = 780
    LV_VER_RES = 450
```

```
# Create the text area
ta = lv.textarea(lv.scr_act())
ta.set_text("A text in a Text Area")
ta.set_one_line(False)
ta.align(lv.ALIGN.OUT_TOP_LEFT, 5, 20)
ta.set_size(LV_HOR_RES, LV_VER_RES // 2)
ta.add_state(lv.STATE.FOCUSED)   # To be sure the cursor is visible
ta.set_max_length(10000)  # The maximum number of characters

# Create a label above the text box
label = lv.label(lv.scr_act())
label.set_text("Text Area")
label.align(lv.ALIGN.OUT_TOP_LEFT, 5, 0)

# Create a keyboard
kb = lv.keyboard(lv.scr_act())
kb.set_size(LV_HOR_RES, LV_VER_RES // 2)
kb.align(lv.ALIGN.OUT_TOP_LEFT, 5, LV_VER_RES // 2 + 20)
kb.set_textarea(ta)

#################################
#             Mouse Device              #
#################################

# Registering a mouse input device
import evdev
indev_mouse = lv.indev_drv_t()
indev_mouse.init()
indev_mouse.type = lv.INDEV_TYPE.POINTER
indev_mouse.read_cb = evdev.mouse_indev().mouse_read
indev_mouse.register()

#################################
#            Keyboard Device             #
#################################

# Registering a keyboard input device
import dso_evdev
indev_kb = lv.indev_drv_t()
indev_kb.init()
indev_kb.type = lv.INDEV_TYPE.KEYPAD
```

```
kb_dev = dso_evdev.kb_indev()
indev_kb.read_cb = kb_dev.kb_read
indev_kb.register()


################################
#          Keyboard Input Handle          #
################################

# Read keyboard input and display corresponding text or
# perform corresponding operations in the textarea
while True:
    key = kb_dev.get_keycode_mods()
    if key["keycode"]:
        keychar = kb_dev.key_to_char(key['shift'],key['keycode'])
        keyname = kb_dev.get_key_name(key["keycode"])
        if keychar is None:
            if keyname == 'KEY_BACKSPACE':
                lv.textarea.del_char(ta)
            elif keyname == 'KEY_DELETE':
                lv.textarea.del_char_forward(ta)
            elif keyname == 'KEY_UP':
                lv.textarea.cursor_up(ta)
            elif keyname == 'KEY_LEFT':
                lv.textarea.cursor_left(ta)
            elif keyname == 'KEY_RIGHT':
                lv.textarea.cursor_right(ta)
            elif keyname == 'KEY_DOWN':
                lv.textarea.cursor_down(ta)
        else:
            lv.textarea.add_text(ta, keychar)
```

Table

In this example, we will demonstrate how to create a table.

(LVGL_Table.py)
<mark>Python code</mark>

```
import lvgl as lv
import fb
import time

if __name__ == '__main__':
    ################################
    #          GUI Initialization          #
    ################################
    # Please refer to LVGL Initialization

    ################################
    #                 Table                 #
    ################################

    # Set the background color
    obj = lv.scr_act()
    scr_style = lv.style_t()
    scr_style.init()
    obj.add_style(scr_style, 0)
    scr_style.set_bg_color(lv.color_hex(0x000000))

    # Create the table
```

```
table = lv.table(lv.scr_act())
table.set_col_cnt(3)
table.set_row_cnt(4)
table.set_size(480,230)
table.align(lv.ALIGN.CENTER, 0, 0)

# Set column widths
table.set_col_width(0,120)
table.set_col_width(1,120)
table.set_col_width(2,120)

# Set cell values
table.set_cell_value(0, 0, "Product")
table.set_cell_value(1, 0, "Quantity")
table.set_cell_value(2, 0, "Price")
table.set_cell_value(3, 0, "Total")

table.set_cell_value(0, 1, "Apple")
table.set_cell_value(1, 1, "30")
table.set_cell_value(2, 1, "10")
table.set_cell_value(3, 1, "300")

table.set_cell_value(0, 2, "Banana")
table.set_cell_value(1, 2, "15")
table.set_cell_value(2, 2, "5")
table.set_cell_value(3, 2, "75")

table.set_cell_value(0, 3, "Orange")
table.set_cell_value(1, 3, "20")
table.set_cell_value(2, 3, "3")
table.set_cell_value(3, 3, "60")

# Apply style to the table
style = lv.style_t()
style.init()
style.set_border_side(lv.BORDER_SIDE.LEFT | lv.BORDER_SIDE.RIGHT |
            lv.BORDER_SIDE.TOP | lv.BORDER_SIDE.BOTTOM)
style.set_border_color(lv.color_hex(0x000000))
style.set_bg_color(lv.color_hex(0xFFFFFF))
table.add_style(style, lv.STATE.DEFAULT | lv.PART.ITEMS)
time.sleep(0.1)
```

Buttons and Switches
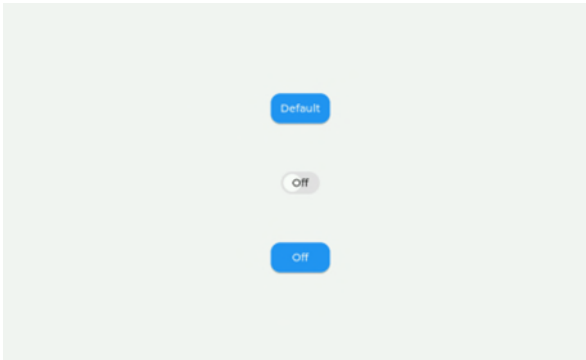
In this example, we will demonstrate how to create buttons and switches, and generate interactive effects by clicking with a USB mouse.

Here, we will create two different types of buttons: "General" and "Toggle". Clicking the Toggle button will produce an effect of turning it on (orange) and off (blue). The actions of the Toggle and General buttons are essentially the same, with only a difference in appearance. Finally, clicking the General button will turn off both the Toggle button and the Switch. In addition, event handling has been added to trigger corresponding actions for the buttons. Refer to the script below to learn how to read or change the button states.

(LVGL_Button.py)

```
import lvgl as lv
import fb

if __name__ == '__main__':
  ###############################
  #              GUI Initialization              #
  ###############################
  # Please refer to LVGL Initialization

  ###############################
  #              Create a Button              #
  ###############################

  def event_handler(evt):
    event_type = evt.get_code()
    target = evt.get_target()
    list_onoff = {True: "On", False: "Off"}

    if event_type == lv.EVENT.CLICKED:
      btn_toggle.clear_state(lv.STATE.CHECKED)
      label_btn_toggle.set_text(list_onoff[btn_toggle.has_state(lv.STATE.CHECKED)])
      sw.clear_state(lv.STATE.CHECKED)
      label_sw.set_text(list_onoff[sw.has_state(lv.STATE.CHECKED)])
    elif event_type == lv.EVENT.VALUE_CHANGED:
      if type(target) == type(btn_toggle):
        state = target.has_state(lv.STATE.CHECKED)
        label_btn_toggle.set_text(list_onoff[state])
      elif type(target) == type(sw):
        state = target.has_state(lv.STATE.CHECKED)
        label_sw.set_text(list_onoff[state])

  # Create a simple button
  btn_simple = lv.btn(lv.scr_act())
  btn_simple.add_event_cb(event_handler, lv.EVENT.CLICKED, None)
  btn_simple.set_height(40)
  btn_simple.set_width(80)
  btn_simple.align(lv.ALIGN.CENTER, 0,-100)
  label_btn_simple = lv.label(btn_simple)
  label_btn_simple.center()
```

```python
label_btn_simple.set_text("Default")

# Create a toggle button
btn_toggle = lv.btn(lv.scr_act())
btn_toggle.add_event_cb(event_handler, lv.EVENT.VALUE_CHANGED, None)
btn_toggle.add_flag(lv.obj.FLAG.CHECKABLE)
btn_toggle.set_height(40)
btn_toggle.set_width(80)
btn_toggle.align(lv.ALIGN.CENTER, 0, 100)
label_btn_toggle = lv.label(btn_toggle)
label_btn_toggle.center()
label_btn_toggle.set_text("Off")

# Create a switch
sw = lv.switch(lv.scr_act())
sw.add_event_cb(event_handler, lv.EVENT.VALUE_CHANGED, None)
sw.center()
label_sw=lv.label(sw)
label_sw.center()
label_sw.set_text("Off")

#################################
#            Mouse Device       #
#################################

# Registering a mouse input device
import evdev
indev_mouse = lv.indev_drv_t()
indev_mouse.init()
indev_mouse.type = lv.INDEV_TYPE.POINTER
indev_mouse.read_cb = evdev.mouse_indev().mouse_read
indev_mouse.register()

while True:
    pass
```

Progress Bars and Sliders
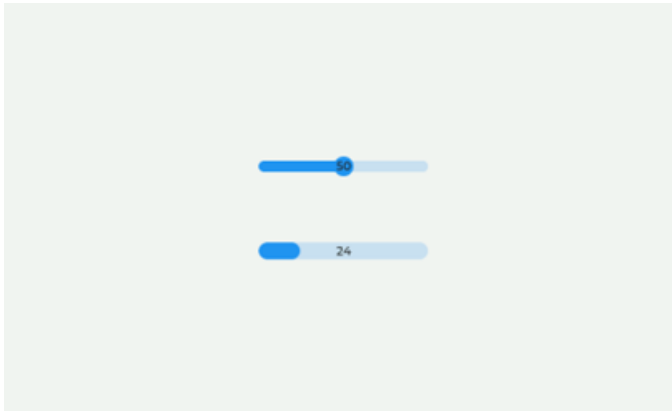
In this example, we will demonstrate how to create a progress bar and a slider.

We have set up two forms of progress bars, divided into regular and range progress bars. The regular progress bar is commonly used to indicate the completion status of a process, while the range progress bar differs in that it allows you to set the starting position of the progress bar slider. This is often used to indicate that a process is in progress.

The slider can be dragged and moved using a USB mouse and also requires adding event handling procedures to handle the corresponding actions generated by the sliding.

(LVGL_Bar.py)

Python code

```
import lvgl as lv
import fb
import time

if __name__ == '__main__':
    ##################################
    #           GUI Initialization           #
    ##################################
    # Please refer to LVGL Initialization

    ##################################
    #             Create a Bar              #
    ##################################

    def event_handler(evt):
        event_type = evt.get_code()
        target = evt.get_target()

        if event_type == lv.EVENT.VALUE_CHANGED:
            value = target.get_value()
            label_slider.set_text(str(value))

    # Create a bar
    bar = lv.bar(lv.scr_act())
    bar.set_size(200, 20)
    bar.align(lv.ALIGN.CENTER, 0, 50)
    bar.set_range(0, 100)
    bar.set_value(0, lv.ANIM.OFF)
    bar.set_start_value(0, lv.ANIM.OFF)
    label_bar = lv.label(lv.scr_act())
    label_bar.set_text(str(bar.get_value()))
    label_bar.align(lv.ALIGN.CENTER, bar.get_x_aligned(), bar.get_y_aligned())

    # Create a slider
    slider = lv.slider(lv.scr_act())
    slider.add_event_cb(event_handler, lv.EVENT.VALUE_CHANGED, None)
    slider.set_width(200)
    slider.set_range(0, 100)
    slider.set_value(50, lv.ANIM.OFF)
```

```
slider.align(lv.ALIGN.CENTER,0, -50)
label_slider = lv.label(lv.scr_act())
label_slider.set_text(str(slider.get_value()))
label_slider.align(lv.ALIGN.CENTER, slider.get_x_aligned(), slider.get_y_aligned())


##################################
#              Mouse Device           #
##################################

# Registering a mouse input device
import evdev
indev_mouse = lv.indev_drv_t()
indev_mouse.init()
indev_mouse.type = lv.INDEV_TYPE.POINTER
indev_mouse.read_cb = evdev.mouse_indev().mouse_read
indev_mouse.register()


##################################
#              Progress Bar            #
##################################

def progress_bar():
    if bar.get_mode() == lv.bar.MODE.NORMAL:
        for value in range(0, 101):
            bar.set_value(value, lv.ANIM.OFF)
            label_bar.set_text(str(bar.get_value()))
            time.sleep(0.02)
    else:
        for value in range(0, 121):
            bar.set_start_value(value-20, lv.ANIM.ON)
            bar.set_value(value, lv.ANIM.ON)
            label_bar.set_text("")
            time.sleep(0.02)
    time.sleep(1)

while True:
    bar.set_mode(lv.bar.MODE.NORMAL)
    progress_bar()
    bar.set_mode(lv.bar.MODE.RANGE)
    progress_bar()
```

DSO Drawing Module

We provide two modules, *dso_gui* and *dso_colors*, to make it easier for users to draw on the oscilloscope display screen using LVGL. The *dso_gui* module includes functions on how to draw lines, curves, rectangles, text, images, and line charts, while the *dso_colors* module defines some commonly used colors.

Text and Styles
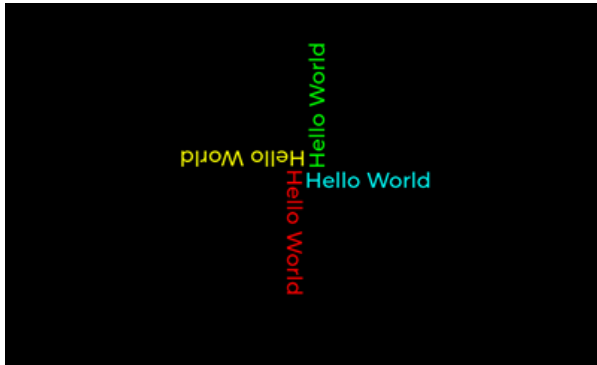
In this example, after changing the background color, we display 'Hello World' text in the center of the screen at four different angles (0, 90, 180, and 270 degrees) and colors.

(LVGL_gui_drawtext.py)
Python code
```
import dso_gui
import dso_colors as color

if __name__ == '__main__':
    # Initialize the GUI
    gui = dso_gui.DrawObject()
    gui.set_bg_color(color.BLACK)


    # Draw four "Hello World" labels with different colors and angles
    label1 = gui.draw_text(400, 220, "Hello World", color.LTCYAN)
    label2 = gui.draw_text(400, 220, "Hello World", color.LTRED, 90)
    label3 = gui.draw_text(400, 220, "Hello World", color.YELLOW, 180)
    label4 = gui.draw_text(400, 220, "Hello World", color.LTGREEN, 270)
```

Font
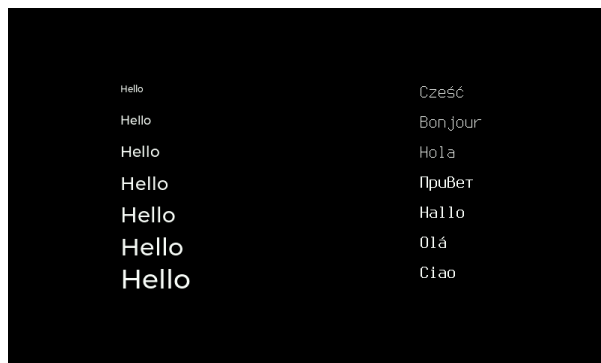
The current system defaults to using built-in fonts with 14 different font sizes: 10, 12, 14, 16, 18, 20, 22, 24, 28, 32, 36, 40, 44, and 48.

In addition to the built-in fonts, we have also included the Terminus font with 9 different font sizes: 12, 14, 16, 18, 20, 22, 24, 28, and 32, available in various weights, and it supports multiple languages.

(Terminus Font is licensed under the SIL Open Font License, Version 1.1)

Due to considerations regarding system resources, CJK fonts are currently not supported.

In the following examples, we have used built-in fonts of different sizes and loaded the Terminus font with different weights. As shown in the above image, the left side represents the system's built-in fonts, while the right side represents the Terminus font.

(LVGL_gui_font.py)

Python code

```python
import dso_gui
import dso_colors as color
import time

if __name__ == '__main__':
    # Initialize the GUI
    gui = dso_gui.DrawObject()
    gui.set_bg_color(color.BLACK)

    # Set different font sizes using built-in fonts
    for i in range(12, 37, 4):
        font = gui.set_font(i)
        if font:
            label = gui.draw_text(150, 100+(i/4-3)*40, 'Hello', color=color.WHITE, font=font)

    # Set font for specific languages using Terminus font
    font = gui.set_font(fnt_path='/home/upypr/lv_font/Terminus_24.fnt')
    if font:
        label = gui.draw_text(550, 100, 'Cześć', color=color.WHITE, font=font)
        label = gui.draw_text(550, 140, 'Bonjour', color=color.WHITE, font=font)
        label = gui.draw_text(550, 180, 'Hola', color=color.WHITE, font=font)

    # Set bold font for specific languages using Terminus-Bold font
    font = gui.set_font(fnt_path='/home/upypr/lv_font/TerminusB_24.fnt')
    if font:
        label = gui.draw_text(550, 220, 'Привет', color=color.WHITE, font=font)
        label = gui.draw_text(550, 260, 'Hallo', color=color.WHITE, font=font)
        label = gui.draw_text(550, 300, 'Olá', color=color.WHITE, font=font)
        label = gui.draw_text(550, 340, 'Ciao', color=color.WHITE, font=font)

    time.sleep(0.5)
```

PNG Images
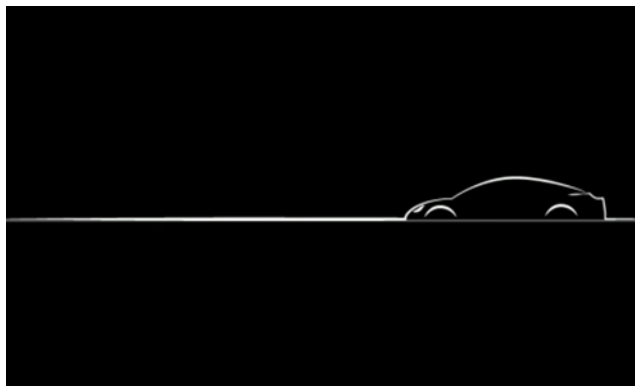
In this example, you can easily and quickly display PNG images once the GUI is initialized.

(LVGL_gui_image.py)
Python code

```
import os
import sys
import dso_gui

if __name__ == '__main__':
    # Initialize the GUI
    gui = dso_gui.DrawObject()

    # Set the current working directory to the directory where the script is located
    os.chdir(sys.path[0])

    # Display the PNG image on the screen
    gui.draw_png(0, 0, 'LVGL_Image.png')
```

Line Chart

In this example, the X-axis represents time, while the Y-axis represents the temperature of Samples A and B. The updating method for this line chart is to shift the old values to the left and add the new values to the right.

The chart's size and position can be customized, and the floating-point numbers for the scales can have up to three decimal places.

(LVGL_gui_chart.py)

```
import dso_gui
import gds_info as gds
import dso_colors as color
import time

if __name__ == '__main__':
    # Initialize the GUI
    gui = dso_gui.DrawObject()
    gui.set_bg_color(gds.Theme().bg_color)

    # Create figure object
    fig = gui.Plot(x=120, y=43, width=630, height=350)
    fig.grid(x_major=12, y_major=10, line_color=gds.Theme().grid_color,
        bg_color=gds.Theme().bg_color, x_minor=5, y_minor=5)
    fig.set_x_axis_on(text_color=gds.Theme().text_color,line_color=gds.Theme().grid_color,
    fmt='%d')
    fig.set_y_axis_on(text_color=gds.Theme().text_color,line_color=gds.Theme().grid_color,
    fmt='%.1f')

    # Add axis labels
    gui.draw_text(400, 440, "Time(h)", gds.Theme().text_color)
    gui.draw_text(30, 300, f"Temperature({chr(176)}C)", gds.Theme().text_color, 270)

    # Add legend
    gui.draw_text(310, 15, "Sample A", gds.Theme().text_color)
    gui.draw_line(270,25,300,25, color.LTGREEN)
    gui.draw_text(460, 15, "Sample B", gds.Theme().text_color)
    gui.draw_line(420,25,450,25, color.YELLOW)

    # Draw data point for Sample A
    valx = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    valy = [20.0, 23.4, 26.8, 29.5, 33.2, 37.6, 41.2, 45.7, 50.0, 56.1, 62.7, 62.5, 62.8]
    fig.plot(valx, valy, color=color.LTGREEN)

    # Draw data point for Sample B
    valy = [20.0, 26.2, 32.5, 38.3, 45.7, 52.4, 59.3, 67.8, 75.4, 75.1, 75.2, 75.6, 75.3]
    fig.plot(valx, valy, color=color.YELLOW)
    time.sleep(0.1)
```

Python Script Editing, Debugging, and Execution

Currently, MPO-2000 only provides simple debugging tools. Users can connect the PC and MPO-2000 in a local network. By accessing the internal micro web server of MPO-2000 through a web browser on the PC, users can perform script editing. Debugging can also be done using the *print()* function and the *REPL* (Read-Eval-Print Loop) environment.

print() function: This is the simplest debugging method. In the *WebIDE* environment, you can insert *print()* statements into the script to output the values of variables and other messages, thus identifying issues in the script. However, when executing Python scripts either from the Python APP menu or directly from disk or a USB drive, the output of *print()* will not be displayed.
REPL: The REPL (Read-Eval-Print Loop) environment allows users to engage in interactive script design and use the *print()* function to output variable values. We can enter a short script in the REPL environment, immediately view its execution result, and make necessary modifications.

Editing Using a Web Editor via Ethernet Connection

WebIDE

WebIDE is a web-based Python script debugging tool provided by MPO-2000, which integrates functions such as file management, text editor, and REPL terminal. Through *WebIDE*, users can edit and execute Python scripts within MPO-2000 on the same local network from their PCs, smartphones, or tablets, or quickly test Python scripts in the REPL terminal.

(This feature is an extension based on the architecture of github.com/vsolina/micropython-web-editor, which is a MIT licensed open-source project. And its author is Vsolina).

WebIDE Startup

In the following operations, we will use a web browser on a PC as an example for illustration.

1. **The Connection of PC and this Machines**
   First, please ensure that the machine is connected to a valid local area network (LAN). Users can configure the network settings in Utility -> I/O -> Network -> Ethernet. Please assign a valid IP address (Instrument IP Address) to the machine. This IP address should be on the same LAN as the device where you open the browser, or you can use DHCP to obtain it automatically.

   If you are not in a LAN environment, you can try another connection method. Connect the MPO-2000 directly to the device

where you want to open the browser using an Ethernet cable. Users can check and configure the machine's IP address (Instrument IP Address) in Utility -> I/O -> Network -> Ethernet.
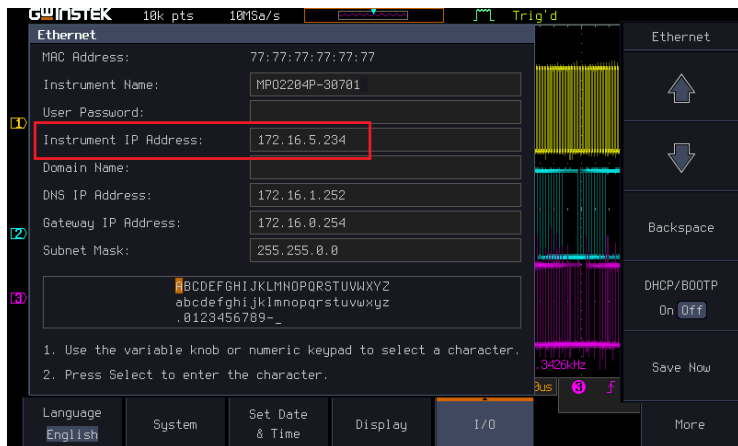
2. **Configure the port**

The URL of the *WebIDE* is composed of the machine's IP and port number, with the port setting stored in the *web_ide* app's configuration file, defaulting to 10180. To modify the port value, you can connect a USB keyboard to the machine and directly edit the configuration file using the built-in editor. Please follow these steps: Click the machine's Python APP menu ("*μPy/Exit*" purple button), turn the knob to select *web_ide*, click *Edit Python APP Config* in the bottom row menu, then click *Edit* on the right-hand menu to access and edit the configuration file (range: 1025 ~ 31025). After editing, remember to save your changes. Please note that this port setting here should be different from the port used by the Socket Server to avoid any errors.
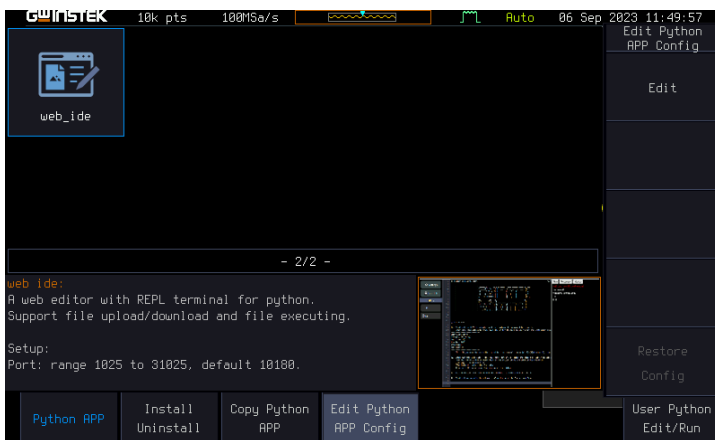
3. **Run the APP and access the WebIDE page**

   Enter the Python APP menu on the machine ("*μPy/Exit*" purple button), turn the knob and select *web_ide* APP. Upon successful execution, the menu will close, and a *μPy* icon will appear in the upper left corner of the screen, indicating that a Python script is running. A prompt for the IP address will also appear (while *WebIDE* is connecting, all buttons on the machine except "*μPy/Exit*" will be non-operational). Afterward, open a web browser and enter the machine's IP address and port in the address bar, such as '172.16.5.234:10180', to access the *WebIDE* page.

Instructions for WebIDE Operations
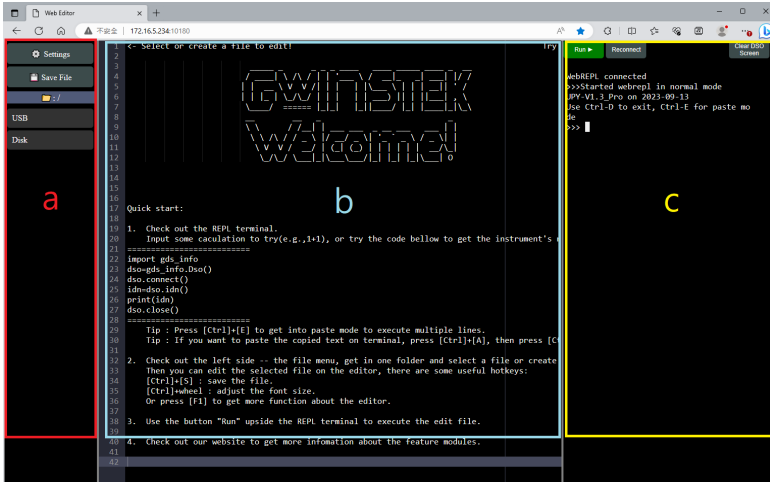
The *WebIDE* page is mainly composed of three sections: file management menu, text editor, and REPL terminal.

a. File Management Menu:

This area displays the machine-side folder paths and their internal files available for user operation. Users can manage the storage of files in each folder. Currently, the locations available for users to use are Disk (internal storage of the instrument) and external USB drive on MPO-2000.

The File Management Menu, as shown in the figure, can be divided into three categories: Title Components, Functional Components, and Object Lists, explained as follows:

1. **Title Components**

   Represented by small icons, these titles indicate whether they represent folders or file documents. The title will display the path location of the current menu list or the name of the file being edited.

2. **Functional Components**

   Includes various management actions for files and paths, such as create, delete, upload, download, and save for the currently edited file. It also includes an interface adjustment menu for opening the *WebIDE*.

3. **Object List**

   Displays the folders and files currently present in the path. Clicking on a folder object will enter that folder and update the display of titles and object lists. Clicking on a file object will display the file's content in the text editor and update the title display. Users can subsequently edit, save, and perform other operations on the file in the editor.

⚠ Note	WebIDE currently only supports text-type files, such as .py, .txt, .csv, .json, etc.

For example, in the left image, the path title indicates the current location is *Disk/* the file title indicates that the currently edited file is *idn.py* and within the current location *Disk/* there is a folder named *proj_1* and two files, *idn.py* and *send_line.py*.

We also provide basic file management functions, as explained below:

- **New folder**
  Create a new folder. Clicking this function will bring up a dialog box where you can enter a name and create it after confirming.

- **Remove folder**
  Delete the current folder (based on the path title). Clicking this function will bring up a dialog box, and after confirming, it will delete the folder along with its subfolders and files.

- **New file**
  Create a new file. Clicking this function will bring up a dialog box where you can enter a name and create it after confirming. (If no file extension is included in the name, a *.py* file will be created by default.)

- **Remove file**
  Delete the currently selected file (based on the file title). Clicking this function will bring up a dialog box, and after confirming, it will delete the file.

- **Upload**
  Upload a file. Clicking this function will open a device file selection dialog, allowing you to choose a file from your device and upload it to the current folder on the machine.

- **Download**

  Download a file. Clicking this function will initiate the download of the currently selected file. After a successful download, a device path dialog will appear to choose the storage location on your PC.

- **Save File**

  Save the currently selected file.

- **Settings**

  Open the settings dialog, where you can adjust font size for the editor and REPL terminal. You can also choose the layout to be displayed in the *WebIDE*.

b. Text Editor:

WebIDE uses the embedded Ace (Ajax.org Cloud9 Editor) as a text editor, supporting practical features such as keyword search, copy-paste, automatic suggestions for defined terms, color-coding based on script language attributes, and automatic highlighting of identical terms.

Common shortcuts:

- **[Ctrl]+[S]**

  Save the currently selected file.
- **[Ctrl]+wheel**

  Adjust the font size of the editor.
- **[Ctrl]+[F]**

  Search for a word.
- **[Ctrl]+[A]**

  Select all content within the editor.
- **[Ctrl]+[C]**

  Copy the selected content.
- **[Ctrl]+[V]**

  Paste the copied content.
- **[F1]**

  View the editor shortcut command list.

c. REPL Terminal:

The REPL terminal allows direct interaction with the built-in Python interpreter on the machine. Users input script fragments here and receive immediate responses from the interpreter. Any errors are also displayed instantly, making it convenient for manual testing of Python scripts. When the 'Run' button is used to execute the file being edited, the execution results and error information will also be displayed here.

REPL terminal shortcut:

- **[Ctrl]+wheel**
  Adjust the font size within the REPL terminal.
- **[Ctrl]+[D]**
  Terminate the REPL Terminal.
- **[Ctrl]+[E]**
  Enter text paste mode, where you can paste multi-line scripts and execute them all at once. After entering this mode, paste the script, press [Ctrl]+[D] to execute, or press [Ctrl]+[C] to cancel and exit paste mode.
- **[Ctrl]+[A]**
  Used to unlock the [Ctrl]+[V] paste function in the REPL terminal. To paste text that has been copied into the REPL terminal, you must first press [Ctrl]+[A], and then press [Ctrl]+[V] to paste the script.

Differences between paste mode and normal mode:

As an example illustrated in the figure below, in normal mode, when you copy the script from the left side and paste it into the right-side terminal, the terminal will execute the line of command as soon as it encounters a newline character.

When entering paste mode, you can paste multiple lines of commands at once and then execute them with [Ctrl]+[D]. The execution information will be consolidated after the script. This is more suitable for pasting and testing a complete set of functions.

REPL function keys:

- **Run**

  Executes the current selected Python script file in the REPL
  terminal. Clicking this button opens a dialog, and upon
  confirmation, the machine executes the script. Messages printed
  using *print()* during execution will be displayed in the REPL
  terminal. While a file is being executed, this button will appear as
  Stop and clicking it will interrupt the current script execution in
  the REPL environment.

- **Reconnect**

  Resets the REPL terminal. Clicking this button opens a dialog, and
  upon confirmation, it updates the REPL terminal environment
  and reconnects. If the REPL terminal appears to be disconnected,
  this function can be used to reestablish the connection. It can also
  be used to initialize the REPL environment.

- **Clear DSO Screen**

  In the process of debugging Python scripts that invoke the LVGL
  graphics library, you may often encounter situations where you need to
  interrupt and exit an ongoing execution. In such cases, you can use this
  function key to clear the graphics drawn by LVGL on the MPO-2000
  screen. Clicking it will bring up a dialog box, and upon confirmation, the
  screen will be cleared.
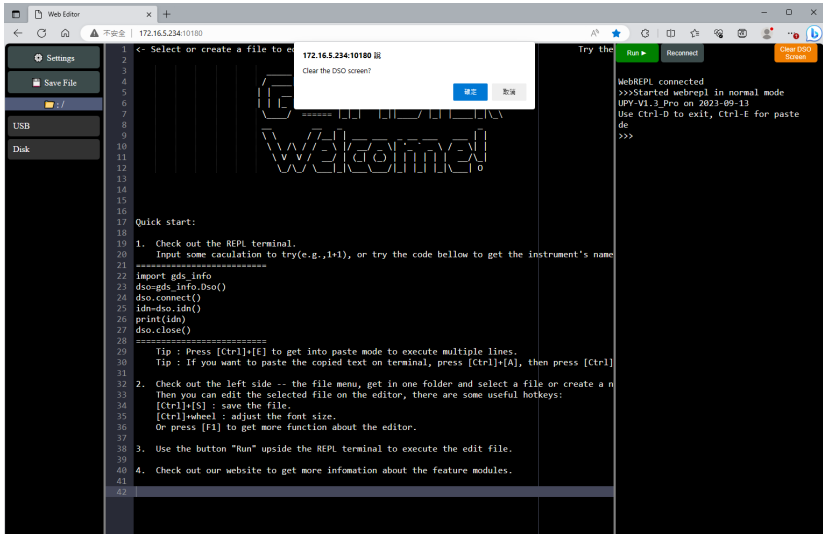
Example of Operating Procedures

Example 1

Creating a Python file, running it to print "Hello World", and machine information such as model and serial number.

Step 1

Navigate to the disk path and click on *New file* to create a new file named *hello_world.py*.
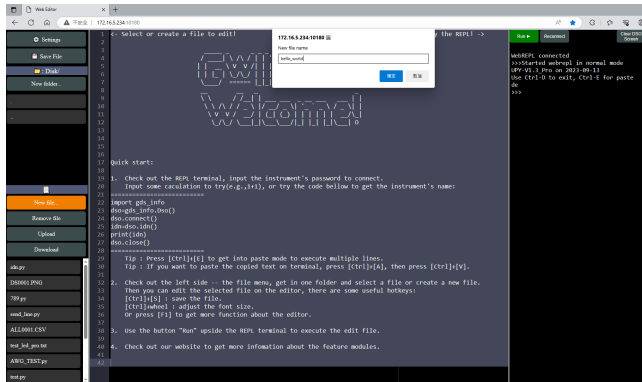
Step 2

Enter the Python script as shown in the figure below into *hello_world.py*. After saving, click *Run* and the REPL terminal will execute the script and display the results.
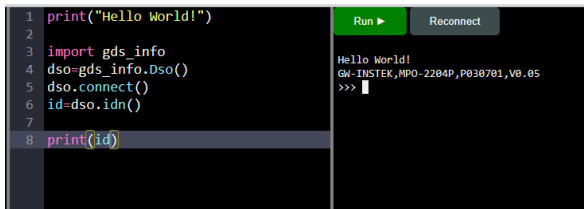
Example 2

Creating a Folder, creating a main script file and module file inside the folder, running the main script file to call a function from the module file and print the result. Here, we demonstrate that files within the same folder can reference each other, allowing users to design their own libraries for use.

Step 1

Navigate to the **Disk** path and click **New folder** to create a new folder named **proj_1**.
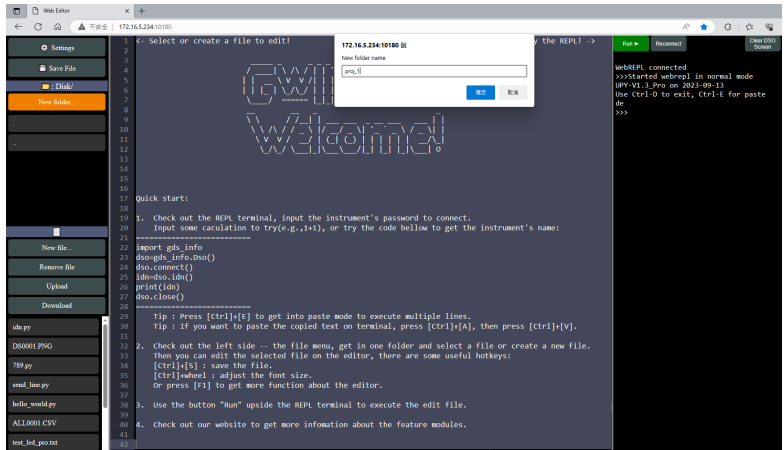
Step 2

Enter the ***proj_1*** path, click on **New file** to create two new files, ***main.py*** and ***func.py***.
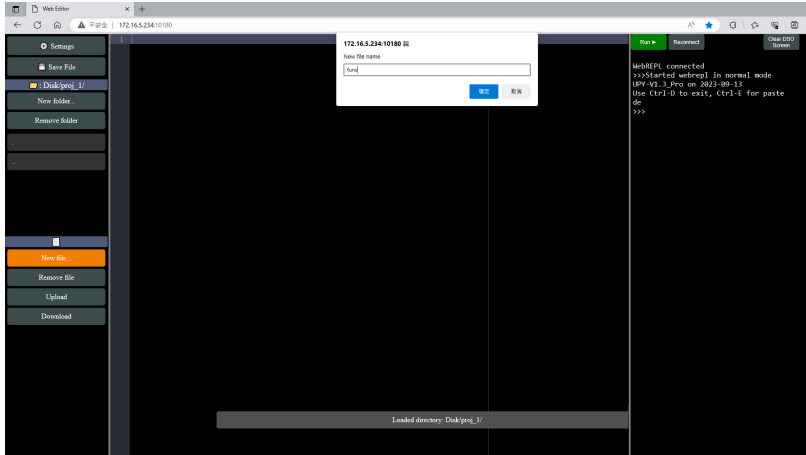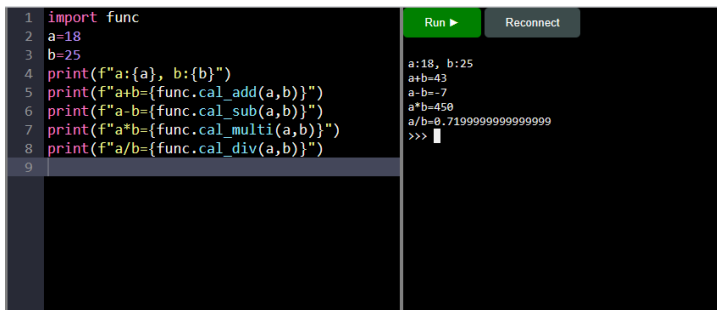
Step 3

Edit *func.py* and input mathematical operation functions. You can also paste the functions into the REPL terminal for testing.
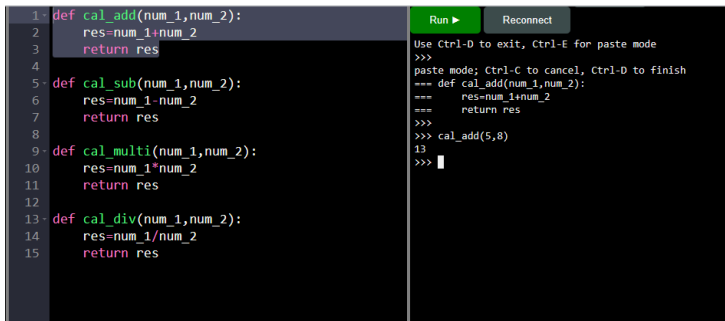
Step 4

Edit *main.py* and reference *func.py* within the script to use the functions defined in *func.py*. After inputting the script and saving it, click *Run* to view the execution results.

Editing Using the Simple Editor on the Machine

We can connect a USB keyboard to the machine and directly edit *.py* and *.txt* files stored inside the machine.

Operation Procedure

Please prepare a USB keyboard that complies with the USB HID protocol and follow the steps below.

Step 1

Insert the USB keyboard into the USB host port on the MPO-2000 panel, and access the Python APP menu. Click on the *User Python Edit/Run* button in the bottom right corner to enter the function menu, then select the *Edit* function.

Step 2

Click on the *Edit* function in the right-side menu and select the file you want to edit from the file list (a *.py* or *.txt* file).
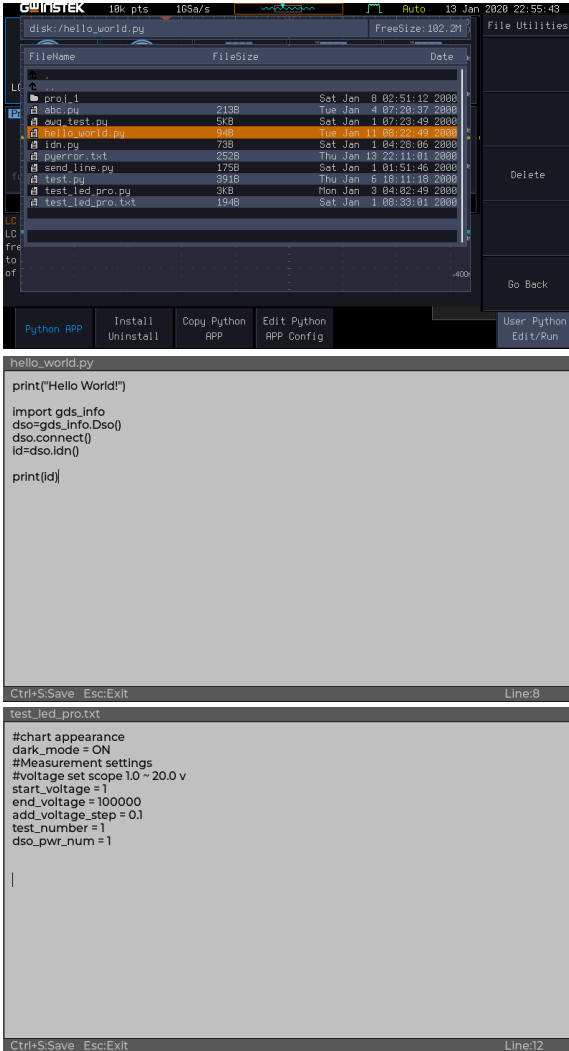
Operating Instructions

The operation of this editor is the same as commonly used text editors. In addition to inputting English words and symbols, it also supports functions such as [Backspace] for deleting text, [Capslock] for toggling between uppercase and lowercase, and [Shift]+letter for quickly toggling input case. You can easily modify the contents of a text file and save it. In addition, there are the following commonly used keyboard function keys:

- **[Tab]**
  Input 4 spaces.
- **[Ctrl]+[S]**
  Save the text file and exit the editor.
- **[Esc]**
  Exit the editor.

Built-in Python APP and its Measurement Applications Guide

In this section, we will explain the operational principles of each built-in Python APP, along with the circuits, measurement methods, and script design considerations required to support their respective measurements. On the menu page, individual Python scripts can be copied to a USB flash drive for users' convenience in viewing or modifying them on a PC. For some apps, we provide parameter configuration files that are loaded during script execution. The purpose is to extend the applicability of the app by modifying parameters without altering the Python script. For example, different BJT components require different Ib and Vcc settings during measurement, and the configuration of horizontal and vertical ranges may also vary. By simply modifying the parameters, the app can be adapted to test a wider range of components.

If you have manually installed third-party Python application scripts (.xpy files) in the Python APP, due to our intellectual property protection mechanisms designed for third-party apps, the Python scripts cannot be copied out.

The built-in Python APP application scripts in MPO-2000 include the following categories:

- BJT Output Characteristics Curve
- LC Oscillator Circuit Frequency vs. Temperature Characteristics Curve
- Fuse Endurance Test
- LED Forward Bias Voltage Characteristics Curve
- Barcode Scanner Measurement Applications

BJT Output Characteristics Curve

BJT (Bipolar Junction Transistor) is a common electronic component, and characteristic curve testing is one of the methods for evaluating its operating range and performance. By setting the BJT at different bias states and measuring the relationship between the output current waveform and the output voltage waveform, BJT's I-V characteristic curve can be plotted. Through characteristic curve measurements, the BJT's operating region can be determined, thus establishing the optimal operating point and evaluating the BJT's performance, parameters, and stability.

We provide two methods for plotting I-V characteristic curves in BJT characteristic curve measurements: using the XY mode of an oscilloscope and using a Python graphic library. The Python script essentially controls Ib, Vcc, and the sampling in the same way.

bjt char curve APP:

In this script, we utilize the XY mode of the oscilloscope to display the I-V characteristic curve. After setting the DC voltage of Vbb each time, the waveform is triggered as V1 rises from 0V to the set voltage. The superimposed waveforms of multiple triggers on the screen can represent the I-V characteristic curves under various Ib conditions.

bjt char curve pro APP:

In this script, we use a GUI library to plot two sets of sampled waveform data (voltage and current in pairs) on a coordinate plane, with horizontal and vertical axis scales. By overlaying waveforms triggered multiple times, you can obtain the I-V characteristic curves under various Ib conditions on the screen. Before actually executing this script, please ensure the horizontal range is appropriate in the **bjt char curve** APP, and that both CH1 and CH2 vertical ranges are suitable, and that the waveform upper and lower parts do not exceed the screen boundaries.

Preparations

For current measurement, we use a differential probe to measure the voltage difference across both ends of Rc, and then divide it by the resistance value of Rc to obtain it. In the measurement process of this characteristic curve, the first half requires manual adjustment of the resistance value of Rb. After confirming Rb, manually adjust the DC output V2 and record the voltage values required for different Ib, which will be sequentially output by the Python script.

Please connect each test lead and probe as listed below:

1. Connect the positive terminal of the DC power supply output1 on MPO-2000 to the V1 (Vcc) terminal of the circuit as shown in the figure, and connect the positive terminal of output2 to the V2 (Vbb) terminal.
2. Connect the probe on CH1 of the oscilloscope to the collector to measure the voltage waveform (Vce).

3. Connect the differential probe on CH2 of the oscilloscope to both ends of Rc (V1 and collector) to measure the current waveform (Ic). Or use a current probe capable of measuring DC current directly from the output1 (relevant parameters in the configuration file need to be modified accordingly).

4. Connect the negative terminal of the DC power supply and the ground terminal of the CH1 probe to the emitter(GND).

In the Python script, we use the voltage waveform as the trigger source, and the trigger level must be properly set to ensure that the waveform data is captured each time. After each change in the voltage at the V2 terminal, the script controls the voltage at the V1 terminal to turn ON and immediately OFF. Capture the sampling waveforms of CH1 and CH2 during this period to draw the I-V characteristic curve.
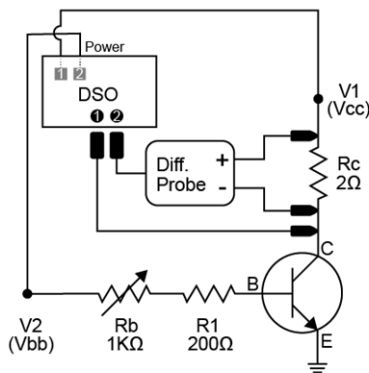
BJT testing circuit and wiring diagram
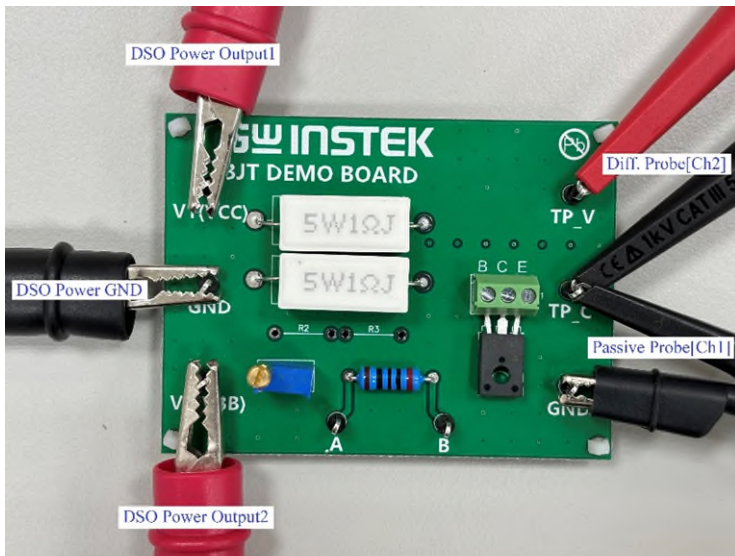
BJT demo board wiring diagram

Precautions

1. Precautions before confirming Vbb voltage and Rb variable resistor value, do not apply voltage on Vcc.
2. Note that the DC Power Supply of MPO-2000 single-channel continuous output has a maximum power of 5W.
3. Rb variable resistor adjustment:

   - Refer to the transistor specification sheet. Assume the maximum output of Vbb voltage to be 10V, and the maximum Ib current to be 10mA. Therefore, choose Rb as 1k ohm. However, since the minimum unit for adjusting the DC voltage output of MPO-2000 is 0.1V, to accurately adjust to the required current, use a 1k ohm variable resistor in this case, and connect a 200-ohm protective resistor (R1) in series.
   - Assuming that when Ib is 1mA and Rb+R1 is 1k ohm, the voltage at both ends is 1V, and since there is a voltage drop of approximately 0.7V across Vbe, set the Vbb voltage to 1.7V initially, and adjust Rb such that the voltage across R1 using a voltmeter is 200mV (1mA × 200 ohms). Measure the resistance value of R1 with a DMM in the power-off state to reduce measurement errors. For example, if the actual measured resistance value of R1 is 201 ohms, the current Ib through R1 will be 1mA when the voltage at both ends of R1 is 201mV.

4. Vbb voltage value adjustment:

   - Adjust the DC voltage output to make the voltage at both ends of R1 become 400mV (2mA × 200 ohms) to obtain the voltage value when Ib is 2mA.

- The remaining Ib current values can all be obtained in the same way to obtain the corresponding Vbb voltage values.

5. Vcc voltage value and Rc resistor value:

- Referring to the characteristic curve of the transistor specification sheet, when Ib is 15mA and Vce voltage is 12V, the Ic current can reach 1.1A. Considering the DC output capability of MPO-2000, Rc is chosen to be a 2-ohm resistor, at this time, the Vcc voltage is approximately 14.2V (1.1A × 2 ohms + 12V).

6. Measure and plot the curve using a differential probe:

- When the voltage at both ends of Rc is too small, there may be problems with unmeasurable or excessive noise.
- As the measurement target is the voltage at both ends of Rc rather than the Ic current, when converting to current, because Rc is 2 ohms, the probe ratio can be adjusted to divide by 2; for example, if the probe ratio is x20, then adjust it to x10, which can correspond to the current scale. The selection of Rc is related to the magnitude of Ic and the attenuation ratio of the probe. If it is not possible to correspond to the correct attenuation ratio, it will not be possible to correspond to the scale correctly in XY mode.

- In the **bjt char curve** APP, we use the XY mode to display the I-V characteristic curve. However, if the selected Rc resistance value is not exactly 2 ohms but 2.1 ohms, because the attenuation ratio can only be specific integers, the characteristic curve presented at this time will have an additional 5% error on the vertical axis.

- In the **bjt char curve pro** APP, the above error of the Rc resistor can be pre-measured using a DMM to obtain its accurate resistance value and then substitute it into the formula I=V/R to calculate the correct current value. Then use the Python graphic library to draw the I-V characteristic curve, thus providing more accurate measurement results.

7. The supply time to Vcc must be kept as short as possible; otherwise, the internal temperature of the components may become significantly higher than the external temperature, resulting in measurement errors. Components like SiC and GaN can endure extremely short conduction times; please refer to the component's specification sheet. Excessive conduction time may lead to component damage, so special attention is required.

File Usage Description

bjt char curve APP:

bjt_char_curve.py: Python script file
bjt_char_curve.txt: parameter configuration file

bjt char curve pro APP:

bjt_char_curve_pro.py: Python script file
bjt_char_curve_pro.txt: parameter configuration file

Execution Steps

1. Modify the parameters in the parameter configuration file according to different requirements, such as the actual measured value of Rc, Vcc voltage, Vbb voltage, and trigger threshold.
2. Execute the Python script.

Python Script Workflow

bjt char curve APP:

1. Load parameter configuration file.
2. Perform initial settings for MPO-2000.
3. Set the two sets of DC voltages for MPO-2000, turn on Output 2, turn on output 1 and immediately turn off output 1 (power-on time is approximately 10ms).
4. Repeat step 3 and 4 according to the number of tests.

bjt char curve APP:

1. Load parameter configuration file.
2. Perform initial settings for MPO-2000.
3. Set the two sets of DC voltages for MPO-2000, turn on Output 2, turn on output 1 and immediately turn it off(power-on time is approximately 10ms).
4. Use Python graphics library to draw a curve on the screen.
5. Repeat step 3, 4, and 5 according to the number of test repetitions.

Characteristics Curve of the Device Under Test

The transistor used in the circuit under test is TTC004B, and its I-V characteristic curve is shown in the figure below:
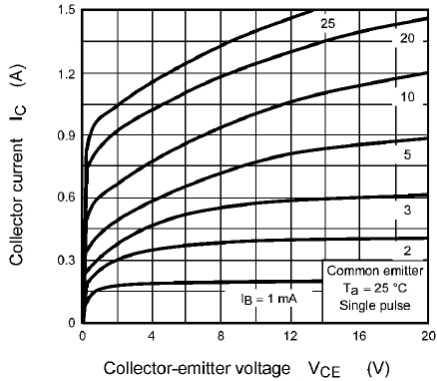
The I-V characteristic curve of TTC004B.

Test Results

The output screen of **bjt char curve** APP
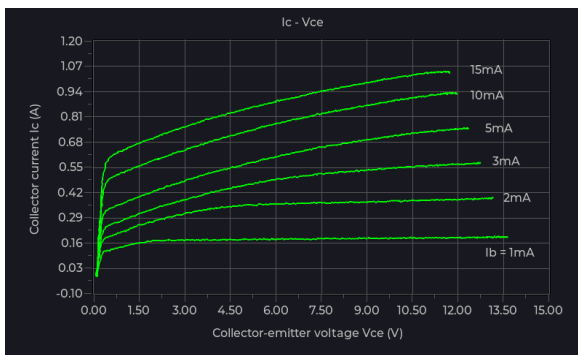
The output screen of **bjt char curve pro** APP

The **bjt char curve** APP and **bjt char curve pro** APP can both run on the Basic and Professional versions of MPO-2000. However, the Python source code modified from the **bjt char curve pro** cannot be executed on the Basic version of MPO-2000, as the Python scripts invoking the graphic library can only run on the Professional version of MPO-2000.

BJT Output Characteristic Curves (Using External DC Power Supply)

In the **bjt char curve pwr pro** APP, we use an external DC power supply (PPX) to power the device under test. The Python script sends remote control commands to the PPX power supply via a USB interface using the USB CDC-ACM protocol, replacing the portion in the **bjt char curve pro** script that uses the built-in DC power supply. Therefore, when performing the same BJT output characteristic curve test, it is possible to test at higher voltages and currents. Except for the initial connection establishment procedure with PPX, the rest of the testing procedure is generally the same as the **bjt char curve pro** APP.

Preparations

Please refer to the operating instructions on page 146 first; the testing principle in this section is the same.

1. Prepare a USB 2.0 USB Hub and connect its device port to the front panel USB slot of MPO-2000.
2. Prepare two PPX DC power suppliers, hereafter referred to as power1 and power2 (the outputs are referred to as output1 and output2, respectively). Connect these two devices to the above-mentioned USB hub using USB cables.
3. Connect the positive terminal output1 to the V1 (Vcc) terminal of the circuit as shown in the figure, and connect the positive terminal of output2 to the V2 (Vbb) terminal.

4. Connect the probe on CH1 of the oscilloscope to the collector to measure the voltage waveform (Vce)
5. Connect the differential probe on CH2 of the oscilloscope to both ends of Rc (V1 and collector) to measure the current waveform (Ic).
6. Connect the negative terminal of the power1 & power2 and the ground terminal of the CH1 probe to the emitter(GND).
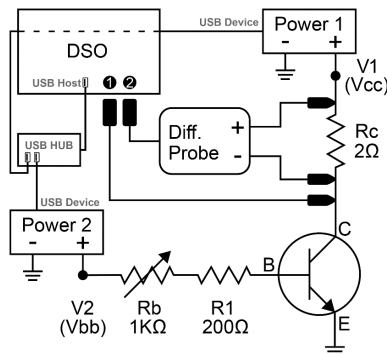
BJT testing circuit and wiring diagram
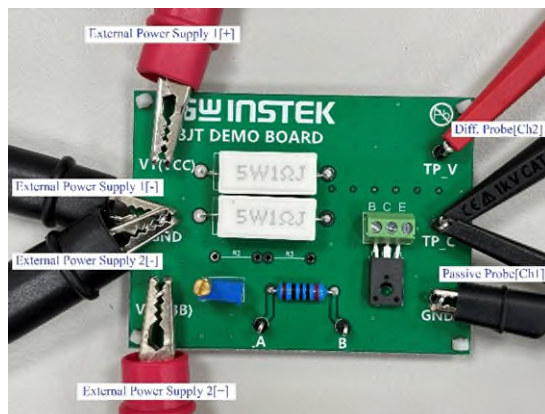
BJT demo board wiring diagram

Precautions

1. Please refer to page 148

2. During the initial setup process, it is necessary to establish a connection with the external power supply through the USB interface, and the product serial number needs to be verified. Please modify the serial numbers on the right side of the equal sign, such as "pwr1ser" and "pwr2ser," in this APP parameter configuration file to match the power supply you are connecting.

3. As the remote control commands to the external device require more time for transmission, special attention should be paid to the possibility that excessive conduction time may cause an increase in the temperature of the test object or even lead to damage.

File Usage Description

bjt_char_curve_pwr_pro.py : Python script file
bjt_char_curve_pwr_pro.txt : parameter configuration file

Execution Steps

1. Modify the parameters in the parameter configuration file according to different requirements, such as the actual measured value of Rc, Vcc voltage, Vbb voltage, and trigger threshold.

2. Execute the Python script.

Python Script Workflow

1. Load parameter configuration file.
2. Perform the initial setup of MPO-2000 and the external power supply unit.
3. Set the power1/2 voltage, turn on output2, turn on output1, and then turn it off after a short delay. Note that the supply time should be as short as possible; otherwise, the internal temperature of the components may be much higher than the external temperature, leading to measurement errors. Also, the conduction time of components such as SiC and GaN should be as short as possible, as the device under test may overheat or even be destroyed.
4. Use Python graphic library to plot the curve on the screen.
5. Repeat step 3, 4, and 5 according to the number of tests.

Characteristics Curve of the Device Under Test

The transistor used in the circuit under test is TTC004B, and its I-V characteristic curve is shown in the figure below:
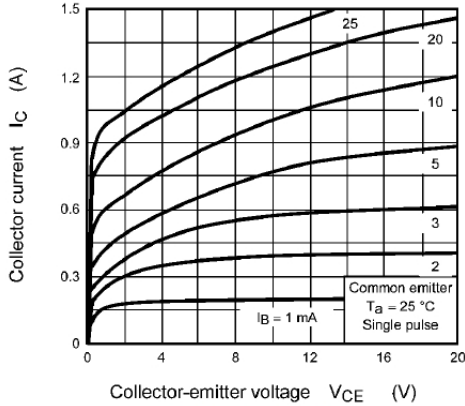
The I-V characteristic curve of TTC004B.

Test Results
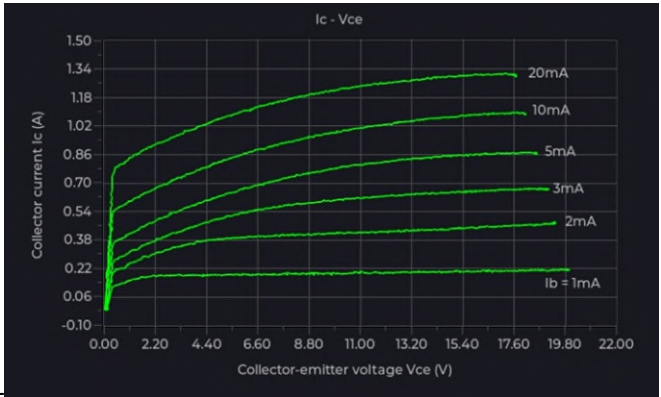
The output screen of **bjt char curve pwr pro** APP

LC Oscillator Circuit Temperature vs.

Frequency Characteristics Curve

This is an application example that utilizes an oscilloscope and a digital multimeter (DMM) in collaboration to measure the frequency vs. temperature characteristics curve of the test object. We use the oscilloscope to measure the frequency of the oscillating circuit, while the DMM, along with a K-type thermocouple, measures the temperature of the test object. Within a certain temperature range, a Python script continuously monitors the temperature readings from the DMM and the frequency measurements from the oscilloscope channel. Here, we take the measurement using an LC resonance circuit as an example, and connect a cement resistor to a DC power source of MPO-2000 as a simple heater. The inductor, capacitor, and K-type thermocouple are all fixed on the cement resistor and coated with thermal conductive adhesive. The DMM measures the temperature, while the oscilloscope probe measures the oscillation frequency, recording the temperature and frequency for every 1-degree Celsius increase. Here, we classify the power source of the heater into the following two types of applications.

LC oscillating APP:

In this script, we utilize the built-in DC power supply of MPO-2000 to output to the cement resistor as a heater, controlling the output voltage to gradually raise the temperature of the cement resistor from room temperature to around 50 degrees Celsius. Throughout the process, the script continuously stores temperature and frequency measurement values in a .csv file

LC oscillating pro APP:

In this script, we use a cement resistor as a heater with the CC (constant current) mode output of the PFR power supply, controlling the output current to gradually raise the temperature of the cement resistor from room temperature to around 50 degrees Celsius. Throughout the process, the script continuously stores temperature and frequency measurement values in a .csv file and updates the temperature and frequency curves. This test configuration is suitable for applications that require higher output (heating) power.

**Preparations**

LC oscillating APP:

1. Connect the positive terminal of the power supply 1 of MPO-2000 to the VDD terminal of the circuit board to provide power for the operation of the oscillation circuit.
2. Connect the K-type thermocouple to the DMM of MPO-2000.
3. Connect the positive and negative terminals of power supply 2 of MPO-2000 to both ends of the cement resistor Rc.
4. Connect the oscilloscope probe of CH1 to the TP terminal.
5. Connect the negative terminal of power supply 1 of MPO-2000 and the ground terminal of the CH1 probe to GND.

LC oscillating pro APP:

1. Connect the positive terminal of the power supply 1 of MPO-2000 to the VDD terminal of the circuit board to provide power for the operation of the oscillation circuit.
2. Connect the K-type thermocouple to the DMM of MPO-2000.
3. Connect the output of the external power supply (e.g., PFR-100M) to both terminals of Rc.
4. Connect the oscilloscope probe of CH1 to the TP terminal.
5. Connect the negative terminal of power supply 1 of MPO-2000 and the ground terminal of the CH1 probe to GND.
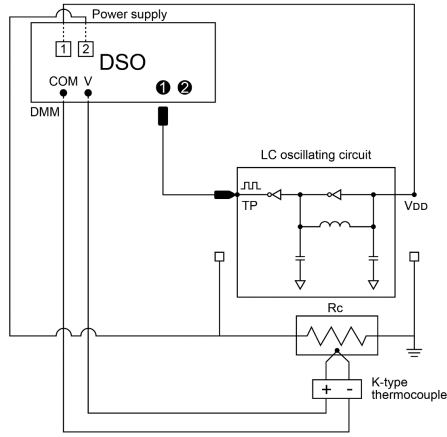
LC oscillating APP testing circuit and wiring diagram
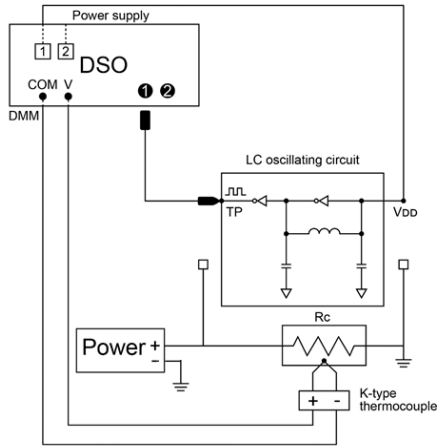
LC oscillating pro APP testing circuit and wiring diagram
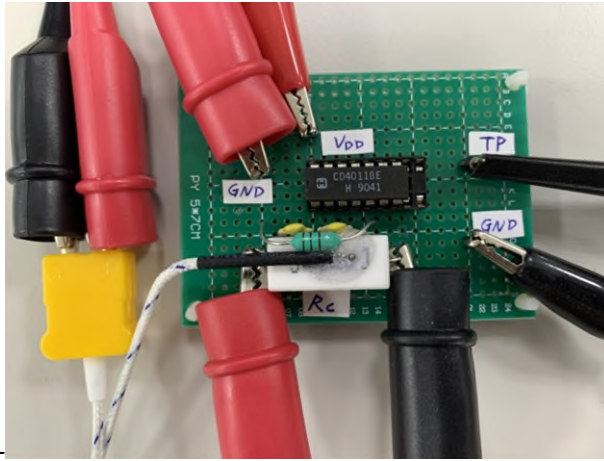
The actual wiring diagram

Precautions

1. The **LC oscillating pro** script needs to establish a connection with an external power supply through the USB interface during the initial setup process, and the product serial number needs to be verified. Please first modify the serial number on the "serial_number" on the right side of this APP parameter configuration file to match the power supply you are connecting to.
2. Pay attention to the maximum power of Rc and the maximum continuous output power of the MPO-2000 series DC power supply single-channel output, which is 5W.
3. Do not let the temperature of the cement resistor rise too quickly, otherwise the cement resistor is prone to cracking, and there will be a larger error in the measured temperature.

File Usage Description

LC oscillating APP:

LC_oscillating.py : Python script file
LC_oscillating.txt : parameter configuration file

LC oscillating pro APP:

LC_oscillating_pro.py : Python script
LC_oscillating_pro.txt : parameter configuration file

Execution Steps

1. Modify the parameters in the parameter configuration file according to different requirements.
2. Execute the Python script.

Python Script Workflow

LC oscillating APP:

1. Load parameter configuration file.
2. Perform initial settings for MPO-2000.
3. Set and activate MPO-2000 DSO power supply 2, start heating Rc.
4. Open the .csv file to prepare for storing temperature and frequency data, then monitor the temperature reading of the DMM. Save the measured temperature and frequency values to the .csv file for every 1°C increase. Note: This temperature-frequency file can be used to draw a curve in Excel.
5. Repeat step 4 and 5 until the temperature reaches 50°C, then stop.

LC oscillating pro APP:

1. Load parameter configuration file.
2. Perform the initial setup for MPO-2000 and external power supply (e.g., PFR-100M).
3. Configure and activate the external power supply, and commence heating Rc.
4. Open the .csv file to prepare for storing temperature and frequency data, then monitor the temperature reading of the DMM. Save the measured temperature and frequency values to the .csv file for every 1°C increase.
5. Use Python graphic library to plot a curve on the screen.
6. Repeat 4, 5, and 6 until the temperature rises to 50 degrees Celsius.
7. Save the screen image as a file.

Test Results

Temperature vs. frequency curve output by **LC oscillating pro** APP

Fuse Endurance Test

fuse endurance pro APP is a Python script testing procedure implemented according to the IEC 60127-1 testing standard. In this testing application, we need to use an external power supply. The specific testing content is as follows:

1. Set the constant current mode using an external DC power supply, allowing the rated current to pass through the test fuse for one hour and then disconnect for 15 minutes, performing 100 cycles continuously. Finally, pass the test fuse with 125% of the rated current for one hour.
2. The difference in resistance values before and after the test should be less than 10%.

Preparations

1. Connect the output terminals of the external power supply (e.g., PFR-100M) to both ends of the fuse.
2. Connect the probe of MPO-2000 (CH1) to both ends of the fuse, ensuring that the grounding end of the probe is connected to the negative terminal of the power supply output.

Wiring diagram of the fuse endurance pro APP

Precautions

1. Please pay attention to the rated current of the fuse, the rated current of the fuse used in this test example is 1A.
2. In this example, we use an oscilloscope channel to measure the voltage difference across the fuse instead of measuring the resistance value. Since the DMM offers better voltage measurement accuracy compared to the oscilloscope, it can be used for applications that require higher precision to measure the voltage difference across the fuse.

File Usage Description

> fuse_endurance_pro.py: Python script file
> fuse_endurance_pro.txt: parameter configuration file

Execution Steps

1. Modify the parameters in the parameter configuration file according to different requirements.
2. Execute the Python script.

Python Script Workflow

1. Load parameter configuration file.
2. Perform the initial setup for MPO-2000 and external power supply (e.g., PFR-100M).
3. Execute the fuse endurance testing script
4. Verify whether the voltage difference before and after the test procedure has changed by less than 10%.

Test Results
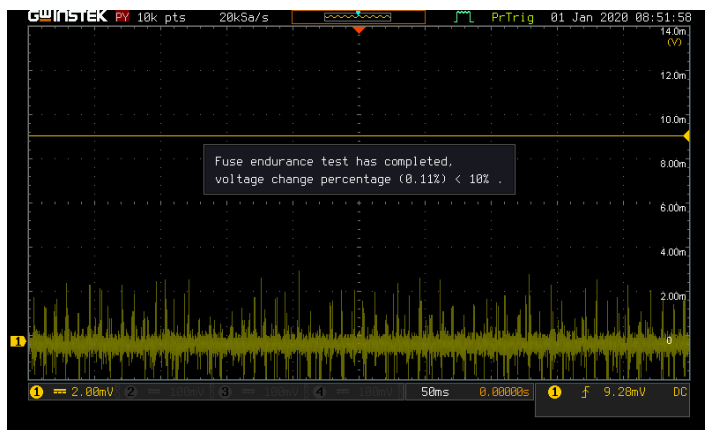
The output screen of **fuse endurance pro** APP

LED Forward Bias Voltage Characteristics Curve

test led pro APP is a script application used to observe the characteristic curve of the relationship between the forward voltage and current of an LED. It can be used to observe the cutoff voltage of the LED, where the LED begins to conduct and emit light when the forward bias exceeds the cutoff voltage. In this test, we used the built-in DC power supply of the MPO-2000 to provide the LED with a forward bias, measured the voltage across the LED using oscilloscope channel 1, and measured the current passing through the LED using the built-in DMM.

Preparations

1. Connect the output of the DC power supply of MPO-2000 (default to channel 1) to the mA probe jack of the DMM for testing.
2. Connect the COM jack of the DMM to the anode of the LED using a test lead.
3. Hook the probe of channel 1 of the oscilloscope to the anode of the LED.
4. Connect the LED cathode to the probe ground and the negative terminal of the DC power supply on the MPO-2000.
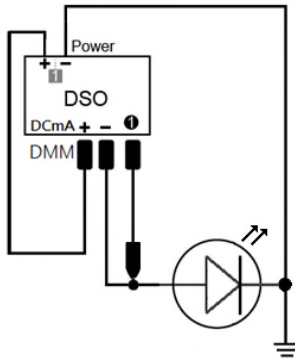
LED testing circuit and wiring diagram

Precautions

1. Before testing, please confirm the positions of the LED anode (positive) and cathode (negative) to avoid connecting them incorrectly and causing damage to the LED.
2. Please pay attention to the applied voltage range in this test to avoid generating excessive current that may cause the internal DMM fuse to blow (maximum acceptable current is 600mA, exceeding 1A will result in a blown fuse).
3. Note that the minimum voltage output of MPO-2000's DC power supply is 1.0V, and it does not provide overcurrent protection with adjustable settings. Therefore, avoid using excessively high test voltages to prevent excessive current that may lead to the burning of the device under test.
4. When testing high-power LEDs, it may be necessary to modify the wiring, by switching the test lead from the 'mA' probe jack of the DMM to the 'A' probe jack, and setting the DMM mode to DCA mode in the script.
5. Pay attention to the specifications of the LED, and do not exceed its maximum voltage tolerance.

File Usage Description

    test_led_pro.py: Python script file
    test_led_pro.txt: parameter configuration file

Execution Steps

1. Modify the parameters in the parameter configuration file according to different requirements.
2. Execute the Python script.

Python Script Workflow

1. Load parameter configuration file.
2. Perform initial setup for MPO-2000 oscilloscope, DMM, and DC power supply.
3. Set the voltage of the DC power supply and enable output.
4. Read the current measurement value from the DMM.
5. Read the voltage measurement value from channel one of the oscilloscope.
6. Use Python graphic library to draw a curve on the screen.
7. Repeat step 3, 4, 5, and 6 until the specified test count limit.

Test Results
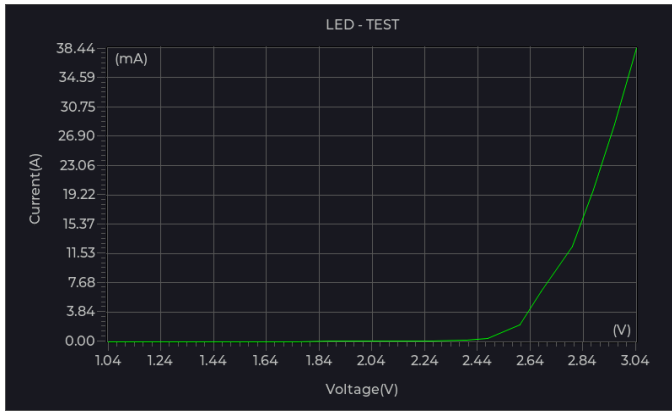
The output screen of **test led pro** APP

LED Forward Bias Voltage Characteristics Curve (Using External Power Supply and Digital Multimeter)

In **test led external device pro** APP, we repeat the test from the previous section and use an external power supply instead (e.g., PFR-100M) to provide a higher power output. We also connect to the RS-232 interface of an external digital meter (such as GDM-8261A) through a USB to RS-232 converter cable. This part is solely to demonstrate the Python script for controlling external RS-232 devices, serving as a reference for users.

Preparations

1. Connect the positive terminal of the PFR power supply to the current jack of the GDM digital multimeter using a test lead.
2. Connect the GDM COM jack to the LED anode (positive) using a test lead.
3. Connect the LED cathode (negative) to the negative terminal of the PFR power supply with a test lead.
4. Connect CH1 of MPO-2000 to the anode (positive) of the LED using a passive probe, and connect the probe ground to the cathode (negative) of the LED to measure the voltage difference across the LED.
5. Connect the USB Host port of MPO-2000 to a USB 2.0 HUB to control the PFR power supply and GDM digital multimeter (requires an RS232 to USB converter cable).
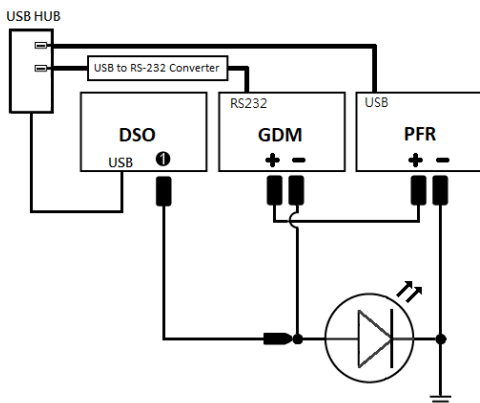
LED testing circuit and wiring diagram

Precautions

1. Pay attention that the voltage setting of the PFR does not exceed the maximum voltage indicated by the LED specifications.
2. Pay attention to the OCP setting of the PFR, ensuring that it does not exceed the maximum current that GDM-8261A can withstand.

File Usage Description

test_led_external_device_pro.py: Python script file
test_led_ external_device_pro.txt: parameter configuration file

Execution Steps

1. Modify the parameters in the parameter configuration file according to different requirements.
2. Execute the Python script.

Python Script Workflow

1. Load parameter configuration file.
2. Perform initial setup for MPO-2000 oscilloscope, DMM, and DC power supply.
3. Set the voltage of the DC power supply and enable output.
4. Read the current measurement value from the DMM.
5. Read the voltage measurement value from channel one of the oscilloscope.
6. Use Python graphic library to draw a curve on the screen.
7. Repeat step 3, 4, 5, 6, and 7 until the specified test count limit.

Test Results
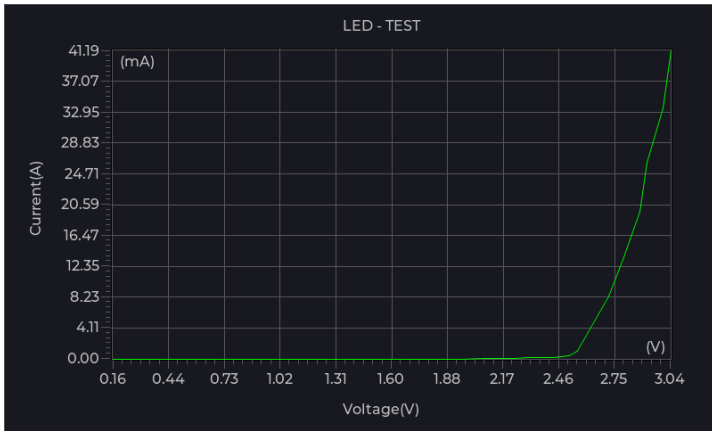
The output screen of **test led external device pro** APP

Barcode Scanner Measurement Application

In the **barcode scanner** APP, we simulated the process of testing a specific electrical parameter of components in the factory production line. And record the measurement values of the test substance along with the serial numbers on the barcode labels one by one in a file. As shown in the figure below, we first set the DMM to the ohm mode and connect the probes to two test points on the demo board. Then, we use a barcode scanner to scan the serial number sticker on the top right of the demo board and immediately read the resistance value of the DUT by the DMM. After completion, the serial number and measurement result are displayed on the screen, and the same data is stored in a buffer. This process continues to record measurement values until the script determines that the scanned content is "EXIT," at which point the contents of the buffer are all saved to the default file under Disk (csvdata.csv) before exiting the Python program.

This example simply demonstrates the application of the barcode scanner, and users can modify it according to different measurement requirements.

Preparations

1. Connect the barcode scanner to the USB host port of the MPO-2000.
2. Connect the DMM probe of MPO-2000 to test point V2 (VBB) and test point B on the demo board to measure the resistance value at both ends of R1.
3. Prepare a barcode stickers with the content "EXIT".

Barcode scanner measurement application wiring diagram

Precautions

1. In this test case, we generate an EXIT barcode encoded in Code39. During the test, the barcode can be scanned by a barcode scanner to exit the Python program.
2. The lv.tick_inc() at the end of the while loop in the script should not be given an excessively large value, as it may cause some characters to be missed during scanning.

EXIT barcode

File Usage Description

barcode_scanner.py: Python script file

Execution Steps

1. Execute the Python script.

Python Script Workflow

1. Continuously monitor barcode scanner input, and 'KEY_ENTER' indicates the end of the string.
2. Read the input barcode and perform measurements.
3. Display the barcode and measurement results on the screen.
4. Upon detecting the end barcode 'EXIT', save all data to a file and exit the program.
5. Return to 1.

Test Results

The graphic frame can be displayed in the local area of the MPO-2000 screen

Can be saved as a CSV file for easy access on a PC using applications such as Excel.

System Limitations

Here are the limitations in the system caused by resource utilization, security considerations, and the inherent limits of the invoked libraries:

1. If a script executed in the form of a Python APP involves file opening and writing, data cannot be stored in the current path of the script execution. This is because all other files under the script's path will be deleted once the script execution is completed. This limitation does not apply when executing the .py file directly on Disk or USB flash disk.
2. The professional version can handle waveform data up to 100k points. However, due to the limitations of the LVGL library itself, when using the GUI library to draw waveform graphics, the total data points cannot exceed 50k points.
3. The file editor provided on the machine is implemented using the LVGL library. It can edit .py and .txt files, but the content of the files opened is limited to 400 lines. For files exceeding this limit, please perform editing on a PC first.
4. Due to system resource limitations, Python APP and Python script are unable to simultaneously run bus decoding functions for CAN FD, USB 2.0 (full speed), FlexRay, USB PD, and I2S. The bus decoding function will automatically terminate when the Python APP and Python script are running.

5. When using a USB to RS-232 converter, we recommend products that internally use FTDI's FT232RL chip or Prolific's PL2303 chip. These are the converters that our drivers support and have been tested for stable operation. Additionally, products with the CH34x chip internally have been tested and found to be incompatible.

6. When a USB Hub is required, consider the power supply capability of the host. It is recommended not to connect more than 4 USB devices to the USB Hub.

7. When using the Python GUI Library for drawing, if you need to save the screen as an image file, please disable the ink-saving mode of the machine's Hardcopy, otherwise the darker parts of the screen may be automatically changed by the system to a lighter color.

8. Includes pre-installed Python apps, with a maximum of up to 100 apps available in the Python APP selection screen. The storage capacity for apps is limited to 20MB, and once exceeded, new apps cannot be installed.

Appendix

Here are some practical application script cases we have actually run on MPO-2000. Due to the testing process, it may be necessary to install some mobile apps and properly configure them to interact with Python scripts on MPO-2000. These mobile apps may be removed from the market after some time, making it difficult to reproduce the process. Users may need to make partial modifications to these Python scripts to interact with similar functioning mobile apps. Such applications typically require a thorough understanding of the functionality, settings, and data format for uploading data from the mobile app. Some may even require the application for a personal account or API key to interact with social software, necessitating a certain level of expertise for successful execution. Therefore, we have not loaded these application scripts onto MPO-2000, only providing a rough description and source code for reference regarding the related applications.

MQTT Remote Control Example

MQTT is a lightweight communication protocol based on the Publish/Subscribe model. It is built on top of the TCP/IP protocol, thus ensuring excellent cross-platform and interoperability, as well as reliable message exchange. MQTT controls messages with a minimum data size of only 2 bytes and can carry up to 256 Mb of data. It is widely used as a communication protocol between IoT devices.

We provide a Python library for MQTT on MPO-2000, enabling the implementation of MQTT's publishing and subscribing functionalities. The role of the broker requires robust computational power and more resources, typically handled by servers and not within our scope of application.

Principles and Explanations

In the script **mqtt_dso_ctrl.py**, MPO-2000 primarily serves as the subscriber, receiving commands forwarded by the broker from the mobile. It interprets the command content and executes the corresponding actions. Here, we define two functionalities, "Run/Stop" and "Autoset", which correspond to two buttons on the mobile app.

After receiving and executing the subscribed commands, the Python script also acts as a publisher, sending the status of specific buttons to the broker. The mobile app can use this to synchronize the displayed button status with MPO-2000.

Similar applications, such as smart lighting and smart home appliances at home, largely employ this mechanism, allowing us to easily control them remotely via the mobile app.

Mobile App Configuration

Here we take the 'IoT-Manager' App for Android phones as an example. The setup process is as follows:

1. Download the free software, install, and open it.
2. Press the function icon in the upper left corner and select 'MQTT Connections' (as shown below).
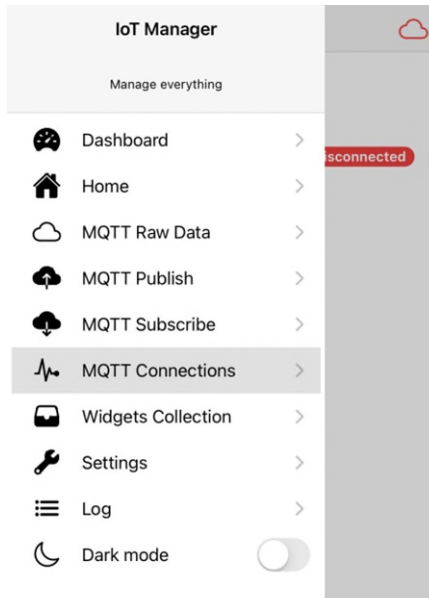
3. Select the second item 'Mosquitto.org' and swipe left to press 'Edit'.
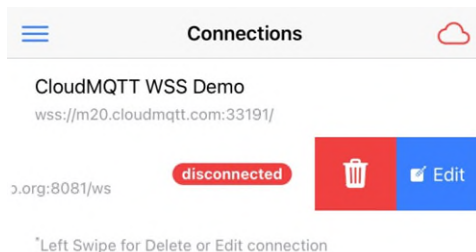
4. The 5th item, 'Prefix," should be changed from "/IoTmanager" to "gw." Then, press the icon at the top right marked "v" to confirm and return.
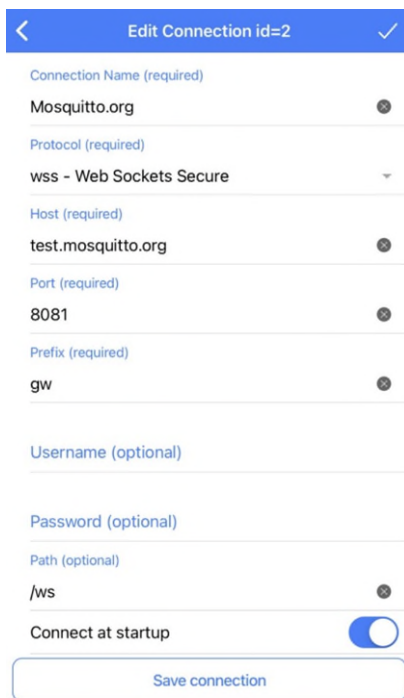
Precautions

1. The network interface of MPO-2000 must be correctly configured first.
2. DNS also needs to be configured, otherwise, it will not be able to connect to the MQTT cloud server (Broker).
3. The data content of MPO-2000's Python script published to the broker needs to reference the definition of the mobile App, with different formats for each provider.

Execution Steps

1. After setting MPO-2000 to its default configuration, appropriately configure and activate AWG channel 1, and connect its output to oscilloscope's CH1.
2. Open the "IoT-Manager" App on your mobile phone.
3. Tap on the top-left corner icon and select "Dashboard" (if the top-right icon is green, it indicates a connection; if red, press it to confirm the connection status).
4. Execute the mqtt_dso_ctrl.py script.
5. The "IoT-Manager" App on your phone will display two buttons, "Run/Stop" and "Autoset". Pressing either of these buttons will prompt MPO-2000 to perform corresponding actions upon receiving the subscribed command.

Python Script Workflow

1. Connect to the broker.
2. Transmit the initial settings to the broker in the role of the publisher. The mobile app, upon activation, will download the settings from the broker in the role of the subscriber and create a button.
3. Check for the receipt of button content subscribed to the broker and instruct MPO-2000 to perform the corresponding action (Run/Stop or Autoset). The mobile app will, upon detecting a button pressed on the screen, transmit the command for the corresponding button to the broker in the role of the publisher.
4. Verify if there has been a change in the Run/Stop status of MPO-2000. If so, transmit the status to the broker in the role of the publisher. The mobile app will check for the receipt of the subscribed button status from the broker and update the appearance of the button accordingly.
5. Repeat step 3 and 4..
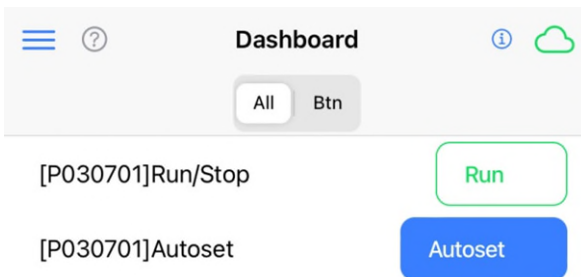
Remote control screen of the mobile app

MQTT Measurement Example

By making slight modifications to the previous application, you can easily switch the roles of the publisher and subscriber. Therefore, MPO-2000 can publish measurement data to a cloud server (Broker) while the mobile app continuously reads the measurement values uploaded by MPO-2000. Before executing this Python script, please refer to the configuration process in the previous section titled 'MQTT Remote Control Example.

Principles and Explanations

In the script **mqtt_dso_meas.py**, MPO-2000 primarily plays the role of the publisher, periodically uploading a certain measurement value to the broker, which then forwards the measurement data to the subscribers. At this point, the mobile App can be configured as a subscriber, with some mobile Apps even providing chart displays, allowing the measured data can be visualized in graphical form.

Python Script Workflow

1. Connect to the broker.
2. Send the initial settings to the broker in the role of the publisher. The mobile app, upon startup, will obtain the settings from the broker in the role of the subscriber and establish the chart.
3. Periodically send MPO-2000 measurement values to the broker in the role of the publisher. The mobile app will continuously check for received measurement data subscribed to the broker and update the chart accordingly.
4. Repeat step 3.
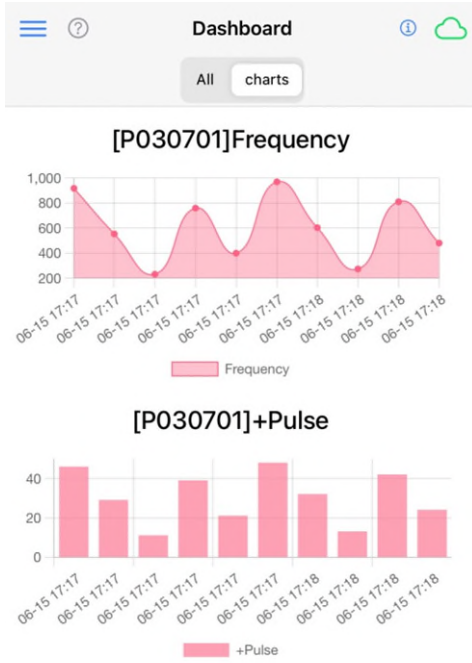
The chart created by the mobile app using the measurement data subscribed to the broker.

Reference Materials

The built-in Python APP source code, oscilloscope library source code, and documentation for MPO-2000:
https://github.com/OpenWave-GW/Python_APP/

MicroPython libraries official website documentation:
https://docs.micropython.org/en/v1.19.1/library/index.html

The LVGL documentation and examples from LVGL Kft. company:
https://sim.lvgl.io/v8.3/micropython/ports/javascript/index.html

The following is Good WillL Instrument's copyright statement for the provided Python APP source code and the Python modules and libraries:

GW Python APP License

Copyright (c) 2023 GOOD WILL INSTRUMENT CO., LTD.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to use the Software for the purposes of development, modification, testing, and distribution of code specifically designed to operate with GOOD WILL INSTRUMENT's programmable instruments, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The following is the copyright statement provided by vsolina:

MIT License

Copyright (c) 2022 vsolina

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The following is the copyright statement provided by LVGL Kft.

The following is the copyright statement provided by Damien P. George.

The MIT License (MIT)

Unless specified otherwise (see below), the above license and copyright applies to all files in this repository.

Individual files may include additional copyright holders.

The various ports of MicroPython may include third-party software that is licensed under different terms. These licenses are summarised in the tree below, please refer to these files and directories for further license and copyright information. Note that (L)GPL-licensed code listed below is only used during the build process and is not part of the compiled source code.

```
/ (MIT)
  /drivers
    /cc3100 (BSD-3-clause)
  /lib
    /asf4 (Apache-2.0)
    /axtls (BSD-3-clause)
      /config
        /scripts
          /config (GPL-2.0-or-later)
        /Rules.mak (GPL-2.0)
    /berkeley-db-1xx (BSD-4-clause)
    /btstack (See btstack/LICENSE)
    /cmsis (BSD-3-clause)
    /crypto-algorithms (NONE)
    /libhydrogen (ISC)
    /littlefs (BSD-3-clause)
    /lwip (BSD-3-clause)
    /mynewt-nimble (Apache-2.0)
    /nrfx (BSD-3-clause)
    /nxp_driver (BSD-3-Clause)
    /oofatfs (BSD-1-clause)
    /pico-sdk (BSD-3-clause)
    /re15 (BSD-3-clause)
    /stm32lib (BSD-3-clause)
```

/tinytest (BSD-3-clause)
/tinyusb (MIT)
/uzlib (Zlib)
/wiznet5k (MIT)
/logo (uses OFL-1.1)
/ports
  /cc3200
    /hal (BSD-3-clause)
    /simplelink (BSD-3-clause)
    /FreeRTOS (GPL-2.0 with FreeRTOS exception)
  /esp32
    /ppp_set_auth.* (Apache-2.0)
  /stm32
    /usbd*.c (MCD-ST Liberty SW License Agreement V2)
    /stm32_it.* (MIT + BSD-3-clause)
    /system_stm32*.c (MIT + BSD-3-clause)
    /boards
      /startup_stm32*.s (BSD-3-clause)
      /*/stm32*.h (BSD-3-clause)
    /usbdev (MCD-ST Liberty SW License Agreement V2)
    /usbhost (MCD-ST Liberty SW License Agreement V2)
  /teensy
    /core (PJRC.COM)
  /zephyr
    /src (Apache-2.0)
/tools
  /dfu.py (LGPL-3.0-only)