Index of Internal (only!) Markdown links

Timebase Notes

1. Asynchronous meter polling

gp.scanON enables/disables polling of the PZEM power meter. **gp.scanT** defines how often to obtain a measurement. Typically 2sec per phase. **gp.scanON** and **gp.scanT** are accessible through **CLI !set**

OLED is always updated when power meter is polled; even if **gp.scanON** is false and Meguno issues a READ

Every polling cycle calls `eng.readPhase( 1 or 2 )` After Phase 2, the function `eng.getPhase( 1+2 )` is used to compute the sum of phase1 and phase2. The global variable **phase** counts 1 or 2 to indicate phase 1 or 2. The above readings are stored in global parameters `vap[1]`, `vap[2]` and `vap[0]` (for phase 1+2)

All CLI functions and all Thinger.io functions must use the `vap[1]`, `vap[2]` and `vap[0]` for respective readings; the **phase** can be used to determine the current sampling statge.

Thinger.io real time display is updated after reading phase #2. The variable **streamON** is used to enable streaming to Thinger.io or not. The **streamON** variable is volatile, not persistent in EEPROM. It is controlled by ThingerIO button.

2. Statistics

Statistics are computed with Phase 2 including:

```
Vmin
Vave
Wmax
Wave
```

Voltage and Power averages are computed on a sample-by-sample basis. Two level of statistics are computed: **ShortT** and **LongT**. The integration/reset period is determined by Thinger.io buckets **"Hourly"** and **"Daily"**. More specifically:

- when the **"Hourly"** bucket requests to read the pson **[MinuteStats]**, the **shortT** statistics are reset.
- when the **"Daily"** bucket requests to read the pson **[HourlyStats]** these **longT** statistics are reset.

This way, the integration period of the statistics are fully determined by the Thinger.io frequency of retrieving data from the physical device.

- **Vmin, Wmax Computations**

    - **Vmin** is computed as the minimum of either V1 or V2, i.e. min(V1, V2, Vmin).
    - **Wmax** is computed as max of the sum of W1 and W2

  All above are computed on a sample-by-sample basis.

- **Vavg, Wavg Computations**
    - Vavg is computed by accumulating the V0 values until it is reset.
    - Wavg is computed by accumulating the W0 values until it is reset

  All above are computed on a sample-by-sample basis

The functions below are to be used by CLI or Thinger.io to view statistics at any point of time:

```
    shortT OR longT .getVmin()
                    .getVavg()
                    .getWavg()
                    .getWmax()
    and             eng.getkWh()
```

Thinger.io "Hourly" buckets collects **Hourly statistics** using the **[HourlyStats]** pson. Similarly, the "Daily" bucket uses the **[DailyStats]** pson.

3. Under-voltage and Over-power Alarms

Upon completion of Phase1 and Phase2, the following checks take place:

- if V1 or V2 is below **Vthreshold** for two-sampling intervals, then **Valarm** is set. This flag remains set while this alarm conditions continues; it is reset only after both V1 and V2 exceed **Vthreshold**
- if W1+W2 (i.e. W0) is above **Wthreshold** for two sampling intervals: **Walarm** is set. This flag remains set while alarm conditions continues; it is reset only after W0 gets below **Wthreshold**

In addition to the flags, the time of the alarm is noted, along with the duration of the alarm in seconds using the following pson variables "**UValarm**" and "**OPalarm**" as follows:

```
when V alarm is set: "UV T0=00:00:00" (exact time it happened)
when V alarm is over: "UV dT=000 sec" (duration of the alarm)

when W alarm is set: "OP T0=00:00:00" (exact time it happened)
when W alarm is over: "OP dT=000 sec" (duration of the alarm)
```

4. Meguno Updates

The variable **gp.display** is used to control the verbosity of Meguno with the following values:

```
0 = when SCAN is ON, CLI and Meguno updates are suspended
1 = when SCAN is ON, CLI text is updated but not Meguno
2 = when SCAN is ON, CLI text and all meguno areas are updated

if SCAN is OFF, all CLI and Meguno areas are updated
```

In case of Alarms, the CLI text is enabled regardless of the **gp.display** value

5. OLED Display

Usual screen during boot. Display screen during operation:

```
0   \a         Ph#1 Err:n
1
2   \a      V1 or V2 A1 or A2
3   \a      UV alarm, OP alarm
4+5 \b           W1+W2
7   a\a           kWh
```

Thinger.io CLI Implementation

Code snipper in `setupThinger()`

```
static char exrsp[1000];        // Buffer used for Thinger CLI
                                // Must be static
BUF exbuf( exrsp, 1000 );       // BUF wrapper of exstr[]

thing["CliCmd"] << [](pson &in)
{
    const char *p = (const char *)in["extCmd"];
    PF("Received cmd [%s]\r\n", p );
    exe.dispatchBuf( p, exbuf );
};
thing["CliRsp"] >> [](pson& ot )
{
    PFN( "Sending response [%s]", exrsp );
    ot["value1"] = (const char *)exrsp;
};
```

Synchronizing Thinger.io and MegunoLink

Code snipper in `setupThinger()`

```
thing["scanON"] << [](pson &inp)
{
    if( inp.is_empty() )
        inp= myp.gp.scanON;         // get state of scanON
    else
        setScan( (bool) inp );
};
```

The `setScan()` function must set the global variable `myp.gp.scanON` and also update the MegunoLink as follows:

```
void setScan( bool value )
{
    myp.gp.scanON = value;                  // save value to structure
    mgn.init( channel );
    mgn.controlSetCheck( "cScan", value );  // update check box
    saveMyParm( "scanON" );                 // update table and EEPROM
}
```

The generic saveMyParm() function is as follows:

```
void saveMyParm( char *parm )
{
    int ok = nmp.getParmType( parm );               // 0 if parm not found
    if( ok )
    {
        myp.saveMyEEParms();                        // save to EEPROM
        nmp.printMgnParm( channel, parm );          // update the table
    }
    nmp.printMgnInfo( channel, parm, (char *) (ok ? "updated": "is unknown") );

}
```

As MegunoLink updates the cScan checkbox, to ensure that Thinger.io is also updated, use the following code in the streaming portion of the main loop:

```
void thingStream()
{
    thing.stream( thing["reading"] );
    thing.stream( thing["scanON"] );    // this actually triggers the ["scanON"]
pson
    .........
}
```

Frequently used CLI in Thinger.io

To change the gain of the current sensor(s):

```
!set ampscale 1.0               change the scale of the Ameter
setTime <hrs> <min> <sec>       Sets the PowerMeter Time
scanT <sec>                     Sets how often to scan/stream. Typically <sec>=2
```

File Dependencies

```
THIS...         Uses...         allocates

myGlobals       nmpClass        Global myp
                                NMP nmp

mySupport       myGlobals       CPU cpu
                                OLED oled
                                PZMEng eng;
                                PZMStats aveS
                                PZMStats aveL
                                PZ16 pz( cpu, oled )

myCliHandlers   above           MGN mgn
                mgnClass
```

```
main            above           Thinger
                                CLI cli
                                EXE exe
                                EEP eep
```

*/

```
//  #define INC_THINGER           // select this to enable THINGER interface
  #undef  INC_THINGER
//  #define INC_OLED
  #undef  INC_OLED
```

```
#define OLED_DISPLAY_PERIOD  2                          // seconds.
#define METER_READING_PERIOD 2                          // seconds. How often to
read the meter
#define ALERT_REPEAT_PERIOD (60/METER_READING_PERIOD )  // seconds. How long to
wait for IFTTT-NOTIFYrepeats

const char *key = "hDvZv76KwaTbEtTK6aGPBow5hVWrXp6hYx5Vu4v9vgU";
```

```
// ----------------------------------------------------------------------------
  #include <FS.h>
  #include <bufClass.h>      // in GKE-L1
  #include <ticClass.h>      // in GKE-L1
  #include <oledClass.h>     // in GKE-L1

//  #include "SimpleSRV.h"     // in GKE-Lw
  #include "SimpleSTA.h"     // in GKE-Lw
  #include "CommonCLI.h"     // in GKE-Lw
```

```
#include "myGlobals.h"      // in this project. This includes externIO.h
#include "myCliHandlers.h"
#include "mySupport.h"      // this also includes pz16Class.h
```

```
//------------------ References and Class Allocations ------------------------
```

```
CLI cli;
EXE exe;
EEP eep;
BUF buffer( 1024 );         // buffer to hold the response from the local or
remote CLI commands.
```

```
// ---------------------------------- THING SPCIFIC ---------------------------------
```

```
#define USERNAME "GeoKon"
#define DEVICE_ID "PZEM004"
#define DEVICE_CREDENTIAL "success"
```

```
//  #define DEBUG
   #define DISABLE_TLS          // very important for the Thinger.io
```

```
#include <ThingerESP8266.h> // always included even if not #ifdef

ThingerESP8266 thing(USERNAME, DEVICE_ID, DEVICE_CREDENTIAL);   // changed in
the setupThinger();
```

```
// ---------------------------------- MY GLOBAL VARIABLES --------------------------
```

```
// ------------------------------------ MAIN SETUP ---------------------------------
```

```
#include <Ticker.h>          // needed to RESET statistics
Ticker tk;
```

```
void setup()
{
   int runcount = setjmp( myp.env );          // env is allocated in myGlobals
   cpu.init( 9600 );
   ASSERT( SPIFFS.begin() );                  // start the filesystem;
```

```
pinMode( 12, OUTPUT );                        // GPIO12 is D4. Used to OE the
tranlator
digitalWrite( 12, LOW);

oled.dsp( O_LED130 );                         // initialize OLED
pz.setOLED( NULL );                           // disable OLED in PZ16
oled.dsp( 0, "Started OK" );
myp.initAllParms();                           // initialize volatile & user EEPROM
parameters

oled.dsp( 1, "Read %d parms", nmp.getParmCount() );

linkParms2cmnTable( &myp );
exe.initTables();                             // clear all tables
exe.registerTable( cmnTable );                // register common CLI tables
exe.registerTable( mypTable );                // register tables to CLI

oled.dsp( 2, "Waiting for CLI" );
startCLIAfter( 5/*sec*/, &buffer );           // this also initializes cli(). See
SimpleSTA.cpp

oled.dsp( 3, "Waiting for WiFi" );
setupWiFi();                                  // use the EEP to start the WiFi

oled.dsp( 4, "%s", WiFi.SSID().c_str() );
oled.dsp( 5, "RSSI:%ddBm", WiFi.RSSI() );
oled.dsp( 6, "IP=%d", WiFi.localIP()[3] );

#ifdef INC_THINGER
setupThinger();                               // if thinger is not included, it
does nothing
#endif
```

```
  //eng.resetDailyEnergy();

  //void setDailyReset();
  //setDailyReset();                              // Start Ticker; used to reset
  startkWh daily

  oled.dsp( 7, "All OK!" );
  delay( 1000 );
  oled.dsp( O_CLEAR );

  tmb.setHMS( 9, 50, 0 );
  ticS.setSecCount( tmb.getSecCount() );
  ticL.setSecCount( tmb.getSecCount() );
  tk.attach_ms( 1000, [](){ tmb.update(), ticS.update(); ticL.update(); } );

  cli.prompt();
```

}
// ---------- MAIN LOOP ----------

```
TICsec readMeterTic( METER_READING_PERIOD );          // how often to read the meter
TICsec displayOledTic( OLED_DISPLAY_PERIOD );         // how often to update the display
TICsec consoleMgnTic( 2 );

void notifyIFTTT_V();                         // see below
void notifyIFTTT_W();
void sendtoDropbox();

static bool process_withWiFi = false;   // this flag is set by stdLoop()to indicate to
                                // main loop() that WiFi processing is necessary

void stdLoop()                    // This is always executed regardless if WiFi connected or not
{
  if( cli.ready() )               // handle serial interactions
  {
    char *p = cli.gets();
    exe.dispatchBuf( p, buffer );   // required by Meguno
    buffer.print();
    cli.prompt();
  }
  if( myp.gp.scanON && readMeterTic.ready() ) // execute this every N-seconds
  {
    if( myp.simulON )
      myp.vap = eng.getSimul();
    else
      myp.vap = eng.readPhase12();       // read both phase 1 and 2, save results

//      aveS.updateStats( &myp.vap );         // update short term stats (used in buckets)
//      aveL.updateStats( &myp.vap );         // update long term stats (used by dropbox)
```

```
    reportVAP( 4 /*myp.gp.display */ );            // show to console
accordingly
    process_withWiFi = true;                // show to Thinger if activated and
WiFi OK
}
if( ticS.ready() )                          // reset stats every hour
{
    tmb.print("Short");
```

```
//      aveS.resetStats();                  // update short term stats (used in buckets)
  }
  if( ticL.ready() )                    // reset stats every day
  {
    tmb.print("Long");
//      aveS.resetStats();                  // update short term stats (used in buckets)
  }
  #ifdef INC_OLED                   // executed only if OLED has be enabled
  if( tuo.checkButton() )
    tuo.flipLoopScreen();
```

```
if( displayOledTic.ready() )        // update screen every time to measure
    tuo.showLoopScreen();
#endif
```

```
}
void loop()                         // Main loop
{
  stdLoop();
  if( checkWiFi() )                 // Good WiFi connection?
  {
    #ifdef INC_THINGER
    if( process_withWiFi )
    {
      process_withWiFi = false;
//        notifyIFTTT_V();          // Check if notifies required. If so, use the (now measurements)
myp.vap
//        notifyIFTTT_W();
//
//        sendtoDropbox();          // check if need to send to dropbox. If so, use long term averages
        if( myp.gp.scanON )
          thing.stream( thing["reading"] );
    }
    thing.handle();                 // Continue polling. If WiFi is diconnected
    #endif                      //   this will have delay loops!
  }
  else
  {
    reconnectWiFi();
  }
}
// ------------------------------ THINGER INITIALIZATION ---------------------------------------------
```

```
static char exrsp[1000];        // buffer used for Thinger CLI
BUF exbuf( exrsp, 1000 );       // BUF wrapper of exstr[]

void setupThinger()
{
    PF("----- Initializing THING\r\n");
    thing.add_wifi( eep.wifi.ssid, eep.wifi.pwd );  // initialize Thinker WiFi

    thing.set_credentials( USERNAME, DEVICE_ID, DEVICE_CREDENTIAL );
```

/*

- 'led' is turned ON or OFF for diagnostics
  /
  thing["led"] << invertedDigitalPin(MYLED);
  /
- 'streamON' is identical to 'scanON' EEPROM parameter. Either Thinger.io or Meguno or CLI can change this
  /
  thing["scanON"] << { if(inp.is_empty())
  inp= myp.gp.scanON; // get state of scanON
  else
  {
  myp.gp.scanON = inp;
  saveMyParm("scanON"); // set scanON in EEPROM
  }
  };
  /
- 'simulV' is set by a slider to values 60V-140V. Any value >90V activates simulation of Volts
- 'simulW' is set by a slider to values 0-10kW. Any value >0W activates simulation of Watts
- 'simulStatus' in 'reading' is used to show the simulation status
  */

```
thing["simulV"] << [](pson &inp) {   VAP v;
                        if( inp.is_empty() )
                        {
                           v = eng.getSimul();
                           inp=v.volts;          // return simulated voltage
                        }
                        else                     // set simulated voltage
                        {
                           v.volts = inp;
                           eng.setSimulV( v.volts );
                        }
                        myp.simulON = (v.volts>90.0) || (v.watts>0.0) ? true : false;
                       };
thing["simulW"] << [](pson &inp){   VAP v;
                        if( inp.is_empty() )
                        {
                           v = eng.getSimul();
                           inp = v.watts/1000.0;     // return simulated kW
                        }
```

```
                        else                    // set simulated voltage
                        {
                           v.watts = ((float)inp) * 1000.0;     // inp is kW
                           eng.setSimulW( v.watts );
                        }
                        myp.simulON = (v.volts>90.0) || (v.watts>0.0) ? true : false;
                       };
/*
```

- notfyV is entered as a slide in Thinger. It indicates the undervoltage condition to trigger IFTTT notification
- notfyW is entered as a slide in Thinger. It indicates the overpower condition to trigger IFTTT notification
- The 'notfy'-bits 1, 2, 4 eeprom parameter are used as indicators/masks of which condition were set

```
*/
   thing["notfyV"]    << inputValue( myp.gp.notifyV,{  if( myp.gp.notifyV >(NO_NOTFYV+0.5))
                                          myp.gp.notify |= 0x1;
                                       else
                                          myp.gp.notify &= 0x6;
                                       saveMyParm("notifyV");
                                       saveMyParm("notify");
                                      });
```

```
thing["notfyW"]     << inputValue( myp.gp.notifykW,{  if( myp.gp.notifykW >
(NO_NOTFYW+0.1))
                                                          myp.gp.notify |=
0x2;
                                                        else
                                                          myp.gp.notify &=
0x5;

saveMyParm("notifykW");

saveMyParm("notify");
                                                    });

thing["dropboxOn"]  << [](pson &inp){   if( inp.is_empty() )
                                            inp = (myp.gp.notify & 4) ? 1 :
0;
                                        else
                                        {
                                            if( inp )
                                                myp.gp.notify |= 4;
                                            else
                                                myp.gp.notify &= 3;
                                            saveMyParm("notify");
                                        }
                                    };
```

```
#ifdef TEMP_INC
    thing["dbMinut"]   << inputValue( myp.dbMinut,{    pf("DBX", (float)myp.dbMinut);
                                    myp.dropboxOn = (myp.dbMinut>0L);
                                    toeep.flags = myp.dropboxOn ? (toeep.flags|2) :
(toeep.flags&~2);
                                    toeep.dbMinut = myp.dbMinut;
                                    eep.saveParms();
                                    });
```

```
    thing["coilx4"]     << inputValue( myp.coilx4,   {   pzem.scaleAmps(
myp.coilx4 ? 4.0 : 1.0);
                                                        toeep.flags = myp.coilx4
? (toeep.flags|1) : (toeep.flags&~1);
                                                        eep.saveParms();
                                                        });

    thing["debug"]      << inputValue( myp.debug,  { pb( "Debug", myp.debug );
});
```

```
    thing["setMinReset"]  << { if( inp.is_empty() )
                                inp=myp.minuteReset;
                              else
                              {
                                myp.minuteReset=inp;
                                toeep.minuteReset = myp.minuteReset;
                                eep.saveParms();
                              }
                             };
#endif
  // -------------------- FROM THE BOARD or CODE, SENT TO THE CLOUD ----------------------------------------
---
/**
```

- pson "reading" is updated by the DEVICE every N-seconds or so. The DEVICE updates the "myp.volts"... section.
- Rotates one type of measurement at a time, i.e. all of them are done in 4xN seconds.
- This is used to display the instanteneous tachometers
-
- pson "engUpdate" is meant to be called every minute or hour with the AVERAGE volts,amps,watts.
- To be used by the LogUpdate
-
- pson "engDaily" is meant to be called every day. It returns the consumed energy during the day.

```
    */
      thing["reading"] >>       // meter is read asynchronously by the loop() every N-seconds
      {
        float base;
        ot["volts"]     = myp.vap.volts;
        ot["amps" ]     = myp.vap.amps;
```

```
    ot["watts" ]    = myp.vap.watts/1000.0;
    ot["energy" ]   = myp.vap.kWh;          // total energy as read by the meter
//          ot["dailyE" ]   = eng.getDailyEnergy( &base );
//          ot["ystday" ]   = base;


    ot["notfyStatus"] = (myp.gp.notify & 1) ? ((myp.gp.notify & 2) ? "UnderV
 OverPwr" :"UnderV")
                                            : ((myp.gp.notify & 2) ?
 "OverPwr":"Disabled");
    ot["dbStatus"]    = (myp.gp.notify & 4) ? "ACTIVE" : "OFF";

    ot["simulStatus"] = myp.simulON ? "ON": "OFF";

    ot["newscan"]     = myp.newscan;

  };
#ifdef TEMP_INC
  thing["engUpdate"] >>      // Buckets (15min or Hourly) in POLLING MODE. Use short term
averages
  {
    static VAP ave;                 // must be static!
    aveS.getStats( &ave );              // insert the averages for this duration
    ot["1:Volts"]     = ave.volts;
    ot["2:Amps" ]     = ave.amps;
    ot["3:kWatts" ]   = ave.watts;
    ot["Daily:kWh" ]  = eng.getDailyEnergy();
    ot["Total:kWh" ]  = myp.vap.energy;


    aveS.resetStats();                  // reset average counters for
 volts/amps/watts

  };
  thing["engDaily"]  >>      // Buckets DAILY. In POLLING MODE.
  {
    ot["Daily-kWh" ]   = eng.getDailyEnergy();
    ot["Total-kWh" ]   = myp.vap.energy;;
  };
#endif
/*
```

- pson "myCmd" is called manually from Thinger/Device API menu.
- This is interpreted as a command to be executed.
- The result is placed in pson ["myRsp"]

```
    */
    thing["myCmd"] << [](pson &in)
    {
       const char *p = (const char )in;
       PF("Received from Thinger [%s]\r\n", p );
       exe.dispatchBuf( p, exbuf );
       thing.stream( thing["myRsp"] );
```

};
thing["myRsp"] >> *// used by above*
{
ot = (const char)*exrsp;
};
/*

- pson "extEmail" is called from IFTTT with two optional ingredients, "extCmd" and "extInfo"
- The "extCmd" is assumed to be a CLI command, which is executed. The result of this execution is
- packaged in pson "emailRsp" and emailed to the endpoint "gkemail".

*/
#ifdef TEMP_INC
thing["extEmail"] <<
{
PF("Received from IFTTT cmd:[%s], info:[%s]\r\n",
(const char *)in["extCmd"], (const char *)in["extInfo"]);

```
strncpy( excmd, (const char *)in["extCmd"], sizeof( excmd ) );       //
this is the CLI command received
excmd[ sizeof( excmd )-1 ] = 0;
exe.dispatchBuf( excmd, exbuf );                          // execute the
command and place results to buffer

PFN("Sending email using Thinger.io Endpoint 'gkemail'" );
thing.call_endpoint( "gkemail", thing["emailRsp"]);     // Send an
email via Thinger. Nothing to do with IFTTT
                                                          // See pson
below. Use CLI email to test
```

};
thing["emailRsp"] >> // Used by above
{
ot["subject"] = rot.sf("Message from %s (%s)", myp.gp.devID, myp.gp.label);
ot["preface"] = rot.sf("This is in response to [%s]:", excmd);
ot["body"] = (const char*) exrsp;

```
IPAddress ip = WiFi.localIP();
ot["epilogue"]  = rot.sf( "RSSI is %ddBm. Device IP:%d.%d.%d.%d",
WiFi.RSSI(),ip[0],ip[1],ip[2],ip[3]  );
```

};
/*

- pson "extButton" is called either from IFTTT or manually from Device with one ingredient, "extCmd"
- Depending on "onoff" the switch is turned on or off
- The state is reported as "notification" to IFTTT using the end point "extNotify"

*/
thing["extButton"] << [](pson &in)
{

```
    const char *p = (const char )in["extCmd"];
    PF("Received from IFTTT trig:[%s]\r\n", p );
    exe.dispatchBuf( p, exbuf );
    thing.call_endpoint( "gknotify", thing["notifyRsp"]);
  };
  thing["notifyRsp"] >>                    // used by above
  {
    PFN( "Notifying with 'value1'=%s", exrsp );
    ot["value1"]   = (const char)exrsp;
  };
  /*
```

- pson "extAssist" is called either from IFTTT or manually from Device with two ingredient, "extNumb" and "extText"

```
  */
  thing["extAssist"] <<
  {
    int numb = atoi((const char *) in["extNumb"]);
    const char *p = (const char *)in["extText"];
    PF("Received from Google Assist via IFTTT:[%s]\r\n", p );
    exbuf.set("Nothing to do");                    // notification if ON/OFF not received

    if( strcmp(p,"on")==0 )
        exe.dispatchBuf( "relay 1", exbuf );
    if( strcmp(p,"off")==0 )
        exe.dispatchBuf( "relay 0", exbuf );
    thing.call_endpoint( "gknotify", thing["notifyRsp"]);

  };
  /*
```

- pson "extCall"

```
  */
  thing["extCall"] <<
  {
    const char *p = (const char *)in["extCmd"];
    PF("Received from IFTTT Call trig:[%s]\r\n", p );
    exe.dispatchBuf( p, exbuf );
    thing.call_endpoint( "gkcall", thing["callRsp"]);
  };
  thing["callRsp"] >>                    // used by above
  {
    PFN( "Calling with 'value1'=%s", exrsp );
    ot["value1"] = (const char *)exrsp;
  };
  #endif
  }

#ifdef TEMP_INC
  // define Authentication:
  void clickSendVoice( char *tel, char *tts )
  {
```

pson object;

```
    object["number"] = "15086471571";
    object["text"] = "This is a test line";

    thing.call_endpoint( "clickSend", object );
}
void cliCall( int n, char *arg[] )
{
    clickSendVoice( "", "" );
}
void cliNotify( int n, char *arg[] )
{
    exbuf.set( n>1 ? arg[1] : "EMPTY" );
    thing.call_endpoint( "gknotify", thing["notifyRsp"] );
}
void cliEmail( int n, char *arg[] )
{
    exbuf.set( n>1 ? arg[1] : "EMPTY" );
    thing.call_endpoint( "gkemail", thing["emailRsp"]);
}
CMDTABLE thsTable[]=            // functions to test IFTTT endpoints
{
    { "call",    "what_to_say",    cliCall },
    { "notify",  "what_to_notify", cliNotify },
    { "email",   "what_to_email",  cliEmail },
    {NULL, NULL, NULL}
};
```

// to activate, turnon myp.notfyVOn. Repeats notifications according to ALERT_REPEAT_PERIOD.
// Stops if alarm is off.

```
void notifyIFTTT_V()
{
  static int count = 0;
  static int times = 0;
  char buf[60];
  pson data;
  if( !myp.notfyVOn )
  {
    count = times = 0;
    return;                     // do nothing if not enabled
  }
  if( (myp.vap.volts < myp.notfyVmin) && (times<3) )
  {
    if( count == 0 )
    {
      sprintf( buf, "PZM#%d Low-Voltage %.1fV (<%.1fV)",times++, myp.vap.volts, myp.notfyVmin
);
      PF("AlarmV %s\r\n", buf );
      data["value1"] = (const char *)buf;
      thing.call_endpoint( "IFTTT_Notify", data );
    }
    count += METER_READING_PERIOD;
    if( count > ALERT_REPEAT_PERIOD )
```

```
            count = 0;
    }
}
void notifyIFTTT_W()
{
    static int count = 0;
    static int times = 0;
    char buf[60];
    pson data;
    if( !myp.notfyWOn )
    {
        count = times = 0;
        return;                         // do nothing if not enabled
    }
    if( (myp.vap.watts > myp.notfyWmax) && (times<3) )
    {
        if( count == 0 )
        {
            sprintf( buf, "PZM#%d Over-power %.3fkW (>%.3fkW)", times++, myp.vap.watts,
myp.notfyWmax );
            PF("AlarmW %s\r\n", buf );
            data["value1"] = (const char *)buf;
            thing.call_endpoint( "IFTTT_Notify", data );
        }
        count += METER_READING_PERIOD;
        if( count > ALERT_REPEAT_PERIOD )
            count = 0;
    }
}
void sendtoDropbox()
{
    static uint32 count = 0;
    char temp1[60], temp2[60];
    pson data;
    if( !myp.dropboxOn )
    {
        count = 0;
        return;                         // do nothing if not enabled
    }
    if( count == 0 )
    {
        VAP ave;
        aveL.getStats( &ave );              // get long term averages
```

```
    sprintf(temp1, "%5.1fV, %5.2fA, %5.3fkW, ", ave.volts, ave.amps, ave.watts
);
    sprintf(temp2, "%6.3fkWh, %6.3fkWh", eng.getDailyEnergy(), myp.vap.energy );
    data["value1"] = (const char *) temp1;
    data["value2"] = (const char *) temp2;
    PF("Dropbox %s%s\r\n", temp1, temp2 );
    thing.call_endpoint( "IFTTT_Dropbox", data );

    aveL.resetStats();                      // reset long term stats
}
count++;    // increments by one, every time a measurement is taken,
METER_READING_PERIOD
if( count > (myp.dbMinut*60L*METER_READING_PERIOD ) )
    count = 0;
```

}
#endif

// ---