# Linduino for Power System Management

Michael Jones

## INTRODUCTION

Most Power System Management designs follow a set and forget model. Setup and debug of Power System Management (PSM) devices is simple with LTpowerPlay® and when combined with a bulk programing solution, there is no need for firmware. However, many large systems require a Board Management Controller (BMC), begging the question: "What can firmware do for PSM?"

The foundation of PSM firmware is PMBus; the foundation of PMBus is SMBus; and the foundation of SMBus is $I^2C$. Building a BMC that adds value with PSM firmware requires some level of knowledge of each protocol, or a pre-existing library that frees the programmer from the details.

The Linduino® libraries handle each protocol layer, and provide an Application Programming Interface (API), that makes writing PSM firmware easy. Linduino PSM is not a replacement for a BMC, but rather a set of libraries and examples that are compatible with typical BMC firmware.

Linduino can also be used with Linear Technology Demo Circuits as a learning tool. Many BMC designs already have an SMBus API and a quick study of how PMBus works is enough. It is quite common to see engineers copy/paste Linduino code snippets into an existing application and use them. But it is also possible to implement one of the Linduino layers and then reuse the whole library, including:

- Device and Rail Discovery
- Command API
- Fault Log Decoding
- In System Programming

This Application Note will present the Linduino libraries, Power System Management programming, setup and use of Linduino PSM with demo circuits, and PSM debugging techniques. For detailed information on the protocols and generic programming questions, refer to Application Note 135, Implementing Robust PMBus Software for the LTC3880, and the industry standards for $I^2C$/SMBus/PMBus.

## LINDUINO PSM HARDWARE

Linduino PSM hardware consists of a Linduino (DC2026), and a shield to connect (DC2294) the $I^2C$ pins of the Linduino to an PMBus/SMBus/$I^2C$ Bus of a demo board or product board.

For optimal learning, start with a DC2026 (Linduino), DC2294 (Shield), DC1962 (Power Stick), and a Total Phase Beagle ($I^2C$ Sniffer). This allows programming, debugging, and learning of controllers (LTC388X) and managers (LTC297X).

Figure 1. Evaluation Hardware, shows the suggested evaluation hardware all connected together. To use this

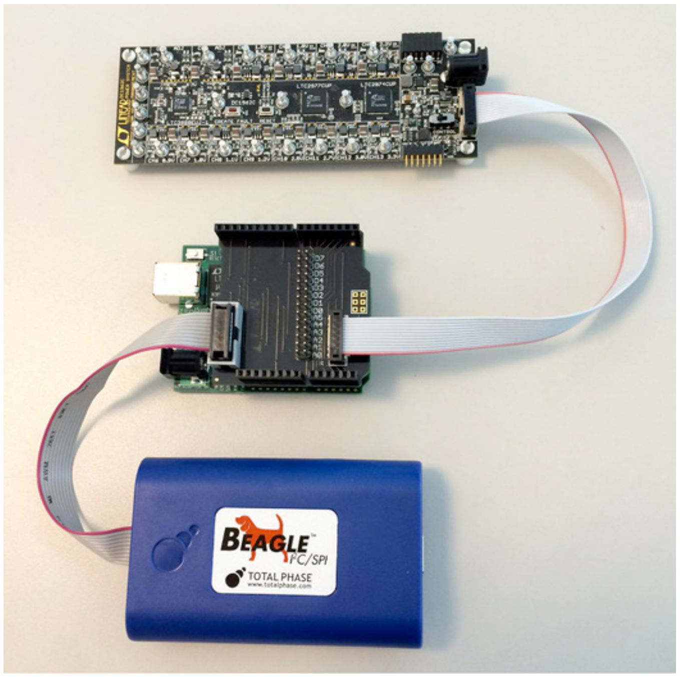


**Figure 1. Evaluation Hardware**

hardware, connect the Linduino and Beagle to a computer with two USB cables. If you do not have the Beagle USB connected, disconnect the Beagle ribbon cable from the DC2294 to prevent interference with PMBus traffic to/from the DC1962.

If you are connecting to a system board, a DC2086 works for most situations.

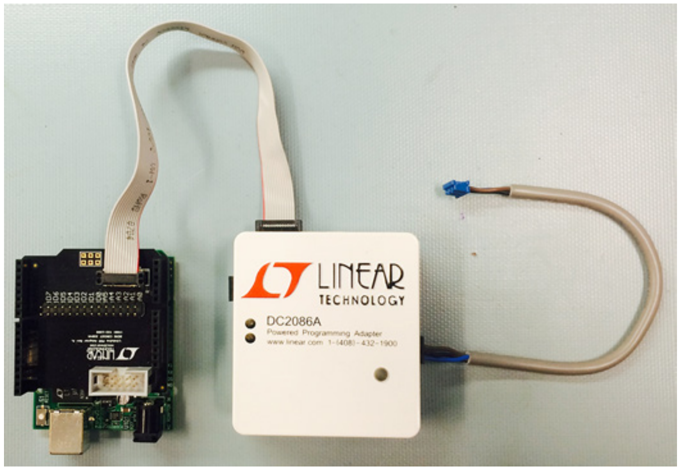


**Figure 2. System Hardware**

The DC2086 will accept a connection from the DC2294, and supports 12-pin ribbon, 14-pin ribbon, and 4-pin cables. The DC2086 also supports an external power input for system boards that require more power than the Linduino can supply.

## LINDUINO PSM SKETCHES

Before jumping into how the PMBus libraries work, a quick walk through of a DC1962 Sketch will clarify the general use model of Linduino PSM. It will also demonstrate how easy it is to write code, even for non-programmers.

To follow along, two downloads are required: the Arduino tools, and the Linduino Sketchbook. The Arduino tools can be downloaded from www.arduino.cc, and the Linduino Sketchbook can be downloaded from www.linear.com/linduino.

The Arduino tools run on multiple platforms. This Application Note was built and run using Arduino 1.6.4 running on 64-bit Ubuntu 14 TLS.

Let's get started:

### Step 1: Configuration

When the Arduino software is run the first time, it will be using a default Sketchbook, not the Linduino Sketchbook downloaded from www.linear.com.

To change to the Linduino Sketchbook, use the File | Preferences selection on the menu bar as shown in Figure 3. Finding Preferences Dialog.

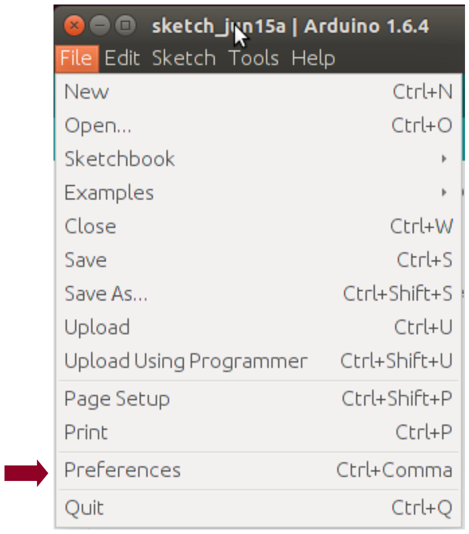
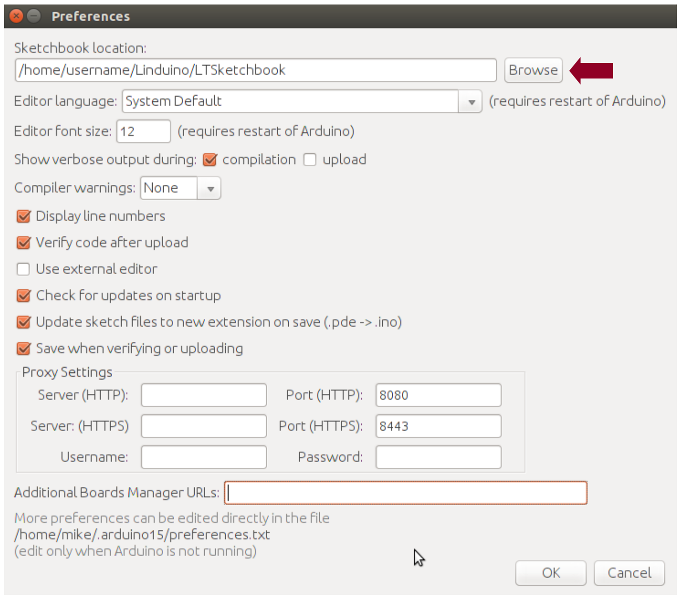


**Figure 3. Finding Preferences Dialog**

Figure 4. Preferences Dialog, shows that the Sketchbook Location is at the top of the dialog box. Using the Browse button, navigate to the LTSketchbook that was downloaded



Sketchbook location: /home/username/Linduino/LTSketchbook

Editor language: System Default (requires restart of Arduino)

Editor font size: 12 (requires restart of Arduino)

Show verbose output during: [✓ compilation] upload

Compiler warnings: None

[✓] Display line numbers

[✓] Verify code after upload

[ ] Use external editor

[✓] Check for updates on startup

[✓] Update sketch files to new extension on save (.pde -> .ino)

[✓] Save when verifying or uploading

Proxy Settings

Server (HTTP): Port (HTTP): 8080

Server: (HTTPS) Port (HTTPS): 8443

Username: Password:

Additional Boards Manager URLs:

More preferences can be edited directly in the file
/home/mike/.arduino15/preferences.txt
(edit only when Arduino is not running)

**Figure 4. Preferences Dialog**

an153fb

from www.linear.com/linduino. It is also helpful to check ON Display line numbers, and Show verbose output during compilation. The latter setting scrolls compiler messages on the command line where they are easier to see.

After setting the path, all Arduino windows must be closed, and the Arduino software must be restarted. When Arduino is restarted, it re-scans the Sketchbook directory and builds up the Arduino menus. If the Arduino software is not restarted, the menus will not reflect the LTSketchbook, and will instead point to the previous Sketchbook.

### Step 2: Load Your First Sketch

Load the hello_world Sketch by imitating Figure 5. Loading hello_world.

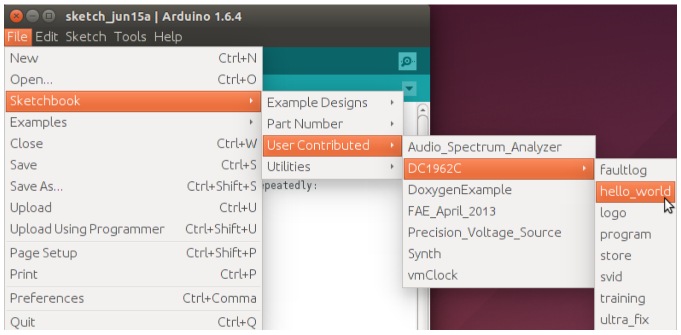


Figure 5. Loading hello_world

After the Sketch is loaded, a window with the Sketch pops up, as shown in Figure 6. Sketch Window.

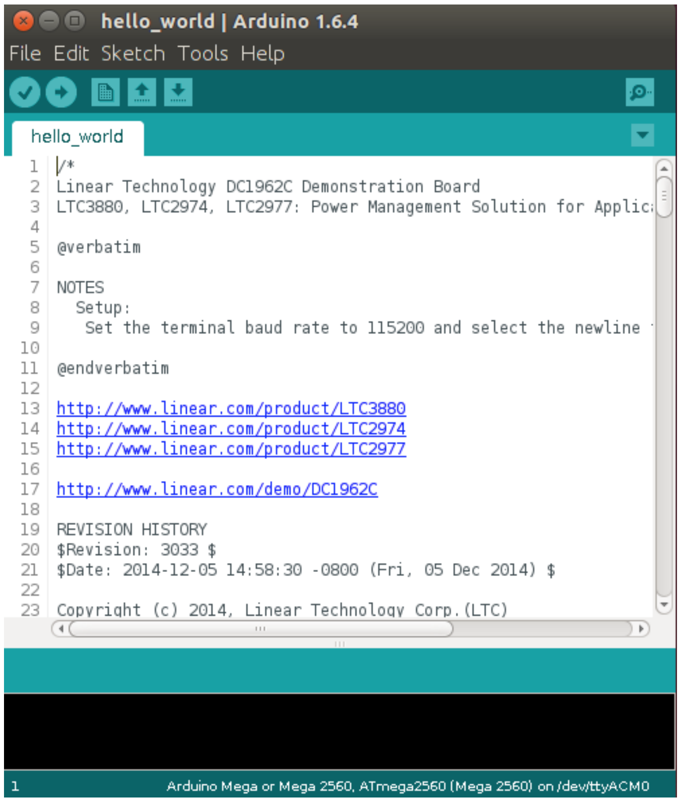
### Step 3: Compile and Run

Compile the Sketch by pressing the checkmark on the toolbar, as shown in Figure 7. Arduino Toolbar.

The arrow pointing right will compile and load the compiled Sketch into the Linduino hardware. The magnifier displays the output of the Sketch. Think of the arrow as sending the code to the display console, or compiling and sending the code to the Linduino hardware so that the display console has something to talk to.

*Note: The Arduino Board type should be set to Arduino Uno, and the port should be selected. See Tools Menu.*

After the Sketch is loaded, press the magnifier on the right side of the toolbar to open the console window. Set the line ending to *Carriage return* and baud rate to 115200 to match Figure 8. Arduino Command Window.



```
/*
Linear Technology DC1962C Demonstration Board
LTC3880, LTC2974, LTC2977: Power Management Solution for Applic

@verbatim

NOTES
  Setup:
   Set the terminal baud rate to 115200 and select the newline

@endverbatim

http://www.linear.com/product/LTC3880
http://www.linear.com/product/LTC2974
http://www.linear.com/product/LTC2977

http://www.linear.com/demo/DC1962C

REVISION HISTORY
$Revision: 3033 $
$Date: 2014-12-05 14:58:30 -0800 (Fri, 05 Dec 2014) $

Copyright (c) 2014, Linear Technology Corp.(LTC)
```

```
1     Arduino Mega or Mega 2560, ATmega2560 (Mega 2560) on /dev/ttyACM0
```

Figure 6. Sketch Window

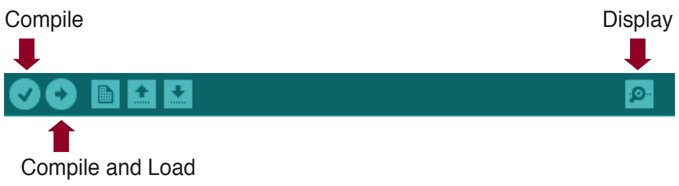


Figure 7. Arduino Toolbar

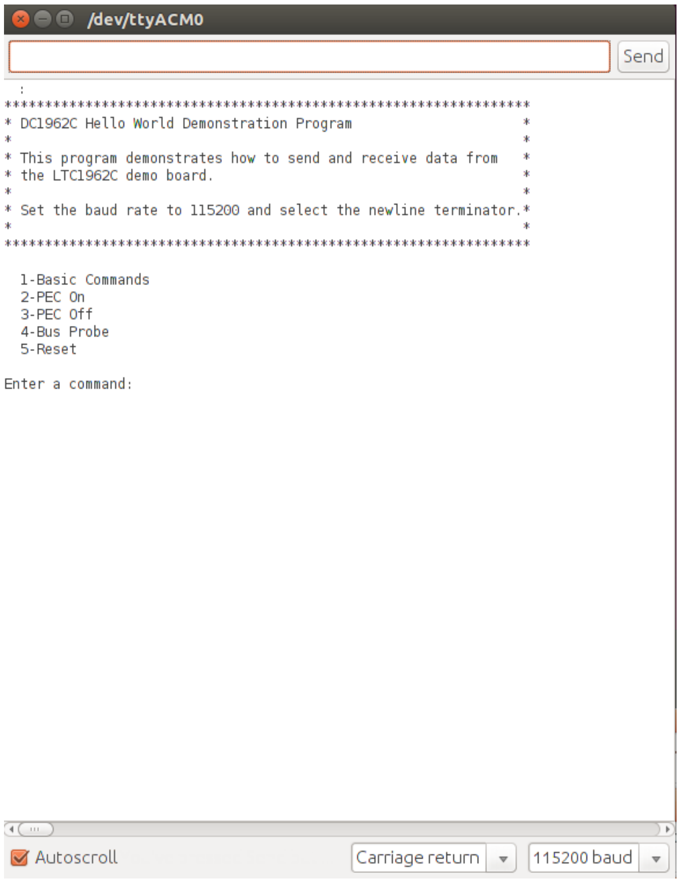
To interact with the sketch, put the cursor in the box at the top (left of the Send button), type a number from the menu, and then press the Send button or <CR>. The Sketch will then execute the command and then re-display the menu.

### Step 4: Explore Menu Items

Figure 9. Sketch Menus, shows what happens when pressing 1 to change to the Basic Commands window, followed by pressing 1 to Read All Voltages. The measured VOUT values of all rails of the DC1962 are read and printed.

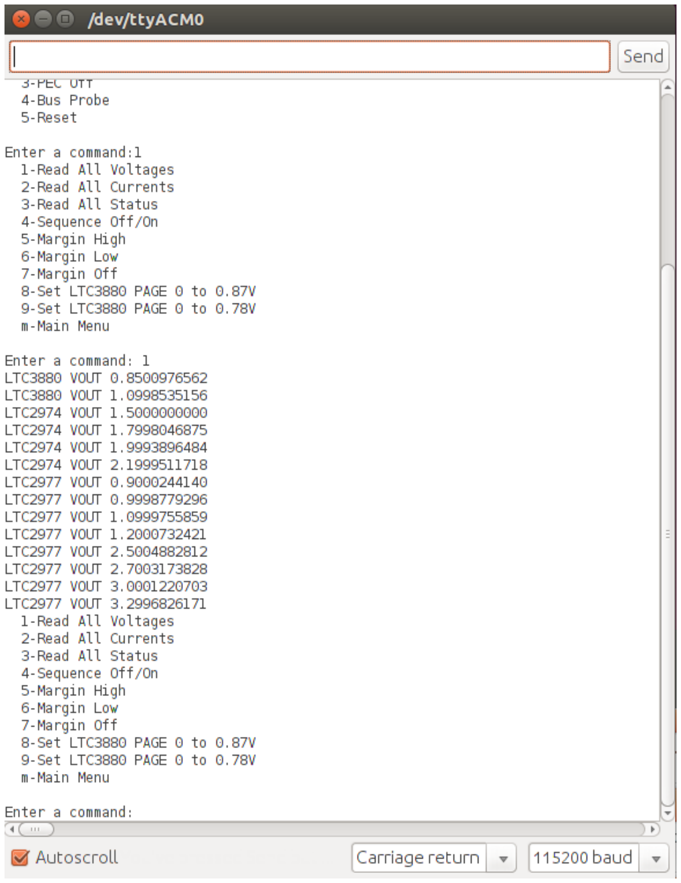
The Sketch runs until the console is closed by pressing the X in the top left corner. If the console is reopened, it will restart the Sketch.

You can now explore the other menu options, and if you are familiar with a Beagle, run some traces and examine the bus transactions.



```
 :
******************************************************************
* DC1962C Hello World Demonstration Program                     *
*                                                               *
* This program demonstrates how to send and receive data from   *
* the LTC1962C demo board.                                      *
*                                                               *
* Set the baud rate to 115200 and select the newline terminator.*
*                                                               *
******************************************************************

  1-Basic Commands
  2-PEC On
  3-PEC Off
  4-Bus Probe
  5-Reset

Enter a command:
```

Figure 8. Arduino Command Window



```
  3-PEC Off
  4-Bus Probe
  5-Reset

Enter a command:1
  1-Read All Voltages
  2-Read All Currents
  3-Read All Status
  4-Sequence Off/On
  5-Margin High
  6-Margin Low
  7-Margin Off
  8-Set LTC3880 PAGE 0 to 0.87V
  9-Set LTC3880 PAGE 0 to 0.78V
  m-Main Menu

Enter a command: 1
LTC3880 VOUT 0.8500976562
LTC3880 VOUT 1.0998535156
LTC2974 VOUT 1.5000000000
LTC2974 VOUT 1.7998046875
LTC2974 VOUT 1.9993896484
LTC2974 VOUT 2.1999511718
LTC2977 VOUT 0.9000244140
LTC2977 VOUT 0.9998779296
LTC2977 VOUT 1.0999755859
LTC2977 VOUT 1.2000732421
LTC2977 VOUT 2.5004882812
LTC2977 VOUT 2.7003173828
LTC2977 VOUT 3.0001220703
LTC2977 VOUT 3.2996826171
  1-Read All Voltages
  2-Read All Currents
  3-Read All Status
  4-Sequence Off/On
  5-Margin High
  6-Margin Low
  7-Margin Off
  8-Set LTC3880 PAGE 0 to 0.87V
  9-Set LTC3880 PAGE 0 to 0.78V
  m-Main Menu

Enter a command:
```

Figure 9. Sketch Menus

## Step 5: Modifying of Code

The Sketch has two entry points. There is a setup() function that is called once, and a loop() function that is called forever in a loop. These are part of the Arduino coding environment. If you are an experienced C programmer, you are probably wondering where is main()? The Arduino libraries have a predefined main() that calls setup() and an infinite loop calling loop().

The menus are coded as helper functions inside the Sketch, and loop() calls the main menu. Each menu is supported with a case statement, where each case handles one menu number.

OK, enough programmer-speak. Modifying the application is simply making changes to the case statements in the Sketch, using the provided API. The functions (API) in the Sketch that issue PMBus commands come from a separate library, and have simple names that sound like what you want the code to do. For example:

```
voltage = pmbus->readVout(0x30, false);
```

means, using the PMBus API, read the output voltage at address 0x30, without polling, and put it into a variable named voltage.

Now you should make a few changes. For example, add a menu item to read and print output power. If you're not ready yet, just read on to learn more about writing code. If your are ready, here is a hint:

```
float readPout(uint8_t address, bool polling);
```

Try this on the LTC3880 at address 0x30 on the Power Stick. To prove it works, add some resistance or a current load to Channel 0 on the Power Stick and verify it matches what the Sketch prints.

## LINDUINO PSM PMBus LIBRARY

The PMBus library resides in the LTSketchbook tree in the directory LTSketchbook/libraries/LT_PMBUS. The library is layered: starting with TWI (Two Wire Interface), then $I^2C$, SMBus, and finally PMBus. There is a number conversion

API to convert values from L11/L16 (PMBus formats) to/from floating point. Finally, there is Group Command Protocol assistance, device and rail discovery, Fault Log decoding, and even In System Programming.

Each layer is a simple C++ class, similar to how Arduino uses a class for Serial and other IO functions. If your final environment is C, don't worry. Simple means you can either use the C++ class without a lot of memory overhead, or you can remove the class wrapping and use it as pure C very easily. The C++ wrapper just simplifies the application code and makes it easier for non-programmers.

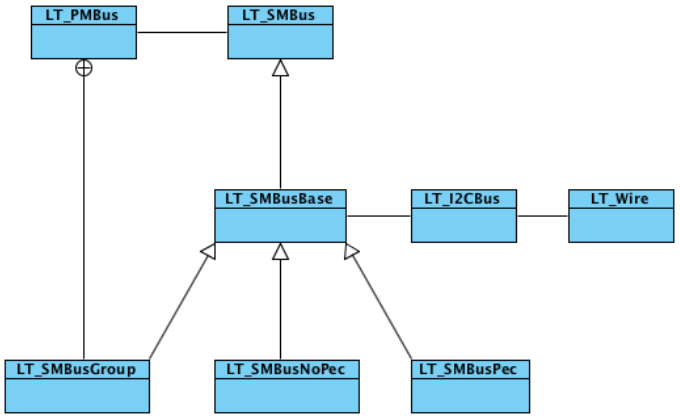


**Figure 10. LT_PMBus Class Diagram**

For programmers that just have to know what is under the hood, the SMBus classes are in a hierarchy so that application code is independent from turning PEC on and off, and aids porting. The LT_I2C.h, LT_SMBus.h, and LT_PMBus.h, form layers of APIs. To port the Linduino PSM libraries, you can choose any one of the APIs and implement it on your platform using your own libraries. The most common port re-implements the LT_SMBusBase class, and then the PMBus class just works, the math conversions just work, and all the other functions and examples just work.

## Using the PMBus Library

The library can be used without understanding all this class stuff; just a few imports and static variables and the Sketch is ready for action.

Normally the library is added with the Sketch menu shown in Figure 11. Include Library, but the most important includes are shown in Figure 12. Base Includes.

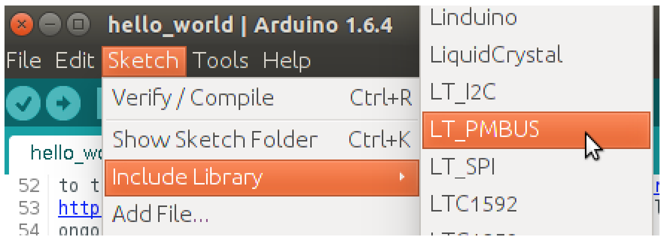


**Figure 11. Include Library**

```
#include <LT_SMBusPec.h>
#include <LT_SMBusNoPec.h>
#include <LT_SMBus.h>
#include <LT_SMBusMath.h>
```

**Figure 12. Base Includes**

A Sketch has at least two static variables, one for SMBus and one for PMBus, as shown in Figure 13. Static Variables. The SMBus variable is either the Pec or NoPec version. One nice feature of the clean layers is one can write application code as SMBus or PMBus code depending on their project needs. Writing an application with the PMBus API hides the details of the command codes and data formatting, and writing an application with the SMBus API enables access to all possible command codes and direct access to raw values.

```
static LT_SMBus *smbus = new LT_SMBusPec();
static LT_SMBus *smbus = new LT_PBusPec(smbus);
```

**Figure 13. Static Variables**

Once the two variables are initialized, the full SMBus API is available via smbus-> and the full PMBus API is available via pmbus->. It is possible to use both APIs in the same application.

## LT_PMBusMath

The LT_PMBusMath class is a highly optimized number conversion library to and from L11/L16 and floating point. Floating point is not required for PMBus code, and some end user applications are written only with integers, especially if voltage and current values are known ahead of time, or if floating-point conversion is too slow on a very small microcontroller. If the conversions are not needed by firmware, they can still be used by an offline application to generate integers for the application. However, it is much easier to write code when functions use floating point.

### The LT_I2C Library

The LT_I2C library is different from the I$^2$C class found in the LT_PMBus library. The version in LT_PMBus is byte order optimized for PMBus in addition to supporting large block operations. The I$^2$C class in the LT_PMBus library is also based on the Wire library, and is more portable to other Arduino boards. For example, it works on an Arduino Mega 2560.

All the non-PSM Sketches use the LT_I2C library. It is best not to use the LT_I2C library for PSM/PMBus devices, and there is no need to.

## WRITING A SIMPLE SKETCH

The best way to learn is to write code from scratch, and that is what this section is about. The following example will be most helpful if you perform these steps yourself one at a time, verifying the results as you go along.

This example Sketch uses a DC1962 and the other hardware mentioned in the first section. The example will accept 5 simple commands:

1. Print Voltages
2. Margin
3. On/Off
4. Bus Probe
5. Reset

### Step 1: Create a Blank Sketch

Create a new Sketch by selecting menu File|New as shown in Figure 14. New Sketch. You will then see an empty sketch as shown in Figure 15. Empty Sketch.

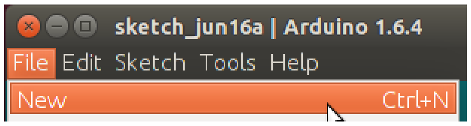
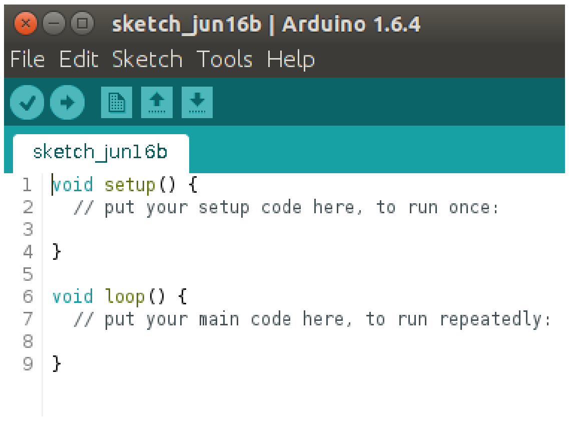


Figure 14. New Sketch

Use the File|Save As… menu and select a path for the new Sketch. Be sure the folder name and Sketch name match as shown in Figure 16. Save As.... The path must be under the LTSketchbook, the same path that is in the File|Preferences dialog, for it to show up in the Sketchbook Menu.



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Figure 15. Empty Sketch

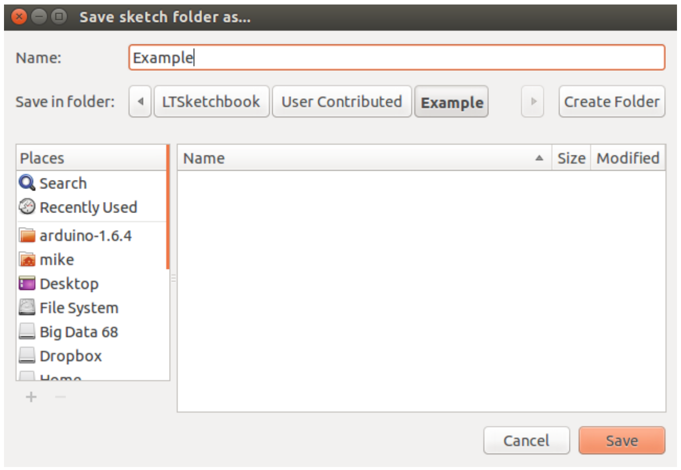


Figure 16. Save As...

### Step 2: Adding Includes

Using the menu Sketch|Include|Include Library, choose the following libraries one at a time:

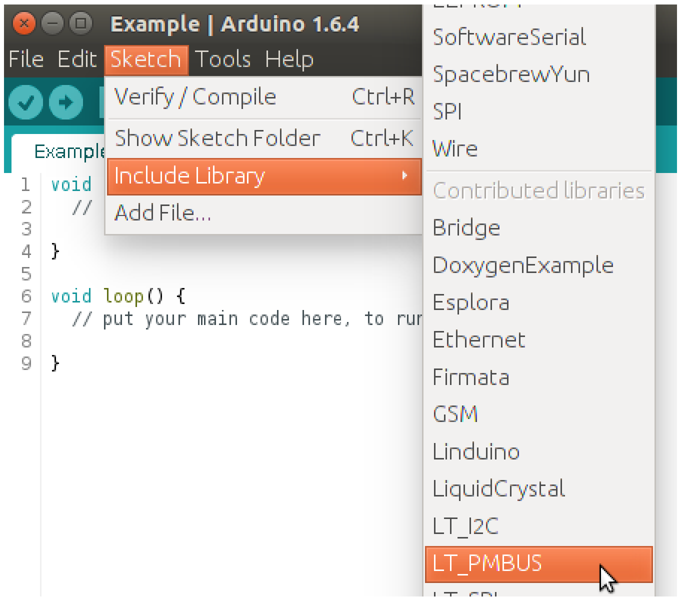
- User Interface
- Linduino
- LT_PMBUS

Figure 17. Includes, shows how to select the LT_PMBUS library. All libraries are in the same Include Library menu.

On the top line of the file add this include statement:

```
#include <Arduino.h>
```

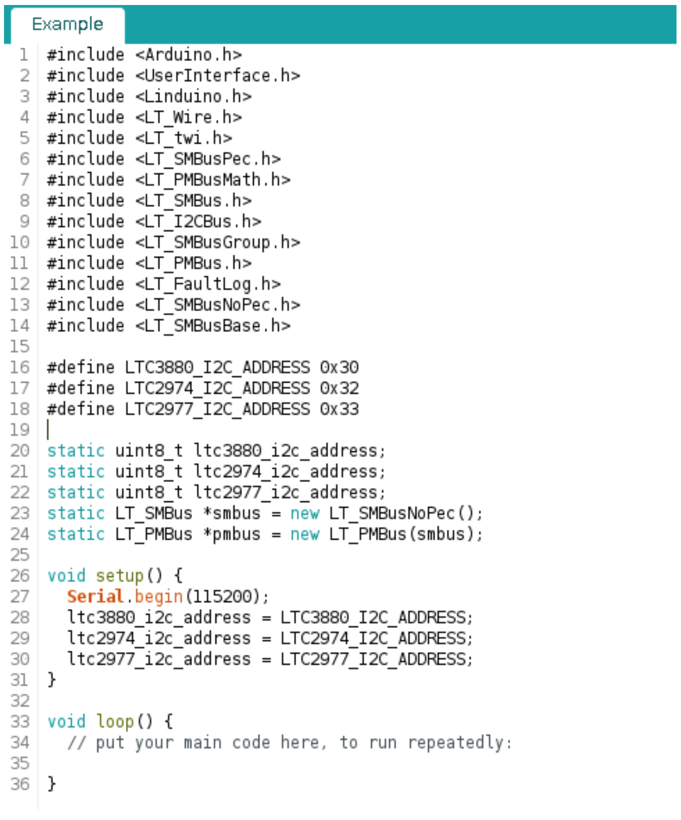
Now, add static variables for the addresses and SMBus/PMBus objects. Add Setup code to initialize variables and serial bus object. Save the code with File|Save. Finally, press the check button on the toolbar to compile the code.



```
void
  //

}

void loop() {
  // put your main code here, to run

}
```

**Figure 17. Includes**

Your code should look like Figure 18. Initialization Code.



```
#include <Arduino.h>
#include <UserInterface.h>
#include <Linduino.h>
#include <LT_Wire.h>
#include <LT_twi.h>
#include <LT_SMBusPec.h>
#include <LT_PMBusMath.h>
#include <LT_SMBus.h>
#include <LT_I2CBus.h>
#include <LT_SMBusGroup.h>
#include <LT_PMBus.h>
#include <LT_FaultLog.h>
#include <LT_SMBusNoPec.h>
#include <LT_SMBusBase.h>

#define LTC3880_I2C_ADDRESS 0x30
#define LTC2974_I2C_ADDRESS 0x32
#define LTC2977_I2C_ADDRESS 0x33

static uint8_t ltc3880_i2c_address;
static uint8_t ltc2974_i2c_address;
static uint8_t ltc2977_i2c_address;
static LT_SMBus *smbus = new LT_SMBusNoPec();
static LT_PMBus *pmbus = new LT_PMBus(smbus);

void setup() {
  Serial.begin(115200);
  ltc3880_i2c_address = LTC3880_I2C_ADDRESS;
  ltc2974_i2c_address = LTC2974_I2C_ADDRESS;
  ltc2977_i2c_address = LTC2977_I2C_ADDRESS;
}

void loop() {
  // put your main code here, to run repeatedly:

}
```

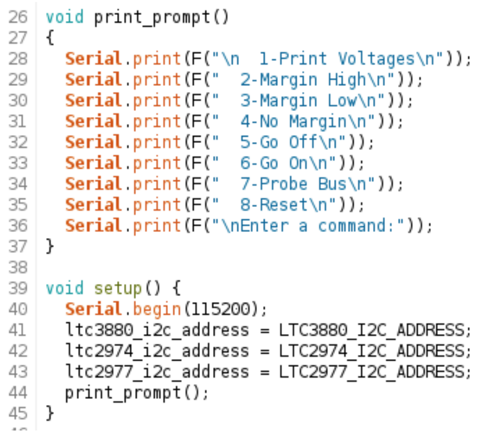
**Figure 18. Initialization Code**

### Step 3: Setup the Menu

A menu requires printing selections, and responding to user choices.

Add a print_prompt() function to print a prompt, and call it from the setup function to print the menu prompt when the Sketch runs. The code should look like Figure 19. Prompt.

Save and compile to ensure the code is error free.



```
void print_prompt()
{
  Serial.print(F("\n  1-Print Voltages\n"));
  Serial.print(F("  2-Margin High\n"));
  Serial.print(F("  3-Margin Low\n"));
  Serial.print(F("  4-No Margin\n"));
  Serial.print(F("  5-Go Off\n"));
  Serial.print(F("  6-Go On\n"));
  Serial.print(F("  7-Probe Bus\n"));
  Serial.print(F("  8-Reset\n"));
  Serial.print(F("\nEnter a command:"));
}

void setup() {
  Serial.begin(115200);
  ltc3880_i2c_address = LTC3880_I2C_ADDRESS;
  ltc2974_i2c_address = LTC2974_I2C_ADDRESS;
  ltc2977_i2c_address = LTC2977_I2C_ADDRESS;
  print_prompt();
}
```

**Figure 19. Prompt**

### Step 4: Add Menu Responses

When a menu option is typed into the console, the code must read it and respond.

The loop function will handle user input. First, it must check that serial bus is available. Then, it must read the input as an integer and pass it to a switch statement. Inside the switch, it must perform some function and then call the prompt function. The code for each command will be written inside each case of the switch statement, and the prompt will be called afterwards. Your code framework should look like Figure 20. User Input.

Save and compile it to make sure it has no mistakes.

### Step 5: Reading Voltages

Now it is time to write actual PMBus code that does something useful.

Figure 21. Read Voltages, shows code that reads all the voltages. The code is shown inside case 1. Two variables hold voltage and page: a float for the voltage, and a uint8_t for the page, shown on lines 57-58. Printing uses the standard Arduino Serial.print… function. The F() around

```
void loop() {
  uint8_t user_command;

  if (Serial.available())
  {
    user_command = read_int();

    switch (user_command)
    {
      case 1:
        break;
      case 2:
        break;
      case 3:
        break;
      case 4:
        break;
      case 5:
        break;
      case 6:
        break;
      case 7:
        break;
      case 8:
        break;
    }
    print_prompt();
  }
}
```

**Figure 20. User Input**

```
    switch (user_command)
    {
      case 1:
        float   voltage;
        uint8_t page;

        Serial.println(F(""));
        for (page = 0; page < 2; page++)
        {
          pmbus->setPage(ltc3880_i2c_address, page);
          voltage = pmbus->readVout(ltc3880_i2c_address, false);
          Serial.print(F("LTC3880 VOUT "));
          Serial.println(voltage, DEC);
        }

        for (page = 0; page < 4; page++)
        {
          pmbus->setPage(ltc2974_i2c_address, page);
          voltage = pmbus->readVout(ltc2974_i2c_address, false);
          Serial.print(F("LTC2974 VOUT "));
          Serial.println(voltage, DEC);
        }

        for (page = 0; page < 8; page++)
        {
          pmbus->setPage(ltc2977_i2c_address, page);
          voltage = pmbus->readVout(ltc2977_i2c_address, false);
          Serial.print(F("LTC2977 VOUT "));
          Serial.println(voltage, DEC);
        }
        break;
      case 2:
        break;
```

**Figure 21. Read Voltages**

the strings puts them in flash so they do not use up precious RAM. For each device, a *for loop* indexes through the pages calling pmbus->setPage(…) followed by reading voltage with pmbus->readVout(…). Then the code prints the voltages in decimal using the DEC type.

You can find all the API function declarations you used in the LT_PMBus library in the PMBus.h file, or in the Doxigen documentation.

## Step 6: Margining and On/Off

The margining code is simpler than the voltage code because the operations are global, meaning all devices can respond to one command, and the page register is not required. Furthermore, there is nothing to print.

Figure 22. Margin and On/Off, shows the code. Case 4 is the No Margin menu selection. It may seem odd to use sequenceOnGlobal() to end margining. Under the hood the PMBus command used for these is the OPERATION (0x01) command.

```
      case 2:
        pmbus->marginHighGlobal();
        break;
      case 3:
        pmbus->marginLowGlobal();
        break;
      case 4:
        pmbus->sequenceOnGlobal();
        break;
      case 5:
        pmbus->sequenceOffGlobal();
        break;
      case 6:
        pmbus->sequenceOnGlobal();
        break;
```

**Figure 22. Margin and On/Off**

Figure 23. OPERATION Command, from the LTC3880 data sheet, shows that there is no specific command to stop margining. Margining is turned off with value 0x80, which means turn on. This is why pmbus->sequenceOnGlobal() is used to turn margining off.

**Table 4. OPERATION Command Detail Register OPERATION Data Contents When On_Off_Config_Use_PMBus Enables Operation_Control**

| SYMBOL | Action | Value |
|---|---|---|
| BITS | | |
| FUNCTION | Turn off immediately | 0x00 |
| | Turn on | 0x80 |
| | Margin Low | 0x98 |
| | Margin High | 0xA8 |
| | Sequence off | 0x40 |

**Figure 23. OPERATION Command**

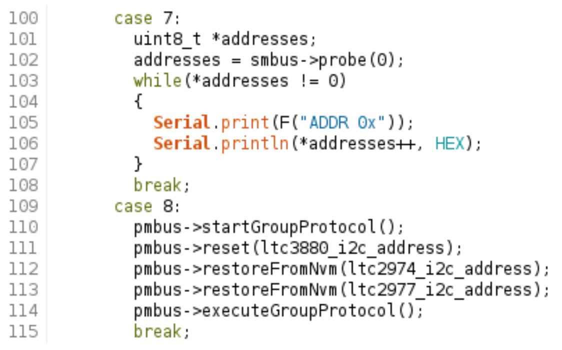
## Step 7: Bus Probing and Resetting

Probing the bus is part of the SMBus API, after all, not all devices are PMBus. Figure 24. Probe and Reset, shows that probing is a call to smbus->probe(0). The zero is the command it probes with, which is the PAGE (0x00) command. The probe will test all valid addresses and return a list of devices that are found. It will find all devices that can ACK a read command 0x00.

The reset command is less obvious. The LTC388X and LTC297X families don't reset the same way. The LTC388X devices support a MFR_RESET (0xFD) command but the LTC297X devices do not. On a LTC2977 for example, command 0xFD is MFR_TEMPERATURE_MIN, not MFR_RESET. The proper way to reset a manager is by restoring the RAM from NVM, because after the transfer, the device resets.

However, to get all devices to reset at the same time, the Group Command Protocol is used. This groups all the operations into a single transaction where all commands are actuated by the PSM devices at the STOP.

Figure 24. Probe and Reset, case 8, shows how to setup a Group Protocol transaction. The transaction is bounded by calls pmbus->startGroupprotocol() and pmbus->executeGroupProtocol().



```
      case 7:
        uint8_t *addresses;
        addresses = smbus->probe(0);
        while(*addresses != 0)
        {
          Serial.print(F("ADDR 0x"));
          Serial.println(*addresses++, HEX);
        }
        break;
      case 8:
        pmbus->startGroupProtocol();
        pmbus->reset(ltc3880_i2c_address);
        pmbus->restoreFromNvm(ltc2974_i2c_address);
        pmbus->restoreFromNvm(ltc2977_i2c_address);
        pmbus->executeGroupProtocol();
        break;
```

**Figure 24. Probe and Reset**

### Step 8: Testing

It is a good time to compile and run the application and make sure it all works.

If the application runs, but does not print sensible data, you might have made a mistake. You can use the debugging techniques below to debug it. Or if you are impatient, you can just double check:

- Addresses
- Pages
- Break Statements

## DEBUGGING

There are a few of ways to debug a Linduino PSM application, or any firmware application for that matter:

- Printing
- Spy Tools
- Debugger

This Application Note will not pursue the third option. It is typically not necessary for simple Sketches. If you want to learn more about debuggers, head to the Arduino website forums to see what tools other people use.

You have seen printing used in the examples above. You can debug by adding more print statements. However, always put strings inside the F() macro so that RAM is not used up. When printing text and numbers, separate it into two calls so the text portion is in Flash.

The PSM libraries use this technique. Errors, such as NACK and PEC Errors are printed in the command window. Therefore, adding debug printing is typically limited to the application code.

You have already seen printing with DEC. You can also use HEX and other formats. Consult the Arduino documentation for more formatting help.

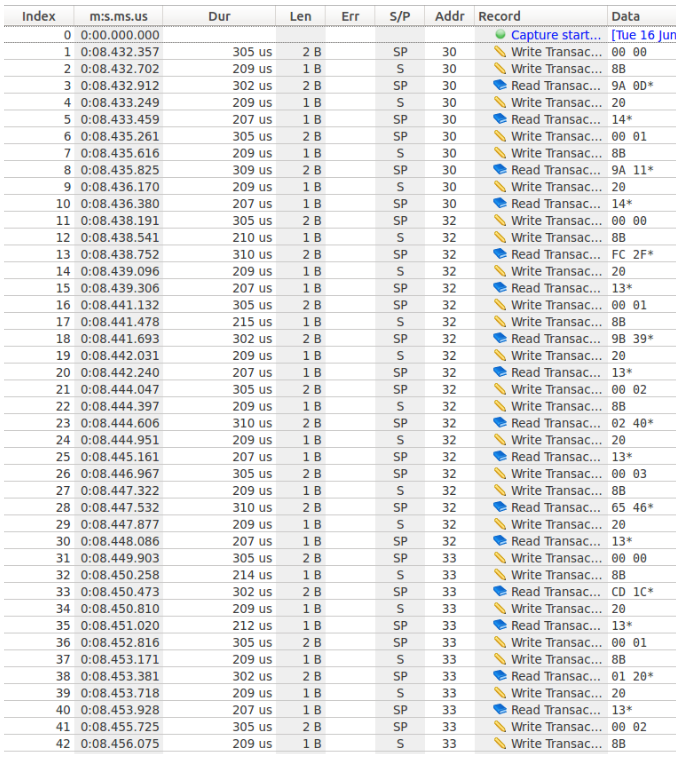
The ultimate debugger for PMBus is a spy tool. A spy tool is nice because you can see the traffic on the bus, and you can send a trace to LTC Field Application Engineers along with your code when you need help.

This Application Note will focus on data generated from the Total Phase Data Center application talking to a Total Phase Beagle. There is information on the Total Phase site to help install the Data Center Application (www.totalphase.com).

The simplest way to get started is to trace the bus using the Sketch you just created. Menu choice 3, read voltages, will be used.

Figure 25. Beagle Trace, shows the data. Let's just jump in and decipher some transactions, using the index to keep track of where we are.

At index #1 (I1) and index #6 (I6), there are two write byte transactions. In SMBus, this is Write Byte Protocol. The address is 0x30, which is the LTC3880, as can be seen in the code. The first byte is the command, which is 0x00, which is the PAGE command.



| Index | m:s.ms.us | Dur | Len | Err | S/P | Addr | Record | Data |
|---|---|---|---|---|---|---|---|---|
| 0 | 0:00.000.000 | | | | | | Capture start... | [Tue 16 Jun |
| 1 | 0:08.432.357 | 305 us | 2 B | | SP | 30 | Write Transac... | 00 00 |
| 2 | 0:08.432.702 | 209 us | 1 B | | S | 30 | Write Transac... | 8B |
| 3 | 0:08.432.912 | 302 us | 2 B | | SP | 30 | Read Transac... | 9A 0D* |
| 4 | 0:08.433.249 | 209 us | 1 B | | S | 30 | Write Transac... | 20 |
| 5 | 0:08.433.459 | 207 us | 1 B | | SP | 30 | Read Transac... | 14* |
| 6 | 0:08.435.261 | 305 us | 2 B | | SP | 30 | Write Transac... | 00 01 |
| 7 | 0:08.435.616 | 209 us | 1 B | | S | 30 | Write Transac... | 8B |
| 8 | 0:08.435.825 | 309 us | 2 B | | SP | 30 | Read Transac... | 9A 11* |
| 9 | 0:08.436.170 | 209 us | 1 B | | S | 30 | Write Transac... | 20 |
| 10 | 0:08.436.380 | 207 us | 1 B | | SP | 30 | Read Transac... | 14* |
| 11 | 0:08.438.191 | 305 us | 2 B | | SP | 32 | Write Transac... | 00 00 |
| 12 | 0:08.438.541 | 210 us | 1 B | | S | 32 | Write Transac... | 8B |
| 13 | 0:08.438.752 | 310 us | 2 B | | SP | 32 | Read Transac... | FC 2F* |
| 14 | 0:08.439.096 | 209 us | 1 B | | S | 32 | Write Transac... | 20 |
| 15 | 0:08.439.306 | 207 us | 1 B | | SP | 32 | Read Transac... | 13* |
| 16 | 0:08.441.132 | 305 us | 2 B | | SP | 32 | Write Transac... | 00 01 |
| 17 | 0:08.441.478 | 215 us | 1 B | | S | 32 | Write Transac... | 8B |
| 18 | 0:08.441.693 | 302 us | 2 B | | SP | 32 | Read Transac... | 9B 39* |
| 19 | 0:08.442.031 | 209 us | 1 B | | S | 32 | Write Transac... | 20 |
| 20 | 0:08.442.240 | 207 us | 1 B | | SP | 32 | Read Transac... | 13* |
| 21 | 0:08.444.047 | 305 us | 2 B | | SP | 32 | Write Transac... | 00 02 |
| 22 | 0:08.444.397 | 209 us | 1 B | | S | 32 | Write Transac... | 8B |
| 23 | 0:08.444.606 | 310 us | 2 B | | SP | 32 | Read Transac... | 02 40* |
| 24 | 0:08.444.951 | 209 us | 1 B | | S | 32 | Write Transac... | 20 |
| 25 | 0:08.445.161 | 207 us | 1 B | | SP | 32 | Read Transac... | 13* |
| 26 | 0:08.446.967 | 305 us | 2 B | | SP | 32 | Write Transac... | 00 03 |
| 27 | 0:08.447.322 | 209 us | 1 B | | S | 32 | Write Transac... | 8B |
| 28 | 0:08.447.532 | 310 us | 2 B | | SP | 32 | Read Transac... | 65 46* |
| 29 | 0:08.447.877 | 209 us | 1 B | | S | 32 | Write Transac... | 20 |
| 30 | 0:08.448.086 | 207 us | 1 B | | SP | 32 | Read Transac... | 13* |
| 31 | 0:08.449.903 | 305 us | 2 B | | SP | 33 | Write Transac... | 00 00 |
| 32 | 0:08.450.258 | 214 us | 1 B | | S | 33 | Write Transac... | 8B |
| 33 | 0:08.450.473 | 302 us | 2 B | | SP | 33 | Read Transac... | CD 1C* |
| 34 | 0:08.450.810 | 209 us | 1 B | | S | 33 | Write Transac... | 20 |
| 35 | 0:08.451.020 | 212 us | 1 B | | SP | 33 | Read Transac... | 13* |
| 36 | 0:08.452.816 | 305 us | 2 B | | SP | 33 | Write Transac... | 00 01 |
| 37 | 0:08.453.171 | 209 us | 1 B | | S | 33 | Write Transac... | 8B |
| 38 | 0:08.453.381 | 302 us | 2 B | | SP | 33 | Read Transac... | 01 20* |
| 39 | 0:08.453.718 | 209 us | 1 B | | S | 33 | Write Transac... | 20 |
| 40 | 0:08.453.928 | 207 us | 1 B | | SP | 33 | Read Transac... | 13* |
| 41 | 0:08.455.725 | 305 us | 2 B | | SP | 33 | Write Transac... | 00 02 |
| 42 | 0:08.456.075 | 209 us | 1 B | | S | 33 | Write Transac... | 8B |

**Figure 25. Beagle Trace**

The LTC3880 data sheet, in Table 2, shows the PAGE command. This table is a quick way to decode the Beagle data. Notice the Type column says R/W Byte. This means the register is Read/Write Byte Protocol, so both directions are supported.

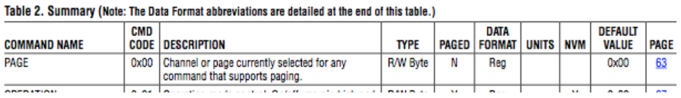


**Table 2. Summary** (Note: The Data Format abbreviations are detailed at the end of this table.)

| COMMAND NAME | CMD CODE | DESCRIPTION | TYPE | PAGED | DATA FORMAT | UNITS | NVM | DEFAULT VALUE | PAGE |
|---|---|---|---|---|---|---|---|---|---|
| PAGE | 0x00 | Channel or page currently selected for any command that supports paging. | R/W Byte | N | Reg | | | 0x00 | 63 |

**Figure 26. PAGE Command**

Looking back at I1 and I6, the second byte is a 0x00 and 0x01. The code is setting the PAGE register to 0 and 1. Referring back to Figure 21. Read Voltages, line 63, this is where the page is being set. We should see the voltage being read right after this at I2 to I5.

We see:

Write 0x8B, Read 0x9A 0x0D

Write 0x20, Read 0x14

0x8B is a READ_VOUT command. Table 2 in the data sheet shows it is an R Word protocol, and is in L16 format.

0x20 is a VOUT_MODE command. Table 2 in the data sheet shows it is an R Byte command.

The Linduino call that caused these to happen is shown in Figure 21. Read Voltages line 64.

Why does a single function call issue two transactions? To see why, we must look at the code behind the API, shown in Figure 27. Read VOUT Code.

```
vout_L16 = smbus_.readWord(address, READ_VOUT);
exp = (int8_t)

   (smbus_.readByte(address, VOUT_MODE) & 0x1F);
return math_.lin16_to_float(vout_L16, exp);
```

**Figure 27. Read VOUT Code**

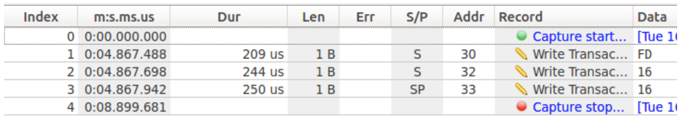
This code shows a smbus_.readWord(address, READ_VOUT) followed by smbus_.readByte(address, VOUT_MODE), and 5 bits are extracted from the mode and put into variable exp. The math conversion then converts from L16 to floating point using exponent exp.

Basically, the code for reading voltage is generic code that reads the exponent used to convert L16 to floating point. LTC388X and LTC297X devices use a different exponent. This is why there are two transactions.

*Note: Code can be written with a prior knowledge of the exponent and it will run a little faster. However, generic code will have fewer bugs and is easier on the application writer. This is a trade-off you have make when writing your own code.*

Before concluding, let's look at one more interesting transaction: the reset.

Look at Figure 28. Reset Trace, and notice there are three write transactions. In the S/P column, there are two S's and one SP. This means I1 is a Start, I2 is a Repeated Start, and I3 is a Repeated Start followed by a Stop. In addition, the addresses are different for each: 0x30, 0x32, and 0x33.



| Index | m:s.ms.us | Dur | Len | Err | S/P | Addr | Record | Data |
|---|---|---|---|---|---|---|---|---|
| 0 | 0:00.000.000 | | | | | | Capture start... | [Tue 1 |
| 1 | 0:04.867.488 | 209 us | 1 B | | S | 30 | Write Transac... | FD |
| 2 | 0:04.867.698 | 244 us | 1 B | | S | 32 | Write Transac... | 16 |
| 3 | 0:04.867.942 | 250 us | 1 B | | SP | 33 | Write Transac... | 16 |
| 4 | 0:08.899.681 | | | | | | Capture stop... | [Tue 1 |

**Figure 28. Reset Trace**

This is Group Command Protocol. All commands will be processed at the end of I3, the STOP. This correlates to code shown in Figure 24. Probe and Reset lines 110-114.

If you spend some time on your own decoding a Beagle trace, you will gain a fuller understanding of the PMBus commands of PSM devices. On the other hand, if your goal is to make code work, the PSM libraries are perfectly happy to do the heavy lifting for you.

## ADVANCED DEBUGGING WITH LTpowerPlay

Back in the introduction, it said that most systems are set and forget, and some systems have a BMC. The truth is systems with a BMC are a combination of set and forget and firmware. Why burden a BMC with complete setup responsibilities? It is far easier to program a base setup into PSM devices, and then use the BMC firmware for added value functions only. This also results in a more reliable system, because most firmware reads telemetry, margins, and makes slight voltage changes and there is no need to have it control critical functions such as sequencing or PWM frequencies which are always static.

Because LTpowerPlay is the universal tool for designing, debugging, and bringing up PSM systems, debugging firmware must contend with another PMBus master on the physical clock and data lines.

Before getting into the practical implications of two masters, it is best to review what happens when a PMBus has two masters. PMBus is based on SMBus, which includes multi-mastering.

The clock and data lines are open drain. This means any device, master or slave, can pull down a line, but cannot pull it up. There is a rule that says when a master does not pull down the data line, and it detects the data line is low, it assumes there is another master pulling the data line low, and aborts its transaction, allowing the other master to continue its transaction.

This technique is sometimes called bit dominance arbitration, which is a fancy way of saying the master asserting a zero in the data always wins.

The Linduino and LTpowerPlay (DC1613) support multi-master, and you might believe all is well with the world. However, there is one more critical consideration.

PMBus defines a PAGE command (0x00), which is like an address into data. Pages are like channels. For example, a LTC2977 can manage 8 power supplies: it has 8 channels, each addressed by the PAGE register/command.

Practically, this means to read some value like voltage, it requires two transactions: one for PAGE and one for READ_VOUT. If two masters are trying to read telemetry from the same slave a the same time, and if one master inserts a page command in between the page command and telemetry command of another master, it will read the wrong page.

When LTpowerPlay is up and running, the primary thing it does is read telemetry, which keeps its status display up to date so you can see plots of outputs, faults, and other important information. Guess what firmware typically does? It reads telemetry!

Even worse, suppose firmware performed a VID (Voltage Identification) function at boot time. What if the firmware wrote a voltage value to the wrong page because LTpowerPlay modified the PAGE register? The system might fault off, or even worse damage something. (Fortunately, the VOUT_MAX register typically prevents system damage)

The basic problem of the PAGE command is inherent in the PMBus specification. It is not unique to LTC's Power System Management devices, and one must deal with it.

There are two basic ways to allow LTpowerPlay and firmware to cohabitate and avoid the PAGE problem. The first is simply, not to let the two masters talk at the same time. The second is to use the PAGE PLUS protocol and other tricks on one or the other master.

Let's get PAGE PLUS out of the way, since it is not often used. PAGE PLUS allows an atomic transaction that includes the PAGE and the COMMAND in one transaction. Because not all devices support it, it is typically only used in special cases, so this note will not focus on PAGE PLUS and other esoteric tricks. If you have no other way to solve the problem either read the LTC PSM data sheets, or call your local Field Application Engineer for help.

More common is to prevent LTpowerPlay and firmware from talking to slaves at the same time. LTpowerPlay has a very simple way to control its behavior. Figure 29. LTpowerPlay Start/Stop shows the Telemetry Plot, which has a red square button on its toolbar.

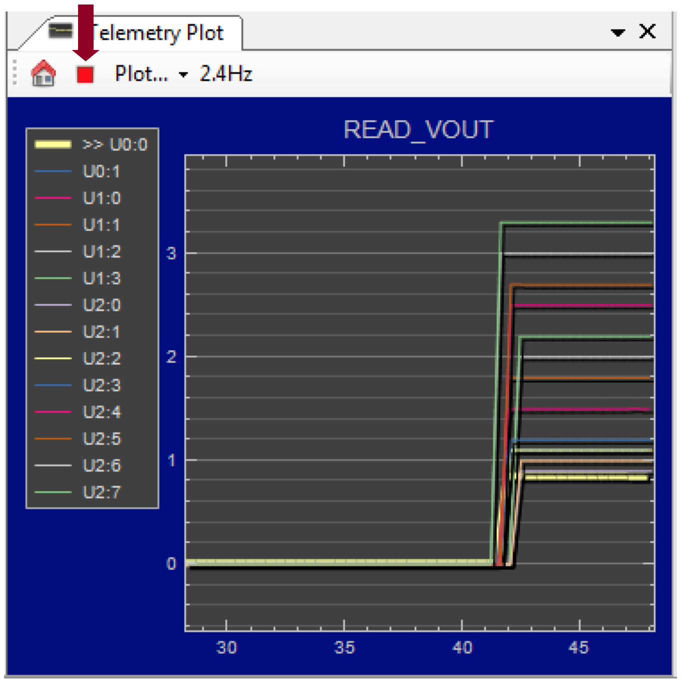


**Figure 29. LTpowerPlay Start/Stop**

When this button is pressed, it stops all telemetry and all LTpowerPlay activity on the bus. It then changes to a green arrow as shown in Figure 30. LTpowerPlay Stopped.

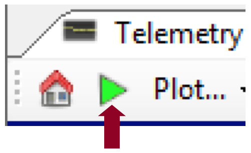


**Figure 30. LTpowerPlay Stopped**

Unfortunately, firmware does not always have a built-in mechanism to silence the bus. LTC simply recommends designing new firmware with a built-in silencing mechanism, or providing a hardware bus switch/MUX or jumpers.

Once there is a way to silence either bus master, LTpowerPlay or firmware, debugging is simply a matter of alternating between tools as necessary.

In summary, there are two choices:

1. Only allowing one master at a time to talk on the bus
2. Work through the details of PAGE PLUS and other advanced techniques with the help of LTC Field Applications Engineers

For a new design, option one is always the best choice.

## SUMMARY

Writing code for PSM devices is made simple with the Linduino PSM Sketchbook, and a Linduino board plus optional DC2294 Shield. The library has a simple API for SMBus as well as PMBus. LTpowerPlay can be still be used for debug, and a Total Phase Beagle or other Spy Tool can be attached to observe traffic on the bus.

Whether you want code to port or just want to learn how PMBus works, Linduino is a great way to get started. Once you have accelerated your learning curve, and understand the basics of PSM programming, you can up your game and use other tools with confidence.

Now that you have completed this Application Note, you may want to look at some other Sketches that come with the LTSketchbook. Try out some more advance Sketches like Fault Log Decoding or In System/Flight Update.

Linear Technology Corporation
1630 McCarthy Blvd., Milpitas, CA 95035-7417
(408) 432-1900 ● FAX: (408) 434-0507 ● www.linear.com

LT 1216 REV B • PRINTED IN USA