87C751 fast NiCad charger AN439

DESCRIPTION

This application note describes a portable standalone, automatic constant current NiCad battery charger using the Philips Semiconductors 87C751 microcontroller. This unit will fast charge NiCad batteries from a 12V source such as an automobile battery or a DC power supply. The charge current is 2.5A which is suitable for charging NiCad cells of 1200mAh capacity in little over 1/2 hour.

Use of a microcontroller provides complete flexibility of design parameters such as the number of cells, their capacity and charge rate requirements. A number of key microcontroller techniques described in this application note are a state machine, a low-cost, high-resolution single slope analog-to-digital converter comprised of the microcontroller and a comparator as well as an analog control system.

As shown in the block diagram (Figure 1), the charger consists of the 87C751, a switching charge current regulator, an analog-to-digital converter and some LED indicators.

DESIGN OBJECTIVES

The goal of this design is to achieve a maximum charge rate without damage to the NiCad cells. To determine the requirements for such a charger, we need to examine the most important characteristics of NiCad cells.

As a NiCad cell charges, gas bubbles are released from the electrolyte and accumulate on the plates, reducing the effective plate area and increasing cell impedance. When the cell gets near full charge the rate of gas generation and temperature rise increase, since the charge current produces gas rather than stored charge. At that point the process goes into thermal runaway. The cell pressure rises sharply causing the case to vent. A large enough venting can destroy the cell immediately. If the venting is of a lesser magnitude, as would be the case with a relatively low charge current, the cell capacity is reduced.

The standard technique is to charge the cells at such low current that there is no risk of thermal runaway. The electrolyte then reabsorbs the gas bubble at the same rate as they are generated. Usually this implies a charge current of 0.1 times the cell capacity and a charge time of 16 hours. We want to achieve full charge in about half an hour. Slow charging also increases the likelihood of dendrite formation. Dendrites are crystalline fingers that can propagate through the plate separators and short the cell internally. Fast charging tends to clear these shorts before they have a chance to become significant.

CHARGE TERMINATION

There are two methods commonly used for terminating NiCad fast charges: delta-peak voltage detection and delta-temperature detection. This charger uses the delta-peak voltage method. It is simpler to implement, especially if interchangeable battery packs are to be charged, because the temperature sensing method requires a temperature sensor to be attached to the battery pack. The temperature change in the battery pack depends on how well the pack is thermally coupled to its surroundings, which also makes temperature sensing somewhat tricky.

The voltage on a NiCad pack rises during charging, steeply at first, and then at a lower rate. When the pack is nearly full, the voltage rate of rise increases a little, then falls to zero as the voltage peaks. As the pack goes into over-charge the voltage starts to drop and the internal temperature and pressure rise. A typical voltage drop used for charge termination is 1% of the peak voltage as shown in the battery voltage waveform (Figure 2). This charger uses 1% of the measured peak voltage as the threshold, which means that it can charge any number of cells from 1 to 6 without any external input to select a number of cells.

THE 87C751 MICROCONTROLLER

The Philips Semiconductors 87C751 contains a 2k byte ROM, a 64 byte RAM, 19 I/O lines, a 16 bit auto-reload timer, a five-source fixed-priority interrupt structure, a bidirectional I^2C serial bus interface and an on-chip oscillator.

In this application note, the timer is used as a 'tick-timer', scheduling events and measuring periods. The ports are used to control and monitor the external analog circuitry. The software is written in C using the Franklin C compiler.

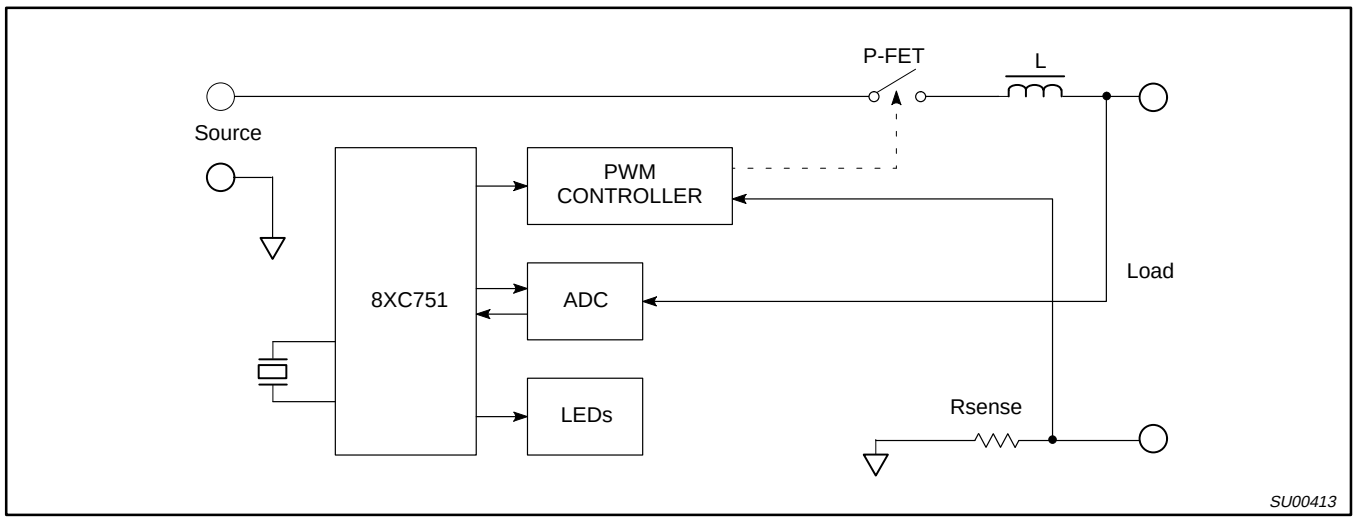

Figure 1. Block Diagram

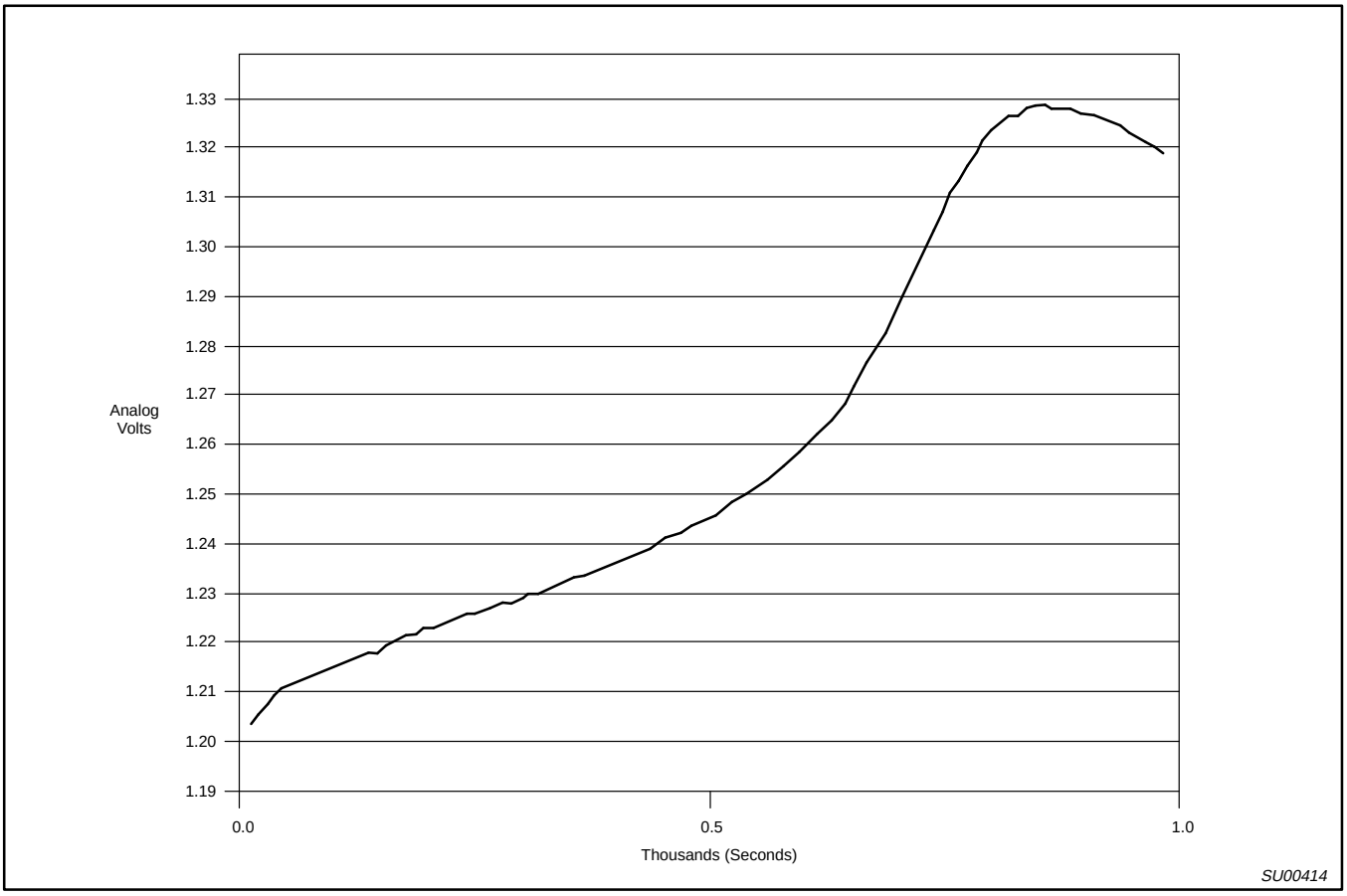

Figure 2. Battery Voltage Waveform During Charging

SOFTWARE

Scheduler

The timer generates events at a fixed interval by updating the variable **tick** in the timer interrupt service routine. The timer is initialized and reloaded with $ffff_{hex}$, which corresponds to a period of 71 milliseconds. The choice of this period is related to the analog-to-digital converter, which is described below.

The variable **tick** is incremented once every 71 ms so that 14 **ticks** make a second. The scheduler calls the **charger** procedure when **tick** equals zero, the **leds** procedure when **tick** equals 7 and it resets **tick** to zero as well as incrementing the **seconds** variable when **tick** reaches 14.

State Machine

The state machine in the **charger** procedure is the heart of the system. The state machine processes the current state and the filtered battery voltage. Figure 3 shows the topology of the state machine. There are four states: FAULT, INITIALIZING, CHARGING and DONE. The variable state keeps track of the current state. The operation of the states is as follows:

FAULT
This is the default state. When the system powers up and is reset, the state variable is initialized with this state. The filtered battery voltage is checked using the **check_limits** procedure (see Figure 3). If the measured voltage is within a predetermined range, it is assumed that a battery has been connected and the state variable is set to the INITIALIZING state. A voltage out of range will always set the state variable to FAULT no matter what the current state is.

INITIALIZING
During this state the running average noise rejection filter and other variables used by the charging algorithm are initialized by sampling the battery voltage for a preset number of passes through the state machine without making any decisions about the charging process. The number of passes is defined in the constant INITIALIZATION_TIME which is an integer number of seconds. When the initialization time is over, the state variable is updated to the CHARGING state.

CHARGING
During this state the battery voltage is processed in the **delta_peak** procedure. A watchdog timer is also used to ensure that the battery is not overcharged in the unlikely event that the **delta_peak** procedure misses the termination conditions. Until the battery peaks or the watchdog timer times out, the state machine remains in the CHARGING state. If a voltage peak is detected and the voltage drops below the peak by 1%, or the watchdog timer expires, the state variable is updated to the DONE state.

DONE
After MAINTENANCE_PERIOD seconds, charging current is applied to the battery for 1 second. If the battery voltage falls out of range (i.e., it has been disconnected), the state variable is updated to FAULT, the default state.

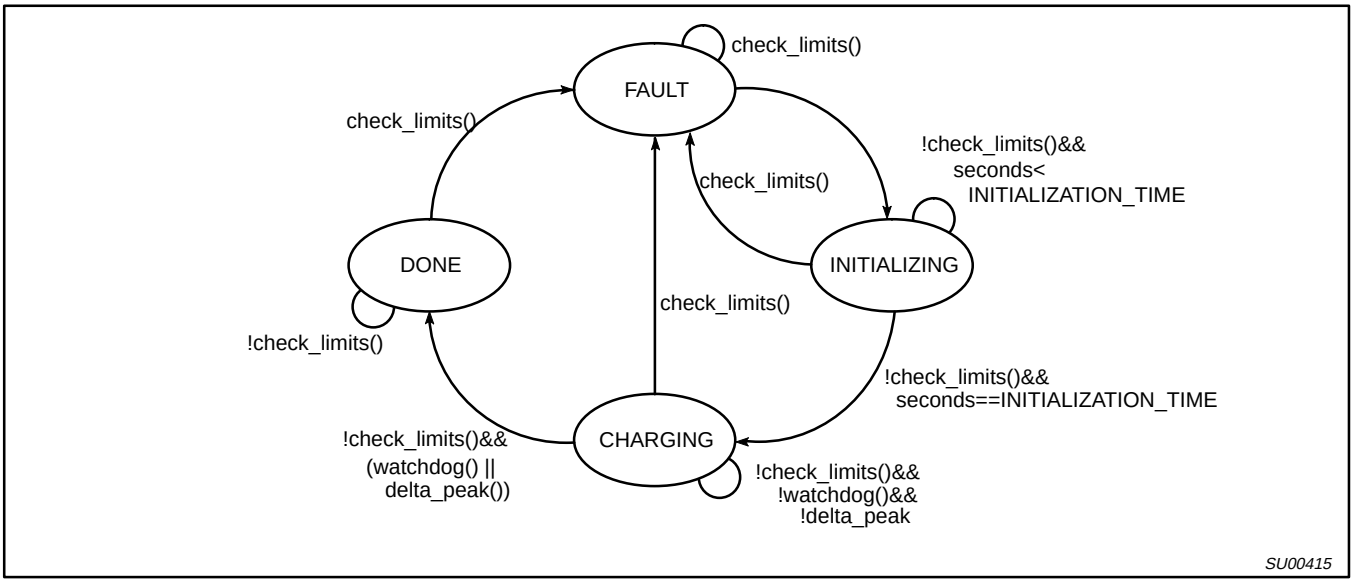

Figure 3. Charger State Machine

HARDWARE

Buck Converter Current Source

There are two basic converter topologies from which the others are derived. They are the buck and boost converters. As the names imply, the buck converter produces an output lower than its input and the boost converter does the opposite. The transformer isolated versions of the buck and boost circuits are known as the forward and flyback converters. Since the input supply voltage is greater than the maximum expected output voltage and no isolation is required, the buck topology is a good choice for simplicity and high efficiency.

The required values of inductors and capacitors for a switching converter are inversely proportional to operating frequency, while switching losses are proportional to frequency, so that a compromise is made in the choice of operating frequency. To keep the converter compact and avoid excessive switching losses, the switching frequency was chosen at 100kHz.

The Unitrode UC3843D is a low-cost surface mount current-mode switching power supply controller. In this application, the voltage feedback loop is left open and the controller is driven from its compensation input (pin 1). The UC3843 data sheet shows that this node in the IC is driven by an active pulldown and a fixed 0.5 mA pullup current source. If the voltage feedback, pin 3, is grounded, the internal voltage error amplifier will turn off its output, allowing control of the node voltage by pulling current out of pin 1.

The ICL7667 is an inverting MOS gate driver. It provides the correct polarity for the mosfet gate signal and its 1.5A peak output improves efficiency by switching the fet quickly. The Amobead is an optional component which reduces EMI at the expense of increased drain voltage swing. The scope traces were taken without the Amobead in place.

Analog-to-Digital Converter

The analog-to-digital converter consists of Q104, C118, R114, U104 as well as R115, R116 and C119 operating in conjunction with the 87C751. In this application a simple, low-cost ADC with relatively high resolution is required, but it does not need linearity, absolute accuracy or long term stability because the detection method is relative to the peak voltage and happens over a period less than one hour. The circuit functions as a voltage-to-period converter. Although the capacitor voltage follows an exponential curve, it is nearly linear in the operating region from 0V to the comparison threshold of 454mV since the battery voltage which drives it is typically 5 to 6V for a four cell pack under charge. With a single cell the period can stretch to near the 71ms tick period. The capacitor voltage is described by the equation

$$V_C=V_{BATT}(1-e^{-t/RC})$$

where C=0.1µF and R=1M in this circuit. The straight line approximation we are using is the tangent to the actual curve at t=0. This can be found by differentiating the above equation with respect to time

$$dVC/dt=V_{BATT}/RC \cdot e^{-t/RC}$$

and setting t=0. Then the expression becomes

$$dV_C/dt=V_{BATT}/RC$$

Integrating the above expression yields a straight line

$$V_C=V_{BATT}/RC \cdot t$$

from the origin to the point (RC,V_{BATT}). If you solve for t using the straight line and V_C=454mV, V_{BATT}=5.50V, you get t=8.255ms. Substituting this back into the first equation and solving for V_{BATT} yields 5.73V, which is within 5% of the actual voltage. The absolute accuracy of the conversion is only important when using the battery voltage measurement to detect a battery connected to the output, versus a short circuit or an open circuit. All the critical sensing is done within 1% of the peak voltage and is relative to the peak.

To avoid contamination of the battery voltage reading by switching noise, the voltage sensing is done during a quite period when the switching current source is turned off by the processor.

Waveforms

The first trace (Figure 4) shows drain to ground voltage on the P-channel FET. Note that the peak negative voltage is –17.66V and there is minimal flyback ringing. The positive spike on turn-on may be due to sense resistor inductance.

The second trace (Figure 5) shows the voltage across the 0.1Ω sense resistor at a DC output current of 2.5A and an input voltage of 12.2V. Voltage spikes due to trace and sense resistor inductance are filtered out by R105, C112 to prevent false triggering of the UC3843 current comparator.

The third trace (Figure 6) shows the FET gate voltage. FET gates are usually rated at ±20V maximum, but reliability is enhanced if the are kept within ±15V. In this case the gate voltage is well within range for reliable operation.

REFERENCES

1. Billings, Keith H.: Switchmode Power Supply Handbook, McGraw Hill 1989
2. Unitrode I.C. Data Handbook, Unitrode Corp. 1990
3. Panasonic NiCad Battery Handbook

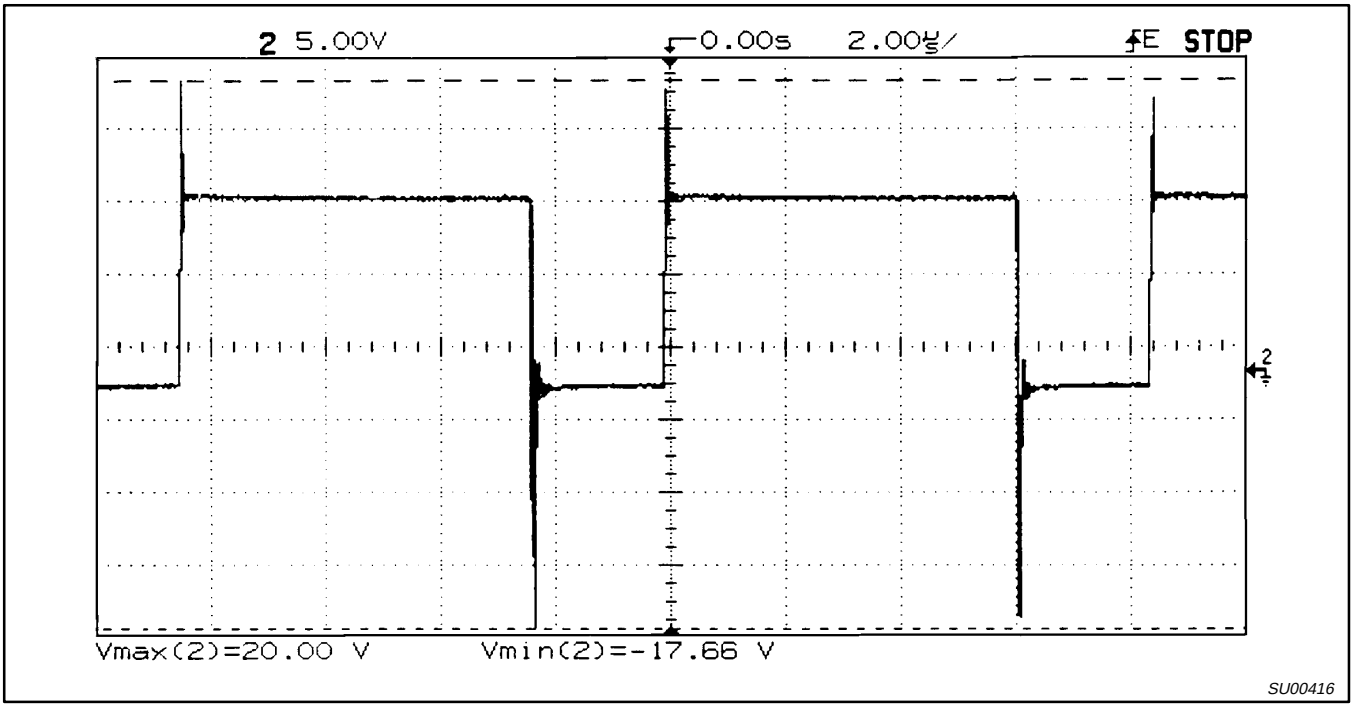

Figure 4. First Trace

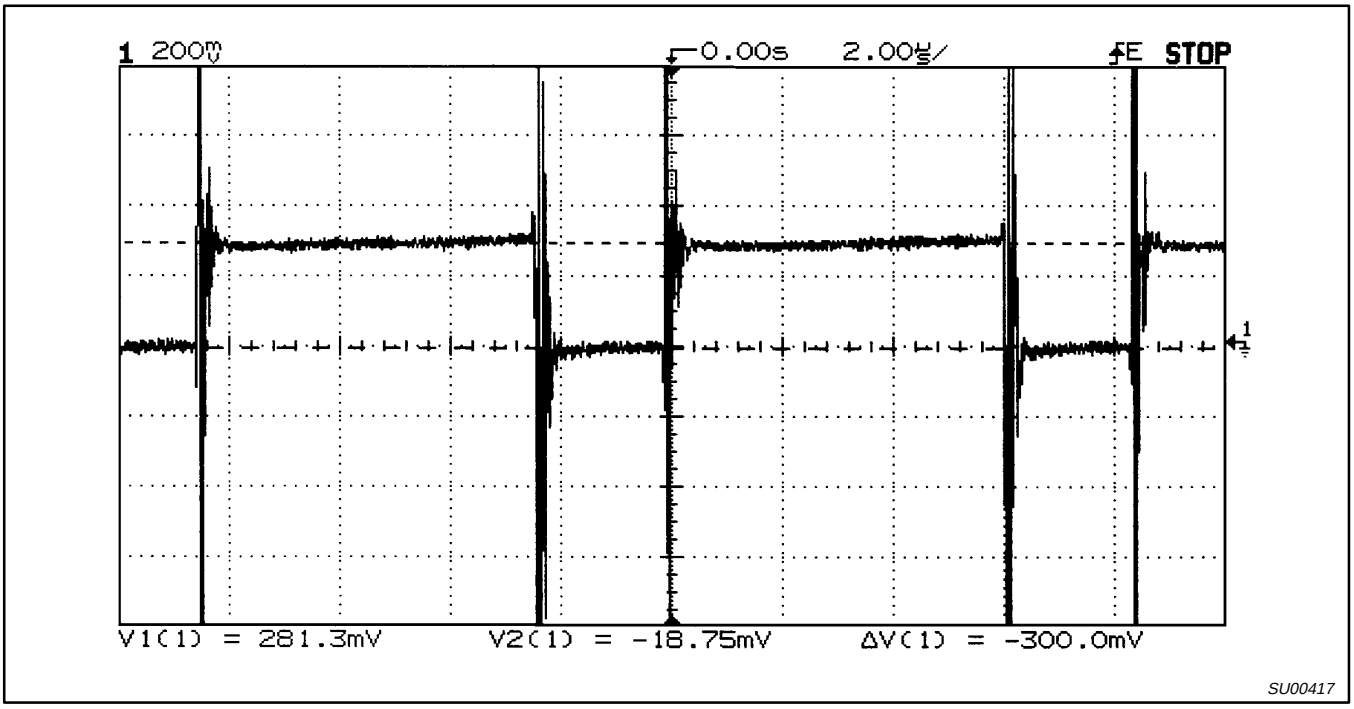

Figure 5. Second Trace

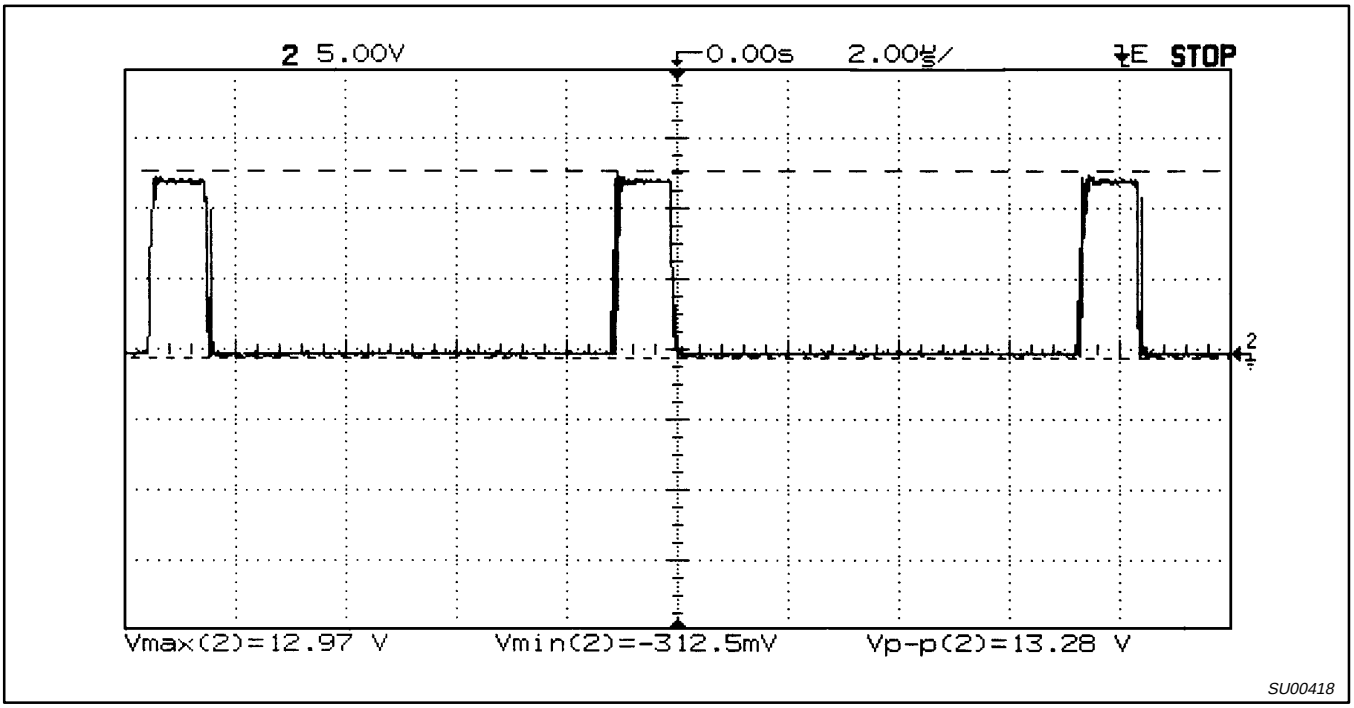

Figure 6. Third Trace

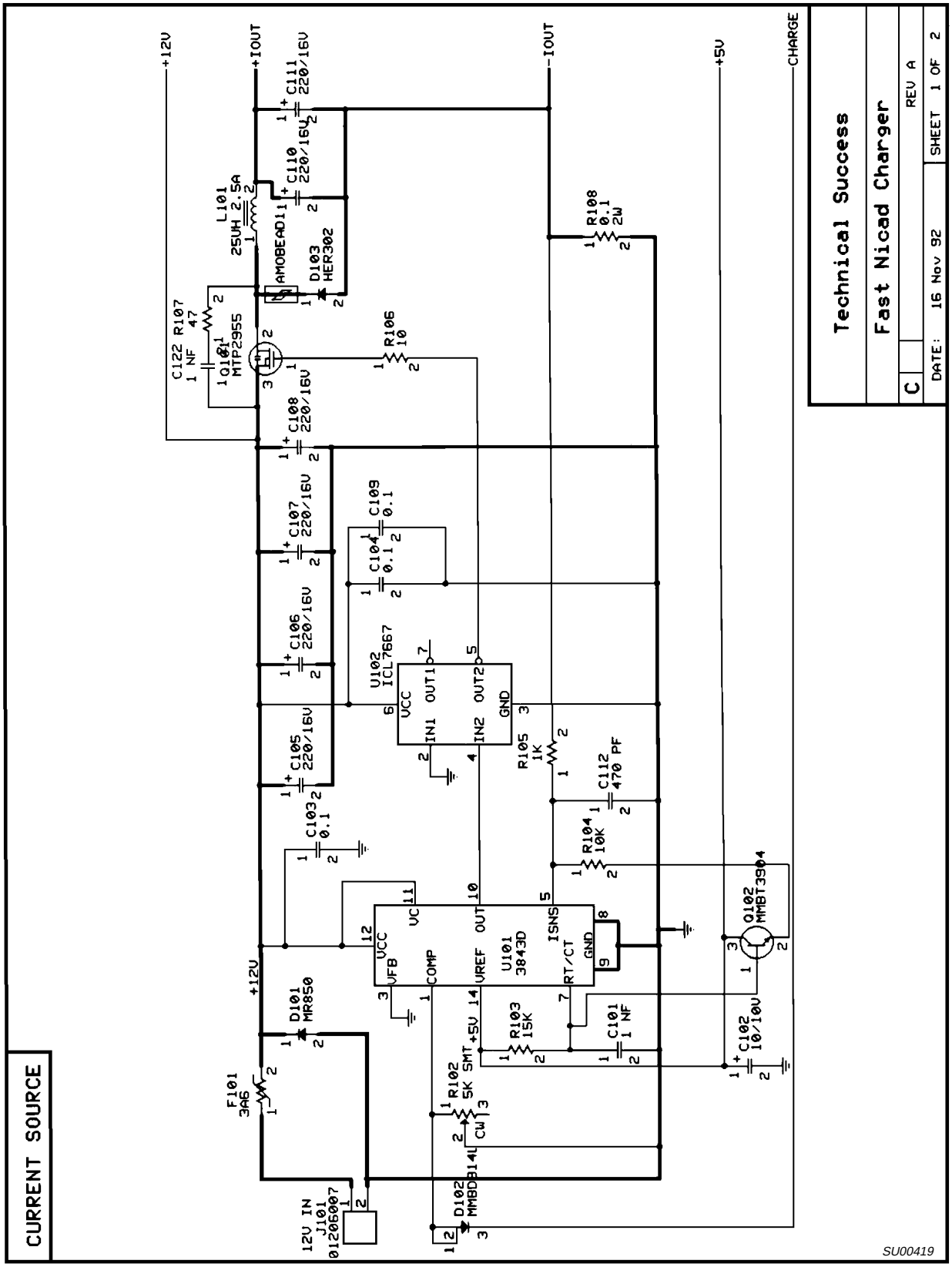

Figure 7. Schematic Diagram of Battery Charger (1 of 2)

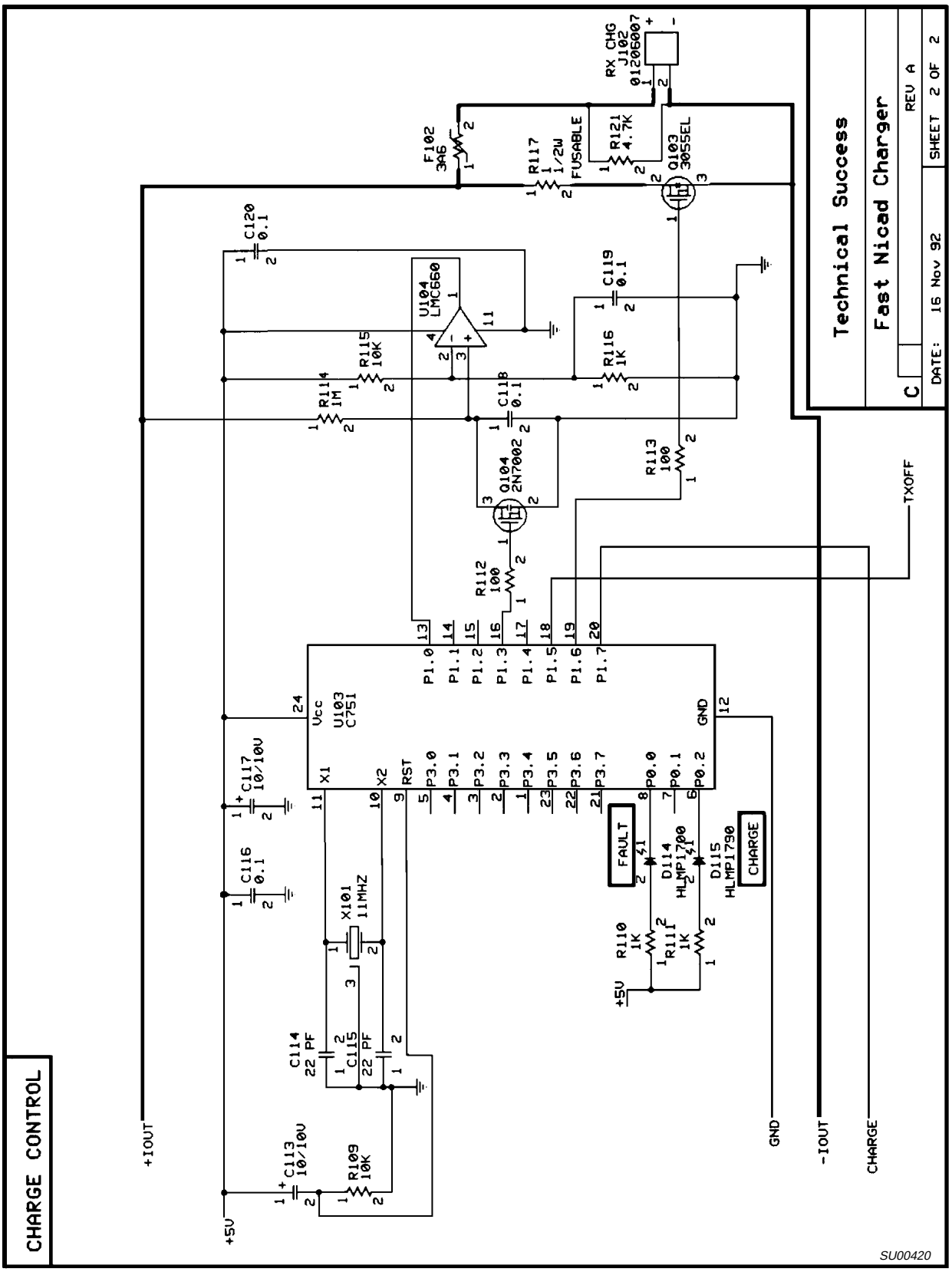

Figure 7. Schematic Diagram of Battery Charger (2 of 2)

```
#pragma PL (60)
#pragma PW (120)
#pragma OT (3)
#pragma ROM (SMALL)

#include <reg751.h>

typedef unsigned char byte;
typedef unsigned short word;
typedef unsigned long dword;

#define TRUE          1
#define FALSE         0
#define ON            0
#define OFF           1
#define FLASH         2

/* parameters */
#define ONE_OVER_DELTA        128
#define ONE_SECOND            14

/* timers */
/* 1 cpu cycle = 1.085uS */
/* 1 tic = 0xffff cpu cycles */
/* 1 ”second” = 14 tics = 995.5uS */
#define MAX_BATT_PERIOD              5530   /* 60ms period, 0.9 volts */
#define MIN_BATT_PERIOD       430    /* 4.67ms period, 10 volts */
#define DEAD_MAN_TIMEOUT      2712   /* 45 minutes */
#define MAINTENANCE_PERIOD    500    /* 1 pulse in 500 Seconds */
#define INITIALIZATION_TIME   20

/* errors */
#define BATTERY_VOLTAGE_OUT_OF_RANGE 0x01
#define BATTERY_CHARGED                     0x02
#define DM_TIMEOUT                   0x03

/* states */
#define FAULT          0x01
#define INITIALIZING   0x02
#define CHARGING       0x03
#define DONE           0x04

/* global variables */
byte           tic;
byte           state;
word           seconds;
word           this_period;
word           last1;
word           last2;
word           last3;
word           last4;
word           last5;
word           last6;
word           last7;
word           valley;

sbit    comp_out       = 0x90;        /* P1.0 */
sbit    clear_cap      = 0x93;        /* P1.3 */
sbit    charge         = 0x97;        /* P1.7 */
sbit    fault_led      = 0x80;        /* P0.0 */
sbit    charge_led     = 0x82;        /* P0.2 */
```

```
word
measure_batt(void) {
        byte    tic_now;
        word    interval;

        tic_now = tic;

        interval = 0;
        clear_cap = FALSE;
        while(!comp_out && tic==tic_now)
                interval++;
        clear_cap = TRUE;
        return (interval);
}

byte
check_limits(
        word    batt_period
         ) {

        if ((batt_period > MIN_BATT_PERIOD)&&(batt_period < MAX_BATT_PERIOD)) {
                return(FALSE);
        }
        else {
                return(BATTERY_VOLTAGE_OUT_OF_RANGE);
        }
}

word
filter (
        word    last0
        ) {

        word temp1, temp2, temp3, temp4;
        word result1, result2;

        temp1 = ((last0 / 2) + (last1 / 2));
        temp2 = ((last2 / 2) + (last3 / 2));
        temp3 = ((last4 / 2) + (last5 / 2));
        temp4 = ((last6 / 2) + (last7 / 2));

        result1 = ((temp1 / 2) + (temp2 / 2));
        result2 = ((temp3 / 2) + (temp4 / 2));

        last7 = last6;
        last6 = last5;
        last5 = last4;
        last4 = last3;
        last3 = last2;
        last2 = last1;
        last1 = last0;
        return((result1 / 2) + (result2 / 2));
}

byte
delta_peak (
        word    period
        ) {
        if (period < valley)
                valley = period;
        if (period > (valley + (valley/ONE_OVER_DELTA)))
                return (BATTERY_CHARGED);
        else
                return (FALSE);
}

byte
watchdog (
        word    now
        )
```

```
{
        if (now < DEAD_MAN_TIMEOUT)
                return (FALSE);
        else
                return (DM_TIMEOUT);
}
void
charger ( void ) {
        this_period = measure_batt();
        this_period = filter(this_period);
        switch (state) {
                case FAULT: {
                        if (!check_limits(this_period)) {
                                seconds = 0;
                                state = INITIALIZING;
                        }
                        else
                                state = FAULT;
                        break;
                }
                case INITIALIZING: {
                        if (check_limits(this_period))
                                state = FAULT;
                        else {
                                charge = TRUE;
                                if (seconds < INITIALIZATION_TIME)
                                        state = INITIALIZING;
                                else {
                                        valley = 0xffff;
                                        state = CHARGING;
                                }
                        }
                        break;
                }
                case CHARGING: {
                        if (check_limits(this_period))
                                state = FAULT;
                        else {
                                if (!watchdog(seconds)) {
                                        if (delta_peak(this_period)) {
                                                state = DONE;
                                                seconds = 0;
                                        }
                                        else {
                                                state = CHARGING;
                                                charge = TRUE;
                                        }
                                }
                                else {
                                        state = DONE;
                                        seconds = 0;
                                }
                        }
                        break;
                }
                case DONE: {
                        if (check_limits(this_period))
                                state = FAULT;
```

```
                        else {
                                if(seconds < MAINTENANCE_PERIOD) {
                                        charge = TRUE;
                                        seconds = 0;
                                }
                                state = DONE;
                        }
                        break;
                }
        }
}
void
leds ( void ) {
        switch (state) {
                case FAULT: {
                        charge_led = OFF;
                        fault_led = ON;
                        break;
                }
                case INITIALIZING: {
                        charge_led = ON;
                        fault_led = OFF;
                        break;
                }
                case CHARGING: {
                        charge_led = ON;
                        fault_led = OFF;
                        break;
                }
                case DONE: {
                        charge_led = !charge_led;
                        fault_led = OFF;
                        break;
                }
        }
}
/* Timer 0 interrupt */
void
timer0(void) interrupt 1 {
        tic++;
}
void
main()
{

        /* initialize pins */
        charge=FALSE;
        charge_led = OFF;
        fault_led = OFF;
        /* initialize timer */
        TR=1;
        CT=0;
        GATE=0;
        RTH=0;
        RTL=0;
        IT0 = TRUE;
        ET0=1;
        EA=TRUE;
        /* initialize globals */
        tic=0;
        seconds = 0;
        valley = 0xffff;
        state = FAULT;
```

```
        /* main program scheduler */
        while(1) {
                switch (tic) {
                        case 0: {
                                charger();
                                while (tic < 1);
                                break;
                        }
                        case 7: {
                                leds();
                                while (tic < 8);
                                break;
                        }
                        case ONE_SECOND: {
                                tic = 0;
                                seconds++;
                                break;
                        }
                }
        }

}
```