# Linux for Power System Management

## ABSTRACT

An open-source, `autoconfig` Linux C/C++ application for digital power system management (PSM) devices is sourced on github. It runs in user space using `/dev/i2c` for I²C communication. This application note introduces the code framework, debugging, and web-based tool support and also discusses two master solutions.

After reading the application note, the goal is to achieve successful downloading, compiling, and running the application to use for custom software design, device programming, or power system debugging.

Experience with writing Linux C applications for user-mode executables is helpful.

## INTRODUCTION

Most power system management (PSM) designs follow a set-and-forget model. When paired with LTpowerPlay™, the setup and debug of PSM devices are simplified. Also, to make matters easier, software/firmware is not required when combined with a bulk programming solution. However, many large systems require a board management controller (BMC). This raises the question: “What can software or firmware do for PSM?”

The foundation of PSM software/firmware is the PMBus, which is the basis of the SMBus, and the cornerstone of the SMBus is I²C. Building a BMC that adds value with PSM software/firmware requires some level of knowledge of each protocol or preexisting classes/codes to free the programmer from the details.

The Linux example application handles each protocol layer and provides an application programming interface (API) to facilitate writing user-space PSM software/firmware. Linux PSM is not a replacement for a BMC, but rather a set of classes and an example application that is compatible with typical BMC software/firmware.

Linux can also be used with ADI evaluation boards as a learning tool. It is quite common for engineers to copy and paste Linux code snippets into an existing application and use them. But it is also possible to reuse the whole class hierarchy, including:

- Device and rail discovery
- Command API
- Fault log decoding
- In-system programming

This application note presents the Linux code base, PSM programming, setup, and use of Linux PSM with evaluation boards as well as 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 I²C/SMBus/PMBus.[1]

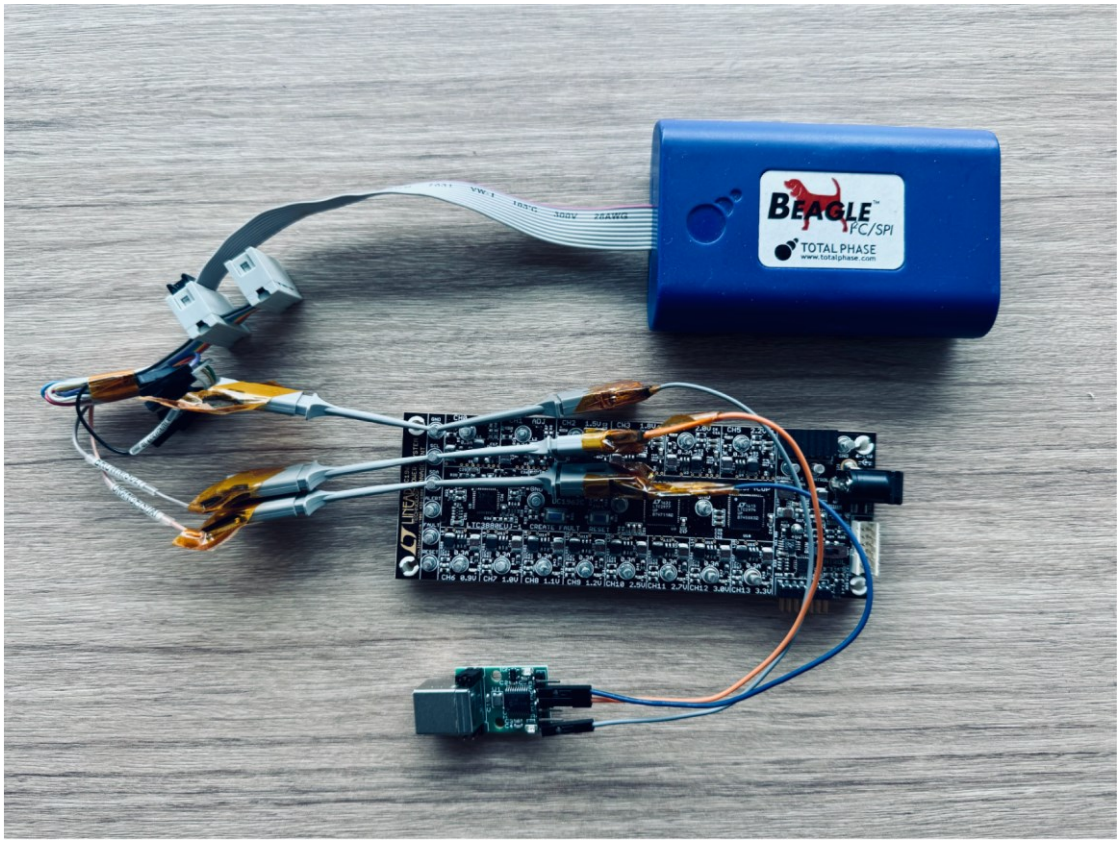


***Figure 1. Evaluation and Development Hardware***

# LINUX PSM

## Evaluation and Development Hardware

Linux PSM hardware consists of a Devantech® USB-to-I²C adapter (a small, three-wire-connected device) to connect to the PMBus, SMBus, or I²C bus of an evaluation board (middle board) or an end-product board.

For optimal learning, start with a DC1613A, a DC1962C[2] , a Devantech adapter[3] , and a Total Phase® Beagle™[4] (I²C sniffer). These devices present the programming, debugging, and learning of controllers (LTC388X/µModules) and managers (LTC297X).

*Figure 1* shows the recommended hardware, which is connected by grabber clips. To use this hardware with the Linux PSM software, connect the Devantech adapter and the Beagle to a Linux computer with two Type A-to-B USB cables (Type A connects to the small board, Type B connects to the computer). If the Beagle's USB is not connected, separate the Beagle's ribbon cable from the grabber clips to prevent interference with PMBus traffic to/from the DC1962C[5] .

## Linux Compatibility

The Linux PSM code is based on an `autoconfig` project, which has been tested with Ubuntu 24 running on a VMWare virtual machine (VM) and a Raspberry Pi using I²C on its I/O pins. This application note is based on using an Ubuntu 24 platform that runs on VMWare, and the code compiles on Ubuntu 24 with a simple `make` command.

When the Devantech adapter is plugged into the PC with a USB cable, the VM Linux instance (which may need to connect the USB device to the VM using its menu) creates a `/dev/i2c-0` file. The Linux code operates I²C through

this file. On a Raspberry Pi platform, I²C is built into the microprocessor chip and creates `/dev/i2c-N`, similar to how the embedded Linux of a BMC creates a `/dev/i2c-N`, where `N` is some number assigned by Linux. Once `N` is known, the code can be modified to match, or `N` can be passed on the command line.

**Note:** The Devantech Linux driver is not installed with Ubuntu 24 and must be manually compiled and installed.

When `/dev/i2c-N` is created, it is not writable from a user login. Rather than run the Linux application as superuser, change the permissions as follows:

```
sudo chmod 666 /dev/i2c-N
```

where `N` is the actual number. Every time the USB is plugged in, this command must be repeated, unless other measures are taken to automatically change the file permissions when plugged in.

## Installing the Devantech Driver

The driver is in the form of source code and can be obtained here:

*groeck/devantech: Devantech USB-ISS Linux kernel driver (github.com)*

Normally, the user installs `git` on their Linux box and then clones the code from the above link. The README file included with the driver code contains the installation information, which follows these steps:

1. `$ sudo make install`.
2. Blocklist the USB-ISS driver by adding: `i2c-devantech-iss to /etc/modules`.
3. Reboot Linux.
4. Perform all the steps in the README file. **Note:** If not running Ubuntu 24, proper Linux headers may require installation.
5. Connect the Devantech adapter through the USB, and ensure it creates `ls -l /dev/i2c*`.
6. Adjust the permissions so the code can write to the file.
7. If there are many I²C devices, test them to find the correct N by: `sudo i2cget 0 0x30 0`.

This action performs a read byte from the device (`N == 0`) at address `0x30` using command `0x00` (`PAGE`). The DC1962C has a PMBus device at that address. Issue this command for each `N` until one of them responds. Then try address `0x32` or `0x33` to make sure they also respond.

Once N is known, it is always the same. On a VM, N is typically 0. On a NUNC box or other embedded system, it can be any number.

## Linux PSM Application

The Linux code can be cloned (using git) from:

*analogdevicesinc/pmbus_dpsm: PMBus/SMBus example code for Digital Power System Management devices (github.com)*

To compile:

1. Change directory (`cd`) to the top-level directory and issue these commands:

```
./configure
make
```

2. Then to run it, issue these commands:

```
cd src
./LT_PMBusApp -i
```

This action runs the application in interactive mode with a text-based interface. Remove -i to get a list of command line options.

**Note:** The aforementioned action is determined by the write permissions on `/dev/i2c-N`.

The command line options are:

```
./LT_PMBusApp
[-d dev] (   // Device Path
[-p file] |   // Program with File
[-v address] | // Dump Faults
[-x address] | // Disable Fault Log
[-e address] | // Enable Fault Log
[-s address] | // Store Fault Log
[-c address] | // Clear Fault Log
[-i])   // Menu
```

The device path can be combined with any other option, but the other options are used one at a time. The device path option is for providing `N`, when it is not the default `0`.

## Menu Operation

To operate the menu system, enter the number at the left of the list of menu selections, and press ***Enter***. Start with menu item 1, which includes selections such as `1 - Read All Voltages` and `4 - Sequence On/Off`. Start with `1 - Read All Voltages`.

The application detects all the PSM devices and maintains an internal structure of them, including the number of pages of each device. Selecting a voltage records the values for the whole tree, address by address, and page by page.

## In-System Programming

In-system programming is a way to code PSM devices by directly writing the settings to EEPROM without affecting operation of the device. *AN-166: In Flight Update with Linduino* describes the process and how to export isphex files (in system programming hex files) from LTpowerPlay.

Program devices using an isphex file with the following command:

```
./LT_PMBusApp -d N -p file
```

This is very useful for programming all the devices at one time in a manufacturing environment when Linux is used for testing.

## Modifying the Code

The code has two entry points: a `main()` function called once, and a `loop()` function called forever in a loop.

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

Modifying the application is simply making changes to the case statements in the code, using a provided API. The functions (API) in the code that issue PMBus commands come from a separate class hierarchy and have simple names that sound like what the code should to do.

For example:

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

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

For practice, make a few simple changes. As an example, add a menu item to read and print output power. Try calling this function from the menu:

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

Test this on the LTC3880 at address `0x30` on the DC1962C. To see it work, add a resistive or a current load to Channel 0 on the DC1962C and verify it matches what the code prints.

## PSM PMBus Classes

The PMBus classes reside in the src directory. The class hierarchy is layered: starting with $I^2C$, then SMBus, and lastly, PMBus. There is a number conversion API to convert values from L11/L16 (PMBus formats) to/from the 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. If the final environment is C, don't worry! Simple means using the C++ class without a lot of memory overhead, or removing the class wrapping and using it as pure C very easily. The C++ wrapper simplifies the application code and makes it easier for nonprogrammers.

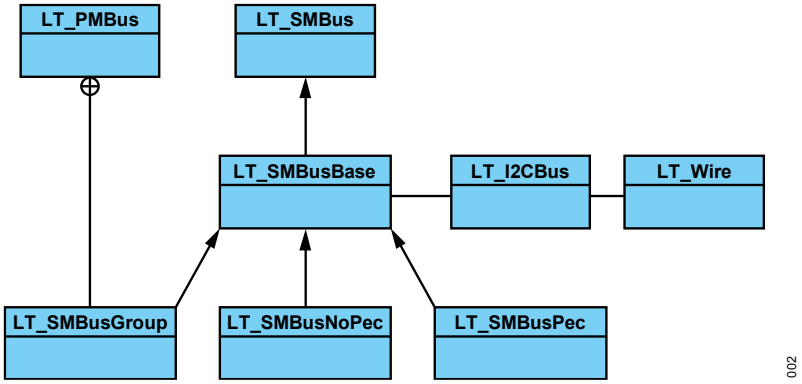


*Figure 2. LT_PMBus Class Diagram*

For programmers who want 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 to aid porting, and the I²C classes are a wrapper around the IO to `/dev/i2c-N` files. If the code were ported to a non-Linux platform, the I²C classes would be rewritten to use registers or a preexisting library.

## Using the PMBus Classes

The API can be used without understanding classes. All that is required are just a few imports and static variables, and the code is ready for action.

The most important ones are:

```
#include <LT_SMBusPec.h>
#include <LT_SMBusNoPec.h>
#include <LT_PMBus.h>
#include <LT_PMBusMath.h>
```

and

```
static LT_PMBus *pmbus = new LT_PMBus(smbus);
```

## Debugging

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

- Printing
- Spy Tools
- Debugger

This application note does not cover the debugger option.

Print statements are a good debugging tool because the code compiles to a command line application. The PSM libraries use this technique for common errors, such as NACK and PEC, because they are printed on the command line. Copying and pasting existing error message code can be modified to print messages from key places in the code.

The ultimate debugger for PMBus is a spy tool (Beagle). A spy tool is nice because it can see the traffic on the bus, and a trace is email-ready to send to ADI field application engineers along with the code to get help.

This application note focuses on data generated from the Total Phase Data Center application talking to a Total Phase Beagle. Visit the Total Phase site for information to install the Data Center application (*www.totalphase.com*).

The simplest way to get started is to trace the bus using the application. Menu choice 3, read voltages, is used as an example.

| 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 |

005

***Figure 3. Beagle Trace***

*Figure 3* 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 defined as the write byte protocol. The address is `0x30`, which is the LTC3880, as in the code. The first byte is `0x00`, which is the `PAGE` command.

**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 |

006

***Figure 4. PAGE Command***

Table 2 in the LTC3880 data sheet shows the PAGE command. This table provides a quick way to decode the Beagle data. Notice the Type column says "R/W Byte." This means the register follows the read/write byte protocol, so both directions are supported.

Reviewing I1 and I6, the second byte is a 0x00 and 0x01. The code is setting the PAGE register to 0 and 1.

Then 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 code behind reading READ_VOUT and VOUT_MODE is:

```
vout_L16 = smbus_.readWord(address, READ_VOUT);
exp = (int8_t)
    (smbus_.readByte(address, VOUT_MODE) & 0x1F);
return math_.lin16_to_float(vout_L16, exp);
```

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 the exp variable. The math conversion then changes L16 to a floating point using exponent exp.

Basically, the generic code reads both the voltage and the exponent to convert L16 to a floating point. The LTC388X and LTC297X devices use a different exponent. This is the reason for two transactions.

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

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

| 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( |

008

***Figure 5. RESET Trace***

Look at *Figure 5.* Notice there are three write transactions. In the S/P column, there are two Ss 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.

This is group command protocol. All commands are processed at the end of I3, the stop.

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

## Solutions for 2 Controller Systems

As stated earlier, most systems are set and forget, while others have a BMC. The truth is systems with a BMC are a combination of set and forget and firmware. So 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 and margins, and makes slight voltage changes. 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 controller on the physical clock and data lines.

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

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

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

Since the Linux PSM and LTpowerPlay (DC1613), both of which support arbitration, it is easy to 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 eight power supplies: it has eight 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 controllers are trying to read telemetry from the same target at the same time, and if one controller inserts a page command in between the page command and telemetry command of another controller, it will read the wrong page.

When LTpowerPlay is up and running, its primary function is reading the telemetry to keep its status display up to date with its plots of outputs, faults, and other important information. Guess what firmware typically does? It reads telemetry!

Even worse, suppose firmware performed a voltage identification (VID) 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 with the PMBus specification. It is not unique to ADI's power system management devices, which requires attention.

There are two ways to allow LTpowerPlay and firmware to cohabitate and avoid the PAGE problem. The first is simply not to let the two controllers talk at the same time. The second is to use the PAGE PLUS protocol and other methods, such as global address and PAGE commands, on one controller or the other.

Let's examine PAGE_PLUS first 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 used in special cases only, so this article does not focus on PAGE PLUS and other esoteric methods. If there is no apparent solution, either read the ADI PSM LTC3887 and LTC2977 data sheets, or call a local field application engineer for help.

The more common way, however, is to prevent LTpowerPlay and firmware from talking to targets at the same time. LTpowerPlay has a very simple way to control its behavior. *Figure 6* shows the telemetry plot, which has a red square button on its toolbar.

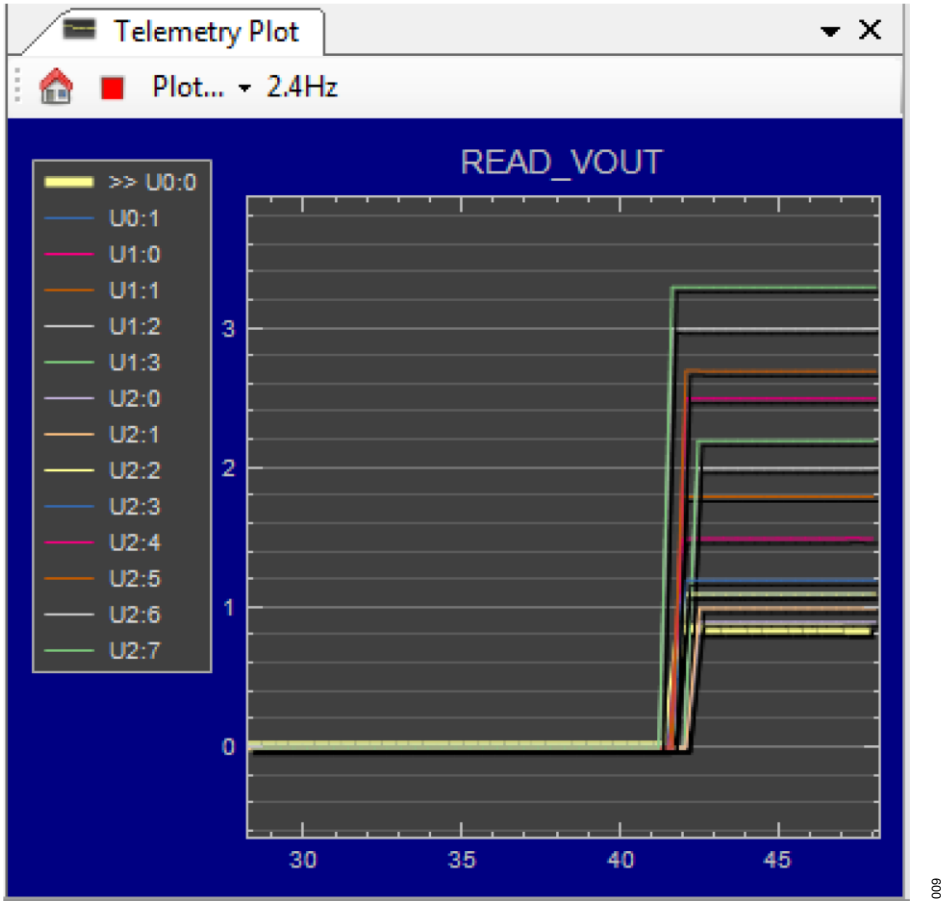


***Figure 6. LTpowerPlay Start/Stop***

When this button is pressed, it stops all telemetry and LTpowerPlay activity on the bus. It then changes to a green arrow as shown in *Figure 7*.

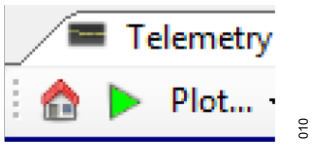


***Figure 7. LTpowerPlay Stopped***

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

Once the bus controllers LTpowerPlay and firmware are silent, debugging is simply a matter of alternating between tools as necessary.

In summary, there are two choices:

- Allowing only one master at a time to talk on the bus.
- Work through the details of PAGE PLUS and other advanced techniques with the help of ADI field applications engineers.

For new designs, the first option is the better choice.

## Summary

The Linux PSM code and a Devantech adapter simplify programming for PSM devices. The code has a simple API for both SMBus as and PMBus. LTpowerPlay can still be used for debugging, while a Total Phase Beagle or other Spy Tool can be attached to observe traffic on the bus.

Linux PSM is a great way to get started whether coding to port or learning how PMBus works. After gaining a basic understanding of PSM programming, improving your skills and using other tools with greater confidence are within reach!

# NOTES

[1] See www.pmbus.org for standards documents.

[2] DC1962C Evaluation Board | Analog Devices

[3] Devantech USB to I2C Interface Adapter | Acroname

[4] Beagle I2C/SPI Protocol Analyzer - Total Phase

[5] The Beagle is not required, it just aids learning because through observation of PMBus traffic to the DC1962C. A DC1613A is highly advised for debug and restoring the default DC1962C configuration.

[6] The DC1962C (Power System Management Demo Board (Power Stick)) has a LTC3880 at address 0x30, a LTC2975 at address 0x32, and a LTC2977 at address 0x33.