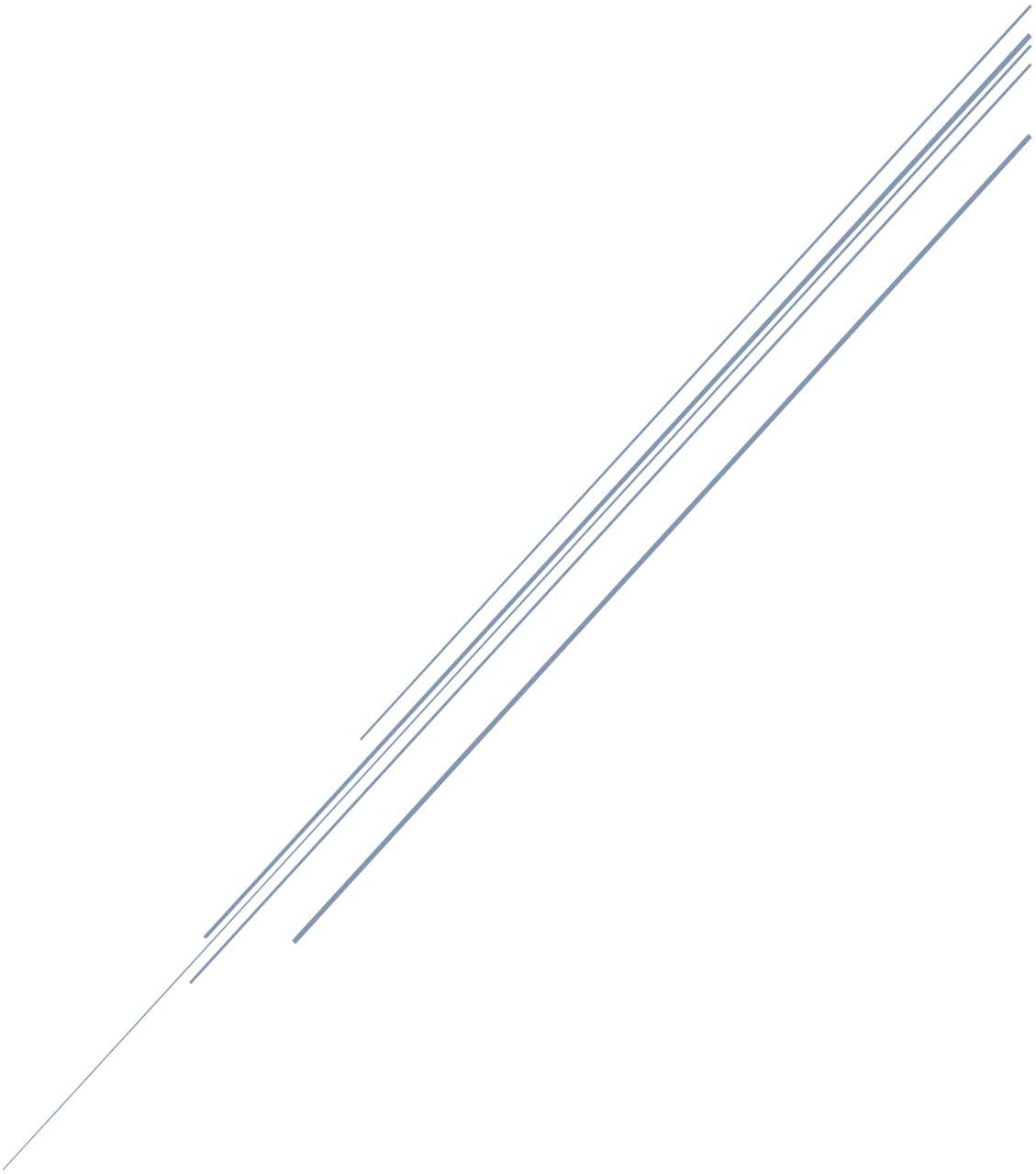
UTG4000A&UTG8000 系列信号源

编程手册



优利德科技（中国）股份有限公司

发布说明：

版本	修改项
V1.0	正式版

简介：

本文档只给出，该系列信号源的指令说明，使用的接口请查看《UCI 帮助文档》。

如果需要的指令使用不正常，请联系技术支持！

引用文件：

1、UTG4162.h ：
　　给出 UTG4000A 的基础定义。
2、UCI 相关文件：
　　见《UCI 帮助文档》。

命令字符串基本格式：

各命令都通过字符串与设备通信，基本字符串命令格式为：
{命令字段名：命令字段值…；}
其中：
‘:’之前的字符串代表命令名；
‘:’之后代表字段值；
‘;’代表一个命令的结束。
详细格式见《UCI 帮助文档》

术语： SG - 信号源简称

通用指令：

IDN?

获取设备名。
数据格式 ： 显示名%内部信息#SN 序列号
数据量是 54Bytes

比如：
UTG8000B%**#SN005

按键：

命令名	命令参数	命令参数类型
KEY	键值	见下面的编码表

KEY	字符编码	KEY	字符编码
F1	F1	0	0
F2	F2	1	1
F3	F3	2	2
F4	F4	3	3
F5	F5	4	4
F6	F6	5	5
USER	USER	6	6
Digital	dig	7	7
Counter	cnt	8	8
MOD	mod	9	9
Sweep	swp	.	.
Burst	bst	+/-	SIGN
Knob Left	FKNL	Trigger	TG

Knob Right	FKNR	CH1	C1
Knob Click	FKN	CH2	C2
CH1 X CH2	XCH	Left	L
Sine	wsn	Right	R
Noise	wns	Preset	PST
Square	wsq	Storage	STG
Ramp	wrp	StorageL	STGL
Pulse	wps	Utility	UTIL
Arb	warb	Help	HP
Harmonic	whm	HelpL	hpl
DC	wdc	Page	PG

属性名	意义	IO	数据
Lock	锁定该按键	W	无数据
Unlock	解锁该按键	W	无数据
Lock?	查询该按键锁状态	R	Integer<4Bytes>： 0 - 未锁； 1 - 已锁
Led?	查询 LED 状态	R	Integer<4Bytes>： 0 - 不亮； 1 - 亮（绿灯）；

示例：

“KEY:c1;” -- CH1

“KEY:c2;” -- CH2

“KEY:c2@lock;” -- CH2 按键锁

“KEY:c2@unlock;” -- CH2 按键解锁

“KEY:c2@lock?;” -- 查询按键是否锁

“KEY:c2@led?;” -- 查询按键 LED 状态， 与”LED;”命令类似。

注意：

参看《uci 帮助文档》的 uci_FormatWrite 的说明部分。

带问号的指令需要用 uci_Read 读取。可以通过缓冲区也可以通过接口返回值获取。

另外：

查询所有按键的锁定状态使用命令：

命令名	命令参数	数据格式
lock?	无	64 位整数，0x086FFF FFFFFFFF,表示全锁定。

使用接口 uci_Read 获取数据，调用 IsKeyLocked 接口（见 UTG4162.h）获取单独按键的状态。

本地/远程状态切换:

命令名	命令参数	命令参数类型
local	状态编码	Enum(Integer<4Bytes>):0/1{远程状态/本地状态}
Local?	无	Enum(Integer<4Bytes>):0/1{远程状态/本地状态}

示例：

“local:0;” -- 远程状态，键盘全锁。
“local:1;” -- 本地状态，键盘全解。
“local?;” -- 查询状态。

注意：

参看《uci 帮助文档》的 uci_FormatWrite 的说明部分。
带问号的指令需要用 uci_Read 读取。可以通过缓冲区也可以通过接口返回值获取。

抓取屏幕：

命令名	命令参数	命令参数类型
PrtScn	图像格式	Enum(String):空/zip/bmp/png {像素数据/压缩后的像素数据/BMP 文件/png 文件}

示例：

“PrtScn:png;” --- 截取屏幕保存为 png 文件；

“PrtScn:bmp;” --- 截取屏幕保存为 bmp 文件；

“PrtScn;” --- 截取屏幕保存为像素数据；

“PrtScn:zip;” --- 截取屏幕保存为压缩像素数据；

注意：

使用 uci_Read 读取数据，这个命令并没有将数据直接保存的文件，而是返回到 uci_Read 指定的缓冲区里面。

如果要缓冲到本地文件，请自行保存。

1、 如果使用命令： “PrtScn;”， 则缓冲区大小必须是 800 * 480 * 4 = 1536000， 读到的是 32 位的像素数据；

2、 如果使用命令： “PrtScn:bmp;”，则缓冲区大小必须是 800 * 480 * 3 + 54 = 1152054，即位图文件大小。

3、 如果使用命令： “PrtScn: png;” ，则缓冲区大小可以设置为 800 * 480 * 4 = 1536000，这个是最大值，PNG 是图像压缩数据，肯定小于该值(1536000)，通过 uci_Read 接口返回值得到有效数据长度。

4、 如果使用命令： “PrtScn:zip;”， 则缓冲区大小可设置为 800 * 480 * 4 = 1536000（最大数据量）， 读到的是压缩像素数据。然后使用：alg_UnCompressPixels 接口解压缩数据。注意 uci_Read 返回的是压缩数据量。

5、 使用 uci_ReadToFile 接口加 “prtscn:bmp;” 命令，可以将位图保存到文件。

效率稳定性对比：

“PrtScn;” 的方式效率和稳定性是最高的，如果要频繁刷新，可以考虑此接口；

“PrtScn:bmp;”的方式效率要低，因为位图文件较大。

“PrtScn: png;” 传输稳定，效率一般，因为压缩后的数据，数据量较小，但压缩本身也耗时。

写任意波形文件：

命令名	命令参数	命令参数类型
WARB	无	无

属性名	意义	IO	数据
CH	通道号	W	Enum(Integer) : 0/1{ CH1/ CH2 }
Mode	加载模式	W	Enum(Integer): 0/1/R {Carrier/Mod}
Disk	存储介质	W	Enum(Integer): 0/1/2/3{RAM/ROM/TF/U-DISK}

示例：

“WARB@CH:0@MODE:0@DISK:0;”

将波形文件以载波的形式加载到 CH1 中，存放在 RAM 中，重启后就会丢失。

注意：

使用 uci_WriteFromFile 写任意波文件。超时设置为 1000.

读写配置文件

命令名	命令参数	命令参数类型
dconfig	无	无

示例：

“dconfig;”

注意：

使用 uci_ReadToFile 读取配置文件，使用 uci_WriteFromFile 写配置文件。
配置文件的后缀是“.set”，请在读取和保存时都保证文件的后缀是.set.

版本：

命令名	命令参数	命令参数类型
Ver?;	无	无

示例：

“Ver?;”

接口：

使用 uci_Read 读取数据。

数据大小是 54 个字节。

通信协议版本：

查询系统信息版本号，用于检查协议接口是否支持。

命令名	命令参数	命令参数类型
CVer?;	无	无

示例：

“CVer?;”

接口：

使用 uci_Read 读取数据。

数据大小是 54 个字节。

数据：

```
二进制数据(54Bytes)：
31      2e      30      2e      30      00      00      00      00      00      | 1.0.0.....
00      00      00      00      00      00      00      00      00      00      | ..........
```

读参数：

命令名	命令参数	命令参数类型
rp	无	无

属性名	意义	IO	数据
CH	通道号	W	Enum(Integer) : 0/1{ CH1/ CH2 }
addr	参数地址	W	Enum(ERemoteMessage): 查看参数地址定义。

示例：

“rp@CH:0@addr:0x8009;” - 读取 CH1 的频率;

注意：

UCI 接口对应： `uci_Read`， 对应的数据量是 8 个字节，dobule 类型!

接口：

读参数使用 `uci_Read` 或者 `uci_ReadX`;

写参数：

命令名	命令参数	命令参数类型
wp	无	无

属性名	意义	IO	数据
CH	通道号	W	Enum(Integer) : 0/1{ CH1/ CH2 }
addr	参数地址	W	Enum(ERemoteMessage): 查看参数地址定义。
v	参数值	W	物理单位请以各参数定义的为准。

示例：

“wp@CH:0@addr:0x8009@v:2000;” - 设置 CH1 的频率为 2kHz;

注意：

UCI 接口对应：`uci_Write`，数据类型全部是 double 类型，8Bytes.

接口：

写参数使用 uci_Write 、 uci_WriteX 或 uci_FormatWrite ；

查询 LED 状态：

命令名	命令参数	命令参数类型
LED	无	无

示例：

“LED;”

注意：

使用 uci_Read 读取状态数据，数据格式为 LEDStatus （见文件 UTG4162.h 定义）

自动重连：

命令名	命令参数	命令参数类型
Reconnect;	无	无

示例：

“Reconnect;”

注意：

使用 *uci_SendCommand* 发送指令。

在已连接的情况下，检测的设备已断开，可以通过发送该命令尝试重新连接。

检测设备是否在线的方式是，定时查询 LED 状态，即执行“LED;”命令，通过接口返回值判断线路状态。

附录：

参数定义：

下面的定义来自文件：include\UTG4162.h

注释说明

比如 ： {IO:WR}{DATA: 0-OFF，1-ON，2-reverse}

IO 指定：参数的读写属性。 W 可写参数； R 可读参数。

Data 指定：数据的取值。

```
////////////////////////////////Parameters Address Define////////////////////////////
        //工作模式
        typedef enum _EWorkMode : long long {
            //@brief : 基波模式
            WM_BASE = 0,
            //@brief : 调制波模式
            WM_MODE,
            //@brief : 扫频模式
            WM_SWEEP,
            //@brief : 猝发
            WM_BURST,
        }EWorkMode;

        //基本波形
        typedef enum _EBaseWave : long long {
            //@brief : 正弦波
            BASE_SINE = 0,
            //@brief : 方波
            BASE_SQUARE = 1,
            //@brief : 斜波
            BASE_RAMP = 3,
            //@brief : 脉冲波
            BASE_PULSE = 4,
            //@brief : 噪声
            BASE_NOISE = 5,
            //@brief : 任意波
            BASE_ARB = 6,
            //@brief : 谐波
            BASE_HARMONIC = 7,
            //@brief : 直流
            BASE_DC = 8
        }EBaseWave;
```

```
typedef enum _EModeType : long long {
    MT_AM = 0,
    MT_FM = 1,
    MT_PM = 2,
    MT_ASK = 3,
    MT_FSK = 4,
    MT_PSK = 5,
    MT_BPSK = 6,
    MT_QPSK = 7,
    MT_OSK = 8,
    MT_QAM = 9,
    MT_PWM = 10,
    MT_SUM = 11,
}EModeType;

//调制波形
typedef enum _EModeWaveType : long long {
    MOD_WAVE_SINE = 0,
    MOD_WAVE_SQUARE = 1,
    MOD_WAVE_UPRAMP = 2,
    MOD_WAVE_DNRAMP = 3,
    MOD_WAVE_NOISE = 4,
    MOD_WAVE_ARB = 5,
}EModeWaveType;

//@brief : 信号源参数编码
//@remark: 读参数使用rp指令，写参数使用wp指令。
//@example: wp@ch:0@addr:0x8000@v:0;
//          rp@ch:0@addr:0x8000;
// 所有参数的对应的数据量是8bytes，double类型
typedef enum _enumRemoteMessage {
    //@brief : 工作模式
    //@remark: {IO:WR}{DATA:EWorkMode}
    RM_WORK_MODE = 0x8000,

    //{ 基波
    //@brief : 通道开关
    //@remark: {IO:WR}{DATA: 0-OFF，1-ON}
    RM_CH_SW = 0x8001,
    //@brief : 同步开关
    //@remark: {IO:WR}{DATA: 0-OFF，1-ON，2-reverse}
    RM_CH_SYNC_SW,
    //@brief : 通道反向
    //@remark: {IO:WR}{DATA:  0：OFF，1：ON}
```

```
RM_CH_REVERTSE,
//@brief : 通道负载阻抗
//@remark: {IO:WR}{DATA: 1~1000}
RM_CH_LOAD,
//@brief : 通道输出限制使能
//@remark: {IO:WR}{DATA:   0：OFF，1：ON}
RM_CH_OUTPUT_LIMIT_ENABLE,
//@brief : 通道输出限制最低电平
//@remark: {IO:WR}{DATA<?>:   -10V~MAX_LEVEL}
RM_CH_OUTPUT_LIMIT_MIN_LEVEL,
//@brief : 通道输出限制最高电平
//@remark: {IO:WR}{DATA:   MIN_LEVEL~10V}
RM_CH_OUTPUT_LIMIT_MAX_LEVEL,

//@brief : 基波类型
//@remark: {IO:WR}{DATA:EBaseWave}
RM_BASE_WAVE_TYPE = 0x8008,
//@brief : 基波频率(单位：Hz)
//@remark: {IO:WR}{DATA:1uHz~但前基波最大频率}
RM_BASE_FREQ,
//@brief : 基波相位(单位：°)
//@remark: {IO:WR}{DATA:-360~360}
RM_BASE_PHASE = 0x800A,
//@brief : 基波幅度(单位：VPP)
//@remark: {IO:WR}{DATA:1mVpp~10Vpp（50欧姆）}
RM_BASE_AMP_VPP,
//@brief : 基波幅度(单位：VRMS)
//@remark: {IO:WR}{DATA:1mVRMS~5Vpp（50欧姆）}
RM_BASE_AMP_VRMS,
//@brief : 基波幅度(单位：VDBM)
//@remark: {IO:WR}{DATA:-53.010VDBM~26.99VDBM（50欧姆）}
RM_BASE_AMP_VDBM,
//@brief : 基波偏移(单位：V)
//@remark: {IO:WR}{DATA:-5VRMS~5Vpp（50欧姆）}
RM_BASE_OFFSET,
//@brief : 基波高电平(单位：V)
//@remark: {IO:WR}{DATA:BASE_LOW~5Vpp（50欧姆）}
RM_BASE_HIGHT = 0x800F,
//@brief : 基波低电平(单位：V)
//@remark: {IO:WR}{DATA:-5VRMS~BASE_HIGHT（50欧姆）}
RM_BASE_LOW = 0x8010,
//@brief : 基波占空比对方波占空比
//@remark: {IO:WR}{DATA:0~100}
RM_BASE_DUTY,
```

```
//@brief : 脉冲波上升时间(单位：s)
//@remark: {IO:WR}{DATA:min ~ 周期*0.4}
RM_BASE_RISETIME,
//@brief : 脉冲波下降时间(单位：s)
//@remark: {IO:WR}{DATA:min ~ 周期*0.4}
RM_BASE_FALLTIME,
//@brief : 任意波播放模式开关
//@remark: {IO:WR}{DATA:0：OFF，1：ON}
RM_BASE_ARB_PLAY_ENABLE,
//@brief : 斜波对称度
//@remark: {IO:WR}{DATA:0~100}
RM_BASE_SYMMETRY,,

//@brief : 谐波 - 工作模式
//@remark: {IO:WR}
//{DATA:
//0:    奇次,
//1:    偶次,
//2:    全部,
//3:    USER,自定义
// }
RM_BASE_HARMOIC_TYPE = 0x8080,
//@brief :谐波开关,在RM_BASE_HARMOIC_TYPE=USER时有效
//@remark: {IO:WR}{DATA:BIT14至BIT0分别对应2~16次谐波开关，BIT15对应基波强制开启}
RM_BASE_HARMONIC_ONOFF,
//设置N(2~16)次谐波,可以可以有两种方法:
//EG:N=3(三次谐波),AMP = 1Vpp, PHASE = 90°
//方法1:
//（1）、指定谐波次数，RM_HARMONIC_NUM = N(2~16),
//（2）、设定谐波的幅度和相位,
//      RM_HARMONIC_SN_AMP_N = 1.0; RM_HARMONIC_SN_PHASE_N = 90.0;
//方法1:直接设置第三次谐波的幅度和相位
//RM_HARMONIC_SN_AMP_3 = 1.0, RM_HARMONIC_SN_PHASE_3 = 90.0,

//@brief : 谐波次数
//@remark: {IO:WR}{DATA:1~15}
RM_HARMONIC_NUM,
//@brief : 谐波幅度Vpp
//@remark: {IO:WR}{DATA:0~基波幅度}
RM_HARMONIC_SN_AMP_N,
//@brief : 谐波相位(单位：°)
//@remark: {IO:WR}{DATA:-360~360}
RM_HARMONIC_SN_PHASE_N = 0x8084,
//@brief : 谐波幅度 同RM_HARMONIC_SN_AMP_N
```

```
RM_HARMONIC_SN_AMP_2,
RM_HARMONIC_SN_AMP_3,
RM_HARMONIC_SN_AMP_4,
RM_HARMONIC_SN_AMP_5,
RM_HARMONIC_SN_AMP_6,
RM_HARMONIC_SN_AMP_7 = 0x808A ,
RM_HARMONIC_SN_AMP_8,
RM_HARMONIC_SN_AMP_9,
RM_HARMONIC_SN_AMP_10,
RM_HARMONIC_SN_AMP_11,
RM_HARMONIC_SN_AMP_12 = 0x808F,
RM_HARMONIC_SN_AMP_13 = 0x8090,
RM_HARMONIC_SN_AMP_14,
RM_HARMONIC_SN_AMP_15,
RM_HARMONIC_SN_AMP_16,
//@brief : 谐波相位 RM_HARMONIC_SN_PHASE_N
RM_HARMONIC_SN_PHASE_2,
RM_HARMONIC_SN_PHASE_3,
RM_HARMONIC_SN_PHASE_4,
RM_HARMONIC_SN_PHASE_5,
RM_HARMONIC_SN_PHASE_6,
RM_HARMONIC_SN_PHASE_7,
RM_HARMONIC_SN_PHASE_8 = 0x809A,
RM_HARMONIC_SN_PHASE_9,
RM_HARMONIC_SN_PHASE_10,
RM_HARMONIC_SN_PHASE_11,
RM_HARMONIC_SN_PHASE_12,
RM_HARMONIC_SN_PHASE_13 = 0x809F,
RM_HARMONIC_SN_PHASE_14 = 0x80A0,
RM_HARMONIC_SN_PHASE_15,
RM_HARMONIC_SN_PHASE_16,
//}

//{MOD
//@brief : 调制模式
//@remark: {IO:WR}{DATA:EModeType}
RM_MOD_TYPE = 0x8100,
//调制波形
//0:  MOD_SINE,
//1:  MOD_SQUARE,
//2:  MOD_UPRAMP,
//3:  MOD_DNRAMP,
//4:  MOD_NOISE,
//5:  MOD_ARB,
```

```
//@brief : 调制波形
//@remark: {IO:WR}{DATA:EModeWaveType}
RM_MOD_WAVE,
//@brief : 调制波频率（单位：Hz）
//@remark: {IO:WR}{DATA: 1uHz~200KHz}
RM_MOD_FREQ,
//@brief : 调制波速率（单位：s）
//@remark: {IO:WR}{DATA<double>: 2ms~1Ms}
RM_MOD_RATE,
//@brief : 调制深度（单位：%）
//@remark: {IO:WR}{DATA: 0~120}
RM_MOD_SCOPE,
//@brief : 调制源
//@remark: {IO:WR}{DATA: 0-Internal,1-External}
RM_MOD_SOURCE,
//@brief : FM频差（单位：Hz）
//@remark: {IO:WR}{DATA: 0~载波但前频率}
RM_MOD_FRE_DEV,
//@brief : PM相差    (单位：°)
//@remark: {IO:WR}{DATA:-360~360}
RM_MOD_PHASE_DEV,
//@brief : FSK跳频（单位：Hz）
//@remark: {IO:WR}{DATA: 0~载波最大频率}
RM_MOD_HOP_FREQ,
//@brief : BPSK调制数据源
//@remark: {IO:WR}
//{DATA<double>:
//0:  PN7,
//1:  PN9,
//2:  PN15,
//3:  PN21,
//}
RM_MOD_DATA_SOURCE = 0x8109,
//@brief : BPSK相位、QPSK相位1,(单位：°)
//@remark: {IO:WR}{DATA:-360~360}}
RM_MOD_PSK_PHASE1,
//@brief : QPSK相位2(单位：°)
//@remark: {IO:WR}{DATA:-360~360}}
RM_MOD_PSK_PHASE2,
//@brief : QPSK相位3(单位：°)
//@remark: {IO:WR}{DATA:-360~360}
RM_MOD_PSK_PHASE3,
//@brief : OSK震荡时间(单位：s)
//@remark: {IO:WR}{DATA: 8ns~200s}
```

```
RM_MOD_OSC_TIME,
//@brief : QAM调制IQ MAM
//@remark: {IO:WR}{DATA: 0-4QAM,1-8QAM,2-16QAM,3-32QAM,4-64QAM,5-128QAM,6-256QAM,}
RM_MOD_IQ_MAP,
//@brief : PWM调制占空比差值
//@remark: {IO:WR}{DATA: 0~PULS DUTY}
RM_MOD_DUTY_DEV,
//}

//{SWEEP
//@brief : 扫频类型
//@remark: {IO:WR}{DATA: 0：线性，1：对数}
RM_SWEEP_TYPE = 0x8200,
//@brief : 扫频触发源
//@remark: {IO:WR}{DATA: 0：内部，1：外部，2：手动}
RM_SWEEP_SOURCE,
//@brief : 扫频时间(单位：s)
//@remark: {IO:WR}{DATA: 1ms~500s}
RM_SWEEP_TIME,
//@brief : 扫频起始频率（单位：Hz）
//@remark: {IO:WR}{DATA: 1uHz~最大载波频率}
RM_SWEEP_START_FREQ,
//@brief : 扫频停止频率（单位：Hz）
//@remark: {IO:WR}{DATA: 1uHz~最大载波频率}
RM_SWEEP_STOP_FREQ,
//@brief : 同步输出触发频率（单位：Hz）
//@remark: {IO:WR}{DATA: RM_SWEEP_START_FREQ~RM_SWEEP_STOP_FREQ}
RM_SWEEP_SYNC_FREQ,
//@brief : 扫频触发输出
//@remark: {IO:WR}{DATA: 0-OFF，1-ON}
RM_SWEEP_TIRG_OUT,
//}

//{BURST
//@brief : 猝发类型
//@remark: {IO:WR}{DATA:
//0:  N周期,
//1:  无限,
//2:  门控
//}
RM_BURST_TYPE = 0x8300,
//@brief : 猝发触发源
//@remark: {IO:WR}{DATA: 0：内部，1：外部，2：手动}
RM_BURST_SOURCE,
```

```
    //@brief : 触发输出
    //@remark: {IO:WR}{DATA: 0-OFF，1-ON}
    RM_BURST_TIRG_OUT,
    //@brief : 猝发周期（单位：s）
    //@remark: {IO:WR}{DATA: 1~500}
    RM_BURST_PERIOD,
    //@brief : 猝发相位(单位：°)
    //@remark: {IO:WR}{DATA:-360~360}
    RM_BURST_PHASE,
    //@brief : 猝发周期数（单位：个）
    //@remark: {IO:WR}{DATA: 1~50000}
    RM_BURST_CYCLES,
    //@brief : 触发边沿
    //@remark: {IO:WR}{DATA: 0-Rise，1-Fall}
    RM_BURST_TIRG_EDGE
    //}
}ERemoteMessage;
```