April 2017

In Flight Update with Linduino

Michael Jones

INTRODUCTION TO INFLIGHT UPDATE

Inflight Update is a method of modifying the stored settings of LTC Power System Management (PSM) devices, including application of the new settings, to a live system[1].

Inflight Update is a two-stage process: EEPROM is modified first, and then EEPROM is copied to RAM. During stage one, a Board Management Controller sends new settings directly to the EEPROM via PMBus while the devices are operating normally, without impacting operation. During stage two, all devices switch to the new configuration via a reset or a power cycle[2].

Decoupling "programming settings" from "application of settings" allows the EEPROM programming mechanism to stage, validate, and recover from programming failure without interruption of delivered power. This minimizes system downtime because all rails are power cycled only once per update.

Inflight Update solves several technical and business problems. Fast product development increases the probability of field problems. For example, the supervisors might require more margin to reduce false positives. An FPGA image update might require small changes to a supply voltage. Inflight Update can also reduce inventory of pre-programmed devices by programming during manufacturing.

Inflight Update is a method unique to LTC Power System Management devices with a supporting infrastructure: LTpowerPlay®, Linduino reference code, and application support.

This application note will explain the Inflight Update process at a high level, plus present the Linduino reference code and and how to port it to another platform.

Note 1. All LTC388X devices and all LTC297X devices, except LTC2978, support Inflight Update. The Inflight Update code will program a LTC2978, but it will power off rails and use RAM to program the EEPROM. All non-LTC2978 devices do not have to power off rails because there is a direct path from PMBus to EEPROM that the LTC2978 does not have.

Note 2. Reset also power-cycles the outputs.

Note 3. The BMC must not be powered by PSM devices that will be programmed with Inflight Update.

INFLIGHT UPDATE PROCESS

The general dataflow of Infight Update is a simple linear progression:

1. Modify the settings of an engineering unit using LTpowerPlay.
2. Export an In System Programming (ISP) file using LTpowerPlay.
3. Transport ISP file/data to a Board Management Controller (BMC) (microcontroller) using Ethernet, disk, or compiling it into its code.
4. BMC programs all PSM devices using ISP data and PMBus[3].
5. BMC resets system to apply new settings or power is cycled.

The ISP file contains a list of instructions that, when applied via the PMBus, directly modify register / command values stored in EEPROM without disturbing operation. The BMC code that applies the instructions to PMBus is independent of the LTpowerPlay tool that generates the instructions.

As an implementer of Inflight Update, you are responsible for:

1. Porting the Inflight Update code
2. Implementing data delivery
3. Adding safety/recovery measures
4. Validating and testing the system

Linear Technology is responsible for:

1. Maintaining and updating the generator
2. Adding support for new PSM products
3. Maintaining and updating LTpowerPlay

This application note will focus on applying the instructions, not the instructions themselves. If you have questions about the instructions themselves or the instruction generator, please contact your local LTC Field Applications Engineer (www.linear.com/contact).

All example code can be found in the Linduino Sketchbook in the library/LTPSM_InFlightUpdate directory (www.linear.com/linduino).

CREATING DATA WITH LTpowerPlay

LTpowerPlay generates the data that feeds the Inflight Update code. Selecting File | Export | Programming File | Export to In System Programming [.isp] Hex File… raises a dialog box that accepts a path to a file, and generates the file with .isp extension[4].

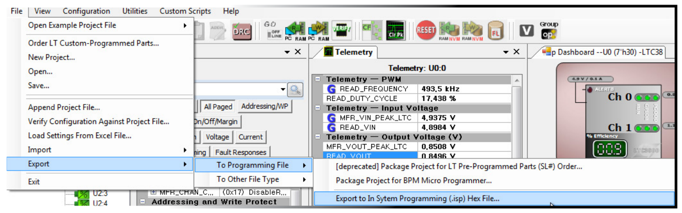
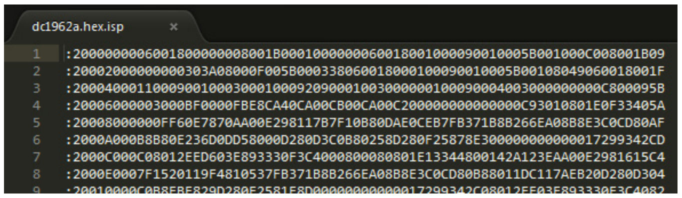

File
View
Configuration
Utilities
Custom Scripts
Help
Open Example Project File
Order LT Custom-Programmed Parts...
New Project...
Open...
Save...
Append Project File...
Verify Configuration Against Project File...
Load Settings From Excel File...
Import
Export
Exit
To Programming File
To Other File Type
[deprecated] Package Project for LT Pre-Programmed Parts (SL#) Order...
Package Project for BPM Micro Programmer...
Export to In Sytem Programming (.isp) Hex File...
Telemetry
Dashboard

Figure 1. Export ISP File

Figure 2 shows an example isp FILE from a DC1962. This data is copied to a sketch, or to your BMC.


dc1962a.hex.isp

```
:2000000006001800000008001B00010000000600180010000900010005B001000C008001B09
:20002000000000303A08000F005B000338060018000100090010005B00108049060018001F
:200040001100090010003000100092090001003000000010009000400300000000000C800095B
:2000600000003000BF0000FBE8CA40CA00CB00CA00C200000000000000000C93010801E0F33405A
:2000800000000FF60E7870AA00E298117B7F10B80DAE0CEB7FB371B8B266EA08B8E3C0CD80AF
:2000A000B8B80E236D0DD58000D280D3C0B80258D280F25878E30000000000000017299342CD
:2000C000C08012EED603E893330F3C4000800080801E13344800142A123EAA00E2981615C4
:2000E0007F1520119F4810537FB371B8B266EA08B8E3C0CD80B88011DC117AEB20D280D304
:20010000C0B8FBE829D280F2581E8D000000000000017299342C08012EE03E893330F3C4082
```

Figure 2. ISP File Contents

RUNNING THE SKETCH

The Linduino Sketchbook has an example sketch for the DC1962[5] that uses the library mentioned above. A walkthrough of the example sketch will give you a general feel for how Inflight Update works, and starter code for experimenting with other DC boards.

Figure 3 shows how to navigate the Arduino menus to open the sketch.

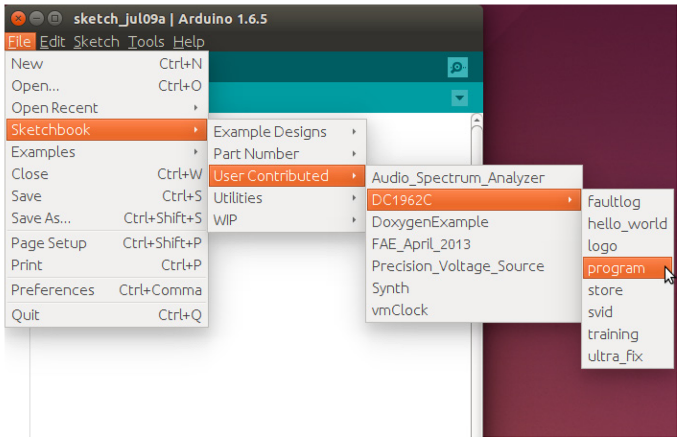
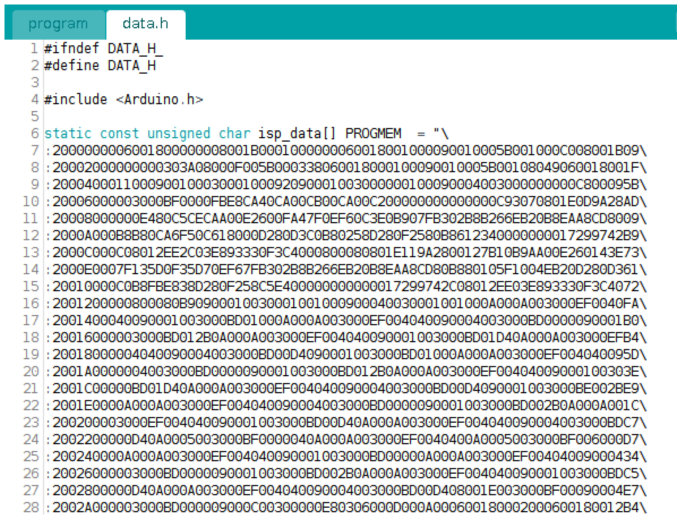


Figure 3. Example Sketch

The data.h file contains the same data found in the *.isp file, but formatted differently. Figure 4 shows that the data is pasted into an array, and each line of data ends with a "\" continuation character.

This data represents both the contents of EEPROM and the programming steps to apply the data to the devices in the original DC1962 project. Because the data is in the data.h file, it will be compiled into the sketch.


program
data.h

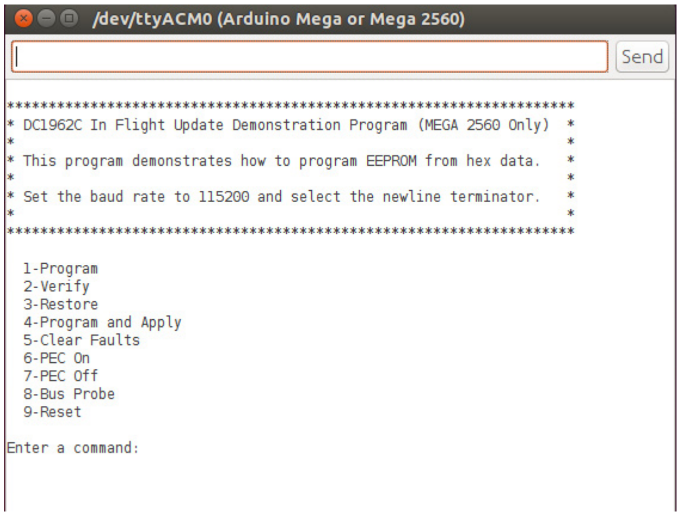
```
#ifndef DATA_H_
#define DATA_H

#include <Arduino.h>

static const unsigned char isp_data[] PROGMEM  = "\
:2000000006001800000008001B00010000000600180010000900010005B001000C008001B09\
:20002000000000303A08000F005B000338060018000100090010005B00108049060018001F\
:200040001100090010003000100092090001003000000010009000400300000000000C800095B\
:2000600000003000BF0000FBE8CA40CA00CB00CA00C200000000000000000C93070801E0D9A28AD\
:20008000000E480C5CECAA00E2600FA47F0EF60C3E0B907FB302B8B266EB20B8EAA8CD8009\
:2000A000B8B80CA6F50C618000D280D3C0B80258D280F2580B861234000000000017299742B9\
:2000C000C08012EE2C03E893330F3C4000800080801E119A2800127B10B9AA00E260143E73\
:2000E0007F135D0F35D70EF67FB302B8B266EB20B8EAA8CD80B880105F1004EB20D280D361\
:20010000C0B8FBE838D280F258C5E40000000000000017299742C08012EE03E893330F3C4072\
:200120000080080B9090001003000100100090004003000100100A000A003000EF0040FA\
:20014000400900010030000BD01000A000A003000EF00404009000400300 0BD00000900001B0\
:200160000003000BD012B0A000A003000EF00404009000100300 0BD01D40A000A003000EFB4\
:200180000040400900040030 00BD00D4090001003000BD01000A000A003000EF004040095D\
:2001A000000040030 00BD00000900010030 00BD012B0A000A003000EF004040090001 00303E\
:2001C00000BD01D40A000A003000EF0040400900040030 00BD00D409000100300 0BE002BE9\
:2001E0000A000A003000EF004040090004003000BD000009000100300 0BD002B0A000A001C\
:20020000 3000EF00404009000100300 0BD00D40A000A003000EF004040090004003000BDC7\
:2002200000D40A00050030 00BF000004 0A000A003000EF0040400A00050030 00BF006000D7\
:20024000 0A000A003000EF004040090001003000BD00000A000A003000EF00404009000434\
:200260000030 00BD0000090001003000BD002B0A000A003000EF004040090001003000BDC5\
:200280000D40A000A003000EF00404009000400300 0BD00D408001E003000BF00090004E7\
:2002A0000003000BD000009000C00300000E80306000D000A000600180002000600180012B4\
```

Figure 4. Top of data.h

Note 4. ISP means In System Programming file. This is a more general term than Inflight Update that includes programming devices without direct PMBus to EEPROM support.

Note 5. A DC1962 has a LTC3880, LTC2974, and LTC2977 on it.

Once the sketch is run, a menu is presented with 9 options.



```
*****************************************************************
* DC1962C In Flight Update Demonstration Program (MEGA 2560 Only)  *
*                                                                *
* This program demonstrates how to program EEPROM from hex data.   *
*                                                                *
* Set the baud rate to 115200 and select the newline terminator.   *
*                                                                *
*****************************************************************

  1-Program
  2-Verify
  3-Restore
  4-Program and Apply
  5-Clear Faults
  6-PEC On
  7-PEC Off
  8-Bus Probe
  9-Reset

Enter a command:
```

Figure 5. Sketch Menu

Option 1 (Program) applies the data to the EEPROM of all three devices on the DC1962. Option 2 (Verify) compares the data with the contents of the EEPROM to ensure they match. Option 3 (Restore) forces the devices to load EEPROM data into RAM.

The supply rails only power cycle during option 3, which transfers EEPROM to RAM. When data in the EEPROM is copied to RAM, all devices automatically power down their rails before the copy and then power them back on after the copy. Disconnecting power from the DC1962 will accomplish the same thing[6].

```
LT_SMBusNoPec *smbusNoPec = new LT_SMBusNoPec();
LT_SMBusPec *smbusPec = new LT_SMBusPec();

NVM *nvm = new NVM(smbusNoPec, smbusPec);

bool worked = nvm->programWithData(isp_data);
pmbus->resetGlobal();

delete(nvm);
delete(smbusPec);
delete(smbusNoPec);
```

Figure 6. Sketch Code

Note 6. For general help with PSM Sketches, refer to application note AN153 – Linduino for Power System Management.

The sketch code is very simple and a condensed version is shown in Figure 6. Two lines do all the work: `programWithData`, and `resetGlobal`. The first line does the actual programming, the second line causes the transfer from EEPROM to RAM which cycles power.

Notice that the code does not worry about addresses, device types, polling, etc. The ISP data handles all of these issues for you. You port the code, and then it just works.

SUMMARY

Inflight Update is a complete end-to-end solution for programming Power System Management devices in a working system. A software engineer ports LTC provided code to their Board Management Controller. This code applies data generated by LTpowerPlay. LTC maintains the generation code to ensure programming is robust and future proof, ensuring that the code ported to the BMC is maintenance free. This allows a Board Management Controller to be used across a broad range of products without changes, even when different products use different Power System Management devices.

This end-to-end solution enables a flexible model for end users. The solution can be used for:

- Manufacturing Programming
- Prototype Programming
- Field Upgrades

INTRODUCTION TO INFLIGHT UPDATE CODE

LTC maintains two versions of Inflight Update Code:

- Linduino
- Linux

The Linduino code is downloadable from www.linear.com/linduino. The Linux code is available through your local FAE.

The remainder of this Application Note will walk through the Linduino version of the code. The only significant difference between the two sets of code is how memory is managed.

If you don't care about how the code works, you can stop reading.

OVERVIEW OF DATA FLOW

Before jumping into the details, let's review a schematic representation of the whole process.

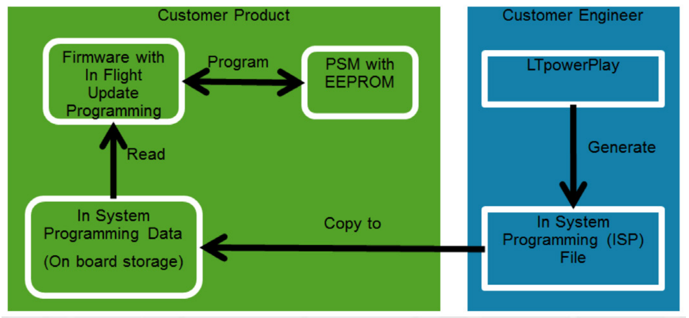

Customer Product
Firmware with In Flight Update Programming
Program
PSM with EEPROM
Read
In System Programming Data (On board storage)
Copy to
Customer Engineer
LTpowerPlay
Generate
In System Programming (ISP) File

Figure 7. Data Flow

The process begins by making changes to a LTpowerPlay project and then exporting an ISP file. The ISP file is copied to the product, which usually means to a Board Management Controller or other micro controller's EEPROM or Disk Drive. The Inflight Update code reads the ISP data and sends commands over the PMBus to the EEPROM. These commands program and verify the EEPROM.

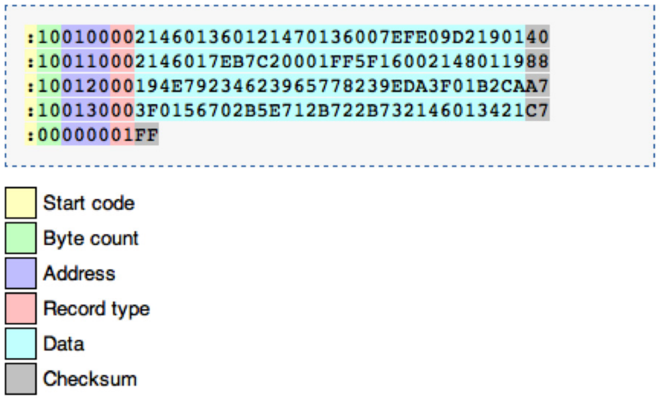
Once the verify succeeds, the system is reset or power cycled safely. If there is a failure, the system retries the programming, while the system continues to operate.

When the reset or power cycle completes, the new settings created with LTpowerPlay will become effective. This is because the reset or power cycle causes the RAM to be loaded with the new EEPROM data.

In this App Note, a Linduino is used to demonstrate this process, so the ISP data is copied into the program code and compiled in.

ISP FILE FORMAT

An ISP file is a list of instructions encapsulated in an Intel Hex File. Figure 8 shows an example.



```
:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

🟨	Start code
🟩	Byte count
🟪	Address
🟥	Record type
🟦	Data
⬜	Checksum

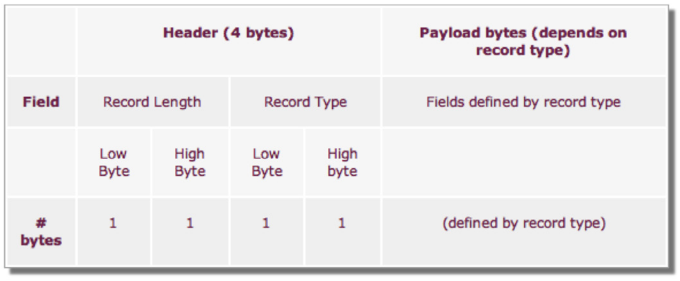
Figure 8. Example Intel Hex File

A record begins with a header (the Start code) and ends with a CRC (the Checksum). The record interior contains a count, address, record type, and hex data. ISP files contain two record types[7]:

- Data Record (00)
- End of File Record (01)

The data inside the Intel Hex Data Records contains another set of LTC defined records. These records are blocks of binary data that exactly map to C Structures. Furthermore, the binary data is packed, meaning there are no empty or unused bytes in the C Structures.

Figure 9 shows the LTC record structure. Like Intel Hex, it has a length, type, and data. However, there is no need for a CRC as the Intel Hex already has a CRC mechanism.



	Header (4 bytes)				Payload bytes (depends on record type)
Field	Record Length		Record Type		Fields defined by record type
	Low Byte	High Byte	Low Byte	High byte	
# bytes	1	1	1	1	(defined by record type)

Figure 9. LTC Defined Record Structure

Note 7. Other Intel Hex data types are not used.

The length is in the first byte so that code can grab all the bytes for a record without knowing anything about the record content. This allows the code to "chunk" the data into record sized blocks. These blocks may occupy anywhere from part of a data record to several data records of the Hex file. The second byte has the record type, so that code can map the block of data to the correct C Structure. Each C structure is an instruction for the BMC to apply to the PMBus, and the complete list of C structures in the ISP file represent the reconfiguration of all PSM devices in the original LTpowerPlay file[8].

Doc: For record details see
http://www.linear.com/solutions/5710

PROCESSING ARCHITECTURE

How the records are processed matters because of potentially limited resources in the BMC. In particular, memory may be as small as 2k of RAM. For limited memory systems, data must be processed incrementally. For larger memory systems, the entire intermediate results may be held in memory, leading to simpler code debugging, but this App Note will focus on small memory systems.

The Inflight Update reference code uses a two level parsing scheme implemented as a "pipe and filter" architecture.

In a "pipe and filter" design, data can be pushed or pulled through the pipeline. The Inflight Update reference code uses a pull design. The main record processor (C structure processor) pulls data through the pipeline. At the head of the pipeline inserts an end-of-data record (C structure) so that the main record processor knows when to stop. This is called an "on demand" implementation.

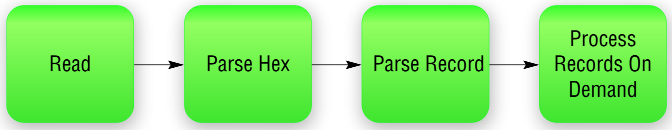


Figure 10. LTC Pipe and Filter

Note 8. There is no correlation between the size of Intel Hex Record data and size of LTC records.

Figure 10 shows the filters used in the reference code. Each box/filter is a C function. The first function called is "Process Records On Demand," which pulls records from "Parse Record," which pulls data from "Parse Hex," which reads data, etc.

Terminology: Each stage is called a "filter" in the literature. However, a "filter" is just a processing step that delivers data to a requesting pipe. Each filter/step is implemented as a C function.

CODING PIPE AND FILTER

Coding filters is simple. When a filter function is called, it requests data from an input function, processes it, and returns transformed data to its caller. The input function is passed to the filter function as a pointer when it is called. The filters are the stable portion of a pipe and filter design.

Connecting the filters with pipes is like gluing the pipeline together. Pipes are the modifiable portions of the pipeline. Pipes are functions that wrap around a downstream filter, and their pointer is passed to the upstream filter. Any time a new pipeline is configured, filter functions are reused, and pipe functions are recoded.

Looking at an example will clarify how this is done.

Figure 11 shows a pipeline that reads data from a file. The filters are the functions:

- `processRecordsOnDemand`
- `parse_record`
- `parse_hex`
- `filter_terminations`
- `read`

And the pipes are the functions:

- `get_record`
- `get_record_data`
- `get_filtered_data`
- `get_hex_data`

```
uint8_t get_hex_data (void)
{
  uint8_t c = ‘\0’;

  c = pgm_read_byte_near (icpFile +
                          flashLocation++);
  if (‘\0’ == c)
    return c;
  else
    return 0;
}

uint8_t filter_terminations(
                  uint8_t (*get_data)(void))
{
  uint8_t c = ‘\0’;

  c = get_data();
  while (c == ‘\n’ || c == ‘\r’)
    c = get_data();
  return c;
}

uint8_t get_filtered_hex_data(void)
{
  return filter_terminations(get_hex_data);
}

uint8_t get_record_data(void)
{
  return parse_hex(get_filtered_hex_data);
}

pRecordHeaderLengthAndType get_record(void)
{
   return parse_record(get_record_data);
}

processRecordsOnDemand(get_record);
```

Figure 11. Gluing a Pipeline Together

One way to tell them apart when looking at code is by the function names and signatures. The filters all accept a function pointer (for a pipe) and return data, plus they have names that are actions. Pipes all accept a void and return data, and they have names that begin with get. The prefix “get” reflects that the pipeline is pulling data from the end of the pipeline from the front of the pipeline. A push pipeline would prefix the functions with “put.” Note that filters designed for pulling will not work for pushing.

```
uint8_t filter_terminations(
                  uint8_t (*get_data)(void))
{
  uint8_t c = ‘\0’;

  c = get_data();
  if (c == ‘:’)
  {
    Serial.print(“.”);
  }
  while (c == ‘\n’ || c == ‘\r’)
    c = get_data();
  return c;
}

uint8_t detect_colons(uint8_t
                        (*get_data)(void))
{
   uint8_t c;

   c = get_data();
   if (c == ‘:’)
   {
     Serial.print(“.”);
   }
  return c;
}

uint8_t get_hex_data_debug(void)
{
   uint8_t c = ‘\0’;
   c = pgm_read_byte_near(icpFile +
                          flashLocation++);

 if (‘\0’ != c)
   return c;
 else
   return 0;
 }

uint8_t get_hex_data_debug_(void)
{
  return detect_colons(get_hex_data_debug);
}

uint8_t get_filtered_hex_data_debug(void)
{
  return
    filter_terminations(get_hex_data_debug_);
}

uint8_t get_record_data_debug(void)
{
  return
     parse_hex(get_filtered_hex_data_debug);
}

pRecordHeaderLengthAndType
                    get_record_debug(void)
{
   return
     parse_record(get_record_data_debug);
}
```

Figure 12. Debug Hex Filter

A library of filters creates a toolkit by function composition. An application designer glues a pipeline together by writing pipe functions. Furthermore, if filters are added to the library with standardized inputs and outputs, the pipeline can be reconfigured at runtime.

For example, if there are multiple ways to read the hex data, there may be multiple read filters, one for each data source, and the code may compose them at runtime based on the source of data. Filters might also be specialized for space/time trade-offs, or provide tracing and debugging support.

Figure 12 shows how to insert a filter called `detect_colons` on line 40. This filter prints a dot on the Linduino display every time a colon passes through it.

Now let's look at the record processing, which is the last filter in the pipeline. Going back to Figure 11 looking at line 32, notice that the pipe function `get_record_data` is passed to `parse_record`. The filter function `parse_record` is the filter that will generate the C Structure/Record.

```
pRecordHeaderLengthAndType parse_record(
    uint8_t (*get_data)(void))
```

Figure 13. Parse Record Signature

Take a look at the signature of `parse_record` in Figure 13. This is the standard way of passing a function pointer in C. Inside the `parse_record` code, `get_data` is called to get data from its pipe. Because `get_data` is passed as a function pointer, it can accept any function with the proper signature, which must return a `uint8_t`. This is the same signature used by the `print_record` filter, which prints the record information and just passes along the record to the caller.

Looking back once more at Figure 11 line 34. Notice the call to `processRecordsOnDemand`[9]. This function applies the C structure Records to the PMBus, thus eventually writing the data to the EEPROM of each device on the bus.

If you need to debug the actual record processing, there are some `#defines` at the top of the file[10].

```
#define DEBUG_SILENT 0
#define DEBUG_PROCESSING 0
#define DEBUG_PRINT 0
```

Figure 14. Processing Debug #defines

`DEBUG_SILENT` just turns the processor (`processRecordsOnDemand`) off so you can debug the pipeline. For example, you could use `print_record` to observe records, but they would not do anything to the PMBus.

`DEBUG_PROCESSING` prints the record information, but also blocks the use of PMBus.

`DEBUG_PRINT` prints the record information and lets the PMBus transactions through so that devices are programmed.

ISP FILE COVERAGE

ISP files (hex data) can cover one or more devices on the PMBus. The number of devices per file is controlled by LTpowerPlay, which creates an ISP file from an LTpowerPlay project file. If you want one ISP file per device, you create one LTpowerPlay project per device. However, generally it is better to make one file for the whole bus[11] because it reduces bookkeeping errors.

Inflight Update is designed to process one file of data at a time. It does not care how many devices the file covers. However, if there is more than one file for a bus and if it matters whether they are all applied before the next reset or power cycle, you must code your implementation to run Inflight Update for each device before reset or power cycling, and you must ensure that if power is lost between files, the software prevents power on until the job is complete.

Note 9. LTSketchbook/libraries/LTPSM_InFlightUpdate/main_record_processor.cpp

Note 10. Change 0 to 1 to activate.

Note 11. Or a segment if the bus is segmented with I^2C multiplexers.

BROKEN CONFIGURATIONS

All PSM devices store one or more CRC values in their EEPROM. During reset, the device calculates the CRC of the data in EEPROM and compares it to the CRC value in EEPROM. If they do not match, the device does not power up any rails and it issues a CML fault.

LTC388X devices with CRC errors default to address 0x7C. LTC297X devices with CRC errors default their base address to the data sheet default[12]. The BMC can't even talk to the PSM devices in this state, because they are not at their correct address, and there may be more than one device at an address. They might even be on top of a non-PSM device.

The probability of losing power while updating EEPROM is very low, and the failure behavior is safe for the loads, but for Inflight Update systems where the system is not in the factory, we can't ignore the problem. There must be a way to recover and repair the system without human interaction.

RECOVERY METHODOLOGY

Because the CRC failure is caused by a power loss, the BMC will reboot on next power up. Therefore, the proper place to detect a CRC error is during start up. This assumes the BMC has power even when the PSM devices can't power up. Hopefully your design has a safe supply to the BMC or you can still influence the design before it is too late.

Warning: The BMC must not be powered by PSM devices that will be programmed with Inflight Update!

When the BMC starts up, it must check for CRC errors, and if any device has errors, it must apply Inflight Update. Inflight Update will work even when there are CRC errors and the devices have defaulted to address 0x7C or the default base address.

Note 12. In rare cases the base address will be random if the EEPROM partially loads data during CRC checking.

Note: Inflight Update and recovery depend on certain design practices. All devices on the bus must have ASEL resistors that guarantee that when all devices on the bus are programmed to a common base address, every device has a unique address. The hex data in the ISP file contains a common base address that will be used to program the system, and the addresses set by the ASEL pins must match the addresses in the hex data. See App Note AN152 for details on proper address configuration.

Robustness: Inflight Update even works when there are CRC errors and the BMC can't control PSM devices in the normal way.

```
uint8_t* LT_PMBus::bricks(uint8_t* addresses,
                      uint8_t no_addresses) {
  uint8_t i,j;
  bool found;
  uint8_t len;

  uint8_t* addresses_on_bus =
                       smbus_.probe(0x00);
  for(len=0;
       addresses_on_bus[len]!='\0'; ++len);

   for (i = 0; i < no_addresses; i++) {

     found = false;
     for (j = 0; j < len; j++) {
       if (addresses[i] ==
                     addresses_on_bus[j]) {
         found = true;
         break;
       }
     }
     if (found && (1 << 4) !=
       (smbus_.readByte(addresses[i],
        STATUS_CML) &
         (1 << 4)))
         addresses[i] = 0x00;
   }

   i = 0;
   while (i < no_addresses) {
     if (addresses[i] == 0x00) {
       j = i + 1;
       while (addresses[i] == 0x00 &&
              j < no_addresses) {
         addresses[i] = addresses[j];
         addresses[j] = 0x00;
         j++;
       }
     }
     i++;
   }


  return addresses;
}
```

Figure 15. Detector for Devices with CRC Mismatch

Recovery starts by probing the bus to determine if there are any devices with CRC mismatches. The reference code in Figure 15 applies three tests. The first test probes the bus with a simple command looking for an ACK. This results in a list of responders stored in `addresses_on_bus` at line 6. Responding is not enough to prove the CRC is ok, but it's enough to know a device exists at an address. The second test compares the addresses from the probe with the expected addresses to remove addresses we don't care about at lines 10-19. The final test reads the STATUS_CML register and checks if there is a memory error, designated by a 1 in bit position 4 at lines 20-25. Any address that does not pass all three tests is returned from the `bricks()` function.

Note: The reason this code returns a list of bad addresses is so that the caller can choose whether to program all devices using Inflight Update, or a subset. This choice depends on whether the final implementation uses one ISP file (hex) for the whole bus, subsets of the bus, or individual devices.

If there are any bricked or missing devices, the BMC runs Inflight Update.

There is always the chance that power fails each time Inflight Update is applied. The chance is remote in a well tested system, but if you are paranoid, use scratch pad memory in the BMC to count number of retries and prevent EEPROM wear out. Each data sheet has a max number of times it can be programmed (see data sheets), and if the system really can get in a loop, it can exceed the max very quickly.

EEPROM should not be programmed on devices that are too hot. In general, a system should not reprogram when above 85°C. If your system is going to run that hot, the BMC should power down rails and let them cool off. Or, anticipate that a retention problem will cause a CRC mismatch and lead to an automatic run of Inflight Update on the following power cycle, which will allow some cooling time.

Checking for CRC mismatches on a power cycle does double duty in protecting against downtime. If at power up, the BMC holds off Inflight Update until the temperature is less than 85°C, it will be very robust.

Tip: For Inflight Update to be robust in a deployed design it must take into account the possibility of power loss during programming and recover on the next power up event. This means the board controller must have a persistent store for the ISP (hex) data, and detect devices that fail their CRC check.

```
LT_SMBusNoPec *smbusNoPec = new LT_SMBusNoPec();
LT_SMBusPec *smbusPec = new LT_SMBusPec();

NVM *nvm = new NVM( smbusNoPec, smbusPec);

bool worked = nvm->programWithData(isp_data);
wait_for_nvm();
pmbus->resetGlobal();

delete(nvm);
delete(smbusPec);
delete(smbusNoPec);
```

Figure 16. Calling Processing Code

Data Processing

The top-level data processing code is simple. Create SMBus objects and pass them to a NVM class. Call `programWithData` on the NVM object passing the data. Wait for the operation to complete with `wait_for_nvm`, then reset all devices on the bus. Finally, clean up by deleting the objects.

Your application code that implements Inflight Update will be this simple after porting the code. If you want to understand the code internals, continue on, otherwise skip ahead to the section Porting Guide.

Hex Parsing

Processing always begins by calling `reset_parse_hex()`. This resets the static variables. The `parse_data_position` variable keeps track of which byte in the data is returned next. The `parse_data_length` variable keeps track of the size of a LTC C structure.

```
void reset_parse_hex()
{
  parse_data_length = 0;
  parse_data_position = 0;
}
```

Figure 17. Resetting the Parse

When the first call is made to `parse_hex`, lines 8-52 in Figure 18, it will parse a full Intel Hex Record and place the contents in the `parse_data` variable. Thereafter, it will skip this step and return the data byte by byte, one byte per call. When the data runs out, it parses another record. It does this as long as the `record_type == 0` on line 28. A 0 is a data record. When the record type is 1, it creates an LTC C structure for end of data, which is code 0x22. This end of data record is not in the ISP data.

The front of the record is parsed in lines 13-27. First it runs to the colon and stops, passing over any linefeed or carriage returns. Then it parses the `byte_count`, address, and `record_type` by chunking the data and converting it. These are fields shown in Figure 8.

If it finds a data record (0), it puts the data into memory at lines 28-34. There are two hex characters per byte in the original hex data, so they are assembled into a little array and converted to a value by calling "hex to integer": `httoi(data)` at line 34.

When an end of data record (1) appears, lines 42-49 create an LTC C Structure end-of-data record, the record processor will stop requesting data and flow stops. Before the next run of Inflight Update, the BMC must call `reset_parse_hex`.

For both record types, the `parse_data_position` and `parse_data_length` variables are updated to manage the processing.

```
1: uint8_t parse_hex(uint8_t (*get_data)(void))
2: {
3:   uint8_t     start_code;
4:   uint16_t    byte_count;
5:   uint16_t    record_type;
6:   uint16_t    i;
7:   char        data[5];

8:   if (parse_data_position ==
9:       parse_data_length) {

10:     start_code = 0x00;
11:     while (start_code != ':')
12:       start_code = get_data();

13:     data[0] = get_data();
14:     data[1] = get_data();
15:     data[2] = '\0';
16:     byte_count = httoi(data);

17:     data[0] = get_data();
18:     data[1] = get_data();
20:     data[2] = get_data();
21:     data[3] = get_data();
22:     data[4] = '\0';
23:     address = httoi(data);

24:     data[0] = get_data();
25:     data[1] = get_data();
26:     data[2] = '\0';
27:     record_type = httoi(data);

28:     if (record_type == 0) {
29:       for(i = 0; i < byte_count; i++) {
30:         data[0] = get_data();
31:         data[1] = get_data();
32:         data[2] = '\0';
33:         parse_data[i] = httoi(data);
34:       }

35:       data[0] = get_data();
36:       data[1] = get_data();
37:       data[2] = '\0';
38:       int crc = httoi(data);

39:       parse_data_position = 0;
40:       parse_data_length = byte_count;
41:     }
42:     else if (record_type == 1) {
43:       parse_data[0] = 4;
44:       parse_data[1] = 0;
45:       parse_data[2] = 0x22;
46:       parse_data[3] = 0;
47:       parse_data_position = 0;
48:       parse_data_length = 4;
49:     }
50:   }
51: return parse_data[parse_data_position++];
52: }
```

Figure 18. Parse Hex

Warning: If `parse_hex` is called too many times, it will process beyond the end of the array feeding the call to `get_data` function that was passed in on line 1. This means the caller must watch for the LTC C structure end of data record and stop processing.

LTC C Structure Parsing

Parsing the LTC C Structures follows the following steps:

1. Get the record size
2. Build a block of data to match
3. Create the structure by manipulating a pointer

```
1: pRecordHeaderLengthAndType parse_record(
2:                 uint8_t (*get_data)(void)) {
3:   uint32_t  header;
4:   uint8_t  *data;
5:   uint16_t size;
6:   uint16_t  pos;
7:   pRecordHeaderLengthAndType record;

8:   header =
9:     (uint32_t)get_data() << 0 |
10:    (uint32_t)get_data() << 8 |
11:    (uint32_t)get_data() << 16 |
12:    (uint32_t)get_data() << 24;

13:   record = (pRecordHeaderLengthAndType)
14:     &header;
15:   size = record->Length;
16:   uint8_t *record_data = getRecordData();

17:   record_data[0] = (header >> 0) & 0xFF;
18:   record_data[1] = (header >> 8) & 0xFF;
19:   record_data[2] = (header >> 16) & 0xFF;
20:   record_data[3] = (header >> 24) & 0xFF;

21:   if (size <= getMaxRecordSize())
22:   {
23:     for (pos = 0; pos < size - 4; pos++)
24:       record_data[pos + 4] = get_data();
25:   }

26:   record = (pRecordHeaderLengthAndType)
record_data;
27:   return record;
28: }
```

Figure 19. Parse Record

The header of a record is 4 bytes, so `get_data`, the passed in function (pipe) that calls `parse_hex` (filter), is called 4 times on lines 8-12 of Figure 19 and shifted into place. This is then typecast to a `pRecordHeaderLengthAndType` so that it can be accessed as a C Structure. Line 15 then pulls out the size of the record and line 16 allocates enough memory to hold it and the type.

On lines 17-20, the header is packed into the allocated memory, and lines 21-25 fill in the remaining data, all packed without spaces. Now the data is in a variable of type `uint8_t*` and includes the header and all the remaining bytes of the C Structure, but it is not yet a C Structure.

Finally, the data is typecast to `pRecordHeaderLengthAndType` and returned. What is returned is a record pointer to the header, but the data behind it holds the entire record. This means the caller can typecast this to the final record pointer type and the data will be in the right place.

Be Careful: Things can go wrong with typecasts so be on the lookout for trouble. C is a powerful but dangerous language.

LTC C Structure Processing

We are now at the final phase, the processing of the C Structures, and close to the stuff you may need to add to the processor.

```
1: uint8_t processRecordsOnDemand(
2: pRecordHeaderLengthAndType (
3:   *getRecord)(void)) {

4: pRecordHeaderLengthAndType record_to_process;
5: uint16_t recordType_of_record_to_process;
6: uint8_t successful_parse_of_record_type =
7:   SUCCESS;

8: while (
9:  (record_to_process = getRecord()) != NULL &&
10:   successful_parse_of_record_type == SUCCESS)
11: {
12:   recordType_of_record_to_process =
13:     record_to_process->RecordType;

14:  switch( recordType_of_record_to_process)
15:  {
```

Figure 20. LTC C Structure Processing

The C Structures (Records) are processed in a while loop with a switch to process each record type. The while loop on line 8 calls `getRecord()`, which returns a pointer to a `pRecordHeaderLengthAndType`. This structure has just enough information to pull out the RecordType, which is a number. The switch then uses the RecordType to pick the correct processing function.

```
uint8_t
recordProcessor___0x01___
processWriteByteOptionalPEC(
t_RECORD_PMBUS_WRITE_BYTE* pRecord);
```

Figure 21. Example Processing Function

Each function that processes a RecordType takes a pointer to the matching C Structure. This means the act of calling the function typecasts the `pRecordHeaderLength-AndType` pointer to the proper record. In this example, it is typecasting to `t_RECORD_PMBUS_WRITE_BYTE*`. Each function will interpret the record and issue PMBus commands or other special transactions that support Inflight Update.

Special Transactions

There are two kinds of special transactions:

1. Memory handling
2. Events

Memory handling transactions allow the record generator to ask the record processor to store a block of data for later use. This economizes the algorithm by passing a block of data for the EEPROM that is used both to program and verify programming. The reference code already does this for you.

Events are hooks the firmware engineer uses for adding special code. For example, if some logic controlled a PSM device's WP pin, it could change its state to allow programming.

The events are handled in one function: `uint8_t recordProcessor___0x18___processEvent(t_RECORD_EVENT* pRecord)` found in file `main_record_processor.cpp`. This function has a switch statement that can be modified. The event IDs (cases) are:

- `BEFORE_BEGIN`
- `BEFORE_INSYSTEM_PROGRAMMING_BEGIN`
- `SYSTEM_BEFORE_PROGRAM`
- `INSYSTEM_CHIP_BEFORE_PROGRAM`
- `SYSTEM_BEFORE_VERIFY`
- `INSYSTEM_CHIP_BEFORE_VERIFY`
- `INSYSTEM_CHIP_AFTER_VERIFY`
- `SYSTEM_AFTER_VERFY`
- `AFTER_DONE`

```
META DATA EVENT (BEFORE_BEGIN):
META DATA EVENT (BEFORE_INSYSTEM_PROGRAMMING_BEGIN):
META DATA EVENT (SYSTEM_BEFORE_PROGRAM):
META DATA EVENT (INSYSTEM_CHIP_BEFORE_PROGRAM):
META DATA EVENT (SYSTEM_BEFORE_VERIFY):
META DATA EVENT (INSYSTEM_CHIP_BEFORE_VERIFY):
META DATA EVENT (INSYSTEM_CHIP_AFTER_VERIFY):
META DATA EVENT (INSYSTEM_CHIP_BEFORE_PROGRAM):
META DATA EVENT (INSYSTEM_CHIP_BEFORE_VERIFY):
META DATA EVENT (INSYSTEM_CHIP_AFTER_VERIFY):
META DATA EVENT (INSYSTEM_CHIP_BEFORE_PROGRAM):
META DATA EVENT (INSYSTEM_CHIP_BEFORE_VERIFY):
META DATA EVENT (AFTER_DONE):
```

Figure 22. Example Event Order (DC1962)

By adding some print statements, and running Inflight Update on a DC1962[13] using the Linduino "program" Sketch, we see the above events printed on the display. It may seem a little unusual, so some explanation is required. `BEFORE_BEGIN` and `AFTER_DONE` are the over all bracketing. Each device has an `INSYSTEM_CHIP_BEFORE/AFTER_PROGRAM_VERIFY` pair. But there are two extra events: `SYSTEM_BEFORE_PROGRAM/VERIFY`. These events come before any programming or verify, but only one time.

Note 13. A DC1962 has a LTC3880, LTC2974, and LTC2977 on it.

If you need events before and after `PROGRAM/VERIFY` use the `CHIP` versions. If you just need to know when `PROGRAM/VERIFY` begins and ends in overall, use the `non-CHIP` versions.

For example, if the code will enable the WP pin, you would enable it with `INSYSTEM_CHIP_BEFORE_PROGRAM` and disable it with `INSYSTEM_CHIP_BEFORE_VERIFY`. But if you wanted to enable multiple devices at the same time, enable at `SYSTEM_BEFORE_PROGRAM`, and disable at `AFTER_DONE`.

EEPROM Protection: You do not need to add code to disable the write protect in the device's register. Inflight Update will do that automatically. Only the WP pin requires special handling. If the BMC can't control the WP pin, and it is high, the BMC has it hands tied behind its back and Inflight Update can't operate.

Porting Guide

Porting the Linduino PSM Inflight Update code is not complicated, but there are a few decisions that have to be made, and a few places things can go wrong.

Where to Slice the Code

The Inflight Update record-processing loop depends on the Lindino LT_SMBus library. This library has two layers:

- SMBus
- I^2C

The LT_SMBus library is layered as C^{++} classes, and either one of these classes can be re-implemented as a layer around a pre-existing library. The classes are also simple enough that they can be converted to C, but most programmers use a C^{++} compiler to compile C, and the classes have very little overhead.

The record processor only makes calls to the SMBus layer. Therefore, you can slice between the SMBus layer and the record processor, or between the I^2C layer and the SMBus layer.

Structure Packing

All the C Structures that hold the LTC records are packed. Most compilers have a pack pragma. This pragma is used with the structure definitions to ensure the compile does not leave any space between items in the structure. If the structures were not packed, it would not be possible to take a block of data and just cast its pointer to a C structure pointer.

Endianess

The data from the hex file has high and low bytes. When these are passed to a SMBus API they must go over the PMBus in the proper order. Anything that causes the bytes to become reversed will cause problems.

Pointers

The Inflight Update code casts pointers to structures. If casting causes a misalignment of the data at the base of the structure, because of some address alignment issue, it will cause problems. In some cases you may have to realign the block of data. In general, copying bytes to a static data array will work fine.

Callbacks

Implement events for special requirements. These events will always be generated when LTpowerPlay creates an ISP file. Do not depend on the order of any other records because they can change as a result of LTpowerPlay maintenance. Do not ignore records or otherwise get creative by inserting behaviors in them. Just allow Inflight Update to process records per the reference code and stick to events for special code.

Debugging

The record processor has some `#defines` that enable printing. If your processor can print to RS232 or a command line you can enable these to observe the behavior of the record processing. One trick is to enable printing only and run Inflight Update on a Linduino, and on your platform, and then compare the two outputs. This is a good way to prove that the hex and record processing is working properly.

If this processing looks good attach a Beagle[14] to a Linduino and allow Inflight Update to program your board, run your ported code, then compare the traces. If there is a mistake in your SMBus library, it will be easy to spot.

If the data feeding the record processor is sending in junk, put debugging filters in the pipeline to see what is happening.

Note 14. Beagle is a I^2C Spy tool from Total Phase. www.totalphase.com

SUMMARY

LTpowerPlay can export an In System Programming (ISP) file that contains all settings for a complete system. The ISP file can be transported to the target system's Board Management Controller via Ethernet or other means, where the system can apply the data to the PSM devices using Inflight Update. Inflight Update programs the EEPROM of all PSM devices without disturbing operation of the system, which can be reset or power cycled at a later time.

If power loss occurs during programming, the Board Management Controller can detect improperly programmed devices, and repair them with Inflight Update without any risk to the load, because the PSM devices will not power up if they are not properly programmed. Detection of incompletely programmed PSM devices and their repair by the Board Management Controller occurs without any human intervention.

Implementation of Inflight Update also enables programming PSM devices in production and performing remote field updates. Remote field updates are rare, but returning boards to the factory for reprogramming is expensive and time consuming.