Technical Article

A Guide to Battery Fast Charging—Part 2

Franco Contadini, Staff Engineer, and
Alessandro Leonardi, Account Manager, Field Sales

In "A Guide to Battery Fast Charging—Part 1," we covered some of the challenges involved in designing fast-charging battery systems. By implementing fuel gauge functionality in the battery pack, original equipment manufacturers (OEMs) can design smart fast chargers that increase system flexibility, minimize power dissipation, ensure safe charging/discharging, and improve the overall user experience. In Part 2, we'll explore the implementation details of a fast-charging system with parallel batteries using evaluation kits and a Raspberry Pi board.

Evaluating 1S2P Architectures

Evaluating a simple charging system and testing its functionality can typically be done with an evaluation kit. These kits include all the necessary hardware and software applications, as well as graphical user interface (GUI)-based tools and APIs, to configure charging systems.

However, complex systems that require multiple cells are correspondingly more complex to evaluate. Complex systems may have several devices that need to be characterized. Developers will need to write some software code to read the signals generated from different system parts, analyze them, and take action. Consider a two Li+ cells in a parallel battery fast charging system using the MAX17330. As described in the data sheet, the MAX17330 can be used to charge and control two Li+ cells simultaneously. This system requires two MAX17330 ICs each managing one Li+ cell, and a buck converter (such as the MAX20743) with the capability to change its output voltage on-the-fly.

A microcontroller is required to configure and manage battery charging as well as to handle communication between the two ICs. Because it is a commonly used platform for system testing, we chose a Raspberry Pi board using Python as the programming language. The Raspberry Pi manages communications over I^{2}C and logs important system parameters useful for evaluation and debugging, including charge current, battery voltage, and battery state of charge (SOC). These values are stored in an Excel file to enable offline analysis.

Testing the 1S2P Architecture

This section shows how the charger and fuel gauge (MAX17330) are tested. It also describes the real performance that can be expected from parallel charging. For the most flexibility and control, the device is programmed by a microcontroller using I^{2}C.

Figure 1 shows the 1S2P system architecture and the connections that are needed to evaluate the charging of two cells in parallel. The Raspberry Pi controls the three EVKITs: one MAX20743EVKIT (buck converter) and two MAX17330EVKITs (charger + fuel gauge). Data is logged in an Excel file.

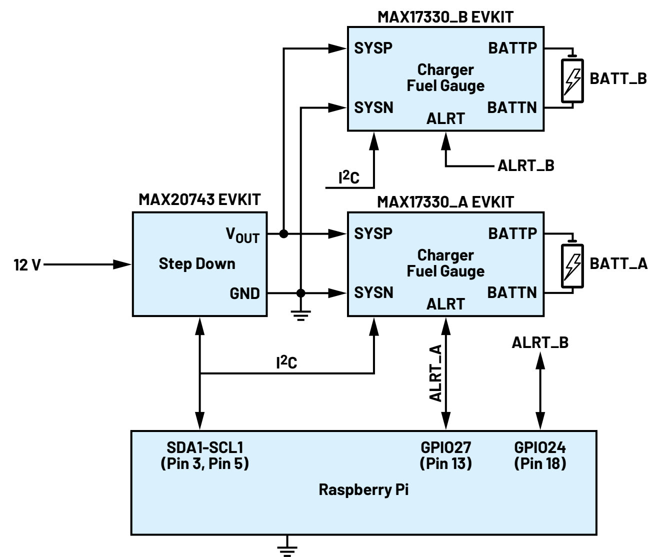

Figure 1. A 1S2P charging system evaluation architecture using Raspberry Pi.

The GUI-based, MAX17330 EV Kit Software is available and can be downloaded from the MAX17330 product page under the Tools and Simulations tab. It can be used to generate initialization files (.INI) for the MAX17330 using the Configuration Wizard (select from the Device tab). The INI file contains the register initialization information for the device in a register address/register value format. This is the file used by the microcontroller to configure the MAX17330 register by register.

The MAX17330EVKIT data sheet details the different steps required to generate the initialization file. The configuration, shown in Figure 2, is used to begin parallel charging. Next, step charging is enabled (see Figure 3). Figure 4 shows the expected step charging profile based on the step charging configuration found in Figure 3.

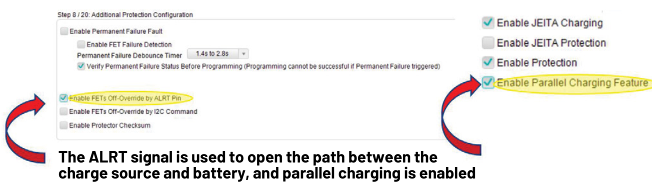

Figure 2. Configuring the MAX17330 for parallel charging.

Figure 3. Enable step charging.

The MAX20734 buck converter is used to increase the voltage applied to the two MAX17330EVKITs when needed. The MAX20734 buck converter changes the output voltage according to the value of the internal register at address 0x21. The buck converter can be controlled via I²C; a class in Python has been written to do so.

Finally, as shown in Figure 5, the MAX20743EVKIT output voltage divider is modified for an output range from 3 V to 4.6 V (using the values R6 = 4K7 and R9 = 1K3).

Table 1. Conversion Output Voltage Based on Register 0x21 for the MAX20743

0x21 Register Value	Voltage
0x014E	3 V
0x0150	3.05 V
0x0158	3.1 V
0x015C	3.15 V
0x0162	3.2 V
0x0166	3.25 V
0x016E	3.3 V
0x0172	3.35 V
0x0178	3.4 V
0x017C	3.45 V
0x0182	3.5 V
0x0188	3.55 V
0x018E	3.6 V
0x0192	3.65 V
0x019E	3.7 V
0x01A4	3.75 V
0x01A9	3.8 V
0x01AE	3.85 V

Table 1. (continued)

0x21 Register Value	Voltage
0x01B4	3.9 V
0x01BA	3.95 V
0x01BF	4 V
0x01C4	4.05 V
0x01CB	4.1 V
0x01D1	4.15 V
0x01D6	4.2 V
0x01DC	4.25 V
0x01E2	4.3 V
0x01E8	4.35 V
0x01ED	4.4 V
0x01F3	4.45 V
0x01F8	4.5 V
0x01FE	4.55 V
0x0204	4.6 V

From Table 1, we can extract the curve:

$$Register = 0 \times 014e + \left(\frac{x-3}{0.1 \times 11}\right)$$

where x is the voltage we want to apply at the output. While this approach will have a slight error, it is a good way to estimate the desired value of the register from the voltage.

Powering Up and Initialization

When the MAX17330 is first connected to a battery, default register value settings force the IC into a shutdown state. To wake the device, press the PKWK button. This will short the temporary protection MOSFETs and wake up both MAX17330EVKITs in this way.

Next, the Raspberry Pi needs to communicate via I²C with all three devices. Carefully initialize the I²C hardware to avoid device address conflicts. By default, the two MAX17330EVKITs use the same I²C address. The first step is to change the address of one of the two fuel gauges.

The MAX17330 has both volatile and nonvolatile registers, with nonvolatile registers identified with the "n" prefix. This also results in a pair of node addresses, 6Ch (volatile registers) and 16h (NV registers).

There are two ways to change device node addresses on the MAX17330:

- Set the nPackCfg NV register using the I²CSid field. This change can be set using the Configuration Wizard. See Table 3.
- The I2CCmd register allows dynamic changes to the I²C bus. See Table 4.

For ease of use, we use the second way to change the address so that the same INI file can be used to initialize both devices. Generating settings that can be shared by the two devices simplifies the configuration of the devices and eliminates the potential for user error when the address must be entered manually.

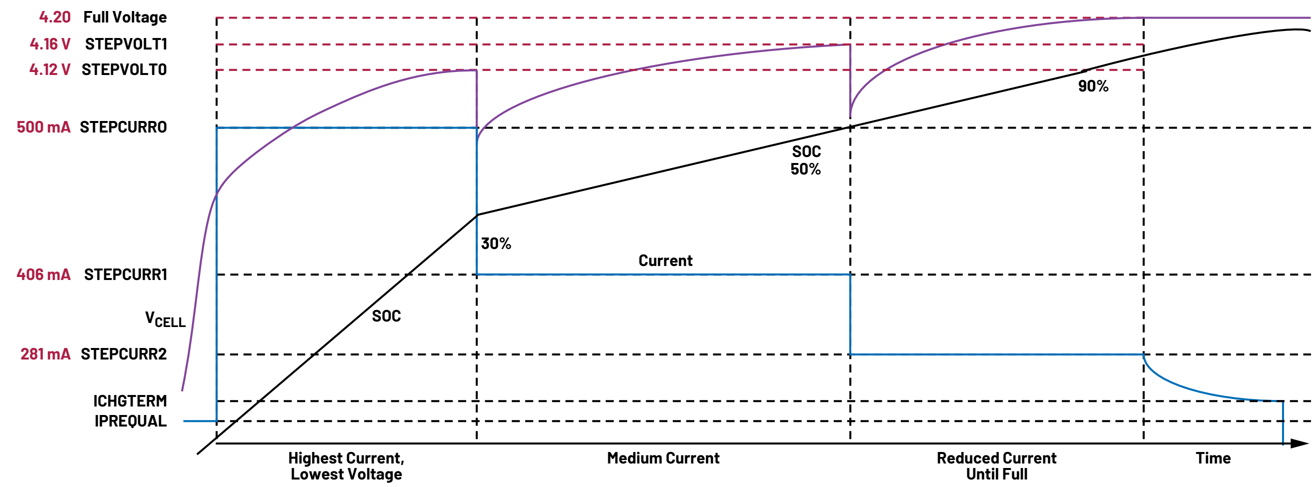

Figure 4. An expected step charging profile based on step charging configuration in Figure 3.

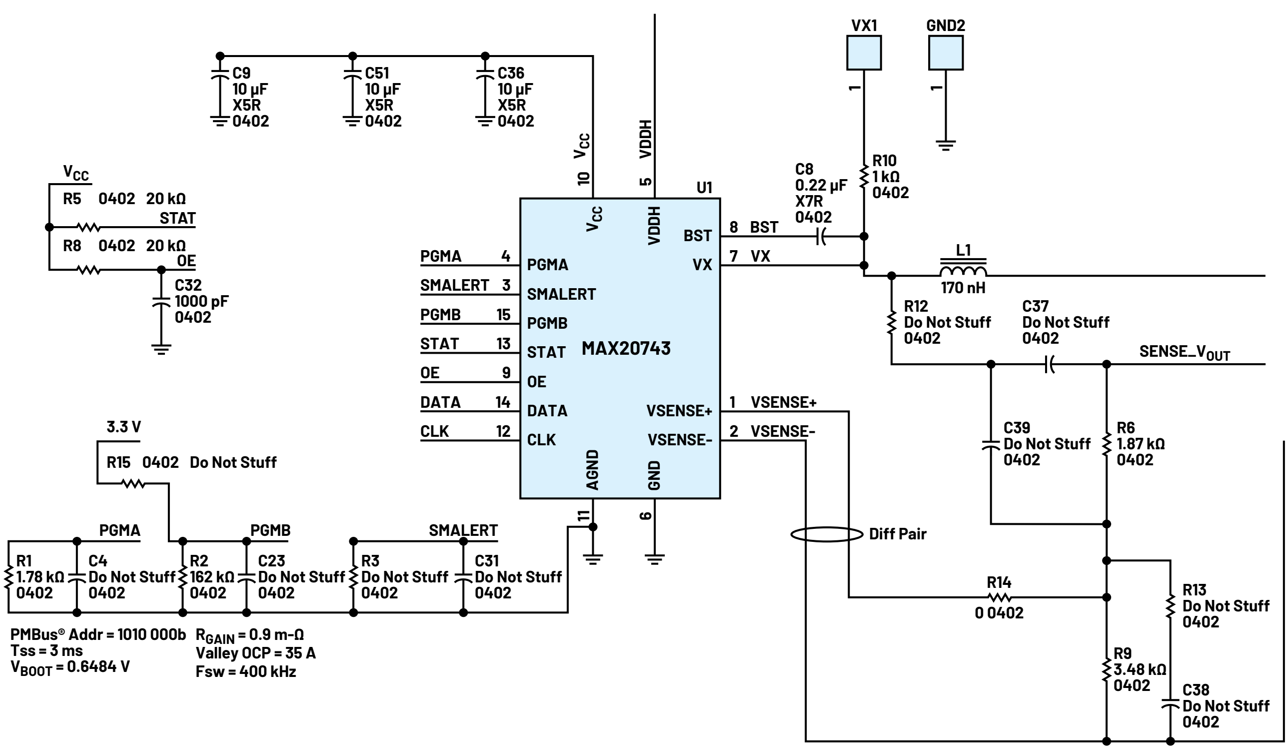

Figure 5. The output voltage divider has been modified for an output range of 3 V to 4.6 V (with R6 = 4 K7 and R9 = 1 K3).

Table 2. MAX17330 Registers

Register Page	Lock	Description	2-Wire Node Address	2-Wire Protocol	2-Wire External Address Range
00 h		Modelgauge M5 EZ data block	6 channels	I^2C	00 h – 4 Fh
01 h – 04 h	Lock 2				
05 h – 0Ah		Reserved			
0 Bh	Lock 2	Modelgauge M5 EZ data block (continued)	6 channels	I^2C	B0 h – BFh
0 Ch	SHA	SHA memory	6 channels	I^2C	C0h – CFh
0 Dh	Lock 2	Modelgauge M5 EZ data block (continued)	6 channels	I^2C	D0h – DFh
0 Eh – 0 Fh		Reserved			
10 h – 17 h		SBS data block	16 channels	SBS	00 h – 7 Fh
18 h – 19 h	Lock 3	Modelgauge M5 EZ nonvolatile memory block	16 channels	I^2C	80 h – EFh
1 Ah – 1 Bh	Lock 1	Life logging and configuration nonvolatile memory block			
1 Ch	Lock 4	Configuration nonvolatile memory block			

Table 3. nPackCfg (1B5h) Register Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	S_Hib	THCfg		THType	000			0	ParEn	I^2CSid		0001			

Table 4. I2CCmd (12Bh) Register Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0										GoToSID		0			IncSID

Since the two MAX17330 devices share the same I^2C bus, this procedure requires that the ALRT signal of one device has to be set low while the other one is set high.

Table 5. I^2C ALRT Settings

GoToSID	Alert High	Alert Low
	Primary/Secondary Address	Primary/Secondary Address
0b00	ECh/96h	6Ch/16h
0b01	64h/1Eh	ECh/96h
0b10	E4h/9Eh	64h/1Eh
0b11	6Ch/16h	E4h/9Eh

Table 4, from the MAX17330 data sheet, shows how the I2CCmd register can dynamically change the address of the device based on the ALERT GPIO pin value. In this case, the GoToSID and INcSID fields are used to change the I^2C address:

- Set ALRT_A logic low
- Set ALRT_B logic high
- Write I2CCmd = 0 × 0001 → MAX17330_A address remains at 6Ch/16h
  → MAX17330_B address set to ECh/96h

Once each device has its own unique address, the entire system can be controlled by a single microcontroller.

Here is the script for the microcontroller to complete I^2C configuration. This will be part of the system initialization.

- Load .INI file
- Assert ALRT_A and ALRT_B to keep the path between SYSP and BATTP open
- Read V_{BATT}_A and V_{BATT}_B
- V_{MAX} = max (V_{BATT}_A, V_{BATT}_B)
- Set $V_{OUT} = V_{MAX}$ + 50 mV
- Release ALRT_A and ALRT_B
- Set nProtCfg.OvrdEn = 0 to use ALRT as Output

See Table 6.

Some registers in the nonvolatile space require the firmware to be restarted for the change to take effect. Thus, the following step is required:

- Assert Config2.POR_CMD to restart firmware

See Table 7.

Next, we need to enable interrupts from the chargers:

- Set (Config.Aen and Config.Caen) = 1

See Table 8.

Now the devices are initialized.

Table 6. nProtCfg (1D7h) Register Format

D15	D14	D13	D12	D11	D10	D9	D8
ChgWDTEn	$\overline{\text{nChgAutoCtrl}}$	FullEn	SCTest		CmOvrdEn	ChgTestEn	PrequalEn
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Reserved	PFEn	DeepShpEn	OvrdEn	$\overline{\text{UVRdy}}$	FetPFEn	BlockDisCEn	DeepShp2En

Table 7. Config2 (0ABh) Register Format

D15	D14	D13	D12	D11	D10	D9	D8
POR_CMD	0	AtRtEn	0	0	0	0	0
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
dSOCen	TAlrtEn	0	1	DRCfg		CPMode	BlockDis

Table 8. Config (00Bh) Register Format

D15	D14	D13	D12	D11	D10	D9	D8
0	SS	TS	VS	0	PBen	DisBlockRead	$\overline{\text{ChgAutoCtrl}}$
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
SHIP	COMMSH	FastADCen	ETHRM	FTHRM	Aen	CAen	PAen

Logging Data and Interrupts

We need to be able to read registers to log data and check if an interrupt has been generated on the ALERT GPIO lines. We can use this script:

- Set 500 ms Timer
- V_{MIN} = min (V_{BATT}_A, V_{BATT}_B)
- Vsys_min = nVEmpty[15:7]
- CrossCharge = False
- If (V_{MIN}<Vsys_min) → CrossCharge = True
  Evaluate if the minimum battery voltage exceeds the minimum operating voltage of the system
- If FProtStat.IsDis = 0
  Charging signal is detected
- Clear Status.AllowChgB
  Indicate charger presence to all batteries
- If (V_{BATT} > V_{MIN} + 400 mV and !Cross Charge)
  Determine which battery to block to avoid cross-charging

  Config2.BlockDis = 1

  else

  Config2.BlockDis = 0
  Allow discharging if the low battery is much lower than the high battery

See tables 9, 10, and 11.

When ALRT is asserted from the MAX17330, the host will perform the following:

Read Status register data
If Status.CA is set
    Read ChgStat register
    If ChgStat.Dropout = 1 → increase V_{OUT}
    If (ChgStat.CP or ChgStat.CT) = 1 → decrease V_{OUT}
    Clear Status.CA

See tables 12 and 13.

Figure 6 shows the parallel charging plot extracted from the logged data (Excel file). Note how it follows the step charging profile.

FProtStat Register

Table 9. FProtStat (0DAh) Register Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X										IsDis	X		Hot	Cold	Warm

Table 10. Status (000h) Register Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PA	Smx	Tmx	Vmx	CA	Smn	Tmn	Vmn	dSOCi	Imx	AllowChgB	X	Bst	Imn	POR	X

Table 11. Config2 (0ABh) Register Format

D15	D14	D13	D12	D11	D10	D9	D8
POR_CMD	0	AtRtEn	0	0	0	0	0

D7	D6	D5	D4	D3	D2	D1	D0
dSOCen	TAlrtEn	0	1	DRCfg		CPMode	BlockDis

Table 12. Status Register (000h) Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
PA	Smx	Tmx	Vmx	CA	Smn	Tmn	Vmn	dSOCi	Imx	AllowChgB	X	Bst	Imn	POR	X

Table 13. ChgStat (0A3h) Register Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Dropout	X	X	X	X	X	X	X	X		X	X	CP	CT	CC	CV

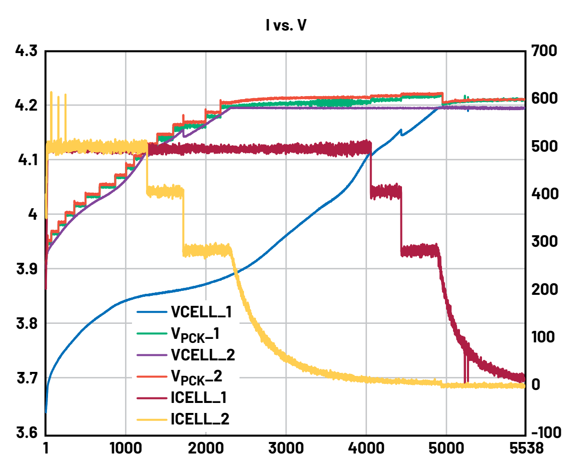

Figure 6. A parallel charging plot.

Optionally, once the device moves from the constant current (CC) phase to the constant voltage (CV) phase, the voltage generated from the step-down converter can be reduced as follows:

- If V_{BATT} = ChargingVoltage
  Read ChgStat Register
  If ChgStat.CV = 1 → decrease V_{OUT} until V_{PCK} = ChargingVoltage + 25 mV

These are all the steps needed to manage a 1S2P charging configuration. Included in MAX17330-usercode.zip is the Python code for configuring the buck converter (MAX20743) as well as the charger and fuel gauge (MAX17330). It also includes the Excel data log to capture important charging parameters and evaluate the step charging profile. By managing alert signals generated from the MAX17330, a microcontroller keeps the linear charger of the MAX17330 close to dropout, minimizing power dissipation and therefore allowing high charging current. A battery pack using the MAX17330 stores the parameters for the installed battery that the host microcontroller needs to implement efficient fast charging. This allows OEMs to replace a standard charger IC device with a simpler and less expensive buck converter without compromising performance or reliability.

Conclusion

Device charging time is one of the most important user experience considerations. Using a buck converter like the MAX17330 makes it possible to efficiently manage a very high current to decrease charging time in a small IC package. The ability to support parallel charging with a very high current, such as with two MAX17330, enables developers to charge multiple batteries in a safe, reliable manner that keeps charging time to a minimum.

About the Authors

Franco Contadini has over 35 years of experience in the electronics industry. After 10 years as a board and ASIC designer, he became a field applications engineer supporting industrial, telecom, and medical customers and focusing on power and battery management, signal chains, cryptographic systems, and microcontrollers. Franco has authored several application notes and articles on signal chains and power. He studied electronics at ITIS of Genoa, Italy.

Alessandro Leonardi is an account manager at Analog Devices, Milan. He studied electronics engineering and received a bachelor's and master's degree from Politecnico di Milano. After graduating, he became part of the field applications trainee program at ADI.

TA24441-5/23