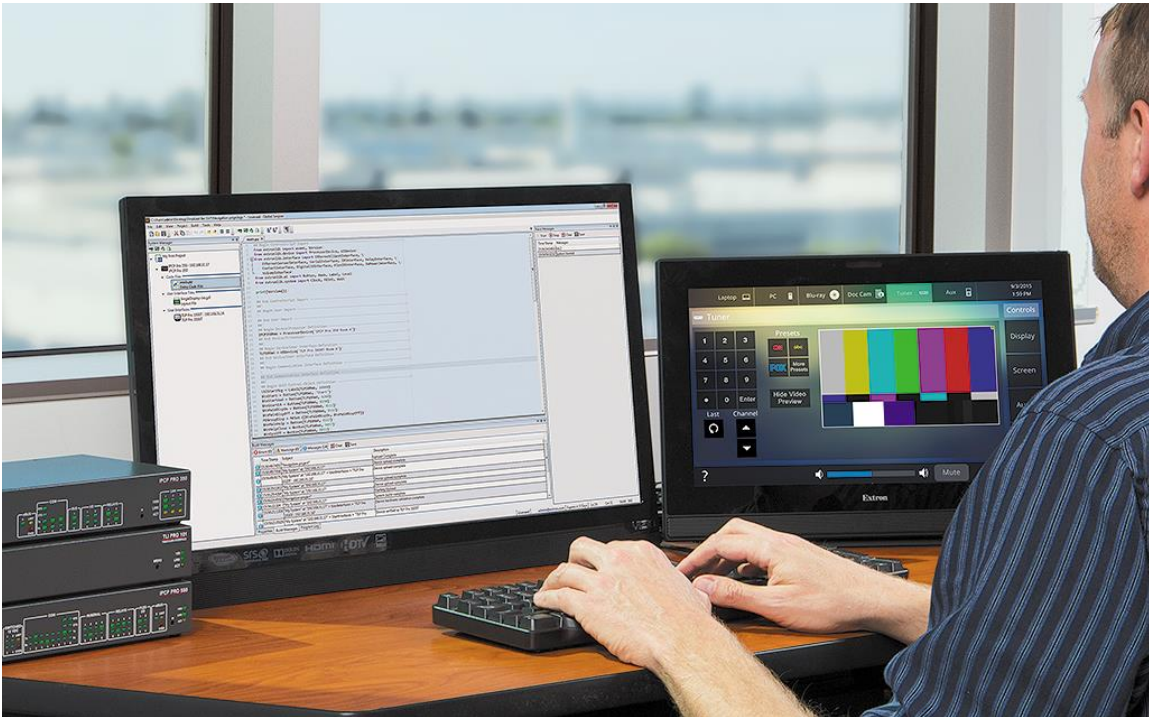


Reference Guide

Extron 控制系统

Python 编程使用指南



Extron Electronics
INTERFACING, SWITCHING AND CONTROL

目录

1 概述	1
1.1 目的	1
1.2 需知	1
2 实施	2
2.1 Python 3.3 解析器	2
2.2 开发环境	2
2.3 Extron 对象库及保留的关键词	2
3 Global Scriptor 介绍	3
3.1 编程界面	3
3.1.1 系统管理 (System Manager)	3
3.1.2 代码编辑器 (Code Editor)	4
3.1.3 诊断面板 (Diagnostics Panels)	5
4 Python 的基础	7
4.1 语法规则	7
4.2 Extron 编程的标准规范	7
4.2.1 通用建议	7
4.2.2 标识符 (Identifiers)	8
4.2.3 行长度和续行	8
4.2.4 字符串	9
4.2.5 ID 范围	9
4.3 解析器和库	9
4.4 面向对象的编程	9
4.5 列表和字典	10
4.6 循环和迭代	10

5 从零开始学编程	12
5.1 在 Global Scripter 中新建项目.....	12
5.2 入口程序代码 main.py	14
导入库文件	14
5.3 视音频系统中的 “Hello World!”	14
Event decorator 事件修饰器.....	16
6 编写用户界面导航程序.....	18
7 编写控制周边设备的程序.....	24
8 编写与受控设备进行数据通讯的程序.....	25
8.1 通过红外端口发送数据	25
8.2 单向串口发送数据.....	28
8.3 双向串口通讯.....	30
8.3.1 异步通讯.....	30
8.3.2 解析从设备回传的数据	30
8.3.3 同步通讯.....	36
8.4 以太网端口的通讯.....	38
8.4.1 建立以太网连接.....	38
8.4.2 客户端通讯 – TCP	39
8.4.3 客户端通讯 – UDP	41
9 音量控制接口的程序.....	42
10 输入和输出接口的程序	44
10.1 继电器接口	44
10.2 开关电源接口.....	44
10.3 触点闭合接口.....	44
10.4 数字输入输出接口	45

10.5 多功能输入/输出接口	46
11 延时和计划任务的程序	47
11.1 使用 Wait 延时执行函数	47
11.1.1 使用事件修饰函数定义一次性 Wait.....	47
11.1.2 命名 Wait.....	47
11.2 基于时钟和日历时间的计划任务事件	49
附录 A-Python 学习资源.....	51
纸质和电子参考书	51
网站和教程	51
附录 B-禁用的 Python 内置函数和模块.....	52
禁用的内置函数	52
禁用的内部模块	52
禁用的外部模块	53
词汇表.....	55

1 概述

Global Scripter 是一款功能强大和灵活的控制系统编程软件。系统使用了简单易懂的 Python 编程语言，提供了全面和功能丰富的开发环境，适用于 Extron 的 Pro 系列控制系统。

Extron 设计的 Global Scripter，可以在各方面提高视音频程序员的编程效率。其独有的 Python 库、实用的贴士、示范代码语句等人性化设计，使得整合视音频控制的系统项目变得更加便捷。Global Scripter 完全是站在程序员的角度设计的。

1.1 目的

该文档主要讲解了如何在 Extron Pro 控制系统中使用 Python 编程语言的方法，它以分章节的形式指导程序员如何有条理地使用 Extron 提供的库进行编程。

该文档虽然不是 Python 编程教程，但其中的确包含很多 Python 的元素，并解释了将 Python 引入 Extron Pro 控制系统的方法。更多关于 Python 的资源，请参考附录 A。

1.2 需知

在使用该手册前，请注意以下事项：

- 程序员需要对 Python 有一定的了解。这会加快学习 Extron Pro 控制系统编程的速度。这份手册仅提供使用 Python 编程的基础内容。
- Extron 所用的 Python 语言的版本是基于 3.3 的版本。市面上有各种不同的 Python 版本，一些早期的版本已经不建议采用，如果你正在浏览指导书或者文档来提高对 Python 了解，请确保这些指导书或文档是基于 Python 3.3 或者更高的版本。
- 编程人员必须熟悉 Extron Pro 系列控制系统硬件，包括 TouchLink Pro 触摸屏。
- 编程人员的电脑需要具有管理员权限，以运行 Extron Global Scripter 软件。
- 在学习过程中，编程人员需要 Extron Pro 系列控制处理器和 Extron Pro 触摸屏设备硬件。通过上传程序到系统，可以观察设备间的交互，会让你受益匪浅。
- 编程人员需要具有设计视音频系统的经验，了解视音频系统典型组件。

2 实施

2.1 Python 3.3 解析器

Extron 采用 Python 3.3 版本的解析器，使用 Extron Global Scripter 软件可直接对 Extron Pro 系列控制系统进行编程，选择 Python 有很多原因：

- 文档齐全并具有大量的资源来帮助编程者快速上手
- 语言简练、不晦涩，可读性比其他语言更强
- Python 是面向对象和可扩展的语言，Extron 将其定制成为适用控制系统的平台

2.2 开发环境

Extron GUI Designer 是创建 Touchlink Pro 触摸屏用户界面的软件。Extron Pro 控制系统编程需要的控制对象（按键）和页面的 ID 号也是由它分配的。

Extron Global Scripter 可以应对高级控制系统编程的需求。它提供了整个控制系统程序的框架，用户可以将控制系统需要的相关文件集合在一起，包括用户界面文件、红外文件、音频文件、数据文件以及程序代码。此外，它还具备上传、测试、调试控制系统的功能。

2.3 Extron 对象库及保留的关键词

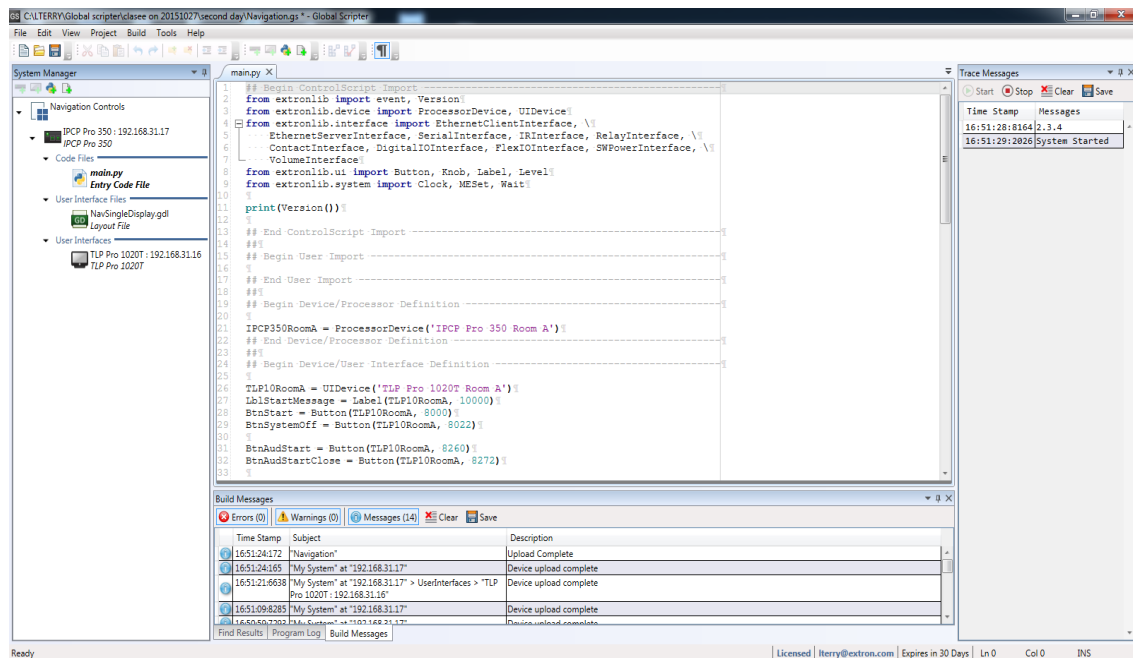
和所有的高级编程语言一样，在编程时会保留一定的关键词为系统使用，这些关键词不能作为您编程的标识符使用。Extron 添加了 Extron 对象库（Extron Object Library）和 Extron Pro 系列控制系统关联，它包含了控制脚本和内置函数。这样，Python 解析器就可以和中控主机及触摸屏固件中定义的控制系统对象进行交互了。在 Global Scripter 软件的帮助文档中，库中每个元素都有详细的解释和使用范例。

3 Global Scriptor 介绍

Global Scriptor 是一个完整的设计平台，专用于 Extron Pro 控制系统的编程。程序员或用户需要拥有 Extron 的 Insider 英赛网帐号并获得授权才能使用。授权的认证方式与 Global Configurator Plus 和 Global Configurator Professional 完全一样，有在线和离线认证两种。关于授权认证如有进一步疑问，请联系 Extron 的销售代表寻求帮助。

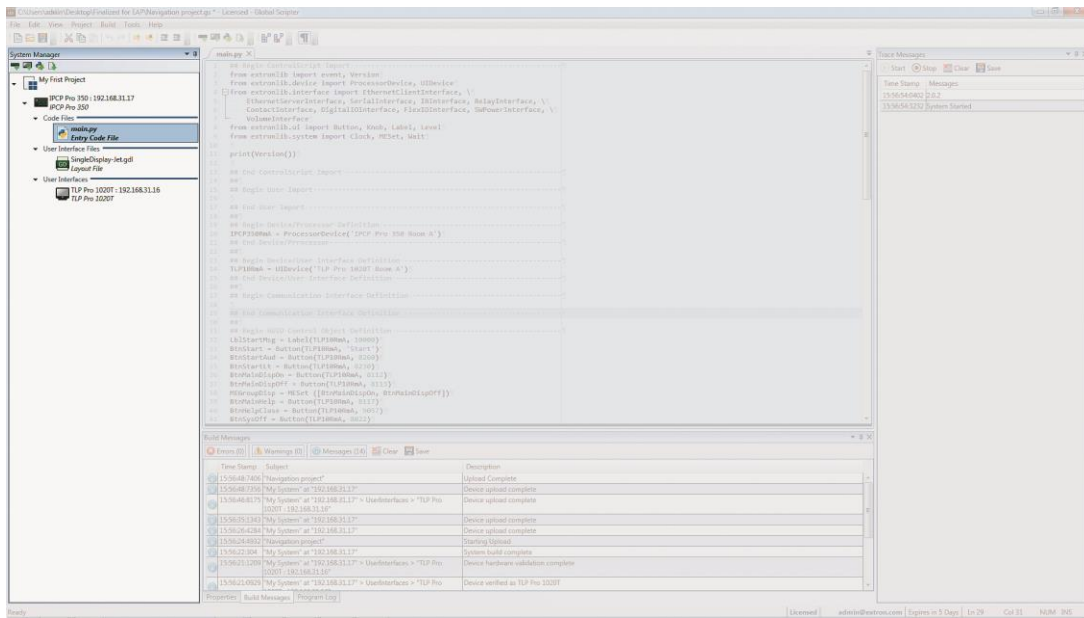
3.1 编程界面

Global Scriptor 包含菜单（menus）、工具栏（toolbars）等独立的界面，提供控制系统相关的各种信息窗口。默认状态下，会显示 4 个部分：系统管理（System Manager）、代码编辑器（Code Editor）、诊断面板（Diagnostics panels）、和状态栏（Status bar）。



3.1.1 系统管理 (System Manager)

系统管理窗口主要用于添加控制处理器、用户界面产品、图形化用户界面，代码文件、和红外文件，当添加完成后，它们将以树状形式显示。

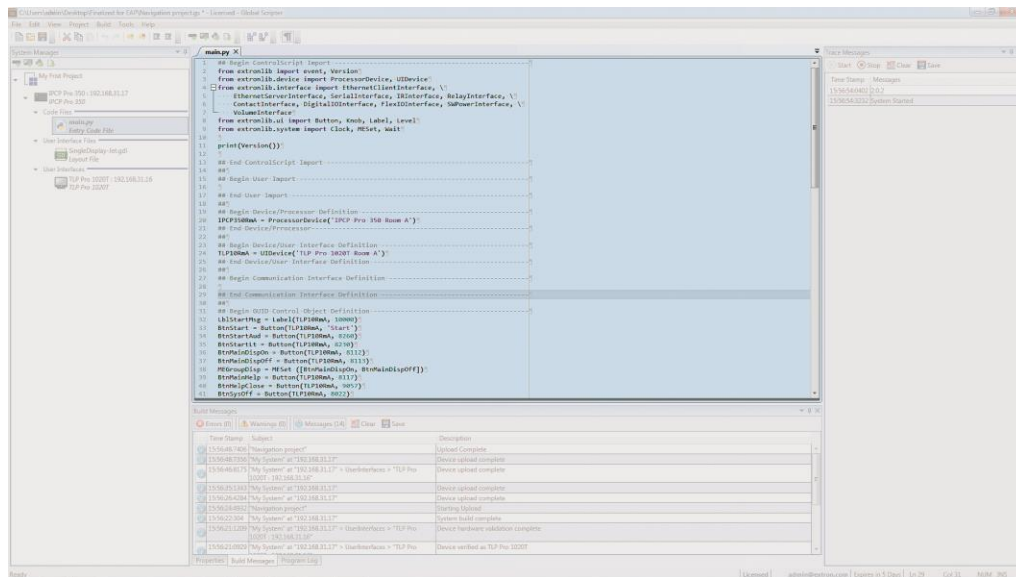


Projects (项目) 是系统管理器的最高等级。现阶段，Global Scripter 同时只能打开一个项目文件。

在 project (项目) 中添加控制系统主机，里面会自动包含名为 main.py 的入口代码文件。

3.1.2 代码编辑器 (Code Editor)

在系统管理 (system manager) 中双击相应的代码文件，代码编辑器窗口中会显示对应的代码。

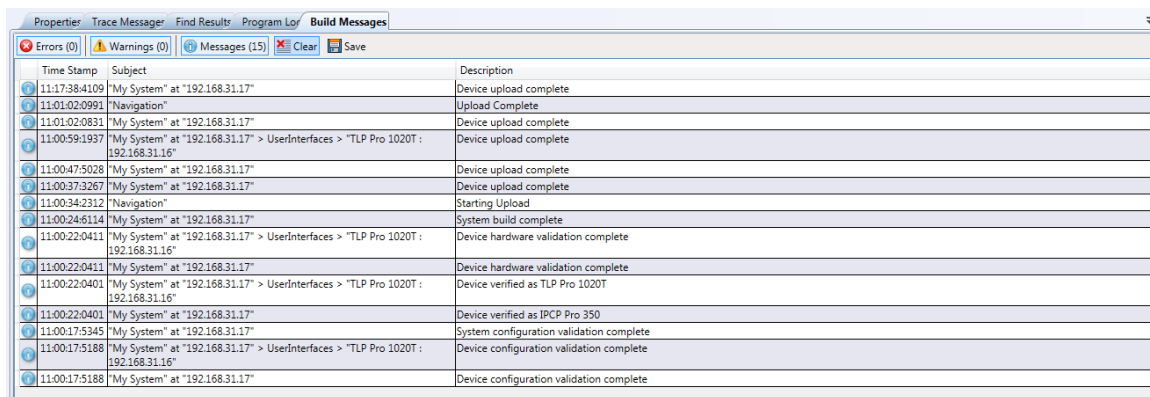


代码编辑器窗口提供标准的代码编辑功能和辅助编写代码的特性，如整理行距、修剪行尾空格、将 tab 制表符转换成空格（这些选项可以在“**Edit**”子菜单中的“**Blank Operations**”中找到）。使用“**Comment/Uncomment**”的子菜单，相关代码行可以进行注释或取消注释。

代码编辑器可设置不同的显示方式，例如行号、syntax 语法高亮显示、tab 制表符转换设置、增加垂直标尺等。这些辅助功能都可以通过菜单、热键实现，或在编辑区单击鼠标右键对应的菜单也能实现同样的功能。

3.1.3 诊断面板 (Diagnostics Panels)

Diagnostics Panels 诊断面板默认情况下会以分页的形式显示在软件的底部。每一个标签页都可以从诊断面板中移开并放置在软件中任何位置，甚至 Global Scripter 窗口外面。



Time Stamp	Subject	Description
11:17:38.4109	"My System" at "192.168.31.17"	Device upload complete
11:01:02.0991	"Navigation"	Upload Complete
11:01:02.0831	"My System" at "192.168.31.17"	Device upload complete
11:00:59.1937	"My System" at "192.168.31.17" > UserInterfaces > "TLP Pro 1020T : 192.168.31.16"	Device upload complete
11:00:47.5028	"My System" at "192.168.31.17"	Device upload complete
11:00:37.3267	"My System" at "192.168.31.17"	Device upload complete
11:00:34.2312	"Navigation"	Starting Upload
11:00:24.6114	"My System" at "192.168.31.17"	System build complete
11:00:22.0411	"My System" at "192.168.31.17" > UserInterfaces > "TLP Pro 1020T : 192.168.31.16"	Device hardware validation complete
11:00:22.0411	"My System" at "192.168.31.17"	Device hardware validation complete
11:00:22.0401	"My System" at "192.168.31.17" > UserInterfaces > "TLP Pro 1020T : 192.168.31.16"	Device verified as TLP Pro 1020T
11:00:22.0401	"My System" at "192.168.31.17"	Device verified as IPCP Pro 350
11:00:17.5345	"My System" at "192.168.31.17"	System configuration validation complete
11:00:17.5188	"My System" at "192.168.31.17" > UserInterfaces > "TLP Pro 1020T : 192.168.31.16"	Device configuration validation complete
11:00:17.5188	"My System" at "192.168.31.17"	Device configuration validation complete

属性栏 (Properties)

系统管理器 (System Manager) 选中的项目信息，将会在属性栏中显示。

程序信息栏 (Build Messages)

程序信息栏 (Build Messages) 会显示当前上传程序的状态，警告 (Warning)，及错误 (Error) 相关的信息。你可以根据需要保存，删除，或隐藏这些信息。

跟踪信息栏 (Trace Messages)

跟踪信息栏 (Trace Messages) 会显示当前运行的程序中 print 指令所输出的结果，可根据需要对显示的信息进行筛选，以方便程序的调试。

程序日志栏 (Program Log)

程序日志栏 (Program Log) , 提供当前程序的运行状态或相关报错的信息。

搜索结果栏 (Find Results)

搜索结果栏 (Find Results) , 显示最近的查找操作结果。

状态栏 (Status Bar)

状态栏 (Status bar) 显示 License 授权信息、鼠标指针位置、键盘锁定状态信息。



4 Python 的基础

Python 是一种面向对象的高级编程语言，在执行前不需要预先编译。其语法和结构简单易懂。Python 的程序会以 .py 的扩展名保存为文本文件，可以使用任意的文本阅读/编辑软件阅读程序代码。

4.1 语法规范

Python 相对其它编程语言而言，对标点符号和特殊字符的要求不那么严格，但是对代码的缩进有严格要求。在 Python 的每一行代码的起始端，空格的作用都是十分重要的，无论是空格按键、tab 制表键还是换行符，都会在程序中生成空格。然而，一些用于编写程序的文本编辑软件会将 tab 制表键转换成一系列的空格，所以编写 Python 程序或者使用由第三方文字编辑软件创建程序时，请注意这些软件是如何处理 tab 制表键和空格键的，因为 Python 使用缩进来定义代码功能块，而非标点符号或者“BEGIN”和“END”指令。所以正确使用空格可以使 Python 代码更容易阅读。

Python 对字母大小写敏感。比如标识符：Touch_Panel、touch_panel、TOUCH_PANEL，虽然都有一样的字母，并代表相同的物理对象，但由于它们大小写不匹配，Python 会把它们当成不同的对象。当程序员在编写 Python 程序时，必须区分标识符的大小写。可以只用小写字母或者只用大写字母来解决问题，但是我们不太建议这样做，因为这会影响程序的可读性。

在 www.python.org 网站上有关 Python 基础知识的文档，例如编码规范 PEP8：Style Guide for Python Code (<https://www.python.org/dev/peps/pep-0008/>)。也可以参考本文档的附录 A 查看详细信息。

4.2 Extron 编程的标准规范

Extron 遵从 PEP 8 (Python 编码规范 8) 的代码规范指南，同时，我们也建议您参考以下几个并应用在您的代码中：

4.2.1 通用建议

- 代码的可读性很重要，因为您读代码的时间将会多于您写代码的时间。
- 有效标注相关代码的功能，以帮助您和其他人了解您所写的代码，并确保代码的注解信息和代码的功能一致。
- 可以自定义模块提供程序调用，并将所需要的参数传递过去完成指定的工作，以提高效率。

- 可能的话，一行只写一条语句。

更多的有益的建议也可参考 PEP 20。

4.2.2 标识符 (Identifiers)

变量和函数标识符遵循“驼峰式”写法。即关键字第一个字母大写其余部分小写，而单一大写或小写的标识我们不建议使用，常量的命名可采用大写字母和下划线的组合。除了常量外，尽量不要使用下划线，除非是 Python 的特别用途。以下是一些命名范例，简要演示了 Extron 相关标识符的命名方式。

对象	范例
Device Object (设备对象)	Room103Processor, Room103PodiumTLP
Interface Object (接口对象)	Room103Projector
Button Object (按键对象)	BlurayPlayButton MainPageStartButton
Level Object (电平对象)	MainAudioLevel Mic1Level
Label Object (标签对象)	RoomNameLabel PageNameLabel
Knob Object (旋钮对象)	MainVolumeKnob
Wait Object (等待对象)	WaitResetTextLabel WaitPopupDelay
Clock Object (时钟对象)	ScheduledShutdownClock

4.2.3 行长度和续行

每行代码长度建议不要超过 80 个字符，包括段落标记。当一行代码长度超过 80 个字符时，可以使用反斜杠字符 (\) 进行续行。Global Scripiter 里的代码编辑器 Code Editor 窗口，也会默认显示相应的纵列标尺来显示代码长度限制范围。如果字符没有超过这条界限，尽可能不要使用反斜杠字符 (\)。

4.2.4 字符串

String 字符串用单引号来标注，如果有多行字符串，在字符串开头和结尾分别使用三个引号。如果字符串中包含单引号，则在字符串两端使用双引号。根据 PEP 8 和 PEP 257 文的建议，三个双引号用于对代码的功能进行注释。

4.2.5 ID 范围

硬件按钮（如 TLP PRO 320M）的 ID 号码的分配，通常从 65535 开始然后递减。其它软按键、标签、用户界面的控制组件则由 1 开始递增。Extron 提供的用户界面模版的 ID，标签（Label）由 4000 开始，按钮（Button）从 5000 开始，用户有充足的 ID 余量用于界面设计。页面和弹出式界面的 ID 则由 GUI Designer 用户界面编辑器软件分配并且不能更改。

4.3 解析器和库

Python 所写的程序和模块直接在 shell 解析器中执行，这样可以保护控制系统和程序员不会因未知的错误而导致不可预计的结果。为了保护系统的安全性和稳定性，不是所有的 Python 元素（指令和模块）都会整合到控制系统中，由于控制系统的硬件中没有相应控制台接口输出这些内容，所以个别元素会被更合适控制系统的函数替代。例如，在 Python 里 print 指令可以在运行解析器的电脑控制台窗口中显示字符内容，但在 Pro 控制系统中，print 指令会输出用户追踪的状态信息，并显示在程序里的 Trace Message Panel 或 Toolbelt 工具栏软件中。未开放的 Python 元素列表可以参考附录 B。

此外，Extron 在库中加入了适用于中控系统的元素，扩展了 Python 编程语言的使用范围。指定的模块主要是由控制系统的通讯接口以及相关的函数所使用。这些模块和解析器就是 ControlScript – 控制脚本。

4.4 面向对象的编程

Python 支持过程式及函数式编程，也支持面向对象的编程方法。Pro 控制系统的程序同样可以采用上述方法编写。对于熟悉其它编程语言或刚开始学习 Python 的人来说，很容易上手。

每个对象都包含属性和方法。对对象进行描述的是属性，对对象进行处理的是方法。

4.5 列表和字典

在 Python 中，数据可以有多种存储方法，可使用列表（List）或键值匹配的字典（Dictionary）这两种方法。列表是一种按顺序收集数据的简单方法，这些数据的类型可以不一致。以下为一组数字的列表：

```
Numbers = [1, 2, 3.0, '4', 'five', 2*3, '8-1', 8.0, 9, 0xA,]
```

列表中的元素都是有编号的（请记住 Python 中的列表元素编号从 0 开始递增），这些元素可以通过编号分别访问。如 Numbers[4] 会返回字符串 “five”，Numbers[5] 会返回数值 6（先计算 2*3 的结果，得出整数 6 后，存在编号为 5 的位置）。列表可以使用在程序的任何地方，类似其它编程语言中的数组。

字典（Dictionary）以键（key）和对应数值的方式存储，字典中的值不是按序排列，只能通过键读取数值。虽然听上去感觉功能十分有限，但实际上功能很强大。在字典中的键通常用单词来定义，之后用冒号标识对应的数值。（使用单个引号或者双引号都可标注字符串，本文会优先使用单引号进行标注，除非字符串内已经含有单引号）。在以下的例子中，使用字典定义密码数值，从而控制访问系统特定功能的权限：

```
PassCodes = {  
    'admin': '1234',  
    'user': '5678',  
    'null': ' ',  
}
```

首先将输入的密码和字典中的内容做对比，只有匹配的密码才可获得程序赋予的访问权限。同时，因为字典里的对象是可变的，所以可以通过程序修改密码。

4.6 循环和迭代

Python 的其中一个优势在于可以基于列表来进行动态循环。以下的例子中，通过循环 6 次来生成相应的按钮对象（相对应的 ControlScript class 对象会在之后的章节中讲解），每一个在 InputButtons 列表的对象会附加在 InputButtonObjects 的列表里，变量 InputButtonID 被赋予当前循环在列表中对应的值，循环启动时，InputButtonID 的值为 601，第 2 次的值为 602，以次类推。

若要在系统中添加输入按键，**InputButtons** 列表也做相应改变。但是，第三行的循环语句保持不变。

```
InputButtons = [601, 602, 603, 604, 605, 606,]
InputButtonObjects = []
For InputButtonID in InputButtons:
    InputButtonObjects.append(Button(TLP, InputButtonID))
```

如果需要增加 4 个输入按键，只需要在列表中添加这些按键对应的 ID，循环 10 次即可完成按键的定义，另外请注意列表里的数据不一定要连贯。

```
InputButtons = [601, 602, 603, 604, 605, 606, 609, 611, 617, 614,]
InputButtonObjects = []
For InputButtonID in InputButtons:
    InputButtonObjects.append(Button(TLP, InputButtonID))
```

当定义好按键对象后，可以定义按键的功能，如按键按下后会触发什么动作，这种迭代循环的方法将会经常在该文档中出现。

理解 Extron 如何使用 Python 编程最好的方法就是实践。接下来会通过几个例子逐步探讨需要使用的控制脚本语言。本文档并非完整的 Extron 控制系统编程指南，也不是培训课程的替代品，只是抛砖引玉，方便您更好地了解正式培训的内容。

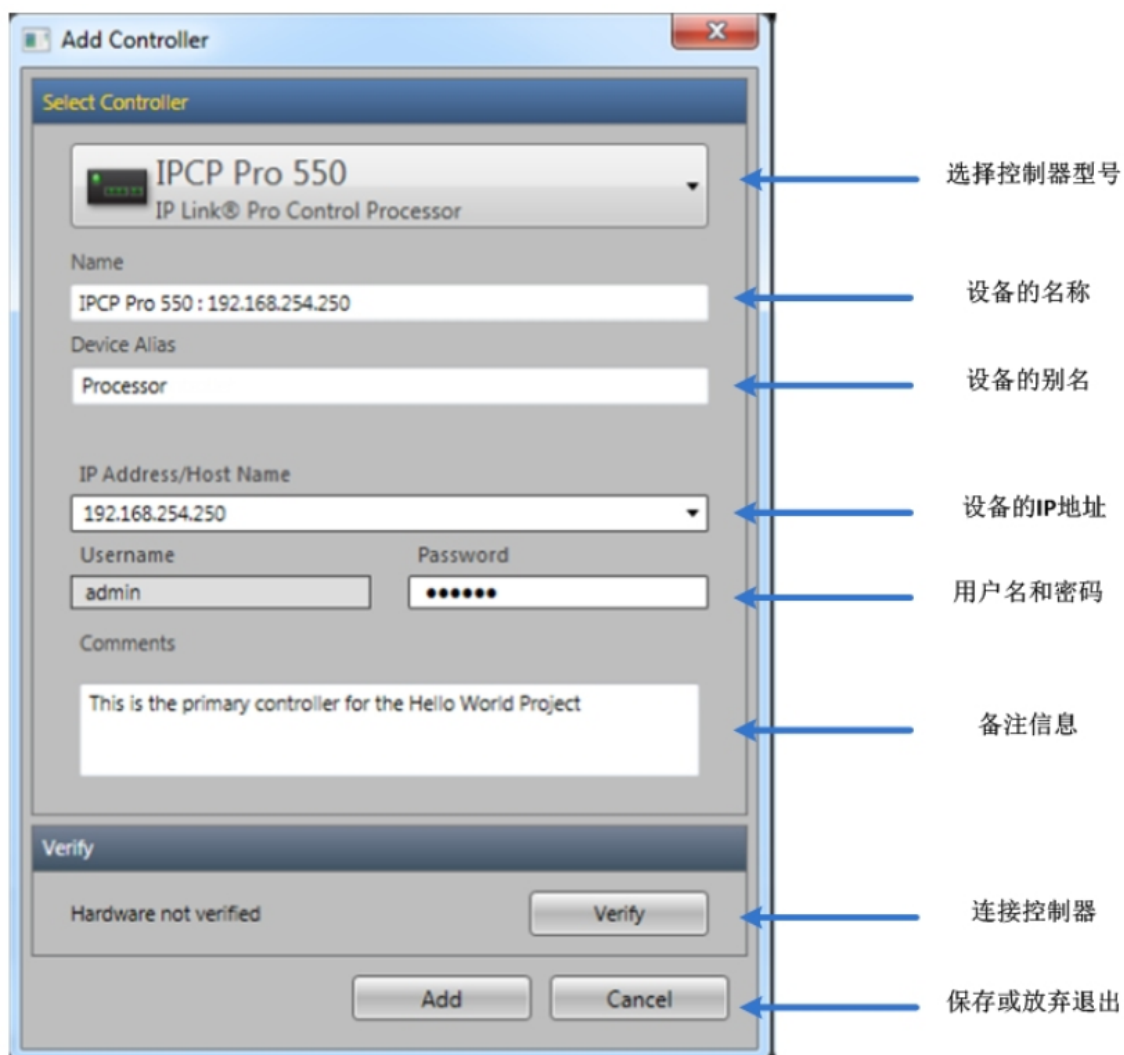
5 从零开始学编程

5.1 在 Global Scriptor 中新建项目

在 Global Scriptor 里的一个项目会包含很多元素：控制处理器，用户界面，代码文件和其它的关联文件，比如触摸屏界面文件。在同一个项目中，不同的系统可以共享同一个文件。如果一个文件有改动，系统中引用过该文件的部分也会同步更新。项目文件会以 .gs 的文件扩展名进行保存。

创建一个新的项目文件，需要从 **File** 菜单中点击 **New**。项目的说明信息可以在 **Project** 菜单里的 **Project properties** 进行设置。

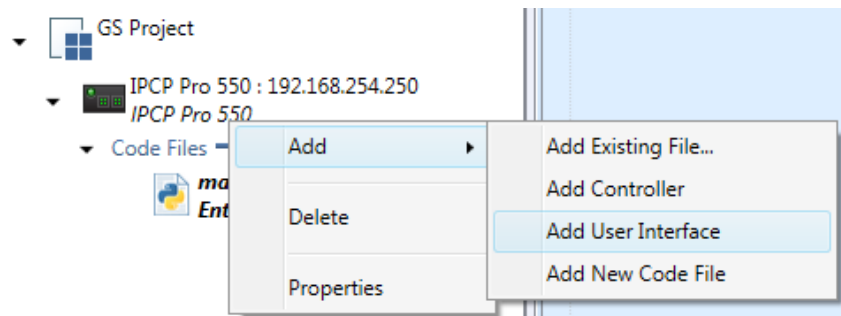
当项目生成后，系统可以添加控制主机，通过点击在 System Manager 工具栏上的 Add controller 按钮后，会弹出 Add controller 添加控制器的对话框。



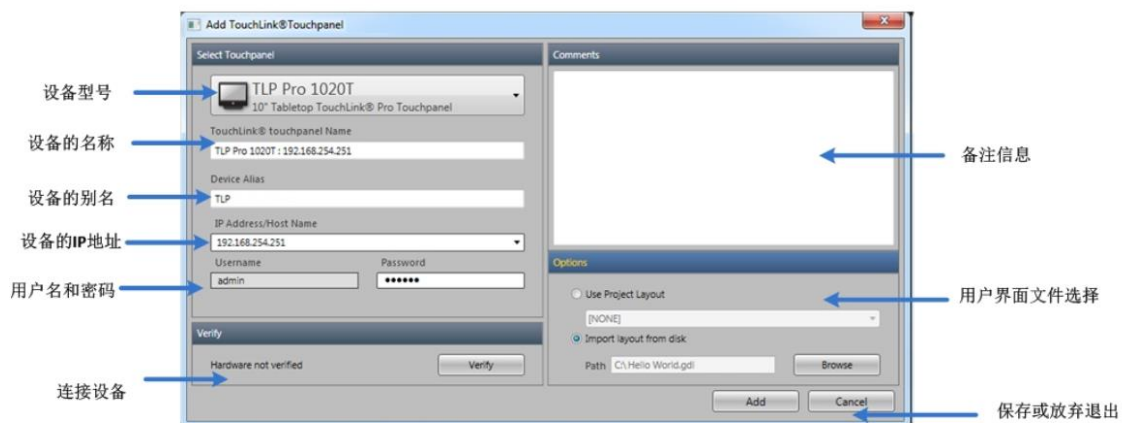
输入系统所需要的信息，包括设备别名（Device Alias），该名称会在之后编程过程中用到。命名的规范要清晰表明设备是什么，尽量避免和其他设备混淆。如果控制器和电脑是在同一个网段，可以点击 verify 按钮检查和控制器的通讯是否正常。当所有的属性都按需填写好后，点击添加 Add 按钮完成。

现在第一个控制处理器已经添加到项目系统中，相关的 Python 代码文件，main.py 也会自动生成。该文件已包含导入 Extron 库和其它用在该系统的语句，import 语句的功能我们会在之后的文档中再详细讨论。

接下来可以根据需要添加用户界面（User Interfaces）或者第 2 个控制处理器，通过点击在 System Manager 工具栏上的相应按钮进行添加，或者在 System Manager 工具栏或 Main 工具栏右单击菜单中进行添加。



选择添加用户界面（Add user Interface），会弹出对应的窗口，选择相关的用户界面并输入相关的属性。可选择添加 GUI 用户界面文件，也可以暂时不添加进项目中，留待之后添加，并通过 Properties 菜单选项进行更改。



其它 GUI 界面文件，音频文件，图纸、系统文档等都可以通过在 Main 工具栏中右单击弹出菜单中的 Add Existing File 进行添加。

当控制器和 GUI 界面添加到项目文件中后，程序员就可以在入口 Code 代码文件 main.py 中进行编程了。

5.2 入口程序代码 main.py

系统会保留一些特定的 Python 代码文件作为入口程序，该文件以 “ main.py ” 进行命名。该文件是控制处理器执行的主程序，所有的其它程序文件、模块，都可导入到该程序文件里。

导入库文件

main.py 文件里的第一段语句是用来导入常用的 Extron 库及其对象。

```
from extronlib import event, Version
from extronlib.device import ProcessorDevice, UIDevice
from extronlib.interface import EthernetClientInterface, \
    EthernetServerInterface, SerialInterface, IRInterface, RelayInterface, \
    ContactInterface, DigitalIOInterface, FlexIOInterface, SWPowerInterface, \
    VolumeInterface
from extronlib.ui import Button, Knob, Label, Level
from extronlib.system import Clock, MESet, Wait
```

其它 Python 文件中定义的类和函数可以导入到 main.py 文件中。这些文件可通过 System Manger (系统管理栏) 里的 Add New Code File (增加新的代码文件) 添加，也可通过工具栏中的快捷键进行添加。

5.3 视音频系统中的 “Hello World!”

大多编程语言的教程中，都会让您尝试输入一些指令或者字符串，并显示相应的结果。通常都会使用开发工具对主设备发送指令文字 “Hello World!” 。Python 也一样，比如下面的指令代码将会在 Python 控制终端显示文字 “Hello World” 。

```
Print ('Hello World!')
```

但是，Extron Pro 系列的控制处理器中是没有专门的 “ 主输出设备 ” 来显示信息的。因此任何的 Print 指令信息，都是通过 GS 软件里的 Trace Message (追踪信息) 窗口进行显示。

```
Print ('Hello World!')
```

我们可以在新增加的用户界面上显示 “Hello World!” 信息。要实现这个功能，需要对用户界面和控制对象进行实例化。使用 Extron Library API (extronlib) 和其自定义的类可以轻松实现这个功能。

在接下来的例子中，首先我们添加一个用户界面，并将 GUI Designer 制作的触摸屏界面文件导入项目和控制处理器进行关联。在程序代码中，触摸屏的用户界面使用 `UIDevice` 类进行实例化，其参数为添加触摸屏硬件时所设置的屏幕的别名 ‘TLP’。之后使用 `Label` 类实例化触摸屏界面中的标签 **MessageLabel**，使用 ID 号码 2 作为标签的另一个参数（ID 号在 GUI Designer 软件中已有定义）。之后使用 **SetText** 方法设置标签 **MessageLabel** 中显示的内容，程序运行后，“Hello World!” 信息就可以在触摸屏上的相关标签位置进行显示了。

```
TouchPanel = UIDevice('TLP')

# Send Message to Trace window
print('Hello World!')

# Send Message to label with ID 2 on the UI Device TouchPanel
MessageLabel = Label(TouchPanel, 2)
MessageLabel.SetText('Hello World!')
```

需要注意的是，实际项目中，一个完整的控制系统项目会包含控制处理器和通过其端口控制的周边设备，不单只是将信息传送到诊断日志或触摸屏。

要实现这些功能，处理器需要将要使用的控制端口进行实例化，就好像上面例子中将屏幕进行实例化一样。每一个设备（控制处理器或触摸屏）或控制端口在使用前都需要在程序代码中进行实例化。程序中需要将标识符和硬件对应的别名进行关联。在下面的例子中，控制处理器使用 `ProcessorDevice` 类进行相应的实例化，其参数为程序员添加控制器时设置的别名 “Processor”。而继电器端口使用 `RelayInterface` 类进行相应的实例化，其参数为控制器的名称和指定继电器端口的 ID。

```
ProProcessor = ProcessorDevice('Processor')
Relay1 = RelayInterface(ProProcessor, 'RLY1')
```

现在控制器和继电器已经定义好了，在这个初学者项目中还需要增加和用户互动的程序。比如说，按下按键后，需要在软件的 Trace Message（跟踪界面）中显示 “Hello World!” 的信息，表明按钮有动作；按钮的反馈颜色需要有变化；并且开启或者关闭指定的继电器端口。另外，为了使程序更具可读性，可以在注解中简要说明程序的功能。编写

按钮功能的程序时，需要对它进行实例化，就像我们实例化用户界面中的标签（Label）那样。在范例 1 中，Button 类使用两个参数，用户界面和界面的 ID 号码。

下面定义一个 **Button1Press** 的函数，用作按钮的切换动作。函数的主体内容需要使用 4 个空格缩进，在代码的第一行写 `"""Button 1 on Welcome Page"""`（欢迎界面的按钮 1），其用途是对这个函数进行说明。之后使用 **print** “输出 Hello World!”。接着使用一段条件语句，对用户界面中对应的按钮状态进行判断，按钮的视觉状态编号可以使用 GUI Desinger（用户界面编辑软件）查看。当按下按钮时，控制器会根据当前按钮的视觉状态，决定对控制器的继电器端口的操作方式是打开或者关闭。在最后一行代码中，每次按下 Button1 按键，触摸屏中已经实例化过的按键 Button1 会被指派运行 **Pressed** 事件中的 **Button1Press** 函数。请将这些代码上传到控制处理器，并测试。现在，我们已经完成了 Extron 控制系统编程的第一个练习。

```
# Toggle the state of a relay when a button is pressed.
Button1 = Button(TouchPanel, 1)

def Button1Press(button, state):
    """ Button 1 on Welcome Page"""
    print('Hello World!')
    if button.State == 1:
        button.SetState(0)
        Relay1.SetState('Open')
    else:
        button.SetState(1)
        Relay1.SetState('Close')

Button1.Pressed = Button1Press
```

无论您以往是否接触过 Python 的编程，或者对代码指令不太熟悉，通过以上的练习，您应该会发现 Extron 的程序代码通俗易懂。因为 Python 的程序大多采用大纲格式进行编排并具有清晰的层次。另外之前也提过，空格在 Python 程序代码中很重要，因此你会看到定义的函数内容都是用空格做缩进处理。另外，每一行末也不需要使用分号作为行休止符。

Event decorator 事件修饰器

上面的例子最后一行代码中，函数 **Button1Press** 是指派给 Button 对象的方法，每次 Button1 按下时都会调用这个函数。这样的写法对于只有几行的程序代码来说，可读性不会有问题。但是，如果这个函数代码很长的话，就会变得难以阅读，编程人员也会很容易忘记函数和实例化的按键是如何关联在一起的。有鉴于此，Extron 建立了一个 python

对象 – event decorator (事件修饰器)，可以使前面的代码表达变得更加清晰易懂，进一步提高程序的可读性。

使用 event decorator (事件修饰器) 的方法是，先定义按键和触发事件的条件，之后再定义按键关联的实际函数功能。

在下面的代码中，@event (事件修饰器) 有两个参数，分别为 **Button1** 和 **Pressed**。之后 Button1 的属性会被传递至函数 Button1Press 函数，很上一段实例代码一样，当触摸屏相应按键按下时，将执行函数 Button1Press 的功能。

```
# Toggle the state of a relay when a button is pressed.
Button1 = Button(TouchPanel, 1)

@event(Button1, 'Pressed')
def Button1Press(button, state):
    """Button 1 on Welcome Page"""
    print('Hello World!') # Send Message to Trace window
    if button.State == 1:
        button.SetState(0)
        dvRelay1.SetState('Open')
    else:
        button.SetState(1)
        dvRelay1.SetState('Close')
```

需要注意的是，Python 中的事件修饰通常是以字符 “@” 为起始符号。本文档中的大多数事件的示范代码都是使用@event。当然，两种方法都是正确的，产生的效果也是相同的。

在中控系统编程中事件修饰器有许多不同的用途，但是主要功能是用于处理程序的“入口”。

6 编写用户界面导航程序

触摸屏界面上需要编程的所有控制对象都需要进行定义赋值。和前面实例化定义按钮的过程一样，创建标识符并定义相应的对象（button 按钮，label 标签，level 音量电平等）。然后定义其关联的功能函数。创建用户界面的导航的方法和前面编写按钮输出“Hello World”的例子类似，不过不是使用 RelayInterface 类，而是使用 UIDevice 类进行实例化。之后使用 ShowPage()方法显示相关的页面。括号里面的参数可以是用户界面的 ID 号码或者是其对应的名称（请查阅 GUI 文件获取），这里我们建议使用页面名称，而不是 ID 编号，这样会增加程序的可读性。但需要注意的是，名称要和 GUI Designer（用户界面设计软件）里的名称完全一致。

接下来的例子中将使用 TLP 1020T 触摸屏的 Single Display（单显示）模版。在定义 1020T 触摸屏之后，和页面导航相关的按键也会被定义。下面代码中实例化了 **StartButton** 和关联的按键事件，按键按下后会显示 ID 号为 6 的页面。之后，实例化了 **SystemOffButton**，按键按下后会显示 ID 号码为 10 的页面。

```
TLP = UIDevice('PrimaryTouchPanel')

StartButton = Button(TLP, 8000)
@event(StartButton, 'Pressed')
def StartButtonPressed(button, state):
    TLP.ShowPage(6)

SystemOffButton = Button(TLP, 8022)
@event(SystemOffButton, 'Pressed')
def SystemOffButtonPressed(button, state):
    TLP.ShowPage(10)
```

下面代码的功能和前面一样，但没有使用按键 ID，而是使用了按键和页面的名称。

```
TLP = UIDevice('PrimaryTouchPanel')

StartButton = Button(TLP, 'Start')
@event(StartButton, 'Pressed')
def StartButtonPressed(button, state):
    TLP.ShowPage('Main SD')

SystemOffButton = Button(TLP, 'System Off')
@event(SystemOffButton, 'Pressed')
def SystemOffButtonPressed(button, state):
    TLP.ShowPage('Start')
```


Popup Page（弹出式界面）导航的程序也使用类似的方法进行编写，使用 ShowPopup() 方法可显示弹出式页面，HidePopup() 方法可隐藏弹出式页面。下面的示例代码可以在开始界面中显示和关闭 Start Audio Control 的弹出式页面。

当按下 Start Audio Control 按键时，会显示 Start Audio Control 的弹出界面并持续显示。当按下 Close Start Audio Control 按键时，会隐藏 Start Audio Control 的弹出界面。

```
StartAudioControl = Button(TLP, 'Start Audio Control')
@event(StartAudioControl, 'Pressed')
def StartAudioControlPressed(button, state):
    TLP.ShowPopup('Start Audio Control', 0)

CloseStartAudioControl = Button(TLP, 'Close Start Audio')
@event(CloseStartAudioControl, 'Pressed')
def CloseStartAudioControlPressed(button, state):
    TLP.HidePopup('Start Audio Control')
```

在前面的例子中，页面的导航是通过在每个按键的事件中分别调用函数实现的。也可通过调用一个自定义函数实现相同的功能（类似 Global Configurator Professional 软件里的宏）。在下面的例子中，我们会建立一个自定义函数，它可以被多个按键调用，并显示不同的弹出界面。

TLP Pro 1020T 的 Single Display（单显示模版）里，主界面中信号源如下图所示：



下面这段代码对每个按键都做了实例化，并创建了事件修饰器。如果有按键被选中，创建的 **SeleceInputButtonPressed** 函数会发送相关的信息到 Trace window（软件的跟踪窗口）。后面的条件逻辑语句会根据按钮的 ID 号码判断哪个按键被选中，并将变量 CurrentInput 的值更改为当前选中的信号源设备，在另外一个自定义函数 ShowSourcePopup 中也会使用这个变量。

```
SelectLaptop = Button(TLP, 8050)
SelectPC = Button(TLP, 8052)
SelectBluray = Button(TLP, 8054)
SelectDocCam = Button(TLP, 8058)
SelectTuner = Button(TLP, 8060)
SelectAux = Button(TLP, 8062)
```

```
@event(SelectLaptop, 'Pressed')
@event(SelectPC, 'Pressed')
@event(SelectBluray, 'Pressed')
@event(SelectDocCam, 'Pressed')
@event(SelectTuner, 'Pressed')
@event(SelectAux, 'Pressed')
def SelectInputButtonPressed(button, state):
    print('button pressed: {0}, ID - {1}'.format(button.Name,
button.ID))
    if button is SelectLaptop:
        CurrentInput = 'Laptop'
    elif button is SelectPC:
        CurrentInput = 'PC'
    elif button is SelectBluray:
        CurrentInput = 'Bluray'
    elif button is SelectDocCam:
        CurrentInput = 'DocCam'
    elif button is SelectTuner:
        CurrentInput = 'Tuner'
    elif button is SelectAux:
        CurrentInput = 'Aux'
    else:
        CurrentInput = 'NotSelected'
    print('CurrentInput: {0}'.format(CurrentInput))
    ShowSourcePopup(CurrentInput, TLP)
```

ShowSourcePopup 函数可以写在程序中的任何地方，该函数会依据 **CurrentInput** 变量的数值来显示对应的弹出式页面。

```
def ShowSourcePopup(input, host):
    if input == 'Laptop':
        host.ShowPopup('Laptop 1')
    elif input == 'PC':
        host.ShowPopup('PC 1')
    elif input == 'Bluray':
        host.ShowPopup('Blu-ray 1')
    elif input == 'DocCam':
        host.ShowPopup('Doc Cam 1')
    elif input == 'Tuner':
        host.ShowPopup('Tuner 1')
    elif input == 'Aux':
        host.ShowPopup('AUX 1')
    else:
        print('No matching popup')
```


如果这些代码可以使第一个触摸屏正常工作，您可以也将其套用在其它的 GUI 用户界面上。前提条件是这些界面中信号源按键的 ID 编号和弹出界面和第一块触摸屏一致（只要你使用 Extron 的用户界面模版，模版里面的 ID 号和弹出界面的名称都是一样的）。如果要让这两个触摸屏实现相同的功能，相关的信号源按钮需要进行相同的实例化操作，并关联相应的事件。

```
TLP720 = UIDevice('SecondTouchPanel')

SelectLaptop720 = Button(TLP720, 8050)
SelectPC720 = Button(TLP720, 8052)
SelectBluray720 = Button(TLP720, 8054)
SelectDocCam720 = Button(TLP720, 8058)
SelectTuner720 = Button(TLP720, 8060)

@event(SelectLaptop720, 'Pressed')
@event(SelectPC720, 'Pressed')
@event(SelectBluray720, 'Pressed')
@event(SelectDocCam720, 'Pressed')
@event(SelectTuner720, 'Pressed')
```

最后，需要再次调用 **ShowSourcePopup** 的函数。一个用于主触摸屏，另一个用于新定义的 TLP720 触摸屏。现在，只要其中一个界面中的信号源按键被选中，两个触摸屏都会显示相同的弹出式界面。完整的程序代码如下：

```
SelectLaptop = Button(TLP, 8050)
SelectPC = Button(TLP, 8052)
SelectBluray = Button(TLP, 8054)
SelectDocCam = Button(TLP, 8058)
SelectTuner = Button(TLP, 8060)
SelectAux = Button(TLP, 8062)
SelectLaptop720 = Button(TLP720, 8050)
SelectPC720 = Button(TLP720, 8052)
SelectBluray720 = Button(TLP720, 8054)
SelectDocCam720 = Button(TLP720, 8058)
SelectTuner720 = Button(TLP720, 8060)
@event(SelectLaptop, 'Pressed')
@event(SelectPC, 'Pressed')
@event(SelectBluray, 'Pressed')
@event(SelectDocCam, 'Pressed')
@event(SelectTuner, 'Pressed')
@event(SelectAux, 'Pressed')
@event(SelectLaptop720, 'Pressed')
@event(SelectPC720, 'Pressed')
@event(SelectBluray720, 'Pressed')
```

```
@event(SelectDocCam720, 'Pressed')
@event(SelectTuner720, 'Pressed')
def SelectInputButtonPressed(button, state):
    print('button pressed: {0}, ID - {1}'.format(button.Name,
button.ID))
    if button is SelectLaptop:
        CurrentInput = 'Laptop'
    elif button is SelectPC:
        CurrentInput = 'Bluray'
    elif button is SelectDocCam:
        CurrentInput = 'DocCam'
    elif button is SelectTuner:
        CurrentInput = 'Tuner'
    elif button is SelectAux:
        CurrentInput = 'Aux'
    else:
        CurrentInput='NotSelected'
    print('CurrentInput: {0}'.format(CurrentInput))
    ShowSourcePopup(CurrentInput, TLP)
    ShowSourcePopup(CurrentInput, TLP720)

def ShowSourcePopup(input, host):
    if input == 'Laptop':
        host.ShowPopup('Laptop 1')
    elif input == 'PC':
        host.ShowPopup('PC 1')
    elif input == 'Bluray':
        host.ShowPopup('Blu-ray 1')
    elif input == 'DocCam':
        host.ShowPopup('Doc Cam 1')
    elif input == 'Tuner':
        host.ShowPopup('Tuner 1')
    elif input == 'Aux':
        host.ShowPopup('AUX 1')
    else:
        print('No matching popup')
```

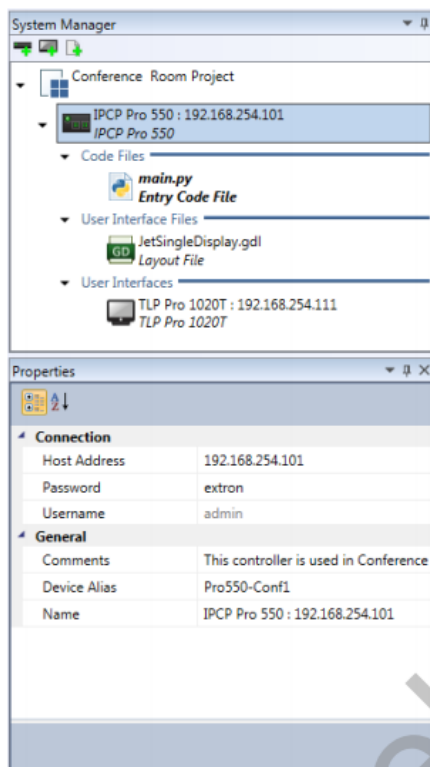
除了刚才讨论过的 `UIDevice` 方法，还有两个方法可以应用到用户界面的导览中。`HideAllPopups()` 以及 `HidePopupGroup()`。`HideAllPopups` 指令无需任何参数，就会关闭所有的弹出窗口。

```
TLP.HideAllPopups()
```

HidePopupGroup 会隐藏指定的分组弹出式界面。例如在 Single Display 单显示模式中，信号源控制的弹出式页面被编成一组，群组 ID 编号为 1。使用以下的代码就可以隐藏这个 ID 为 1 的弹出界面小组。

```
TLP.HidePopupGroup(1)
```

7 编写控制周边设备的程序



Extron Pro 系列控制器的每一个端口都可以与受控的设备进行独立的通讯。要编写控制这些设备的程序，需要创建标识符，并实例化对应的接口。在之前的代码例子中，我们使用了 Relay1 的示范，控制控制器的继电器端口闭合或断开。

每个端口的实例化都有指定的参数。有两个参数是必须的，分别是控制处理器主机名和对应的控制端口的名称。控制处理器主机的名字是指控制处理器实例化时使用的标识符名称。有了这个名称，就可以定义这台中控处理器上的任何端口，并编写相关的程序了。

端口的名称请查阅以下表格（端口名称中的 n 代表端口编号，如 1 或 2 等）：

端口类型	支持的处理器型号	端口名称	接口类代码
Serial Port 串口	IPCP Pro 550, IPCP Pro 350, IPCP Pro 250, IPL Pro S1, S3, S6	COMn	SerialInterface
IR/Serial Port 红外/串口	IPCP Pro 550, IPCP Pro 350, IPCP Pro 250, IPL Pro IRS8	IRSn	IRInterface SerialInterface
Relays 继电器端口	IPCP Pro 550, IPCP Pro 350, IPCP Pro 250, IPL Pro CR88	RLYn	RelayInterface
Flex I/O Port 多功能输入/输出端口	IPCP Pro 550	FION	FlexIOInterface
Digital I/O Port 数字输入/输出端口	IPCP Pro 350, IPCP Pro 250	DION	DigitalIOInterface
Contact Closure Port 接点闭合端口	IPL Pro CR88	CIIn	ContactInterface
12 VDC Port 12VDC 端口	IPCP Pro 550	SPIn	SWPowerInterface
Volume Port 音量端口	IPCP Pro 250	VOLn	VolumeInterface

8 编写与受控设备进行数据通讯的程序

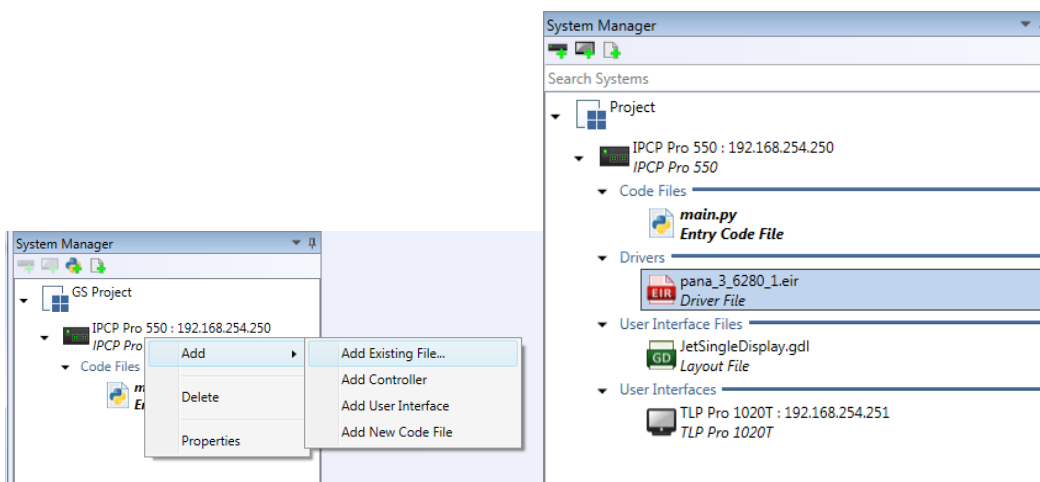
通过控制处理器发送数据指令至受控设备需要 2 个步骤：首先需要对控制处理器的端口实例化，之后可以发送指令给受控设备。在该章节中，首先会介绍红外和单向串口的操作，之后会介绍双向串口和以太网端口的通讯操作。

8.1 通过红外端口发送数据

如“Hello World”的例子一样，红外或者单向的串口端口需要先在系统中进行实例化。调用 interface 接口数据库里的类，对相应的控制处理器的端口添加其所需要的参数进行实例化。接下来的例子里，将会使用 IRInterface 类对 IR 红外端口进行实例化，有 3 个参数，分别为处理器的名称、红外端口号、红外驱动文件名。

```
Bluray = IRInterface(ProProcessor, 'IRS01', 'pana_3_6280_1.eir' )
```

首先要将红外的驱动文件添加到项目文件中，在 GS 软件里 System Manager（系统管理）栏右击鼠标，并选择 **Add**（添加），之后选择 **Add Existing File...**（添加文件），之后在对话框中选择指定的红外驱动文件，并加入到项目文件中。这样，该 IR 红外文件就可以在程序中被处理器使用了。



IRInterface 提供了 3 种方法发送红外指令，分别是 PlayContinuous(), PlayCount() 和 PlayTime()。括号里的第一个参数是需要发送的红外指令所对应的功能名称，该名称可以在 IR 红外驱动文件中找到。PlayContinuous() 是重复发送指定的红外代码，直到执行 Stop() 或其它 Play 方法时才停止发送。PlayCount() 是按指定的次数发送红外指令，如果该参数设置为 0，将会以红外驱动文件里默认的次数来进行发送。PlayTime() 是按指定的

时间长度发送红外指令。具体使用哪一种方法发送指定的红外代码，需要根据受控设备的响应情况而定。

下面的例子中，当 **BlurayPlay** (ID:5000)按钮按下时，处理器将持续发送 'PLAY' 的红外代码，当松开该按钮时，会执行 **Stop()** 方法，停止红外代码的发送。

```
BlurayPlay = Button(TLP, 5000)
@event(BlurayPlay, 'Pressed')
def BlurayPlayPressed(button, state):
    Bluray.PlayContinuous('PLAY')
    button.SetState(1)
    print('Play function sent')

@event(BlurayPlay, 'Released')
def BlurayPlayReleased(button, state):
    Bluray.Stop()
    button.SetState(0)
    print('Play function stopped')
```

我们也可以创建一个自定义函数被多个按钮调用，发送对应功能的红外指令。首先，查阅红外驱动文件，找出对应的红外功能名称，存储在列表中。下面的例子中我们将该列表命名为 **BlurayIRFunctions**：

```
BlurayIRFunctions = [
    'PLAY', 'STOP', 'PAUSE', 'FFWD', 'REW', 'F_STEP', 'R_STEP',
    'UP', 'DOWN', 'LEFT', 'RIGHT', 'ENTER', 'MENU', 'TOP_MENU',
]
```

在 TLP Pro 1020T GUI 用户界面的 Single Display（单显示）模版中，“Blu-ray 1”弹出页面里所对应的按键，使用的 ID 号是由 5000 到 5013。我们对所有的这些按键进行实例化，并在 **Pressed**（按键按下）的事件中做了修饰。之后，按下 **BluRay** 控制页面中的任何按键，都会调用 **BlurayButtonPressed** 的函数，其中将选中的按键作为参数使用。

需要注意的是，在 Python 中列表的位置编号是从 0 开始的，所以进行索引时，'PLAY' 的功能是位于列表是的 0 号位置。仔细观察代码我们可以发现，在 GUI（用户界面）文件中，按钮相对应的 ID 号码是从低到高的顺序进行排列的，可以对应我们所创建的列表位置。如 **PLAY** 是 5000，**TOP_MENU** 是 5013。当按键进行按下时，会执行 **BlurayButtonPressed** 函数，这时候我们可以将该按键的 ID 减去 5000，使用得到的结

果作为位置编号进行索引，正好对应列表中红外功能的名称。控制器会根据 **BlurayIRFunctions** 列表里所对应的功能，发送红外指令，由于发送红外使用的是 **PlayCount()** 方法，所示无须在 **released**（松开按钮）时，再使用 **Stop()** 方法停止红外指令的发送。

```
BlurayPlay = Button(TLP, 5000)
BlurayStop = Button(TLP, 5001)
BlurayPause = Button(TLP, 5002)
BlurayFFwd = Button(TLP, 5003)
BlurayRew = Button(TLP, 5004)
BlurayFStep = Button(TLP, 5005)
BlurayRStep = Button(TLP, 5006)
BlurayUp = Button(TLP, 5007)
BlurayDown = Button(TLP, 5008)
BlurayLeft = Button(TLP, 5009)
BlurayRight = Button(TLP, 5010)
BlurayEnter = Button(TLP, 5011)
BlurayMenu = Button(TLP, 5012)
BlurayTopMenu = Button(TLP, 5013)
@event(BlurayPlay, 'Pressed')
@event(BlurayStop, 'Pressed')
@event(BlurayPause, 'Pressed')
@event(BlurayFFwd, 'Pressed')
@event(BlurayRew, 'Pressed')
@event(BlurayFStep, 'Pressed')
@event(BlurayRStep, 'Pressed')
@event(BlurayUp, 'Pressed')
@event(BlurayDown, 'Pressed')
@event(BlurayLeft, 'Pressed')
@event(BlurayRight, 'Pressed')
@event(BlurayEnter, 'Pressed')
@event(BlurayMenu, 'Pressed')
@event(BlurayTopMenu, 'Pressed')
def BlurayButtonPressed(button, state):
    # define the Bluray buttons to send the appropriate IR function
    ButtonIndex=button.ID - 5000
    Bluray.PlayCount(BlurayIRFunctions[ButtonIndex])
    button.SetState(1)
    print('{0} function sent'.format(BlurayIRFunctions[ButtonIndex]))
```

在下面这段代码中，为了在触摸屏上获得按钮松开后的视觉反馈，我们需要定义 **Released**（松开按钮）事件，将按钮的 **control state**（控制状态）设为 0。这段代码也可作为后续代码的基础结构，比如可以在此基础上增加系统所需要的红外控制功能。

```

@event(BlurayPlay, 'Released')
@event(BlurayStop, 'Released')
@event(BlurayPause, 'Released')
@event(BlurayFFwd, 'Released')
@event(BlurayRew, 'Released')
@event(BlurayFStep, 'Released')
@event(BlurayRStep, 'Released')
@event(BlurayUp, 'Released')
@event(BlurayDown, 'Released')
@event(BlurayLeft, 'Released')
@event(BlurayRight, 'Pressed')
@event(BlurayEnter, 'Pressed')
@event(BlurayMenu, 'Pressed')
@event(BlurayTopMenu, 'Pressed')
def BlurayButtonPressed(button, state):
    ButtonIndex=button.ID - 5000
    button.SetState(0)
    print('{0} function stopped'.format(BlurayIRFunctions[ButtonIndex]))

```

8.2 单向串口发送数据

定义单向串口端的方法和定义红外端口类似。

`SerialInterface` 类定义对象时必须指定对应的参数，这些参数包括处理器的主机名和对应的端口号，用于标识实际使用的中控和对应的物理端口。剩下参数为可选项，如果默认值适合您的受控设备，则可以不填。串口默认的参数如下所示：

```

SerialDevice = SerialInterface(ProProcessor, 'COM1', Baud=9600,
Data=8, Parity='None',
    Stop=1, FlowControl='OFF', CharDelay=0, Mode='RS232'
)

```

控制指令可以用引号标注字符串后，通过相应的端口发送。

```
'POWER ON\r'
```

发送的字符串可以包括一些无法打印显示的符号，如回车符和换行符、或者其它可以用十六进制定义的字符、转义字符等。字符串中可混合使用上述不同类型的字符。

接下来的例子会通过指定中控处理器上指定的串行接口，发送 **POWER ON\r** 指令，实现对受控设备的控制。


```
SerialDevice.Send('POWER ON\r')
```

在接下来的例子中，我们会使用 IPCP Pro 550 的 IR/Serial (红外/串口) 2 号端口，控制 Extron DXP HDMI 矩阵。先实例化处理器，然后用 SerialInterface 类定义 DXP 矩阵。

```
DXPMatrix = SerialInterface(ProProcessor, 'IRS2')
```

当串口端口定义好后，就可以通过 SerialInterface 类里面的 Send() 的方法将指令发送到 DXP HDMI 矩阵了。以下为矩阵输入 1 切换到输出 1 的 SIS 控制指令。

```
DXPMatrix.Send('1*1!')
```

下面的界面导航例子中，不同按钮按下，可触发矩阵不同的输入信号选择。

```
def SelectInputButtonPressed(button, state):
    print('button pressed: {0}, ID - {1}'.format(button.Name,
button.ID))
    if button is SelectLaptop:
        CurrentInput = 'Laptop'
        DXPMatrix.Send('1*1!')
    elif button is SelectPC:
        CurrentInput = 'PC'
        DXPMatrix.Send('2*1!')
    elif button is SelectBluray:
        CurrentInput = 'Bluray'
        DXPMatrix.Send('3*1!')
    elif button is SelectDocCam:
        CurrentInput = 'DocCam'
        DXPMatrix.Send('4*1!')
    elif button is SelectTuner:
        CurrentInput = 'Tuner'
        DXPMatrix.Send('7*1!')
    elif button is SelectAux:
        CurrentInput = 'Aux'
        DXPMatrix.Send('8*1!')
    else:
        CurrentInput = 'NotSelected'
    print('CurrentInput: {0}'.format(CurrentInput))
    ShowSourcePopup(CurrentInput, TLP)
```

8.3 双向串口通讯

定义双向串行端口和定义单向串口通讯端口的方法是相同的，使用的类结构也是类似的。唯一改变的只是对应硬件端口的名称。如果控制 DXP HDMI 矩阵的端口从 IPCP Pro 550 的单向红外/串行端口转移到双向串行端口中，代码中只需要将 **DXPMatrix** 参数中端口的名称，从单向的 'IRS2' 改为双向的 'COM3'，其它代码不需要改动。现在，处理器不仅可以发送代码给矩阵，还可以从矩阵中接收反馈代码。

```
DXPMatrix = SerialInterface(ProProcessor, 'COM3')
```

这种中控和受控设备之间的互动通讯，只是双向串口通讯功能的一部分。处理器可以在收到受控设备的反馈后再发送指令，也可接收来自受控设备未经请求的反馈。比如在矩阵前面板进行信号切换后，矩阵直接将反馈信息发送给控制器。两者之间的差别在于返回的信息是预期的还是非预期的。

8.3.1 异步通讯

受控设备除了接收来自控制系统的指令之外，设备也可能主动传输信息给控制系统。如果你可以预计到这种情况的发生，最好使用异步的通讯方法（asynchronous method）。

8.3.2 解析从设备回传的数据

通过处理器分析设备回传数据的过程称为解析（Parsing）。为了对数据进行解析，需要生成一个事件的处理器（event handler）接收并处理信息。通常都会使用串口作为通讯方式，异步通讯模式下，常使用 Send() 的指令，向受控设备发送控制指令。Pro 系列的控制系统工作在异步的通讯模式时，控制器在收到受控设备发送的信息时，会立刻发送返回数据。但由于通讯的波特率速率、信息的长度、字符间的延迟、硬件握手与否、处理器的负荷等因素，受控设备向控制器发送的信息可能未必完整。出于这个原因，需要建立一个变量保存来自受控设备的数据。接收到的数据都会填充到这个变量中。在以下的例子中，创建了变量 **MatrixBuffer** 用于接收 DXP 矩阵的信息，初始值设置为空的字符串。

```
MatrixBuffer = ''
```

在之前的例子中，已将矩阵端口的定义为 `DXPMatrix`。接下来用这个端和 `'ReceiveData'` 参数来定义接收数据的事件（event）。

```
@event(DXPMatrix, 'ReceiveData')
```

然后，需要定义一个函数处理接收的信息。以下的例子中，会生成一个用于解析的函数，命名为 **MatrixParse**。函数需要 2 个参数，一个是控制器的数据发送和接收端口，另一个是代表从设备接到的信息。另外由于接收的信息是字节（Byte）格式，要使用 Python 里的 `decode()` 指令将字节格式的信息转换成字符串格式的信息。

当执行 **MatrixParse** 函数时，我们可以继续引用之前我们创建的变量 **MatrixBuffer**，无需新建变量。在变量前加入关键字 `'global'`，表示函数引用了一个变量 **MatrixBuffer** 并且，将其定义为全局变量，这样就可以在函数里修改变量的值。之后的代码中，我们将函数接收的信息转换成字符串格式后，存储到 **MatrixBuffer** 变量中。

```
def MatrixParse(interface, response):  
    global MatrixBuffer  
    MatrixBuffer = MatrixBuffer + response.decode()
```

在处理数据之前，需要确定回传的数据是否已经全部接收完成。根据矩阵说明书中 SIS 指令协议，如果信息包含回车符和换行符的话，代表设备已完成信息的回传。我们可以生成一个条件逻辑来评估 **MatrixBuffer** 是否含有这些符号。如果确认回传信息已经完成（回传信息包含 `\r\n`），使用 **DelimiterIndex** 变量存储第一组回传数据的位置信息（回传的信息可能包含不止一个回车和换行符，若我们仅处理其中一个，就需要对 **MatrixBuffer** 变量进行分割）。用 **DelimiterIndex** 从 **MatrixBuffer** 中检索数据，将第一部分信息的数值单独抽离（第一个 `\r` 前的所有字符串）并存储在变量 **CurrentResponse** 中，这就是随后要处理的数据。如果 **MatrixBuffer** 变量中还有残余信息，则使用 `DelimiterIndex+2` 作为索引位置（第一个回车符和换行符之后的内容）为 **MatrixBuffer** 变量重新赋值。

如果回传未完成（没检查到有 `\r\n`），打印信息提示需要等待更多的数据，直到到下一次事件处理器被触发。

```
if MatrixBuffer.rfind('\r\n') > 0:
    DelimiterIndex = MatrixBuffer.rfind('\r')
    CurrentResponse = MatrixBuffer[:DelimiterIndex]
    MatrixBuffer = MatrixBuffer[DelimiterIndex+2:]
else:
    print('waiting for more data')
```

实际项目中，我们常常需要根据通讯的协议，还有从设备得到的信息处理相关的数据。对于 DXP 矩阵，控制处理器往往需要知道矩阵的切换关系是否发生变化（特别是从前面板切换或内置的网页进行切换时），或矩阵是否根据预设程序进行了信号的切换。

当 DXP 矩阵是从前面板或者内置的网页时行切换时，矩阵会发送 ‘Qik\r\n’（详情请查阅 DXP 的用户手册）。

Switcher-initiated Messages

When a local event such as a front panel operation occurs, the switcher responds by sending a message to the host. The switcher-initiated messages are listed below (underlined). In these messages, *Vn.nn* is the firmware version number and *60-nnnn-01* is the DXP part number.

With an RS-232 or RS-422 connection:

(c) Copyright 2011, Extron Electronics DXP DVI-HDMI, *Vn.nn*, 60-*nnnn*-01

The switcher initiates the copyright message if it is powered on while connected to the computer.

Qik

The switcher initiates the Qik message when a front panel switching operation has occurred.

如果需要判断矩阵是否发生了切换，不管是来自前面板、内置网页、还是（Quick Tie command）切换的指令。之前所建立 **CurrentResponse** 的变量就可以发挥作用了。该变量可以用来评估是否反馈含有字符串 ‘Qik’ 信息。要注意的是我们只是判断矩阵是否发生了切换的动作，而不是具体切换的结果。如果了解哪路输入和哪路输出连接，中控需要向矩阵发送指令进行查询。（下面的例子中是查询矩阵输入 1，2，3，4 的切换状态。）

```
if CurrentResponse == 'Qik': # Front Panel or Quick Tie Response
    print('front panel switch made')
    DXPMatrix.Send('1%2%3%4%')
```

当 DXP 矩阵设置为 Verbose 模式时，收到的反馈和发送指令获得的是一样的，可以使用相同的方法进行解析。回到刚才的例子中，当 **CurrentResponse** 不等于 ‘Qik’ 时，将执行后面的 elif 语句，判断前三个字符是否为 “Out”，若符合条件，我们可使用正则

表达式的方式从返回的信息中检索出具有意义的信息，并且分类存储到数据组，为后面的程序做准备。

```
elif CurrentResponse[0:3] == 'Out': # Tie command response
    print('typical tie response')
    ResponsePattern = re.compile(
        'Out([0-9]{1,3}) In([0-9]{1,3}) (Aud|Vid|RGB|All)'
    )
    matchObject = ResponsePattern.search(CurrentResponse)
    if matchObject:
        input = int(matchObject.group(2))
        output = int(matchObject.group(1))
        tietype = matchObject.group(3)
        MatrixOutput[output-1]=input
        print('in={0} out={1} type={2}'.format(input, output, tietype))
```

正则表达式提供了搜寻、识别、替换所接收信息的方法，经过解析后使其更有意义。这个模块可以导入到任何 Python 程序中，由于 Extron ControlScript 也是基于 Python 设计的，如果要使用该模块，只需用 import 语句导入即可获得其强大的功能。

```
import re
```

上面的例子中，如果 GUI 用户界面中的 Laptop 笔记本按钮被选中，会触发 **SelectInputButtonPressed** 函数，从 DXP 矩阵返回的信息是经过请求后才返回的，

Print Command	Laptop button is Pressed at 3:01:55 PM
Print('Response:{0}'.format(Response))	3:01:58 PM Response: b'Out1In1 All\r\n'

查看设备的通讯协议，矩阵的输入和输出的反应用 'Out' 和 'In' 表示，'Out' 后面对应输出端口号，'In' 后面对应输入端口号。最后跟着相应的切换类型，为 Video、Audio、和 All。因此，我们可以用关键词 'Out'、'In' 和切换类型来解析矩阵的反馈信息。另外字符的结尾也是用了回车符和换行符作为结尾，用来分隔反馈信息。

由于我们知道要返回的数据格式，所以可以建立一个正则表达式，用来搜索需要匹配的数据，并将匹配的结果存储到关键字数据组中。要做到这点最简单的方法是将设备返回的字符串分组。另外，这条正则表达式可能会被反复使用，所以建议将其存储在变量中。

```
ResponsePattern = re.compile('Out1 In1 All\r\n')
```

需要设置一个模式规则来匹配所有的输入和输出数字的组合，因此里面的数字需要替换成正则表达式，用于查找匹配我们需要的信息。下面创建一个模式规则，用一个包含所有数字（0-9）的列表来替代‘Out1’里面的数字“1”，模式规则（pattern）的语法是用[0123456789]表示，或者简单用[0-9]表示。这样模式规则可以匹配反馈信息中的字符‘1’。但是如果矩阵路数比较大，输出路数可能是上百的通道，我们需要匹配3位数字。这时候我们可以重复使用[0-9]，或在[0-9]的后面添加限定符{1,3}，表示匹配项有可能是3位数字。这样我们就建立了第一组匹配的模式规则，并用圆括号包围起来。

```
ResponsePattern = re.compile('Out([0-9]{1,3}) In1 All\r\n')
```

对输入也是使用同样的处理方式，捕捉输入的编号，并存储到第二组匹配项中。

```
ResponsePattern = re.compile('Out([0-9]{1,3}) In([0-9]{1,3}) All\r\n')
```

请注意，正则表示式的字符串内空格是很重要的。如果在关键字‘In’之前没有包含空格，则创建的表达式将无法匹配来自矩阵的反馈信息。

最后一步是判断切换类型，从DXP说明书中可以看到，切换的类型有‘All’，代表视音频一起切换，还有其它缩写‘RGB’、‘Vid’和‘Aud’代表视频或音频的切换。这些关键字都需要添加到ResponsePattern中。

Command and Response Table for DXP SIS Commands

Command	ASCII Command (Host to Switcher)	Response (Switcher to Host)	Additional Description
Create Ties			
NOTES: <ul style="list-style-type: none"> Commands can be entered back-to-back in a string, with no spaces. Example: 1*1!02*02&003*003%4*8\$ The quick multiple tie and tie input to all output commands activate all I/O switches simultaneously. The DXP switchers support 1-, 2-, and 3-digit numeric entries (1*1!, 02*02&, or 003*003%). The & tie command for RGB and the % tie command for video can be used interchangeably. To untie an output from all inputs, enter a tie command in which x2 = 0. 			
Tie input x2 to output x3 , video and audio <i>Example:</i>	x2*x3! 1*3!	Out x3 •In x2 •All← Out3•In1•All←	Tie Input x2 video and audio to Output x3 . Tie Input 1 video and audio to Output 3.
Tie input x2 to output x3 , RGB only <i>Example:</i> (See the second bullet point in the notes above.)	x2*x3& 8*4&	Out x3 •In x2 •RGB← Out4•In8•RGB←	Video breakaway Tie Input 8 RGB to Output 4.
Tie input x2 to output x3 , video only <i>Example:</i> (See the second bullet point in the notes above.)	x2*x3% 7*5%	Out x3 •In x2 •Vid← Out5•In7•Vid←	Video breakaway Tie Input 7 video to Output 5.
Tie input x2 to output x3 , audio only <i>Example:</i>	x2*x3\$ 6*4\$	Out x3 •In x2 •Aud← Out4•In6•Aud←	Audio breakaway Tie Input 6 audio to Output 4.

为了匹配所有可能的切换类型。我们需要在第三个关键字组中添加所有的可能性，用 '|' 符号进行分隔。

```
ResponsePattern = re.compile('Out([0-9]{1,3}) In([0-9]{1,3}) (Aud|Vid|RGB|All)')
```

另外在建立规则时要注意，除非特别说明，匹配内容一定要区分字母的大小写。

现在正则表达式规则已经建立，设备的反馈信息已存储在变量 **CurrentResponse** 中，控制处理器可以使用已创建的规则对该信息进行分析，然后从正则表示式建立的三个关键字组中，取得所需要的数据。要实现这个功能，需要创建一个新的变量 **MatchObject** 存储搜索到的匹配数据，每个关键字匹配的值会存储在对应的变量里，最后通过 `print` 指令显示在 GS 软件的跟踪栏中。

```
MatchObject = ResponsePattern.search(CurrentResponse)
if MatchObject:
    input = int(MatchObject.group(2))
    output = int(MatchObject.group(1))
    type = MatchObject.group(3)
    print('in={0} out={1} type={2}'.format(input, output, tietype))
```

我们可以使用解析的方法处理任何从矩阵返回的数据。接下来的例子中会创建两种不同的解析规则。一个是应对一路输入切换到所有输出端口的情况，另外一个是对传统的矩阵输入输出切换。这两个正则表达式指令先在函数外进行编译（`re.compile`），之后函数调用时就不需要再次编译了。相关解析矩阵反馈信息的代码如下所示：

```
# Response pattern for one input to all outputs
ResponsePatternIn = re.compile('In([0-9]{1,3}) (Aud|Vid|RGB|All)')
# Response pattern for typical matrix tie response
ResponsePatternTie = re.compile('Out([0-9]{1,3}) In([0-9]{1,3}) (Aud|Vid|RGB|All)')

@event(DXPMatrix, 'ReceiveData')
def MatrixParse(interface, response):
    global MatrixBuffer
    global MatrixOutput
    global ResponsePatternIn
    global ResponsePatternTie
    MatrixBuffer = MatrixBuffer + response.decode()
    if MatrixBuffer.rfind('\r\n') > 0:
        MatrixResponse = MatrixBuffer
        DelimiterIndex = MatrixResponse.rfind('\r\n')
        CurrentResponse=MatrixResponse[:DelimiterIndex]
```

```

MatrixBuffer=MatrixResponse[DelimiterIndex+2:]
if CurrentResponse == 'Qik': # Front Panel or Quick Tie Response
    print('front panel switch made')
    dvMatrix.Send('1%2%3%4%')
elif CurrentResponse[0:2] == 'In': # Selected Input to All
Outputs
    print('selected input to all outputs')
    MatchObject = ResponsePatternIn.search(CurrentResponse)
    if matchObject:
        input = int(MatchObject.group(1))
        tietype = MatchObject.group(2)
        print('in={0} type={1}'.format(input, tietype))
        for item in MatrixOutput:
            MatrixOutput[MatrixOutput.index(item)]=input
        print(MatrixOutput)
    elif CurrentResponse[0:3] == 'Out': # Tie command response
        print('typical tie command response')
        MatchObject = ResponsePattern.search(CurrentResponse)
        if MatchObject:
            input = int(MatchObject.group(2))
            output = int(MatchObject.group(1))
            tietype = MatchObject.group(3)
            MatrixOutput[output-1]=input
            print('in={0} out={1} type={2}'.format(input, output,
tietype))
            print(MatrixOutput)
        else:
            print('response not captured')
            print('MatrixBuffer is {0} characters
long'.format(len(MatrixBuffer)))

```

8.3.3 同步通讯

在同步通讯时，在指令传送后应有预期的反馈出现。上面异步通讯例子中的 ReceiveData 的 @event 事件处理方式可以应用在同步通讯中，我们还为同步通讯开发了 SerialInterface 类中的 SendAndWait() 的方法。使用这个方法，指令会首先用 Send 的方法进行发送，之后它会在一个指定的时间内等待反馈。同时，可以指定分割结束符（delimiter），一旦接收的字符匹配分割结束符，停止等待倒计时，然后回传从设备接收的数据。然后关于 SendAndWait 的指令有一点需要注意，在 Timeout 倒计时结束或发现匹配分隔符前，不可以执行其他任何操作。我们来做个演示，先将单向串口通讯中例子中的 Send 指令改为 SendAndWait 指令，Timeout（超时等待时间）参数设为 3 秒，不

设定分割结束符。当信号源选择按钮按下后，会发送矩阵切换指令，但弹出界面这时候没有马上显示，而是直到 3 秒超时等待时间过去后才会显示。

```
def SelectInputButtonPressed(button, state):
    print('button pressed: {0}'.format(button.ID))
    if button is SelectLaptop:
        CurrentInput = 'Laptop'
        Response = DXPMatrix.SendAndWait('1*1!', 3)
    elif button is SelectPC:
        CurrentInput = 'PC'
        Response = DXPMatrix.SendAndWait('2*1!', 3)
    elif button is SelectBluray:
        CurrentInput = 'Bluray'
        Response = DXPMatrix.SendAndWait('3*1!', 3)
    elif button is SelectDocCam:
        CurrentInput = 'DocCam'
        Response = DXPMatrix.SendAndWait('4*1!', 3)
    elif button is SelectTuner:
        CurrentInput = 'Tuner'
        Response = DXPMatrix.SendAndWait('7*1!', 3)
    elif button is SelectAux:
        CurrentInput = 'Aux'
        Response = DXPMatrix.SendAndWait('8*1!', 3)
    else:
        CurrentInput = 'NotSelected'
    print('CurrentInput: {0}'.format(CurrentInput))
    print('Response: {0}'.format(Response))
    ShowSourcePopup(CurrentInput, TLP)
```

随意按下一个按键，SelectInputButtonPressed 的函数功能将被执行，之后根据程序内容，将会有三条不同信息通过 print 指令显示在 GS 软件的跟踪窗口中，请留意按下按键至收到反馈所需的时间，三种不同的方法有三秒的时间差。

Print Command	Laptop button is Pressed at 3:01:55 PM
Print('button pressed:{0}, ID-{1}'.format(button.Name,button.ID))	3:01:55PM button pressed: Laptop, ID-8050
Print('CurrentInput:{0}'.format(CurrentInput))	3:01:55PM CurrentInput: Laptop
Print('Response:{0}'.format(Response))	3:01:55PM Response; b'Out1In1 All\r\n'

在确保中控能够接收完整反馈信息的前提下，Timeout（等待时间）的参数应尽可能设置的短一点（0.1 秒是程序的最小值）。另外，如果加入 delimiter（分割结束符）参数，当满足预期的条件后，Timeout 倒计时会立刻停止。例如，根据 DXP HDMI 矩阵里

的 SIS 所描述的，返回的数据应该是最少 14 个字符串长度，这个可以在 **SelectInputButtonPressed** 函数的追踪信息中得到验证，GS 软件的 Trace Message 中会显示：“Response:...”。如果 SendAndWait 中第三个参数 deliLen 的长度设为 14，如下面例子所示，不会感受到明显的延时。

```
Response = DXPMatrix.SendAndWait('1*1!', 3, deliLen=14)
```

Extron 的矩阵中，指令的返回代码都是以回车符和换行符为结束。在 Trace message 跟踪窗口中你可以看到，返回信息都是以 \r\n 的字符作为结尾。在 SendAndWait 代码中，我们可以使用 '\r\n' 作为参数取代 deliLen。这样，无论反馈的代码长度有多少，控制器一接收到 \r\n 字符，就代表反馈信息发送完毕。

```
Response = DXPMatrix.SendAndWait('1*1!', 3, deliTag=b'\r\n')
```

第三个参数 delimiter（分割结束符），也可使用正则表达式作为参数，用来寻找匹配的字符串。Extron 的矩阵 SIS 代码中，矩阵的切换指令反馈的结尾是切换的类型：All, RGB, Video 和 Audio。我们可以搜索正则表达式中定义的内容作为 delimiter（分割结束符）。一旦在反馈代码中找到以上匹配的字符串，则取消 Timeout（超时等待时间）的计时，反馈信息接收就此完成。

```
Response = DXPMatrix.SendAndWait('1*1!', 3, deliRex='(All|RGB|Vid|Aud)')
```

由于我们已经收到了设备的反馈信息，所以可以利用它来改变按钮的视觉反馈，或发送给其他对象，亦或者用作它途。这些处理对于同步或者异步的通讯都是适用的。同步或异步通讯都适合这种处理方式。

8.4 以太网端口的通讯

受以太网控制的设备通讯和串口设备类似，不同之处在于以太网的连接需要进行管理。如果将 DXP 矩阵的通讯从串口改为以太网的话，可以参考使用串口发送数据和接收数据的程序代码。

8.4.1 建立以太网连接

要定义通过以太网受控的设备，我们需要知道以下几个关键参数：

- 该设备是服务器端 (server) 还是客户端 (client) ?
- 它是使用哪种类型的通讯协议 – TCP、UDP 还是 HTTP ?
- 它的 IP 端口号是多少 ?
- 设备的 hostname (主机名称) 或 IP 地址是多少 ?

这些信息通常可以在用户说明书或 API 文档上找到。

8.4.2 客户端通讯 – TCP

许多设备都可通过以太网控制，例如 Extron 设备也会使用传输控制协议-TCP 受控。每台 Extron 设备的用户指南中都可找到相关的信息，（ DXP 矩阵用户指南有关 TCP 连接的内容如下图所示），其他通过以太网控制的设备的制造商一般都会提供类似的信息。

Establishing an Ethernet Connection

Establish a network connection to a DXP switcher as follows:

1. Open a TCP connection to port 23, using the IP address of the switcher. A variety of methods are available for making this connection, including Telnet or utilities such as Extron DataViewer.

The switcher responds with a copyright message that includes the date, the name of the product, firmware version, part number, and the current date and time.

NOTES:

- If the switcher is not password-protected, the device is ready to accept SIS commands immediately after it sends the copyright message.
- If the switcher is password-protected, a Password prompt appears below the copyright message.

2. If the switcher is password-protected, enter the appropriate administrator or user password.
3. If the password is accepted, the switcher responds with Login User or Login Administrator.
4. If the password is not accepted, the Password prompt reappears.

这个文档中指明了 DXP 矩阵是作为 TCP 服务器工作的（因为它可以被 DataViewer 或 Telnet 控制）。所以，**DXPMatrix** 对象使用的通讯方式是以太网，以及 EthernetClientInterface 类。在这个范例中，DXP 使用的是矩阵默认 IP 地址 192.168.254.254，和默认 TCP 端口号 23。由于该设备使用的是默认的 TCP 协议，只需要指定 IP 地址或主机名，以及 IP 的端口号即可。

```
DXPMatrix = EthernetClientInterface('192.168.254.254', 23)
```

一旦定义了矩阵的通讯接口，需要使用 `EthernetClientInterface` 类里的 `Connect` 方法（Method）建立和设备的连接。程序会根据连接的状态返回字符串：‘Connected’，‘TimedOut’，或 ‘HostError’。程序员可以根据返回的字符串内容，尝试重新建立连接，或简单地提供状态信息。

```
ConnectResponse = DXPMatrix.Connect()
```

如果没有在 `Connect` 方法中提供超时参数，或控制器无法和指定的 IP 地址建立连接，连接管理器会无限制等待。对于 DXP 矩阵这类设备，如果 5 分钟内没有检测到任何动作，它会关闭连接。使用 `StartKeepAlive()` 方法，可以避免此类情况发生。这种方法会以一定的时间间隔向受控设备发送指令，程序员需要专门选择发送出去的指令和从受控设备收到的反馈代码。如果 `StartKeepAlive()` 影响系统正常运行，通常使用 `StopKeepAlive()` 方法终止 `StartKeepAlive()`。例如，有些现实设备在开启或关闭过程中不接受任何指令。

对于 DXP 矩阵，根据用户指南的建议，需要向其发送状态查询指令 ‘Q’。我们可以使用 `StartKeepAlive()` 方法在指定时间间隔内重复向矩阵发送这条指令来查询矩阵状态。

Connection Timeouts

The Ethernet link times out after a designated period of no communications. By default, this timeout value is set to 5 minutes, but the value can be changed (see the [Configure current port timeout](#) command in the Command and Response Table for IP-specific SIS Commands, page 70).

NOTE: Extron recommends leaving the timeout at 5 minutes (default) and periodically issuing the Query (Q) command to keep the connection active. If there are long idle periods, Extron recommends disconnecting and reopening the connection when another command must be sent.

下面的范例中，中控和受控设备建立连接或是断开连接时都会触发一个事件（event），建立连接时调用 **MatrixDisconnectHandler** 函数并执行 `StopKeepAlive()` 方法；建立连接时调用 **MatrixConnectHandler** 函数，并发送一条指令，并使用 `StartKeepAlive()` 方法，每隔 60 秒发送指令 ‘Q’ 查询矩阵状态

```
@event(DXPMatrix, 'Disconnected')
def MatrixDisconnectHandler(interface, ConnectionState):
    print('Matrix Disconnected')
    DXPMatrix.StopKeepAlive()

@event(DXPMatrix, 'Connected')
def MatrixConnectHandler(interface, ConnectionState):
```

```
print('Matrix Connected')
DXPMatrix.Send('W3CV\r')
DXPMatrix.StartKeepAlive(60, 'Q')
```

另一种判断以太网连接状态的方法是使用一个函数处理 Connected 和 Disconnected 堆叠事件。这个函数和前面针对连接和断开分别定义函数实现的功能时相同的。

```
@event(DXPMatrix, 'Disconnected')
@event(DXPMatrix, 'Connected')
def MatrixConnectionHandler(interface, ConnectionState):
    if 'Connected'in ConnectionState:
        print('Matrix Connection: {0}'.format(ConnectionState))
        DXPMatrix.Send('W3CV\r')
        DXPMatrix.StartKeepAlive(60, 'Q')
    elif 'Disconnected'in ConnectionState:
        print('Matrix Connection: {0}'.format(ConnectionState))
        DXPMatrix.StopKeepAlive()
```

关于 Connected 和 Disconnected 事件有一点需要注意，这两个事件的触发表示控制处理器对程序中 Connect() 或 Disconnect() 方法有响应。如果是其他原因导致网络连接断开，则不会触发 Connected 和 Disconnected 这两个事件，例如网线被断开，超出网络范围，供电问题等。

8.4.3 客户端通讯 – UDP

一些设备使用用户数据报协议-UDP。这种协议不含 TCP 中所使用的握手或纠错功能。UDP 通讯模式下，发送者并不清楚对方是否收到信息，通常发送完毕后立刻断开连接，这样处理信息可以更快捷，只是传输的信息未必完整。

为使用 UDP 通讯协议的设备创建 EthernetClientInterface 类的对象和使用 TPC 协议的设备类似，需要定义参数 ServicePort。

```
BACnet = EthernetClientInterface('192.168.254.254', 0xBAC0, 'UDP')
```

所有其他的方法和之前描述的 TCP 客户端完全一致。

9 音量控制接口的程序

在 Pro 控制处理器定义音量控制接口的方法和其他接口类似。IPCP Pro250 包含音量控制接口，可使用 VolumeInterface 类编写其音量控制接口的程序，实现平滑地调节音量的功能。

程序中音量增加的程序是采用分步设定电平的方式实现的。在下面的范例中，前两行是实例化 IPCP Pro250 控制处理器和音量控制接口。接下来，在字典 Pro250Vol 中定义音量控制的相关参数。请注意，定义音量控制接口时不是每次都需要这些代码。最后，字典 Pro250Vol 的键 “Level” 被设定为当前的音量接口的电平值。

```
Pro250Controller = ProcessorDevice('Pro250')
Pro250VolumePort = VolumeInterface(Pro250Controller, 'VOL1')
Pro250Vol = {
    'Level':0,
    'Step':2,
}
Pro250Vol['Level'] = Pro250VolumePort.Level
```

音量的上限和下限可使用 SetRange() 方法 (method) 设定，在下面的范例中，TLP Pro 720T 触摸屏界面模板中的 Main Volume 按键，将对音量控制接口进行操作。触摸屏和用户界面中 ID 号为 8023 的音量条被实例化。音量条显示当前音量控制接口的电平值。接下来的代码对音量上升，下降按键进行了实例化，并设定了按键被按下和处于按住状态时发生的事件。请注意，每个音量控制按键的函数都对应 ‘Pressed’ 和 ‘Repeated’ 两种事件，当音量上升或下降按键中任意一个被按住或保持超过 2 秒，会发送音量控制指令。如果按键按下时间超过 2.2 秒，之后每 0.2 秒发送一次指令。如果希望指令在按键保持按下 2 秒时发送，可在程序中使用 “Held event”。

```
TLP720 = UIDevice('720T')
TLP720MainVolLevel = Level(TLP720, 8023)
TLP720MainVolLevel.SetLevel(Pro250Vol['Level'])
MainVolUpButton = Button(TLP720, 8120, holdTime=2, repeatTime=0.2)
@event(MainVolUpButton, 'Pressed')
@event(MainVolUpButton, 'Repeated')
def MainVolUpPressed(button, state):
    MainVolUpButton.SetState(1)
    if Pro250Vol['Level'] < 100:
        Pro250Vol['Level'] += Pro250Vol['Step']
        Pro250VolumePort.SetLevel(Pro250Vol['Level'])
        TLP720MainVolLevel.SetLevel(Pro250Vol['Level'])
```

```
MainVolDownButton = Button(TLP720, 8121, holdTime=2, repeatTime=0.2)
@event(MainVolDownButton, 'Pressed')
@event(MainVolDownButton, 'Repeated')
def MainVolDnPressed(button, state):
    MainVolDownButton.SetState(1)
    if Pro250Vol['Level'] > 0:
        Pro250Vol['Level'] -= Pro250Vol['Step']
        Pro250VolumePort.SetLevel(Pro250Vol['Level'])
        TLP720MainVolLevel.SetLevel(Pro250Vol['Level'])
```

此外，请注意 ‘Released’ 和 ‘Tapped’ 事件中（功能等同于 Global Configurator Professional 里面的 Quick Release）将按键的状态设为 0。

```
@event(MainVolUpButton, 'Released')
@event(MainVolUpButton, 'Tapped')
def MainVolUpReleased(button, state):
    MainVolUpButton.SetState(0)
@event(MainVolDownButton, 'Released')
@event(MainVolDownButton, 'Tapped')
def MainVolDnReleased(button, state):
    MainVolDownButton.SetState(0)
```

Mute 功能的实现方法和前面类似，只是在代码中写成了开关（Toggle）的形式。当按键按下时，如果 Mute 状态是 “On”，则将其设为 “Off”，按键的视觉反馈设置成 Control State 0。由于之前的按键视觉反馈是 Control State1，所以用户可以立刻看到静音状态的变化。

```
MainVolMuteButton = Button(TLP720, 8021)
@event(MainVolMuteButton, 'Pressed')
def MainVolMutePressed(button, state):
    if Pro250VolumePort.Mute == 'On':
        Pro250VolumePort.SetMute('Off')
        MainVolMuteButton.SetState(0)
    else:
        Pro250VolumePort.SetMute('On')
        MainVolMuteButton.SetBlinking('Medium', [1, 0])
```


10 输入和输出接口的程序

Extron 的 Pro 系列控制处理器有很多种类型的输入和输出接口。如要针对每种类型的接口编写程序，控制处理器和接口都需要在程序中实例化。

10.1 继电器接口

控制处理器的每个继电器接口都通过端口名称进行识别（如 RLY1, RLY2, 等）。

```
Relay1 = RelayInterface(Proc, 'RLY01')
Relay2 = RelayInterface(Proc, 'RLY02')
```

控制继电器状态的方法（method）有：Pulse，SetState 和 Toggle。

```
Relay1.Pulse(2.25) # Pulse the relay on for 2.25 seconds
Relay2.SetState(1) # Close the relay
Relay2.SetState(0) # Open the relay
Relay1.Toggle() # Set the relay to the opposite state - no parameter
required
```

继电器的状态可存储在变量中，可根据其状态属性（State）用于后面的代码中。

```
Relay2Status = Relay2.State
```

10.2 开关电源接口

IPCP Pro 550 控制处理器包含 12V 开关电源接口，可采用与继电器类似的方法控制。这些电源接口将使用 Interface 数据包中的 SWPowerInterface 类进行实例化。

```
Power1 = SWPowerInterface(Proc, 'SPI1')
Power2 = SWPowerInterface(Proc, 'SPI2')
Power1.SetState('Off') # Set the switched power outlet #1 off
Power2.SetState('On') # Set the switched power outlet #2 on
```

10.3 触点闭合接口

比如 IPL Pro CR88 就包含触点闭合接口，可检测连接在输入端（Input）和接地端（Ground）的触点或开关是否闭合。触点闭合接口使用 Interface 数据包中的 ContactInterface 类进行实体化。需要的参数是处理器的标识符合触点闭合接口的名称。

这个控制接口没有方法（Method）可以使用。但包含接口当前的状态属性。如果接口检测到闭合信号，可使用“StateChanged”事件（event）触发相应的函数。在下面的函数中，当接口状态改为“On”时，指定的触摸屏上会显示一条信息。当然，它可以在系统中触发任何动作。

```

Contact1 = ContactInterface(Proc, 'CII1')

@event(Contact1, 'StateChanged')
def Contact1Change(interface, state):
    if state == 'On':
        TLP.ShowPopup('PartitionMessageClose')
    else:
        TLP.ShowPopup('PartitionMessageOpen')
    
```

10.4 数字输入输出接口

在 Extron Pro 系列控制系统中，数字输入和输出接口的实际功能是由实例化的方式决定的。在数字输入模式下，其工作原理和触点闭合接口类似。DigitalIOInterface 的类中有一个名为 Mode 的关键字参量，它将决定端口的工作方式，默认值为“DigitalInput”。在下面的范例中，接口 1 被实例化为数字输入接口，取名 DigitalIO1，接口 2 为数字输出接口，取名 DigitalIO2。

```

DigitalIO1 = DigitalIOInterface(Proc, 'DIO1', Mode='DigitalInput')
DigitalIO2 = DigitalIOInterface(Proc, 'DIO2', Mode='DigitalOutput')
    
```

还有一个关键字参数“Pullup”没有在范例代码中显示。它的默认值是 False。所有 Python 的类和函数关键字都有默认值，并且是可选项。调用类和函数时无须声明。因此，如果 Mode 和 Pullup 的默认值适合实际应用，DigitalIO1 可做如下定义：

```

DigitalIO1 = DigitalIOInterface(Proc, 'DIO1')
    
```

如果接口的电路中使用上了拉（Pull Up）电阻，应将 Pullup 的值改为 True。

```

DigitalIO1 = DigitalIOInterface(Proc, 'DIO1', Pullup=True)
    
```

然后，好的程序需要将可读性作为首要目标，提供每个参数的值会让代码的功能更清晰。

```
DigitalIO1 = DigitalIOInterface(Proc, 'DIO1', Mode='DigitalInput', Pullup=True)
```

10.5 多功能输入/输出接口

多功能输入输出接口将输入模拟电压的功能加入了 Extron 控制系统。多功能输入输出接口有三种工作模式：数字输入，数字输出和模拟输入。当设置为数字输入和数字输出接口时，其功能和之前 DigitalIOInterface 类中描述的相同。当设置为模拟输入（“AnalogInput”）时，它从输入接口可以读取的电压范围是 0V 至 25.3V。在下面的范例中，多功能输入/输出接口分别被定义为模拟输入，数字输入，数字输出。

```
FlexIO1 = FlexIOInterface(Proc, 'FIO1', Mode='AnalogInput')
FlexIO2 = FlexIOInterface(Proc, 'FIO2', Mode='DigitalInput',
Pullup=False)
FlexIO3 = FlexIOInterface(Proc, 'FIO3', Mode='DigitalOutput')
```

当多功能输入/输出接口被定义为数字输入模式时，需要提供 2 个附加的参数定义电压的上限和下限。可以在该接口实例化时进行设置，或使用下面的代码进行初始化：

```
FlexIO2.Initialize(Upper=3.0, Lower=1.9)
```

当是指为数字输入模式时，当状态改变超出设定的阈值时，触发 “StateChanged” 事件。在模拟输入模式下，输入电压数值由多功能输入/输出接口的 “AnalogVoltage” 属性决定。

11 延时和计划任务的程序

ControlScript 提供了 Wait 类，可延时指定的时间长度后执行函数。此外，还提供了 Clock 类，在指定的时间和日期执行函数。虽然两个类的作用都是延时执行函数，Wait 类的延时长度是相对的，而 Clock 类延时执行程序的时间是固定的。关于这两个类需要说明的是，他们都不会终止连续运行的程序。

11.1 使用 Wait 延时执行函数

Wait 可定义成一次性时间，也可在命名后再程序中重复使用。

11.1.1 使用事件修饰函数定义一次性 Wait

Wait 事件可使用修饰函数定义一次性的延时操作。我们先查看下面的代码：

```
# Function containing tasks to perform when power is restored to the
system
# or when the program is reloaded.
def PowerupTasks():
    print('Begin Powerup Tasks')
    TLP1.ShowPage('Start')
    RoomNameLabel.SetText(RoomName)
    UserMessageLabel.SetText('System is Ready for Use')
PowerupTasks()
```

如果经过一些测试后，确定系统初始化需要几秒钟的延时。可以在函数代码前添加 @wait，正如下面范例代码最后一行注释所说，@wait 后的修饰函数会自动执行，不需要专门调用。

```
# Function containing tasks to perform when power is restored to the system
# or when the program is reloaded.
@Wait(5) # Wait 5 seconds before executing the Powerup Tasks
def PowerupTasks():
    print('Begin Powerup Tasks')
    TLP1.ShowPage('Start')
    RoomNameLabel.SetText(RoomName)
    UserMessageLabel.SetText('System is Ready for Use')
# The function is called automatically from the @Wait decorator
```

11.1.2 命名 Wait

如果需要多次使用 Wait，最好为 Wait 命名，这样 Wait 就可以在程序或用户界面中重复使用，或根据需要做些变更。Wait 命名后可以延长，修改或取消延时的长度，或根

据需要重新启动。例如，当一个弹出式界面显示后，如果页面自动关闭之前有按键动作，可以延长自动关闭的时间。比如，有一个用于调整 AV 系统音量的弹出式页面，首次显示时，Wait 会定义延时的长度，如果此时音量控制按键被按下或释放，程序可以改变延时的长度，以保证在调整音量的过程中或按键释放后的一段时间内，弹出式页面不会隐藏。

```
AudioControlButton = Button(TLP1, 8110, holdTime = None, repeatTime = None)
PopupPageDelay = Wait(15)
def ClosePopupPage():
    TLP1.HidePopup(37)

@event(AudioControlButton, 'Pressed')
def AudioControlButtonPressed(button, state):
    global PopupPageDelay
    TLP1.ShowPopup(37, 0)
    PopupPageDelay = Wait(15, ClosePopupPage)
```

当按下按键时，Wait 时间会重启，放置页面关闭过快（为简化代码，范例中没有添加实际控制音量的指令。）

```
MainVolUpButton = Button(TLP720, 8120, holdTime = 2, repeatTime = 0.2)
@event(MainVolUpButton, 'Pressed')
def MainVolUpPressed(button, state):
    global PopupPageDelay
    PopupPageDelay.Restart() # Restart the delay with the initial delay
    time
    MainVolUpButton.SetState(1)
```

当释放按键时，减少延时的时间。

```
@event(MainVolUpButton, 'Released')
def MainVolUpReleased(button, state):
    PopupPageDelay.Change(5) # Change delay to close 5 secs after button is released
    MainVolUpButton.SetState(0)
```

当弹出式页面的 Close 按键被按下，当前的 Wait 将被取消。

```
AudioControlCloseButton = Button(TLP, 9027)
@event(AudioControlCloseButton, 'Pressed')
def AudioControlClosePressed(button, state):
    PopupPageDelay.Cancel() # Cancel Wait
    TLP.HidePopup(41)
```

使用 Wait 类中的 Add 方法 (method) 也可增加 Wait 的时间。它将在原有基础上继续延长指定的时间。例如，当按下 Mute 按键后，将在目前 Wait 的延时的基础上增加指定的时间。

```
MainVolMuteButton = Button(TLP, 8021)
@event(MainVolMuteButton, 'Pressed')
def MainVolMute(button, state):
    PopupPageDelay.Add(5) # Add 5 seconds to the Wait time
```

11.2 基于时钟和日历时间的计划任务事件

使用 Clock 类，可在指定的时间和日期执行指定的函数。计划任务通常用于自动执行系统启动，关闭这类功能，或用于将变量设为初始值。若要使用 Clock 类，需要从 extronlib 库中的 system 数据包导入。

```
from extronlib.system import Clock
```

实例化一个计划任务需包含 clock 类的标识符，以及几个相关的参数：时间 (Times)，日 (Days)，以及需要执行的函数名称。

时间在 list 中使用 'HH:SS:MM' 的字串格式并使用 24 小时制。

```
['19:30:00', '21:00:00']
```

在字符串列表中定义一周的每一天：

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday',]
```

如果列表中包含了多个时间，或一星期中的每一天，它就会变得冗长。最好的方法是将列表实例化和计划任务的事件配合使用。请参考下面的代码：

```
def ScheduledShutdown(interface, event):
    SystemPower=False
    TLP.ShowPopup(9) # Show Shutdown Page
@Wait(30)
def HidePopupShutdown():
    TLP.HidePopup(9)
    # Other functions as needed for shutdown
    ShutdownDays=['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday',]
    ShutdownClock=Clock(['19:30:00','21:00:00'], ShutdownDays, ScheduledShutdown)
```

不同于Wait的是，Clock的计划任务在定义后，必须启动后才能在预定的时间执行。在下面的范例中，用户可以通过用户界面上的按钮启动或取消计划任务。

```
ScheduledShutdownEnableButton = Button(TLP, 8010)
@event(ScheduledShutdownEnableButton, 'Pressed')
def ScheduledShutdownEnablePressed(button, state):
    global ScheduledShutdownEnabled
    if ScheduledShutdownEnabled == True:
        ScheduledShutdownEnabled = False
        ShutdownClock.Disable()
        ScheduledShutdownEnableButton.SetState(0)
    else:
        ScheduledShutdownEnabled = True
        ShutdownClock.Enable()
        ScheduledShutdownEnableButton.SetState(1)
    print('Schedule Enabled: {0}'.format(ScheduledShutdownEnabled))
```

附录 A-Python 学习资源

对于缺乏经验的程序员或仅需要了解进阶编程主题的人来说，很多资源都有助于帮助你理解 Python。这个列表只是学习资源的冰山一角，也许并不能使你获益良多，但至少可以帮助你迈出学习 Python 的第一步。

纸质和电子参考书

- McGrath, Mike. Python in Easy Steps. Leamington Spa, Warwickshire, U.K.: In Easy Steps, 2013
- Lutz, Mark. Learning Python. 5th ed. Sebastopol: O'Reilly, 2013.
- Pilgrim, Mark. Dive into Python 3. New York: Apress, 2009.
- (Also available as an e-book <http://www.diveinto.org/python3/> at no cost.)
- Harrison, Matt. Guide To: Learning Python Decorators.: CreateSpace Independent Platform, 2013.

网站和教程

- www.python.org
- <https://docs.python.org/3.3/tutorial/>
- <http://www.python.org/dev/peps/pep-0008/> - Python Style Guide
- <http://www.coursera.org/course/interactivepython>
- <http://www.learnpython.org>
- <http://www.pythontutor.org>
- <http://www.codecademy.com/tracks/python> - This tutorial references

Python 2.7, but does give a good

- overview of Python

附录 B-禁用的 Python 内置函数和模块

下列各項 Python 函式庫和內建函式是被排除在 ControlScript 中使用：

禁用的内置函数

compile	Exit	locals
dir	globals	Open
Eval	Help	Quit
Exec	input	SystemExit
vars		

禁用的内部模块

bz2	imp	readline
crypt	marshall	resource
ctypes	mmap	select
email	mslib	signal
faulthandler	msvcrt	spwd
fcntl	nis	sys
fpectl	ossaudiodev	syslog
gc	parser	termios
grp	PixmapWrapper	winreg
hotshot	posix	winsound
idlelib	pwd	zipimport

禁用的外部模块

<code>__future__.py</code>	<code>imghdr.py</code>	<code>shutil.py</code>
<code>_pyio.py</code>	<code>importlib</code>	<code>site.py</code>
<code>_strptime.py</code>	<code>inspect.py</code>	<code>smtpd.py</code>
<code>_threading_local.py</code>	<code>io.py</code>	<code>smtplib.py</code>
<code>aifc.py</code>	<code>linecache.py</code>	<code>sndhdr.py</code>
<code>argparse.py</code>	<code>locale.py</code>	<code>socket.py</code>
<code>ast.py</code>	<code>logging</code>	<code>socketserver.py</code>
<code>asynchat.py</code>	<code>lzma.py</code>	<code>sqlite3</code>
<code>asyncore.py</code>	<code>mailbox.py</code>	<code>sre_compile.py</code>
<code>atexit.py</code>	<code>mailcap.py</code>	<code>sre_constants.py</code>
<code>bdb.py</code>	<code>mimetypes.py</code>	<code>sre_parse.py</code>
<code>cgi.py</code>	<code>modulefinder.py</code>	<code>ssl.py</code>
<code>cgitb.py</code>	<code>netrc.py</code>	<code>stat.py</code>
<code>cmd.py</code>	<code>nntplib.py</code>	<code>subprocess.py</code>
<code>code.py</code>	<code>ntpath.py</code>	<code>sunau.py</code>
<code>codecs.py</code>	<code>nturl2path.py</code>	<code>symbol.py</code>
<code>codeop.py</code>	<code>opcode.py</code>	<code>symtable.py</code>
<code>compileall.py</code>	<code>optparse.py</code>	<code>sysconfig.py</code>
<code>concurrent</code>	<code>os.py</code>	<code>tabnanny.py</code>
<code>configparser.py</code>	<code>os2emxpath.py</code>	<code>tarfile.py</code>
<code>copyreg.py</code>	<code>ossaudiodev.py</code>	<code>telnetlib.py</code>

cProfile.py	pdb.py	tempfile.py
curses	pickle.py	test
dbm	pickletools.py	threading.py
difflib.py	pipes.py	tkinter
dis.py	pkgutil.py	token.py
disutils	platform.py	tokenize.py
doctest.py	plistlib.py	trace.py
dummy_threading.py	posixpath.py	traceback.py
filecmp.py	profile.py	tty.py
fileinput.py	pstats.py	turtle.py
fnmatch.py	pty.py	unittest
ftplib.py	py_compile.py	uu.py
genericpath.py	pyclbr.py	venv
getopt.py	pydoc.py	wave.py
getpass.py	quopri.py	webbrowser.py
gettext.py	rlcompleter.py	wsgiref
glob.py	runpy.py	xml
gzip.py	shelve.py	xmlrpc
http	shlex.py	zipfile.py

词汇表

A

Application Licensing (应用程序使用授权)

由使用者/程序设计证书和应用程序授权所组成。Global Scripter 要求用户拥有 Extron 会员账号 (Extron Insider Account) , 只有经过认证的程序员才有权使用该应用软件。授权认证有在线与离线两种方式。

Argument (实际参数)

在函数调用的时候可将值传入至函数中。在函数调用时的实际参数前方加入相关参数的标识符, 及设定参数的关键值就称为关键字实参, 这样在调用时就可以不需要依照函数定义时的参数顺序来传入变量。变量位置在默认情况下是根据对象定义时的顺序而定。

Attribute (属性)

与特定实例化类别相关连的变量, 或是类别本身。

B

Boolean (布林值)

使用 “True” 和 “False” 两个常数表达一个真值。用数值来说, 布林值的内容分别如同整数的 0 与 1。

Build Code Only (只上传程序代码)

此程序只上传与项目有关的程序代码 (code file) 到主控制处理器, 不会更新软件里面的其它系统文件。

Build Message (上传信息栏)

系统进行建立程序时所生成的警告、错误等相关讯息, 显示在 Build Messages 视窗。

Build Project (上传项目)

该程序准备项目中的所有档案, 验证项目中硬件的状态, 再上传这些档案设定到主控制处理器及硬件。

Built-In Function (内建函数)

将许多代码块集中，并在 Python 中重复使用的程序码，并可以在程序内部中进行调用。

注意：部分 Python 内建函数在 ControlScript 软件中是不支持的。

C

Class (类别)

定义一些可表示类别中任何物件的属性。这些属性包含类别变量、实体变量、以及指令方法 (methods)，可经由点 (.) 符号调用这些资料。

Code Editor (程序代码编辑器)

在 Global Scripser 应用程序中的主要区域，进行程序设计代码编辑功能。

Code File (程序代码档案)

一个包含程序所需的代码文件。

Code Snippet (程序代码片段)

程序代码片段是可重复使用的程序码区块，可以插入程序码，当并入常用的功能，可节省编程时间。

Comment/Uncomment (程序注解/解除注解)

加上注解是使用#号将部分程序代码进行注解，作为编程人员对该段代码的注解说明。对一行程序码加上注解可以在软件编译时暂时被忽略。取消注解则是移除先前加入的注解字符。

Constant (常量)

常量值是定义在模块中，并且在程序中的任何地方都不会被改变的变量。常量应只使用大写字母，并在单字与单字之间加上底线。

Control Port Interface (控制接口)

不同的控制端口有各自的类别进行定义，比如继电器端口或串口端口，在 Global Scripser 的程序编辑视窗，可使用特定程序代码将其实例化。

ControlScript (控制脚本)

是 Extron Global Scripter 软件里面随附的程序库，是一个既简易又能保持一致性的控制方法，能处理信息队列和底层界面和并实现自动化处理各种数据。

D

DataType (数据类型)

一种特定类型的数据，可以通过它接受不同的值，从而进行不同的操作定义。

Decorator (修饰)

使用在函数或指令 (method) 的名称前面加上一个 “@” 符号语法，可修饰封装函数或指令，并修改其行为。

Deprecated (弃用)

指那些不建议使用或者应该避免使用的关键词。

Device Alias (设备别名)

用户可以对控制系统中的硬件 (处理器和屏幕) 进行二次命名，称之为设备别名。

Dictionary (字典)

可以进行收集各种没有规律的值并进行存储，并用 key (键) 进行检索其内容的方法，每一个 key 对应一个数值。

Docstring (函数帮助文档)

python 的说明文档字符，用于 python 的模块、函数、类和指令的说明中。使用三个引号 (“ ”) 进行相关内容的说明。

E

Entry Code File (项目初始代码文件)

一个 python 代码文件，附带最基本的模块导入指令。为程序添加新文件时的初始代码文件。

Escape Sequence (转义字符串)

字符型常量所表示的值是字符型变量所能包含的值。我们可以用 ASCII 表达式来表示一个字符型常量，或者用单引号内加反斜杠表示转义字符。

Event (事件)

事件修饰器通常用 @ 符号进行事件的标注。

Extensible (可扩展)

代码具有可扩展性。Python 3.3 是以对象为导向并且是可扩展的，可以使 Extron 对 AV 控制系统进行自定义的事件应用。

Extron Library (Extron 程序库)

Extron 程序库代码提供了程序的集合，在编写程序时可以进行重复使用，最大限度的减少写代码的日常工作。

F

Floating Point Number (浮点数)

Python 支持的一种数字类型，浮点数为小数点后没有固定的数字。

Function (函数)

将一段代码进行整合，并重复使用的方法。

G

Global Scripter

一个用于 Extron Pro 控制系统的综合软件，它包含了所有用于开发控制系统的工具。软件需要用户（编程者）有 Extron 的内部帐号并得到授权才可以使用。

Global Variable (全局变量)

在主文件中进行定义的变量，通常需要在整个程序中进行调用的变量。

Graphical Object (图像对象)

GUI 用户界面布局中的按键等对象，可以指定用做触发相应的控制控制。

GUI Layout (用户界面布局)

控制屏幕的用户自定义图型界面，设计用于控制视音频设备，如信号切换等功能。

I

Immutable (不可变的数据类型)

Python 的数据类型分为可变 (mutable) 与不可变 (immutable)。不可变类型包含字符串 (str)，整数 (int)，元组 (tuple)；可变类型包含列表 (list)，字典 (dict)。

Import (导入)

将一个模块的内容代码应用在别外一个模块中，可以在代码文件外进行相对应函数功能的调用。

Indentation (缩进)

在开始代码行中用空格符号缩进，将代码成组，进行定义相关的代码。

Instantiation (实例化)

生成一个对象叫做实例化。

Integer (整数)

Python 支持的一种数据类型，整数是指一个完整的数字，不包括小数的位。

Interpreter (程序解析器)

电脑程序通过编程语言编写后直接执行，没有编译为机器语言。解析器在其中就作为一种检查和修改的程序的桥梁。

Iterator (迭代)

代是重复反馈过程的活动，其目的通常是为了逼近所需目标或结果。每一次对过程的重复称为一次“迭代”，而每一次迭代得到的结果会作为下一次迭代的初始值。

L

List (列表)

按顺序收集数据并提供索引，在 Python 中列表的位置从 0 开始。

List Comprehension (列表解析)

提供一种便利的方法去修改列表中的信息到一个新的列表中。

Local Variable (本地变量)

在函数中的变量，只会在函数执行时存在。

Loop (循环)

按照一定的顺序不停的重复指令，直到条件满足为止。

M

Main.py (主文件)

当初次新建 Global Scripter 软件的代码文件时所生成 Python 的代码文件。

Method (调用方法)

定义在类中的相应功能指令。

Mirrored User Interface (镜像的用户界面)

和主用户界面一样，显示其相同的图像内容和控制选项。

Module (模块)

程序中单独开发并实现特定功能的程序代码，使用模块可以使程序的可读性提高，并且更容易修改和维护。

Mutable (可变的数据类型)

Python 的数据类型分为可变 (mutable) 与不可变 (immutable)。不可变类型包含字符串 (str)，整数 (int)，元组 (tuple)；可变类型包含列表 (list)，字典 (dict)。

O

Object (对象)

用类的方法实例化数据，对象包含类的变量和方法。

Object Oriented Programming (面向对象的编程)

一种将真实生活中的对象物品例证到程序中，使程序更容易编写的方法。例如，触摸屏硬件有图形画面接口、亮度、动作感应、音量等属性，这些属性可以在程序中进行模块化并进行功能调用。

P**Primary Control Processor (主控制处理器)**

运行主要代码文件的处理器，并可以添加第二个控制器，用户接口等。

Primary User Interface (主用户界面)

用于视音频系统的主要控制点，如 TouchLin Pro 的控制触摸屏幕。

Processor Device (处理器设备)

一段实例化代码，用于在代码编辑器中定义控制处理器。

Program (程序)

一套相应的代码，用于控制处理器中，实现相关的功能。

Program Log (程序日志)

当程序运行时，记录其错误和其它帮助信息的文件。

Project Level File (项目文件)

在软件中添加的其它文件，可以协助用于该控制系统。

Python 3.3

Extron Pro 控制系统所用的 Python 版本号。

R**Regular Expression (正则表达式)**

计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则。正则表达式通常被用来检索、替换某些符合某个模式的文本或进行一些特殊的处理方法，用于帮助查找匹配独立的字符串。其用 'import re' 进行调用。

Reserved Keywords (预留的关键词)

预留一定的关键词以给程序内部进行使用，这些关键词不能用于定义变量使用。

S**Secondary Control Processor (辅助控制处理器)**

增加在主控制处理器，用于扩展系统的控制端口。

Secondary User Interface (第二用户界面)

用于视音频系统的次要控制点，如 Extorn 的网页控制界面，但需要主用户界面进行硬件的连接。

Slice (切片)

一个对象中提取元素，如从字符串列表中提取内容。

String (字符串)

一段语句，如 ‘Hello World’ ，用于在程序中显示文字。字符串可以用单引号或双引号或者三重引号进行标注。

Subclass (子类)

一个从其它类中继承过来的类的称呼，含有父类的特性。

Syntax (语法)

根据不同的规则进行编排，将符号和文字整合到一起，并使程序可以正确的识别的规则。

System Level File (系统文件)

.py 的系统文件。只能用在相关控制系统的硬件中。

T

Toolbelt (控制硬件管理工具)

Extron Pro 控制系统硬件的管理工具。

Trace Message (跟踪信息)

用于 debug 排错代码的信息，用于代码维护。

U

UIDevice (用户界面的类)

用于实例化用户界面的代码。

V

Variable (变量)

变量来源于数学，是计算机语言中能储存计算结果或能表示值抽象概念。变量可以通过变量名访问。在指令式语言中，变量通常是可以变化的对象。

Variable Scope (变量作用域)

变量定义和可使用的部分。其范围的规则决定变量可以或者不可以指向同一个东西。

Version (版本)

项目文件的不同特殊编号，以区分更早的项目文件。

W

Whitespace (空格)

空格，tab 键，和换行符号在程序中交替使用，从而定义不同的代码。