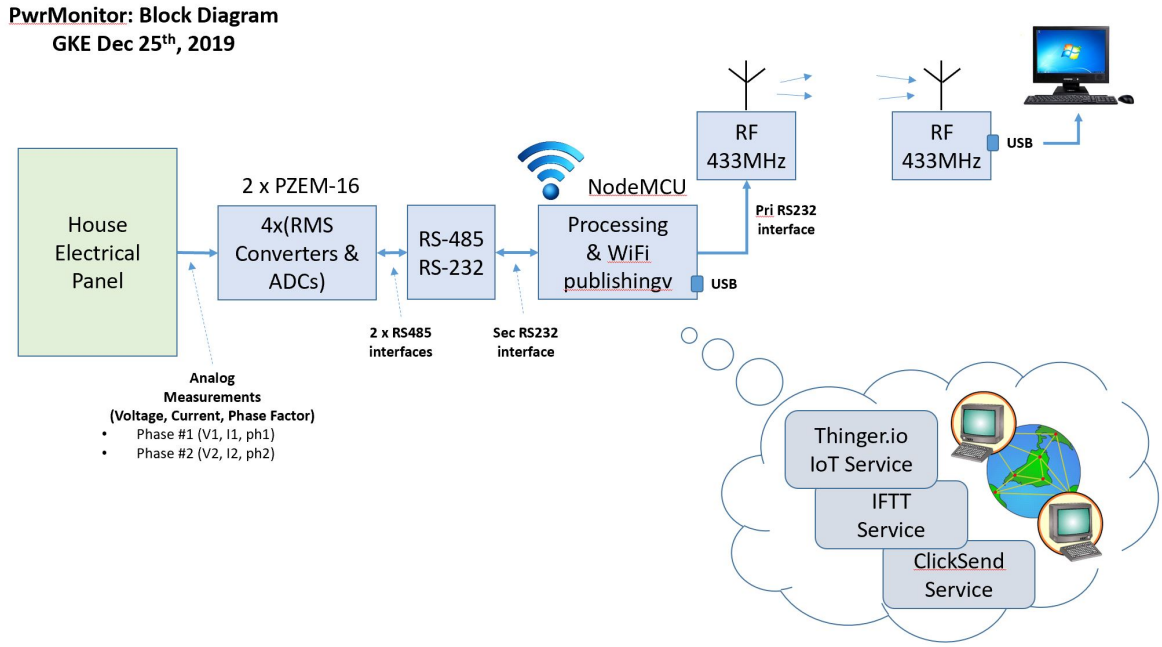
Power Monitor System

Designed by: George Kontopidis, Dec 2019

The **Power Monitor System** is a comprehensive implementation of a whole house electricity tracking system. It consists of the appropriate hardware interface to the the main electrical panel, a microcontroller connecting with the WiFi Access Point of the house and a cloud application.

The block diagram of the Power Monitoring System is shown below.



As shown, the system consists of the following subsystems:

- the electrical panel sensors
- the controller consisting of:
  - two converter of analog measurements (voltage, current, power and phase for the two phases of the house
  - the main controller including a WiFi interface
- the cloud services

Key features

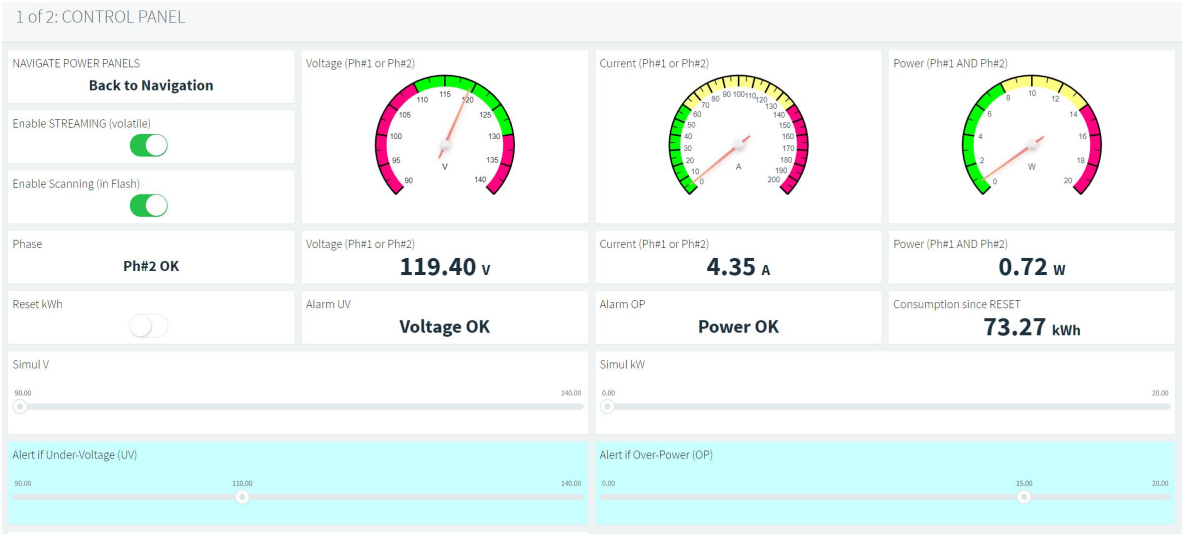
The Power Monitoring System is fully featured as follows:

1. Monitors both phases of the Electrical Panel, each of 120V (240V combined), up to 200A
2. Tracks the voltage, current, power-factor and real power of each phase
3. Compute the total energy consumed since system reset, as decided by the user.
4. Notifies/alerts the user when,
   - the voltage of either phase goes below a specified threshold
   - the total power consumption of the house exceeds a specified threshold .
   - Notifications are in form of text messages or phone-calls (using Text-to-speech)

5. Tracks all measurements on a minute, hour and daily basis and makes them available via a browser to a computer or a mobile device.
6. Stores important historical statics and notification/alarms to the Dropbox cloud service
7. Code is updateable via an Over-the-Air (OTA) service.

WEB Interface

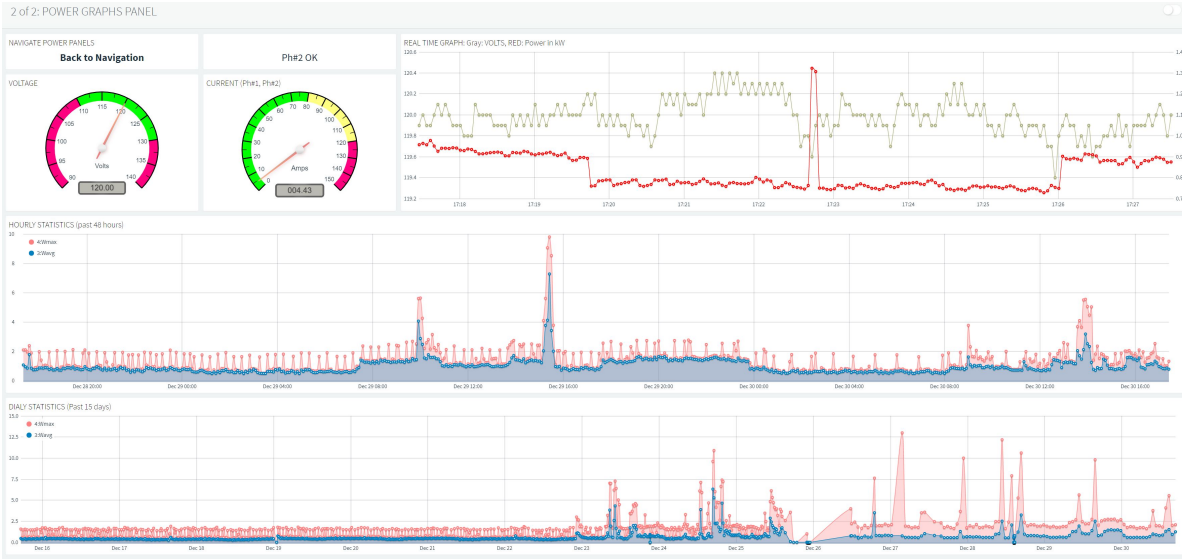
The WEB interface is implemented using standard Thinkger.io widgets. A brief navigator directs the user to one of the two screens below.



In this example, two alarm conditions have been activated: an under-voltage condition of 110V and over-power of 15kW.

The sliders labeled "SimulV" and "SimulW" allow the user to set various simulation conditions to Voltage and Power measurements. This way the alarm conditions can be verified.

The historical statistics are displayed in the WEB interface that follows



The upper right graph indicates the past minute measurements. The next two graphs represent the past 480-hour and past 15-days statistics.

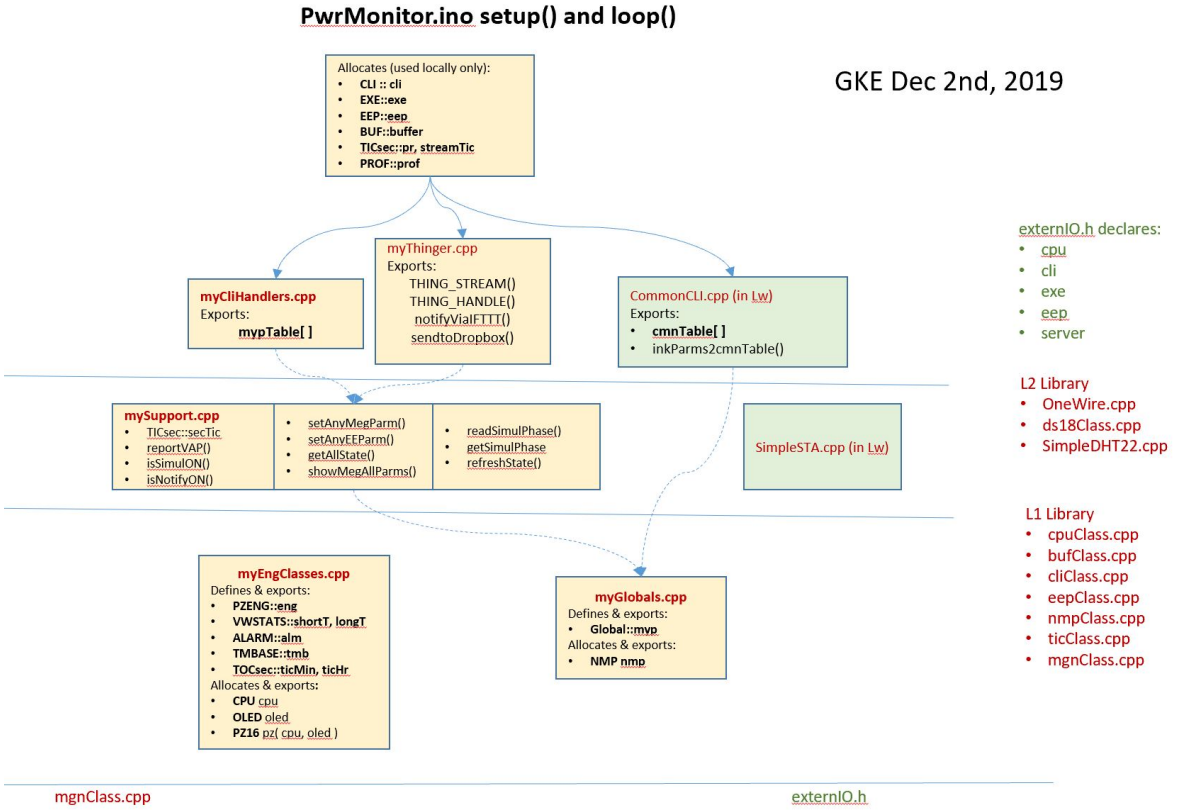
Note that Thinger.io Buckets HOURLY_STATS and DAILY_STATS accumulate respective statistics. The graphs above, can be adjusted to display any interval in the respective buckets.

Implementation Details

The following sections describe the code organization and various flows.

1. High Level File/Class Organization

The file/class dependency tree is shown below.



Summary of Low Level functions (App Layer 1)

The lowest level consists of **myGlobals** and **myEngClasses**. They define an export functions used by all layers above. There are no dependencies between each other, i.e. **myEngClasses** does not use any members of **myGlobals**.

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

myGlobals       nmpClass        Global myp      Global myp
                                NMP nmp         NMP nmp

myEngClasses    cpuClass        CPU cpu         CPU cpu
                oledClass       OLED oled       OLED oled
                pz16Class       PZ16 pz
                                PZENG eng       PZENG eng
                                VWSTATS shortT  VWSTATS shortT
                                VWSTATS longT   VWSTATS longT
                                ALARM alm       ALARM alm
                                TOCSec ticMin   TOCSec ticMin
                                TOCsec ticHr    TOCSec ticHr
```

Summary of Mid Layer functions (App Layer 2)

mySupport.cpp contains functions extensively used by **myCliHandlers** and **myThinger** App Level 3 and main loop(). It allocates and exports the following functions:

```
TICsec::secTic      reportVAP()     isSimulON()     isNotifyON()
setAnyMegParm()     setAnyEEParm()  getAllState()   showMegAllParms()
readSimulPhase()    getSimulPhase   refreshState()
```

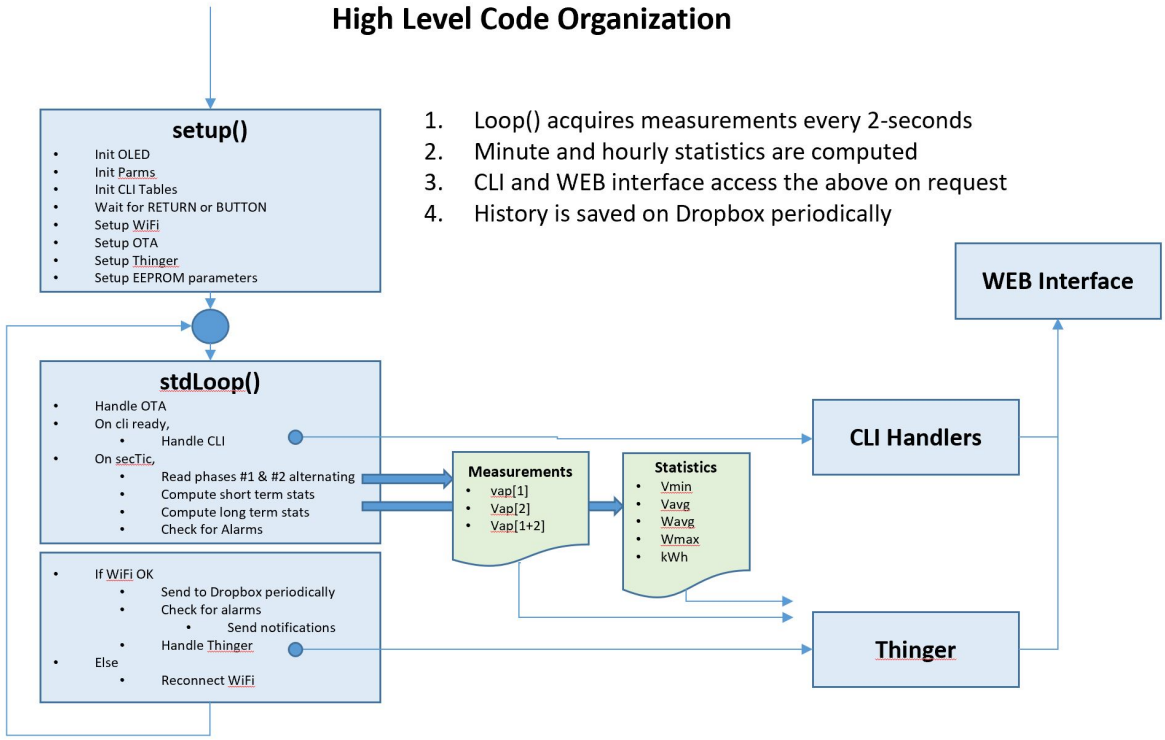
Summary of High Level functions (App Layer 3)

This layer consists of two files, **myCliHandlers** and **myThinger** which are independent of each other. The first includes all CLI handlers which are called directly from the main loop dispatcher. The second includes all necessary functions to enable the Thinger.io, that is:

```
Macros Used by main setup()
        SETUP_THINGER() i.e. setupThinger()
        CMDTABLE thsTable[];

Macros and Functions used by main loop()
        THING_HANDLE()  i.e. thingHandle()
        THING_STREAM()  i.e thingStream()
        void timedToDropbox();
```

2. High Level Process/flow Model



2.1 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.

2.2 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.

2.3 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)
```

2.4 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

2.5 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
```

2.6 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;
};
```

2.7 Synchronizing Thinger.io and MegunoLink

Code snippet 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
    .........
}
```

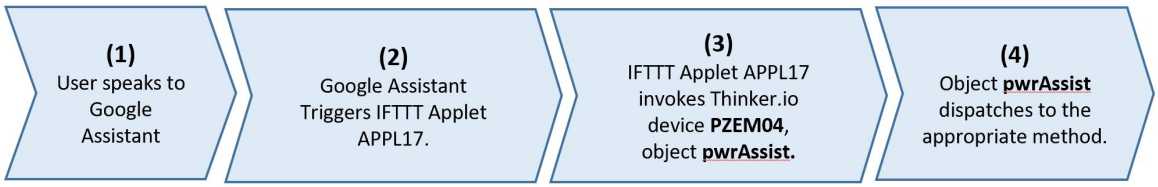
2.8 Frequently used CLI in Thinger.io

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

```
setEE ampscale 4.0              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
```

3. Google Assistant Interface

Voice interface using the Google Assistant was implemented as follows:



1. The user uses the mobile to invoke the assistant saying
    - "Report Power", or
    - "Report Power **by text**", or
    - "Report Power **by email**", or
    - "Report Power **by phone**", or
    - "Report Power **to Dropbox**"
2. Google Assistant triggers IFTTT applet "APPL17: Report Power by $" which makes as **Web Request** to Thinger.io {device=PZEM004, pson="pwrAssist"} and posting the type of request, i.e. by text.
3. Thinger.io uses the IoT channel to invoke the embedded pson object "pwrAssist", This object includes the following code:

```
thing["pwrAssist"] << [](pson &in)
        {
            const char *methd = (const char *)in["value2"];
            PF("Received [%s] [%s]\r\n", (const char *)in["value1"], methd
);

            if( !strcasecmp( methd, "by email" ) )
                sendByEmail();
            else if(!strcasecmp( methd, "by text" ))
                sendByText();
            else if(!strcasecmp( methd, "by phone" ))
                sendByPhone();
            else if(!strcasecmp( methd, "to dropbox" ))
                sendToDropbox();
            else
                sendByText();
                //sendByIFTTT();
            thing.stream( thing["CliRsp"] );      // update Thinger
notificatons
        };
```

4. The embedded code dispatches to the appropriate method to execute the request. In addition, it posts to the **cliRsp** object the acknowledgement of this response.

4. Reporting handlers

The following reporting handlers have been implemented

4.1 By Email

The reporting handler **sendByEmail()** works as follows:

1. Constructs a pson object consisting of the email subject and email body.
2. Invokes the Thinger.io Endpoint "gkemail". This Endpoint if of the **type "Email"** and it is responsible of sending the email to the pre-specified user address (george@42woodland.com in this case.)

4.2 By Text

The reporting handler **sendByText()** works as follows:

1. Constructs a pson object consisting of the text message to be sent.
2. Invokes the Thinger.io Endpoint "CLICK_TEXT". This Endpoint if of the **type "HTTP Request"** which:
    - calls: https://rest.clicksend.com/v3/sms/send
    - sets the appropriate HTTP headers and authentication to **ClickSend.com** service
    - posts the message:

      ```
      {"messages":[{"to":"+15085967673", "source":"sdk", "body":"
      {{value1}}"}]}
      ```

      which includes the number to be called and reference to "'value1" which receives the pson object constructed in (1)
3. ClickSend "does its magic" and sends the text (SMS) to the specified mobile phone.

4.3 By Phone

The reporting handler **sendByPhone()** works as follows:

1. Constructs a pson object consisting of the TTS message.
2. Invokes the Thinger.io Endpoint "CLICK_CALL". This Endpoint if of the **type "HTTP Request"** which:
    - calls: https://rest.clicksend.com/v3/voice/send
    - sets the appropriate HTTP headers and authentication to **ClickSend.com** service
    - posts the message:

      ```
      {"messages":[
          {"to":"15085967673",
          "body":"Hi George... This is your home calling... {{value1}}.
                  Use Google Assistant or your WEB SITE for more details.
                  Bye for now.", "voice":"male",
          "custom_string":"none",
          "country":"usa"}]}
      ```

      which includes the number to be called and reference to "'value1" which receives the pson object constructed in (1)

3. ClickSend "does its magic" and places the call to the specified mobile phone.

4.4 To Dropbox

The reporting handler **sendToDropbox()** works as follows:

1. Constructs a pson object consisting of the text to be saved.
2. Invokes the Thinger.io Endpoint "IFTTT_Dropbox". This Endpoint if of the **type "HTTP Webhook Trigger"** of the Applet "PZEM STAT".
3. IFTTT constructs a message using the ingredients Value1, Value2, Value3, as follows: "PZEM, {{OccurredAt}}, {{Value1}}{{Value2}}{{Value3}}". Then, appends this to Dropbox file **PZEM_STAT.TXT**