

Kernel Debug 介绍

- 2021/02/01

• Agenda

- 调试简介
- printk和分级log系统
- 动态debug系统
- Ftrace跟踪器简介
- 源码级调试器简介(GDB/KGDB/KDB)
- 探针简介
- Linux dump简介
- 示波器，万用表，逻辑分析器等仪器简介

- 调试简介
- 由于Linux kernel的特殊性，使得对于kernel的调试没有应用程序调试方便，为了解决kernel调试的问题，内核开发者们开发出了各种调试的工具，以此来解决调试的问题，我们将从接下来的方面介绍kernel调试；

- **printf和分级log系统**

- ❖ 控制log是否打印的常用手段是log等级，例如printf(), 支持的等级有KERN_DEBUG、 KERN_INFO以及KERN_ERR等，例如
 - `printf(KERN_INFO "test log");`
- ❖ cat /proc/sys/kernel/printk可查看控制台相关log等级，结果默认为4个数字，从左到右其含义依次是
 - 控制台日志级别：优先级高于该值的消息将被打印至控制台
 - 默认消息日志级别：将用该优先级来打印没有优先级的消息
 - 最低控制台日志级别：控制台日志级别可被设置的最小值(最高优先级)
 - 默认控制台日志级别：控制台日志级别的缺省值

- 动态debug系统

- ❖ 仅仅根据等级来控制log不够灵活，pr_debug()提供了动态控制log是否输出的方法：
 - 可通过[`/d/dynamic_debug/control`](#)控制某个目录、源文件、代码行、函数所调用的pr_debug()是否输出。

❖ 为了动态控制pr_debug(), 需要打开如下内核配置

- CONFIG_DEBUG_FS=y
- CONFIG_DYNAMIC_DEBUG=y

❖ 若debugfs没被自动挂载, 可通过下面命令手动挂载

- mount debugfs /d
- mount debugfs /sys/kernel/debug

❖ 此时可`cat /d/dynamic_debug/control`查看pr_debug()的调用情况

```
root@ubuntu:/sys/kernel/debug/dynamic_debug# cat control | head
# filename:lineno [module]function flags format
init/main.c:741 [main]initcall_blacklisted =p "initcall %s blacklisted\012"
init/main.c:717 [main]initcall_blacklist =p "blacklisting initcall %s\012"
init/initramfs.c:483 [initramfs]unpack_to_rootfs =_ "Detected %s compressed data\012"
arch/x86/kernel/tboot.c:104 [tboot]tboot_probe =_ "tboot_size: 0x%x\012"
arch/x86/kernel/tboot.c:103 [tboot]tboot_probe =_ "tboot_base: 0x%08x\012"
arch/x86/kernel/tboot.c:102 [tboot]tboot_probe =_ "shutdown_entry: 0x%x\012"
arch/x86/kernel/tboot.c:101 [tboot]tboot_probe =_ "log_addr: 0x%08x\012"
arch/x86/kernel/tboot.c:100 [tboot]tboot_probe =_ "version: %d\012"
arch/x86/kernel/cpu/common.c:1356 [common]cpu_init =_ "Initializing CPU%d\012"
```

- ❖ 上图每行格式为

filename:lineno [module]function flags format

- ❖ 以第3行为例说明如下

属性	具体值	说明
filename	init/main.c	调用pr_debug()的代码路径
lineno	741	调用pr_debug()的代码行号
module	main	调用pr_debug()的代码模块
function	initcall_blacklisted	调用pr_debug()的函数
flags	=p	p、f、l、m、t等标志，_表示空
format	"initcall %s blacklisted\012"	打印语句

- ❖ flags的值说明如下

flags	说明
p	打开动态打印语句
f	打印函数名
l	打印行号
m	打印模块名称
t	打印线程ID

- ❖ 特别说明：flags为p表示该行pr_debug()被使能！

- ❖ 可往 /d/dynamic_debug/control 写入特定值来修改上面属性值，关键词如下所示

关键词	说明
file	指定文件
line	指定代码行
module	指定模块
func	指定函数
format	指定log格式

- ❖ +<flag>表示增加<flag>标志； -<flag>表示删除<flag>标志，如下面命令可将test.c文件的flags设置为p

```
echo -n 'file test.c +p' > /d/dynamic_debug/control
```

❖ pr_debug()默认不打印， 可通过如下方式进行控制



- 使能test.c所有pr_debug()

```
echo -n 'file test.c +p' > /d/dynamic_debug/control
```

- 使能test.c第9行pr_debug()

```
echo -n 'file tes.c line 9 +p' > /d/dynamic_debug/control
```

- 使能test_dir目录所有pr_debug()

```
echo -n 'file test_dir/* +p' > /d/dynamic_debug/control
```

• Ftrace跟踪器简介

- ❖ ftrace是一个内核中的追踪器，用于帮助系统开发者或设计者查看内核运行情况，它可以被用来调试或者分析延迟/性能问题。
- ❖ 最早ftrace是一个function tracer，仅能够记录内核的函数调用流程。如今ftrace已经成为一个framework，采用plugin的方式支持开发人员添加更多种类的trace功能。

❖ debugfs

- CONFIG_DEBUG_FS=y
- CONFIG_DYNAMIC_DEBUG=y
- 打开上述配置后，可通过下面命令挂载debugfs
 - mount debugfs /sys/kernel/debug

❖ ftrace

- CONFIG_FTRACE
- CONFIG_FUNCTION_TRACER
- CONFIG_FUNCTION_GRAPH_TRACER
- CONFIG_CONTEXT_SWITCH_TRACER
- CONFIG_NOP_TRACER
- CONFIG_PREEMPT_TRACER

- ❖ ftrace相关目录为`/sys/kernel/debug/tracing`, 例如S820平台 tracing目录结构如下所示

```
msm8996:/sys/kernel/debug/tracing # ls
README                                instances          trace_clock
available_events                      options           trace_marker
available_tracers                     per_cpu           trace_options
buffer_size_kb                        printk_formats   trace_pipe
buffer_total_size_kb                 saved_cmdlines  tracing_cpumask
cpu_freq_switch_profile_enabled     saved_cmdlines_size tracing_on
current_tracer                       saved_tgids       tracing_thresh
events                               set_event
free_buffer
```

- ❖ ftrace相关操作都可通过该目录及其子目录的属性实现, 例如打开或关闭ftrace等。

❖ 可通过下面命令查看可用的tracer

- cat available_tracers

```
msm8998:/sys/kernel/debug/tracing # cat available_tracers
blk function_graph preemptirqsoff preemptoff irqsoff function nop
```

❖ 可通过下面命令选择特定tracer

- echo <tracer_name> > current_tracer
 - 注: <tracer_name>是available_tracers的结果之一, 例如function_graph

❖ 开启ftrace

- echo 1 > /sys/kernel/debug/tracing/tracing_on

❖ 关闭ftrace

- echo 0 > /sys/kernel/debug/tracing/tracing_on

- ❖ ftrace的输出信息主要保存在trace文件中，例如S820平台current_tracer为function_graph时结果如下所示

```
# tracer: function_graph
#
# CPU  DURATION          FUNCTION CALLS
# |    |    |
#)  0.156 us  } /* _raw_spin_unlock_irq */
#)  0.209 us  _raw_spin_lock_irq();
#)
#)  0.208 us  __wake_up() {
#)
#)  0.990 us  __raw_spin_lock_irqsave();
#)  0.209 us  __wake_up_common() {
#)
#)  0.157 us  pollwake() {
#)
#)  1.979 us  default_wake_function() {
#)
#)  3.802 us  try_to_wake_up() {
#)
#)  0.209 us  __raw_spin_lock_irqsave();
#)  0.209 us  __raw_spin_lock();
#)  0.157 us  sched_ktime_clock() {
#)
#)  1.979 us  ktime_get() {
#)
#)  3.802 us  arch_counter_read();
#)
```

- 源码级调试器简介
- GDB
- KDB
- KGDB

- GDB(The GNU Project Debugger), 是GNU开源组织发布的一个强大的UNIX下的程序调试工具。GDB 支持断点、单步执行、打印变量、观察变量、查看寄存器、查看堆栈等调试手段，GDB可以调试应用程序，也可以和KGDB配合来调试内核源代码；

❖ GDB常用命令

命令	功能	命令	功能
breakpoint (简写b)	添加断点	finish (简写fin)	执行到退出当前函数
continue(简写c)	继续执行	info threads	查看当前进程的线程
printf (简写p)	打印变量的值	info registers	查看当前各寄存器信息
set	修改某个变量的值	x/<n/f/u><addr>	查看变量内存中的值
enable breakpoints [num]	使能之前加过的断点，可以指定 num，不指定就是对所有断点都使能。默认断点加入后就是使能状态	attach <pid>	关联指定进程
disable breakpoints [num]	去使能之前加过的断点	show args	显示运行时的参数
next (简写n)	执行一行源程序代码	detach	直接取消当前挂住的pid
step (简写s)	单步跟踪进入	run(简写r)	执行程序
until	在for循环中可以使用，直接运行到循环结束	file <文件名>	加载被调试的可执行程序文件
info break	查看断点信息		
list	显示源代码，需要版本编译时 -g		
watch	监视某块内存，当内存被改变时触发		
bt	查看当前堆栈信息		

- KGDB是一个Linux系统的内核调试器，在内核2.6.26版本时正式引入内核，调试时需要两台机器，通过和主机端的GDB来配合，从而实现内核代码的远程调试。

内核配置

CONFIG_HAVE_ARCH_KGDB=y

CONFIG_KGDB=y

CONFIG_KGDB_SERIAL_CONSOLE=y

CONFIG_KGDB_KDB=y

在高通平还需要关闭CONFIG_MSM_WATCHDOG_V2

- KDB(Linux kernel debugger)是一个Linux系统的内核调试器，它是由SGI公司开发的遵循GPL许可证的开放源码调试工具；它适合于调试内核空间的程序代码，不需要两台机器进行调试。

- 内核配置

CONFIG_KGDB=y

CONFIG_KGDB_KDB=y

Turn off CONFIG_DEBUG_RODATA

CONFIG_FRAME_POINTER=y

CONFIG_HAVE_FUNCTION_GRAPH_TRACER=y

CONFIG_KGDB_SERIAL_CONSOLE=y

- 使用

添加到内核引导参数:kgdboc=ttyAMA0,115200

设备终端输入: echo ttyAMA0 > /sys/module/kgdboc/parameters/kgdboc

进入kdb模式: echo g > /proc/sysrq-trigger

• 探针简介



- **Kprobes**是专门为了便于跟踪内核函数执行状态所设计的一种轻量级内核调试技术，利用**kprobe**技术可以在内核函数中动态插入探测点，以此来得到所需要的调试信息。
- **Kprobes**包含三种探测方法

kprobe

jprobe

kretprobe

- `kprobe`是最基本的探测方式，是`jprobe`和`kretprobe`的实现基础，`kprobe`可以在任意的位置放置探测点，可以细化到指令级，它提供探测点调用前，调用后以及内存访问出错的回调方式。

- struct kprobe {
.....
 const char *symbol_name; //需要探测的函数名称
 unsigned int offset; //被探测点在函数内部的偏移，用于探测函数内核的指令，如果该值为0表示函数的入口。
 /* Called before addr is executed. */
 kprobe_pre_handler_t pre_handler;
 /* Called after addr is executed, unless... */
 kprobe_post_handler_t post_handler;
 /* ... called if executing addr causes a fault (eg. page fault).
 * Return 1 if it handled fault, otherwise kernel will see it.*/
 kprobe_fault_handler_t fault_handler;
.....
};

- 涉及的API

int register_kprobe(struct kprobe *p); //注册一个探测点

void unregister_kprobe(struct kprobe *p); //注销一个探测点

int register_kprobes(struct kprobe **kps, int num); //同时注册多个探测点

void unregister_kprobes(struct kprobe **kps, int num); //同时注销多个探

测点

int disable_kprobe(struct kprobe *kp); //停止一个探测点

int enable_kprobe(struct kprobe *kp); //使能一个探测点

- 基于kprobe来只对函数进行探测，不能在函数内部插入探测点，只能在入口处插入探测点。
- jprobe的回调函数应当和被探测函数有同样的原型，jprobe的回调函数返回时必须调用jprobe_return()

- jprobe结构体

```
struct jprobe {  
    struct kprobe kp;  
    void *entry; /* probe handling code to jump to */  
};
```

- 结构变量定义样例

```
static struct jprobe jp = {  
    .entry           = j_kprobe_target, //探测点回调函数  
    .kp.symbol_name = "kprobe_target" //需要探测的函数名称  
};
```

备注：kprobe的pre_handler此时固定为已有函数setjmp_pre_handler，它对栈或者寄存器进行相关的操作，保存现场以备调用结束后恢复。

- 涉及的API

int register_jprobe(struct jprobe *jp) //注册jprobe探测点

void unregister_jprobe(struct jprobe *jp) //注销jprobe探测点

int register_jprobes(struct jprobe **jps, int num) //注册探测函数向量，
包含多个不同探测点

void unregister_jprobes(struct jprobe **jps, int num) //注销探测函数向量，
包含多个不同探测点

int disable_jprobe(struct jprobe *jp) //停止指定探测点的探测

int enable_jprobe(struct jprobe *jp) //使能指定探测点的探测

- **kretprobe**基于**kprobe**实现，可用于探测函数的返回值以及计算函数执行时间。

• kretprobe结构体

```
struct kretprobe {  
    struct kprobe kp;  
    kretprobe_handler_t handler; //在被探测函数返回后被回调  
    kretprobe_handler_t entry_handler; //在被探测函数执行之前被回调  
    int maxactive; //支持并行探测的上限(被探测函数被多个进程同时调用)  
    int nmissed;  
    size_t data_size; //kretprobe私有数据的大小  
    struct hlist_head free_instances; //表示空闲的kretprobe运行实例链表， 实例结构见下页  
    raw_spinlock_t lock;  
}
```

```
struct kretprobe_instance {  
    struct hlist_node hlist;  
    struct kretprobe *rp;  
    kprobe_opcode_t *ret_addr;//保存原始被探测函数的返回地址  
    struct task_struct *task;//用于绑定其跟踪的进程  
    char data[0];//保存用户使用的kretprobe私有数据  
}
```

这个结构体表示kretprobe的运行实例，由于被探测函数有可能存在并发执行的情况，kretprobe_instance用来表示一个执行流。

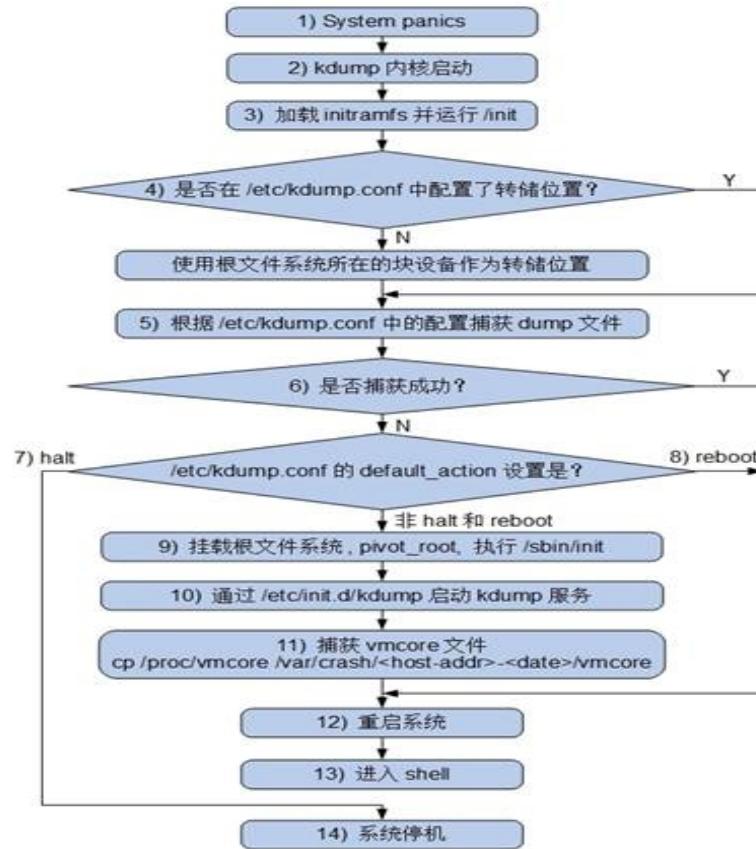
- 涉及的API

```
int register_kretprobe(struct kretprobe *rp)
void unregister_kretprobe(struct kretprobe *rp)
int register_kretprobes(struct kretprobe **rps, int num)
void unregister_kretprobes(struct kretprobe **rps, int num)
int enable_kretprobe(struct kretprobe *rp)
int disable_kretprobe(struct kretprobe *rp)
```

• Linux dump简介

- kernel dump可以用于分析kernel crash或hang的原因，Linux使用kdump工具来捕捉kernel dump并保存到文件中。
- kdump的原理是启动一个特殊的dump-capture kernel，它把系统内存里的数据保存到文件里，dump-capture kernel可以是独立的，也可以与系统内核集成在一起，当正常的kernel发生crash时，会触发dump-capture kernel的启动。

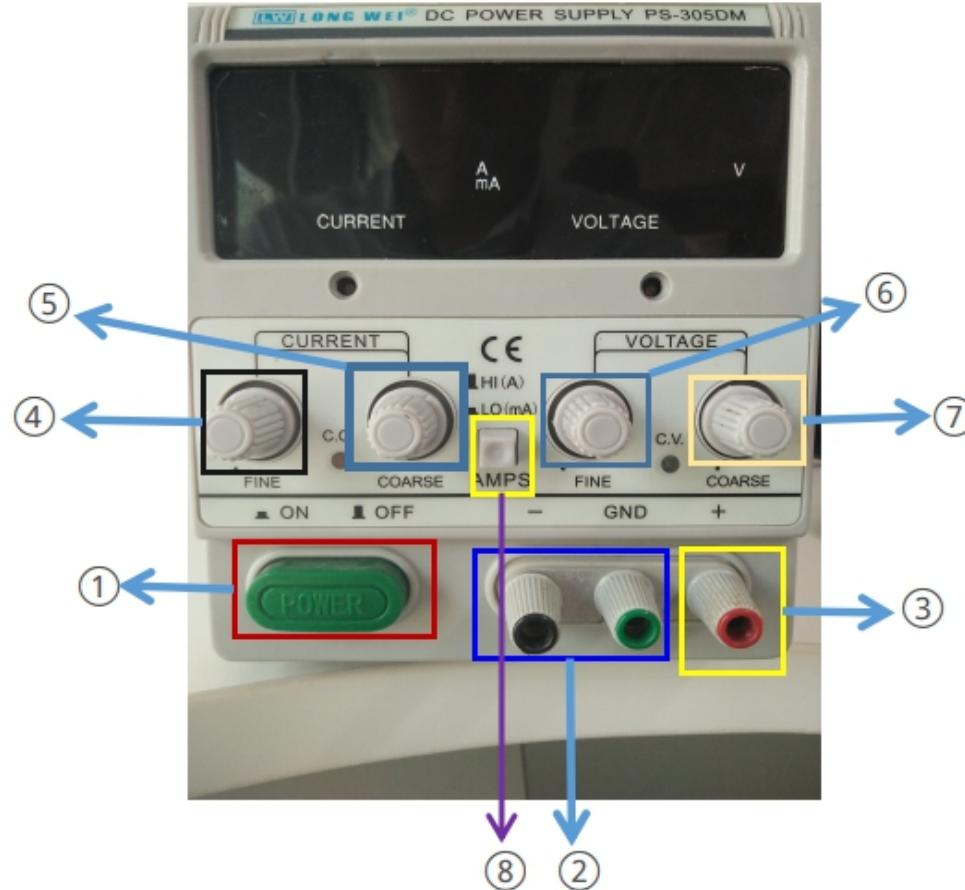
❖ kdump工作流程



- 解析dump文件
- 获取dump文件后，通过Crash工具进行解析，命令如下：
`crash vmlinux(kernel 符号表) dumpfile`
- crash解析dump文件可以得当时的调用栈及寄存器等信息，方便问题的定位。

- 示波器，万用表，逻辑分析器等仪器
- 稳压直流电源
- 万用表
- 示波器
- 逻辑分析仪

❖ 稳压直流电源



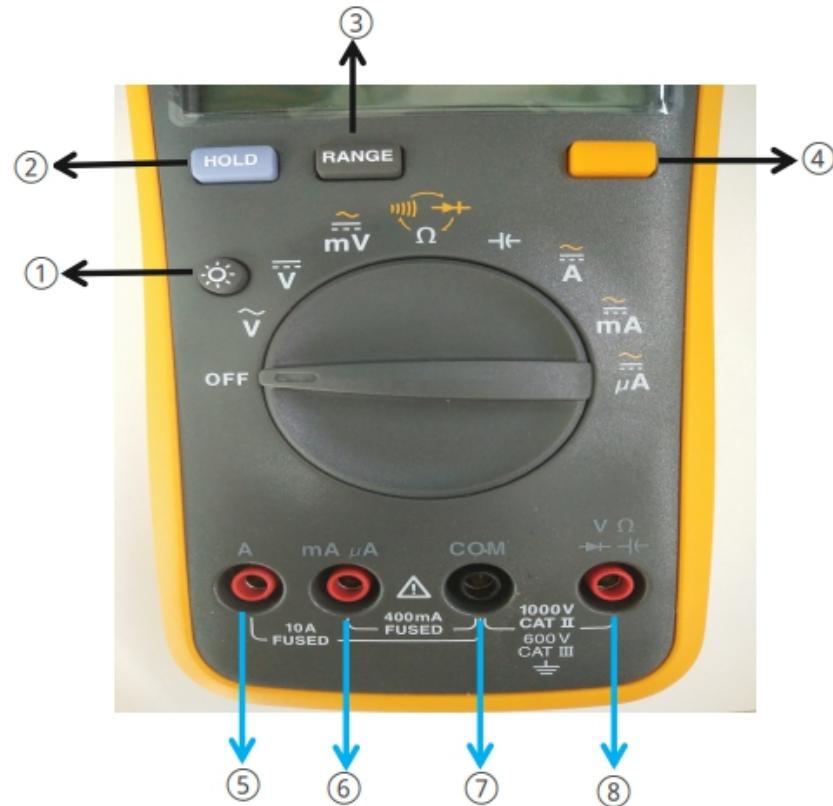
概述：

- ①开关
- ②输出接地端
- ③输出正级端
- ④稳流细调
- ⑤稳流粗调
- ⑥稳压细调
- ⑦稳压粗调
- ⑧A /mA量程转换

使用方法：

- (1)按下power按钮
- (2)调节电压，先调节⑦进行粗调，然后调节⑥进行细调（顺时针调大，逆时针调小）
- (3)调节电流，先调节⑤进行粗调，然后调节④进行细调（顺时针调大，逆时针调小，一般没有特殊要求，调到最大）
- (4)根据情况来调节⑧进行 A /mA量程转换，按下为mA量程，弹出为A 量程
- (5)连接设备，连接设备前一定要检查供电电压，防止烧毁设备

❖ 万用表



概述：

- ①用来调节显示屏幕背光亮度
- ②锁定、保持键，保持显示的测量值不变
- ③手动改变测量的量程
- ④模式切换
- ⑤用于电流的测量
- ⑥用于电流的微安以及毫安测量
- ⑦公共接线端
- ⑧用于电压、电阻、通断性、二极管、电容测量

1. 测量电压

- ① 黑线插入“COM”口， 红线插入标有电压、二极管、电阻等符号的口
- ② 这里有交流电压测量和直流电压测量， 根据实际情况， 旋转旋钮指到对应的电压测量标志。
- ③ 根据要测量的电压范围， 按“RANGE”按钮改变测量范围。
- ④ 万用表与被测电路并联。
- ⑤ 红表笔接测量端， 黑表笔接地。

2. 测量电流

- ① 根据要测量的电流范围，黑线插入“COM”口，
红线插入“A”或者“mA uA”口。
- ② 根据要测量的电流范围，旋转旋钮指到对应的
标志。
- ③ 万用表与被测电路串联。
- ④ 断开电路，红表笔应接在和电源正极相连的断
点，黑表笔应接在和电源负极相连的断点。

3. 测量电阻、通断性测试、测试二极管

- ① 黑线插入“COM”口，红线插入标有电压、二极管、电阻等符号的口。
 -
- ② 旋转旋钮指到对应的标志。
- ③ 默认是测量电阻，即右上角可以看到“MΩ”，通过按“RANGE”按钮切换量程。
- ④ 红表笔接电阻一端，黑表笔接电阻另一端。
- ⑤ 按下右上侧橘黄色按钮，会看到左上角有音量图标，右上角有“Ω”图标，表示切换到了通断性测量。一般用于测量是否短路或者导通。
 -

如测导线是否导通，红表笔连接导线一端，黑表笔连接导线另一端，如果导通会有“滴滴”声。

- ⑥ 按下右上侧橘黄色按钮，会看到右上角有个二极管图标，表示切换到了测试二极管。一般用于确认二极管正负极以及二极管是否损坏。
 -

4. 测量电容

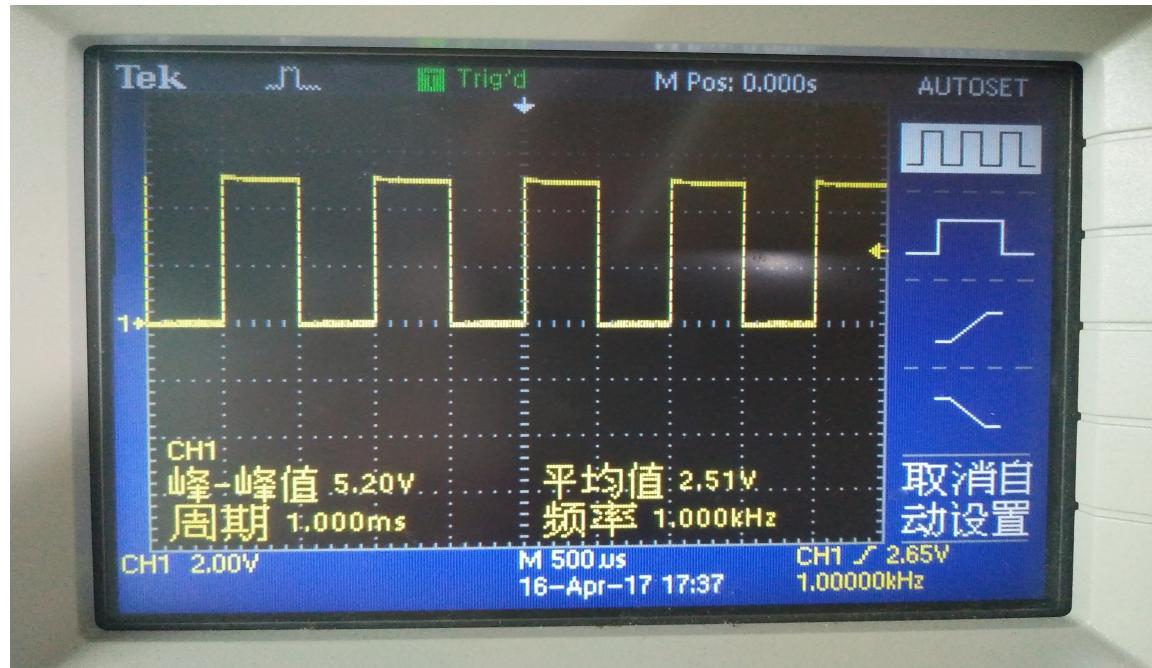
- ① 黑线插入“COM”口， 红线插入标有电压、二极管、电阻等符号的口。
- ② 旋转旋钮指到对应的标志。
- ③ 将红黑表笔分别接到电容两端， 待稳定后， 读出读数。

❖ 示波器

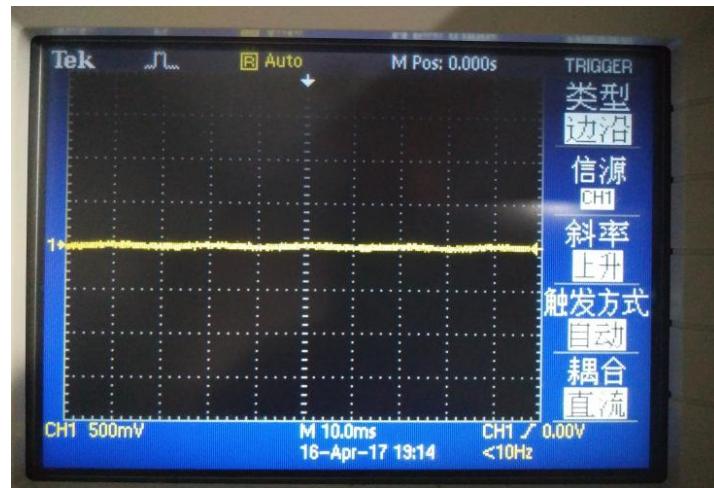


概述：

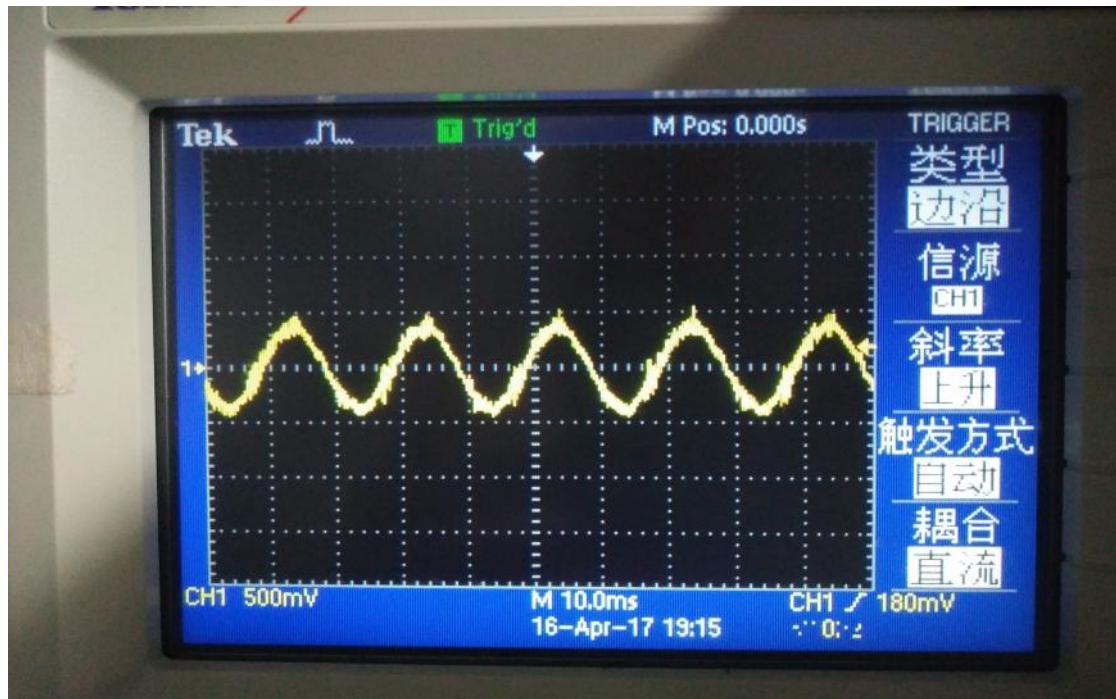
- 1.通道
- 2.自动设置



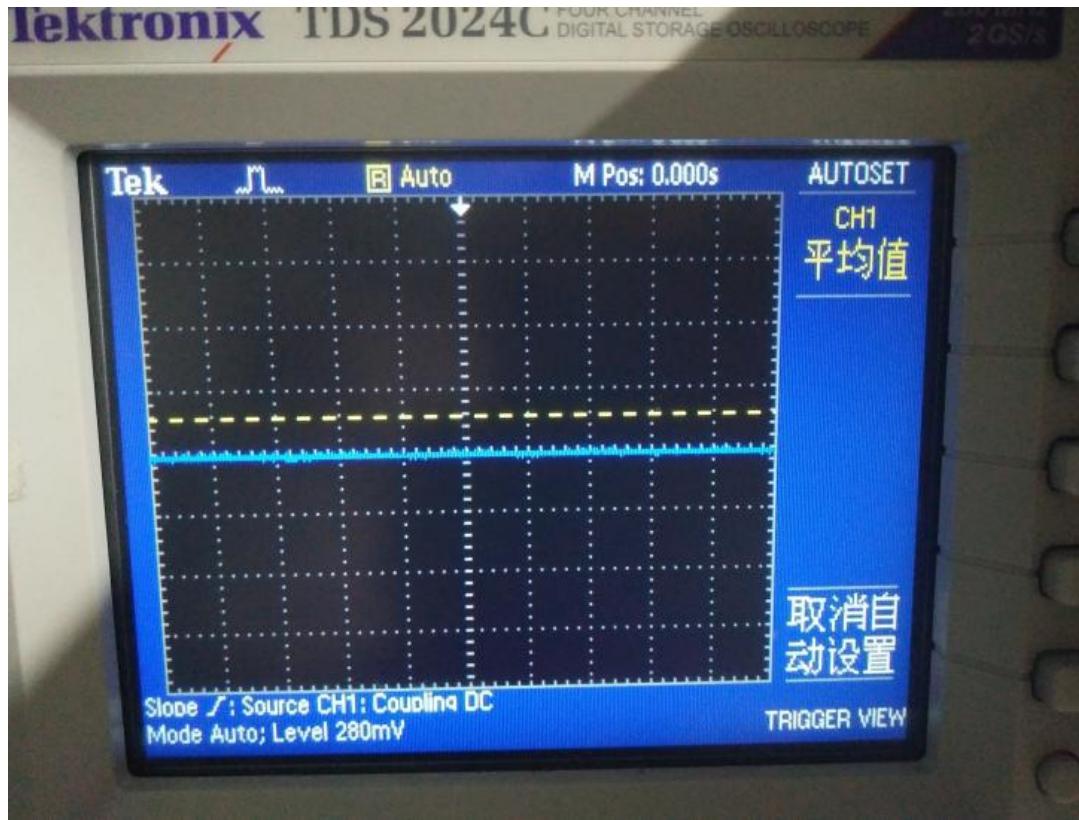
3. 默认设置, 调出示波器的出厂默认配置, 显示通道 1 的波形, 清除之前波形。
4. 运行和停止, 默认模式是“run”, 此时会在屏幕上实时输出波形, 按下一次后, 波形被锁定。
5. 单次触发, 显示第一个满足触发条件的信号波形, 并停止采集, 每次按下都会采集另一个波形。
6. 触发菜单, 用来设置触发的条件等, 如: 触发类型、触发方式等。



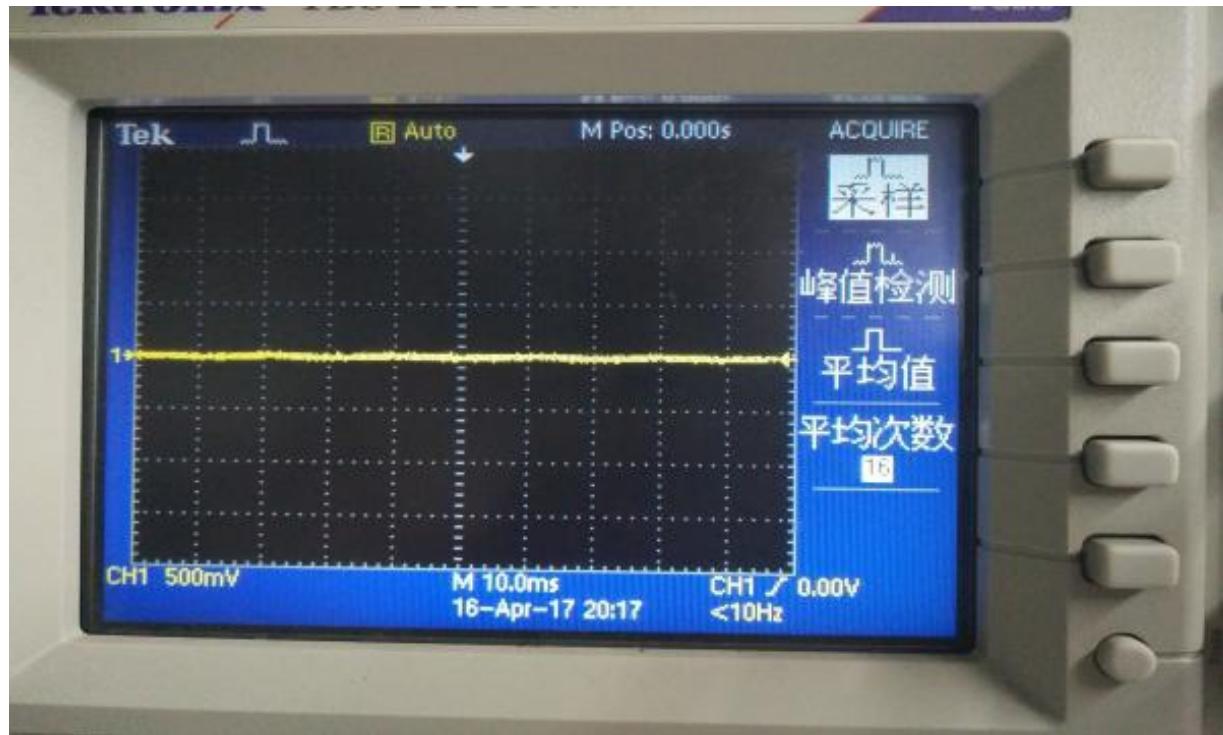
7.电平按钮，如下图右下角所示为调节此按钮产生的值，如果是上升沿触发，上升沿的值一定要大于这个值才能触发，如果是下降沿触发，下降沿的值一定要小于这个值才能触发，抓波形时要注意。



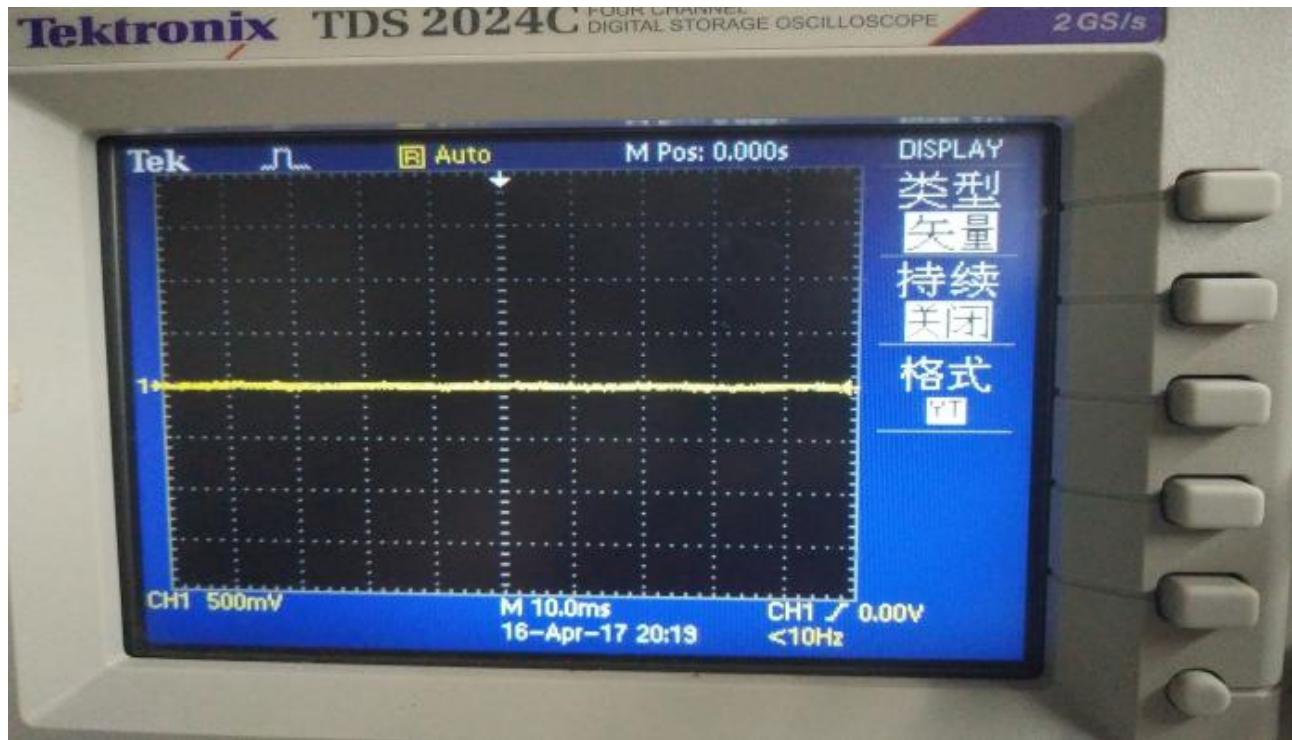
8.trig view, 查看触发设置



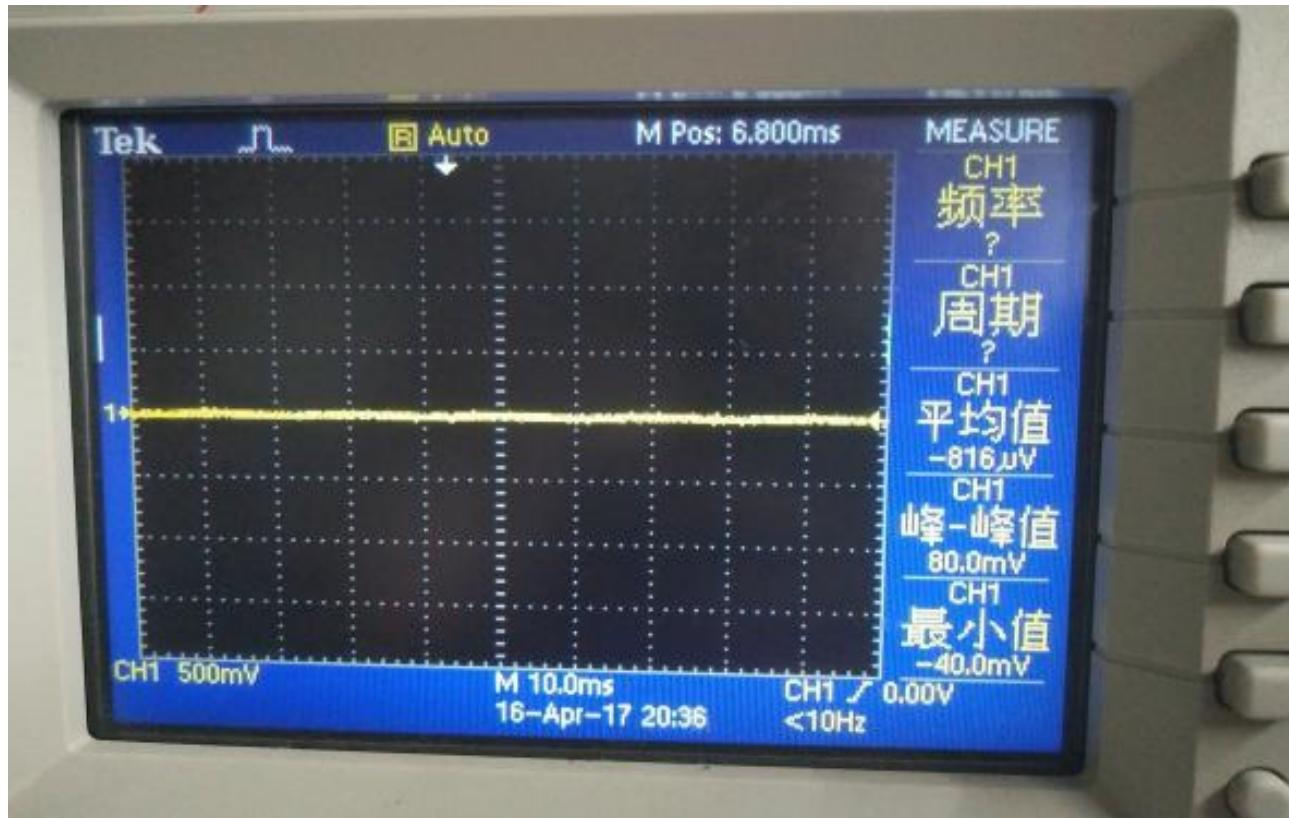
9. 强制触发，没有达到触发条件也进行触发。
10. Set to 50%，将触发电平调至示波器波形中点。
11. 获取按钮，设置采集参数，如采样、峰值检测等。



12. 显示按钮，选择波形如何出现以及如何改变整个显示的外观，选项包括类型、格式等

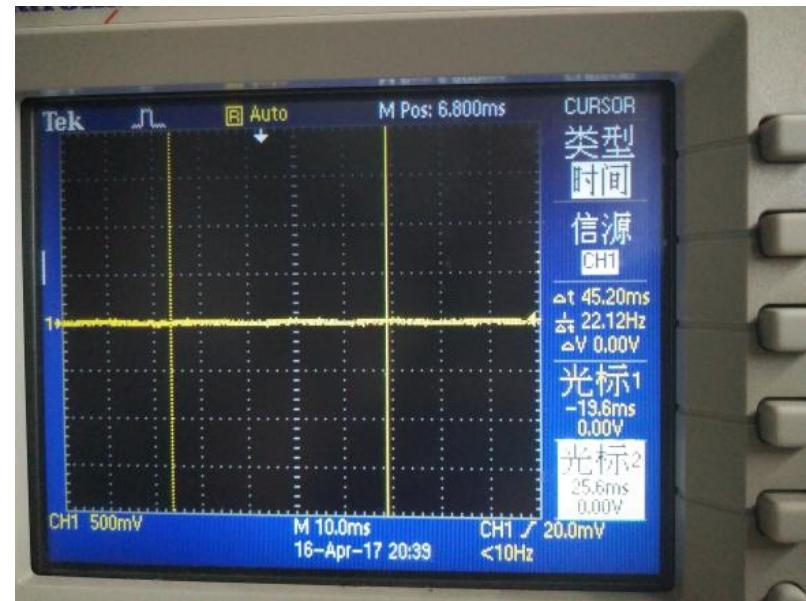
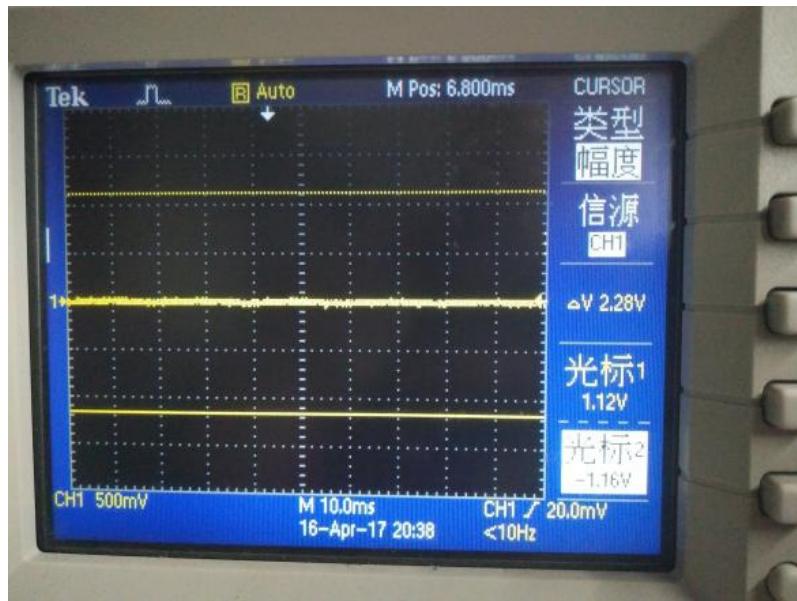


13. 测量按钮，共有16中测量类型可以选择，一次最多可以显示5个，如，频率、周期、平均值、峰—峰值、最小 / 大值、上升时间、下降时间等

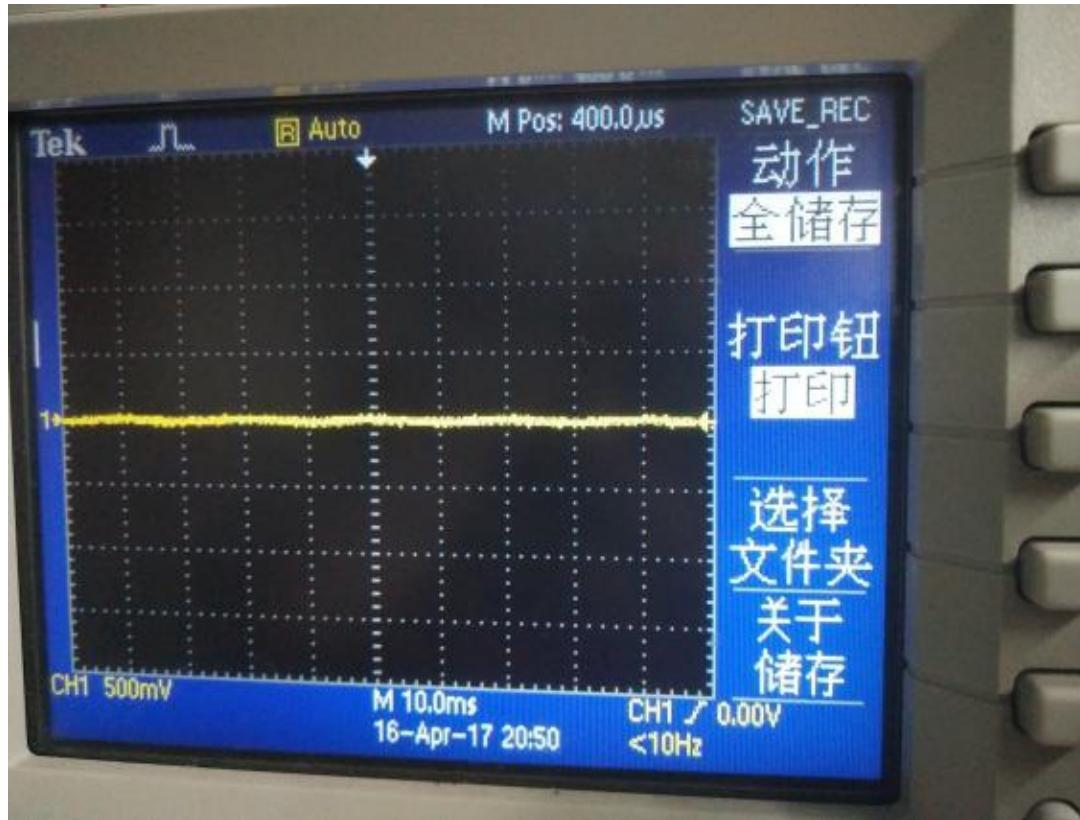


14.多用途旋钮,当旋钮处于活动状态时, 对应的LED灯会亮, 在~~其他功能~~菜单中使用

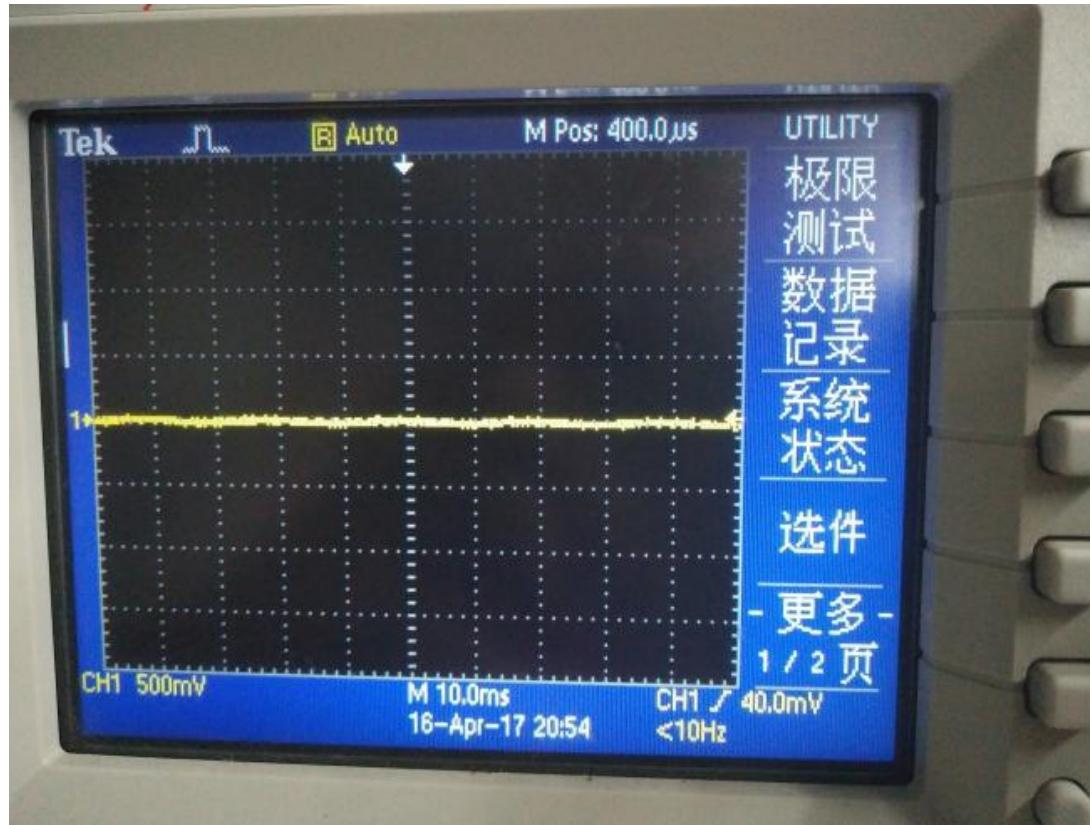
15.光标, 显示测量光标和光标菜单, 使用多用途旋钮 () 改变光标的位置, 如幅度、时间、信源等



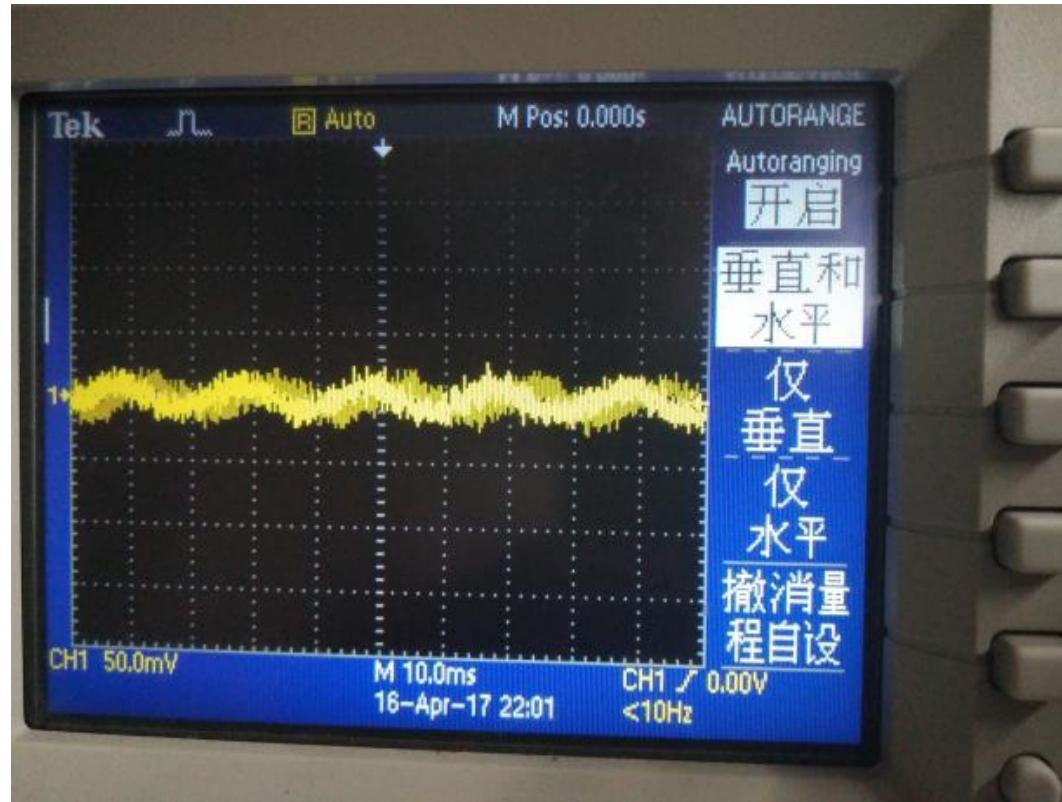
16.保存 / 调出, 储存示波器设置、屏幕图像或波形, 或者调出示波器设置或波形。如, 全部存储、存图像、存设置等



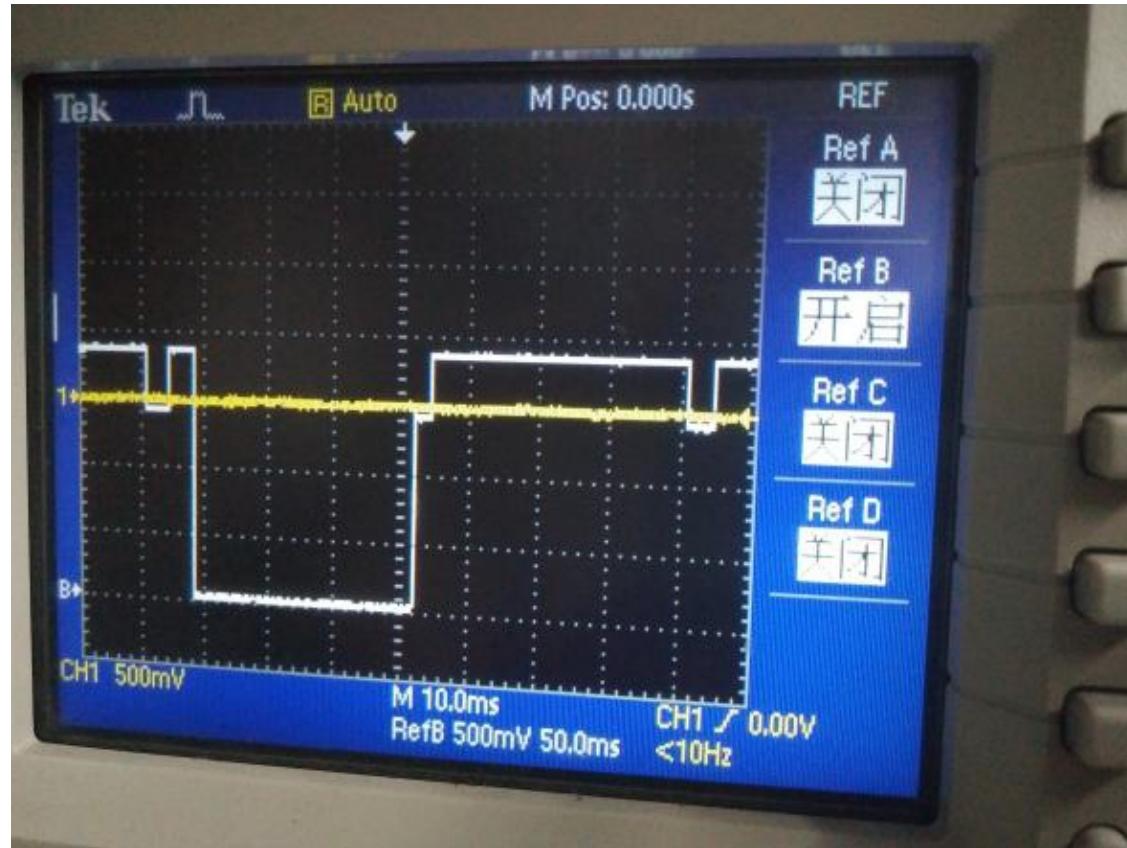
17.辅助功能，显示示波器的辅助功能，如，系统状态、选项、语言等



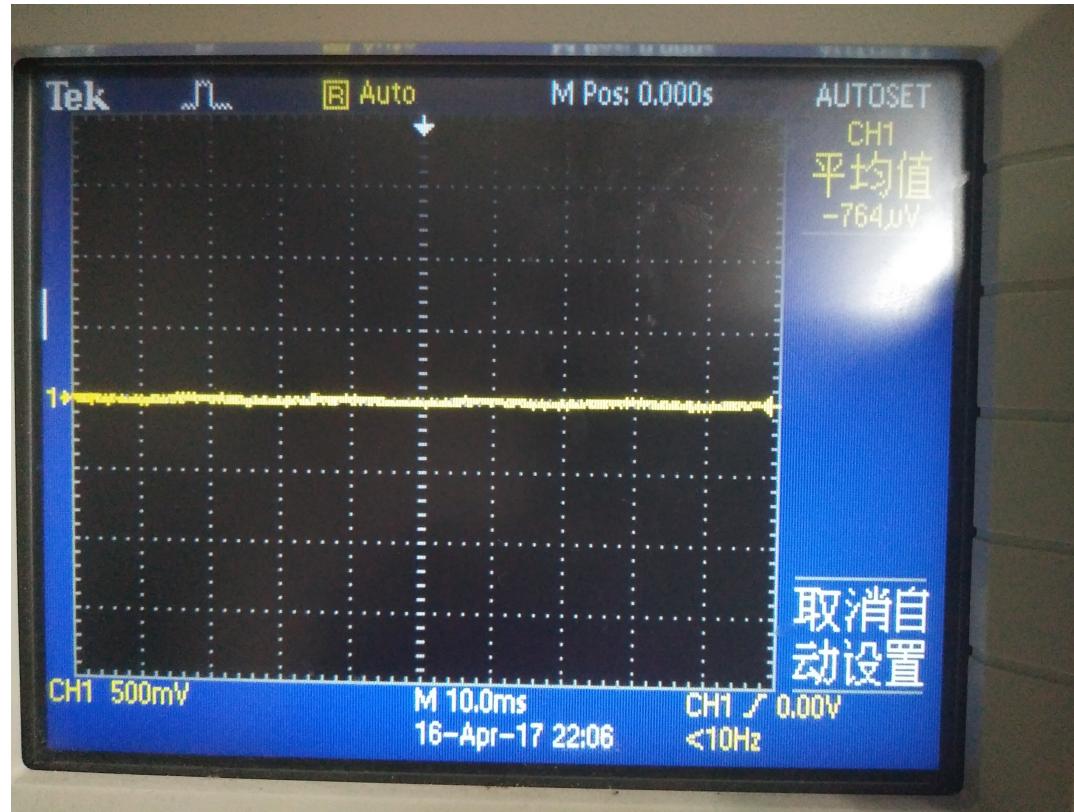
18.自动量程，显示自动量程菜单，并激活或禁用自动量程功能，激活时，对应的LED灯变亮。可以自动调整设置值来跟踪信号，如果信号发生变化，其设置将持续跟踪信号



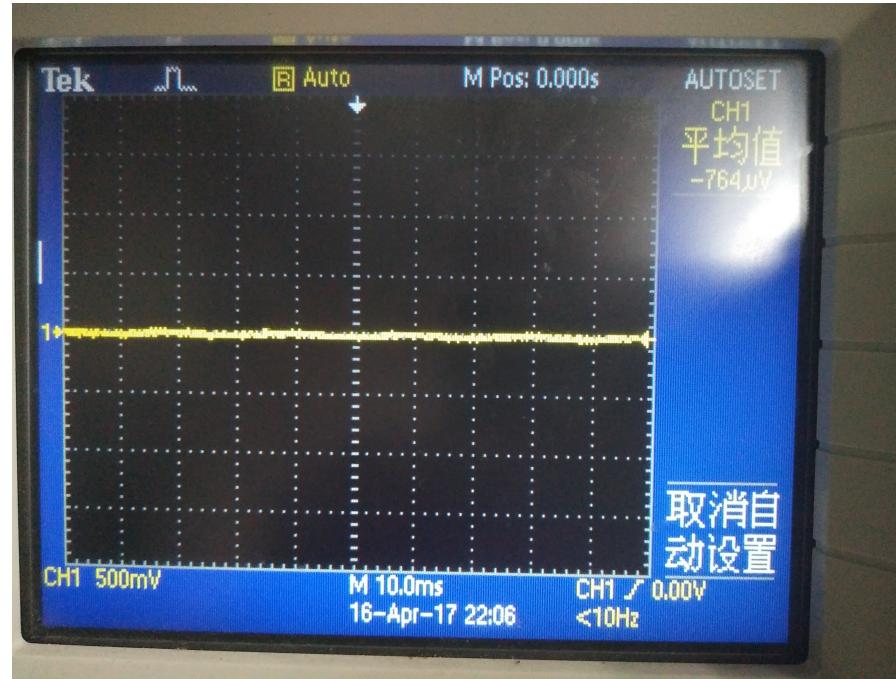
19. 参考波形，可以从显示打开或关闭参考内存波形，可以同时显示两个参考波形，但参考波形无法缩放或平移。



- 20.保存 / 打印，示波器连上打印机后，可以将图像打印出来。
- 21.伏 / 格，调节竖直方向上每格代表的电压值。



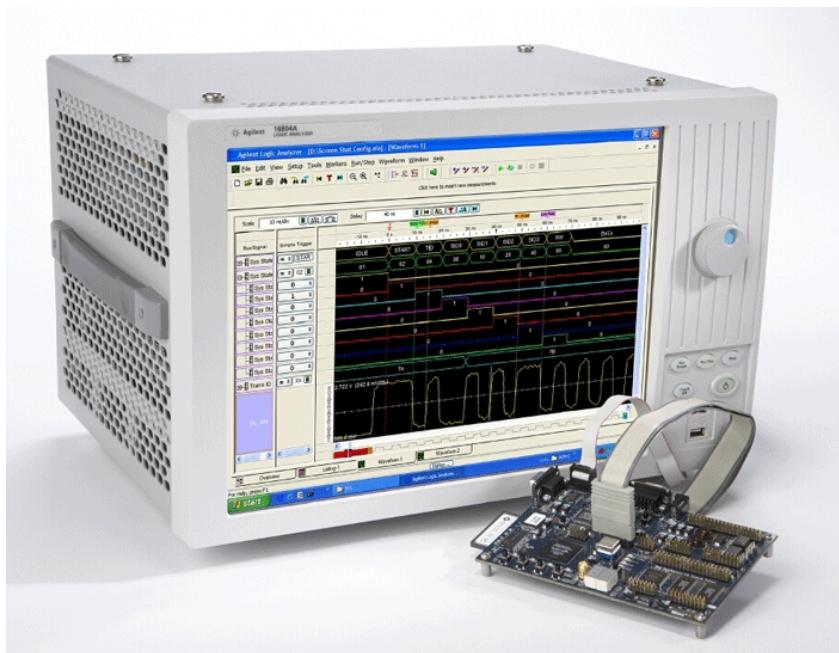
22. 垂直位置，改变波在竖直方向上的位置。
23. Set to zero，使波形回到水平位置的中心。
24. 水平位置，改变波在水平方向上的位置。
25. 秒 / 格，调节水平方向上每格代表的时间。



❖ 逻辑分析仪

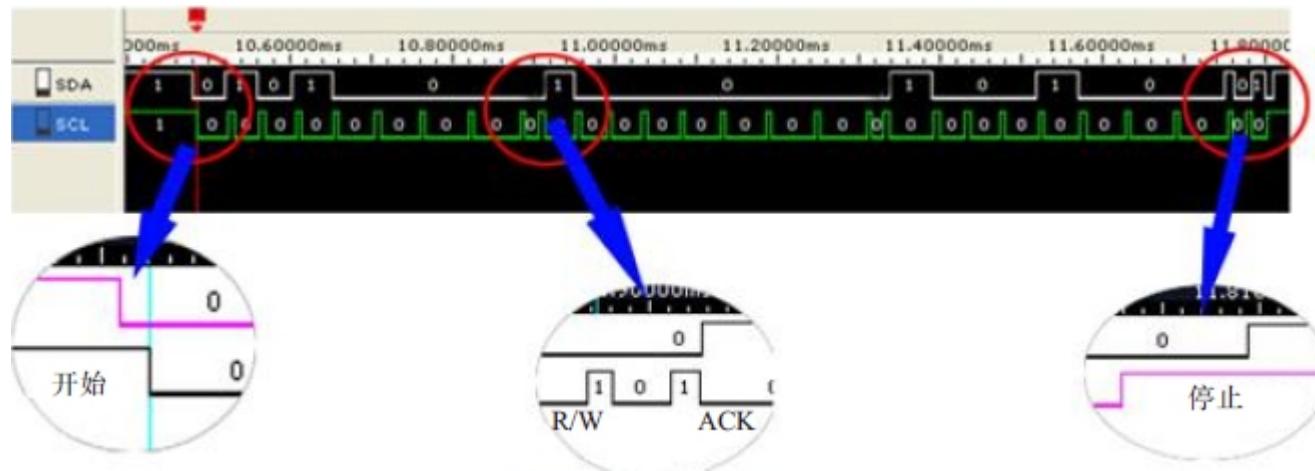
逻辑分析仪是利用时钟从测试设备上采集数字信号并进行显示的仪器，其最主要的作用是用于时序的判定。与示波器不同，逻辑分析仪并不具备许多电压等级，通常只显示两个电压（逻辑1 和0）。在设定了参考电压之后，逻辑分析仪通过比较器来判定待测试信号，高于参考电压者为1，低于参考电压者为0。

两种逻辑分析仪：



逻辑分析仪的波形可以显示地址、数据、控制信号及任意外部探头信号的变化轨迹，在使用之前应先编辑每个探头的信号名。之后，根据波形还原出总线的工作时序。目前，很多逻辑分析仪都自带了协议分析能力，可以自动分析出总线上传输的命令、地址和数据等信息。

从逻辑分析仪波形还原I2C 总线：



End!