Tektronix®

無線IoT、低電力デバイスの パワー・プロファイルのデータ・ロギング

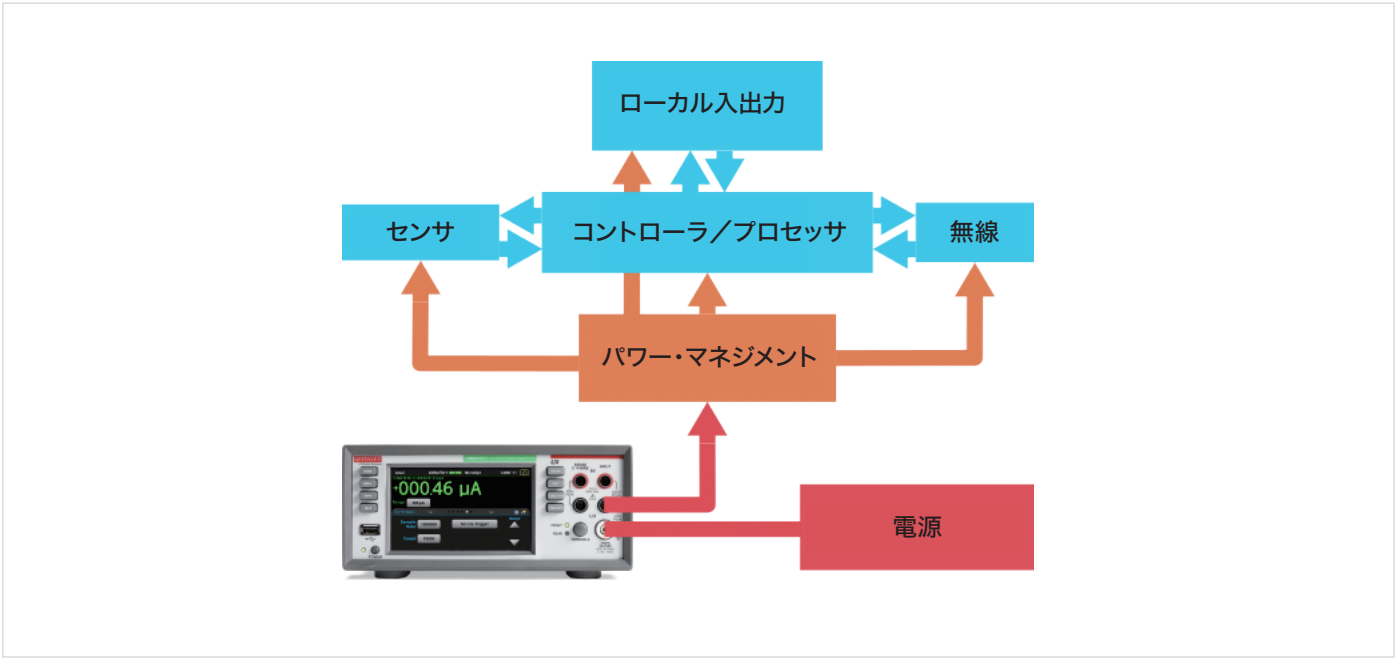
アプリケーション・ノート

KEITHLEY
A Tektronix Company

はじめに

一般に、データ・ロギングは決められた時間の間、データを取得します。取得したデータを解析することで、回路、モジュールまたは製品の性能を把握します。時間の長さは分から月に及ぶことがあり、HALT（Highly-Accelerated Life Testing）試験、HASS（Highly-Accelerated Stress Testing）試験などにより、過酷な環境条件でエージング・プロセスを加速して製品をテストすることがあります。電気デバイスの開発、テストでは、電流、電圧、電力、温度が一般的に保存されるデータ項目となります。一定の負荷条件で安定した出力のシンプルなデバイスでは、秒間隔でなくても、分間隔のサンプル間隔で十分です。短時間で数多くのパワー・ステートがあるデバイスでは、より高速のサンプリングが必要になります。

IoT（Internet of Things）デバイスなどの無線デバイスでは、電力消費が最も重要になります。このようなデバイスは、長時間放置され、手の届かないところにあったり、商用電源に接続できないことがあったりします。これは、センサ、リモート検知器、バッテリ駆動の家庭製品などの設計エンジニアには馴染みのあることです。このような無線デバイスは、大抵の時間がスリープ・モードや待機モードであり、理想的にはわずか数百nAから数十μAの電流しか流れません。しかし、この同じデバイスでも、センサを読み込んだり、セルフチェックを実行したり、LEDを点灯させたり、Bluetooth、LoRa、Wi-Fiなどの無線周波数で伝送するなど、タスクを実行するためにデバイスをアクティブ・モードにします。このような動作にはより多くの電力が必要であり、非常に短い時間であっても数百mA以上の電流が必要になることがあります。このようなデバイスをテストするには、高速の測定サンプリングが必要になります。高速にサンプリングすることで短時間の電流スパイクを見落とすことなく、デバイスの消費電力を見誤ることがなくなります。さまざまなステートにおけるIoTデバイスのパワー・プロファイルを理解することは、正しく動作させるだけでなく、デバイスのバッテリ寿命を延ばすことにもつながります。



無線IoTデバイスの電力測定の簡易ブロック図

ほとんどのDMM（デジタル・マルチメータ）は信号が動かない状態、またはゆっくりとした変化は測定できますが、無線デバイスが伝送するような短時間の負荷の電流変化は正しく取込めません。ケースレーのDMM6500型には16ビットのサンプリングA-Dコンバータが備わっており、最高1MS/sで電流または電圧測定のデジタイジングが行えます。この1µs間隔のサンプリングにより、非常に高速の負荷電流バーストを取得、負荷電流における予期しない伝送を観測することができます。デジタイジング機能は、DMMが従来から持っている電圧／電流機能と同じ、100mV～1000Vまたは10µA～10Aの測定レンジを持っています。デジタイジングの感度は、電圧、電流においてそれぞれ10µV、10nAです。DMM6500型は5型（12.7cm）マルチタッチ・スクリーン・ディスプレイも装備しているため、計測器に取得されたデータを優れたグラフ・ツール、データ解析ツールで表示することができます。収集したログ・データは、USBメモリに簡単に保存できます。または、テスト中であっても読み値をPCに転送できます。

このアプリケーション・ノートでは、デバイスの負荷電流特性、またはバッテリの電圧放電特性の取得における、さまざまな方法を説明します。まず、DMM6500型で利用可能なトリガ・モデル・テンプレートの一つを利用し、ゆっくりとした、シンプルなデータ収集方法を説明します。次に、DMM6500型のハイスピード・デジタイジング・モードでアクティブ状態のデバイス波形にトリガし、信号の平均電流がすばやく求められることを説明します。最後に、測定データのDMM6500型からPCへの直接ストリーミングを考察し、必要な解析を簡素化する方法をご紹介します。

従来のデータ収集方法

単純なデータ収集アプリケーションでは、特定の時間間隔で長時間、一つまたは複数の電気信号をモニタリングします。エンジニアは、いくつかの適当なタスクを実行し、長時間において動作をモニタリングすることで、設計の安定度を検証することがあります。

DMM6500型は、このようなデータ収集の設定と実行を簡単に行えるツールです。以下に示す例では、内蔵のトリガ・モデルのテンプレートを使用し、1秒間隔で信号をサンプリングして1時間のDC電圧を収集します。最後に、すべてのデータはUSBメモリにエクスポートしてPCで観測します。

DMM6500型の前面パネルから電圧の読み値を収集する

この例は、デバイスの全負荷状態における、デバイス内のリチウムイオン電池の電圧モニタリングを想定しています。毎秒ごとに読み値を取込み、1時間にわたって観測します。DMM6500型に内蔵されているトリガ・モデルを使用すると、このような測定が簡単に設定できます。グラフ機能で結果を表示し、表示されたイメージはスクリーンショットとして取込みます。最後に、データをUSBメモリにエクスポートします。

1. 前面パネルの**POWER**ボタンを押して電源を入れます。
2. FUNCTIONSスワイプ画面で**DCV**（DC電圧）を選択します。
3. **MENU**ボタンを押します。
4. Triggerヘッダから**Template**を選択します。
5. トリガ・モデルとして**SimpleLoop**を選択します。
6. カウントを3600（毎秒の読み値を1時間：60秒×60分）に設定します。
7. Delayを1秒に設定します。
8. **TRIGGER**ボタンを押します。測定方法の変更確認を求められるので、**Yes**を選択します。
9. 表示されるダイアログから**Initiate Trigger Model**を選択します。
10. **MENU**ボタンを押します。
11. Viewsヘッダで**Graph**を選択します。
12. グラフは、デフォルトではSmartScaleに設定されており、直近に取込まれたデータが画面右に表示されます。
13. **Scale**のタブを選択します。
14. X-AxisのMethodでShow All Readingsを選択します。

Style	Standard	
Append Mode	1	
Fill Mode	1	
Capacity	100000	
Count	2021	
Base Time Seconds	1517218170	
Base Time Fractional	0.598689284	
Base Time	29:30.6	
Reading	Relative Time	
3.301080526	0	
3.297693003	1.017782	
3.296889096	2.035536	
3.296503709	3.053297	

図1. CSVファイル出力の一部

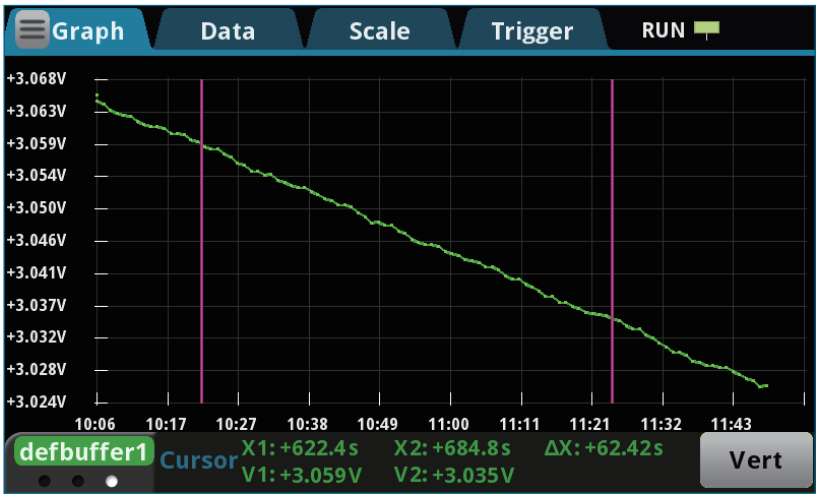


図2. グラフ表示でのカーソル使用

15. **Graph**タブを選択し、取込んだすべてのデータ・セットを観測します。画面下部に、取込んだデータの統計値であるBuffer Statsが表示されます。これは、すべてのグラフ・データから自動的に生成されます。

16. Buffer Stats情報を右から左にスワイプすると、カーソル情報が表示されます。

17. **Off**を選択して垂直軸カーソルを追加します。

18. カーソルの1本を選んでグラフ上に移動します。画面下部に、カーソルによる統計値が自動的に更新されて表示されます。

19. 前面パネルのUSBドライブにUSBメモリを差し込みます。

20. **HOME**ボタンを押したまま**ENTER**ボタンを押すと、画面イメージはUSBメモリに保存されます。

21. **MENU**ボタンを押します。

22. Viewsで**Reading Table**を選択します。

23. グラフのイメージを右端までドラッグすると、直近に取込んだデータの数値が表示されます。

24. 画面左上にあるメニュー・アイコン  を選択します。

25. **Save to USB**を選択します。

26. Reading Format、Status、Channelを押して無効にします。

27. 読み値のタイムスタンプ形式は変更できます。デフォルトは相対値であり、各読み値のトリガ・モデルの開始からの経過時間を示します。

28. **OK**を選択します。

29. USBメモリを外してPCに差し込み、データを解析します。

データはCSVフォーマットで保存されているため、代表的なスプレッドシート、解析プログラムなどで開くことができます。

このデータ収集方法は多くの場合に有効であり、この例ではトリガ・モデルの遅延を設定することでサンプル・レートが決まります（この例では1秒）。より短い測定間隔にする場合は、遅延を消去してアパーチャ時間を最小値に設定します。DMM6500型の場合、積分測定モードで8.333µsです。さらに高速でサンプリングし、信号の詳細を知りたい場合は、DMM6500型のデジタイジング機能を使用することで1µsのサンプリングが可能になります。

デジタイジング機能によるデータ収集

マイクロプロセッサを搭載したリモート無線デバイスは、統合されたセンサ、無線周波数（RF）トランシーバ、パワー・マネジメント集積回路（PMIC）とやりとりしており、それらに電力を供給するバッテリへの影響度合いが変わります。設計エンジニアは、すべてのデバイスの動作モード（スリープ、待機、伝送など）の入出力を知った上で、デバイスに有効なパワー・プロファイル構築し、パワー・バジェットに適合させる必要があります。

パワー・プロファイルから、問題となっている各状態の電流プロファイル（または平均電流）を知ることができます。電流プロファイルを正しく測定するには、高速のデジタイザを使用してトランジェント動作（予測する、または予測しないものも含めて）を観測し、計測します。DMM6500型には、1MS/sのサンプル・レートで電流または電圧の信号をデジタイジングする機能があり、特定のイベント信号を取得する、レベルトリガ・ツールを備えています。

長時間にわたってデバイスのパワー・プロファイルを観測し、すべてのデジタイズ測定データをPCに送って解析したいことがあります。ケースレーのDMM6500型は、高速の通信接続（通常はUSBまたはEthernet）により、リモートPCで制御し、このストリーミング動作を実行することができます。しかし、ストリーミングする時間の長さによってはギガバイトのデータ量になることがあり、コンパイルや解析に十分な時間がとれないことがあります。このような場合、デジタイザの使用をお勧めしますが、DMM6500型内部の演算機能を使用することで、データ保存と積算されたアンペアアワーまたはワットアワーの値を計算できます。

以下の例では、波形を取得して信号の平均電流を求める手順を説明します。また、TSPスクリプトにより、ストリーミングした測定結果からの詳細なデータ解析が容易になる、またはその手間が省けることを説明します。

デジタイジング機能で電流波形を取込む

先にも説明したように、ハイスピード・デジタイザを使用すると、デバイスの動作モードごとの電流プロファイルの検証に役立ち、目標の電源バジェットへの適合性検証に役立ちます。DMM6500型のデジタイジング機能は、従来のDMMと同じ電流／電圧レンジで使用できますが、サンプル・レートはより高速の1MS/sとなっています。選択された波形（数値データ、グラフの両方）は1µsの高分解能で取得されているため、設計どおりの動作をしているかを確認したり、または要件に適合するために何を修正すればよいかわかります。

この例では、レベル・トリガを設定し、Bluetoothデバイスの電流波形を自動的に取得します。DMM6500型は、下図に示すように無線デバイスとバッテリに直列に接続します。トリガ・モデルは、Bluetoothデバイスのイベント電流波形を取込めるように設定します。

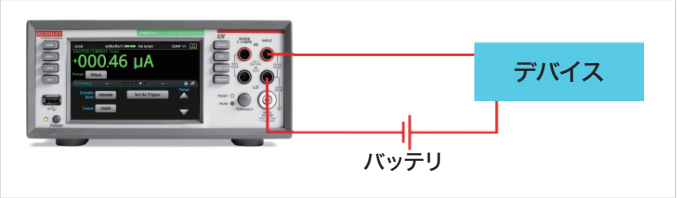


図3. DMMとデバイスの接続

1. 前面パネルの**POWER**ボタンを押して電源を入れます。
2. ディスプレイ右上に表示されている**CONT**を選択し、**Manual Trigger Mode**を選択します。
3. FUNCTIONSのスワイプ画面で**Dig I**（電流デジタイズ機能）を選択します。
4. **MENU**ボタンを押します。
5. Measureの項目で**Reading Buffers**を選択します。
6. defBuffer1のCapacityを1,000,000に設定し、OKを押します。
7. **MENU**ボタンを押します。

8. Viewsの項目で**Graph**を選択します。

9. **Trigger**タブを選択します。

10. Source Eventで**Waveform**を選択します。

11. **Analog Edge**を選択します。

12. レベル・トリガの電流値を適切な値に設定します。この例では、10mAに設定します。

13. **TRIGGER**ボタンを押すと、デジタイジングが開始します。

14. デバイスが動作を開始してトリガが動作すると、トリガ前の波形データが、バッファの50%（500,000回の読取り）になるように波形を取得します。

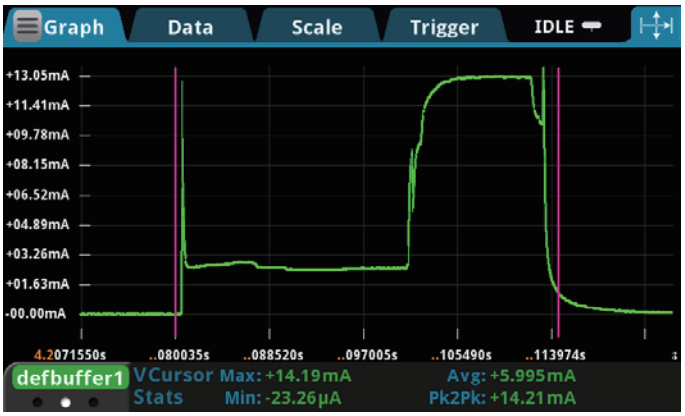


図4. 垂直軸カーソルを使用して、波形の特定の部分における平均電流を求める

Graphタブを選択して取込んだ波形を観測します。グラフをピンチして1つの波形をズーム表示します。この状態でカーソルを使用してより多くの情報を得たり、先の例で説明したようにデータをエクスポートしたりできます。波形の特定の部分に垂直軸カーソルを置くと、**図2**の例で示したようにVCursor Stats（垂直軸カーソル統計値）が自動的に計算され、その部分の重要な情報が表示されます。DMMの高速サンプリング機能により、すべてのデバイスの電力消費の詳細が観測できます。ロング・メモリと強力なグラフ機能により、電力のトランジェントまたは異常時におけるパワー・プロファイルが簡単にわかります。長い時間において多くのデータを取込むのは面倒なものです。1,000,000個のデータ・ポイントの取得に、わずか2～3秒しかかかっていませんが、これを短縮する一つのソリューションとしては、DMMに波形を取得しながら、外部PCを使用して、そのデータの受信と処理をします。

デジタイジングした値をPCに送る

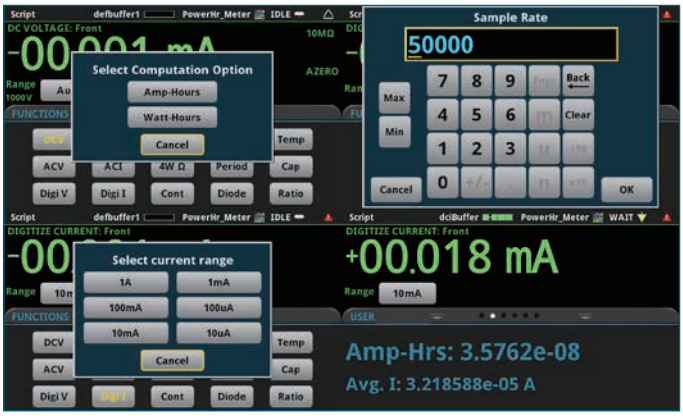
前の例では、特定の動作モードにおける平均電流の個々のセグメントの取得方法を説明しました。また、リモート・デバイス、無線デバイスが現実的なコンディションでどのように動作すべきかをテストするため、すべての動作モードが長時間にわたり、どのように連携して動作するのかもテストする必要があります。一般的には、規定の時間でデバイスを動作させ、高いサンプル・レートで電流の読み値を連続的に収集し、より包括的なパワー・プロファイルを求めます。この方法では、電流（または電力）と時間の両方に関係しているため、アンペアアワーまたはワットアワーを知ることで、設計で選定したバッテリが、電源バジェットを基にした寿命条件を満たすことを証明できます。

DMM6500型はこのデータ・ストリーミング方法に対応しており、最高100kS/sのサンプル・レートで取込み、同時に100,000回／秒でPCに直接デジタイジング・データを転送します。この速度は、使用する通信プロトコルによって異なります（USBが最速です）。データ解析はユーザによって異なるため、代わりにここではDMM6500型を使用して、測定をセットアップし、トリガし、データ・ストリーミングに必要なデータ抽出の詳細を説明します。例の詳細を以下に示します。

- メインの制御コードはPython 3.5以降で記述し、**付録A**に記載しています。
- 通信プロトコルはEthernetであり、DMM6500型とはTCP/IPソケットを使用して接続を確立します。
- 接続できたならば、TSPコマンドを含むスクリプト・ファイルをDMM6500型にアップロードします。これにより、Pythonコードからシンプルなローカル関数の呼び出しで、より詳細なリモート設定、トリガ動作、データ抽出処理が実行できます。テスト機器に対してより多くの処理を任せることで、PCは別のタスクに専念できます。このスクリプト・ファイルであるfunctions.luaは、付録Bに記載しています。
- データはDMM6500型から特定の量（または、コードで定義された量）を取出し、バイナリ・フローティング・ポイントとして受け取り、ASCIIに変換します。
- ASCIIフォーマットの読み値は、制御PCに保存されるデータ・ファイルに書き込まれます。

TSPによるDMM6500型のアンペアアワーまたはワットアワーの読み値収集

先に説明したデータ・ストリーミング方法では、使用するソフトウェアで解析する情報がギガバイトになることがあり、ユーザによってはすべてのデータでの作業を避けることがあります。このようなユーザは、動作しているデバイスのアンペアアワーまたはワットアワーを示す即時の、蓄積された値が示されるのを好みます。TSPのスクリプトにより、DMM6500型の動作をカスタマイズすることができます。



DMM6500型によるサンプル・スクリプトは、**付録C**に記載されており、その概要は以下の通りです。

- 1k～125kS/sで設定可能であり、レートが高いほどトランジェント信号の取得に有効です。
- Amp-Hours（アンペアアワー）またはWatt-Hours（ワットアワー）の計算が選択できます。
- 前面パネルには、デバイスのアンペアアワーまたはワットアワーの値と、デバイスの平均電流が表示されます。
- アンペアアワーまたはワットアワーのデータのグラフを出力

スクリプトのすべて、または一部をコピーし、お使いのエディタでペーストする場合の、スクリプトのロード、実行方法を以下に説明します。

スクリプトの実行

ここでご紹介する例は、TSP ScriptとケースレーのTest Script Builder（ウェブ・サイト：jp.tek.com/keithley でダウンロード可能）を使用し、無線デバイスで消費されるアンペアアワーを30日間記録します。TSPスクリプトはUSBメモリでDMMにロードされますが、ケースレーのTest Script Builderが実行できる外部PCからも実行できます。

コードは、本アプリケーション・ノート末尾の付録Cに記載されています。DMMの電流デジタイズ機能により、電流を測定して上書き可能なバッファに保存します。1秒に1回、このバッファの平均電流とテストの経過時間からデバイスのアンペアアワー（または、ソース電圧がわかっている場合はワットアワー）を計算し、別のバッファに記録します。この方法により、電流の小さなピークによって生ずる異常な電力を取込みながら、扱いやすいデータ・ポイント数を維持します。

1. 前面パネルの**POWER**ボタンを押して電源を入れます。
2. このアプリケーション・ノート末尾の付録Cに記載されているスクリプトをコピーしてUSBメモリに入れます。
3. 前面パネルのUSBドライブにUSBメモリを差し込みます。
4. ディスプレイ上部に表示される**No Script**を選択し、コピーしたスクリプト（usb1/PowerHr_Meterなど）を選択します。
5. **Amp-Hours**を選択します。
6. Sample Rateで50,000を選択します。
7. デバイスの最大電流値に対応する電流レンジを選択します。小さな電流ピークであってもレンジに収まるようにすることで、デバイスの特性が正確に評価できます。
8. ただちに測定が開始され、電流のアンペアアワーと平均電流がDMMのホーム画面に表示されます。

9. **MENU**ボタンを押します。
10. Viewsヘッダで**Graph**を選択します。
11. **Data**タブを選択します。
12. **defBuffer1**を選択し、次に**Remove**を選択します。
13. **Add**、次に**ampHrsBuffer**を選択し、グラフにアンペアアワー測定を追加します。
14. **Graph**タブを選択し、アンペアアワーの読み値を観測します。
15. データ取得を終了して、トリガ・モデルを停止するには、**TRIGGER**ボタンを押します。

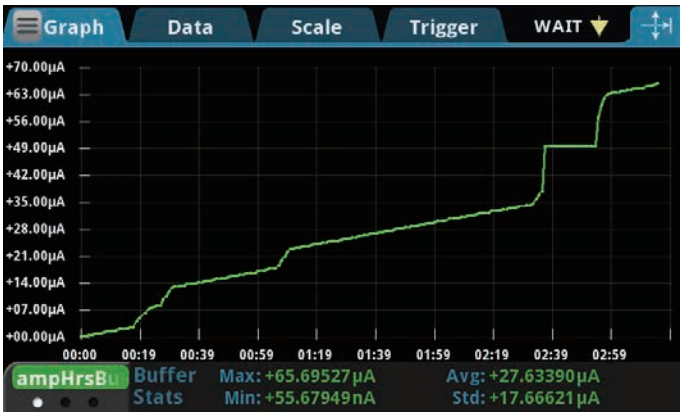


図7. アンペアアワー測定グラフの例。グラフのスロープ変化が電力消費の変化になる

データはCSVファイルでエクスポートすることにより、外部PCで解析できます。アンペアアワーの測定は期待値または検索された異常な電力（突然のスロープ変化となって現れる）と比較されます。この種の測定の他の一般的な用途には、デバイスに電力を供給するバッテリの仕様決定、または消費電力の仕様生成があります。

複数ポイントにおける、データ・ロガーによる複数の測定

このアプリケーション・ノートでご紹介した例の他にも、DMM6500型を使用して、熱電対、RTD、サーミスタによる温度、2つの電圧の電圧比など、15種類の測定機能によるデータ・ロガー機能が使用できます。オプションのスイッチ・カードを使用すると、DMM6500型は1つのデバイスまたは複数のデバイスのさまざまなポイントにおける、最大10チャンネルの測定データのログをとることができます。より多くのチャンネルが必要な場合は、ケースレーのDAQ6510型を使用すると、DMMと同じ機能で80チャンネルの測定が可能になります。

まとめ

顧客にとってバッテリ寿命は非常に重要であるため、IoTデバイスの負荷電流を理解し、これを抑えることは、市場での成功の鍵になります。ケースレーのDMM6500型6.5桁グラフィカル・サンプリング・マルチメータは、スリープ状態から送信状態まで、すべての動作におけるデバイスの負荷電流測定に必要な性能を備えています。この測定データとDMM6500型の解析機能により、設計エンジニアはデバイスの総合的なパワー・プロファイルを理解するための詳細情報が得られます。

付録A：Pythonスクリプトによるデータ・ストリーミングの実行

このPython 3.5スクリプトは付録BのTSP機能を使用しており、デジタイズした電流の読み値を、LAN接続した外部PCにストリーミングします。このスクリプトでは、60,000回／秒の読込みが期待できますが、この速度はLAN接続で使用するEthernetケーブルで制約を受けます。PyVISAによるUSB接続など、高速なデータ・バスを使用することでより高速なストリーミング速度が可能になります。PyVISAの詳細については、ウェブ・サイト（https://pyvisa.readthedocs.io）をご覧ください。

このスクリプトを実行するには、以下の変更が必要になります。

- ip_addressの変数をDMM6500型のIPアドレス（デフォルトは192.168.1.78）に合わせて設定します。
- seconds_to_captureの変数を、ロギングするデータの回数（デフォルトは10回／秒）に設定します。
- 付録BのTSP機能はfunctions.luaというファイルにあり、このスクリプトと同じディレクトリにあることを確認してください。

```
#!/usr/bin/python
import socket
import struct
import math
import time

#settings
seconds_to_capture = 10 # Modify this value to adjust your run time.
#minutes_to_capture = seconds_to_capture * 60
sample_rate = 60000 # NOTE: 60kS/s is the max rate we have observed under
                    #       certain conditions/circumstances. To attain
                    #       higher sampling and data transfer rates, use
                    #       USB.
chunkSize = 249     # This value is the max binary format transfer value
                    # we can implement for data transfer, and is limited
                    # by the ethernet protocol where the max frame size
                    # is < 1500 bytes, and this includes header/trailer
                    # information for each of the networking layers
                    # involved in the TCP/IP (physical, data link, network,
                    # and transport). The "chunkSize" variable defines how
                    # many readings to to transfer for a given poll of the
                    # instrument.
ip_address = "192.168.1.71"     # Place your instrument's IP address here.
output_data_path = "data.txt"   # This is the output file that is created which
                                # will hold your readings provided in ASCII
                                # format in a text file.
functions_path = "functions.lua"    # This file holds the set of TSP (Lua-
                                    # based) functions that are called by
                                    # the Python script to help minimize the
                                    # amount of bytes needed to setup up and
                                    # more importantly, extract readings from
                                    # the instrument. The file is opened and
                                    # written directly to instrument memory.

#helpers
# implement "chunkSize" instead of a fixed value
chunks = math.floor((seconds_to_capture * sample_rate) / chunkSize)

def load_functions(s):
    # This function opens the functions.lua file in the same directory as
    # the Python script and trasfers its contents to the DMM6500's internal
    # memory. All the functions defined in the file are callable by the
    # controlling program.
    func_file = open(functions_path, "r")
    contents = func_file.read()
```

```
    func_file.close()
    s.send("if loadfuncs ~= nil then "
                "script.delete('loadfuncs') "
           "end\n".encode())
   s.send("loadscript loadfuncs\n{0}\nendscript\n"
        .format(contents)
        .encode())
    s.send("loadfuncs()\n".encode())
    print(s.recv(100).decode())

def send_setup(s):
    # This function sends a string that includes the function
    # call and arguments that set up the DMM6500 for digitizing
    # current for the requested time and sample rate.
    s.send("do_setup({0}, {1})\n"
        .format(seconds_to_capture, sample_rate)
        .encode())
    s.recv(10)

def send_trigger(s):
    # This function sends a string that calls the function
    # to trigger the instrument.
    s.send("trig()\n".encode())
    s.recv(10)

def write_block(ofile, floats):
    # This function writes the floating point data to the
    # target file.
    for f in floats:
        ofile.write("{0:.4e}\n".format(f))

def get_block(s):
    # This function extracts the binaray floating point data
    # from the DMM65000.
    s.send("get_data()\n".encode())
    response = s.recv(1024)
    return response

def set_display(screen, state):
    # This function changes the display view and backlight settings
    s.send("disp_state({0}, {1})\n".format(screen, state).encode())
    s.recv(10)

#configure, trigger, transfer
s = socket.socket()                  # Establish a TCP/IP socket object
s.connect((ip_address, 5025))        # Connect to the instrument
ofile = open(output_data_path, "w") # Open/create the target data

load_functions(s)
send_setup(s)
set_display(16, 0)                  # Change to MENU screen; backlight off
send_trigger(s)

t1 = time.time()                     # Start the timer...
for i in range(0, int(chunks)):      # Loop to collect the digitized data
    write_block(ofile, get_block(s))# Write the data to file
    if i % 10 == 0:                  # This is here for debug purposes, printing
        print("{0:.1f}%".format(i/chunks * 100)) # out the % of run time elapsed
                                     # and technically it could be commented out.
t2 = time.time()                     # Stop the timer...

set_display(0, 1)                    # Change to HOME screen; backlight on
ofile.close()                       # Close the data file.
s.close()                            # Close the socket.
```

```
# Notify the user of completion and the data streaming rate achieved.
print("done")
print("{0:.0f} rdgs/s".format((chunks * chunkSize)/(t2-t1)))

input("Press Enter to continue...")
```

付録B：TSPスクリプト・ファイルによるデータ・ストリーミングのサポート

この機能は、**付録A**のPythonスクリプトで使用されます。付録AのPythonスクリプトと同じディレクトリにあり、functions.luaのファイルにあります。Pythonスクリプトはこの機能を機器の内部メモリにロードするため、最小の遅延で実行できます。データ・バスで必要となる通信も減るため、エラーの可能性も低くなります。

```
readings_captured = 0

function do_setup(capture_time, sample_rate)
    reset()
    dmm.digitize.func = dmm.FUNC_DIGITIZE_CURRENT
    dmm.digitize.range = 1
    dmm.digitize.samplerate = sample_rate
    format.data = format.REAL32

    trigger.model.setblock(1, trigger.BLOCK_DIGITIZE,
                           defbuffer1,
                           trigger.COUNT_INFINITE)

    trigger.model.setblock(2, trigger.BLOCK_DELAY_CONSTANT,
                           capture_time)

    trigger.model.setblock(3, trigger.BLOCK_DIGITIZE,
                           defbuffer1,
                           trigger.COUNT_STOP)
    waitcomplete()
    print("ok")
end

function trig()
    readings_captured = 0
    trigger.model.initiate()
    print("ok")
end

function get_data()
    chunker = 249
    while buffer.getstats(defbuffer1).n - readings_captured < chunker do
       delay(0.001)
    end
    local index1 = math.mod(readings_captured, 100000) + 1
    local index2 = index1 + (chunker - 1)
    if index2 > 100000 then
        index2 = 100000
    end
    printbuffer(index1, index2, defbuffer1)
    readings_captured = readings_captured + chunker
end

function disp_state(screen, state)
    if screen == 0 then
       display.changescreen(display.SCREEN_HOME)
    elseif screen == 1 then
```

```
        display.changescreen(display.SCREEN_GRAPH)
    elseif screen == 16 then
        display.changescreen(16)
    end

    if state == 0 then
        display.lightstate = display.STATE_LCD_OFF
    else
        display.lightstate = display.STATE_LCD_100
    end
    print("ok")
end

print("functions loaded")
```

付録C：TSPスクリプトによるアンペアアワーまたはワットアワー測定のロギング

ケースレーのDMM6500型でこのスクリプトを実行すると、デバイスのアンペアアワーまたはワットアワーが測定できます。スクリプトはテキスト・ファイルで、PowerHr_Meter.tspなどの名前にします。DMM6500型を接続したPCのKeithley Instruments Test Script Builderプログラム、またはDMMの前面パネルに差し込んだUSBメモリから直接実行できます。

実行すると、まず測定するのはアンペアアワーか、またはワットアワーかを確認されます。ワットアワーの場合はさらに、アンペアアワー測定と掛け算することでワットアワーを求めるための、印加するDC電圧を求められます。次に、サンプル・レートを設定します（デフォルトは50kS/s）。最後に、計算を簡単にするために電流レンジを設定します。

```
-- create functions

function setup_DMM6500_buffer(BufSize)
   dciBuffer = buffer.make(BufSize, buffer.STYLE_STANDARD)
   dciBuffer.clear()
   buffer.clearstats(dciBuffer)
   dciBuffer.capacity = 1 * BufSize
   dciBuffer.fillmode = buffer.FILL_CONTINUOUS
end -- function

function setup_DMM6500_measure(sampleRate, measRange)
   -- setup our refilling buffer
   setup_DMM6500_buffer(sampleRate)  -- BufSize = sampleRate = 1 second of buffering
   opc()

     -- setup measure type, ranges, etc.
   dmm.digitize.func = dmm.FUNC_DIGITIZE_CURRENT
   opc()

   dmm.digitize.range = measRange

   dmm.digitize.samplerate = sampleRate

   dmm.digitize.aperture = dmm.APERTURE_AUTO
   --Changing count is optional.  The reading buffer capacity is the determining factor
   dmm.digitize.count = 1  -- CANNOT be zero; 1 to 55Million

   -- control the swipe screen
   display.clear()
   display.changescreen(display.SCREEN_USER_SWIPE)

   -- clear any existing trigger blocks
   trigger.clear()
   trigger.model.load("Empty")
   opc()
```

```
   --Define a trigger model that will capture until we push front panel trigger button
   trigger.model.setblock(1, trigger.BLOCK_BUFFER_CLEAR, dciBuffer)
   trigger.model.setblock(2, trigger.BLOCK_DELAY_CONSTANT, 0)
   trigger.model.setblock(3, trigger.BLOCK_DIGITIZE, dciBuffer, trigger.COUNT_INFINITE)
   trigger.model.setblock(4, trigger.BLOCK_WAIT, trigger.EVENT_DISPLAY) -- wait until the
TRIGGER key is pressed
   trigger.model.setblock(5, trigger.BLOCK_DIGITIZE, dciBuffer, trigger.COUNT_STOP) -- stop
making digitized measurements

   opc()
end -- function

function my_dmm6500_waitcomplete(useWattHrs, dcvVal)
   local i = 1
   local cbIndex = 1
   local tempVal = 0

   -- check trigger model state on Amp-Hr meter (DMM6500)
   present_state, n = trigger.model.state()  -- state, present block number

   --STATE_RUNNING, IDLE, WAITING, EMPTY, FAILED, ABORTING, ABORTED, BUILDING
   while present_state == (trigger.STATE_WAITING or trigger.STATE_RUNNING) do
       reading_stats = buffer.getstats(dciBuffer)
       i_avg = reading_stats.mean
       runtime = dciBuffer.relativetimestamps[dciBuffer.n]
        AmpHrs = i_avg * runtime/3600
        if useWattHrs == 0 then
           display.settext(display.TEXT1, string.format("Amp-Hrs: %.4e", AmpHrs));
           display.settext(display.TEXT2, string.format("Avg. I: %.6e A", i_avg));
           tempVal = AmpHrs
           buffer.write.reading(ampHrsBuffer, tempVal, runtime)
       else
           WattHrs = AmpHrs * dcvVal
           display.settext(display.TEXT1, string.format("Watt-Hrs: %.4e", WattHrs));
           display.settext(display.TEXT2, string.format("Avg. I: %.6e A", i_avg));
           tempVal = WattHrs
           buffer.write.reading(wattHrsBuffer, tempVal, runtime)
       end

       delay(1)
       i = i + 1
       present_state, n = trigger.model.state()  --update the trigger model state var
       end  -- while loop
end -- function

function get_amphrs()
   present_state, n = trigger.model.state()

   reading_stats = buffer.getstats(defbuffer1)
   runtime = defbuffer1.relativetimestamps[defbuffer1.n]
   i_avg = reading_stats.mean
   AmpHrs = i_avg * runtime/3600
end --function

function set_dci_range()
   optionID = display.input.option("Select current range", "1A", "100mA", "10mA", "1mA",
"100uA", "10uA")
   if optionID == display.BUTTON_OPTION1 then -- 1A
       return 1.0
   elseif optionID == display.BUTTON_OPTION2 then -- 100mA
       return 100e-3
```

```
    elseif optionID == display.BUTTON_OPTION3 then -- 10mA
        return 10e-3
    elseif optionID == display.BUTTON_OPTION4 then -- 1mA
        return 1e-3
    elseif optionID == display.BUTTON_OPTION5 then -- 100uA
        return 100e-6
    elseif optionID == display.BUTTON_OPTION6 then -- 10uA
        return 10e-6
    end
end

function set_output_hrs_format()
    optionID = display.input.option("Select Computation Option", "Amp-Hours", "Watt-Hours")
    if optionID == display.BUTTON_OPTION1 then -- Amp-Hrs
        return 0
    elseif optionID == display.BUTTON_OPTION2 then -- Watt-Hrs
        return 1
    end
end -- function

function get_user_sample_rate()
    return display.input.number("Sample Rate", display.NFORMAT_INTEGER, 50000, 1000, 125000)
end -- function

function get_user_dcv_value()
    -- for the watt-hours, have the user input the applied voltage to their device
    return display.input.number("DCV Level Applied", display.NFORMAT_DECIMAL, 3.25, 0.0, 24.0)
end — function
-- ************************ MAIN PROGRAM *************************

reset()  --reset the DMM6500

eventlog.clear()

-- set default sample_rate and current_range
local sample_rate = 15e3
local DMMcurrentMeasRange = 0.01
local dcvVal = 0.0

-- downsize the default buffers to ensure room for the new ones
defbuffer1.capacity = 10
defbuffer2.capacity = 10

-- let us size this for 1 sample per second for up to 30 days: 60*60*24*30 = 2,592,000
local hrsFormat = set_output_hrs_format()
if hrsFormat == 0 then     -- provide semi-acceptable units to be visible on the graph and in
the reading table
    ampHrsBuffer = buffer.make(2592000, buffer.STYLE_WRITABLE)
    buffer.write.format(ampHrsBuffer, buffer.UNIT_AMP, buffer.DIGITS_6_5)
else
    wattHrsBuffer = buffer.make(2592000, buffer.STYLE_WRITABLE)
    buffer.write.format(wattHrsBuffer, buffer.UNIT_WATT, buffer.DIGITS_6_5)
end

if hrsFormat == 1 then
    dcvVal = get_user_dcv_value()
end

dmm.digitize.func = dmm.FUNC_DIGITIZE_CURRENT
dmm.digitize.range = DMMcurrentMeasRange
```

```
sample_rate    = get_user_sample_rate()-- let the user select the sample rate to use
DMMcurrentMeasRange  = set_dci_range()-- let the user select the current range to use
setup_DMM6500_measure(sample_rate, DMMcurrentMeasRange)

-- start our DMM6500 High Speed Digitizing
trigger.model.initiate()
delay(0.5)  -- allow some data to accumulate....

-- start the DMM6500 Amp-Hr status reporting loop
-- Press TRIGGER button to exit the loop
my_dmm6500_waitcomplete(hrsFormat, dcvVal)

-- clean up DMM6500 (Amp-Hr)
trigger.model.abort()
```

お問い合わせ先：

オーストラリア 1 800 709 465
オーストリア 00800 2255 4835
バルカン諸国、イスラエル、南アフリカ、その他ISE諸国 +41 52 675 3777
ベルギー 00800 2255 4835
ブラジル +55 (11) 3759 7627
カナダ 1 800 833 9200
中央／東ヨーロッパ、バルト海諸国 +41 52 675 3777
中央ヨーロッパ／ギリシャ +41 52 675 3777
デンマーク +45 80 88 1401
フィンランド +41 52 675 3777
フランス 00800 2255 4835
ドイツ 00800 2255 4835
香港 400 820 5835
インド 000 800 650 1835
インドネシア 007 803 601 5249
イタリア 00800 2255 4835
日本 81 (3) 6714 3086
ルクセンブルク +41 52 675 3777
マレーシア 1 800 22 55835
メキシコ、中央／南アメリカ、カリブ海諸国 52 (55) 56 04 50 90
中東、アジア、北アフリカ +41 52 675 3777
オランダ 00800 2255 4835
ニュージーランド 0800 800 238
ノルウェー 800 16098
中国 400 820 5835
フィリピン 1 800 1601 0077
ポーランド +41 52 675 3777
ポルトガル 80 08 12370
韓国 +82 2 6917 5000
ロシア +7 (495) 6647564
シンガポール 800 6011 473
南アフリカ +41 52 675 3777
スペイン 00800 2255 4835
スウェーデン 00800 2255 4835
スイス 00800 2255 4835
台湾 886 (2) 2656 6688
タイ 1 800 011 931
イギリス、アイルランド 00800 2255 4835
アメリカ 1 800 833 9200
ベトナム 12060128

2017年4月現在

jp.tek.com

テクトロニクス／ケースレーインスツルメンツ

お客様コールセンター： 技術的な質問、製品の購入、価格・納期、営業への連絡
TEL: 0120-441-046（ヨッ!良い オシロ） 営業時間／9：00～12：00・13：00～18：00（土日祝日および当社休日を除く）

サービス・コールセンター： 修理・校正の依頼
TEL: 0120-741-046（なんと良い オシロ） 営業時間／9：00～12：00・13：00～17：30（土日祝日および当社休日を除く）

〒108-6106 東京都港区港南2-15-2 品川インターシティB棟6階

記載内容は予告なく変更することがありますので、あらかじめご了承ください。

2018年5月　1KZ-61357-0