SILICON LABORATORIES

AN145

THREE-CHANNEL POWER SEQUENCER

Relevant Devices

This application note applies to the following devices: C8051F330.

Introduction

The C8051F330 can provide low-cost power sequencing and supervision in systems with up to three power supply rails. During power ramp up, it controls in-rush current by limiting the output slew rate and performs real-time tracking to minimize the voltage difference between supply rails. Once ramping is complete, it monitors the outputs and shuts down all supply rails if an over-voltage, under-voltage, over-current, or single-rail failure condition is detected.

This reference design includes:

- Background and theory of operation.
- Hardware and software description including how to customize and use the firmware.
- Typical performance examples.
- A schematic, bill of materials, and PCB area estimate for a three rail solution.
- Complete firmware (source and object code included) that can be used as-is for a three rail solution. A two rail solution can be achieved by turning off the third rail.

Background

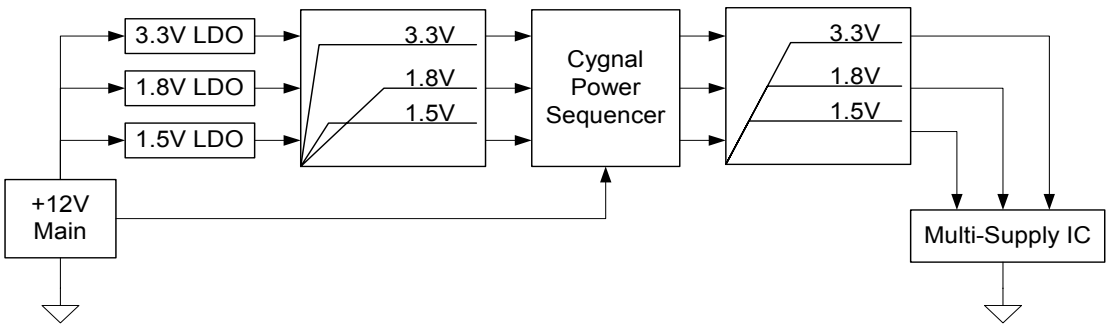
An increasing number of devices such as FPGAs, DSPs, RISC-CPUs, and communications ICs operate at two or more supply voltages. The core logic commonly operates from 1.3V to 2.5V in order to save power while the I/O supply operates from 3.3 to 5 V to interface with other devices.

A common requirement for these ICs is that supply rails should track while ramping and stabilize at their target voltages 100 ms or more before the device is brought out of reset. Also, these devices often require large decoupling capacitors. Controlling in-rush current by limiting the slew rate on power up can prevent damage to the decoupling capacitors.

Many systems with multiple supplies require system health monitoring and protection. The health monitor checks for overvoltage, undervoltage, or overcurrent on any of the supply rails. It provides protection by putting the system into reset and shutting down all supply rails if abnormal conditions are detected on any of the supply rails.

The Silicon Labs Power Sequencing Solution is ideal for multi-supply systems where the reliability requirements or IC replacement cost are key issues.

Figure 1. Multi-Supply IC that Requires Power Sequencing and Management



The C8051F330 MCU is available in a 20-pin MLP with a 4x4mm footprint. The entire solution including power MOSFETs can be implemented in less than 1 square inch of PCB space.

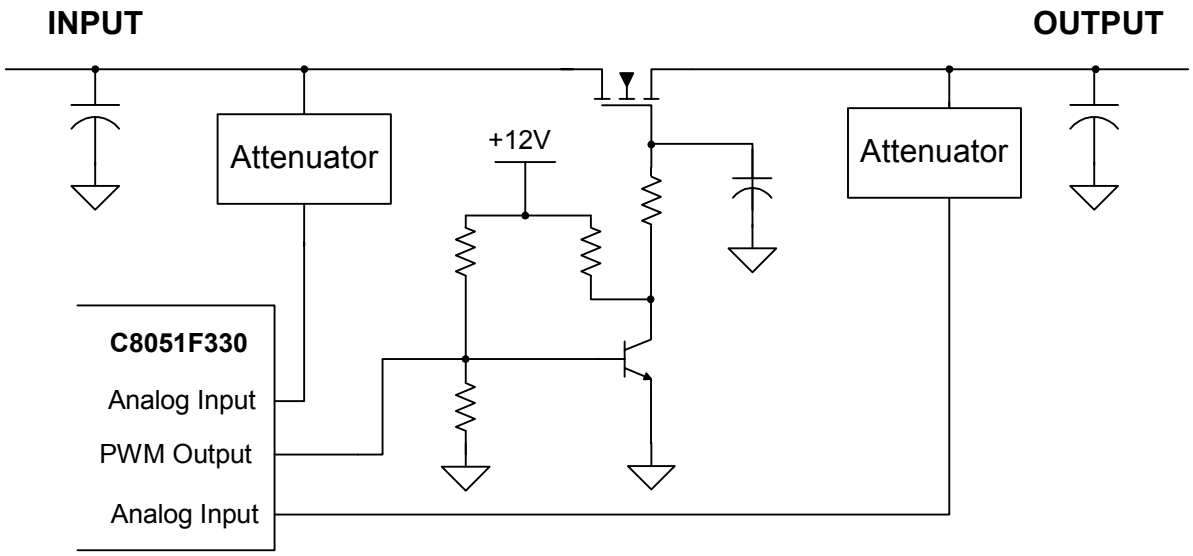
Theory of Operation

The Silicon Labs power sequencer uses its on-chip ADC and PWM output capabilities to implement a state-machine-based feedback control loop. This control loop performs real-time voltage tracking during ramp up and provides power supply monitoring and protection once ramping is complete. It also manages the "system reset" (S_RESET) and "power good" (POWER_G) signals.

The power sequencer controls the output voltages by varying the source-drain resistance of a power MOSFET placed between the input and output terminal. The MOSFET resistance can vary from being an open circuit to a virtual short circuit, based on the gate voltage. The power sequencer controls the MOSFET resistance using a low-pass filtered PWM signal at the gate, as shown in Figure 2.

Using its on-chip ADC, the power sequencer measures the input and output voltages for each channel. This allows it to perform real-time voltage tracking on power up and provide protection from over-voltage, under-voltage, over-current, or single-rail failure conditions after all channels have stabilized.

Figure 2. Single Channel Model



System States

The power sequencer has four states of normal operation, shown in Figure 3. On reset, the system starts normal operation unless the calibrate/shutdown (CAL/SD) switch is pressed. If the CAL/SD switch is pressed, the system enters Configuration mode as shown in Figure 4.

Input Validation

The VALIDATE state monitors the channel input voltages until they are within specified operating ranges. It also checks if the 12V supply is powered. Once all supply rails are "on", it waits a programmable wait time (10 ms - 30 sec with a default of 100 ms) for the supplies to stabilize. At the end of the wait time, the system measures and records the actual supply voltages at the channel inputs. This measurement is used by the RAMP state to determine when ramping is complete.

Ramping Algorithm

While ramping, the outputs rise at a programmable monotonic slew rate until each of the channel outputs reaches its target voltage. The slew rate can be programmed to any value from 250 V/s to 500 V/s.

Figure 3. System State Definitions

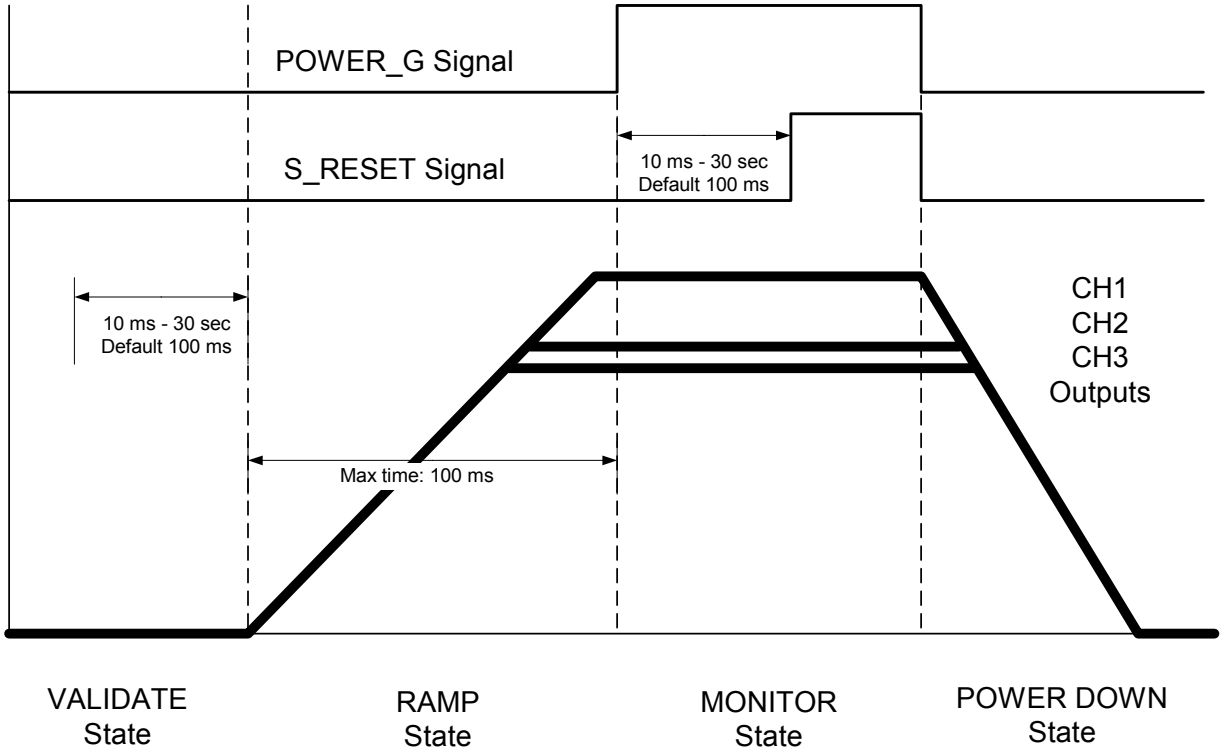
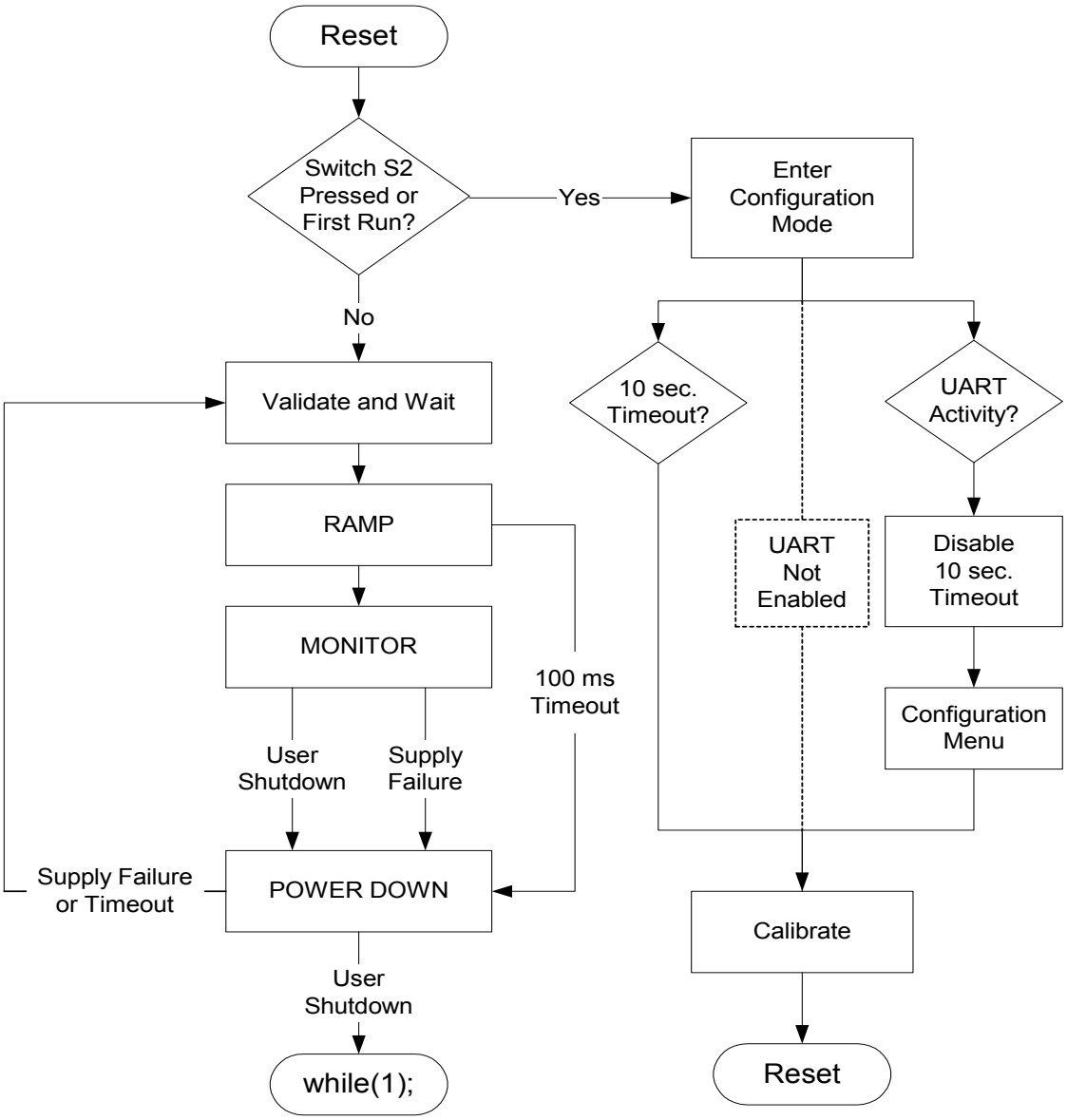


Figure 4. System State Flow Diagram



The ramping algorithm keeps the output voltages as close as possible to a calculated "ideal" line, as shown in Figure 5. At each sampling point, the algorithm compares the output voltage to the "ideal" line and decides whether to hold or increment the output voltage and the ideal line. Figure 5 shows that the output voltage for a single channel can be much lower, slightly lower, or higher than the "ideal" line.

The three cases are discussed below:

Case 1. The output voltage is much lower than the "ideal" line (see Point 1 in Figure 5). **Action Taken.** Increase output voltage and hold the ideal line.

Case 2. The output voltage is slightly lower than the "ideal" line (see Point 2 in Figure 5).
Action Taken. Increase output voltage and increase the ideal line.

Case 3. The output voltage is higher than the "ideal" line (see Point 3 in Figure 5).
Action Taken. Hold output voltage and increase the ideal line.

The ramping algorithm also checks the differential voltage between each channel and the other channel(s). As shown in Figure 6, if a channel starts lagging (i.e. the differential voltage between the channel and another channel has exceeded the tracking threshold), the remaining channel(s) (and their ideal lines) are held at their current values until the slower channel "catches up".

Once ramping is complete, the power good (POWER_G) signal is driven HIGH and the system moves to the MONITOR state. If the system does not finish ramping before the 100 ms timeout, all outputs are shut down and the inputs are re-validated.

System Monitoring and Protection

In the MONITOR state, channel inputs and outputs are monitored to provide protection from overvoltage, undervoltage, overcurrent, and single rail failure conditions.

The system reset (S_RESET) signal is driven HIGH a programmable wait time after the system

Figure 5. Example RAMP State Decisions

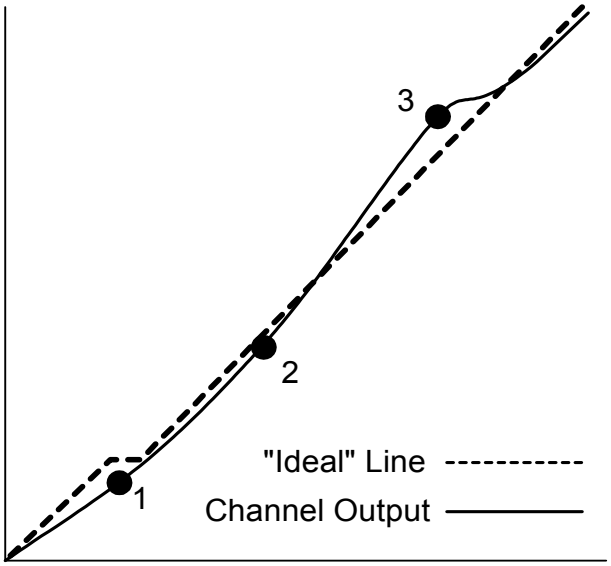
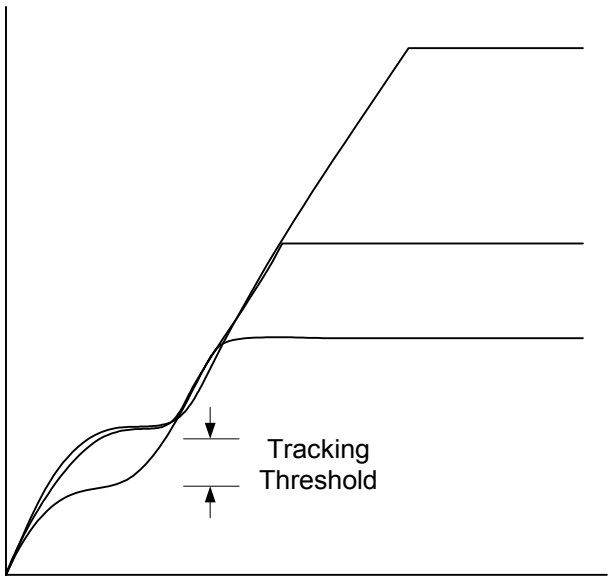


Figure 6. Example Tracking Decision



has entered the MONITOR state (10 ms to 30 sec with a default wait time of 100 ms).

Overvoltage and undervoltage conditions are detected when the output voltage for a channel has exceeded or fallen below 8% of the channel voltage (3.3V, 1.8V, or 1.5V). Overcurrent is detected when the input and output voltages vary by greater than 400 mV on any channel.

If any of the above conditions is detected on any rail, the S_RESET and POWER_G signals are driven LOW and the system powers down and re-validates the inputs. A 100 mV hysteresis on re-validation prevents the system from continuously ramping up and down if a supply voltage is hovering near the overvoltage or undervoltage threshold.

While in the MONITOR state, the user can issue a User Shutdown by pressing the CAL/SD switch. A User Shutdown initiates a soft ramp down on all outputs and puts the CPU in a low power Stop mode until the next reset or power-on event.

Soft Power Down

The Power Down State is entered when the 100 ms ramp timeout has expired, a power failure is detected, or a User Shutdown has occurred.

In this state, the POWER_G and S_RESET signals are driven LOW. The channel outputs are ramped down to provide a “soft” shutdown. The outputs track based on stored calibration data.

Hardware Description

In this design, PCA0 is used to generate the PWM signals that control the power MOSFET resistance. ADC0 is used to measure the voltages at the inputs and outputs.

A complete schematic for a 3 channel system using the C8051F330 is shown in Appendix A on page 20. A bill of materials for this system is shown in Appendix B.

PWM Generation

The Program Counter Array (PCA) in 8-bit PWM mode generates three PWM signals used to control the output voltages. These signals are named CEX0, CEX1, and CEX2. The frequency of the PWM signals is configured to 95.7 kHz, or 256 SYSCLK cycles between falling edges. The SYSCLK frequency is 24.5 MHz, derived from the calibrated internal oscillator.

The duty cycle of the PWM signal is set by writing an 8-bit PWM code to the high byte of a PCA Capture Module Register (PCA0CPH0, PCA0CPH1, or PCA0CPH2), as described in the PCA chapter of the C8051F33x Datasheet.

Power MOSFET Control

The power MOSFET resistance decreases as the gate voltage increases. When the gate voltage is 0V, the MOSFET is an open circuit. As the gate voltage moves closer to 12V, the MOSFET resistance decreases until it becomes a virtual short.

The NPN Transistor and +12V pull-up resistor translate the 0V to 3.3V PWM signal to a 0V to 12V PWM signal. This signal is low pass filtered before reaching the MOSFET gate. This reduces ripple on the output voltage during the ramp phase. Once ramping is complete, the MOSFET is turned completely on.

ADC Sampling

The on-chip 10-bit ADC0 is configured to sample at a rate of 47.85 kHz, exactly twice the PWM frequency. Synchronizing the ADC sampling with the PWM signal reduces digital noise in the ADC samples.

ADC0 starts conversions on Timer 2 overflow. During device initialization, Timer 2 and the PCA counter are started together so that ADC samples are always aligned with the falling edge of the PWM signal.

The ADC positive input MUX determines which analog input is currently being sampled. The MUX can be changed on-the-fly and is managed by the *Timer2_ISR* during the RAMP and MONITOR states. In all other states, MUX switching is handled by polled code.

Software Description

The main power sequencing and monitoring control loop in this design is implemented in software. Figure 7 shows an overview of program flow starting from a device reset.

Device Calibration

On first run, the Silicon Labs power sequencer goes through a calibration sequence to characterize the external circuitry. During calibration, the output voltage is measured at each PWM code and a calibration table is built in FLASH mapping PWM codes to output voltages. This allows the output waveform to be controlled in 50 mV steps regardless of MOSFET device-to-device variations or differences in channel loading.

The initial calibration ramp rises slower than a a typical ramp in normal operation. After all channels reach their target value and the calibration table is complete, the CAL_DONE flag is cleared to indicate a successful calibration and a software reset is issued. Throughout the calibration sequence, the S_RESET and POWER_G signals are held LOW.

After device initialization, the C8051F330 verifies the calibration data stored in non-volatile FLASH memory. If the CAL_DONE byte reads 0xFF, the value of uninitialized FLASH memory, verification will fail and the device will re-enter the calibration sequence.

Variable Initialization

Global variables are used to share data and system state information between polled code and interrupt code. The initializations are performed each time the system enters the VALIDATE state.

VALIDATE State

The VALIDATE state is implemented in polled code by the *ValidateInput()* routine. It first verifies that the 12V supply is turned on. Next, it monitors the channel inputs until they are all within their overvoltage and undervoltage thresholds.

After validation, the system pauses for a programmable wait time *<VAL_WAITTIME>* using the *wait_ms()* support routine to allow the inputs to settle. The *wait_ms()* routine uses Timer 3 to pause polled code for a given number of milliseconds. Interrupts can be serviced while polled code is paused.

After all channel inputs have settled, the *ValidateInput()* routine records the ADC code measured from each channel in its corresponding *<CHx_TARGET_CODE>* variable. These variables are used during the RAMP state to determine when ramping is complete.

RAMP State

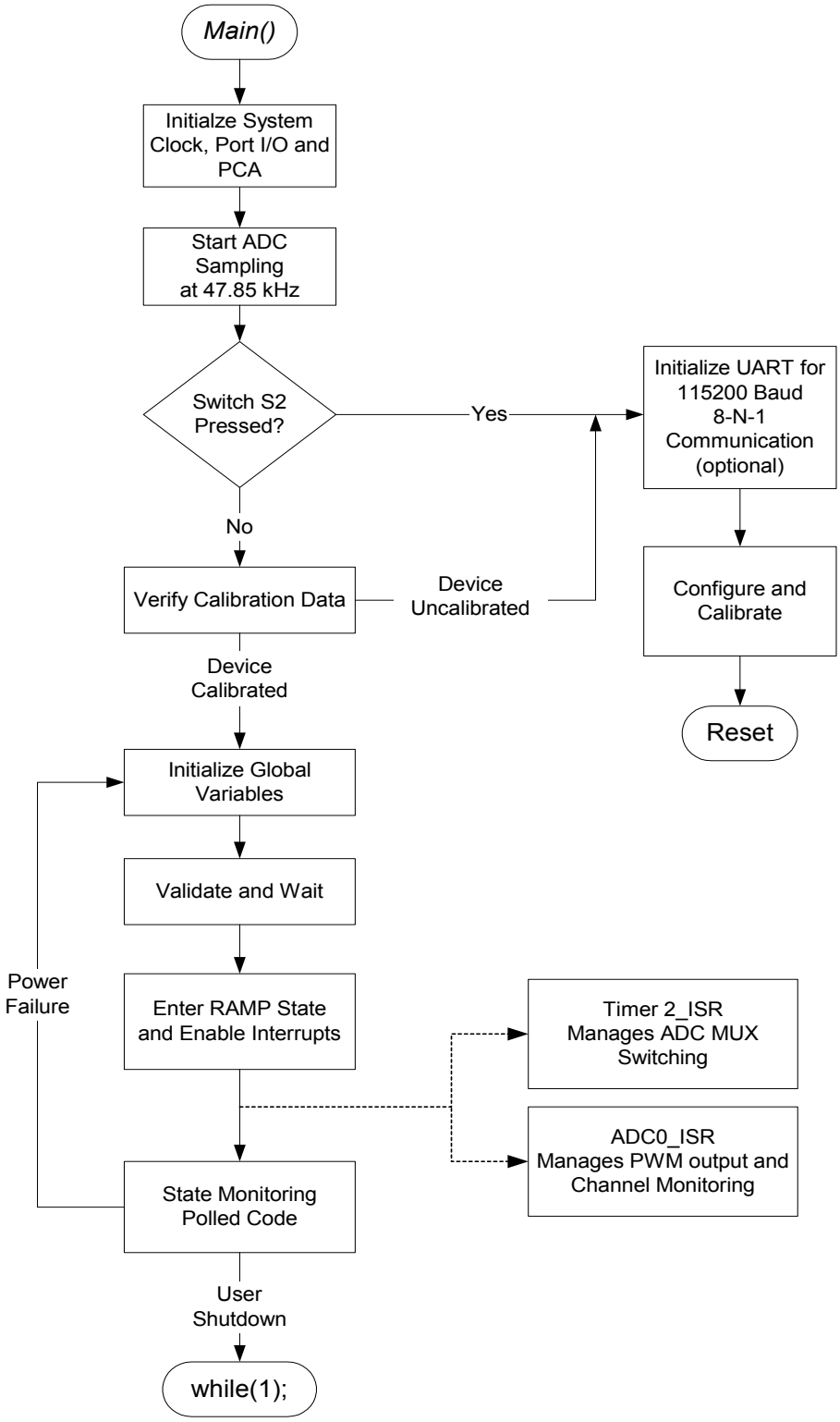
The RAMP state is implemented in interrupt driven code using *Timer2_ISR* and *ADC0_ISR*. *Timer2_ISR* cycles the ADC positive input MUX through the channel outputs on Timer 2 overflows. Note that the Timer 2 overflow event also starts a new ADC conversion before the ADC MUX is changed.

Figure 8 shows program flow in the ADC0_ISR when the system is in the RAMP state. The ADC0_ISR handles tracking and ramping decisions made while the outputs are rising. These decisions are shown in Figure 5 and Figure 6 and explained on page 5.

Once ramping is complete, the PWM signals are parked LOW, making the MOSFET a virtual short. The system state is changed to MONITOR.

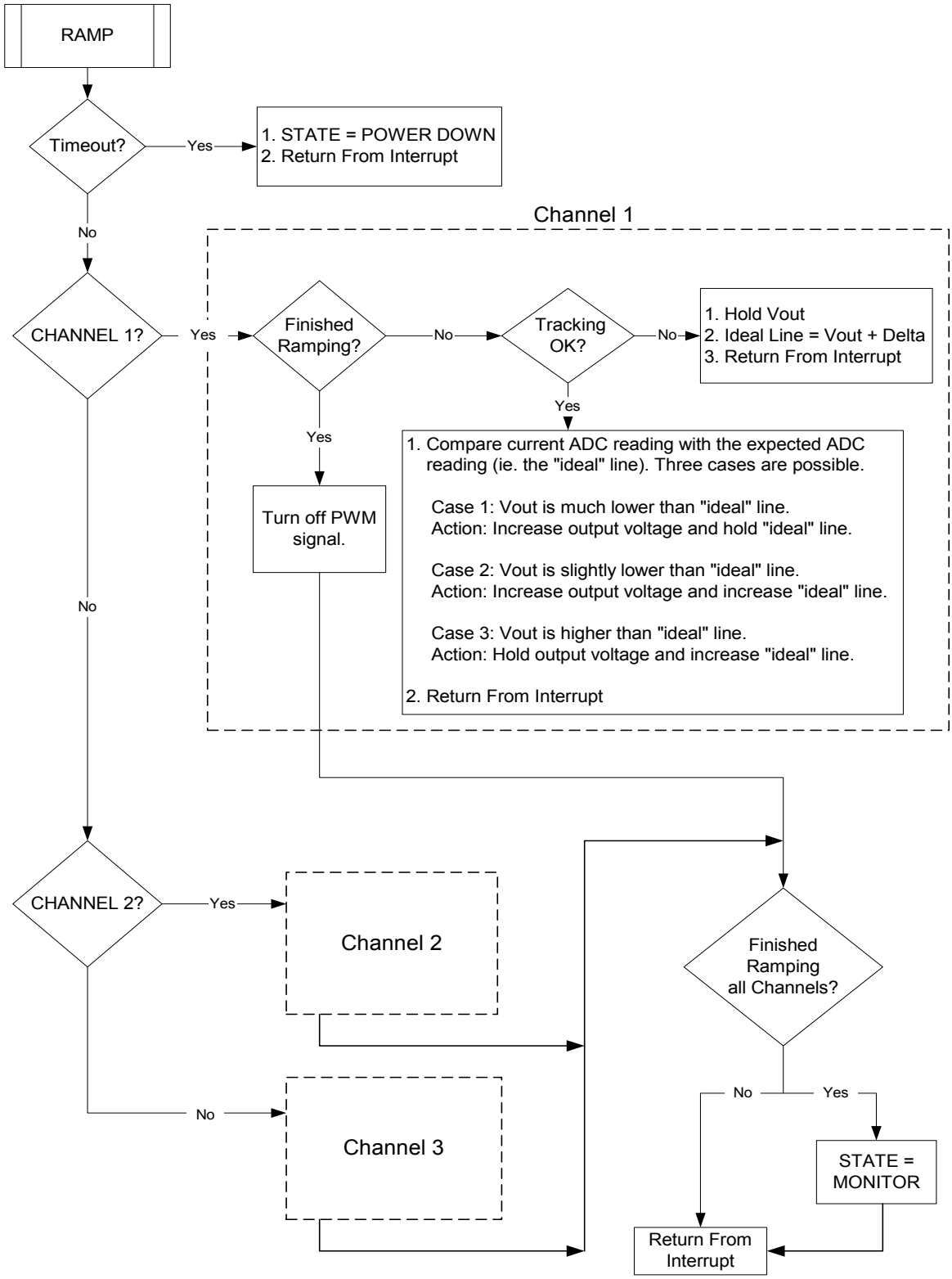
MONITOR State

Figure 7. Software Flow Diagram



In the MONITOR state, *Timer2_ISR* cycles the ADC positive input MUX through the channel outputs

Figure 8. RAMP State of ADC0_ISR



and inputs. This allows ADC0_ISR to detect overvoltage, undervoltage, overcurrent, and single rail failure conditions.

A programmable wait time <MON_WAITTIME> after the system has entered the MONITOR state, the S_RESET signal is de-asserted by state-monitoring polled code executing in the *main()* routine.

Figure 9 shows program flow in *ADC0_ISR* when the system is in the MONITOR state. If a power failure or User Shutdown is detected, *ADC0_ISR* sets the system state to POWER DOWN.

POWER DOWN State

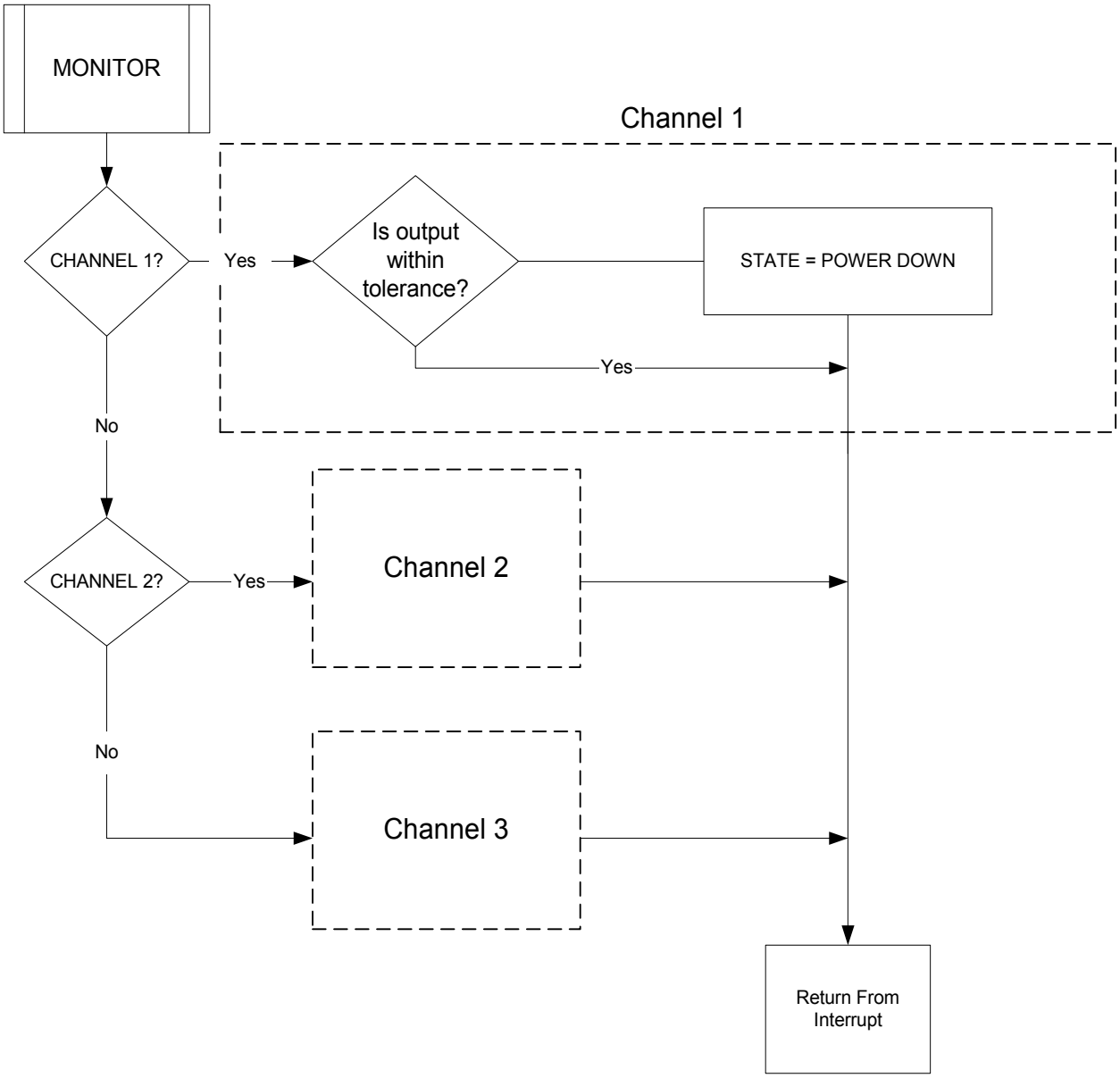
In the POWER DOWN State, the outputs ramp down at a rate of approximately 250 V/s, managed by the ADC0_ISR.

The 3.3V channel starts ramping down first. Once it has fallen to 1.8V, both the 3.3V and 1.8V ramp down until they reach 1.5V. From this point, all three channels ramp down until all outputs are turned off.

Once all outputs are turned off, state-monitoring code executing in the *main()* routine restarts the validation process or puts the CPU in Stop mode until the next reset.

How to Configure the Firmware

Figure 9. MONITOR State of ADC0_ISR



The firmware provided in APPENDIX A is ready for use, as-is, in an end system. The firmware consists of two files: *PS_V1.3.h* and *PS_V1.3.c*, and can be built using the KEIL C51 development tools. The object code for the C8051F330 in HEX format is included in the file *PS_F330_V1.3.0.hex*.

Table 1 describes system-level parameters located in *PS_V1.3.h* that can be modified to customize the firmware.

Table 1. System Parameters Defined in the *PS_V1.3.h* Header File

Constant	Factory Setting	Description
F330	1	Specifies that the Target MCU is a C8051F330.
UART_ENABLE	0	Enables ‘1’ or disables ‘0’ configuration over UART. When disabled, the system parameters are specified at compile time.
THREE_CHANNEL	1	Enables ‘1’ or disables ‘0’ the third channel. When the third channel is disabled, it is not validated and its output remains at 0V.
DEFAULT_RAMP_RATE	500	Default maximum slew rate (in V/s) on power up.
DEFAULT_VAL_WAITTIME	100	Default time (in ms) between inputs validated and the start of ramping.
DEFAULT_MON_WAITTIME	100	Default time (in ms) between outputs valid (POWER_G rising) and the S_RESET rising edge.
OVERVOLTAGE_PROTECTION	1	Enables ‘1’ or disables ‘0’ overvoltage protection.
OVERCURRENT_PROTECTION	1	Enables ‘1’ or disables ‘0’ overcurrent protection.
RAMP_TIMEOUT_ENABLE	1	Enables ‘1’ or disables ‘0’ the ramp timeout. When the ramp timeout is enabled, the outputs will shut down if all channels have not reached their target voltage before the timeout occurs.
RAMP_TIMEOUT	100	Maximum time (in ms) allowed for ramping.
NUM_RETRIES	3	Maximum number of power up attempts allowed after the first power failure.

Table 1. System Parameters Defined in the *PS_V1.3.h* Header File

Constant	Factory Setting	Description
CH1_VTARGET_MIN CH2_VTARGET_MIN CH3_VTARGET_MIN	3036L 1656L 1380L	Determines the undervoltage threshold (in mV) for a channel. These constants must end in the letter 'L'. ex. 3036L for a 3036 mV undervoltage threshold
CH1_VTARGET_MAX CH2_VTARGET_MAX CH3_VTARGET_MAX	3564L 1944L 1620L	Determines the overvoltage threshold (in mV) for a channel. These constants must end in the letter 'L'. These constants are ignored if overvoltage protection is disabled.
OVERCURRENT_VTH	400L	Determines the overcurrent threshold in mV. If the voltage drop between the input and output side of the MOSFET on any channel exceeds this threshold after ramping is complete, an overcurrent condition will be detected. This constant must end in the letter 'L'.
STRICT_VAL_DELTA	100L	Determines the Power-Fail Hysteresis. Overvoltage threshold is decreased and undervoltage threshold in increased by this amount (in mV) after a power failure. This constant must end in the letter 'L'.
R10 R11	2800L 5360L	The value of resistors R10 and R11 (3.3V channel input and output voltage attenuators and corresponding output resistors) in Ohms. These constants must end in the letter 'L'.

Performance Examples

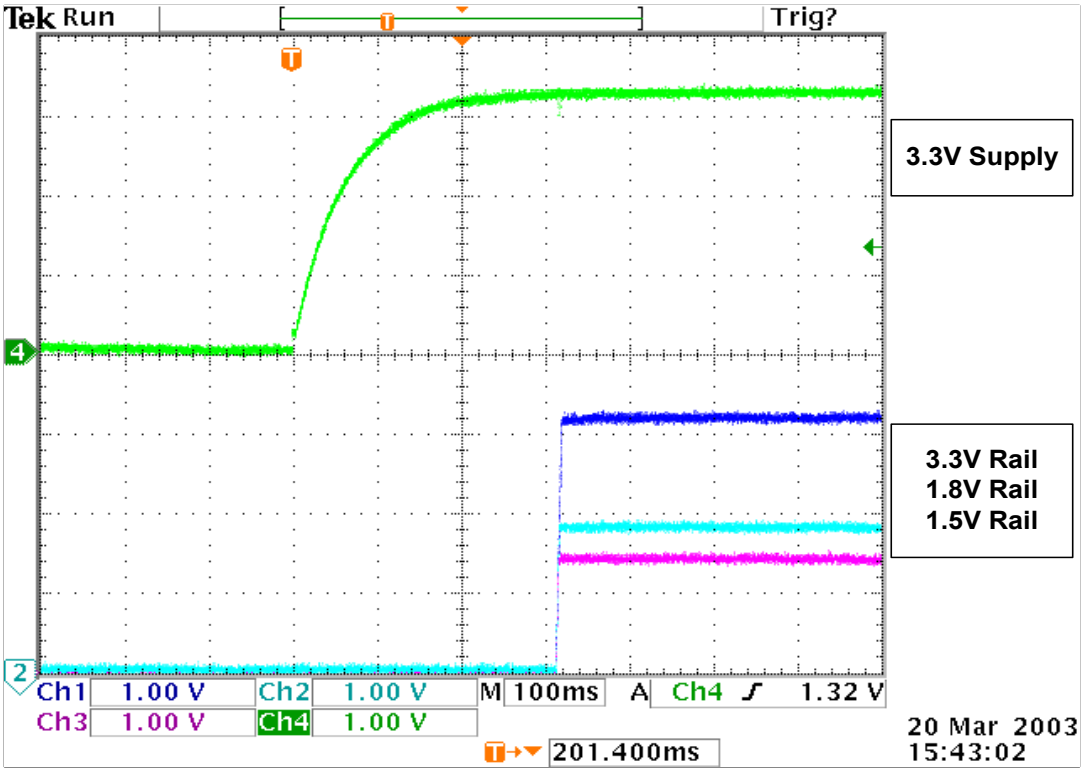
Features

- Low cost 25 MIPS FLASH MCU in a 4x4 mm 20-pin MLP. PCB area requires less than 1 square inch for entire design, including power MOSFETs.
- Two or three channel power supply sequencer/supervisor with real-time voltage tracking on ramp up and soft shut down.
- Adjustable monotonic slew rate from 250 V/s to 500 V/s.
- "System Reset" and "Power Good" signals with adjustable time-outs.
- Devices calibrate on first-run to compensate for MOSFET device-to-device variations and differences in channel loading.
- UART Interface for optional reconfiguration.

Power-up Example

Figure 10 shows the behavior of the outputs as the 3.3V supply is turned on. The C8051F330 is powered from the input side of the 3.3V supply. The 1.5V, 1.8V, and 12V supplies are available prior to the 3.3V supply in this example. Note that the system can tolerate the supplies rising in any order.

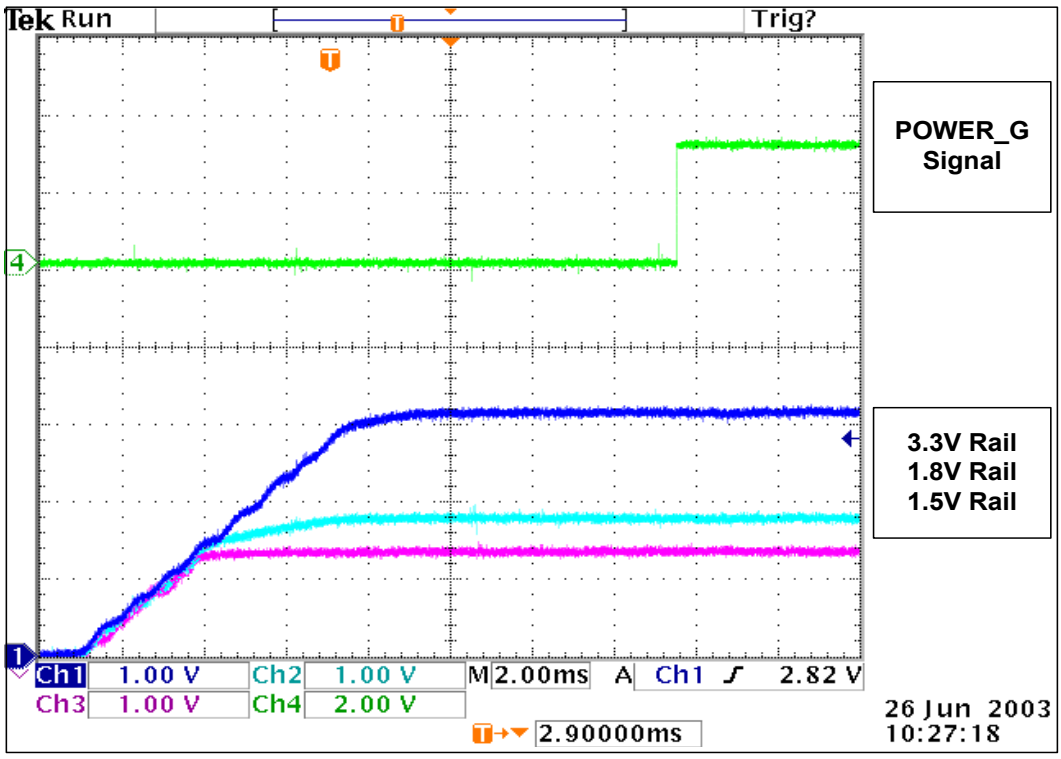
Figure 10. 3.3V Supply Turning On



Ramp Up and POWER_G Signal Example

Figure 10 shows typical system ramp up behavior and the POWER_G signal rising after all outputs have stabilized. The ramp rate is configured to 500 V/s.

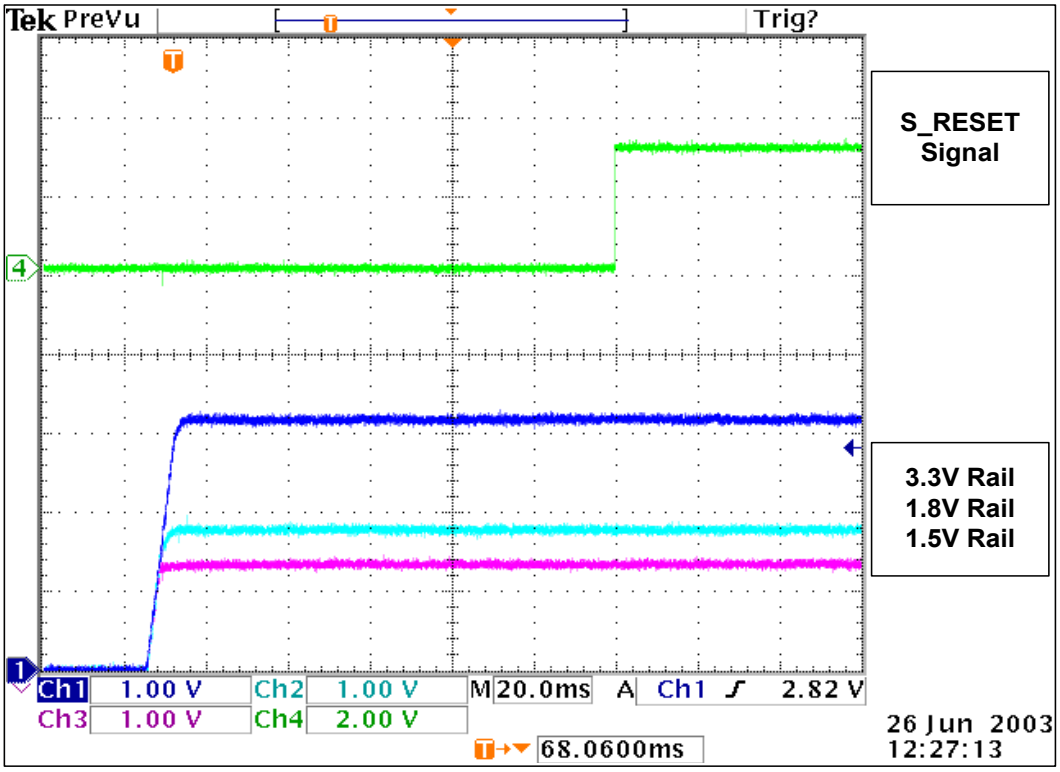
Figure 11. Typical Ramp Up and POWER_G Signal



Ramp Up and S_RESET Signal Example

Figure 10 shows typical system ramp up behavior and the S_RESET signal. The S_RESET signal is configured to rise 100 ms after the POWER_G signal. The ramp rate is configured to 500 V/s.

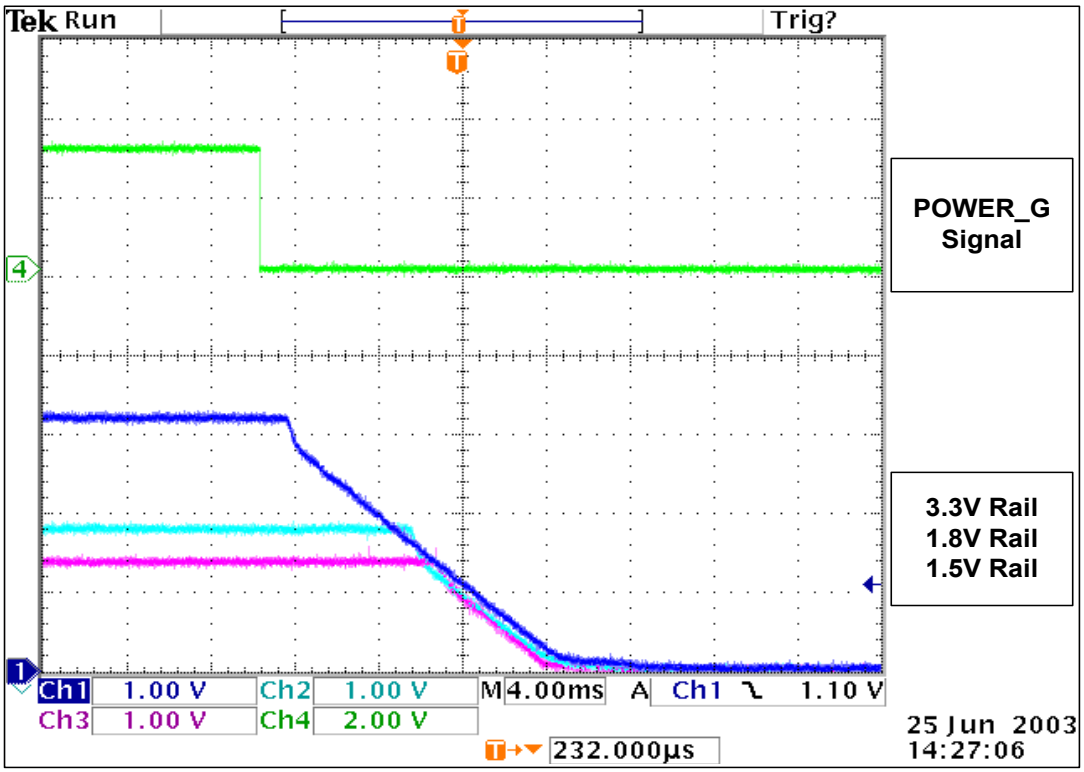
Figure 12. Typical Ramp Up and S_RESET Signal



User Shutdown and POWER_G Signal Example

Figure 10 shows typical system ramp down behavior and the POWER_G signal falling when the user presses the CAL/SD switch.

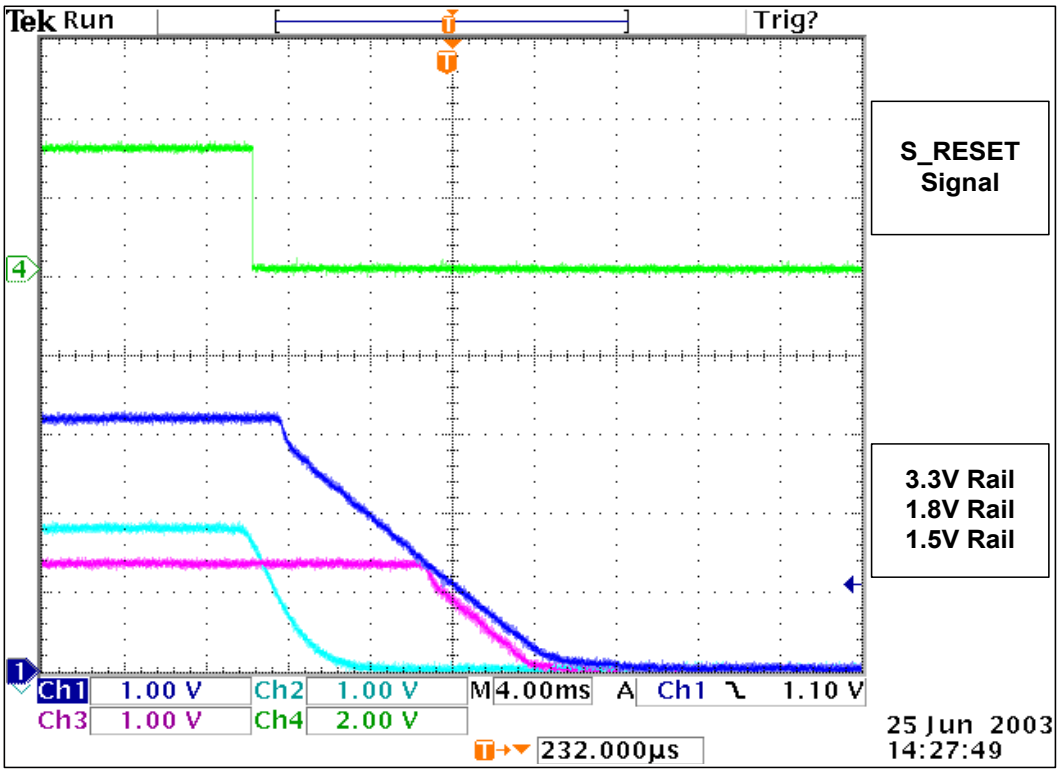
Figure 13. Typical User Shutdown and POWER_G Signal



Power Failure and S_RESET Signal Example

Figure 10 shows system ramp down behavior and the S_RESET signal falling when a failure is detected on the 1.8V channel.

Figure 14. 1.8V Supply Failure and S_RESET Signal



Configuration Using the Serial Port (Optional)

After the C8051F330 has been programmed, system parameters can be configured over a 115200 BAUD 8-N-1 UART link with a PC. Once configured, these parameters are stored in non-volatile FLASH memory. The interface is an ASCII based command line. This functionality could also be implemented using the SMBus/I2C serial port.

The parameters that can be specified at run-time are:

- Slew Rate (V/s)
- Input Valid to Ramp Start Wait Time (ms)
- Output Valid to S_RESET rising edge (ms)

The configuration menu can be accessed by holding down the CAL/SD switch during a reset, as shown in Figure 4 on page 4.

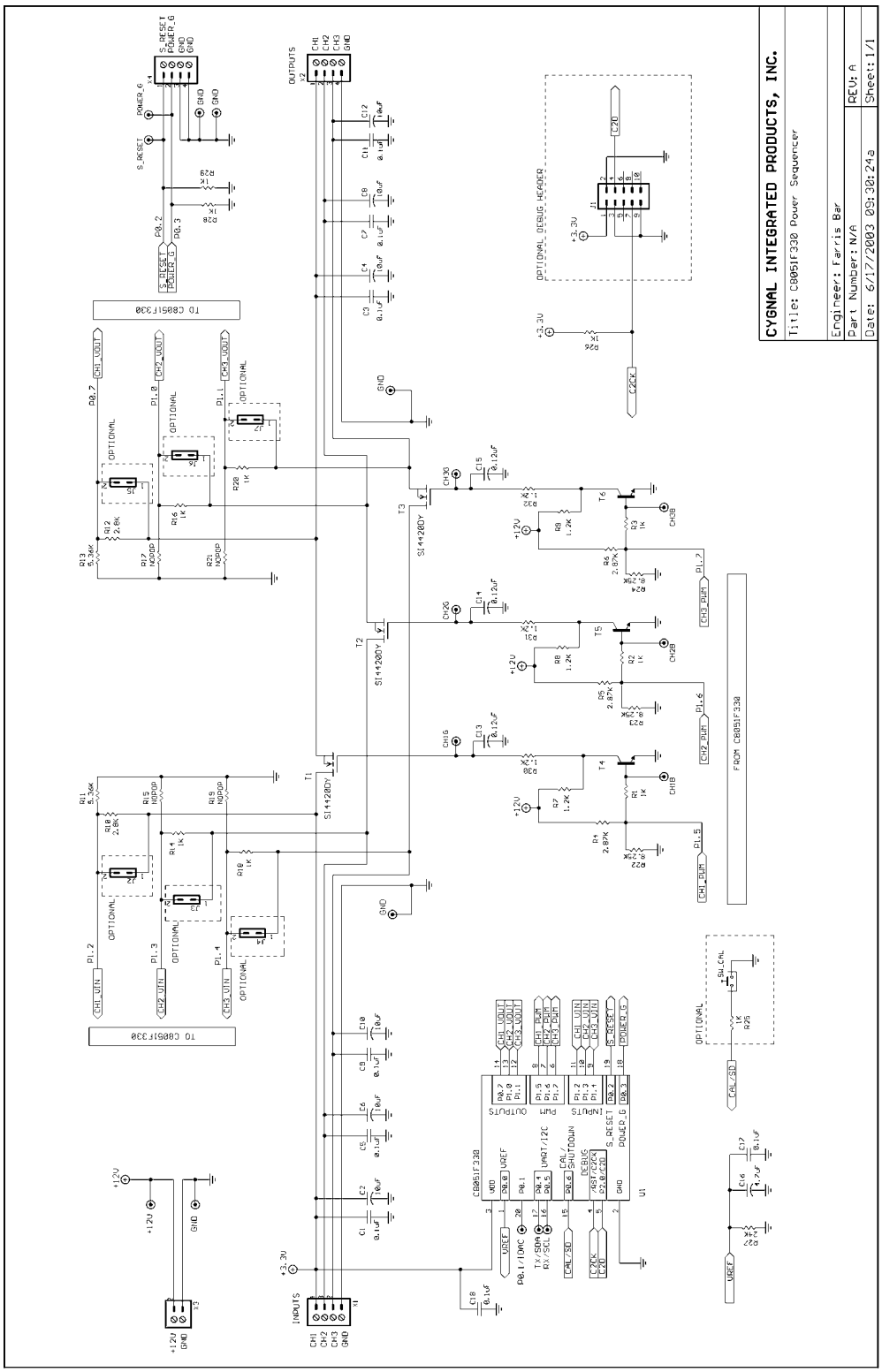
Estimated Board Real-Estate

The PCB area required for this design can be estimated by totaling the area required by the individual components as shown in Table 2 . The total area requirement for this design is less than 1 square inch.

Table 2. Estimated Component PCB Area

Device	Area (sq. inch)	Quantity	Total Area (sq. in)
C8051F330 4 x 4 mm 20-pin MLP	0.025	1	0.025
SI4420DY Power MOSFET	0.056	3	0.168
MMBT2222A NPN Amplifier (BJT)	0.018	3	0.054
10µF Tantalum Capacitor (2 per channel)	0.023	6	0.138
0.1µF Decoupling Capacitor (0805) (2 per channel)	0.008	6	0.048
4.7uF Tantalum Capacitor at VREF pin	0.012	1	0.012
0.12uF Filtering Capacitor (0805) (1 per channel)	0.008	3	0.024
0.1uF Capacitor at VREF pin (0805)	0.008	1	0.008
0.1uF Capacitor at VDD (0805)	0.008	1	0.008
Resistor (0805)	0.008	27	0.216
		Total Area	**0.701 sq. in**

Appendix A - Schematic



Appendix B - Bill of Materials

Qty	Part	Value	Package	Manufacturer
1	U1	C8051F330	MLP-20	Silicon Labs
3	T1, T2, T3	SI4420DY	SO-8	Vishay
3	T4, T5, T6	MMBT2222A	SOT-23	Fairchild
1	C16	4.7uF	CP-3216	
6	C2, C4, C6, C8, C10, C12	10uF	CP-3216	
3	C13, C14, C15	0.12uF	CNP-0805	
8	C1, C3, C5, C7, C9, C11, C17, C18	0.1uF	CNP-0805	
6	R7, R8, R9, R30, R31, R32	1.2K	R-0805	
10	R1, R2, R3, R14, R16, R18, R20, R26, R28, R29	1K	R-0805	
2	R10, R12	2.8K	R-0805	
3	R4, R5, R6	2.87K	R-0805	
2	R11, R13	5.36K	R-0805	
3	R22, R23, R24	8.25K	R-0805	
1	R27	24K	R-0805	
Note: NOPOP and Optional Items are not shown.				

Appendix C - Firmware (Source File)

```
//-----------------------------------------------------------------------------
// PS_V1.3.c
//-----------------------------------------------------------------------------
//
// AUTH: FB
// DATE: 26 JUN 03
//
// VERSION: 1.3.0
//
// Two or Three Channel Power Sequencing Solution for the
// C8051F330 and C8051F300.
//
// Target: C8051F330 and C8051F300
// Tool chain: KEIL C51
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include "PS_V1.3.h"

#if(F330)
   #include <c8051f330.h>                 // SFR declarations
   #include <stdio.h>
   #include <stdlib.h>

   //-----------------------------------------------------------------------------
   // 16-bit SFR Definitions for 'F33x
   //-----------------------------------------------------------------------------

   sfr16 DP       = 0x82;                 // data pointer
   sfr16 TMR3RL   = 0x92;                 // Timer3 reload value
   sfr16 TMR3     = 0x94;                 // Timer3 counter
   sfr16 IDA0     = 0x96;                 // IDAC0 data
   sfr16 ADC0     = 0xbd;                 // ADC0 data
   sfr16 ADC0GT   = 0xc3;                 // ADC0 Greater-Than
   sfr16 ADC0LT   = 0xc5;                 // ADC0 Less-Than
   sfr16 TMR2RL   = 0xca;                 // Timer2 reload value
   sfr16 TMR2     = 0xcc;                 // Timer2 counter
   sfr16 PCA0CP1  = 0xe9;                 // PCA0 Module 1 Capture/Compare
   sfr16 PCA0CP2  = 0xeb;                 // PCA0 Module 2 Capture/Compare
   sfr16 PCA0     = 0xf9;                 // PCA0 counter
   sfr16 PCA0CP0  = 0xfb;                 // PCA0 Module 0 Capture/Compare

#else

   #include <c8051F300.h>

   //-----------------------------------------------------------------------------
   // 16-bit SFR Definitions for 'F30x
   //-----------------------------------------------------------------------------

   sfr16 DP       = 0x82;                 // data pointer
   sfr16 TMR2RL   = 0xca;                 // Timer2 reload value
   sfr16 TMR2     = 0xcc;                 // Timer2 counter
```

```
   sfr16 PCA0CP1  = 0xe9;                 // PCA0 Module 1 Capture/Compare
   sfr16 PCA0CP2  = 0xeb;                 // PCA0 Module 2 Capture/Compare
   sfr16 PCA0     = 0xf9;                 // PCA0 counter
   sfr16 PCA0CP0  = 0xfb;                 // PCA0 Module 0 Capture/Compare

#endif // (F330)

//-----------------------------------------------------------------------------
// Function Prototypes
//-----------------------------------------------------------------------------

void main (void);

// Initialization Routines
void VDM_Init (void);
void SYSCLK_Init (void);
void PORT_Init (void);
void EX0_Init(void);
void ADC0_Init_AD0BUSY (void);
void ADC0_Init (void);
void PCA_Init (void);
void UART0_Init (void);
void Timer2_Init (int counts);

// State Implementation Routines
void ValidateInput (void);
void GlobalVarInit (void);

// Interrupt Service Routines
void EX0_ISR (void);
void Timer2_ISR (void);
void ADC0_ISR (void);

// Support Routines
void wait_ms (int ms);
void FLASH_ErasePage(unsigned addr);
void FLASH_Write(unsigned dest, char *src, unsigned num);
void Print_Menu(void);

// Calibration Routines
void Calibrate (void);
void CH1_Calibrate (int v_target);
void CH2_Calibrate (int v_target);
void CH3_Calibrate (int v_target);

//-----------------------------------------------------------------------------
// Global Constants
//-----------------------------------------------------------------------------

#if(F330)
   #define F300                  0
#else
   #define F300                  1
   #define UART_ENABLE           0        // Must be '0' for the 'F300
   #define THREE_CHANNEL         0        // Must be '0' for the 'F300
#endif // (F330)


#if(F330)
```

```
   sbit S2 = P0^7;                        // CAL/SD Switch on target board
   sbit S_RESET = P1^7;                   // System Reset Signal
   sbit POWER_G = P0^2;                   // Power Good Signal
#else
   sbit S2 = P0^0;                        // CAL/SD Switch on target board
   sbit S_RESET = P0^7;                   // System Reset Signal
#endif // (F330)

#define TRUE            1
#define FALSE           0

#define CH1             0
#define CH2             1
#define CH3             2

// System Level Constants
#define SYSCLK          24500000          // SYSCLK frequency (Hz)
#define BAUDRATE        115200            // Baud rate of UART (bps)

#define SAMPLE_RATE     15951             // ADC0 sampling rate per channel (Hz)
#define NUMCHANNELS     3                 // Number of channels
#define ADC_SAMPLERATE  48                // ADC sampling rate (kHz)

// Define ADC Resolution and VREF
#if(F330)
   #define ADC_RES         1024L          // 10-bit ADC
   #define VREF            2430L          // ADC voltage reference (mV)
#else
   #define ADC_RES         256L           // 8-bit ADC
   #define VREF            3300L          // ADC voltage reference (mV)
#endif // (F330)

enum { CAL, VAL, RAMP, MON, SHUTDOWN, OFF };   // System state definitions

// Addresses for user variables stored in FLASH
#define RAMP_RATE_ADDR     0x1A00         // Address for Ramp Rate
#define VAL_WAITTIME_ADDR  0x1A02         // Address vor Validate Wait Time
#define MON_WAITTIME_ADDR  0x1A04         // Address for Monitor Wait Time
#define CAL_DONE_ADDR      0x1A06

#define CH1_DATA_ADDR      0x1A07         // Starting address of CH1 cal data
#define CH2_DATA_ADDR      0x1B00         // Starting address of CH2 cal data
#define CH3_DATA_ADDR      0x1B80         // Starting address of CH3 cal data



// Constants used for calibration
#define VSTEP              50             // Voltage step size in mV

#define DETECT_MV          300            // # of mV allowed for detecting a
                                          // channel has reached its target
                                          // voltage

#define DETECT_ERR ((DETECT_MV*ADC_RES)/VREF)
                                          // # of codes allowed for detecting
                                          // a channel has reached its target
                                          // voltage

#define CAL_DETECT_MV   (DETECT_MV-50)    // # of mV allowed for detecting a
```

```
                                        // channel has reached its target
                                        // voltage

#define CAL_DETECT_ERR ((CAL_DETECT_MV*ADC_RES)/VREF)
                                        // # of codes allowed for detecting
                                        // a channel has reached its target
                                        // voltage during calibration


#define TRACK_ERR          52           // # of codes allowed for tracking error

#define OVERCURRENT_ERR  ((OVERCURRENT_VTH*ADC_RES)/VREF)
                                        // If the input and output differ by
                                        // greater than this number of ADC
                                        // codes (equivalent to 400mV) during
                                        // the Monitor state, the system shuts
                                        // down all outputs if overcurrent
                                        // protection is enabled

#define STRICT_VAL_ERR  ((STRICT_VAL_DELTA*ADC_RES)/VREF)
                                        // Number of ADC codes to restrict the
                                        // inputs for validation after a failure
                                        // has been detected

// Type definition allowing access to any byte of a 32-bit variable
typedef union LONGS {                   // A variable of this type can
                                        // be accessed as:
      long Long;                        // (1) 32-bit long,
      int  Int[2];                      // (2) 16-bit ints,
      char Char[4];                     // (4) 8-bit chars,

      struct S {                        //  or a Struct.
         char High;
         int Mid;
         char Low;
      }S;

} LONGS;



//-----------------------------------------------------------------------------
// Global Variables
//-----------------------------------------------------------------------------

// The current system state initialized to the Validate state
char STATE = VAL;

// The number of retries allowed before the system goes into an off state
char RETRY = NUM_RETRIES;

// The currently selected channel initialized to Channel 1
char CH = CH1;

// The current ADC0_ISR iteration while in the ramp state
// Used to determine a timeout
unsigned int ADC0_ISR_i;

// ADC0 Positive MUX input channel selection.
```

```
// When these constants are written to the AMX0P register, the corresponding
// channel is selected as the ADC input.
// The arrays are initialized as follows for the 'F330:
// { CH1, CH2, CH3, CH1 }
// and as follows for the 'F300:
// { CH1, CH2, X, CH1 }
// CH1 is repeated to simplify Timer2_ISR
#if(F330)
   char code VIN_PIN[NUMCHANNELS + 1] = { 0x06, 0x09, 0x0C, 0x06 };
   char code VOUT_PIN[NUMCHANNELS + 1] = { 0x08, 0x0B, 0x0E, 0x08};
   char code PWM_PIN[NUMCHANNELS] = { 0x01, 0x0A, 0x0D };
#else
   char code VIN_PIN[NUMCHANNELS + 1] = { 0xF2, 0xF1, 0xF0, 0xF2 };
   char code VOUT_PIN[NUMCHANNELS + 1] = { 0xF5, 0xF6, 0xF0, 0xF5};
   char code PWM_PIN[NUMCHANNELS] = { 0xF3, 0xF4, 0xF0 };
#endif // (F330)

// User Shutdown Signal, set when user presses the S2 switch while the
// system is in the Monitor state
bit USER_SHUTDOWN;

// Used by the MONITOR State to determine whether the system is currently
// sampling the input or the output side of the currently selected
// channel <CH>
bit MONITOR_INPUT;

// Used to signal if at least one of the power supply rails is near the
// maximum or minimum cutoff points.
// This bit is set to 0 at reset and set to 1 in the Monitor state if
// a power supply failure occurs.
bit STRICT_VALIDATION = 0;

// Boolean values used to determine system state
bit CH1_INPUT_VALIDATED;
bit CH2_INPUT_VALIDATED;
#if(THREE_CHANNEL)
bit CH3_INPUT_VALIDATED;
#endif // THREE_CHANNEL

// Boolean values used to determine system state
bit CH1_OUTPUT_VALIDATED;
bit CH2_OUTPUT_VALIDATED;
#if(THREE_CHANNEL)
bit CH3_OUTPUT_VALIDATED;
#endif // THREE_CHANNEL

// ADC codes used to determine if outputs are meeting the tracking specification
int CH1_PREV_VALUE;
int CH2_PREV_VALUE;
#if(THREE_CHANNEL)
int xdata CH3_PREV_VALUE;
#endif // THREE_CHANNEL

// PWM codes used to control the channel outputs;
unsigned char CH1_PWM;
unsigned char CH2_PWM;
#if(THREE_CHANNEL)
unsigned char CH3_PWM;
#endif // THREE_CHANNEL
```

```
// Variable declarations for user constants and calibration data stored in FLASH
// All variables in this section are stored in the same 512 byte FLASH sector
int code RAMP_RATE    _at_ RAMP_RATE_ADDR;        // ramp rate in V/s
int code VAL_WAITTIME _at_ VAL_WAITTIME_ADDR;     // Input Valid to Ramp Start [ms]
int code MON_WAITTIME _at_ MON_WAITTIME_ADDR;     // Output Valid to S_RESET rising
char code CAL_DONE    _at_ CAL_DONE_ADDR;         // Calibration complete flag. This
                                                  // byte is cleared after calibration
                                                  // is complete.

#define USER_DATA_SIZE 7                          // Size of user defined varibles

// CH1 Calibration Data
// Since the entire 512 byte "calibration data" FLASH page is erased by software,
// it must not contain any program code. The CH1_DATA array grows to fill all
// unused space on the FLASH page.
unsigned char code CH1_DATA[256 - USER_DATA_SIZE] _at_ CH1_DATA_ADDR;
// CH2 Calibration Data
unsigned char code CH2_DATA[128] _at_ CH2_DATA_ADDR;
// CH3 Calibration Data
unsigned char code CH3_DATA[128] _at_ CH3_DATA_ADDR;

// Indices for the calibration data arrays
unsigned char CH1_i;                  // CH1 Array Index
unsigned char CH2_i;                  // CH2 Array Index
#if(THREE_CHANNEL)
unsigned char CH3_i;                  // CH3 Array Index
#endif // THREE_CHANNEL

// These variables are set to advance through the last few
// PWM codes before turning off the PWM signal
bit CH1_RAMP_END;
bit CH2_RAMP_END;
#if(THREE_CHANNEL)
bit CH3_RAMP_END;
#endif // THREE_CHANNEL

bit ch1_tracking_disabled = 0;
bit ch2_tracking_disabled = 0;
#if(THREE_CHANNEL)
bit ch3_tracking_disabled = 0;
#endif // THREE_CHANNEL

// Counter for Power Good signal
static int pgcounter = 0;

// Target ADC code for each channel
// These variables are set to by the ValidateInput() routine after all inputs
// have settled. They are used to determine when the output voltage has reached
// the input voltage.
int CH1_TARGET_CODE;
int CH2_TARGET_CODE;
#if(THREE_CHANNEL)
int xdata CH3_TARGET_CODE;
#endif // THREE_CHANNEL

// Minimum and Maximum specified ADC readings once the inputs or outputs have
// stabilized. Used in the VALIDATE and MONITOR states to determine if the
// rail voltages are within the specified limits.
```

```
int CH1_TARGET_CODE_MIN;
int CH2_TARGET_CODE_MIN;
#if(THREE_CHANNEL)
int xdata CH3_TARGET_CODE_MIN;
#endif // THREE_CHANNEL

int CH1_TARGET_CODE_MAX;
int CH2_TARGET_CODE_MAX;
#if(THREE_CHANNEL)
int xdata CH3_TARGET_CODE_MAX;
#endif // THREE_CHANNEL

LONGS  CH1_DELTA_CODE;               // The ADC code used to set the
LONGS  CH2_DELTA_CODE;               // next expected code
#if(THREE_CHANNEL)
LONGS  xdata CH3_DELTA_CODE;
#endif // THREE_CHANNEL


LONGS  CH1_EXPECTED_CODE;            // This value is the ADC code
LONGS  CH2_EXPECTED_CODE;            // for an "ideal" curve
#if(THREE_CHANNEL)
LONGS  xdata CH3_EXPECTED_CODE;
#endif // THREE_CHANNEL


//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void) {
   int temp_int;                     // temporary int

   PCA0MD &= ~0x40;                  // disable Watchdog timer

   S_RESET = 0;                      // Clear S_RESET Signal

   #if(F330)
   POWER_G = 0;                      // Clear POWER_G Signal
   #endif // F330

   VDM_Init ();                      // initialize VDD Monitor
   SYSCLK_Init ();                   // initialize System Clock
   PORT_Init ();                     // initialize Port I/O
   PCA_Init();                       // initialize PCA
   EX0_Init();                       // initialize External Interrupt 0
                                     // and leave disabled

   // Initialize the ADC to start conversions on Timer 3 overflows
   // and to generate an End of Conversion Interrupt
   ADC0_Init();

   // Initialze Timer 2 to update at one half the the PWM frequency
   Timer2_Init(512);

   // Synchronize the PCA and Timer 3 (Timer 3 is stopped and initialized
   // to its reload value)
   CR = 0;                           // stop PCA
   PCA0 = 0x0000;
```

```
CR = 1;                            // start PCA
TMR2CN  = 0x04;                    // start Timer 2

// If the S2 switch is pressed, then enter configuration mode
if(!S2) {

   while(!S2);                     // Wait until switch released

   #if(UART_ENABLE)
      // Initialize UART0
      UART0_Init ();
   #endif // UART_ENABLE

   // Print Configuration menu and store calibration data in FLASH
   Calibrate ();

   // Issue a software reset
   RSTSRC = 0x12;
}

// Verify that the system level parameters stored in FLASH
// are initialized properly

// If RAMP_RATE is not initialized, set it to its default value
if(RAMP_RATE == 0xFFFF){
   temp_int = DEFAULT_RAMP_RATE;
   FLASH_Write(RAMP_RATE_ADDR, (char*) &temp_int, 2);
}
// If VAL_WAITTIME is not initialized, set it to its default value
if( VAL_WAITTIME == 0xFFFF){
   temp_int = DEFAULT_VAL_WAITTIME;
   FLASH_Write(VAL_WAITTIME_ADDR, (char*) &temp_int, 2);
}
// If MON_WAITTIME is not initialized, set it to its default value
if( MON_WAITTIME == 0xFFFF){
   temp_int = DEFAULT_MON_WAITTIME;
   FLASH_Write(MON_WAITTIME_ADDR, (char*) &temp_int, 2);
}

#if(THREE_CHANNEL)
// If CH1, CH2, or CH3 data is not available, enter configuration mode.
if((CH1_DATA[0]==0xFF) || (CH2_DATA[0]==0xFF) || (CH3_DATA[0]==0xFF)
  ||( CAL_DONE != 0x00)){
#else
// If CH1, or CH2 data is not available, enter configuration mode.
if((CH1_DATA[0]==0xFF) || (CH2_DATA[0]==0xFF) ||( CAL_DONE != 0x00)){
#endif // THREE_CHANNEL

   #if(UART_ENABLE)
      // Initialize UART0
      UART0_Init ();
   #endif // UART_ENABLE

   // Print Configuration menu and store calibration data in FLASH
   Calibrate ();

   // Issue a software reset
   RSTSRC = 0x12;
```

```
}

while (1){

   S_RESET = 0;                     // Assert S_RESET Signal

   #if(F330)
      POWER_G = 0;                  // De-Assert the POWER_G signal
   #endif // F330

   // Disable Interrupts
   EA = 0;

   // Call the GlobalVarInit() routine to initialize global variables
   GlobalVarInit();

   // Sets the VTARGET for each channel to its Vin and verifies
   // that Vin is within 8% of the channel values.
   ValidateInput();



   // If the output has passed strict validation, loosen the validation
   // requirements and re-validate
   if(STRICT_VALIDATION){
      STRICT_VALIDATION = 0;
      GlobalVarInit();
      ValidateInput();
   }

   // Set the system state to RAMP
   STATE = RAMP;

   // Set current channel to CH1
   CH = CH1;

   // Change ADC positive input MUX to CH1 and discard an ADC sample
   #if(F330)
      AMX0P = VOUT_PIN[CH1];
   #else
      AMX0SL = VOUT_PIN[CH1];
   #endif // F330

   // Discard the first ADC reading after the MUX change
   AD0INT = 0;                      // clear conversion complete flag
   while(!AD0INT);                  // wait for conversion to complete

   // Clear Interrupt flags to avoid immediately servicing an
   // interrupt
   AD0INT = 0;                      // clear conversion complete flag
   TMR2CN &= ~0x80;                 // Clear Timer 2 Interrupt Flag

   // Enable ADC0 ISR and Timer 2 ISR
   ET2 = 1;                         // enable Timer 2 interrupts
   #if(F330)
      EIE1   |= 0x08;               // Enable ADC0 End of Conversion
   #else                            // Interrupts
      EIE1   |= 0x04;
```

```
#endif // F330

// Enable Global Interrupts to start ramping the output voltage
EA = 1;


//-----------------------------------------------------------------
// RAMP State
//-----------------------------------------------------------------

while(STATE == RAMP);          // Polled code does not perform any
                               // tasks in the ramp state

//-----------------------------------------------------------------
// MON State
//-----------------------------------------------------------------
//
// After the RAMP state, the system can only be in the Monitor,
// Shutdown, Validate, or Off states.
//

// If the system has entered the Monitor state, assert the
// POWER_G signal and start the S_RESET timeout
if( STATE == MON){

   // assert the POWER_G signal
   #if(F330)
      POWER_G = 1;
   #endif

   // start the Monitor state timeout
   wait_ms(MON_WAITTIME);
}

// The Monitor state timeout has now expired.
// If the system is still in the Monitor state, de-assert S_RESET
if( STATE == MON){
   S_RESET = 1;
}

while(STATE == MON);           // wait in this loop until the state
                               // changes

//-----------------------------------------------------------------
// SHUTDOWN and OFF States
//-----------------------------------------------------------------
//  After a successful shutdown, the state will change to
//  OFF or VALIDATE.
//

while(STATE != VAL){

   if(STATE == OFF){
      while(1){
         RSTSRC = 0x02;        // Disable missing clock detector
         PCON |= 0x02;         // Put CPU in Stop Mode
      }
   }
```

```
      }


      // We have now entered the validate state after a power failure
      if(RETRY){

         RETRY--;

      } else {

         while(1){
            RSTSRC = 0x02;          // Disable missing clock detector
            PCON |= 0x02;           // Put CPU in Stop Mode
         }

      } // if(RETRY)

   } // while(1)

} // main

//-----------------------------------------------------------------------------
// GlobalVarInit
//-----------------------------------------------------------------------------
//
// This function initializes global variables used by the ADC0_ISR while
// the system is in the RAMP state.
//
void GlobalVarInit (void)
{
   long temp_long;



   //---------------------------------------------------------------
   // Validate State Initializations
   //---------------------------------------------------------------
   //
   //
      // Calculate <TARGET_CODE_MIN> and <TARGET_CODE_MAX> for all
      // three channels
      // <TARGET_CODE_MIN> is VTARGET_MIN converted to an ADC code
      // and <TARGET_CODE_MAX> is VTARGET_MAX converted to an ADC code
      // Equations:
      // <TARGET_CODE_MIN> = <VTARGET_MIN>/VREF * 2^10 (10-bit ADC)
      // <TARGET_CODE_MAX> = <VTARGET_MAX>/VREF * 2^8  ( 8-bit ADC)

         // Calculate the <TARGET_CODE_MIN> for CH1 and translate down
         CH1_TARGET_CODE_MIN =
         ((((CH1_VTARGET_MIN * ADC_RES)/VREF) * R11) / (R10+R11));
         if(STRICT_VALIDATION) {
            CH1_TARGET_CODE_MIN += ((STRICT_VAL_ERR * R11) / (R10+R11));
         }

         // Calculate the <TARGET_CODE_MIN> for CH2
         CH2_TARGET_CODE_MIN = ((CH2_VTARGET_MIN * ADC_RES)/VREF);
         if(STRICT_VALIDATION) {
            CH2_TARGET_CODE_MIN += STRICT_VAL_ERR;
         }
```

```
      #if(THREE_CHANNEL)
      // Calculate the <TARGET_CODE_MIN> for CH3
      CH3_TARGET_CODE_MIN = ((CH3_VTARGET_MIN * ADC_RES)/VREF);
      if(STRICT_VALIDATION) {
         CH3_TARGET_CODE_MIN += STRICT_VAL_ERR;
      }
      #endif // THREE_CHANNEL

      // Calculate the <TARGET_CODE_MAX> for CH1 and translate down
      CH1_TARGET_CODE_MAX =
      ((((CH1_VTARGET_MAX * ADC_RES)/VREF) * R11) / (R10+R11));
      if(STRICT_VALIDATION) {
         CH1_TARGET_CODE_MAX -= ((STRICT_VAL_ERR * R11) / (R10+R11));
      }

      // Calculate the <TARGET_CODE_MAX> for CH2
      CH2_TARGET_CODE_MAX = ((CH2_VTARGET_MAX * ADC_RES)/VREF);
      if(STRICT_VALIDATION) {
         CH2_TARGET_CODE_MAX -= STRICT_VAL_ERR;
      }

      #if(THREE_CHANNEL)
      // Calculate the <TARGET_CODE_MAX> for CH3
      CH3_TARGET_CODE_MAX = ((CH3_VTARGET_MAX * ADC_RES)/VREF);
      if(STRICT_VALIDATION) {
         CH3_TARGET_CODE_MAX -= STRICT_VAL_ERR;
      }
      #endif // THREE_CHANNEL

   // Set the <INPUT_VALIDATED> flags to FALSE
   CH1_INPUT_VALIDATED = FALSE;
   CH2_INPUT_VALIDATED = FALSE;
   #if(THREE_CHANNEL)
   CH3_INPUT_VALIDATED = FALSE;
   #endif // THREE_CHANNEL


//-----------------------------------------------------------------
// RAMP State Initializations
//-----------------------------------------------------------------
//
//
   // Initialize the indexes to the calibration data in FLASH
   CH1_i = 0;
   CH2_i = 0;
   #if(THREE_CHANNEL)
   CH3_i = 0;
   #endif // THREE_CHANNEL

   // Set the initial PWM Codes to zero.
   CH1_PWM = 0;
   CH2_PWM = 0;
   #if(THREE_CHANNEL)
   CH3_PWM = 0;
   #endif // THREE_CHANNEL

   // Clear the Power Good Counter
   pgcounter = 0;
```

```
// Select Channel 1 as the current channel
CH = CH1;


// Calculate <DELTA_CODE> for all three channels
// <DELTA_CODE> is the number of ADC codes (multiplied by
// 256 to maintain precision) that should increment during each
// sampling period to achieve the desired ramp rate.
// Equation:
// <DELTA_CODE> = (<RAMP_RATE/SAMPLE_RATE>/VREF * ADC_RES) * 256

   // Calculate the <DELTA_CODE> for all channels
   temp_long = RAMP_RATE;        // read the ramp rate from FLASH

   // Multiply by ADC_RES
   #if(ADC_RES == 1024L)
   temp_long <<= 10;             // multiply by ADC_RES = 2^10
   #elif(ADC_RES == 256L)
   temp_long <<= 8;              // multiply by ADC_RES = 2^8
   #elif
      #error("Unsupported ADC Resolution")
   #endif // ADC_RES

   // Shift ADC code to the two middle bytes of a long
   temp_long <<= 8;              // multipy by 256

   temp_long /= VREF;            // divide by VREF (mV)

   // Divide by the sample rate (kHz)
   temp_long *= 1000;            // multiply numerator by 1000 (Hz->kHz)
                                 // equivalent to
                                 // temp_long/(SAMPLE_RATE/1000)
   temp_long /= SAMPLE_RATE;     // divide by SAMPLE_RATE (Hz)

   CH1_DELTA_CODE.Long = temp_long;
   CH2_DELTA_CODE.Long = temp_long;
   #if(THREE_CHANNEL)
   CH3_DELTA_CODE.Long = temp_long;
   #endif // THREE_CHANNEL



// Set the <EXPECTED_CODE[ch]> one <DELTA_CODE> below 0V
CH1_EXPECTED_CODE.Long = - (CH1_DELTA_CODE.Long);
CH2_EXPECTED_CODE.Long = - (CH2_DELTA_CODE.Long);
#if(THREE_CHANNEL)
CH3_EXPECTED_CODE.Long = - (CH3_DELTA_CODE.Long);
#endif // THREE_CHANNEL

// Set the <OUTPUT_VALIDATED[ch]> flag to FALSE
CH1_OUTPUT_VALIDATED = FALSE;
CH2_OUTPUT_VALIDATED = FALSE;
#if(THREE_CHANNEL)
CH3_OUTPUT_VALIDATED = FALSE;
#endif // THREE_CHANNEL

// Set the <RAMP_END> flags to zero
CH1_RAMP_END = 0;
```

```
      CH2_RAMP_END = 0;
      #if(THREE_CHANNEL)
      CH3_RAMP_END = 0;
      #endif // THREE_CHANNEL

      // Set PREV_VALUE to zero
      CH1_PREV_VALUE = 0;
      CH2_PREV_VALUE = 0;
      #if(THREE_CHANNEL)
      CH3_PREV_VALUE = 0;
      #endif // THREE_CHANNEL

      // Clear Tracking disabled flags
      ch1_tracking_disabled = 0;
      ch2_tracking_disabled = 0;
      #if(THREE_CHANNEL)
      ch3_tracking_disabled = 0;
      #endif // THREE_CHANNEL

      // Set Ramp State iteration counter to zero
      ADC0_ISR_i = 0;

   //-----------------------------------------------------------------
   // MON State Initializations
   //-----------------------------------------------------------------
   //
   //
      // Clear the user shutdown signal
      USER_SHUTDOWN = 0;

      // Disable External Interrupt 0 interrupts
      // They will be enabled after ramping is complete
      EX0 = 0;

      // Initialize the monitoring bit for outputs.
      // When this bit is set to 1, the ADC samples the currently
      // selected channel's input.
      MONITOR_INPUT = 0;

}

//-----------------------------------------------------------------------------
// ValidateInput
//-----------------------------------------------------------------------------
//
//  This routine exits when all inputs are at their target voltage.
//
void ValidateInput(void)
{
   int target_code_min;               // target ADC code
   LONGS acc;
   char ch = CH1;                     // currently selected channel
   int i;
   int target_code;
   int target_code_max;
   bit supply_ok = 0;

   // Verify that the 12V supply is working properly
```

```
// Disable PCA I/O and make (CH2_PWM) an analog input
#if(F330)
   XBR1   &= ~0x40;                 // disable Crossbar
   XBR1   &= ~0x03;                 // update Crossbar to disable PCA I/O
   P1MDIN &= ~0x04;                 // make CH2 PWM_PIN an analog input
   XBR1   |=  0x40;                 // re-enable the Crossbar
#else
   XBR2   &= ~0x40;                 // disable Crossbar
   XBR1   &= ~0xC0;                 // update Crossbar to disable PCA I/O
   P0MDIN &= ~0x10;                 // make CH2 PWM_PIN (P0.4) an analog input
   XBR2   |=  0x40;                 // re-enable the Crossbar
#endif // F330

// Configure the ADC Positive MUX to CH2 PWM_PIN
#if(F330)
   AMX0P = PWM_PIN[CH2];
#else
   AMX0SL = PWM_PIN[CH2];
#endif // F330

// Skip an ADC reading
AD0INT = 0;                         // clear conversion complete flag
while(!AD0INT);                     // wait for conversion to complete

while(!supply_ok){

   // Take an ADC reading;
   AD0INT = 0;                      // clear conversion complete flag
   while(!AD0INT);                  // wait for conversion to complete

   // The voltage at the CH3 PWM pin should be around 3.3V if the +12V
   // supply is ok.
   // Check if Voltage at CH3 PWM pin is greater than 1.5 volts (+12V
   // supply is ok)
   supply_ok = (ADC0 > ((1500L*ADC_RES)/VREF));

} // while(!supply_ok)


// Re-enable PCA I/O
#if(F330)
   XBR1   &= ~0x40;                 // disable Crossbar
   #if(THREE_CHANNEL)
   XBR1   |= 0x03;                  // update Crossbar to enable CEX 1, 2 and 3
   #else
   XBR1   |= 0x02;                  // update Crossbar to enable CEX 1 and 2
   #endif // THREE_CHANNEL
   P1MDIN |= 0x04;                  // configure CEX1 to digital mode
   P1MDOUT|= 0x04;                  // configure CEX1 to push-pull mode
   XBR1   |= 0x40;                  // re-enable the Crossbar
#else
   XBR2   &= ~0x40;                 // disable Crossbar
   XBR1   |=  0x80;                 // update Crossbar to enable PCA I/O
   P0MDIN |=  0x10;                 // configure CEX1 to digital mode
   P0MDOUT|=  0x10;                 // configure CEX1 to push-pull mode
   XBR2   |=  0x40;                 // re-enable the Crossbar
#endif // F330
```

```
// Stay in this loop until all input channels have reached their minimum
// specified target voltage
do{

   // Configure the ADC Positive Input MUX to the input of the
   // currently selected channel.
   #if(F330)
      AMX0P = VIN_PIN[ch];
   #else
      AMX0SL = VIN_PIN[ch];
   #endif // F330

   // Select a minimum target code based on the current channel
   switch(ch){
      case CH1: target_code_min = CH1_TARGET_CODE_MIN;
                target_code_max = CH1_TARGET_CODE_MAX;
                break;
      case CH2: target_code_min = CH2_TARGET_CODE_MIN;
                target_code_max = CH2_TARGET_CODE_MAX;
                break;
      #if(THREE_CHANNEL)
      case CH3: target_code_min = CH3_TARGET_CODE_MIN;
                target_code_max = CH3_TARGET_CODE_MAX;
                break;
      #endif // THREE_CHANNEL

      default:  break;

   } // switch(ch)

   // Skip an ADC reading
   AD0INT = 0;                // clear conversion complete flag
   while(!AD0INT);            // wait for conversion to complete

   // Take an ADC reading;
   AD0INT = 0;                // clear conversion complete flag
   while(!AD0INT);            // wait for conversion to complete

   // Set the <INPUT_VALIDATED> flag for this channel if the ADC
   // reading is within the overvoltage and undervoltage spec.
   switch(ch){
      case CH1: CH1_INPUT_VALIDATED =

                #if(OVERVOLTAGE_PROTECTION)
                  ((ADC0 >= target_code_min) && (ADC0 <= target_code_max));
                #else
                  (ADC0 >= target_code_min);
                #endif // OVERVOLTAGE_PROTECTION

                break;

      case CH2: CH2_INPUT_VALIDATED =

                #if(OVERVOLTAGE_PROTECTION)
                  ((ADC0 >= target_code_min) && (ADC0 <= target_code_max));
                #else
                  (ADC0 >= target_code_min);
                #endif // OVERVOLTAGE_PROTECTION
```

```
                  break;
         #if(THREE_CHANNEL)
         case CH3: CH3_INPUT_VALIDATED =

                  #if(OVERVOLTAGE_PROTECTION)
                    ((ADC0 >= target_code_min) && (ADC0 <= target_code_max));
                  #else
                    (ADC0 >= target_code_min);
                  #endif // OVERVOLTAGE_PROTECTION

                  break;
         #endif

         default:  break;

      } // switch(ch)

      // Advance to the next channel. If past the last channel, set
      // the current channel to CH1.
      ch++;

      #if(THREE_CHANNEL)
      if(ch >= 3){
         ch = CH1;
      }
      #else
      if(ch >= 2){
         ch = CH1;
      }
      #endif // THREE_CHANNEL

   #if(THREE_CHANNEL)
   } while( !(CH1_INPUT_VALIDATED && CH2_INPUT_VALIDATED && CH3_INPUT_VALIDATED) );
   #else
   } while( !(CH1_INPUT_VALIDATED && CH2_INPUT_VALIDATED) );
   #endif

   // Now all channel inputs are within their specified voltage range
   // Wait for all inputs to settle to their steady state value.
   // This timeout is user-defined and can be set from the configuration menu.
   // The default timeout is 100 ms.

   wait_ms(VAL_WAITTIME);


   // Now all channel inputs have settled to their steady-state values.
   // Record the ADC code measured at each input in the corresponding
   // <CHx_TARGET_CODE>

   // Repeat the following for all channels
   #if(THREE_CHANNEL)
   for( ch = 0; ch < 3; ch++) {
   #else
   for( ch = 0; ch < 2; ch++) {
   #endif

      // Configure the ADC Positive Input MUX to the input of the
      // currently selected channel.
      #if(F330)
```

```
         AMX0P = VIN_PIN[ch];
      #else
         AMX0SL = VIN_PIN[ch];
      #endif // F330

      // Discard the first ADC reading after the MUX change
      AD0INT = 0;                         // clear conversion complete flag
      while(!AD0INT);                     // wait for conversion to complete

      // obtain 1024 samples
      acc.Long = 0;
      for(i = 0; i < 1024; i++){

         // obtain one sample
         AD0INT = 0;
         while(!AD0INT);

         // add to accumulator
         acc.Long += ADC0;

      } // for(i = 0; i < 1024; i++)

      // take the average (divide by 1024 = 2^10)
      target_code = acc.S.Mid >> 2;       // Accessing the middle two
                                          // bytes of the long variable
                                          // is equivilant to an 8-bit
                                          // shift or divide by 256

      // Set the <CHx_TARGET_CODE> for the currenly selected channel
      switch(ch){
         case CH1: CH1_TARGET_CODE = target_code;
                   break;
         case CH2: CH2_TARGET_CODE = target_code;
                   break;
         #if(THREE_CHANNEL)
         case CH3: CH3_TARGET_CODE = target_code;
                   break;
         #endif // THREE_CHANNEL

         default:  break;

      } // switch(ch)

   } // for( ch = 0; ch < 3; ch++)

} // ValidateInput


//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
// EX0_ISR
//-----------------------------------------------------------------------------
void EX0_ISR (void) interrupt 0
{
   USER_SHUTDOWN = 1;
   EX0 = 0;                            // Disable External Interrupt 0
                                       // interrupts
```

```
}

//-----------------------------------------------------------------------------
// Timer2_ISR
//-----------------------------------------------------------------------------
void Timer2_ISR (void) interrupt 5 using 2
{

   if(STATE == RAMP){

      // Change the ADC MUX to VOUT for the next channel
      #if(F330)
         AMX0P = VOUT_PIN[(CH+1)];
      #else
         AMX0SL = VOUT_PIN[(CH+1)];
      #endif // F330

   } else

   if(STATE == MON){
      // Change the ADC MUX to VOUT or VIN for the next channel
      // The MONITOR_INPUT bit is managed by the ADC0_ISR
      if(MONITOR_INPUT){

         #if(F330)
            AMX0P = VIN_PIN[(CH+1)];
         #else
            AMX0SL = VIN_PIN[(CH+1)];
         #endif // F330

      } else {

         #if(F330)
            AMX0P = VOUT_PIN[(CH+1)];
         #else
            AMX0SL = VOUT_PIN[(CH+1)];
         #endif // F330

      } // if(MONITOR_INPUT)


   }

   TMR2CN &= ~0x80;                 // Clear Timer 2 Interrupt Flag


}

//-----------------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------------
#if(F330)
   void ADC0_ISR (void) interrupt 10 using 1
#else
   void ADC0_ISR (void) interrupt 8  using 1
#endif // F330
{
   static int adc_code;           // The raw ADC reading for CH1, CH2, CH3
```

```
static LONGS ch1_adc_code;     // The scaled ADC reading for CH1, valid until
                               // the end of the third channel.
static int ch2_adc_code;       // Used to temporarily hold the CH2 ADC Code
                               // until the end of the third channel, when
                               // all three CHx_PREV_VALUE variables are
                               // updated.
static bit pg_bit = 0;

// Variables used during ramp end to increment the PWM code.
static char ch1_inc = 0;
static char ch2_inc = 0;
#if(THREE_CHANNEL)
static xdata char ch3_inc = 0;
#endif // THREE_CHANNEL

bit shutdown = 0;

AD0INT = 0;                     // Clear ADC Conversion Complete Interrupt
                                // Flag
// read the current ADC code
adc_code = ADC0;

switch(STATE){

//-----------------------------------------------------------------------------
// RAMP State
//-----------------------------------------------------------------------------
//
// Increase and track the output voltages on all enabled channels at
// <RAMP_RATE> mA/sec until all channels have reached their target
// voltage.
//
// If Vout has not yet reached the target voltage and is within tracking
// requirements for the channel. There are the possible states that Vout
// can be in with respect to the ideal curve.
//
// Case 1: Vout is much less than ideal line.
// Action: Increment Vout and hold ideal line.
//
// Case 2: Vout is slightly below ideal line.
// Action: Increment Vout and increment ideal line.
//
// Case 3: Vout is slightly above the ideal line.
// Action: Hold Vout and increment ideal line.
//
case RAMP:

   ADC0_ISR_i++;                          // increment iteration counter

   // If the ISR stays in the RAMP state for more than 100ms, shutdown and
   // go back to the VAL state.
   // RAMP_TIMEOUT [ms] * sampling rate[kHz]
   #if(RAMP_TIMEOUT_ENABLE)
   if(ADC0_ISR_i > (RAMP_TIMEOUT * ADC_SAMPLERATE))
   {
      ADC0_ISR_i = 0;
      STATE = SHUTDOWN;
      STRICT_VALIDATION = 1;              // Set the Strict Validation Flag
```

```
}
#endif

// CHANNEL 1
if(CH == CH1){

   // If Vout is not already at the target voltage for this channel
   if(!CH1_OUTPUT_VALIDATED){

      if(!CH1_RAMP_END){

         // TRACKING REQUIREMENT:
         // If Vout is (TRACK_ERR ADC codes) greater than the
         // other two channels, hold Vout and the Ideal line

         // Multiply by (R10+R11)/R11 * 65536
         ch1_adc_code.Long = (long) (adc_code * (((R10+R11)*65536)/R11));

         #if(THREE_CHANNEL)
         if( (ch1_adc_code.Int[0]) > (CH2_PREV_VALUE + TRACK_ERR) ||
             (ch1_adc_code.Int[0]) > (CH3_PREV_VALUE + TRACK_ERR) ){
         #else
         if( (ch1_adc_code.Int[0]) > (CH2_PREV_VALUE + TRACK_ERR) ){
         #endif // THREE_CHANNEL

             // Hold Vout and the adjust ideal line to current ADC value + Delta Code
             CH1_EXPECTED_CODE.S.Mid =(ch1_adc_code.Int[0] + CH1_DELTA_CODE.Int[0]);
             CH1_EXPECTED_CODE.Char[0] = 0;
             CH1_EXPECTED_CODE.Char[3] = CH1_DELTA_CODE.Char[3];

         } else

         // CASE 1: Vout is much less than the ideal line
         if(ch1_adc_code.Int[0] <=
           (CH1_EXPECTED_CODE.S.Mid - CH1_DELTA_CODE.S.Mid)){

            // Increment Vout and hold the ideal line
            CH1_PWM = CH1_DATA[CH1_i++];

            // If end of table has been reached,
            // Go to Ramp End
            if(CH1_PWM == 0xFF){
               CH1_PWM = PCA0CPH0;
               CH1_RAMP_END = 1;
            }

         } else

         // CASE 2: Vout is slightly less than the ideal line
         if(ch1_adc_code.Int[0] <= CH1_EXPECTED_CODE.S.Mid){

            // Increment Vout to the next table entry
            CH1_PWM = CH1_DATA[CH1_i++];

            // If end of table has been reached,
            // Go to Ramp End
            if(CH1_PWM == 0xFF){
               CH1_PWM = PCA0CPH0;
               CH1_RAMP_END = 1;
```

```
      }

      // Increment Ideal Line
      CH1_EXPECTED_CODE.Long += CH1_DELTA_CODE.Long;

   } else

   // CASE 3: Vout is higher than the ideal line
   {
      // Hold Vout and increment the ideal line
      CH1_EXPECTED_CODE.Long += CH1_DELTA_CODE.Long;

   }

} // if(!CH1_RAMP_END)


// Check if Vout has reached the target voltage for the channel
if(adc_code >=
  (CH1_TARGET_CODE - DETECT_ERR)  || CH1_RAMP_END) {

   // Set the Ramp End Flag to force execution of the following
   // code until ramping has ended
   if(!CH1_RAMP_END){

       CH1_RAMP_END = 1;

   }

   // Disable tracking if we are within the window
   // set to maximum positive code minus tracking error
   if((adc_code >= (CH1_TARGET_CODE - DETECT_ERR))
      && !ch1_tracking_disabled
     ){

      CH1_PREV_VALUE = adc_code;
      ch1_tracking_disabled = 1;

   } else {

      // For CH1, Tracking is not required if we have reached
      // ramp end since this is only remaining channel

   } // if( adc_code >= ... )

   // If the PWM code is less than 0xFF, then increment it
   // by 1/5 codes until it is >= 0xFF. Once it has reached 0xFF,
   // validate the output for the channel and output a
   // 0% duty cycle.
   if(CH1_PWM < 0xFF){

      if(ch1_inc == 0){
         CH1_PWM = PCA0CPH0 + 1;
         ch1_inc = 5;
         pg_bit = !pg_bit;
      } else {
         ch1_inc--;
      }
```

```
            if(ch1_tracking_disabled){

               // Enter Loop every 620us
               if((ch1_inc == 1) && pg_bit){

                  // Compare ADC code to previous value + 18mV (5 ADC Codes)
                  if( adc_code <= CH1_PREV_VALUE + 5){
                     pgcounter++;
                  } // adc_code

                  // Update previous value for the MONITOR state
                  CH1_PREV_VALUE = adc_code;

                  // If output stablilizes for 8 iterations (min 4.96us) after
                  // CH1 has exceeded the ramp-end threshold, validate all
                  // channels.
                  if(pgcounter == 8){

                     // clear the ECOM bit for this channel to produce a 0%
                     // duty cycle
                     PCA0CPM0 &= ~0x40;
                     PCA0CPM1 &= ~0x40;
                     PCA0CPM2 &= ~0x40;

                     CH1_OUTPUT_VALIDATED = TRUE;
                     CH2_OUTPUT_VALIDATED = TRUE;
                     CH3_OUTPUT_VALIDATED = TRUE;

                     // Update previous value for the MONITOR state
                     CH1_PREV_VALUE = 0x7FFF - TRACK_ERR;

                  } // if(pgcounter == 8)

               } // if((ch1_inc == 1) && pg_bit)

            } // if(ch1_tracking_disabled)


         } else {

             // validate the output for this channnel
             CH1_OUTPUT_VALIDATED = TRUE;

             // Set CH1_PREV_VALUE for the Monitor State
             CH1_PREV_VALUE = 0x7FFF - TRACK_ERR;

             // clear the ECOM bit for this channel to produce a 0%
             // duty cycle
             PCA0CPM0 &= ~0x40;

         } // CH1_PWM < 0xFF

      } else {

         // Tracking variable <ch1_adc_code> already updated above

      } // if(adc_code ... || ch1_ramp_end)

   } // (!CH1_OUTPUT_VALIDATED)
```

```
} else


// CHANNEL 2
if(CH == CH2){

   // If Vout is not already at the target voltage for this channel
   if(!CH2_OUTPUT_VALIDATED){

      if(!CH2_RAMP_END){

         // TRACKING REQUIREMENT:
         // If Vout is (TRACK_ERR ADC codes) greater than the
         // other two channels, hold Vout and the Ideal line
         #if(THREE_CHANNEL)
         if( (adc_code) > (CH1_PREV_VALUE + TRACK_ERR) ||
             (adc_code) > (CH3_PREV_VALUE + TRACK_ERR) ){
         #else
         if( (adc_code) > (CH1_PREV_VALUE + TRACK_ERR) ){
         #endif // THREE_CHANNEL

            // Hold Vout and the adjust ideal line to current ADC value + Delta Code
            CH2_EXPECTED_CODE.S.Mid = (adc_code + CH2_DELTA_CODE.Int[0]);
            CH2_EXPECTED_CODE.Char[0] = 0;
            CH2_EXPECTED_CODE.Char[3] = CH2_DELTA_CODE.Char[3];

         } else

         // CASE 1: Vout is much less than the ideal line
         if(adc_code <=
           (CH2_EXPECTED_CODE.S.Mid - CH2_DELTA_CODE.S.Mid)){

            // Increment Vout and hold Ideal line
            CH2_PWM = CH2_DATA[CH2_i++];

            // If end of table has been reached,
            // Go to Ramp End
            if(CH2_PWM == 0xFF){
               CH2_PWM = PCA0CPH1;
               CH2_RAMP_END = 1;
            }

         } else

         // CASE 2: Vout is slightly less than the ideal line
         if(adc_code <= CH2_EXPECTED_CODE.S.Mid){

            // Increment Vout to the next table entry
            CH2_PWM = CH2_DATA[CH2_i++];

            // If end of table has been reached,
            // Go to Ramp End
            if(CH2_PWM == 0xFF){
               CH2_PWM = PCA0CPH1;
               CH2_RAMP_END = 1;
            }
```

```
            // Increment the ideal line
            CH2_EXPECTED_CODE.Long += CH2_DELTA_CODE.Long;

         } else

         // CASE 3: Vout is higher than the ideal line
         {
            // Hold Vout and increment the ideal line
            CH2_EXPECTED_CODE.Long += CH2_DELTA_CODE.Long;

         }

      } // if(!CH2_RAMP_END)

      // Check if Vout has reached the target voltage for the channel
      if(adc_code >=
        (CH2_TARGET_CODE - DETECT_ERR)  || CH2_RAMP_END) {

         // Set the Ramp End Flag to force execution of the following
         // code until ramping has ended
         if(!CH2_RAMP_END){

             CH2_RAMP_END = 1;

         }

         // Disable tracking
         // set to maximum positive code minus tracking error
         if(adc_code >= (CH2_TARGET_CODE - DETECT_ERR)){

            CH2_PREV_VALUE = 0x7FFF - TRACK_ERR;
            ch2_tracking_disabled = 1;

         } else {

            // Update Previous Value for tracking
            ch2_adc_code = (adc_code);
         }

         // If the PWM code is less than 0xFF, then increment it
         // by 1/5 codes until it is >= 0xFF. Once it has reached 0xFF,
         // validate the output for the channel and output a
         // 0% duty cycle.
         if(CH2_PWM < 0xFF){

            if(ch2_inc == 0){
               CH2_PWM = PCA0CPH1 + 1;
               ch2_inc = 5;
            } else {
               ch2_inc--;
            }

         } else {

             // validate the output for this channnel
             CH2_OUTPUT_VALIDATED = TRUE;

             // clear the ECOM bit for this channel to produce a 0%
             // duty cycle
```

```
                PCA0CPM1 &= ~0x40;

            }

         } else {

            // Update Previous Value for tracking
            ch2_adc_code = (adc_code);
         }

      } // if(!CH2_OUTPUT_VALIDATED)

   } else

   // CHANNEL 3
   { // CH == CH3

      #if(THREE_CHANNEL)
      // If Vout is not already at the target voltage for this channel
      if(!CH3_OUTPUT_VALIDATED){

         if(!CH3_RAMP_END){

            // TRACKING REQUIREMENT:
            // If Vout is (TRACK_ERR ADC codes) greater than the
            // other two channels, hold Vout and the Ideal line
            if( (adc_code) > (CH1_PREV_VALUE + TRACK_ERR) ||
                (adc_code) > (CH2_PREV_VALUE + TRACK_ERR) ){

                // Hold Vout and the adjust ideal line to current ADC value + Delta Code
                CH3_EXPECTED_CODE.S.Mid = (adc_code + CH3_DELTA_CODE.Int[0]);
                CH3_EXPECTED_CODE.Char[0] = 0;
                CH3_EXPECTED_CODE.Char[3] = CH3_DELTA_CODE.Char[3];

            } else

            // CASE 1: Vout is much less than the ideal line
            if(adc_code <=
              (CH3_EXPECTED_CODE.S.Mid - CH3_DELTA_CODE.S.Mid)){

               // Increment Vout and hold the ideal line
               CH3_PWM = CH3_DATA[CH3_i++];

               // If end of table has been reached,
               // Go to Ramp End
               if(CH3_PWM == 0xFF){
                  CH3_PWM = PCA0CPH2;
                  CH3_RAMP_END = 1;
               }

            } else

            // CASE 2: Vout is slightly less than the ideal line
            if(adc_code <= CH3_EXPECTED_CODE.S.Mid){


               // Increment Vout to the next table entry
               CH3_PWM = CH3_DATA[CH3_i++];
```

```
            // If end of table has been reached,
            // Go to Ramp End
            if(CH3_PWM == 0xFF){
               CH3_PWM = PCA0CPH2;
               CH3_RAMP_END = 1;
            }

            // Increment the ideal line
            CH3_EXPECTED_CODE.Long += CH3_DELTA_CODE.Long;

         } else


         // CASE 3: Vout is higher than the ideal line
         {
            // Hold Vout and increment the ideal line
            CH3_EXPECTED_CODE.Long += CH3_DELTA_CODE.Long;

         }

      }// if(!CH3_RAMP_END)

      // Check if Vout has reached the target voltage for the channel
      if(adc_code >=
        (CH3_TARGET_CODE - DETECT_ERR)  || CH3_RAMP_END ) {

         // Set the Ramp End Flag to force execution of the following
         // code until ramping has ended
         if(!CH3_RAMP_END){

             CH3_RAMP_END = 1;

         }

         // Disable tracking or update tracking variable
         // set to maximum positive code minus tracking error
         if(adc_code >= (CH3_TARGET_CODE - DETECT_ERR)){
            CH3_PREV_VALUE = 0x7FFF - TRACK_ERR;
            ch3_tracking_disabled = 1;

         } else {

            // Update Previous Value for tracking
            if(!ch3_tracking_disabled) {
               CH3_PREV_VALUE = (adc_code);
            }
         }

         // If the PWM code is less than 0xFF, then increment it
         // by 1/5 codes until it is >= 0xFF. Once it has reached 0xFF,
         // validate the output for the channel and output a
         // 0% duty cycle.
         if(CH3_PWM < 0xFF){

            if(ch3_inc == 0){
               CH3_PWM = PCA0CPH2 + 1;
               ch3_inc = 5;
            } else {
               ch3_inc--;
```

```
            }

         } else {

            // validate the output for this channnel
            CH3_OUTPUT_VALIDATED = TRUE;

            // clear the ECOM bit for this channel to produce a
            // 0% duty cycle
            PCA0CPM2 &= ~0x40;

         }

      } else {

         // Update Previous Value for tracking
         CH3_PREV_VALUE = (adc_code);
      }

   } // end if(!CH3_OUTPUT_VALIDATED)

   #endif // THREE_CHANNEL

   // Make sure array index is less than 128 (clear the MSB)
   CH1_i &= ~0x80;
   CH2_i &= ~0x80;
   #if(THREE_CHANNEL)
   CH3_i &= ~0x80;
   #endif // THREE_CHANNEL

   // UPDATE THE PWM OUTPUT FOR CH1, CH2, CH3
   if(PCA0CPM0 & 0x40) { PCA0CPH0 = CH1_PWM; }
   if(PCA0CPM1 & 0x40) { PCA0CPH1 = CH2_PWM; }
   #if(THREE_CHANNEL)
   if(PCA0CPM2 & 0x40) { PCA0CPH2 = CH3_PWM; }
   #endif // THREE_CHANNEL

   // UPDATE TRACKING VARIABLES
   if(!ch1_tracking_disabled) { CH1_PREV_VALUE = ch1_adc_code.Int[0];}
   if(!ch2_tracking_disabled) { CH2_PREV_VALUE = ch2_adc_code;}
   // CH3 already updated above

}


// If all channels have been validated, switch to the monitor state
#if(THREE_CHANNEL)
if(CH1_OUTPUT_VALIDATED && CH2_OUTPUT_VALIDATED && CH3_OUTPUT_VALIDATED) {
#else
if(CH1_OUTPUT_VALIDATED && CH2_OUTPUT_VALIDATED) {
#endif // THREE_CHANNEL

   STATE = MON;                        // Change system state to Monitor
   EX0 = 1;                            // Enable External Interrupt 0
                                       // interrupts to set the
                                       // USER_SHUTDOWN bit when the
                                       // CAL/SD switch is pressed while
                                       // in the Monitor State
```

```
   }

break;

//-----------------------------------------------------------------------------
// MON State
//-----------------------------------------------------------------------------
//
// Monitor the output and input voltages on all enabled channels to ensure
// proper operation
//
// Note: Upon Entry into this state, all <CHx_PREV_VALUE> variables should
// be set to a very large positive number (ex. 0x7FFF).
// This is required for overcurrent protection.
//
case MON:

   shutdown = 0;

   if(CH == CH1){

      // Verify that the voltage on CH1 is within spec.
      #if(OVERVOLTAGE_PROTECTION)
         if(adc_code < CH1_TARGET_CODE_MIN ||
            adc_code > CH1_TARGET_CODE_MAX     ){
            shutdown = 1;
         }
      #else
         if(adc_code < CH1_TARGET_CODE_MIN){
            shutdown = 1;
         }
      #endif // OVERVOLTAGE_PROTECTION

      #if(OVERCURRENT_PROTECTION)
         if(MONITOR_INPUT){

            if( (adc_code - CH1_PREV_VALUE) > ((OVERCURRENT_ERR * R10) / (R10+R11)) ){
               shutdown = 1;
            }

         } else {

            CH1_PREV_VALUE = adc_code;

         }
      #endif // OVERCURRENT_PROTECTION

   } else

   if (CH == CH2){

      // Verify that the voltage on CH2 is within spec.
      #if(OVERVOLTAGE_PROTECTION)
         if(adc_code < CH2_TARGET_CODE_MIN ||
            adc_code > CH2_TARGET_CODE_MAX    ){
            shutdown = 1;
         }
      #else
         if(adc_code < CH2_TARGET_CODE_MIN){
```

```
            shutdown = 1;
         }
      #endif // OVERVOLTAGE_PROTECTION

      #if(OVERCURRENT_PROTECTION)
         if(MONITOR_INPUT){

            if( (adc_code - CH2_PREV_VALUE) > (OVERCURRENT_ERR)){
               shutdown = 1;
            }

         } else {

            CH2_PREV_VALUE = adc_code;

         }
      #endif // OVERCURRENT_PROTECTION

   } else

   // CH == CH3
   {
      #if(THREE_CHANNEL)
      // Verify that the voltage on CH3 is within spec.
      #if(OVERVOLTAGE_PROTECTION)
         if(adc_code < CH3_TARGET_CODE_MIN ||
            adc_code > CH3_TARGET_CODE_MAX    ){
            shutdown = 1;
         }
      #else
         if(adc_code < CH3_TARGET_CODE_MIN){
            shutdown = 1;
         }
      #endif // OVERVOLTAGE_PROTECTION

      #if(OVERCURRENT_PROTECTION)
         if(MONITOR_INPUT){

            if( (adc_code - CH3_PREV_VALUE) > (OVERCURRENT_ERR)){
               shutdown = 1;
            }

         } else {

            CH3_PREV_VALUE = adc_code;

         }
      #endif // OVERCURRENT_PROTECTION
      #endif // THREE_CHANNEL
   }


   // If the system or the user requests a shutdown, assert the
   // S_RESET signal, de-assert the POWER_G signal, and switch
   // to the SHUTDOWN state
   if( shutdown || USER_SHUTDOWN){

      S_RESET = 0;                    // Assert the S_RESET signal
```

```
      #if(F330)
         POWER_G = 0;                  // De-Assert the POWER_G signal
      #endif // F330

      STRICT_VALIDATION = 1;           // Set the Strict Validation Flag
                                       // to avoid oscillation

      STATE = SHUTDOWN;                // Switch to validate state

      ADC0_ISR_i = 0;                  // Clear the ADC0_ISR iteration counter

   }

break;

//-----------------------------------------------------------------------------
// SHUTDOWN State
//-----------------------------------------------------------------------------
//
// Shut down all outputs
//
case SHUTDOWN:

   if(ADC0_ISR_i >= 10){

      ADC0_ISR_i = 0;

      // If all indexes are at table entry zero, change the state to VAL
      // or OFF
      #if(THREE_CHANNEL)
      if( (CH1_i == 0) && (CH2_i == 0) && (CH3_i == 0)) {
      #else
      if( (CH1_i == 0) && (CH2_i == 0)) {
      #endif // THREE_CHANNEL

         // Force all outputs to 0V by setting duty cycle to 100%
         CH1_PWM = 0;
         CH2_PWM = 0;
         #if(THREE_CHANNEL)
         CH3_PWM = 0;
         #endif // THREE_CHANNEL

         // If a user shutdown has been detected (CAL/SD switch pressed),
         // then put the system in the OFF state. The OFF state puts the CPU
         // in Stop Mode. Othewise re-validate the inputs and start
         // ramping again. Assume a power failure has occured.
         if(USER_SHUTDOWN){
            STATE = OFF;
         } else {
            STATE = VAL;
         }

      }

      // Start decrementing CH1 output. When the CH1 index reaches CH2 index,
      // then decrement both channels.
      // When CH1 and CH2 indexes fall to the CH3 index, decrement all three
      // channels.
      if(CH1_i > 0){
```

```
            CH1_PWM = CH1_DATA[CH1_i--];
         }

         if( (CH2_i > 0) && (CH2_i >= (CH1_i - 1)) ){

            CH2_PWM = CH2_DATA[CH2_i--];
         }

         #if(THREE_CHANNEL)
         if( (CH3_i > 0) && (CH3_i >= (CH1_i - 1)) ){

            CH3_PWM = CH3_DATA[CH3_i--];
         }
         #endif // THREE_CHANNEL


        // UPDATE THE PWM OUTPUT FOR CH1, CH2, CH3
        PCA0CPH0 = CH1_PWM;
        PCA0CPH1 = CH2_PWM;
        #if(THREE_CHANNEL)
        PCA0CPH2 = CH3_PWM;
        #endif // THREE_CHANNEL

     } else {

         ADC0_ISR_i++;

     }
   break;



   } // end switch(STATE)



   // switch to next channel
   CH++;
   if(CH >= 3){
      CH = 0;
      if(STATE == MON){
         MONITOR_INPUT = !MONITOR_INPUT; // Toggle monitoring between
      }                                  // inputs and outputs
   }



}// end ADC0_ISR


//-----------------------------------------------------------------------------
// Support Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// wait_ms
//-----------------------------------------------------------------------------
//
```

```
// This routine inserts a delay of <ms> milliseconds.
//
void wait_ms(int ms)
{
  int ms_save = ms;

   #if(F330)
      TMR3CN = 0x00;                     // Configure Timer 3 as a 16-bit
                                         // timer counting SYSCLKs/12
      TMR3RL = -(SYSCLK/1000/12);        // Timer 3 overflows at 1 kHz
      TMR3 = TMR3RL;

      TMR3CN  |= 0x04;                   // Start Timer 3

      while(ms){
         TMR3CN  &= ~0x80;               // Clear overflow flag
         while(!(TMR3CN & 0x80));        // wait until timer overflows
         ms--;                           // decrement ms
      }

      TMR3CN  &= ~0x04;                  // Stop Timer 3
   #else

      // DELAY <MS> milliseconds using Timer 0
      TMOD = 0x02;                       // Timer0 Mode 2
      CKCON &= ~0x0F;                    // Clear Timer0 bits
      CKCON |= 0x02;                     // Timer0 counts SYSCLK/48

      TH0 = -(SYSCLK/3000/48);           // Timer 0 overflows at 3 kHz

      TR0 = 1;                           // Start Timer 0

      // repeat this loop three times, each loop taking 0.333 ms * <MS>
      while(ms){
         TF0 = 0;                        // clear overflow flag
         while(!TF0);                    // wait until timer overflows
         ms--;                           // decrement ms
      }

      ms = ms_save;

      while(ms){
         TF0 = 0;                        // clear overflow flag
         while(!TF0);                    // wait until timer overflows
         ms--;                           // decrement ms
      }

      ms = ms_save;

      while(ms){
         TF0 = 0;                        // clear overflow flag
         while(!TF0);                    // wait until timer overflows
         ms--;                           // decrement ms
      }

      TR0 = 0;                           // Stop Timer 0
   #endif // F330
}
```

```
//-----------------------------------------------------------------------------
// FLASH_ErasePage
//-----------------------------------------------------------------------------
//
// This routine erases the FLASH page at <addr>.
//
void FLASH_ErasePage(unsigned addr)
{
   bit EA_SAVE = EA;                     // Save Interrupt State
   char xdata * idata pwrite;            // FLASH write/erase pointer

   pwrite = (char xdata *) addr;         // initalize write/erase pointer

   EA = 0;                               // Disable Interrupts

   FLKEY = 0xA5;                         // Write first key code
   FLKEY = 0xF1;                         // Write second key code

   PSCTL |= 0x03;                        // MOVX writes target FLASH
                                         // Enable FLASH erasure

   *pwrite = 0;                          // Initiate FLASH page erase

   PSCTL = 0x00;                         // Disable FLASH writes/erases

   EA = EA_SAVE;                         // Restore Interrupt State

}

//-----------------------------------------------------------------------------
// FLASH_Write
//-----------------------------------------------------------------------------
//
// This routine writes a set of bytes to FLASH memory.  The target address
// is given by <dest>, <src> points to the array to copy, and <num> is the
// size of the array.
//
void FLASH_Write(unsigned dest, char *src, unsigned num)
{
   unsigned idata i;                     // loop counter
   char xdata * idata pwrite;            // FLASH write/erase pointer
   char the_data;                        // holds data to write to FLASH
   bit EA_SAVE = EA;                     // Save Interrupt State

   pwrite = (char xdata*) dest;          // initialize write/erase pointer
                                         // to target address in FLASH

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

      the_data = *src++;                 // read data byte

      EA = 0;                            // disable interrupts

      FLKEY = 0xA5;                      // Write first key code
      FLKEY = 0xF1;                      // Write second key code
```

```
      PSCTL |= 0x01;                        // PSWE = 1; MOVX writes target FLASH
      *pwrite = the_data;                   // write the data
      PSCTL &= ~0x01;                       // PSWE = 0; MOVX writes target XRAM

      EA = EA_SAVE;                         // restore interrupts
      pwrite++;                             // advance write pointer
   }

}

//-----------------------------------------------------------------------------
// Print_Menu
//-----------------------------------------------------------------------------
//
// This routine prints the system menu to the UART
//

#if(UART_ENABLE)

void Print_Menu(void)
{
   puts("\n\nConfig Menu:\n");
   puts("1. Set Ramp Rate");
   puts("2. Set VAL wait time");
   puts("3. Set MON wait time") ;
   puts("4. Calibrate and Save Changes");
   puts("?. Print Menu");


}

#endif // UART_ENABLE

//-----------------------------------------------------------------------------
// CAL State Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// Calibrate
//-----------------------------------------------------------------------------
//
//  This Routine configures and calibrates the device.
//
void Calibrate(void)
{
   int  RAMP_RATE_SAV = RAMP_RATE;
   int  VAL_WAITTIME_SAV = VAL_WAITTIME;
   int  MON_WAITTIME_SAV = MON_WAITTIME;

   char temp_char;                       // temporary char
   int v_cal;
   bit cal_complete = 0;

#if(UART_ENABLE)


   int xdata timeout = 10000;            // 10 second delay

   #define input_str_len 10              // buffer to hold characters entered
```

```
char xdata input_str[input_str_len];   // at the command prompt
int xdata input_int;

// Start 10 sec timeout and also poll for UART activity on RX
RI0 = 0;
while (timeout > 0){
   if(RI0){
      break;
   } else {
      puts("PRESS ANY KEY TO CONTINUE");
      wait_ms(1000);
      timeout -= 1000;
   }

}

// timeout has passed or user pressed a key

// execute the following code if the user pressed a key,
// otherwise, skip this code and calibrate device
if(RI0){



   RI0 = 0;
   Print_Menu();

   while(1){

      puts("\nEnter a command >");
      gets(input_str, input_str_len);

      switch(input_str[0]){

         case '1':                     // Set Ramp Rate
            puts("\nEnter the new Ramp Rate (250 - 500)[V/s]:");
            gets(input_str, input_str_len);
            input_int = atoi(input_str);

            // validate
            while(!(input_int >= 250 && input_int <= 500)){
               puts("\nEnter a valid Ramp Rate between 250 and 500 [V/s]:");
               gets(input_str, input_str_len);
               input_int = atoi(input_str);
            }


            RAMP_RATE_SAV = input_int;

            break;

         case '2':                     // Set input settling time
            puts("\nEnter the new (Input Valid -> Ramp Start) wait time [ms]:");
            gets(input_str, input_str_len);
            input_int = atoi(input_str);

            // validate
            while(!(input_int >= 10 && input_int <= 30000)){
               puts("\nEnter timeout between 10 and 30000ms:");
```

```
                gets(input_str, input_str_len);
                input_int = atoi(input_str);
              }

              VAL_WAITTIME_SAV = input_int;

              break;

             case '3':                    // Set S_RESET wait time
              puts("\nEnter the new (Output Valid -> S_RESET Rising) wait time [ms]:");
              gets(input_str, input_str_len);
              input_int = atoi(input_str);

              // validate
              while(!(input_int >= 10 && input_int <= 30000)){
                 puts("\nEnter timeout between 10 and 30000ms:");
                 gets(input_str, input_str_len);
                 input_int = atoi(input_str);
              }

              MON_WAITTIME_SAV = input_int;

              break;

             case '4':
              break;

            default:
              puts("** Invalid Input **\n");
              Print_Menu();
              break;
         }

         // If user selected calibrate and save
         if(input_str[0] == '4'){

            // exit the while loop
            break;
         }
      }

   }

#endif // UART_ENABLE

       #if(UART_ENABLE)
       puts("\nVALIDATING INPUTS");
       #endif // UART_ENABLE

       GlobalVarInit();
       ValidateInput ();

       #if(UART_ENABLE)
       puts("CALIBRATING");
       #endif // UART_ENABLE

       // Erase the FLASH data page
       FLASH_ErasePage(CH1_DATA_ADDR);
```

```
// Write parameters to FLASH
FLASH_Write(RAMP_RATE_ADDR,    (char*) &RAMP_RATE_SAV,    sizeof(int));
FLASH_Write(VAL_WAITTIME_ADDR, (char*) &VAL_WAITTIME_SAV, sizeof(int));
FLASH_Write(MON_WAITTIME_ADDR, (char*) &MON_WAITTIME_SAV, sizeof(int));

// Set PWM codes to the minimum
PCA0CPH0 = 0x00;
PCA0CPH1 = 0x00;
#if(THREE_CHANNEL)
PCA0CPH2 = 0x00;
#endif // THREE_CHANNEL

// Start Calibration
v_cal = 0;
while ( v_cal < 3500) {
  CH1_Calibrate(v_cal);
  CH2_Calibrate(v_cal);
  #if(THREE_CHANNEL)
  CH3_Calibrate(v_cal);
  #endif // THREE_CHANNEL

  if(v_cal > 50 ){

     v_cal += VSTEP;

  } else {

     v_cal += VSTEP/16;
  }

}

// Use the ADC0_ISR to ramp down all outputs
USER_SHUTDOWN = 1;
STATE = SHUTDOWN;

// Enable ADC0 End of Conversion Interrupts
#if(F330)
  EIE1   |= 0x08;
#else
  EIE1   |= 0x04;
#endif // F330

EA = 1;                         // Enable Global Interrupts
while(STATE != OFF);            // Wait until outputs shut down
EA = 0;                         // Disable Global Interrupts

// Set the CAL_DONE flag to 0x00 to indicate that calibration is
// complete
temp_char = 0x00;
FLASH_Write(CAL_DONE_ADDR, (char*) &temp_char, 1);

#if(UART_ENABLE)
puts("CALIBRATION COMPLETE\n\nCHANGES SAVED");
#endif // UART_ENABLE
```

```
}
//-----------------------------------------------------------------------------
// CH1_Calibrate
//-----------------------------------------------------------------------------
//
//  This routine increments CH1 output voltage until it reaches <v_target>.
//  It then records the current PWM code in the calibration table.
//
void CH1_Calibrate(int v_target)
{
   int i;                           // software loop counter
   int adc_code;

   int v = 0;                       // voltage measured at output (mV)

   unsigned long acc;               // accumulator for ADC integrate and dump

   static int pData;                // initialize data pointer

   char temp_char;                  // temporary char

   bit done = 0;                    // completion flag

   // Select CH1 output as ADC mux input
   #if(F330)
      AMX0P = VOUT_PIN[CH1];
   #else
      AMX0SL = VOUT_PIN[CH1];
   #endif // F330

   // wait for output to settle
   wait_ms(1);

   // If the target voltage is 0V, initialize the pointer to the calibration
   // table to the beginning of CH1_DATA
   if(v_target == 0){

      pData = CH1_DATA_ADDR;

   }


   // Check the CH1 output voltage and keep increasing until v >= v_target
   // Do not allow the PWM code to overflow
   do{

      // obtain 256 samples
      acc = 0;
      for(i = 0; i < 256; i++){

            // obtain one sample
            AD0INT = 0;
            while(!AD0INT);

            // add to accumulator
            acc += ADC0;
      }

      // average the samples
```

```
      acc >>= 8;                       // divide by 256

      adc_code = (int) acc;

      // convert <acc> from a code to a voltage and translate up
      // Vin = Vin_ADC * (R10+R11)/R11
      acc *= VREF;                     // multiply by VREF
      #if(ADC_RES == 1024L)
         acc >>= 10;                   // divide by ADC_RES = 2^10
      #elif(ADC_RES == 256L)
         acc >>= 8;                    // divide by ADC_RES = 2^8
      #elif
         #error("Unsupported ADC Resolution")
      #endif // ADC_RES
      acc *= (R10+R11);                // scale by attenuator ratio
      acc /= R11;

      // The accumululator now contains CH1 output voltage (mV)
      v = (int) acc;                   // copy output voltage to <v>

      // If output voltage has not yet reached the target and we have not
      // yet reached the maximum PWM code, increment the PWM code
      if( (v < v_target) && (PCA0CPH0 != 0xFF) &&
          (adc_code <= (CH1_TARGET_CODE-(CAL_DETECT_ERR)))
        ){
          PCA0CPH0++;
      } else {

          done = 1;
      }

   } while (!done );


   // At this point (v >= v_target) or (PCACP0H == 0xFF).
   // The current output voltage is greater than the target voltage
   // or we have reached the maximum PWM code.

   // If we have not reached the maximum PWM code, record this code
   // in FLASH. No action is required if the current PWM code is 0xFF
   if(PCA0CPH0 != 0xFF && (adc_code <= (CH1_TARGET_CODE-(CAL_DETECT_ERR)))){
      temp_char = PCA0CPH0;
      FLASH_Write(pData, &temp_char, sizeof(char)); // Write to FLASH
      pData++;                                      // Increment FLASH write pointer
   }


}

//-----------------------------------------------------------------------------
// CH2_Calibrate
//-----------------------------------------------------------------------------
//
//  This routine increments CH2 output voltage until it reaches <v_target>.
//  It then records the current PWM code in the calibration table.
//
void CH2_Calibrate(int v_target)
{
   int i;                            // software loop counter
```

```
char temp_char;                    // temporary char
int adc_code;

int v = 0;                         // voltage measured at output (mV)

unsigned long acc;                 // accumulator for ADC integrate and dump

static int pData;                  // initialize data pointer

bit done = 0;                      // completion flag

// Select CH2 output as ADC mux input
#if(F330)
   AMX0P = VOUT_PIN[CH2];
#else
   AMX0SL = VOUT_PIN[CH2];
#endif // F330

// wait for output to settle
wait_ms(1);

// If the target voltage is 0V, initialize the pointer to the calibration
// table to the beginning of CH2_DATA
if(v_target == 0){

   pData = CH2_DATA_ADDR;

}


// Check the CH2 output voltage and keep increasing until v >= v_target
// Do not allow the PWM code to overflow
do{

  // obtain 256 samples
   acc = 0;
   for(i = 0; i < 256; i++){

         // obtain one sample
         AD0INT = 0;
         while(!AD0INT);

         // add to accumulator
         acc += ADC0;
   }

   // average the samples for a 10-bit result
   acc >>= 8;                      // divide by 256

   adc_code = acc;

   // convert <acc> from a code to a voltage
   acc *= VREF;                    // multiply by VREF
   #if(ADC_RES == 1024L)
      acc >>= 10;                  // divide by ADC_RES = 2^10
   #elif(ADC_RES == 256L)
      acc >>= 8;                   // divide by ADC_RES = 2^8
   #elif
      #error("Unsupported ADC Resolution")
```

```
      #endif // ADC_RES

      // The accumululator now contains CH2 output voltage (mV)
      v = (int) acc;                  // copy output voltage to <v>

      // If output voltage has not yet reached the target and we have not
      // yet reached the maximum PWM code, increment the PWM code
      if( (v < v_target) && (PCA0CPH1 != 0xFF) &&
          (adc_code <= (CH2_TARGET_CODE-CAL_DETECT_ERR))
        ){
         PCA0CPH1++;
      } else {

          done = 1;
      }

   } while (!done );


   // At this point (v >= v_target) or (PCACP0H == 0xFF).
   // The current output voltage is greater than the target voltage
   // or we have reached the maximum PWM code.

   // If we have not reached the maximum PWM code, record this code
   // in FLASH. No action is required if the current PWM code is 0xFF
   if(PCA0CPH1 != 0xFF && (adc_code <= (CH2_TARGET_CODE-CAL_DETECT_ERR))){
      temp_char = PCA0CPH1;
      FLASH_Write(pData, &temp_char, sizeof(char)); // Write to FLASH
      pData++;                                      // Increment FLASH write pointer
   }


}


//-----------------------------------------------------------------------------
// CH3_Calibrate
//-----------------------------------------------------------------------------
//
//  This routine increments CH3 output voltage until it reaches <v_target>.
//  It then records the current PWM code in the calibration table.
//
#if(THREE_CHANNEL)
void CH3_Calibrate(int v_target)
{
   int xdata i;                       // software loop counter
   char xdata temp_char;              // temporary char
   int xdata adc_code;

   int xdata v = 0;                   // voltage measured at output (mV)

   unsigned long xdata acc;           // accumulator for ADC integrate and dump

   static int pData;                  // initialize data pointer

   bit done = 0;                      // completion flag

   // Select CH3 output as ADC mux input
   #if(F330)
```

```
   AMX0P = VOUT_PIN[CH3];
#else
   AMX0SL = VOUT_PIN[CH3];
#endif // F330

// wait for output to settle
wait_ms(1);

// If the target voltage is 0V, initialize the pointer to the calibration
// table to the beginning of CH3_DATA
if(v_target == 0){

   pData = CH3_DATA_ADDR;

}


// Check the CH3 output voltage and keep increasing until v >= v_target
// Do not allow the PWM code to overflow
do{

   // obtain 256 samples
   acc = 0;
   for(i = 0; i < 256; i++){

         // obtain one sample
         AD0INT = 0;
         while(!AD0INT);

         // add to accumulator
         acc += ADC0;
   }

   // average the samples
   acc >>= 8;                       // divide by 256

   adc_code = acc;
   // convert <acc> from a code to a voltage
   acc *= VREF;                     // multiply by VREF

   #if(ADC_RES == 1024L)
      acc >>= 10;                   // divide by ADC_RES = 2^10
   #elif(ADC_RES == 256L)
      acc >>= 8;                    // divide by ADC_RES = 2^8
   #elif
      #error("Unsupported ADC Resolution")
   #endif // ADC_RES

   // The accumululator now contains CH3 output voltage (mV)
   v = (int) acc;                   // copy output voltage to <v>

   // If output voltage has not yet reached the target and we have not
   // yet reached the maximum PWM code, increment the PWM code
   if( (v < v_target) && (PCA0CPH2 != 0xFF) &&
       (adc_code <= (CH3_TARGET_CODE-CAL_DETECT_ERR))
     ){
      PCA0CPH2++;
   } else {
```

```
            done = 1;
         }

      } while (!done );


      // At this point (v >= v_target) or (PCACP0H == 0xFF).
      // The current output voltage is greater than the target voltage
      // or we have reached the maximum PWM code.

      // If we have not reached the maximum PWM code, record this code
      // in FLASH. No action is required if the current PWM code is 0xFF
      if(PCA0CPH2 != 0xFF && (adc_code <= (CH3_TARGET_CODE-CAL_DETECT_ERR))){
         temp_char = PCA0CPH2;
         FLASH_Write(pData, &temp_char, sizeof(char)); // Write to FLASH
         pData++;                                      // Increment FLASH write pointer
      }


}
#endif


//-----------------------------------------------------------------------------
// Initialization Routines
//-----------------------------------------------------------------------------

#if(F330)
//-----------------------------------------------------------------------------
// VDM_Init
//-----------------------------------------------------------------------------
//
// Initialize VDD Monitor for the F330 and the F300
//
void VDM_Init (void)
{
   VDM0CN |= 0x80;                  // Enable VDD monitor
   while(!(VDM0CN & 0x40));         // wait for VDD to stabilize

   RSTSRC = 0x02;                   // Set VDD monitor as a reset source

}

#else

//-----------------------------------------------------------------------------
// VDM_Init
//-----------------------------------------------------------------------------
//
// Initialize VDD Monitor for the F330 and the F300
//
void VDM_Init (void)
{
   RSTSRC = 0x02;                   // Set VDD monitor as a reset source
}

#endif // F330
```

```
//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// Configure the system clock to use the internal 24.5MHz oscillator as its
// clock source and enable the missing clock detector.
//

#if(F330)

void SYSCLK_Init (void)
{
   OSCICN = 0x83;                   // set clock to 24.5 MHz


   RSTSRC  = 0x06;                  // enable missing clock detector
                                    // and leave the VDD monitor enabled

}

#else

void SYSCLK_Init (void)
{
   OSCICN = 0x07;                   // set clock to 24.5 MHz

   RSTSRC  = 0x06;                  // enable missing clock detector
                                    // and leave the VDD monitor enabled
}

#endif // F330

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO pins to the following pinout for the 'F330:
//
// Port 0
//  P0.0 - GPIO
//  P0.1 - CEX0         (CH1 PWM Output)
//  P0.2 - GPIO         (POWER_G Signal)
//  P0.3 - GPIO
//  P0.4 - UART TX
//  P0.5 - UART RX
//  P0.6 - Analog Input (CH1 Vin)
//  P0.7 - GPIO         (S2 Switch)
//
// Port 1
//  P1.0 - Analog Input (CH1 Vout)
//  P1.1 - Analog Input (CH2 Vin)
//  P1.2 - CEX1         (CH2 PWM Output)
//  P1.3 - Analog Input (CH2 Vout)
//  P1.4 - Analog Input (CH3 Vin)
//  P1.5 - CEX2         (CH3 PWM Output)
//  P1.6 - Analog Input (CH3 Vout)
//  P1.7 - GPIO         (S_RESET Signal)
//
// Port 2
```

```
//  P2.0 - C2D
//
#if(F330)

void PORT_Init (void)
{
   // Momentarily discharge the output capacitors

   P1 &= ~0x49;                        // Write a '0' to P1.0, P1.3 and P1.6
                                       // to discharge the capacitors on CH1_VOUT,
                                       // CH2_VOUT, CH3_VOUT.
   P1 |=  0x49;                        // Return the same port pins above to
                                       // their reset state



   // Make sure the CAL/SD switch has charged up after a reset
   // to prevent calibrating the device when the S2 switch is not pressed.

   // Momentarily drive the S2 (P0.7) signal high.
   XBR1 = 0x40;                        // Enable Crossbar

   P0MDOUT |=  0x80;                   // Configure to push-pull
   S2 = 1;
   wait_ms(1);
   P0MDOUT &= ~0x80;                   // Configure to open-drain

   XBR1 = 0x00;                        // Disable Crossbar


   // F330 Port 0 Initialization

   P0SKIP   = ~0x32;                   // Skip all Port0 pins except for
                                       // P0.1 (CH1 PWM output),
                                       // P0.4 and P0.5 (UART pins)
                                       // in the Crossbar

   P0MDIN   = ~0x40;                   // Configure P0.6 (CH1 Vin) as an
                                       // analog input

   P0MDOUT  = 0x16;                    // Enable UART TX, CEX0, and POWER_G
                                       // signal as push-pull



   // F330 Port 1 Initialization

   P1SKIP   = ~0x24;                   // Skip all pins configured as analog
                                       // inputs in Port1
   P1MDIN   = ~0x5B;                   // Configure P1.0, P1.1, P1.3, P1.4, and
                                       // P1.6 as analog inputs

   P1MDOUT  = 0xA4;                    // PWM outputs and S_RESET is push-pull

   // F330 Crossbar Initialization

   XBR0     = 0x01;                    // UART TX0,RX0 routed to P0.4 and P0.5

   XBR1     = 0x43;                    // Enable crossbar and weak pullups
```

```
                                    // CEX0, CEX1, and CEX2 routed to
                                    // P0.1, P1.2, and P1.5
}

#else

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO pins to the following pinout for the 'F300:
//
// Port 0
//  P0.0 - GPIO/VREF    (CAL/SD switch)
//  P0.1 - Analog Input (CH2 Vin)
//  P0.2 - Analog Input (CH1 Vin)
//  P0.3 - CEX0         (CH1 PWM Output)
//  P0.4 - CEX1         (CH2 PWM Output)
//  P0.5 - Analog Input (CH1 Vout)
//  P0.6 - Analog Input (CH2 Vout)
//  P0.7 - GPIO         (S_RESET Signal)
//
void PORT_Init (void)
{
   // Momentarily discharge the output capacitors

   P0 &= ~0x60;                     // Write a '0' to P0.5 and P0.6
                                    // to discharge the capacitors on CH1_VOUT
                                    // and CH2_VOUT
   P0 |=  0x60;                     // Return the same port pins above to
                                    // their reset state

   // Make sure the CAL/SD switch has charged up after a reset
   // to prevent calibrating the device when the S2 switch is not pressed.

   // Momentarily drive the CAL/SD/VREF (P0.0) switch

   XBR2 = 0x40;                     // Enable Crossbar

   P0MDOUT |=  0x01;                // Configure to push-pull
   S2 = 1;
   wait_ms(1);
   P0MDOUT &= ~0x01;                // Configure to open-drain

   XBR2 = 0x00;                     // Disable Crossbar



   // F300 Port 0 Initialization
   XBR0 = 0x07;                     // Skip the first three pins in P0
                                       // in the Crossbar

   P0MDIN = ~0x66;                  // Configure Vin and Vout pins as
                                       // analog inputs

   P0MDOUT = 0x98;                  // Configure CEX0, CEX1 and S_RESET
                                       // to push-pull
```

```
   XBR1 = 0x80;                        // Enable CEX0 and CEX1 in the Crossbar

   XBR2 = 0x40;                        // Enable Crossbar and Weak Pullups
}

#endif // F330

//-----------------------------------------------------------------------------
// EX0_Init
//-----------------------------------------------------------------------------
//
// This routine initializes external interrupt 0 to monitor the CAL/SD switch.
//
#if(F330)

void EX0_Init(void)
{

      IT01CF &= ~0x0F;                    // Clear EX0 bits
      IT01CF |= 0x07;                     // Monitor P0.7 (active low)
      IT01CF &= ~0x08;                    // active low
      IT0 = 1;                            // Edge Triggered
      IE0 = 0;                            // Clear Interrupt Flag

}

#else

void EX0_Init(void)
{

      IT01CF &= ~0x0F;                    // Clear EX0 bits
      IT01CF |=  0x00;                    // Monitor P0.0 (active low)
      IT01CF &= ~0x08;                    // active low
      IT0 = 1;                            // Edge Triggered
      IE0 = 0;                            // Clear Interrupt Flag
}

#endif  // F330


//-----------------------------------------------------------------------------
// ADC0_Init
//-----------------------------------------------------------------------------
//
// Configure ADC0 to start conversions on Timer 2 overflows.
//
#if(F330)

void ADC0_Init (void)
{
   ADC0CN = 0x02;                      // ADC0 disabled; normal tracking
                                       // mode; ADC0 conversions are initiated
                                       // on Timer 2 overflows;
   // Configure ADC MUX input
   AMX0P = VOUT_PIN[CH1];              // Select CH1 Output as ADC0 input
   AMX0N = 0x11;                       // select GND as negative mux input

   // Configure SAR Clock frequency
```

```
   ADC0CF = (SYSCLK/3000000) << 3;     // ADC conversion clock <= 3MHz
                                       // Right Justified Data

   // Configure Voltage Reference
   // Turn on internal reference buffer and output to P0.0
   REF0CN = 0x03;                      // VREF pin used as reference,
                                       // Bias generator is on.
                                       // Internal Reference Buffer Enabled

   wait_ms(2);                         // Wait 2 ms for VREF to settle

   ADC0CN |= 0x80;                     // enable ADC

}

#else

void ADC0_Init (void)
{
   ADC0CN = 0x02;                      // ADC0 disabled; normal tracking
                                       // mode; ADC0 conversions are initiated
                                       // on Timer 2 overflows;
   // Configure ADC MUX input
   AMX0SL = VOUT_PIN[CH1];             // Select CH1 Output as ADC0 input
                                       // and GND as negative mux input

   // Configure SAR Clock frequency
   ADC0CF = (SYSCLK/6000000) << 3;     // ADC conversion clock <= 6MHz
                                       // 8-bit data in 'F300

   // Configure PGA gain for 'F300 ('F330 gain always set to 1)
   ADC0CF |= 0x01;                     // PGA gain = 1


   // Configure Voltage Reference
   REF0CN = 0x0A;                      // VDD used as voltage reference
                                       // Bias generator is on.

   wait_ms(2);                         // Wait 2 ms for VREF to settle

   ADC0CN |= 0x80;                     // enable ADC
}

#endif

//-----------------------------------------------------------------------------
// PCA_Init
//-----------------------------------------------------------------------------
//
// Configure all PCA modules to PWM output mode, using SYSCLK as a timebase.
//
void PCA_Init(void)
{
   PCA0MD = 0x08;                      // Set PCA timebase to SYSCLK
                                       // Disable PCA overflow interrupt

   // Configure Capture/Compare Module 0 (Channel 1 PWM output)
   PCA0CPM0 = 0x42;                    // configure for 8-bit PWM
```

```
   // Configure Capture/Compare Module 1 (Channel 2 PWM output)
   PCA0CPM1 = 0x42;                    // configure for 8-bit PWM

   // Configure Capture/Compare Module 2 (Channel 3 PWM output)
   #if(THREE_CHANNEL)
   PCA0CPM2 = 0x42;                    // configure for 8-bit PWM
   #endif // THREE_CHANNEL

   PCA0CPH0 = 0x00;
   PCA0CPH1 = 0x00;

   #if(THREE_CHANNEL)
   PCA0CPH2 = 0x00;
   #endif // THREE_CHANNEL

   CR = 1;                             // start PCA timer


}

//-----------------------------------------------------------------------------
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <BAUDRATE> and 8-N-1.
// The minimum standard baud rate supported by this function is 57600 when
// the system clock is running at 24.5 MHz.
//

#if(UART_ENABLE)

void UART0_Init (void)
{
   SCON0 = 0x10;                       // SCON0: 8-bit variable bit rate
                                       //        level of STOP bit is ignored
                                       //        RX enabled
                                       //        ninth bits are zeros
                                       //        clear RI0 and TI0 bits

   TH1 = -(SYSCLK/BAUDRATE/2);
   CKCON |=  0x08;                     // T1M = 1; SCA1:0 = xx


   TL1 = TH1;                          // init Timer1
   TMOD &= ~0xf0;                      // TMOD: timer 1 in 8-bit autoreload
   TMOD |=  0x20;
   TR1 = 1;                            // START Timer1
   TI0 = 1;                            // Indicate TX0 ready
}

#endif // UART_ENABLE

//-----------------------------------------------------------------------------
// Timer2_Init
//-----------------------------------------------------------------------------
//
// Configure Timer2 to auto-reload at interval specified by <counts>
// using SYSCLK as its time base.
//
```

```
//
//
#if(F330)

void Timer2_Init(int counts)
{
   TMR2CN  = 0x00;                     // resets Timer 2, sets to 16 bit mode

   CKCON  |= 0x10;                     // use system clock

   TMR2RL  = -counts;                  // initialize reload value
   TMR2 = TMR2RL;                      // initialize Timer 2
}

#else

void Timer2_Init(int counts)
{
   TMR2CN  = 0x00;                     // resets Timer 2, sets to 16 bit mode

   CKCON  |= 0x20;                     // use system clock

   TMR2RL  = -counts;                  // initialize reload value
   TMR2 = TMR2RL;                      // initialize Timer 2
}

#endif // F330
```

Appendix D - Firmware (Header File)

```
//-----------------------------------------------------------------------------
// PS_V1.3.h
//-----------------------------------------------------------------------------
//
// AUTH: FB
// DATE: 26 JUN 03
//
// VERSION: 1.3.0
//
// Header file contianing configuration settings for Power Sequencing
// Software (PS_V1.3.c).
//
// Target: C8051F330 and C8051F300
// Tool chain: KEIL C51
//

// Step 1 - Select a device
#define F330                     1        // Select '1' for the 'F330 and
                                          // select '0' for the 'F300

// Step 2 - Select features to enable on the 'F330. Note that these features are
// not available for the 'F300.
#if(F330)
   #define UART_ENABLE           0        // Enables configuration through UART
   #define THREE_CHANNEL         1        // Enables the third channel
#endif // F330

// Step 3 - Define System Parameters for the 'F330 and the 'F300

#define DEFAULT_RAMP_RATE        500      // Default ramp rate in V/s
#define DEFAULT_VAL_WAITTIME     100      // Default validation wait time in ms
#define DEFAULT_MON_WAITTIME     100      // Default time between POWER_G and
                                          // S_RESET rising edge in ms.

#define OVERVOLTAGE_PROTECTION   1        // Enable or Disable Overvoltage Protection
#define OVERCURRENT_PROTECTION   1        // Enable or Disable Overcurrent protection

#define RAMP_TIMEOUT_ENABLE      1        // Enables Ramp Timeout
#define RAMP_TIMEOUT             100      // Maximum time allowed for ramping (ms)

#define NUM_RETRIES              3        // The number of power-up retries the system
                                          // will attempt after a power failure

// The minimum specified Target voltage for each channel (-8% of target voltage)
#define CH1_VTARGET_MIN          3036L    // Channel 1 min target voltage in mV
#define CH2_VTARGET_MIN          1656L    // Channel 2 min target voltage in mV
#define CH3_VTARGET_MIN          1380L    // Channel 3 min target voltage in mV

// The maximum specified Target voltage for each channel (+8% of target voltage)
#define CH1_VTARGET_MAX          3564L    // Channel 1 max target voltage in mV
#define CH2_VTARGET_MAX          1944L    // Channel 2 max target voltage in mV
#define CH3_VTARGET_MAX          1620L    // Channel 3 max target voltage in mV


#define OVERCURRENT_VTH          400L     // Overcurrent threshold in mV
```

```
#define STRICT_VAL_DELTA         100L     // Overvoltage threshold is decreased and
                                          // undervoltage threshold is increased by
                                          // this amount after a power failure

// Resistor values for the attenuator on CH1
#define R10                      2800L    // Resistance in Ohms
#define R11                      5360L    // Resistance in Ohms
```

Notes:

Contact Information

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032
Email: productinfo@silabs.com
Internet: www.silabs.com