Application Notes
ZHCA725 – Aug 2017

用脚本实现 TPS6598X 的控制和自动化操作

Kevin Dang *Sales and Marketing/Shenzhen*

ABSTRACT

TPS6598x 系列包括 TPS65981/2/3/6/7/8 等，内置 MCU，是能够独立运行的实现 USB Type C 接口的 CC 控制器和 PD 控制器，符合 Type C 规范和 PD 2.0/3.0 标准。另外还集成了高压 LDO，Power Path 的 MOSFET 开关及相应的过压过流反向电流保护，高速信号 Mux 等。是高性能的 Type C 一体化解决方案。除此之外，TPS6598x 还提供了强大的软件支持，简单易用。本文介绍图形界面软件之外，隐藏在背后的可以使用的另外一个强大的工具 Python 脚本的使用，一些自动化的测试和研究都可以使用 Python 脚本来实现。本文的目的则是让平时没有怎么使用脚本的工程师可以从这里开始根据自己的需要编写一定功能的脚本。

Contents

Figures

1 TPS6598x 的脚本安装位置和结构

Python 是直译式的计算机编程语言，执行的时候是一句一句解释执行的。基本上在所有平台上都可以执行，比如 Mac OS, Linux, Windows 都没有问题。因为其面向对象，支持泛型设计，API 丰富等先进的特性，和完全开放的开发，使其变得非常流行，功能也变得非常强大。特别是在 Google 内部广泛使用和推动下，基本成为脚本语言的不二之选。

TPS6598x 也正是选用了 Python 作为其开发使用的脚本语言。在申请和安装了“TPS65982 主机接口实用程序工具”并安装之后，在安装位置（默认为“C:\Program Files\Texas Instruments\TPS6598x Utilities\tps6598x-utilities”）可以看到脚本相关的内容：

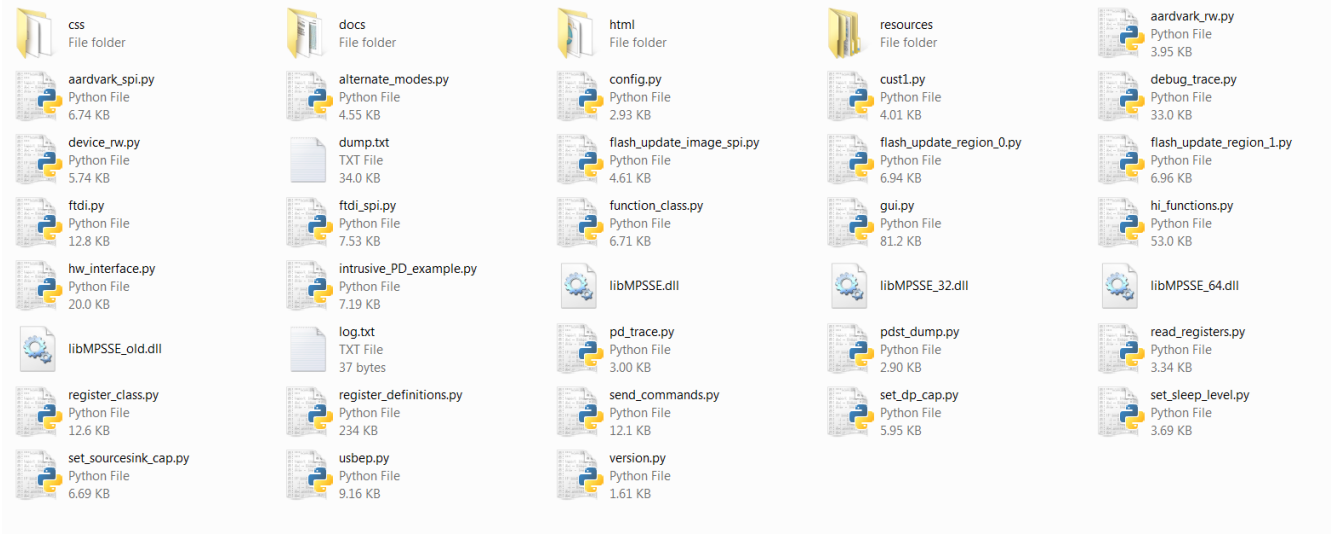


Figure 1. TPS6598x 脚本目录概览

这些结尾是 py 的文件就是 Python 的脚本代码。可以用文本编辑器比如 Notepad++来打开查看和编辑。在安装的时候，Python 的集成开发工具 IDLE 也默认安装并关联了 py 文件，所以也可以使用 IDLE 来打开编辑并可以调试执行脚本文件。

从功能上来讲，基本上分为三层：

实用工具→寄存器类和高层次 4CC/PD 操作函数→tps6598x 寄存器读写和 4CC 命令传输→底层硬件通讯比如 USB to I2C 适配器。

底层硬件通讯：aardvark_rw.py 和 aardvark_spi.py 用于 Aardvark I2C/SPI Host Adapter; ftdi.py, ftdi_spi.py 和 libMPSSE*.dll 几个库是用于 FTDI 的芯片的适配器，usbep.py 用于 TI 的适配器包括一些 USB to I2C 的评估板，这个通过 USB 的控制通道 EP0 来写 Vendor defined message 来实现硬件读写。这些底层通讯的文件，带有_spi 后缀的主要是用于更新 SPI Flash。hw_interface.py 则封装了这些不同硬件的打开关闭和读写操作。相当于适配器的硬件抽象层，是上层和最底层的接口。

寄存器读写和 4CC 命令传输：在这一层封装 tps6598x 的基本操作。device_rw.py 包含了 tps6598x 的寄存器读写函数以及 4CC 就是 4 个字符的命令的读写函数，提供了可以对 tps6598x 进行基本操作的接口。一般来说我们最可能用到的就是这一个文件中的函数。读写 tps6598x 寄存器和 4CC 操作，都通过这个文件里面的函数来实现。但是这里面的具体封装操作可以不用深入研究。

寄存器类和高层次 PD 操作：register_class.py 提供了一些高层次用于 TPS6598x 的寄存器操作的类。function_class.py 和 hi_functions.py 提供了一些 4CC 操作的函数以及一些 PD 的操作函数。寄存器操作的类和 4CC/PD 的操作函数是很好的可以利用的实用库，有了这些支持，很多 PD 和 TPS6598x 的基本的操作我们不需要再编写复杂的代码来处理。

上层控制实用工具：这里包含了一些工具和例子，比如 flash_update_image_spi.py，flash_update_region_0.py 和 flash_update_region_1.py 用于 flash 的更新和部分更新/烧写。gui.py 实现了” TPS6598x Host Interface Tools”。register_definitions.py 和 read_registers.py 则演示了打印寄存器信息的一个例子。这一层对我们实际的开发和使用比较有借鉴意义的是 send_commands.py 和 intrusive_PD_example.py，前者相当于演示了基本操作的使用，后者则演示了非自动协商的操作以便于在特殊情况下使用比如使用动态的 PDO 同时包含了如何轮询中断状态。

其他：config.py 用于全局的设定选项的配置。debug_trace.py 则用于输出调试信息时做一些提高信息可读性的转换。

2 TPS6598x 的标准命令行脚本和中断处理

TPS6598x 提供的 send_commands.py 提供了一些命令可以直接来做一些低层次的操作，这个文件也是非常值得参考的脚本。运行一下可以看到这个脚本提供了哪些操作，运行的方式可以从下图看到，先进入到脚本目录，然后运行 python 来执行这个脚本：

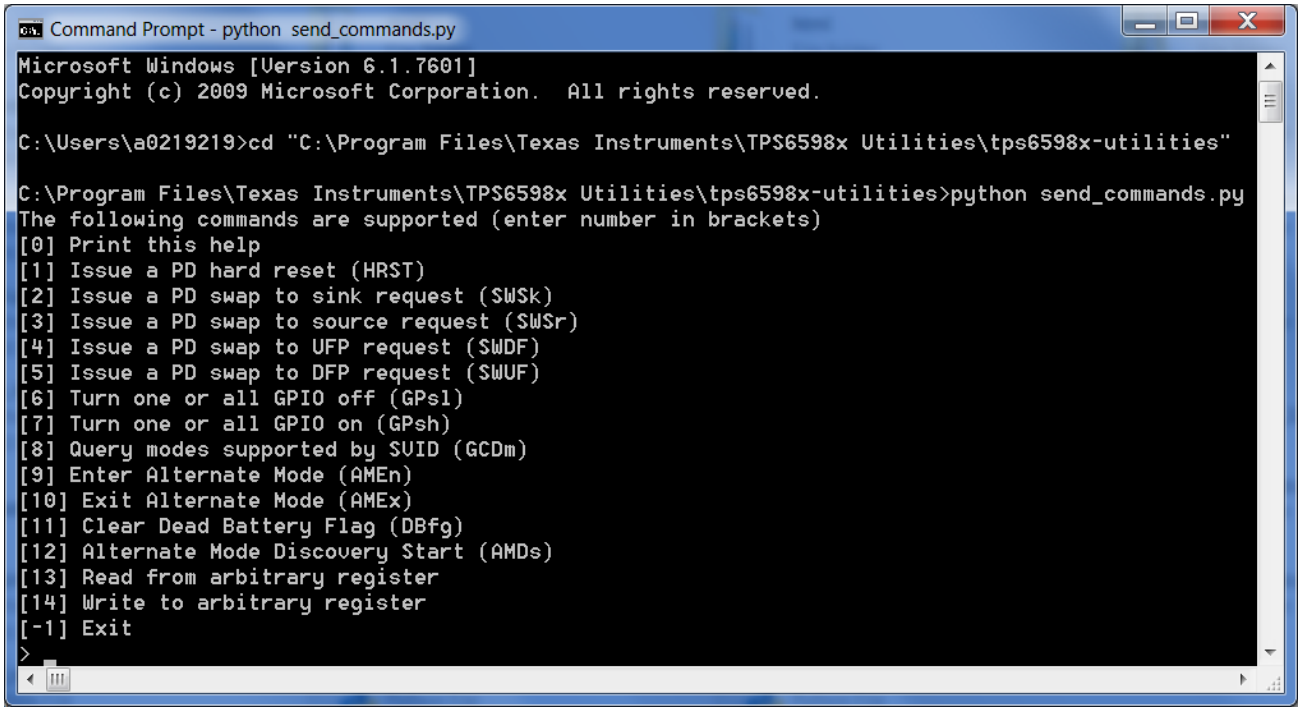


Figure 2. send_commands.py 脚本输出

是否发现了一些想要的东西？可以执行一些 4CC 命令，还能读写任意寄存器……但是如果仅限于这些操作可能在有些应用场景下就显得力不从心了，而且实际的使用中会发现，需要人机交互太频繁，这样就无法实现一些自动化测试类的工作。比如写任意寄存器要三次输入操作才能输入所有信息（Figure 3），这样的方式在批量化的操作中是无法接受的，批量操作的时候要么按照需求修改完全定制化操作的脚本，要么在需要人机交互的时候能够一次性输入所有信息便于拷贝粘贴类操作避免输入错误（比如一次性输入 address, bytes to write, value to write）。打开 send_commands.py 源文件可以看到，为了便于代码处理，这个脚本把不同信息分开进行了输入（Figure 4），所以才导致操作过程复杂。当然这种信息分开输入的方式在做一般的低频次调试的时候，是有好处的，对操作人员来说，这个信息更加易读。

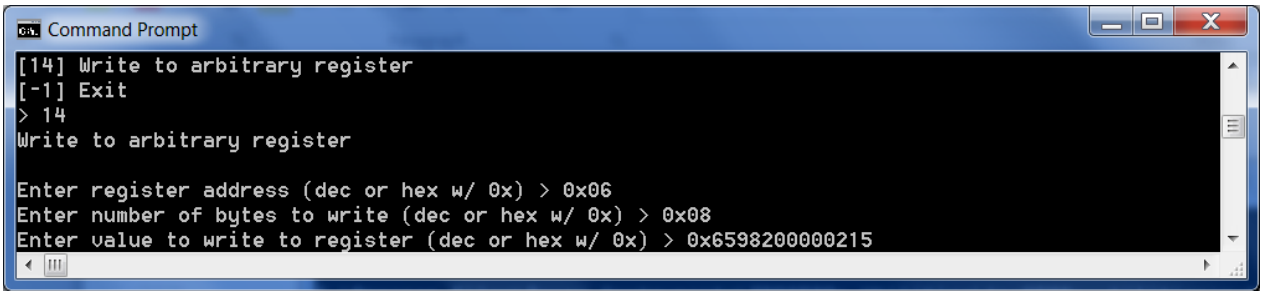


Figure 3.　send_commands.py 写寄存器操作

```
313         if input_int == 14:
314             print 'Write to arbitrary register\n'
315             input_address = input('Enter register address (dec or hex w/ 0x) > ')
316             input_numbytes = input('Enter number of bytes to write (dec or hex w/ 0x) > ')
317             input_value = input('Enter value to write to register (dec or hex w/ 0x) > ')
318 
319             write_reg(handle, int(input_address), buildByteArray(int(input_value), int(input_numbytes)) )
```

Figure 4.　send_commands.py 写寄存器代码

然后我们再来看中断的处理，中断处理在特殊的情况下才需要，最常见的可能是获取相关错误信息，模式发生了变化比如进入或者退出 DP/HDMI alt mode，处理 VDM 消息，介入 PD 协商（Intrusive mode）等。可以用于在自动化处理时记录错误信息和运行情况，改变测试流程和项目等。关于中断的处理方法，可以参照 intrusive_PD_example.py。因为目标板和 PC 通过 USB 或者其他接口来连接，所以实际的事件中断是通过 pulling 的方式来获取的。脚本 intrusive_PD_example.py 提供了一个非常有价值的参考，它实现了 Source/Sink 的 intrusive mode 协商。同时他还充分利用了 hi_functions.py 和 register_class.py 里面提供的寄存器操作，4CC 和 PD 操作，代码看起来非常的简洁：

```
58  #操作INT_MASK1寄存器，这里用到了cRegister类，在register_class.py中定义
59  def enable_interrupt(handle, int_name) :
60      INT_MASK1.read(handle)
61      INT_MASK1.fieldByName(int_name).value = 1
62      INT_MASK1.write(handle)
```

Figure 5.　使用 cRegister 寄存器类进行寄存器操作

```
69  def negotiate_Source(handle):
70      print('Sending Source Capabilities:')
71      TX_SOURCE_CAP.read(handle)
72      TX_SOURCE_CAP.show()
73
74      enable_interrupt(handle, 'RDO Received from Sink')
75      clear_interrupt(handle, 'RDO Received from Sink')
76
77      print 'Issuing SSrC'
78      #Send Source Capabilities, 这里直接调用hi_functions.py中的函数，非常方便
79      print SSrC(handle)
80
81      poll_interrupt(handle, 'RDO Received from Sink')
82
```

Figure 6.　使用 hi_functions.py 提供的函数简化操作

总结一下，send_commands.py 和 intrusive_PD_example.py 是非常值得研读的。这里不便列出所有具体内容做展开讲解，可以在安装软件后自行详细阅读和理解。

3　开始编写一个自己的脚本

这里从一个简单功能需求开始，做一个自己的脚本。目标是使用 TPS65982 的 EVM 来模拟一个 Type C 的 Sink 设备，来测试不同的电压选项比如最高 12V 最高 20V 等。

如果使用 send_commands 来改变一组新的 PDO 选项来测试的话，操作过程是写寄存器（输入地址，长度，内容），然后 PD 做一个 Reset 动作 HRST 来完成。这里面要做 4 次不同的输入，操作繁琐容易发生错误。所以用定制的脚本来实现这个功能是非常合适的，但是如果完全定制的话每次测试内容发生改变就需要修改脚本也比较麻烦，而且每次脚本修改后都要对脚本进行充分测试避免错误，所以这里我们还允许一定的交互性，但是减少不必要的人机交互的环节，这样测试内容发生变化不用修改脚本。

因为篇幅关系略去一步步的编写过程，这里列出大概的流程图，以及对已经写好的代码进行详细的解释。

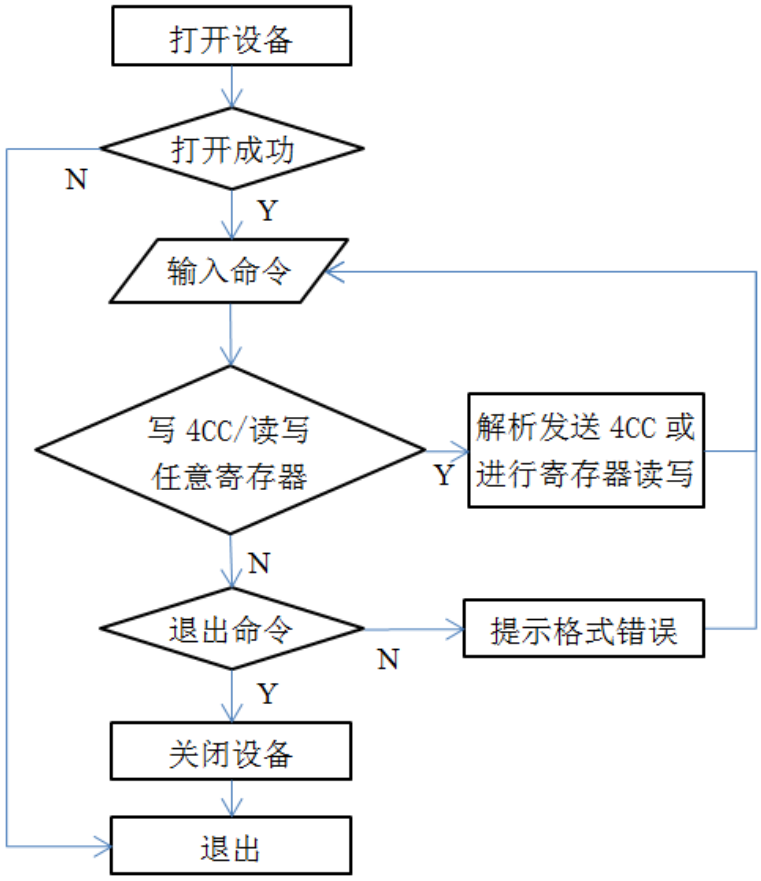


Figure 7.　自定义脚本的处理流程图

下面开始对具体的代码进行详细的解释，Python 的注释是以'#'开头，所以这里对代码的解释也会放在注释里面。

```
#!/bin/env python
# -*- coding: utf-8 -*-
# 上面第一行指定了本脚本使用的解释器，第二行指定了字符和字符串的编码格式，因为我这里注释使用了中文，
# 所以使用utf-8编码，其他文件默认都是cp1252编码，如果文件中存在非ASCII码会出错。
# 工厂测试用的话建议使用英文，然后错误提示可以包含错误码比如ERROR 01便于记录和上报
# 这里略去版权声明，TI提供的这些脚本文件都是开源的可以修改重新发行，但是不承担任何责任和连带责任
#=============================================================================
# IMPORTS
#=============================================================================
# 下面引用系统标准库，包含比如输入输出等功能
import sys
# 引用标准库：数组处理等功能
from array import array, ArrayType
# 引用寄存器类和高层函数支持，hi_functions定义的函数当前是没有用到的，但是他引用了一些硬件操作的支持，
# 比如引用了device_rw，而且这个文件里面的函数很有用，将来可能很可能会用到，所以这里就直接引用这个文件
from register_definitions import *
```

```
from hi_functions import *
# 引用硬件操作类，通过这个接口访问硬件
from hw_interface import *
#============================================================================
# GLOBALS: 全局变量
#============================================================================
# config.py 中定义了硬件相关的配置，这里直接引用这些配置，建立了一个hw_interface类的实例
# 如果需要修改配置，则修改config.py
handle = hw_interface(config.DEVICE_I2C_ADDR, config.HW_INTERFACE)
# 这里定义的函数，用于把输入数据转为发送大量数据使用的字节数组
def buildByteArray(data, numBytes) :
  retval = array('B')
  for i in range(numBytes) :
    retval.append((data & (0xFF << 8*i)) >> 8*i)
  return retval
#============================================================================
# main: 主程序，代码入口在这里，就是从这里开始执行
#============================================================================
if __name__ == "__main__":
  # 显示一下代码的功能，‘\n’表示回车换行，print 函数会自动在最后加上换行回车所以最后不用加’\n’
  print '\nWrite to arbitrary register'
  print '    Input: addr + len + val'
  print '    e.g. 01020304\naddr is 0x01, length is 0x02, values are 0x0304'
  print 'Read register: r + addr + len. E.g. r0102 means read 0x02 bytes from 0x01'
  print 'Write 4CC is just input 4CC, e.g. HRST'
  print 'To Exit enter 0'
  # 下面开始打开硬件，Python是面向对象的模块化语言，所以其错误处理也主要是和C++类似的try-catch方式
  # 对于可能发生错误并需要对错误进行处理的操作（错误如果不处理就会直接退出程序），就需要用这种方式
  # 把操作放在try模块中，然后捕获想要处理的exception进行错误处理
  try :
    handle.hw_open(config.HW_INTERFACE, config.PORT, config.SPI_PORT, config.BITRATE,
config.SPIBITRATE, config.DEVICE_I2C_ADDR)
  except Exception as e :
    # 这里只是简单打印错误信息并退出。另外一种需求可能要继续尝试不退出，那么就要把try-except这段放
    # 在一个循环中，打印错误信息然后等待一段时间 (使用定时器time.sleep) ，继续循环尝试打开设备。
    print e.message
    sys.exit()
  # 主循环开始，while后面是判定条件，如果是true那就一直执行
  while 1:
    # 提示输入命令数据并获取。这里使用raw_input而不是input，因为我们要的是输入的原始字符串
    input_values_ori = raw_input('Please Enter Command > ')
    # 开始处理输入。这里我们对将要进行处理的命令进行分析，发现比较好分类，对输入的字符串，因为读写命令
    # 都要用到地址和长度所以其长度大于4个字节，4CC的话就只有4个字节，读和写要区分，这里要求读命令前面
    # 加一个字符’r’，这样的话读命令就成为固定的5个字符，写的话大于等于5个字符但是不是以’r’开头，
    # 如果输入’0’的话，就退出。这样就区分开了所有的命令。为了简单说明问题，下面略去了对输入合法性检查
    # 的处理。就是默认输入都是正确的，不对输入错误进行检查。
    # len函数获取字符串的长度
    if len(input_values_ori) < 5:
      if input_values_ori[0] == '0':
        print 'To Exit'
        # break用于跳出while循环。
        break
```

```
      # 判断退出后接下来如果不是'0'则假定为写4CC操作，略去合法性检查，打印提示，然后直接调用4CC操作。
      # 本文用到的寄存器读写操作和4CC操作，都定义在device_rw.py中。
      print 'Write 4CC: %s' %(input_values_ori)
      try:
        write_reg_4cc(handle, 0x08, input_values_ori)
      except Exception as e:
        print e.message
      # continue 用于跳转到while循环的最开始继续执行。
      continue
    # 读寄存器操作的判断
    if input_values_ori[0] == 'r':
      # 这里进行了字符串提取的操作，Python的字符串有类似数组的索引操作[]。截取字符串也非常方便
      # 比如这里input_values_ori[1:3]截取从位置1开始到位置3之前的字符串
      addr = input_values_ori[1:3]
      length = input_values_ori[3:5]
      # 调用read_reg读寄存器，返回的数值是包含了读的字节数和数据。给read_reg传入的参数则是设备句柄
      # 地址和长度。这里使用数据转换int来把字符串转为数值，因为我们是要求16进制数据，所以转换基数16
      try:
        (count, readBytes) = read_reg(handle, int(addr, 16), int(length, 16))
      except Exception as e:
        count = 0
        print e.message
      # 按照寄存器格式转为32位单位的数值，同时对字序进行调整。因为PC来讲都是Little Endian的，而读回
      # 的数值是Big Endian，所以32位数的四个字节刚好次序颠倒
      j = 0
      for i in range((count-1) // 4) :
        value32 = readBytes[4*i]|(readBytes[4*i+1] << 8)|(readBytes[4*i+2] << 16)|
(readBytes[4*i+3] << 24)
        print '0x%08x' %value32
        j+=1
      # 32位对齐的补齐操作
      if (count-1) > 4*j :
        value32 = 0x0
      for i in range(count - 1 - 4*j) :
        value32 |= readBytes[4*j + i] << 8*i
        print '0x%08x' %value32
      # 完成读寄存器操作，继续while循环
      continue
    # 下面是写寄存器操作，把地址长度和数据分别提取出来，然后转换为数值和字节数组，调用write_reg写操作
    addr = input_values_ori[:2]
    length = input_values_ori[2:4]
    values = input_values_ori[4 - len(input_values_ori):]
    print 'addr: 0x%02x len: 0x%02x values: 0x%x' %(int(addr, 16), int(length, 16),
int(values, 16))
    try:
      write_reg(handle, int(addr, 16), buildByteArray(int(values, 16), int(length, 16)))
    except Exception as e:
      print e.message
  # Close the device
  handle.hw_close()
```

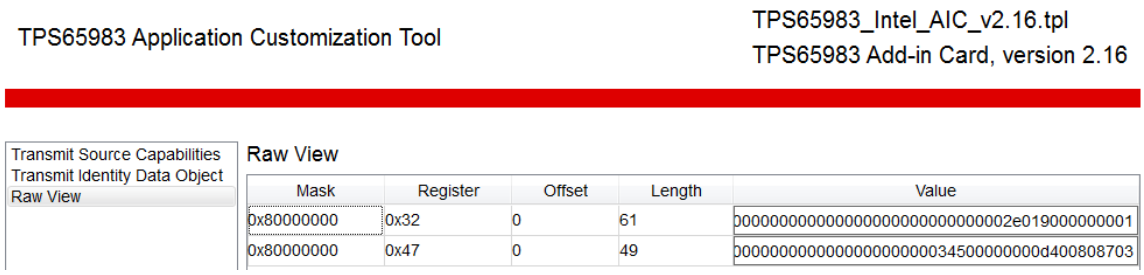
保存为 py 为后缀的文件方便区分，这里保存为 my_script.py。运行一下：

```
C:\Program Files\Texas Instruments\TPS6598x Utilities\tps6598x-utilities>python my_script.py

Write to arbitrary register
    Input: addr + len + val
    e.g. 01020304
addr is 0x01, length is 0x02, values are 0x0304
Read register: r + addr + len. E.g. r0102 means read 0x02 bytes from 0x01
Write 4CC is just input 4CC, e.g. HRST
To Exit enter 0

TPS65982 Debug Tool Version 2.0

 FTDI Interface loading Error:
```

因为没有插设备所以会报错然后退出。设备如果连接成功会显示

```
TPS65982 Debug Tool Version 2.0

I2C Bitrate set to 100 kHz
SPI Bitrate set to 8000 kHz
```

写什么数值是要根据需要和特定的 firmware 来确定的。一般可以使用“TPS6598x Application Customization Tool”来获得需要的寄存器和数值:



这里随便选一个数值测试然后做一个 Reset:

```
To Exit enter 0
Please Enter Command > 323d2e01912c1001000000000000000000000000000000000000000000000002e019000000001
addr: 0x32 len: 0x3d values: 0x2e01912c1001000000000000000000000000000000000000000000000002e019000000001
Please Enter Command > HRST
Write 4CC: HRST
```

发送 HRST 后可以看到板子上指示灯闪烁一下，然后这边会抛出 exception 断开。如果做自动化测试用途，则需要用 try-except 来接管这些 exception 做相应的处理以便测试能够自动继续。

4 注意事项

初次接触 Python 特别是原来使用 C/C++的工程师，看过去可能感觉 Python 代码挺乱，感觉不容易看准确代码分段的开始和结束。这正是 Python 的特点，用缩进对齐来标记代码块。而且这个对齐非常的严格，比如 4 个空格和 1 个占 4 个空格位置的 Tab 是不等同的，虽然看起来是一样的。当你碰到类似错误，一般都是这个原因引起，比如我把随便一行应该对齐的空格改为一个 Tab：

```
C:\Program Files\Texas Instruments\TPS6598x Utilities\tps6598x-utilities>python my_script.py
  File "my_script.py", line 64
    print '    e.g. 01020304\naddr is 0x01, length is 0x02, values are 0x0304'
                                                                             ^
IndentationError: unexpected indent
```

另外一个就是 print 函数的使用，后面的变量和前面的字符串中的占位符要一一对应，然后建议变量放在括号中比如 print 'Write 4CC: %s' %(input_values_ori)。%s 字符串占位符对应 input_values_ori。

5 参考文献

1. *TPS65981, TPS65982, and TPS65986 Host Interface Technical Reference Manual (Rev. A) (slvuan1a)*
2. https://zh.wikipedia.org/wiki/Python *建议通读此文，内容简单明了，非常适合初学者*
3. https://www.python.org/

有关 TI 设计信息和资源的重要通知

德州仪器 (TI) 公司提供的技术、应用或其他设计建议、服务或信息，包括但不限于与评估模块有关的参考设计和材料（总称“TI 资源”），旨在帮助设计人员开发整合了 TI 产品的 应用； 如果您（个人，或如果是代表贵公司，则为贵公司）以任何方式下载、访问或使用了任何特定的 TI 资源，即表示贵方同意仅为该等目标，按照本通知的条款进行使用。

TI 所提供的 TI 资源，并未扩大或以其他方式修改 TI 对 TI 产品的公开适用的质保及质保免责声明；也未导致 TI 承担任何额外的义务或责任。TI 有权对其 TI 资源进行纠正、增强、改进和其他修改。

您理解并同意，在设计应用时应自行实施独立的分析、评价和 判断， 且应全权负责并确保 应用的安全性， 以及您的 应用 （包括应用中使用的所有 TI 产品）） 应符合所有适用的法律法规及其他相关要求。你就您的 应用声明，您具备制订和实施下列保障措施所需的一切必要专业知识，能够 (1) 预见故障的危险后果，(2) 监视故障及其后果，以及 (3) 降低可能导致危险的故障几率并采取适当措施。您同意，在使用或分发包含 TI 产品的任何 应用前， 您将彻底测试该等 应用 和该等应用所用 TI 产品的 功能。除特定 TI 资源的公开文档中明确列出的测试外，TI 未进行任何其他测试。

您只有在为开发包含该等 TI 资源所列 TI 产品的 应用时， 才被授权使用、复制和修改任何相关单项 TI 资源。但并未依据禁止反言原则或其他法理授予您任何TI知识产权的任何其他明示或默示的许可，也未授予您 TI 或第三方的任何技术或知识产权的许可，该等产权包括但不限于任何专利权、版权、屏蔽作品权或与使用TI产品或服务的任何整合、机器制作、流程相关的其他知识产权。涉及或参考了第三方产品或服务的信息不构成使用此类产品或服务的许可或与其相关的保证或认可。使用 TI 资源可能需要您向第三方获得对该等第三方专利或其他知识产权的许可。

TI 资源系“按原样”提供。TI 兹免除对 TI 资源及其使用作出所有其他明确或默认的保证或陈述，包括但不限于对准确性或完整性、产权保证、无屡发故障保证，以及适销性、适合特定用途和不侵犯任何第三方知识产权的任何默认保证。

TI 不负责任何申索，包括但不限于因组合产品所致或与之有关的申索，也不为您辩护或赔偿，即使该等产品组合已列于 TI 资源或其他地方。对因 TI 资源或其使用引起或与之有关的任何实际的、直接的、特殊的、附带的、间接的、惩罚性的、偶发的、从属或惩戒性损害赔偿，不管 TI 是否获悉可能会产生上述损害赔偿，TI 概不负责。

您同意向 TI 及其代表全额赔偿因您不遵守本通知条款和条件而引起的任何损害、费用、损失和/或责任。

本通知适用于 TI 资源。另有其他条款适用于某些类型的材料、TI 产品和服务的使用和采购。这些条款包括但不限于适用于 TI 的半导体产品 (http://www.ti.com/sc/docs/stdterms.htm)、评估模块和样品 (http://www.ti.com/sc/docs/sampterms.htm) 的标准条款。

邮寄地址：上海市浦东新区世纪大道 1568 号中建大厦 32 楼，邮政编码：200122