NEC

Application Note

Battery Charging

with the KSeries Microcontroller

Document No. U17173EE2V0AN00
Date Published: October 2006

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

- The information in this document is current as of 27.05, 2004. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC sales representative for availability and additional information.

- No part of this document may be copied or reproduced in any form or by any means without prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such NEC Electronics products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.

- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of customer's equipment shall be done under the full responsibility of customer. NEC Electronics no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.

- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.

- NEC Electronics products are classified into the following three quality grades: “Standard”, “Special” and “Specific”.

The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated “quality assurance program” for a specific application. The recommended applications of NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.

"Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.

"Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).

"Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is “Standard” unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact NEC Electronics sales representative in advance to determine NEC Electronics 's willingness to support a given application.

Notes: **1.** " NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.

2. " NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02.10

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics America Inc.
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 1101
Fax: 0211-65 03 1327

Sucursal en España
Madrid, Spain
Tel: 091- 504 27 87
Fax: 091- 504 28 60

Succursale Française
Vélizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

Filiale Italiana
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

Branch The Netherlands
Eindhoven, The Netherlands
Tel: 040-244 58 45
Fax: 040-244 45 80

Branch Sweden
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

United Kingdom Branch
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
Singapore
Tel: 65-6253-8311
Fax: 65-6250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

Chapter 1 KSeries Introduction

Rechargable batteries are found in many products and systems nowadays, from small hand held devices such as camcorders and portable games, through medium size appliances like power tools, up to large systems such as emergency lighting and building fire and security systems. In effect, such batteries are found in consumer products, industrial systems, building management systems and many more either to replace disposable zinc-carbon or alkaline cells or to provide a form of backup if normal (mains) power fails.

NEC's KSeries microcontroller family are ideally suited for battery charging applications owing to a large range of devices, from low pin count parts featuring the cost-effective 78K0S core through to the 32-bit V850 series. This range of microcontrollers feature a large collection of peripherals, including the analog to digital converters and pulse width modulators (PWM) that are necessary for battery chargers.

This application note outlines several methods of using NEC microcontrollers to assist with the charging of Lithium-Ion (Li-Ion), Sealed Lead-Acid (SLA) and Nickel Cadmium (Ni-Cd) batteries. By describing several approaches, it is hoped the engineer will be able to make a choice that is optimum for his / her design in terms of cost, performance and complexity.

One approach uses the microcontroller itself as the basis for a switching step-down (Buck) converter, thus reducing external component count to a minimum. Sensing of battery voltage and current is performed by the microcontroller and the charger output is varied in order to keep the charge parameters constant. The alternative approach explains how the microcontroller can be used to control a dedicated battery charger IC. Here the closed loop control of battery current & voltage is done by the charger IC, and the microcontroller is performing more of a supervisory role, such as detection of battery insertion, state-of-charge indication and termination of charging.

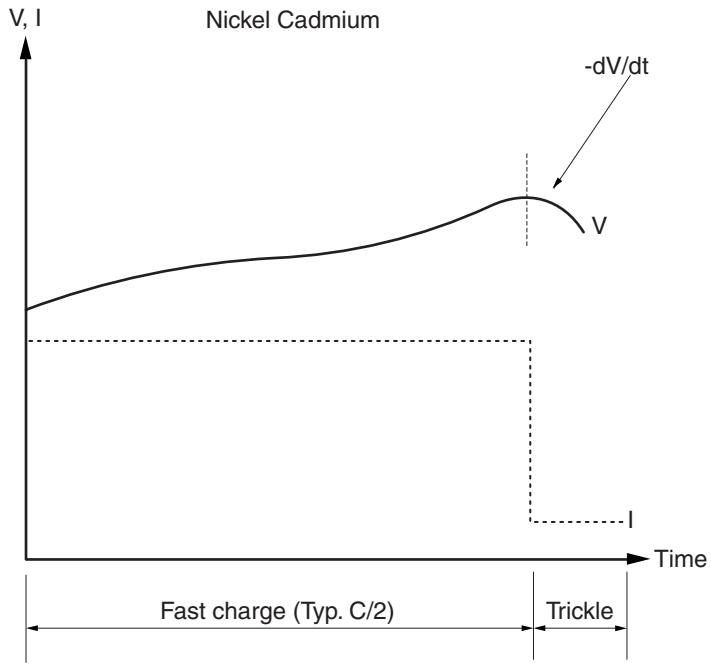
The charging requirements of SLA, Ni-Cd and Li-Ion batteries are different in several respects and will be briefly explained below.

Incidentally, battery capacity is measured in mAh (milliampere hours), and is denoted by the abbreviation 'C', so for example a battery stated as having C = 500 mAh can in theory provide 500 mA for 1 hour, 250 mA for 2 hours, etc. C is also used to express charge / discharge rate, so C/2 for a 500 mAh battery would imply a charge / discharge current of 250 mA, C/10 implies a current of 50 mA and so forth.

Chapter 2 Battery Charging Characteristics

2.1 Nickel Cadmium

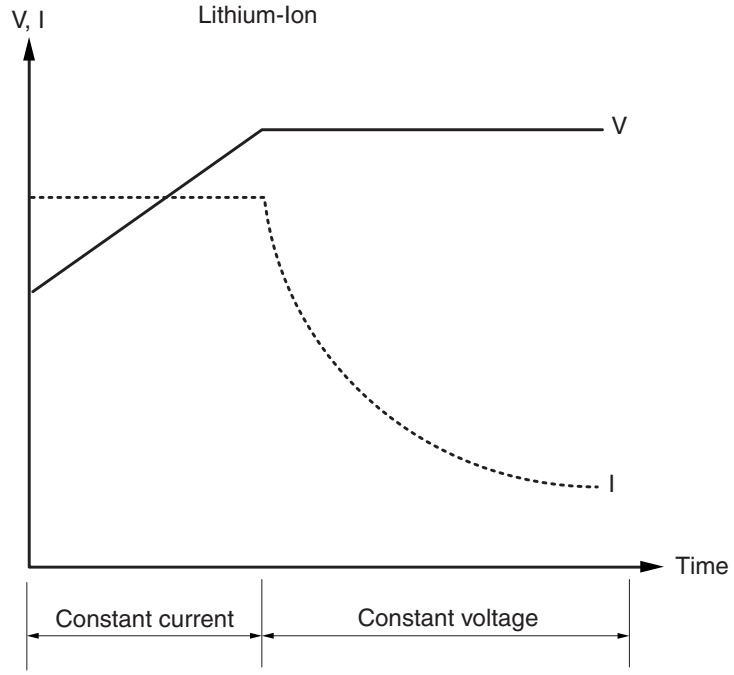
Figure 2-1: Nickel Cadmium battery charging profile



Nickel Cadmium batteries are charged at a constant current which can be as high as C, but to prolong the life of the battery and to prevent overheating C/2 is often used as the maximum charge rate. The battery voltage (typically 1.45 V/cell for “AA” size) will rise as the battery is charged; when the battery is fully charged its voltage will actually start to drop. This is known as the –dV/dt condition, and can be used by the charger to detect the full charge condition. The temperature of the battery will also rise quite abruptly as full charge is reached, this can be used as an additional mechanism for the charger to detect full charge.
When the battery is fully charged, the charger output can be switched off entirely or a low current *trickle charge* can be applied to maintain charge. A rate of C/50 to C/10 is suitable for this; alternatively the fast charge phase can be omitted entirely and the battery only trickle charged. Fast (C/2) charging is sometimes known as “1 hour charging” (although it may not necessarily take 1 hour) and trickle charging as “overnight charging”.

2.2 Lithium-Ion

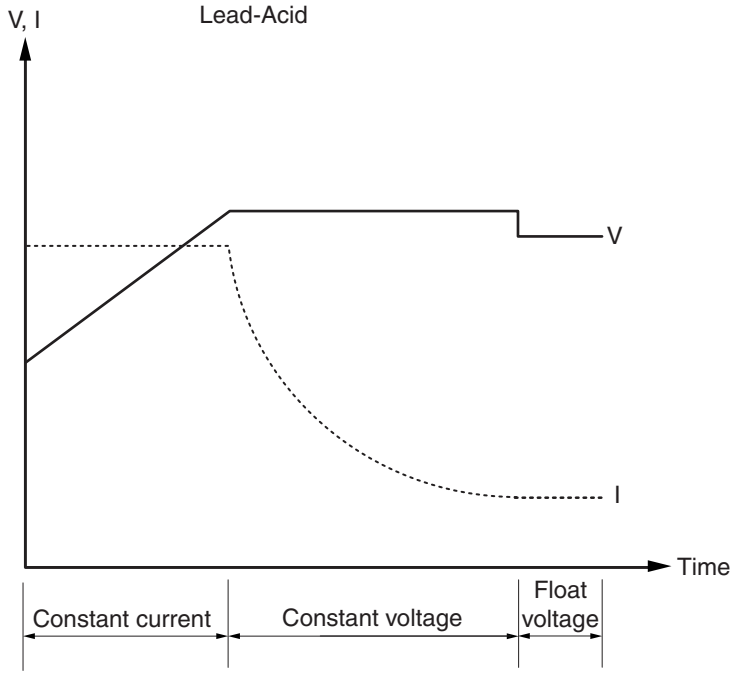
Figure 2-2: Lithium-Ion battery charging profile



Lithium-Ion batteries have different charging requirements to nickel cadmium in that the tolerance of the charge voltage is tighter, and after an initial constant current fast charge period a constant *voltage* period is used. Some chargers do not apply the constant voltage and simply terminate charge when the desired battery voltage is met, but this will only charge the battery to approximately 70% of its maximum. The tolerance of the voltage output should be less than 0.75%, though this figure depends upon the battery under charge. When the battery has reached the full charge voltage the current drawn from the charger will taper down (see Figure 2-2); when this current has fallen below a certain level (typically 100-200 mA) the charger can consider the battery fully charged and can end the charging process. Trickle charging is not recommended for Lithium Ion batteries as they do not accommodate overcharging. Li-Ion batteries are typically 3.6 V/cell.

2.3 Lead-Acid

Figure 2-3: Lead acid battery charging profile

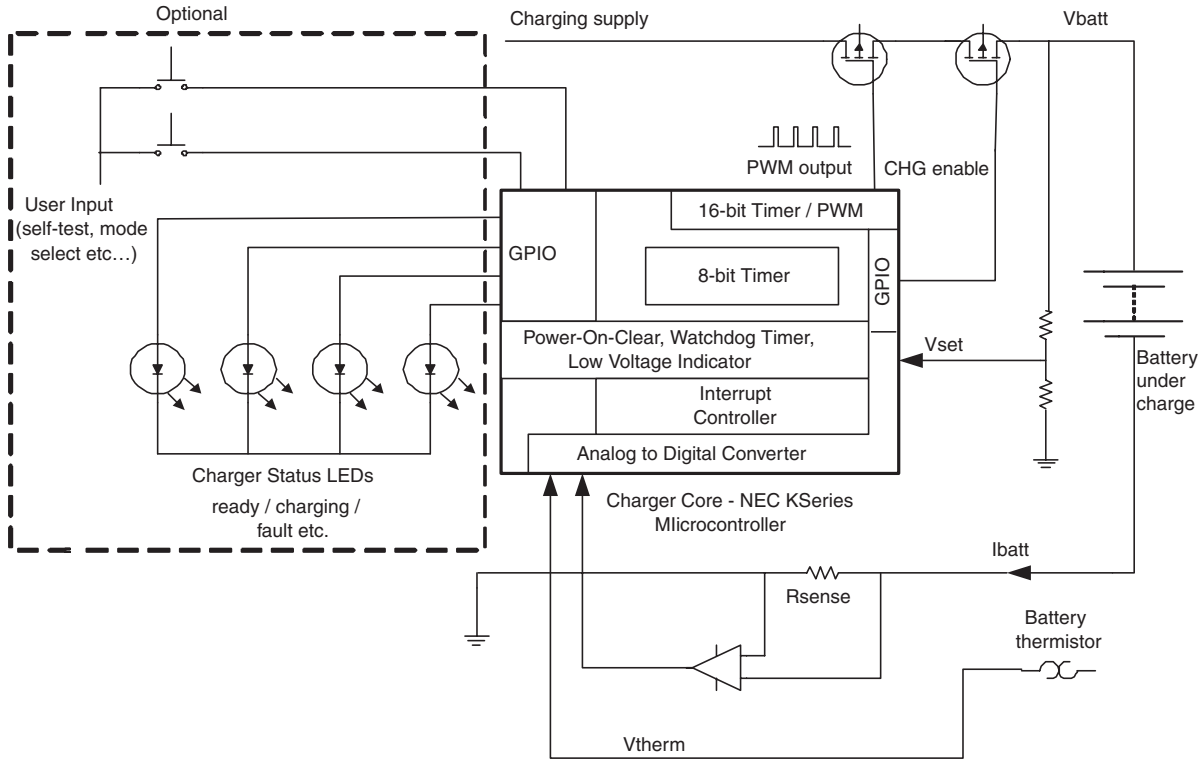


Lead acid batteries have similar charge requirements to Lithium Ion in that they both use the constant current and constant voltage charging phases, although the charge voltage applied has a less stringent tolerance than for Li-Ion. In addition, when the current has tapered down to about 3% of maximum rated current the charge voltage can be reduced from the nominal 2.4 V/cell to what is known as the *float* voltage (typically 2.25 V/cell) and this charge can be left indefinitely to avoid self discharge of the battery.

With all the above battery types, there are some common points to observe. Safety is always of prime importance, and mechanisms can be built into the charger to enhance this. Fail-safe timers can be used to end charging after a preset time if other charge termination methods fail. The temperature of the battery can be monitored (if the pack has a thermistor fitted) and drastic changes in pack temperature can trigger a “fail” mode. Incidentally, the thermistor is sometimes used as an aid to the detection of a battery inserted into the charger. If the charger will be used in an environment where ambient temperature is likely to vary considerably an additional temperature sensor can be incorporated, and its reading compared to that of the battery temperature sensor. Close monitoring of the battery (*not* charger) voltage and current is also mandatory in a safe charging system.

Chapter 3 Charger Basics

Figure 3-1: Block Diagram of a Typical Charger, Showing NEC KSeries Peripheral Usage



Remark: GPIO = General Purpose Input/Output

Figure 3-1 shows a simple block diagram of a battery charging system. The parts marked as optional may be included in a desktop / bench charger, but are unlikely to be required if the charger is embedded into a piece of equipment.
The main component of the system will be a regulator to convert the raw AC or DC power into a steady voltage within the range allowed by the battery. Current limiting is also required for the constant current charging stages mentioned before. The regulator can be a simple linear type, however for reasons of power dissipation and efficiency a switching type is usually used, hence the PWM output drive in the diagram. A current sensing amplifier is needed to convert the current drawn by the battery into a voltage which can be used by the charger core to keep the current constant. The battery voltage must also be fed back to the core to enable it to provide regulation. The battery thermistor must be monitored and appropriate action taken if it is too hot; a sensor for ambient temperature is sometimes included too. Additional functions can include switches to set mode / battery type etc. and LEDs to provide status information and / or battery state-of-charge in the form of a bargraph type display.

The block “charger core” above could be either a microcontroller operating in a stand-alone mode or controlling a battery charger IC.

NEC’s family of microcontrollers are well suited to battery charging applications as they feature the PWM channels, timers and analog to digital converters needed for this function. The charging can either be the sole function of the microcontroller or a secondary function with the microcontroller also performing other tasks. Additionally NEC also manufactures a range of low voltage MOSFETs which can be used to implement the switch S in the following section.

Chapter 4 The Step-Down (Buck) Converter

Figure 4-1: The Buck Converter

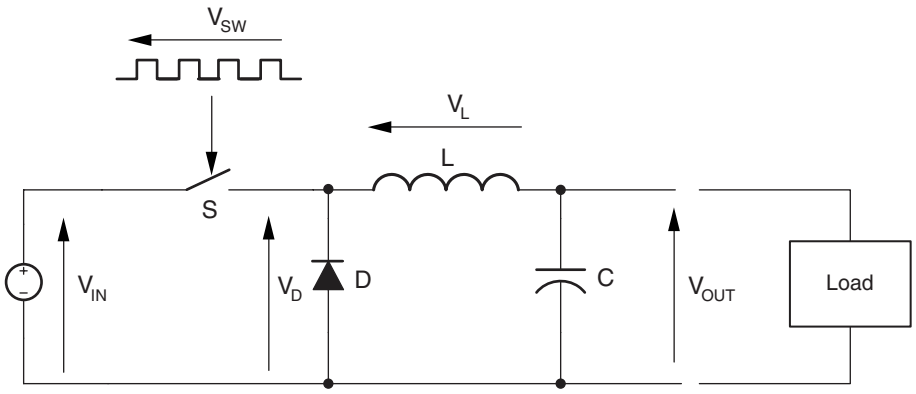
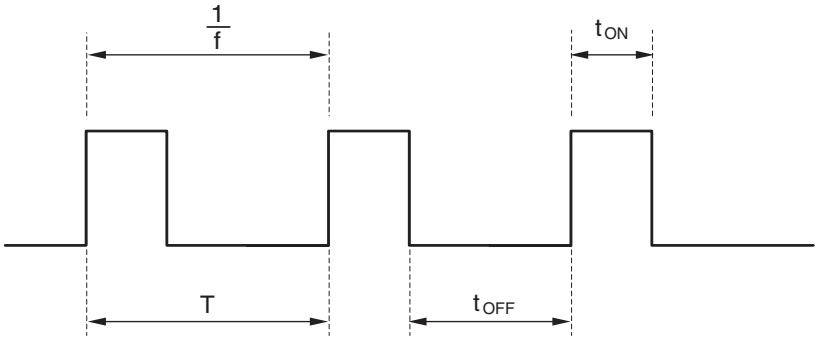


Figure 4-1 shows a Buck (step-down) converter. The switch ‘S’ is in practice a transistor (often a MOSFET) driven by the variable duty cycle PWM output from the microcontroller or battery charging IC. When switch ‘S’ is closed, current flows through it, inductor L, capacitor C and the load causing energy to be stored in the inductor. When the switch is opened, the energy stored in L is released and flows through the freewheeling diode D, capacitor C and the load. The rapid repetition of this on-off switching produces a DC level at V_{OUT} (with some ripple); this voltage is proportional to the duty cycle of the switching.

Figure 4-2: Switching Waveform at switch ‘S’



$$\text{Duty Cycle} = \frac{t_{ON}}{T + t_{ON}} \times 100\%$$

In practice, some form of regulation is required to compensate for changes in supply voltage and load. The current and voltage feedback to the charger core in Figure 3-1 are used to modify the duty cycle of the switch to keep the output constant.

For a Buck converter, the inductor value is given by:

$$L = \frac{V_{IN} - V_{OUT} - V_{SW}}{I_{PK}} \times t_{ON} \qquad \text{Equation (1)}$$

V_{IN} = Input voltage
V_{OUT} = Required output voltage
V_{SW} = Saturation voltage of switch (transistor)
I_{PK} = Peak current drawn from converter
t_{ON}= switch time ‘on’

For a step down converter,

$$I_{PK} = 2\ I_{MAX} \qquad \text{Equation (2)}$$

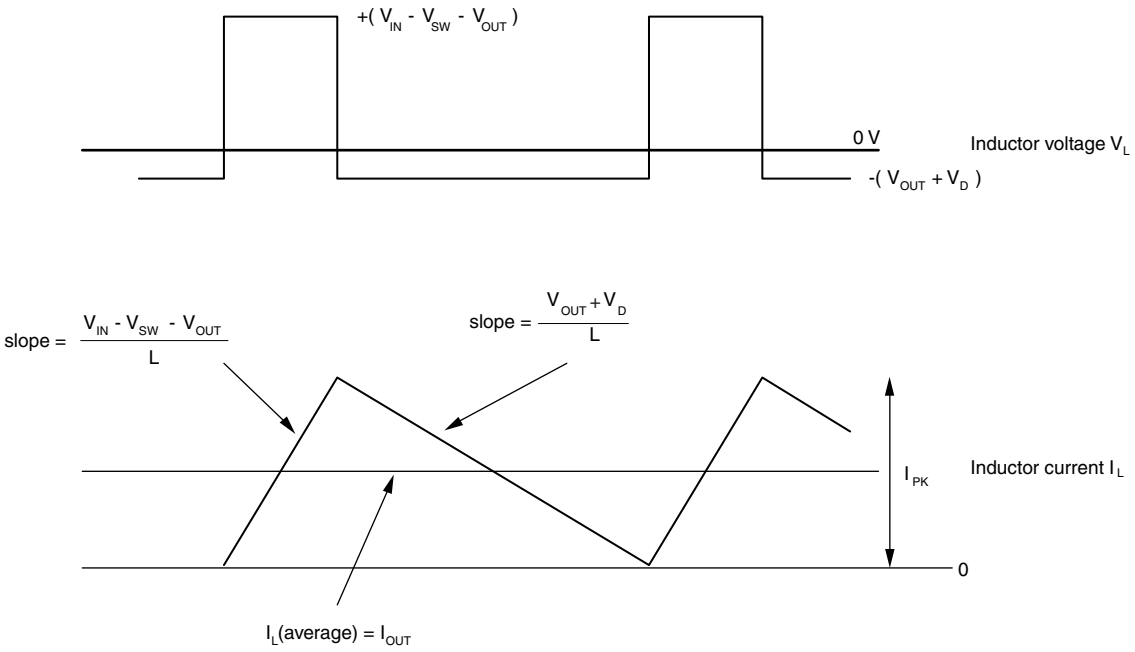
where I_{MAX} is the maximum output current.

To determine the capacitance of capacitor C use the following formula:

$$C \geq \frac{I_{PK}\ (t_{ON} + t_{OFF})}{8\ V_{RIPPLE}} \qquad \text{Equation (3)}$$

Figure 4-3 below shows the current and voltage waveforms associated with the inductor.
Since the periods t_{ON} and t_{OFF} are short compared to the time constant of L they may be approximated to a straight line.

Figure 4-3: Inductor waveforms



Example:

V_{IN} = 24 V
V_{OUT} = 12 V
V_{SW} = 0.2 V
I_{MAX} = 2 A
f = 25 KHz

Firstly,

$$T = \frac{1}{f} = 40\ \mu s$$

Assuming 50% duty cycle, t_{ON} = 20 µs

From equation (2), $I_{PK} = 2\ I_{MAX} = 2 \times 2 = 4$ A

Now, from equation (1),

$$L = \frac{24 - 12 - 0.2}{4} \times 20 \times 10^{-6} = 59\ \mu H$$

Now the value of C can be determined using equation (3):

Assuming 50 mV ripple is acceptable:

$$C = \frac{4 \times 40 \times 10^{-6}}{8 \times 50 \times 10^{-3}} = 400\ \mu F$$

so take 470 µF.

Chapter 5 Charger Circuit

Figure 5-1: Charger Schematic

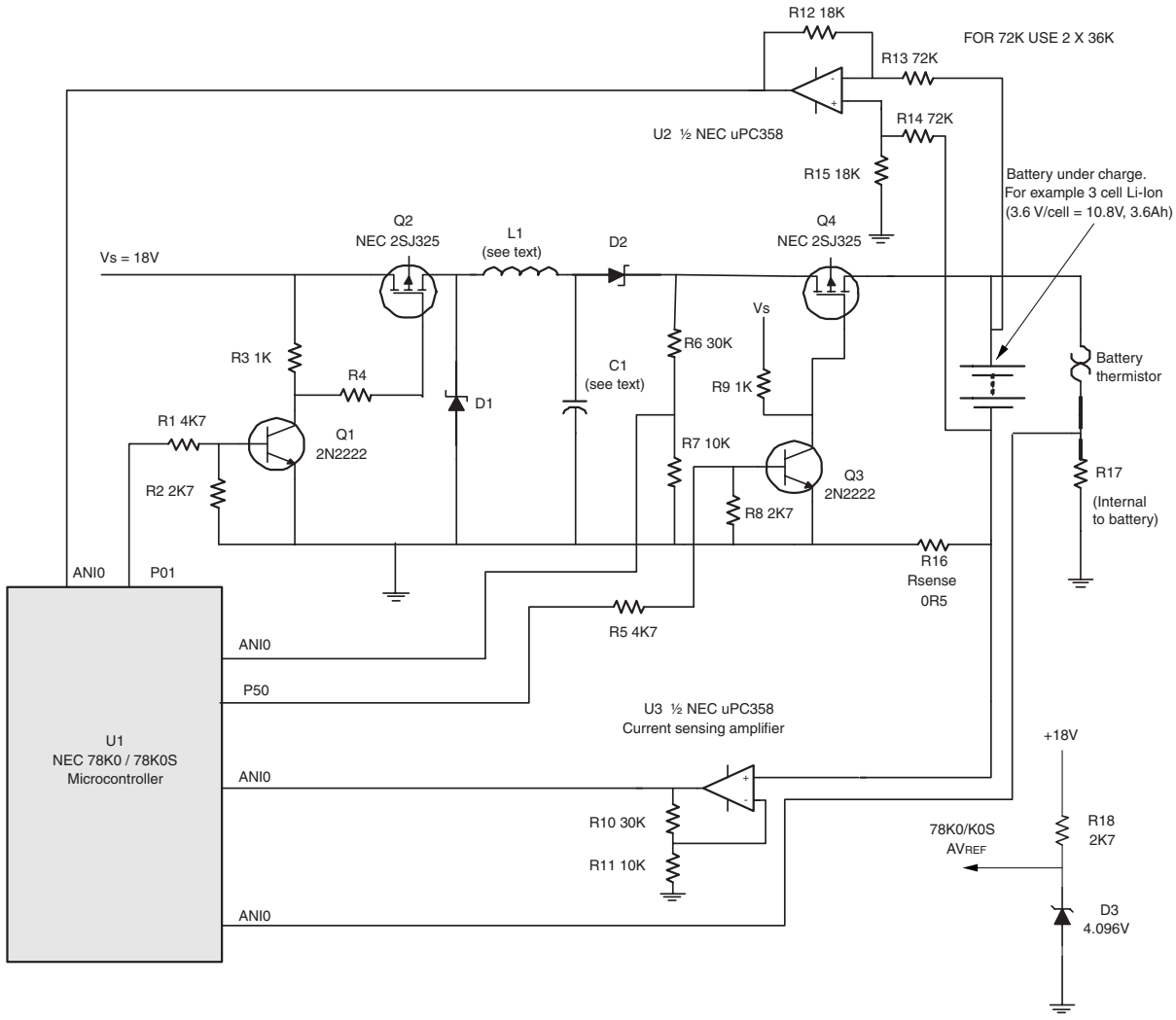


Figure 5-1 shows the full schematic of the battery charging circuit. The Buck converter is comprised of D1, L1, Q2 and C1. When laying out this circuit, it is important to keep connections between these components as short as possible. Q4 is used to switch the charger output on or off, as measurements of battery voltage will be inaccurate if the battery is still connected to the charger. NPN transistors Q1 and Q4 provide the gate drive for MOSFETs Q2 and Q4. R6 and R7 provide a potential divider to feed the ADC for charger output voltage measurement. U2 forms a differential amplifier for battery voltage measurement, and U3 forms a current sensing amplifier, measuring the voltage across sense resistor R16. D3 and R18 form a voltage reference to the microcontroller analog to digital converter. Component tolerances have not been specified here; they are largely dependant on the type of battery being charged. For example, lithium-ion batteries may require 0.1% resistors around the voltage sensing amplifier and a 0.1% tolerance voltage reference diode D3, whilst for other battery chemistries standard 1% components may be used. Similarly, exact values for L1 and C1 will be dependant upon battery charge voltage and can be determined by the formulae above. D1 and D2 should be Schottky diodes of sufficient current rating. If charge currents less than 1 A are required, NEC offer the uPA507TE P-channel MOSFET with schottky barrier diode in a space saving SC-95 package that can be used in the charger with some modification to the circuit. R4 is to prevent RF oscillations around the MOSFET, its value depends on the input capacitance of the MOSFET (C_{ISS}) and generally a value in the order of 100 R is adequate.

To take an example of charging a lithium-ion battery with a 12.3 V charge voltage at a maximum current of 2 A:

F_{CPU} = 10 MHz. V_{IN} = 18 V, V_{OUT}= 12.3 V, I_{MAX} = 2 A

For 8-bit PWM, PWM value will be

$$\frac{12.3}{18} \times 256 = 205$$

$$@\ 10\ \text{MHz},\ t_{ON} = \frac{1}{10 \times 10^{6}} \times 255 \times \frac{205}{255} = 2.05 \times 10^{-5}\ \text{s}$$

If I_{MAX} = 2 A, I_{PK} = 4 A.

So from equation (1),

$$L = \frac{18 - 12.3 - 0.2}{4} \times 2.05 \times 10^{-5} = 28\ \mu\text{H}$$

so take nearest preferred value.

It is important that the charge voltage provided to the battery is of sufficient accuracy. 8-bit PWM is used here, so the error in producing, say, 12.3 V for the above example is:

$$1\ \text{LSB} = \frac{18\ \text{V}}{256} = 0.070\ \text{V}$$

$$\frac{0.070}{12.3} \times 100\% = 0.57\%$$

Higher accuracy is possible by increasing PWM resolution (up to 10 bit) but the PWM period (and therefore t_{ON}) will increase, making a larger (and more expensive) inductor necessary.

Chapter 6 Charger Firmware

The code for the charger is implemented in C with IAR's Embedded Workbench. See the firmware listing for the versions of compiler, assembler and linker used.

This application note was developed using the µPD78F0148, from NEC's KSeriesKSeries range as used in the NEC "Play-It" development kit. Owing to the scalability of NEC's microcontroller family, the application can be easily ported down to the cost-effective 78K0S family, or up to the more powerful 32-bit V850 range. The peripherals in the KSeries range are compatible so minimal code modifications are required.

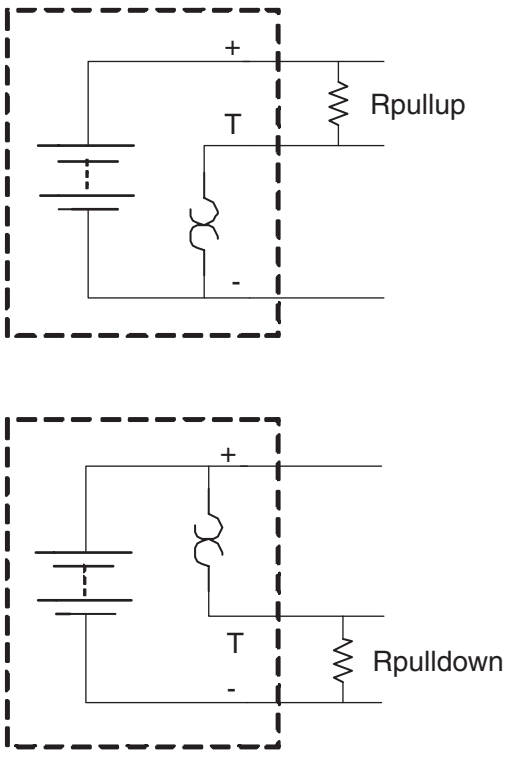
The code is in a state machine form, the intention being that the battery charging application will not block any other application the user wants to run on the microcontroller. For example a 2 second delay is used in the charger; waiting in a for-loop will stop any other functions from running which in many applications is unacceptable. Instead, a software-based timer is started which is checked every time round the main loop. Therefore other parts of the application are not blocked.

The program can accommodate one of the three battery types, set by:

```
charger_mode = CHARGER_MODE_LION;  // or CHARGER_MODE_NICD or CHARGER_MODE_SLA
```

The code could be modified to automatically detect different battery types, often the battery thermistor value / wiring arrangement are used for this. Figure 6-1 shows the thermistor arrangements possible; by switching in pull up / down resistors and making ADC measurements the program can determine the battery type. Because of the variations possible here, and the range of possible thermistor values, it is left to the reader to determine a suitable value for the pull up / down resistor and a suitable threshold for the program to compare the thermistor ADC reading to.

Figure 6-1: Possible battery thermistor configurations



6.1 Program Flow

Figure 6-2 below shows a simple diagram of program flow. After initialization the general housekeeping tasks are performed in a loop along with the detection / charging state machine. As mentioned before the charger operation is non-blocking, so other code can be added where shown, provided it does not block the charger code from running.

Figures 6-3, 6-4 and 6-5 show state diagrams for the three battery types which differ to allow the different stages of charging for each type (i.e. SLA has CC, CV and float, NICD just CC and trickle). The diagrams are self-explanatory and are very similar to each other, but a few comments are made here:

1. The system waits for detection of a battery, by measuring the voltage at the battery terminals and comparing it to limits.

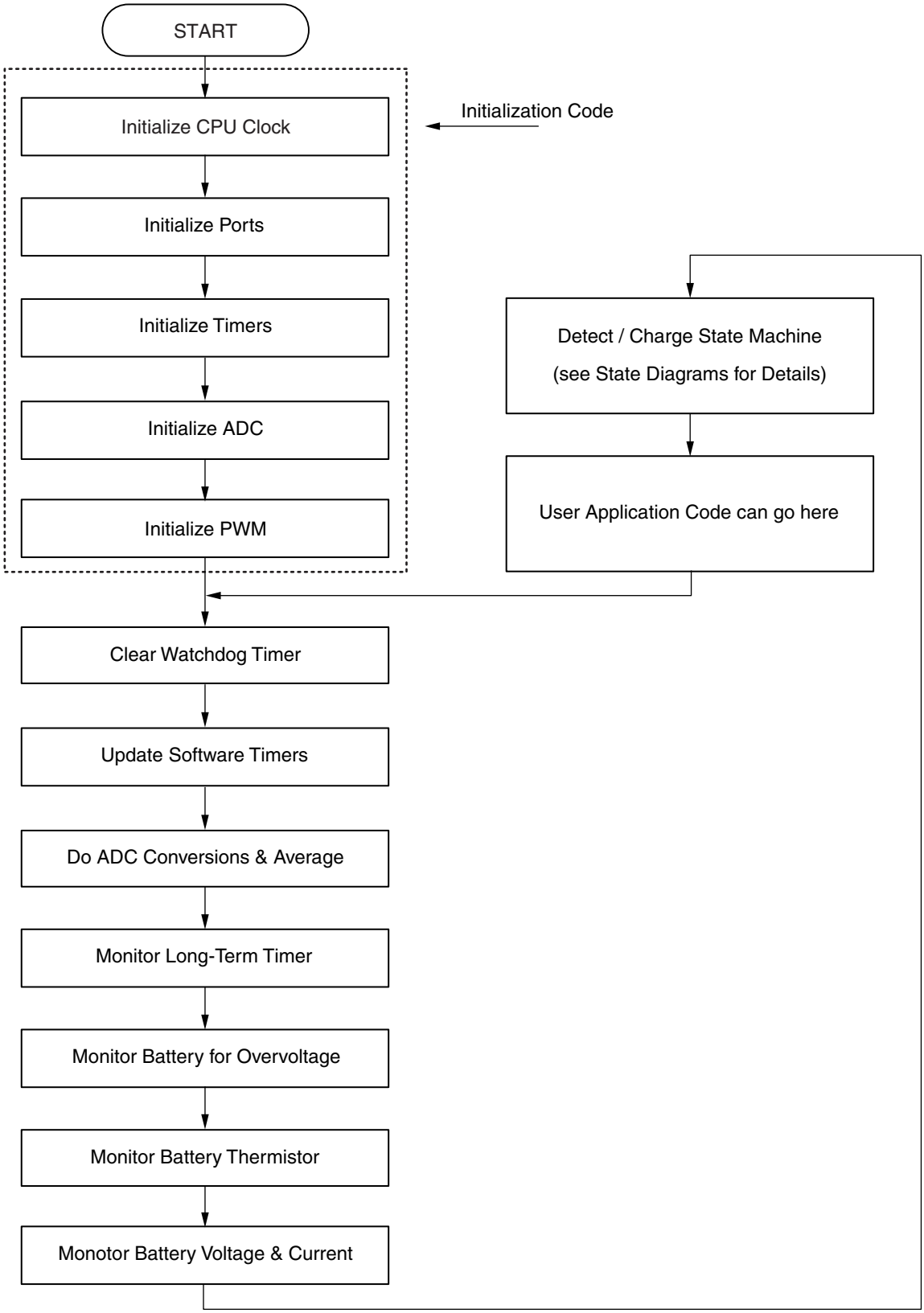
2. Upon detection, the system proceeds through the CC – CV states for Li-Ion, the CC – trickle states for Ni-Cd and the CC – CV – float states for SLA.

3. At all times during these states the common “housekeeping” block monitors for over temperature and over voltage conditions as well as looking for removal of the battery. Over temperature and over voltage conditions will put the system in a “fault” state where it will remain. The reader can implement code to act upon this condition as required.

4. A long-term timer (2 hours) is started when a battery is detected, and is used as a fail-safe if the other termination methods fail. It also terminates trickle and float charging (Ni-Cd and SLA respectively).

5. When a battery charge cycle is complete the system waits in the “terminate” state until battery removal, then it returns to the “detect” state.

Figure 6-2: Program Flow



6.2 State Diagrams

Figure 6-3: Lithium Ion charging state diagram

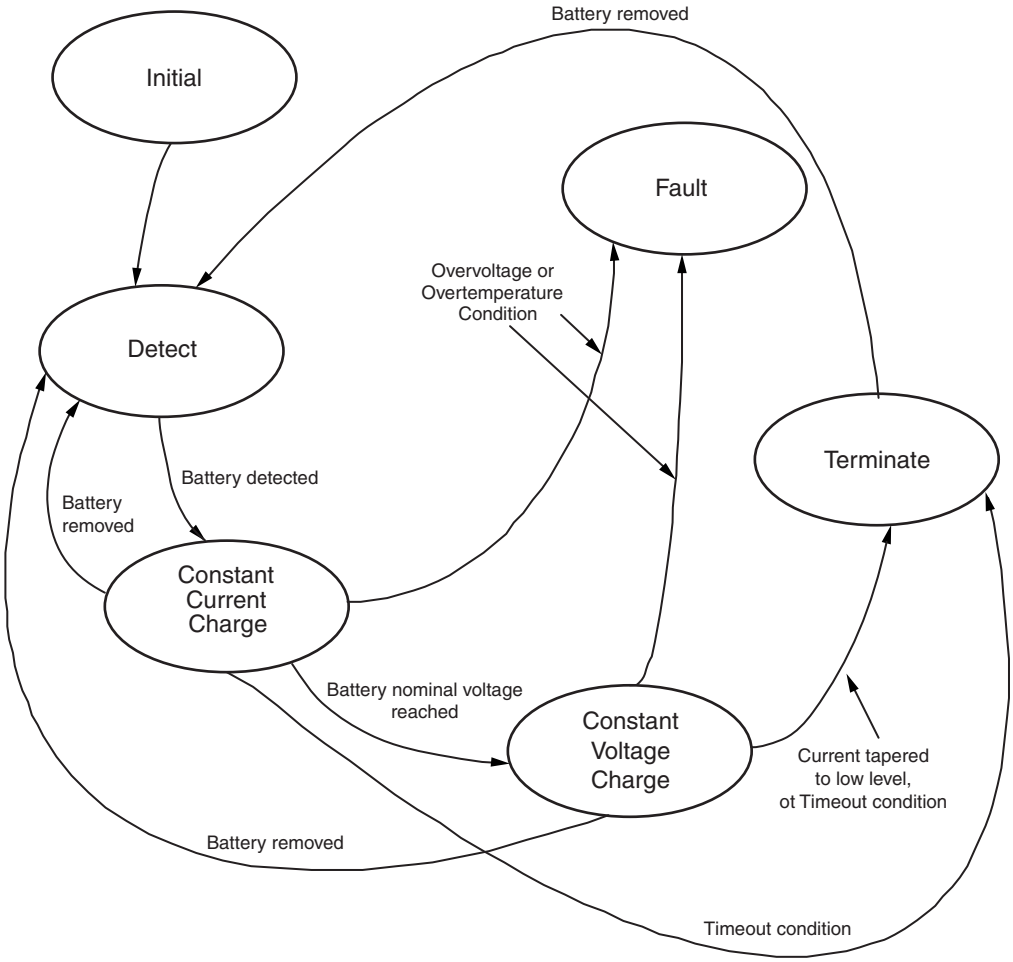


Figure 6-4: Nickel Cadmium charging state diagram

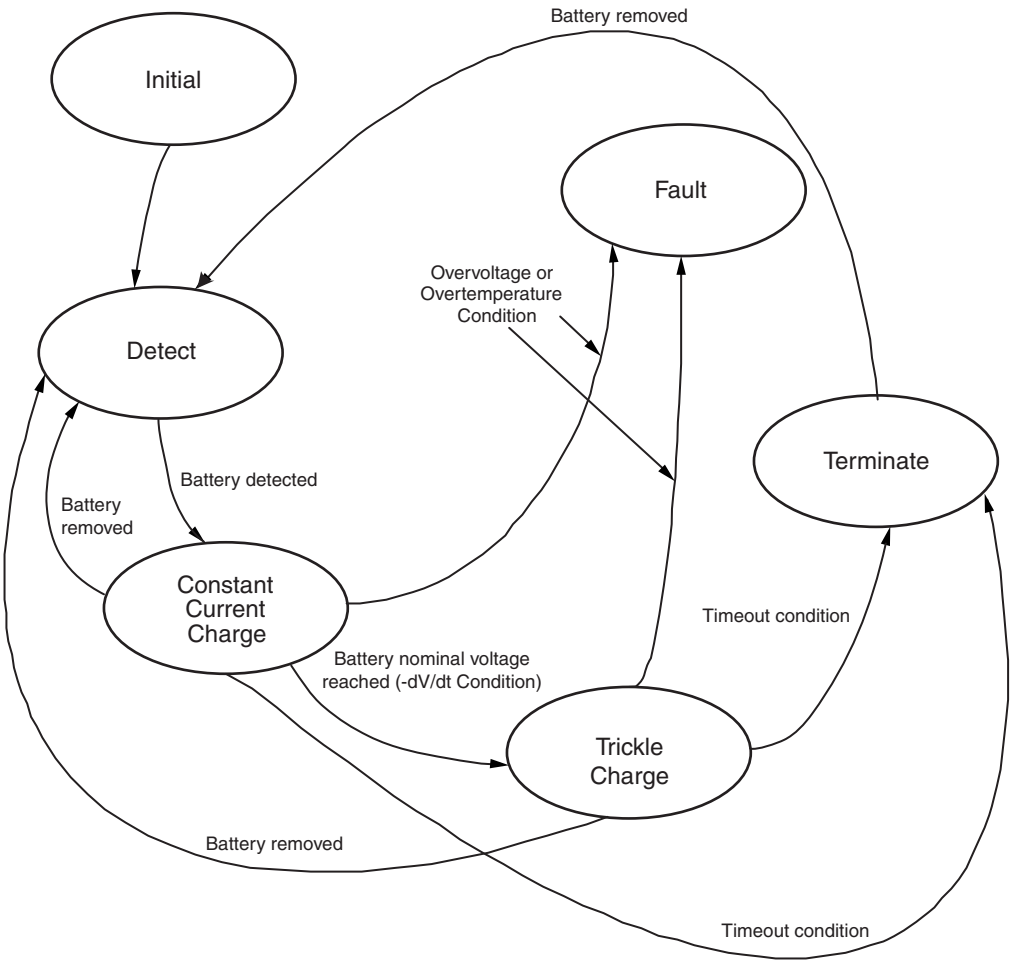
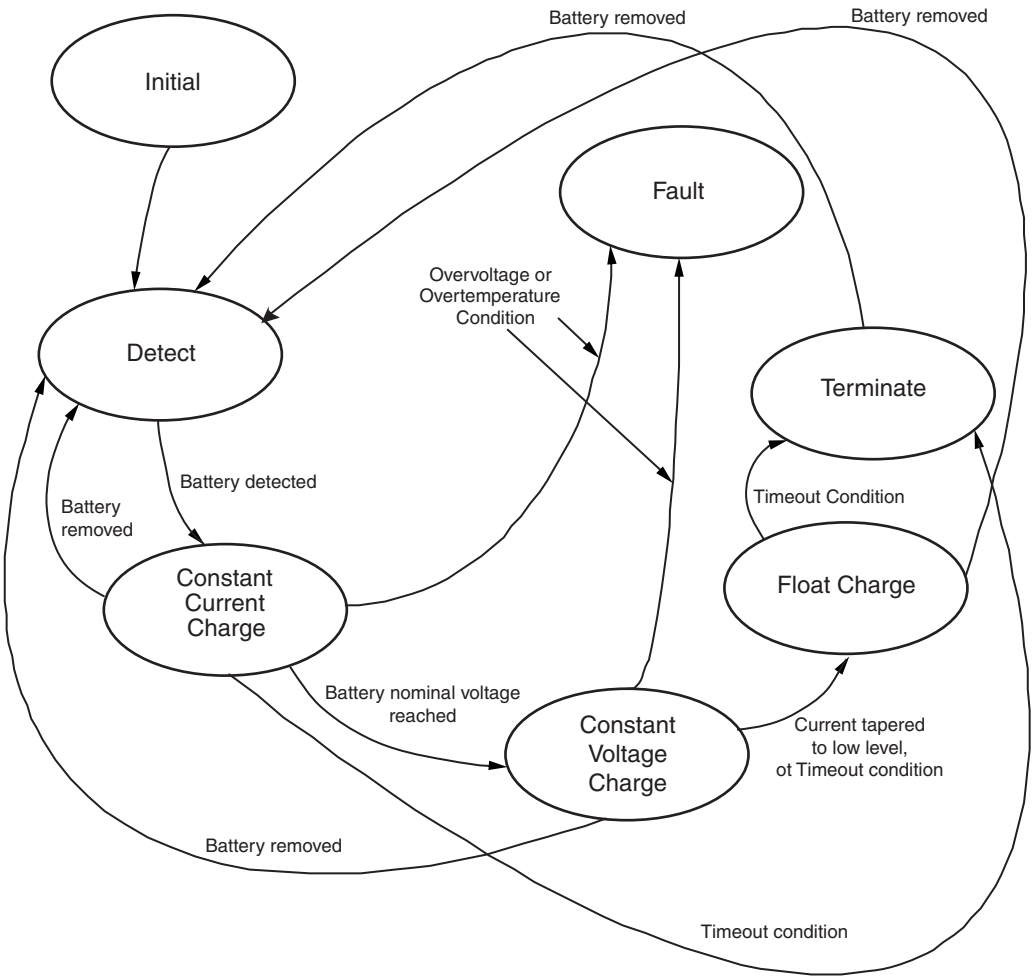


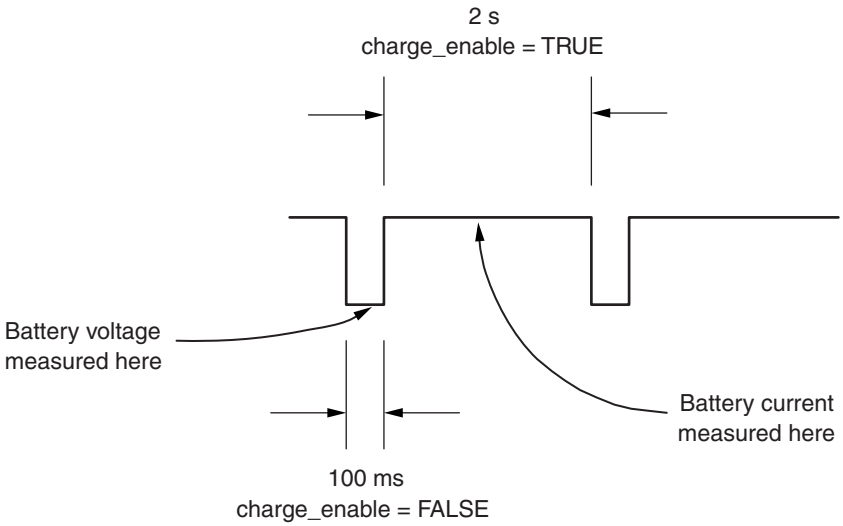
Figure 6-5: Sealed Lead Acid charging state diagram



6.3 Battery Monitor

The battery voltage and current must both be monitored whilst the battery is charged. It is necessary that the battery be disconnected from the charger to measure voltage, and connected to measure current. The charge_enable line can be used to disconnect the battery as required. It is important that a short delay is completed after charge output is switched off in order to let the terminal voltage of the battery to settle. Figure 6-6 shows the operation of the battery monitor. The action shown is in the “housekeeping” section of the program and is thus repeated continuously during charging, and the do_adc_conversions() function (also running continuously) will read battery voltage or current dependant on the state of charge_enable.

Figure 6-6: Battery Monitor



6.4 Function Descriptions

`void update_timers(void);`	Called upon timeout of 8 bit hardware timer TM50 100 times per second. Decrements the software timers, and sets them to timeout state as necessary for reading by rest of program
`void load_timer (char timer_id, int timer_val);`	Loads a software timer in centiseconds (up to 65535)
`void start_timer (char timer_id);`	Starts specified timer
`void stop_timer (char timer_id);`	Stops specified timer. (Does not reset it)
`char check_timer (char timer_id);`	Returns: TIMER_STOPPED, TIMER_RUNNING, TIMER_TIMEOUT or TIMER_IDLE
`void reset_timer (char timer_id);`	Clears specified timer and sets its state to TIMER_IDLE
`void initialize_pwm (int initial_pw);`	Initialises microcontroller PPG (PWM) module to value passed
`int get_adc_value (char channel);`	Makes a single reading of specified analog to digital converter channel (channels 0 to 3 used here)
`void write_pwm (int pwm_value);`	Writes pwm_value to PWM module. pwm_value maximum value determined by chosen PWM resolution (255 for 8-bit)
`void initialize_adc (void);`	Sets up analog to digital converter
`void cv_cc_control (signed int setpoint, signed int parameter);`	Computes PWM output. Limits result and drives PWM module with write_pwm function
`void do_adc_conversions (void);`	Called from the main “housekeeping” loop, reads ADC channels for battery voltage, battery current, thermistor value and charger voltage. Takes a moving average and stores results in variables for use by rest of program

6.5 Definitions

Individual battery parameters can be set in the #defines below:

```
#define     LION_CC_CURRENT_NOMINAL       1800    // milliamps
#define     LION_CV_VOLTAGE_NOMINAL       12300   // millivolts
#define     LION_TERMINAL_VOLTAGE_NOMINAL 10800   // millivolts

#define     NICD_CC_CURRENT_NOMINAL       1000    // milliamps
#define     NICD_TRICKLE_CURRENT_NOMINAL  200     // milliamps

#define     SLA_CC_CURRENT_NOMINAL        1500    // milliamps
#define     SLA_CV_VOLTAGE_NOMINAL        14700   // millivolts
#define     SLA_TERMINAL_VOLTAGE_NOMINAL  12000   // millivolts
#define     SLA_FLOAT_VOLTAGE_NOMINAL     13500   // millivolts
```

Parameters are either in mV or mA

Voltage thresholds for battery detection are below:

```
#define     LION_DETECT_VOLTAGE_MIN       8000    // millivolts
#define     LION_DETECT_VOLTAGE_MAX       14000   // millivolts
#define     NICD_DETECT_VOLTAGE_MIN       8000    // millivolts
#define     NICD_DETECT_VOLTAGE_MAX       14000   // millivolts

#define     SLA_DETECT_VOLTAGE_MIN        8000    // millivolts
#define     SLA_DETECT_VOLTAGE_MAX        14000   // millivolts
```

These are very broad limits, and can be “tightened” to, for example, 10000mV to 11000mV for a 10.8V nominal lithium ion battery. The charger firmware could be modified to detect batteries of different voltages by comparing the measured terminal voltage with different limits, and if this is combined with thermistor detection as mentioned before, a very flexible charger can be produced.

```
#define     AVERAGING_POINTS   4
```

Sets the number of points for the moving average of the ADC readings. In practice it is a compromise between execution time and “smoothness” of battery parameter readings.

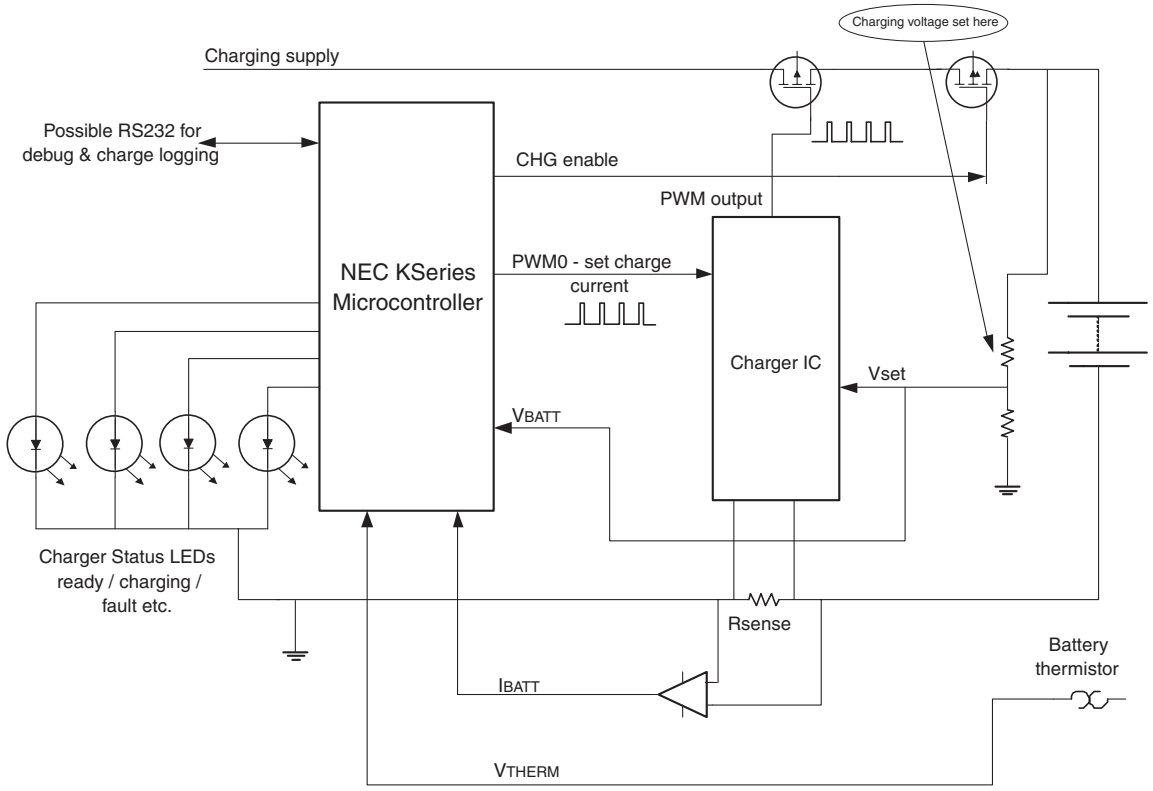
Chapter 7 Other Charging Methods

An alternative to using the microcontroller alone to charge batteries is mentioned briefly here. Battery charging ICs are available from several sources and range from relatively simple switched mode controllers / power supplies to more sophisticated devices featuring system management bus, synchronous rectifiers etc.

The advantages of using an external IC include the freeing-up of the microcontroller CPU time; a loop continuously measuring and correcting battery voltage and current may leave little time for other CPU tasks. A charger IC will require the voltage and current be initially set by the microcontroller (usually through PWM) but will then use its own internal analog circuitry to maintain the closed loop control. Additionally the PWM module of a charger IC will typically run up to ten times the frequency than that of a microcontroller, allowing a smaller inductor and higher efficiency. The main disadvantage is cost; the charger IC may cost significantly more than the microcontroller.

The microcontroller is used here more in a supervisory role since as well as setting charge voltage and current it can monitor battery voltage independently of the charger IC for safety reasons, or to provide state-of-charge information to the user through a bargraph LED display or some form of LCD. Charger ICs do not generally feature a means of reading temperature via a thermistor, so this and any appropriate over-temperature shutdown mechanism must be undertaken by the microcontroller. See Figure 7-1 below.

Figure 7-1: Use of external charger IC



Notes: 1. 78K microcontroller functions:
Fixed o/p voltage assumed (for a charger built into equipment)
o/p current PWM variable for fast / slow / trickle charge.

2. ADC measurements:
V_{BATT} - for charge termination (Ni-Cd -dV/dt), & safety monitoring
I_{BATT} - for charge termination (Li-Ion - current taper) & safety monitoring
V_{THERM} - for battery detection & safety monitoring

Chapter 8 Conclusion

For this application the resource usage was as follows:

```
2227 bytes of CODE memory
209 bytes of DATA memory
6 bits of BIT memory
```

so it therefore leaves plenty of program memory space for other application code.

This application note has outlined some of the typical types of rechargable battery commonly in use and their requirements for charging, and has shown several methods of how the NEC KSeries micro-controller family can suit this application. The ideas presented here can be used how they are, or adapted to fit in with the users system.

Chapter 9 Firmware Listing

```
/*============================================================================
** PROJECT = BATTCHG1
** MODULE = battchg1.c
** VERSION = V1.0
** DATE = 11.03.2004
** LAST CHANGE = 12.05.2004
**
** ============================================================================
** Description: Multi-chemistry battery charger firmware
**
** ============================================================================
** Environment:     Device: uPD78F0148
** Assembler:       A78000 Version 3.34.2.4
** C-Compiler:      ICC78000 Version 3.34.2.4
** Linker:          XLINK Version 4.55.9.0
** ============================================================================
** By: NEC Electronics (Europe) GmbH
** Cygnus House
** Sunrise Parkway
** Milton Keynes
**
** ============================================================================
Changes:
** ============================================================================
*/
/* ============================================================================
** pragma
** ============================================================================
*/
#pragma language = extended
/* ============================================================================
** include
** ============================================================================
*/


#include <in78000.h>
#include <Df0148.h>

#define     TRUE                        1
#define     FALSE                       0
#define     INPUT                       1
#define     OUTPUT                      0
#define     DISABLED                    1
#define     ENABLED                     0


#define     MAX_TIMERS                  2

#define     TIMER_STOPPED               0
#define     TIMER_RUNNING               1
#define     TIMER_TIMEOUT               2
#define     TIMER_IDLE                  3

#define     AVERAGING_POINTS            4

#define     BATTERY_VOLTAGE             0   // ADS values to select ADC channel
#define     CHARGER_VOLTAGE             1   //      "
#define     BATTERY_THERMISTOR          2   //      "
#define     BATTERY_CURRENT             3   //      "

#define     BATT_V_SCL_FACTOR           16
#define     CHG_V_SCL_FACTOR            16
#define     BATT_I_SCL_FACTOR           2

#define     PWM_RESOLUTION              8
#define     XTAL_FREQ                   10000000
#define     PWM_PERIOD                  (1 << PWM_RESOLUTION)-1
#define     PWM_OUT_PIN                 P0.1
#define     PWM_OUT_DIR                 PM0.1
#define     TOC004                      TOC00.4         // not in header file
#define     PWM_VAL_MIN                 50              // 50 OK with emulator @ 10MHz 8 bit PWM

#define     CHARGE_ENABLE               P5.0

#define     CHARGER_MODE_LION           0
#define     CHARGER_MODE_NICD           1
#define     CHARGER_MODE_SLA            2

#define     LION_DET_V_MIN              8000        // millivolts
#define     LION_DET_V_MAX              14000       // millivolts
#define     NICD_DET_V_MIN              8000        // millivolts
#define     NICD_DET_V_MAX              14000       // millivolts
```

```
#define     SLA_DET_V_MIN               8000        // millivolts
#define     SLA_DET_V_MAX               14000       // millivolts

#define     LION_MAX_V                  16000       // millivolts
#define     NICD_MAX_V                  16000       // millivolts
#define     SLA_MAX_V                   16000       // millivolts

#define     BATT_REMOVED_LIMIT          6000        // millivolts

#define     LION_CC_CURRENT_NOM         1800        // milliamps
#define     LION_CV_VOLTAGE_NOM         12300       // millivolts
#define     LION_TERM_VOLTAGE_NOM       10800       // millivolts

#define     NICD_CC_CURRENT_NOM         1000        // milliamps
#define     NICD_TRICKLE_CURRENT_NOM    200         // milliamps

#define     SLA_CC_CURRENT_NOM          1500        // milliamps
#define     SLA_CV_VOLTAGE_NOM          12000       // millivolts
#define     SLA_TERMINAL_VOLTAGE_NOM    12000       // millivolts
#define     SLA_FLOAT_VOLTAGE_NOM       11000       // millivolts

#define     NICD_CHG_TERM_V             250         // millivolts

// thermistor
#define     BATTERY_THERMISTOR_LIMIT    500

// SYSTEM STATES
#define     STARTUP_STATE           0
#define     DETECT_STATE            STARTUP_STATE + 1
#define     CC_LION_STATE           DETECT_STATE + 1
#define     CV_LION_STATE           CC_LION_STATE + 1
#define     CC_NICD_STATE           CV_LION_STATE + 1
#define     TRICKLE_NICD_STATE      CC_NICD_STATE + 1
#define     CC_SLA_STATE            TRICKLE_NICD_STATE + 1
#define     CV_SLA_STATE            CC_SLA_STATE + 1
#define     FLOAT_SLA_STATE         CV_SLA_STATE + 1
#define     FAILURE_STATE           FLOAT_SLA_STATE + 1
#define     TERMINATE_STATE         FAILURE_STATE + 1

#define     ADC_CHANNELS_USED       4

#define     BATT_MON_STATE_0        0
#define     BATT_MON_STATE_1        BATT_MON_STATE_0 + 1
#define     BATT_MON_STATE_2        BATT_MON_STATE_1 + 1


int adc_result = 0;
int data_samples[AVERAGING_POINTS + 1][ADC_CHANNELS_USED];
char sample_pointers[ADC_CHANNELS_USED];

char system_state;

char batt_mon_state;

char sub_state_1;
bit wait_for_interrupt;
int global_pwm_value = PWM_VAL_MIN;
char charger_mode;
char minute_counter;
bit charge_in_progress;
bit battery_removed;
bit safety_timer_expired;
bit charger_error;
bit over_temperature;


// battery & charger parameters
int battery_voltage;
int charger_voltage;
int battery_thermistor_value;
int battery_current;
int highest_battery_voltage;


// prototypes
void update_timers(void);
void load_timer (char timer_id, int timer_val);
void start_timer (char timer_id);
void stop_timer (char timer_id);
char check_timer (char timer_id);
void reset_timer (char timer_id);
void initialize_pwm (int initial_pw);
int get_adc_value (char channel);
void write_pwm (int pwm_value);
void initialize_adc (void);
void cv_cc_control (signed int setpoint, signed int parameter);
void do_adc_conversions (void);
```

```
typedef struct timer_struct{
    int timer_value;                // counts centiseconds
    char timer_state;
}timers;

timers timer_block[MAX_TIMERS];


void main (void){

    _DI();                          // disable interrupts

    PCC = 0x00;                     // processor clock control register
                                    // oscillation possible
                                    // on-chip feedback resistor used
                                    // X1 input clock or ring-osc clock
                                    // CPU clock (fCPU) selection = 1:1

    OSTS = 0x05;                    // Osc stabilization time = 2^16/Fxp = 6.55 ms @10MHz
    MOC = 0x00;                     // start main osc.
    while (OSTC <= 0x18){           // wait for main oscillator to stabilize
        _NOP();
    }
    MCM0 = 1;                       // set cpu clock = main osc.

    PM5.0 = 0;                      // set data direction for CHARGE_ENABLE


                                    // initialize 8 bit timer TM50
    TCL50 = 0x07;                   // clock is fx/(2^13)
    CR50 = 12;
    TMC50 = 0x80;                   // enable timer 50
    TMIF50 = FALSE;

    load_timer (0, 25);
    start_timer (0);

    // ensure bit variables are cleared
    charge_in_progress = FALSE;
    battery_removed = FALSE;
    safety_timer_expired = FALSE;
    charger_error = FALSE;
    over_temperature = FALSE;

    CHARGE_ENABLE = FALSE;

    initialize_adc();

    wait_for_interrupt = FALSE;
    initialize_pwm (50);            // initialising with 0 gives high DC (> 99%)
    TMIF000 = FALSE;
    _EI();                          // global interrupt enable

    charger_mode = CHARGER_MODE_LION;       // set battery type here
//      charger_mode = CHARGER_MODE_NICD;
//      charger_mode = CHARGER_MODE_SLA;

    for(;;){

        WDTE = 0xAC;

        if (TMIF50){
            TMIF50 = FALSE;
            update_timers();
        }

        do_adc_conversions();

        if (charge_in_progress == TRUE){
            if (check_timer(1) == TIMER_TIMEOUT){
                load_timer (1, 6000);
                minute_counter--;
                if (minute_counter == 0)
                    safety_timer_expired = TRUE;
            }

        }

        switch (charger_mode){

            case (CHARGER_MODE_LION):{

                if (battery_voltage >= LION_MAX_V){
                    system_state = FAILURE_STATE;
```

```
                    charger_error = TRUE;
                }

                break;

            }

            case (CHARGER_MODE_NICD):{

                if (battery_voltage >= NICD_MAX_V){
                    system_state = FAILURE_STATE;
                    charger_error = TRUE;
                }

                break;

            }

            case (CHARGER_MODE_SLA):{

                if (battery_voltage >= SLA_MAX_V){
                    system_state = FAILURE_STATE;
                    charger_error = TRUE;
                }

                break;

            }

        }

        if (battery_thermistor_value >= BATTERY_THERMISTOR_LIMIT){
            over_temperature = TRUE;
            system_state = FAILURE_STATE;

        }

        switch (batt_mon_state){
            case (BATT_MON_STATE_0):{
                CHARGE_ENABLE = FALSE;
                load_timer (0, 10);
                start_timer (0);
                batt_mon_state = BATT_MON_STATE_1;
                break;
            }

            case (BATT_MON_STATE_1):{
                if (check_timer (0) == TIMER_TIMEOUT){
                     if ((charge_in_progress == TRUE) && (system_state != TERMINATE_STATE) && (charger_error
== FALSE))
                        CHARGE_ENABLE = TRUE;       // don't enable charge output
                                                    // when detecting battery,
                                                    // in terminate state or error state

                    load_timer (0, 200);
                    start_timer (0);
                    batt_mon_state = BATT_MON_STATE_2;
                }
                break;
            }

            case (BATT_MON_STATE_2):{
                if (check_timer (0) == TIMER_TIMEOUT){
                    batt_mon_state = BATT_MON_STATE_0;
                }
                break;
            }

        }

        switch (system_state){

            case (STARTUP_STATE):{

                    system_state = DETECT_STATE;

                break;

            }

            case (DETECT_STATE):{

                switch (charger_mode){

                    case (CHARGER_MODE_LION):{
                        if ((battery_voltage >= LION_DET_V_MIN) && (battery_voltage <= LION_DET_V_MAX)){
                            CHARGE_ENABLE = TRUE;
```

```
                charge_in_progress = TRUE;
                minute_counter = 120;           // 2 hours
                load_timer (1, 6000);           // 1 minute period
                start_timer (1);
                system_state = CC_LION_STATE;

            }

            break;

        }

        case (CHARGER_MODE_NICD):{
            if ((battery_voltage >= NICD_DET_V_MIN) && (battery_voltage <= NICD_DET_V_MAX)){
                highest_battery_voltage = 2000; // initial (low) value
                CHARGE_ENABLE = TRUE;
                charge_in_progress = TRUE;
                minute_counter = 120;           // 2 hours
                load_timer (1, 6000);           // 1 minute period
                start_timer (1);
                system_state = CC_NICD_STATE;

            }

            break;

        }

        case (CHARGER_MODE_SLA):{
            if ((battery_voltage >= SLA_DET_V_MIN) && (battery_voltage <= SLA_DET_V_MAX)){
                CHARGE_ENABLE = TRUE;
                charge_in_progress = TRUE;
                minute_counter = 120;           // 2 hours
                load_timer (1, 6000);           // 1 minute period
                start_timer (1);
                system_state = CC_SLA_STATE;

            }

            break;

        }

    }

    break;

}

case (CC_LION_STATE):{

    if (charge_in_progress == FALSE)            // battery removed
        system_state = DETECT_STATE;
    else if (safety_timer_expired == TRUE){
        safety_timer_expired = FALSE;
        system_state = TERMINATE_STATE;
        reset_timer (1);
    }
    else if (battery_voltage >= LION_TERM_VOLTAGE_NOM)
            system_state = CV_LION_STATE;       // ready for constant voltage charge
    else
        cv_cc_control (LION_CC_CURRENT_NOM, battery_current);


    break;

}

case (CV_LION_STATE):{

    if (charge_in_progress == FALSE)            // battery removed
        system_state = DETECT_STATE;
    else if (safety_timer_expired == TRUE){
        safety_timer_expired = FALSE;
        system_state = TERMINATE_STATE;
        reset_timer (1);
    }
    else if (battery_current <= (LION_CC_CURRENT_NOM / 10)){
            system_state = TERMINATE_STATE;     // current has tapered down, battery charged

    }
    else
        cv_cc_control (LION_CV_VOLTAGE_NOM, battery_voltage);   // apply constant voltage


    break;
```

```
            }

            case (CC_NICD_STATE):{

                if (charge_in_progress == FALSE)               // battery removed
                    system_state = DETECT_STATE;
                else if (safety_timer_expired == TRUE){
                    safety_timer_expired = FALSE;
                    system_state = TERMINATE_STATE;
                    reset_timer (1);
                }
                else if ((battery_voltage < (highest_battery_voltage - NICD_CHG_TERM_V)) && (battery_voltage
> (highest_battery_voltage - (2 * NICD_CHG_TERM_V))))
                        system_state = TRICKLE_NICD_STATE;      // ready for trickle charge
                else{
                    cv_cc_control (NICD_CC_CURRENT_NOM, battery_current);
                    if (battery_voltage > highest_battery_voltage)
                        highest_battery_voltage = battery_voltage;
                }

                break;

            }

            case (TRICKLE_NICD_STATE):{

                if (charge_in_progress == FALSE)               // battery removed
                    system_state = DETECT_STATE;
                else if (safety_timer_expired == TRUE){
                    safety_timer_expired = FALSE;
                    system_state = TERMINATE_STATE;
                    reset_timer (1);
                }
                else
                    cv_cc_control (NICD_TRICKLE_CURRENT_NOM, battery_current);

                break;

            }

            case (CC_SLA_STATE):{

                if (charge_in_progress == FALSE)               // battery removed
                    system_state = DETECT_STATE;
                else if (safety_timer_expired == TRUE){
                    safety_timer_expired = FALSE;
                    system_state = TERMINATE_STATE;
                    reset_timer (1);
                }
                else if (battery_voltage >= SLA_TERMINAL_VOLTAGE_NOM)
                        system_state = CV_SLA_STATE;           // ready for constant voltage charge
                else
                    cv_cc_control (SLA_CC_CURRENT_NOM, battery_current);

                break;

            }

            case (CV_SLA_STATE):{

                if (charge_in_progress == FALSE)               // battery removed
                    system_state = DETECT_STATE;
                else if (safety_timer_expired == TRUE){
                    safety_timer_expired = FALSE;
                    system_state = TERMINATE_STATE;
                    reset_timer (1);
                }
                else if (battery_current <= (SLA_CC_CURRENT_NOM / 50)){
                       system_state = FLOAT_SLA_STATE;         // current has tapered down, battery charged,
ready for float voltage

                }
                else
                    cv_cc_control (SLA_CV_VOLTAGE_NOM, battery_voltage);    // apply constant voltage

                break;

            }

            case (FLOAT_SLA_STATE):{

                if (charge_in_progress == FALSE)               // battery removed
                    system_state = DETECT_STATE;
                else if (safety_timer_expired == TRUE){
                    safety_timer_expired = FALSE;
                    system_state = TERMINATE_STATE;
                    reset_timer (1);
```

```
                }
                else
                    cv_cc_control (SLA_FLOAT_VOLTAGE_NOM, battery_voltage); // apply constant voltage

                break;

            }

            case (TERMINATE_STATE):{

                CHARGE_ENABLE = FALSE;                          // disable charger output

                if (charge_in_progress == FALSE){               // wait for battery removal
                    system_state = DETECT_STATE;

                }
                break;

            }

            case (FAILURE_STATE):{

                CHARGE_ENABLE = FALSE;

                break;

            }

        }

    }

}

// ===========================================================================

void update_timers (void){

    char loop;

    for (loop = 0; loop < MAX_TIMERS; loop++){
        if (timer_block[loop].timer_value){     // if timer value is non-zero
            if (!--timer_block[loop].timer_value)
                timer_block[loop].timer_state = TIMER_TIMEOUT;
        }
    }
}

// ===========================================================================

void load_timer (char timer_id, int timer_val){

    timer_block[timer_id].timer_value = timer_val;
    timer_block[timer_id].timer_state = TIMER_RUNNING;

}

// ===========================================================================

char check_timer (char timer_id){

        return (timer_block[timer_id].timer_state);

}

// ===========================================================================

void start_timer (char timer_id){

    timer_block[timer_id].timer_state = TIMER_RUNNING;

}

// ===========================================================================

void reset_timer (char timer_id){

    timer_block[timer_id].timer_state = TIMER_IDLE;
    timer_block[timer_id].timer_value = 0;

}

// ===========================================================================

void initialize_pwm (int initial_pw){

    PWM_OUT_PIN = FALSE;        // set up PWM output
```

```
    PWM_OUT_DIR = OUTPUT;

    TMC00 = 0x00;               // stop timer TM00
    PRM00 = 0x00;               // 1:1 prescale to TM00
                                // *** CANNOT LOAD DCs < 50 as PWM running too fast ***
    CRC00 = 0x00;               // CR000 operates as compare register
    CR000 = PWM_PERIOD;
    CR010 = initial_pw;         // set PWM duty cycle
    TOC00.0 = 1;
    TOC00 |= 0x1B;
    TOC00 &= 0x13;              // inversion enabled on match of CR000 and TM00
                                // timer output F/F set (1)
                                // inversion enabled on match of CR010 and TM00

    TMC00 = 0x0c;               // 16-bit timer mode control register
                                // 16-bit timer counter TM0n starts operation when TMC0n2 and TMC0n3 are set
                                // Clear & start occurs on match between TM00 and CR000
                               // inversion occurs on match between TM00 and CR000 or match between TM00 and
CR010


}

// ============================================================================

interrupt [INTTM000_vect] void timer00 (void){

    TMC00 = 0x00;
    wait_for_interrupt = FALSE;
    CR010 = global_pwm_value;
    TMC00 = 0x0C;

}

// ============================================================================

int get_adc_value (char channel){

    int adc_result;

    ADS = channel;                      // select ADC channel

    ADCS = TRUE;                        // start conversion
    while (!ADIF)                       // wait for EOC
        ;
    adc_result = (ADCR >> 6);
    ADCS = FALSE;                       // to stop continuous conversions
    ADIF = FALSE;                       // clear interrupt flag

    return (adc_result);

}

// ============================================================================

void write_pwm (int pwm_value){

    global_pwm_value = pwm_value;
    TMIF000 = FALSE;
    TMMK000 = ENABLED;                  // load PWM at end of period
    wait_for_interrupt = TRUE;
    while (wait_for_interrupt == TRUE)
        WDTE = 0xAC;
    TMMK000 = DISABLED;

}

// ============================================================================

void initialize_adc (void){

    ADM = 0x38;             // ADC mode register
    ADS = 0x02;             // analog input channel specification register
    PFM = 0x00;             // power-fail comparison mode register - POR value
    PFT = 0x00;             // power-fail comparison threshold register - POR value
    ADIF = FALSE;

}

// ============================================================================

void cv_cc_control (signed int setpoint, signed int parameter){

    signed int temp;

    temp = setpoint - parameter;
```

```
    if (temp > 0)
        global_pwm_value++;
    else if (temp < 0)
        global_pwm_value--;

    if (global_pwm_value <= PWM_VAL_MIN)
        global_pwm_value = PWM_VAL_MIN;
    else if (global_pwm_value >= 254)
        global_pwm_value = 254;

    write_pwm (global_pwm_value);

}

// ============================================================================

void do_adc_conversions (void){

    char loop_1;
    char loop_2;
    int total;
    static char pointer;

    if (CHARGE_ENABLE == TRUE)  // measure battery current if charger output enabled...
        data_samples[pointer][BATTERY_CURRENT] = get_adc_value(BATTERY_CURRENT);
    else                        // ...otherwise measure battery voltage
        data_samples[pointer][BATTERY_VOLTAGE] = get_adc_value(BATTERY_VOLTAGE);

    data_samples[pointer][CHARGER_VOLTAGE] = get_adc_value(CHARGER_VOLTAGE);
    data_samples[pointer][BATTERY_THERMISTOR] = get_adc_value(BATTERY_THERMISTOR);

    if (++pointer >= AVERAGING_POINTS)
        pointer = 0;

    for (loop_1 = 0; loop_1 < ADC_CHANNELS_USED; loop_1++){
        total = 0;
        for (loop_2 = 0; loop_2 < AVERAGING_POINTS; loop_2++){
            total += data_samples[loop_2][loop_1];
        }
        total /= AVERAGING_POINTS;
        data_samples [AVERAGING_POINTS][loop_1] = total;
    }

    battery_voltage = data_samples [AVERAGING_POINTS][BATTERY_VOLTAGE] * BATT_V_SCL_FACTOR;
    charger_voltage = data_samples [AVERAGING_POINTS][CHARGER_VOLTAGE] * CHG_V_SCL_FACTOR;
    battery_thermistor_value = data_samples [AVERAGING_POINTS][BATTERY_THERMISTOR];
    battery_current = data_samples [AVERAGING_POINTS][BATTERY_CURRENT] * BATT_I_SCL_FACTOR;

    if ((charge_in_progress == TRUE) && (battery_voltage < BATT_REMOVED_LIMIT)){
        charge_in_progress = FALSE;         // then battery removed
        CHARGE_ENABLE = FALSE;              // disable charger
        reset_timer (1);
    }

}

// ============================================================================
// ============================================================================
```

[MEMO]