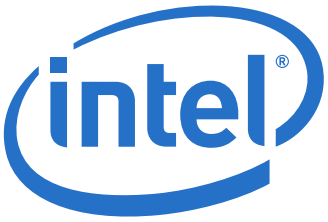


Redundant SPI Flash Support with Failover Boot Design for Intel® Architecture Platforms

Application Note

October 2014

Order Number: 331439-001

Contents

1 Introduction 5
1.1 Definition of Terms 5
1.2 Related Documents 5
2.0 Redundant SPI Flash Design Consideration 6
2.1 Redundant System Boot SPI Flash Overview 6
2.2 Failover Boot Methodologies Operational Overview 6
2.3 Redundant SPI Flash Logical Implementation 8
2.4 Redundant SPI Flash Physical Implementation 10
2.5 BIOS Consideration 13
2.6 Programmable Logic Device (PLD) Source Code 14
3.0 Summary 40

Figures

1 Redundant SPI Flash Process Overview 7
2 Redundant SPI Flash High-Level Block Diagram 9
3 Redundant SPI Flash Physical Implementation 10
4 Write/Read Timing Between the SoC and PLD 11

Tables

1 Definition of Terms 5
2 Reference Documents 5
3 Redundant SPI Signal Map 11
4 Redundant SPI PLD Registers 12

Revision History

Date	Revision	Description
October 2014	001	Initial release

1 Introduction

This application note provides detail reference designs for the redundant SPI Flash with failover boot design consideration. This application note is intended for design reference only and the implementation use is as is. This application note design is implemented in one of the Intel proof of designs for the Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure (Rangeley) Comm Collateral (RCC) Platform. In general, this application note design can apply to most Intel® Architecture (IA) with CPU/PCH or SoC design applications.

This application note provides an example reference Programmable Logic Device (PLD) code to support this feature. The application note does not contain the BIOS code implementation to this example, but provides a high-level interaction and requirement between the BIOS and the PLD. Work with the BIOS vendors to ensure the necessary steps are taken for the BIOS to handle this feature.

1.1 Definition of Terms

Table 1. Definition of Terms

Term	Definition
PDG	Platform Design Guide
PoD	Proof of Design
SKU	Stock Keeping Unit
SoC	System-on-a-Chip
SPI	Serial Peripheral Interface
PLD	Programmable Logic Device or FPGA

1.2 Related Documents

Table 2 provides a list of related documents:

Table 2. Reference Documents

Document	Document Number
[Rangeley] SoC Communications Collateral (RCC) Platform - Schematics	519127

Note: Unless otherwise noted, this document is available through the local Intel field sales representative.

§ §

2.0 Redundant SPI Flash Design Consideration

2.1 Redundant System Boot SPI Flash Overview

Many systems may require a redundant Serial Peripheral Interconnect (SPI) Flash implementation to provide failover boot methodologies and added robustness. This implementation methodology allows the platform to have two SPI devices with the same Flash image or a different version Flash image of choice. This section mentions ways to achieve the desired functionality and presents considerations when implementing a redundant SPI Flash.

The design presented in this section is one option; other designs are possible.

Many systems have two main requirements (which help define a robust update and recovery mechanism):

- Field update
  - In the field, the BIOS updates for embedded systems may need to be implemented remotely, and unit downtime must be minimal.

Note: The actual BIOS update application is beyond the scope of this design document. Refer to the specific system BIOS vendor.

- Field update failure recovery
  - During a BIOS update, a system failure event (power outage, system reset, etc.) may occur, resulting in an irrecoverable system crash. Should this occur, the boot image will likely be irrecoverable without a technician available to replace the BIOS image at the physical site. This occurrence will prevent the system from being operational.
  - Systems that implement the redundant SPI Flash with failover boot methodologies can be used to ease recovery and reduce downtime.

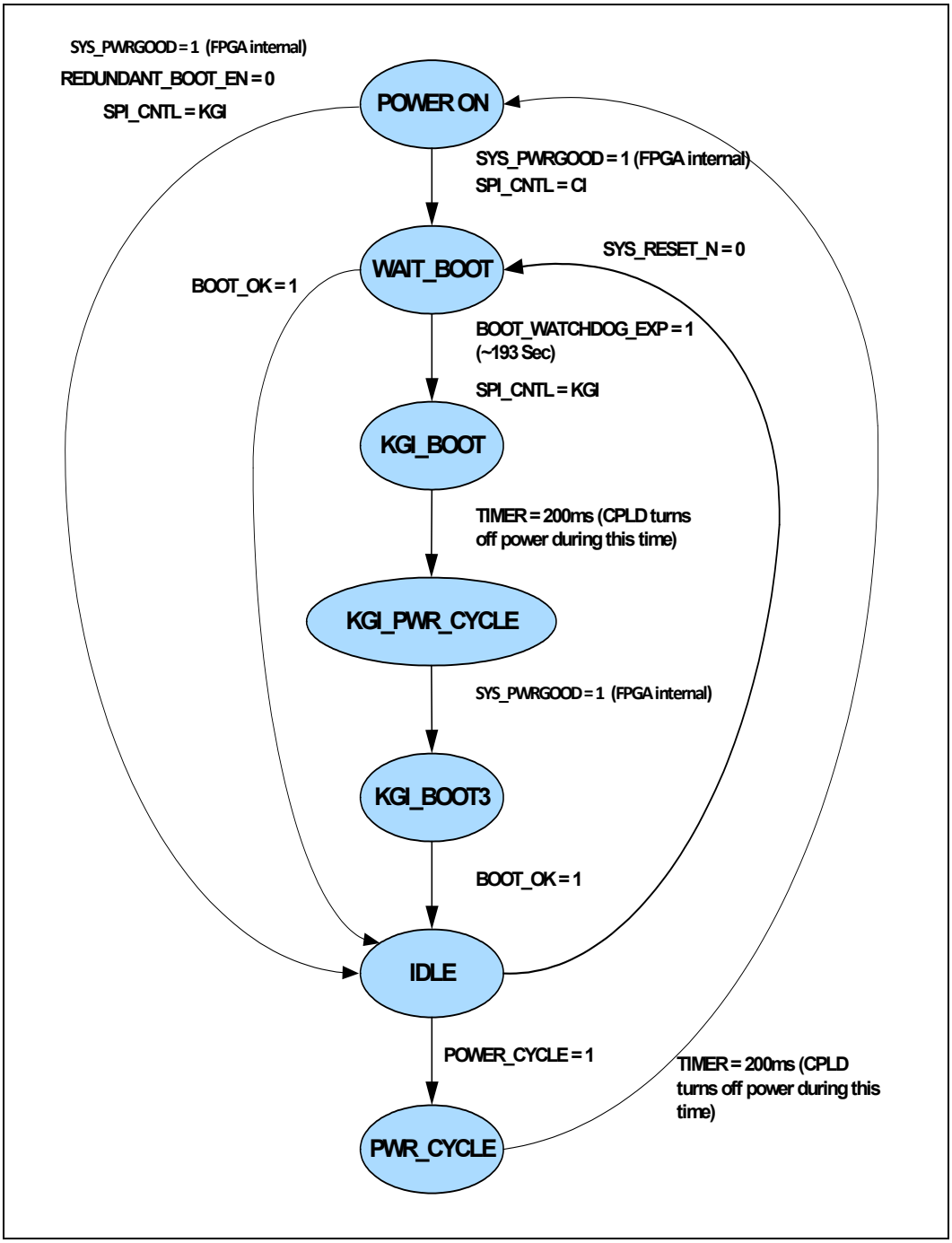
    Technicians may no longer need to travel to the physical site to update the BIOS when an SPI Flash (BIOS) update failure event occurs.

    In the event of a system failure due to a corrupted BIOS or an SPI Flash device failure, the system will automatically recover without technician involvement. The system will automatically reboot and switch to a redundant SPI Flash and bring up the system to a normal operation with minimal downtime.

2.2 Failover Boot Methodologies Operational Overview

The redundant SPI Flash with failover boot methodologies shown in Figure 1 can maintain a Known Good Image (KGI) in one Flash and allow field upgrades to occur on a second device known as the Current Image (CI). Updates only occur to the CI. If a corruption event occurs during programming of the CI, logic will determine that the system failed to boot and will boot from the KGI. Once the system is back online with the KGI, the remote user can access the system with custom remote-system-management software to determine the failed system's BIOS image and process a remote update procedure. When the update has completed, the remote user can follow the procedure to switch the chip select back to the CI and attempt a reload. If the reload is successful, then the CI remains selected and operational.

Figure 1. Redundant SPI Flash Process Overview



- The platform supports redundant SPI BIOS boot Flash with auto failover.
  - Known Good Image (KGI) Flash - factory pre-programmed
  - Current Image (CI) Flash - factory pre-programmed and field re-programmable

- The board will always try to boot from the CI Flash first.
  - If the BIOS asserts BOOT_OK, then the board proceeds to the idle state.
  - If the watchdog timer asserts BOOT_WATCHDOG_EXP, the boot failed, and the board resets and boots from the KGI Flash.
- The CI boot failure indicates a corrupted image, code bug, or no image.
- During the idle state, the software can only read the KGI Flash, but can read/write the CI Flash.

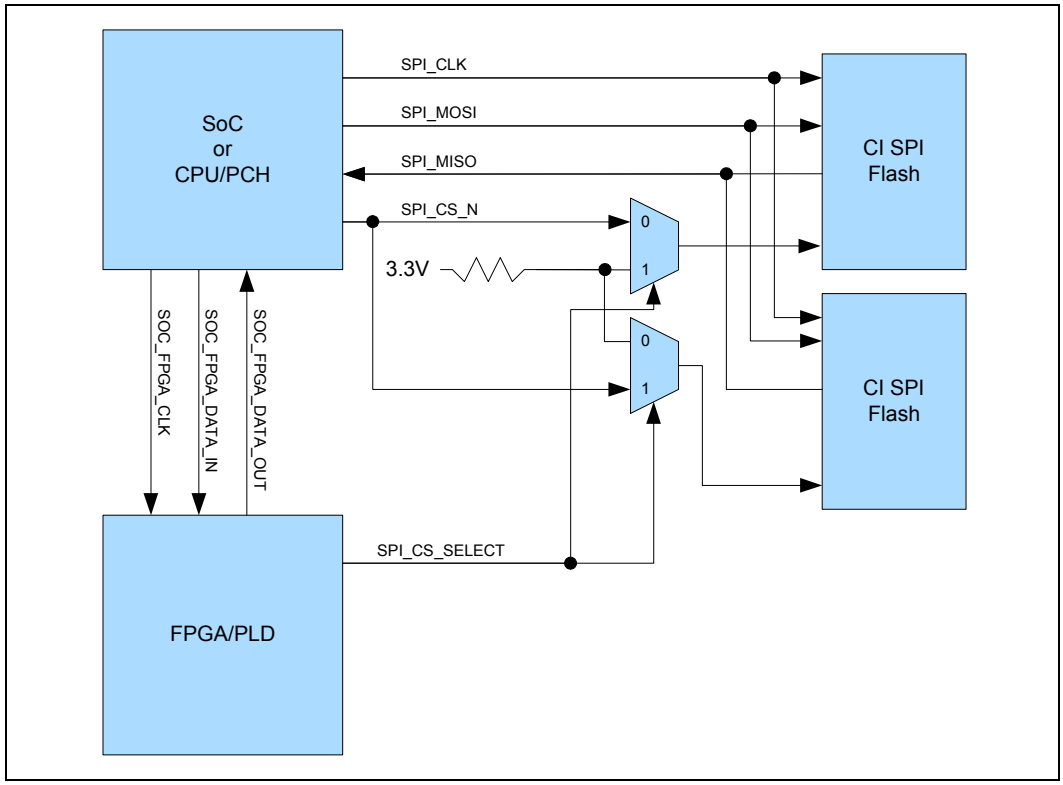
2.3 Redundant SPI Flash Logical Implementation

The platform design supports redundant SPI BIOS boot Flash with auto failover. See Figure 2. This mode is enabled by the REDUNDANT_BOOT_EN on-board jumper. There are two SPI Flash devices: the re-programmable default Current Image (CI) Flash and the read-only Known Good Image (KGI) Flash. The CI Flash can be field-reprogrammed by application software and is the latest BIOS image. The KGI Flash contains the factory-programmed BIOS image.

The failover mechanism is controlled by a PLD (or FPGA). See Figure 1. The SoC/PCH can communicate with the PLD using GPIO signals to access internal registers related to the redundant SPI feature (see Table 4, “Redundant SPI PLD Registers” on page 12). The PLD will always try to boot the board from the CI Flash first. If the BIOS sets the BOOK_OK bit, then the PLD knows the CI boot was successful and proceeds to the IDLE state. This bit should be asserted right before the BIOS hands over control to the Operating System (OS). The PLD has an internal watchdog timer which is used to help identify an unsuccessful boot. If the watchdog timer expires, the PLD will power cycle the board and boot from the KGI Flash. The duration of this watchdog is ~193 seconds (can be changed by the designer based on the platform requirement). The BIOS has the ability to disable this watchdog and should only do so if the user has selected to go into the BIOS setup screen. This is done by clearing the BOOT_WATCHDOG_EN bit. After leaving the setup screen, the BIOS should re-enable the watchdog.

Once the board has successfully booted from either the CI or KGI Flash, the PLD is in the IDLE state. At this point, application software can switch between the Flash devices by controlling the SPI_CNTL bit. This will allow the application software to switch from KGI to CI and perform a recovery procedure on the CI Flash.

Figure 2. Redundant SPI Flash High-Level Block Diagram



Note: The above diagram depicts logic connections. See the related datasheets and platform design guidelines for specific board-level schematic and layout requirements.

2.4 Redundant SPI Flash Physical Implementation

The physical implementation for the failover mechanism is controlled by a PLD (or FPGA). The design uses only three GPIO signals from the SoC/PCH. The SoC/PCH can communicate with the PLD using GPIO signals to access internal registers related to the redundant SPI feature (see Table 4, "Redundant SPI PLD Registers" on page 12). The PLD will always try to boot the board from the CI Flash first. Figure 3 shows the diagram of the design implemented consideration the platform.

Figure 3. Redundant SPI Flash Physical Implementation

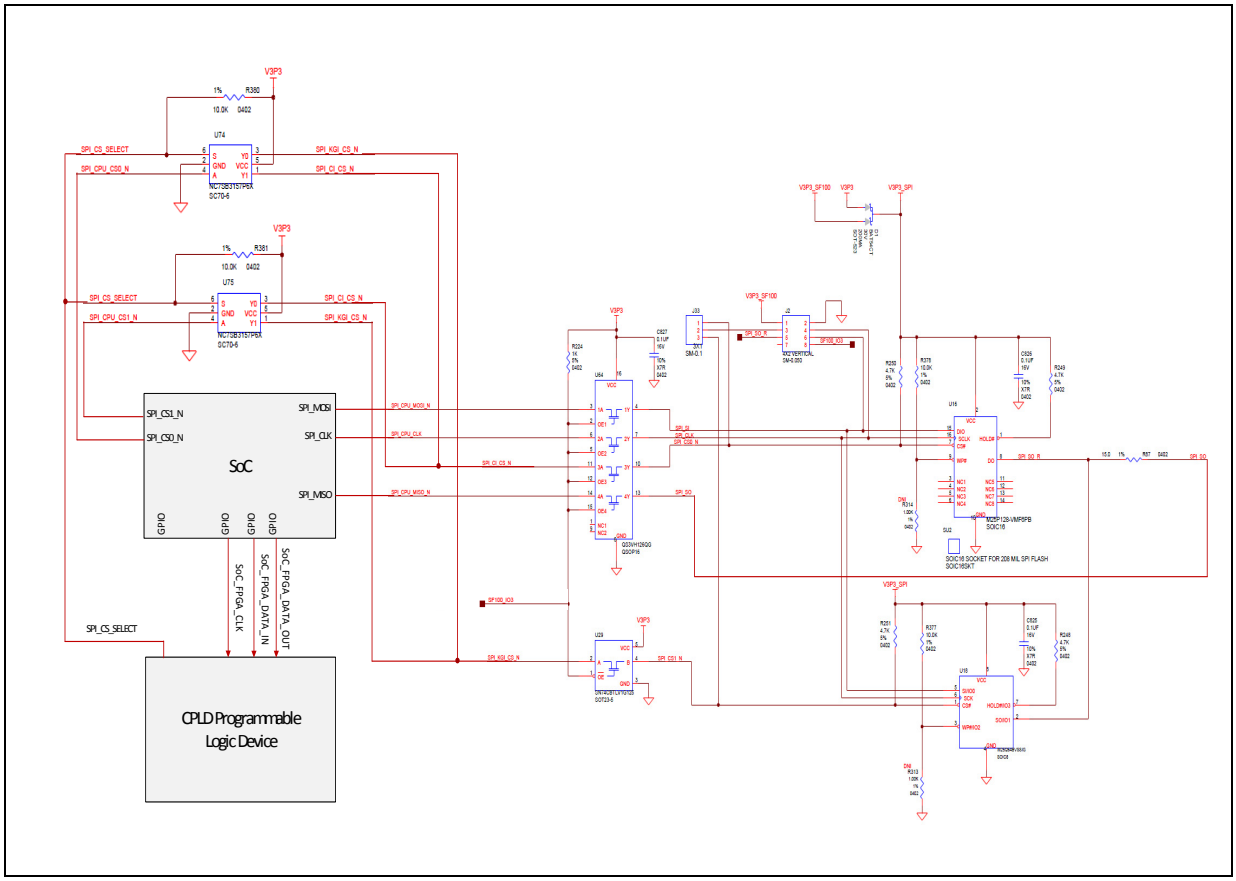


Table 3 provides a guide for the PLD (or FPGA) signals mapping to the SoC signals.

Table 3. Redundant SPI Signal Map

Source -- Signal	Destination -- Signal	Description
FPGA -- SOC_FPGA_CLK	SoC - GPIO_SUS6[1]	Clock output from the SoC to the FPGA
FPGA -- SOC_FPGA_DATA_IN	SoC - GPIO_SUS7	Data output from the SoC to the FPGA
FPGA -- SOC_FPGA_DATA_OUT	SoC - GPIO_SUS2	Data input to the SoC from the FPGA
FPGA -- SPI_CS_SELECT	CS MUX Device	The SPI CS is selected for the SPI devices from the FPGA in order to select the SPI KGI or SPI CI.
SoC -- SPI_CS0_B	CS MUX Device	The SoC SPI CS0 is selected to connect to the SPI devices which are controlled by the FPGA. The connection is to either the SPI KGI for redundant SPI device CS or SPI CI for the main SPI device CS.
SoC -- SPI_CS1_B	CS MUX Device	The SoC SPI CS1 is optional. This is not being used, but the signal is routed for the future.
SoC -- SPI_MOSI	SPI Devices -- SI/DI	The SoC SPI interface connects to both SPI devices.
SoC -- SPI_MISO	SPI Devices -- SO/DO	The SoC SPI interface connects to both SPI devices.
SoC -- SPI_CLK	SPI Devices -- SCLK	The SoC SPI interface connects to both SPI devices.

1. The GPIO number from the table is an example used in the actual design. In general, any GPIO operation under SUS well configurable as input or output can be used for the design.

The interface between the PLD and SoC is closely related to the SPI. The protocol consists of an opcode, followed by the address, then the data. Figure 4 shows the write/read timing relationship.

Figure 4. Write/Read Timing Between the SoC and PLD

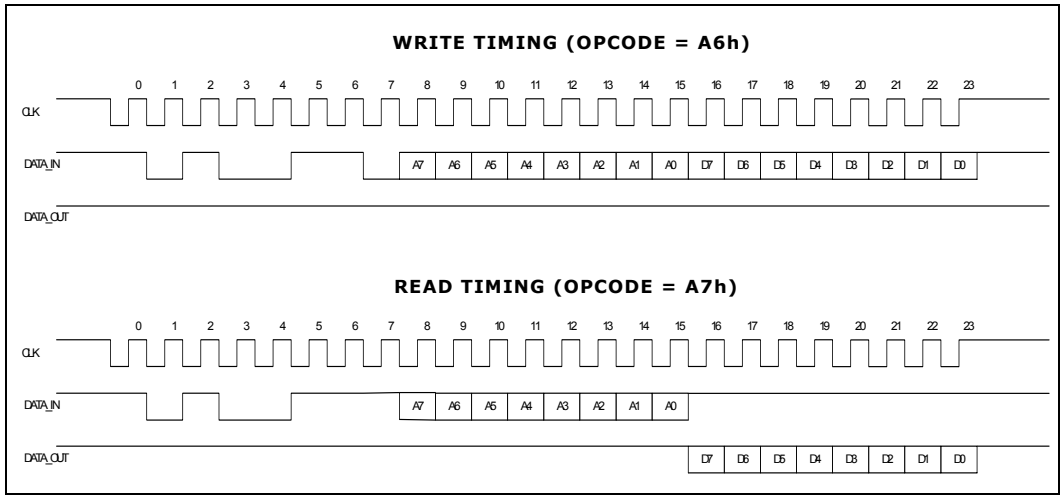


Table 4. Redundant SPI PLD Registers

Signal Name	Register/Bit Address	Read/Write	Description
REDUNDANT_BOOT_EN	Reg = 0x0 Bit = 0x0	R	Redundant Boot Enable. This is set by a jumper on the board. 1 = Enable 0 = Disable
SPI_CNTL	Reg = 0x0 Bit = 0x1	W/R	Indicates which SPI Flash is being used. The PLD will override this setting during all other states except IDLE (PLD state machine). Until the IDLE state, this bit will be read only. 0 = KGI 1 = CI
BOOT_WATCHDOG_EXP	Reg = 0x0 Bit = 0x2	R	The PLD will set this bit to indicate that the boot watchdog timer expired (193 seconds) at least once. This indicates that the board failed to boot from the CI Flash and instead booted from the KGI Flash. 1 = Boot watchdog expired. 0 = Boot watchdog not expired (default).
BOOT_WATCHDOG_EN	Reg = 0x1 Bit = 0x0	W/R	Boot watchdog enable. By default, the boot watchdog will be enabled. The BIOS should disable the watchdog if the user has entered the BIOS setup screen. 1 = Boot watchdog enabled (default). 0 = Boot watchdog disabled.
BOOK_OK	Reg = 0x2 Bit = 0x0	W/R	This bit should be set by the BIOS to indicate a successful boot. Typically this will be done right before the BIOS hands over control to the OS. 1 = Boot successful 0 = Boot in process (default)
POWER_CYCLE	Reg = 0x3 Bit = 0x0	W	Setting this bit will cause the system to power cycle. This bit would be used if the system booted from the KGI Flash and wanted to restart with the CI Flash. This bit is self clearing. 1 = Power cycle board (self clearing) 0 = Will always read back as 0.

2.5 BIOS Consideration

The designer works with the BIOS vendor of choice to implement this feature set for the platform. The BIOS needs to identify the action for the next platform boot and communicates with the Programmable Logic Device (PLD) controller to select the correct the SPI Flash device. The BIOS will interact with the PLD with read/write protocol consisting of an opcode, followed by the address, then the data shown in Figure 4 via SoC 3 GPIO signals configured as clock, data in, and data out. The BIOS also needs to recognize the registers set setting shown in Table 4.

The platform supports redundant SPI BIOS boot flash with auto failover.

- Known Good Image (KGI) Flash - factory pre-programmed
- Current Image (CI) Flash - factory pre-programmed and field re-programmable

- The BIOS will take appropriate action based on the PLD register setting. The platform will always try to boot from the CI Flash first.
  - If the BIOS asserts BOOT_OK, this bit should be set by the BIOS to indicate a successful boot = 1 (or boot in process = 0). This process will be done right before the BIOS hands over control to the OS.
  - The PLD will set the boot watchdog enabled bit = 1 at power on by default. The BIOS must disable the watchdog if the user has entered the BIOS setup screen.
  - If the BIOS did not assert BOOT_OK before the watchdog timer asserts BOOT_WATCHDOG_EXP, the boot failed, and the PLD asserts a reset and boots from the KGI Flash.
- The CI boot failure indicates corrupted image, code bug, or no image.
- During the idle state, the software can only read the KGI Flash, but can read/write the CI Flash.

2.6 Programmable Logic Device (PLD) Source Code

The PLD code represents a hardware function. Although the design has been validated on the proof of design platform (*[Rangeley] SoC Communications Collateral (RCC) Platform - Schematics*, document number 519127), **the example PLD code is for reference only and should be used as is**. This example code is also platform dependent.

```
------------------------------------------------------------------------------/
/

-- CONTENTS: Intel RCC Redandant SPI with Fail Over Boot

-- PROJECT: Intel RCC

-- DESCRIPTION:

-- This block contains the logic related to the redundant SPI feature with Fail
Over Boot.

------------------------------------------------------------------------------

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_unsigned.ALL;

ENTITY INTEL_RCC_SPI IS

PORT(

   ------------------------------------------------------------------------------
-------------

   -- Core Signal

   ------------------------------------------------------------------------------
-------------

   CLK_IN                        : IN std_logic;   -- 12.5Mhz

   RESET_N                       : IN std_logic;

------------------------------------------------------------------------------
----------

-- CPU TO FPGA Serial Communication

------------------------------------------------------------------------------
----------

SOC_FPGA_CLK                  : IN std_logic;

   SOC_FPGA_DATA_IN              : IN std_logic;

   SOC_FPGA_DATA_OUT             : OUT std_logic;

------------------------------------------------------------------------------
----------

-- External Redundant SPI Signals

------------------------------------------------------------------------------
----------
```

```
SPI_PWR_OK                    : IN std_logic;
   REDUNDANT_SPI_ENABLE          : IN std_logic;
   RESET_COMPLETE                : IN std_logic;
   SPI_CS_SELECT                 : OUT std_logic;
   PWR_ENABLE                    : OUT std_logic
 );
END INTEL_RCC_SPI;

ARCHITECTURE rtl OF INTEL_RCC_SPI IS
-- Constants
   CONSTANT cKGI : std_logic := '0';
   CONSTANT cCI : std_logic := '1';
   CONSTANT cREAD : std_logic := '1';
   CONSTANT cWRITE : std_logic := '0';
-- Power State Machine
   TYPE spi_state_typ IS (START_BOOT,
                          WAIT_BOOT,
                          KGI_BOOT,
                          KGI_BOOT2,
                          KGI_BOOT3,
                          SPI_IDLE,
                          PWR_CYCLE);
   SIGNAL spi_state: spi_state_typ;
   SIGNAL next_spi_state: spi_state_typ;
-- Serial Protocol State Machine
   TYPE serial_reg_typ IS (REG_IDLE,
                           REG_ADDRESS,
                           REG_RD_DATA,
                           REG_WR_DATA
                           );
   SIGNAL reg_state: serial_reg_typ;
   SIGNAL next_reg_state: serial_reg_typ;
   SIGNAL counter       : std_logic_vector(32 DOWNTO 0);
   SIGNAL next_counter  : std_logic_vector(32 DOWNTO 0);
```


```
SIGNAL timer_en    : std_logic;

SIGNAL timer     : std_logic;

SIGNAL next_timer : std_logic;

SIGNAL spi_mux : std_logic;

SIGNAL next_spi_mux : std_logic;

SIGNAL power_enable : std_logic;

SIGNAL next_power_enable : std_logic;

SIGNAL reg_pwr_cycle : std_logic;

SIGNAL next_reg_pwr_cycle : std_logic;

SIGNAL reg_pwr_cycle_wr_en : std_logic;

SIGNAL next_reg_pwr_cycle_wr_en : std_logic;

SIGNAL rst_reg_pwr_cycle : std_logic;

SIGNAL reg_boot_ok : std_logic;

SIGNAL next_reg_boot_ok : std_logic;

SIGNAL reg_boot_ok_wr_en : std_logic;

SIGNAL next_reg_boot_ok_wr_en : std_logic;

SIGNAL rst_reg_boot_ok : std_logic;

SIGNAL reg_watchdog_enable : std_logic;

SIGNAL next_reg_watchdog_enable : std_logic;

SIGNAL reg_watchdog_wr_en : std_logic;

SIGNAL next_reg_watchdog_wr_en : std_logic;

SIGNAL rst_reg_watchdog : std_logic;

SIGNAL reg_spi_status_wr_en : std_logic;

SIGNAL next_reg_spi_status_wr_en : std_logic;

SIGNAL watchdog_expired : std_logic;

SIGNAL next_watchdog_expired : std_logic;

SIGNAL soc_fpga_clk_del1 : std_logic;

SIGNAL soc_fpga_clk_del2 : std_logic;

SIGNAL soc_fpga_clk_del3 : std_logic;

SIGNAL soc_fpga_clk_rising : std_logic;

SIGNAL soc_fpga_clk_falling : std_logic;

SIGNAL sreg : std_logic_vector(7 DOWNTO 0);

SIGNAL next_sreg : std_logic_vector(7 DOWNTO 0);

SIGNAL reg_command : std_logic;
```

```
   SIGNAL next_reg_command : std_logic;
   SIGNAL bit_count  : std_logic_vector (3 DOWNTO 0);
   SIGNAL next_bit_count  : std_logic_vector (3 DOWNTO 0);
   SIGNAL reg_addr : std_logic_vector(7 DOWNTO 0);
   SIGNAL next_reg_addr : std_logic_vector(7 DOWNTO 0);
   SIGNAL serial_data_out : std_logic;
   SIGNAL next_serial_data_out : std_logic;
   SIGNAL read_address : std_logic_vector (7 DOWNTO 0);
   SIGNAL soc_fpga_data_in_del1 : std_logic;
   SIGNAL soc_fpga_data_in_del2 : std_logic;
   SIGNAL soc_fpga_data_in_del3 : std_logic;
   SIGNAL start_condition : std_logic;
   SIGNAL next_start_condition : std_logic;
BEGIN
   ------------------------------------------------------------------------
   -- Outputs
   ------------------------------------------------------------------------
   PWR_ENABLE <= power_enable;
   SPI_CS_SELECT <= spi_mux WHEN (SPI_PWR_OK = '1') ELSE 'Z';
   SOC_FPGA_DATA_OUT <= serial_data_out WHEN (SPI_PWR_OK = '1') ELSE 'Z';
   ------------------------------------------------------------------------
   -- Clocked Logic
   ------------------------------------------------------------------------
   fpga_ff: PROCESS (CLK_IN, RESET_N) IS
   BEGIN
      IF (RESET_N = '0') THEN  -- asynchronous reset (active low)
         spi_state         <= START_BOOT;
         spi_mux           <= cKGI;
         counter           <= (OTHERS => '0');
         timer             <= '0';
         power_enable      <= '1';
         soc_fpga_clk_del1 <= '0';
         soc_fpga_clk_del2 <= '0';
         soc_fpga_clk_del3 <= '0';
```

```
        sreg              <= (OTHERS => '0');
        reg_state         <= REG_IDLE;
        reg_command       <= '0';
        bit_count         <= (OTHERS => '0');
        reg_addr          <= (OTHERS => '0');
        serial_data_out   <= '1';
        reg_pwr_cycle     <= '0';
        reg_boot_ok           <= '0';
        reg_pwr_cycle_wr_en <= '0';
        reg_boot_ok_wr_en <= '0';
        reg_watchdog_enable <= '1';
        reg_watchdog_wr_en <= '0';
        reg_spi_status_wr_en <= '0';
        watchdog_expired <= '0';
        soc_fpga_data_in_del1 <= '0';
        soc_fpga_data_in_del2 <= '0';
        soc_fpga_data_in_del3 <= '0';
        start_condition <= '0';
    ELSIF (rising_edge(CLK_IN)) THEN  -- rising clock edge
        spi_state         <= next_spi_state;
        spi_mux           <= next_spi_mux;
        counter           <= next_counter;
        timer             <= next_timer;
        power_enable      <= next_power_enable;
        soc_fpga_clk_del1 <= SOC_FPGA_CLK;
        soc_fpga_clk_del2 <= soc_fpga_clk_del1;
        soc_fpga_clk_del3 <= soc_fpga_clk_del2;
        sreg              <= next_sreg;
        reg_state         <= next_reg_state;
        reg_command       <= next_reg_command;
        bit_count         <= next_bit_count;
        reg_addr          <= next_reg_addr;
        serial_data_out   <= next_serial_data_out;
        reg_pwr_cycle     <= next_reg_pwr_cycle;
```

```
      reg_boot_ok       <= next_reg_boot_ok;
      reg_pwr_cycle_wr_en <= next_reg_pwr_cycle_wr_en;
      reg_boot_ok_wr_en <= next_reg_boot_ok_wr_en;
      reg_watchdog_enable <= next_reg_watchdog_enable;
      reg_watchdog_wr_en <= next_reg_watchdog_wr_en;
      reg_spi_status_wr_en <= next_reg_spi_status_wr_en;
      watchdog_expired <= next_watchdog_expired;
      soc_fpga_data_in_del1 <= SOC_FPGA_DATA_IN;
      soc_fpga_data_in_del2 <= soc_fpga_data_in_del1;
      soc_fpga_data_in_del3 <= soc_fpga_data_in_del2;
      start_condition <= next_start_condition;
   END IF;
END PROCESS fpga_ff;
-------------------------------------------------------------------------
-- Watchdog Timer
-------------------------------------------------------------------------
counter_proc: PROCESS (timer_en, counter, reg_watchdog_enable)
BEGIN
   IF ((timer_en = '0') OR (reg_watchdog_enable = '0')) THEN
      next_counter <= (OTHERS => '0');
   ELSE
      next_counter <= counter + '1';
   END IF;
END PROCESS counter_proc;
-------------------------------------------------------------------------
-- Registers
-------------------------------------------------------------------------
-- This is set when the register is accessed by the CPU.
-- It is cleared once the redundant state machine starts
-- the power cycle.
pwr_cycle_reg_proc : PROCESS (rst_reg_pwr_cycle, reg_pwr_cycle_wr_en, sreg,
                              reg_pwr_cycle)
BEGIN
   IF (rst_reg_pwr_cycle = '1') THEN
```

```
            next_reg_pwr_cycle <= '0';
         ELSIF (reg_pwr_cycle_wr_en = '1') THEN
            next_reg_pwr_cycle <= sreg(0);
         ELSE
            next_reg_pwr_cycle <= reg_pwr_cycle;
         END IF;
      END PROCESS pwr_cycle_reg_proc;
   -- This can be set and cleared by the CPU. It is reset
      -- by the redundant state machine whenever a rebooting
      -- the flash.
      boot_ok_reg_proc : PROCESS (rst_reg_boot_ok, reg_boot_ok_wr_en,
                                  sreg, reg_boot_ok)
      BEGIN
         IF (rst_reg_boot_ok = '1') THEN
            next_reg_boot_ok <= '0';
         ELSIF (reg_boot_ok_wr_en = '1') THEN
            next_reg_boot_ok <= sreg(0);
         ELSE
            next_reg_boot_ok <= reg_boot_ok;
         END IF;
      END PROCESS boot_ok_reg_proc;
      -- This is enabled and disabled by the CPU.  It is reset
      -- by the redundant state machine whenever a rebooting
      -- the flash.
      watchdog_enable_reg_proc : PROCESS (rst_reg_watchdog, reg_watchdog_wr_en,
                                  sreg, reg_watchdog_enable)
      BEGIN
         IF (rst_reg_watchdog = '1') THEN
            next_reg_watchdog_enable <= '1';
         ELSIF (reg_watchdog_wr_en = '1') THEN
            next_reg_watchdog_enable <= sreg(0);
         ELSE
            next_reg_watchdog_enable <= reg_watchdog_enable;
         END IF;
```

```
END PROCESS watchdog_enable_reg_proc;
-------------------------------------------------------------------------
-- Redundant SPI State Machine
-------------------------------------------------------------------------
spi_state_proc: PROCESS (spi_state, spi_mux, timer, watchdog_expired,
                         SPI_PWR_OK, REDUNDANT_SPI_ENABLE, reg_boot_ok,
                         counter, RESET_COMPLETE, reg_pwr_cycle,
                         reg_spi_status_wr_en, sreg
                          )
BEGIN
   -- Outputs
   timer_en <= '0';
   next_spi_state <= spi_state;
   next_spi_mux <= spi_mux;
   next_timer <= timer;
   next_power_enable <= '1';
   rst_reg_pwr_cycle <= '0';
   rst_reg_boot_ok <= '0';
   rst_reg_watchdog <= '0';
   next_watchdog_expired <= watchdog_expired;
   CASE spi_state IS
   -------------------------------------------------------------------------
   --  * Wait for the RCC to be powered.
   --  * Examine Jumper and deterime which Flash to use
   -------------------------------------------------------------------------
      WHEN START_BOOT =>
          rst_reg_boot_ok <= '1';
          rst_reg_watchdog <= '1';
          next_watchdog_expired <= '0';
          IF (SPI_PWR_OK = '1') THEN
            IF (REDUNDANT_SPI_ENABLE = '0') THEN
               next_spi_mux <= cKGI;
               next_spi_state <= SPI_IDLE;
            ELSE
```

```
                next_spi_mux <= cCI;
                next_spi_state <= WAIT_BOOT;
             END IF;
          END IF;
-------------------------------------------------------------------------
--  A timer is running during this state.
--  * if it expires, then switch to KGI
--  * if boot ok, then go to IDLE
-------------------------------------------------------------------------
   WHEN WAIT_BOOT =>
      -- Timer Expired
      IF (timer = '1') THEN
         next_timer <= '0';
         timer_en <= '0';
         next_watchdog_expired <= '1';
         next_spi_mux <= cKGI;
         next_spi_state <= KGI_BOOT;
      -- BOOK OK
      ELSIF (reg_boot_ok = '1') THEN
         next_timer <= '0';
         timer_en <= '0';
         next_spi_state <= SPI_IDLE;
      ELSE
          next_timer <= counter(31) AND counter(28); -- 193s
--           next_timer <= counter(19); -- simulation
         timer_en <= '1';
      END IF;
-------------------------------------------------------------------------
--  Start power cycle.
-------------------------------------------------------------------------
   WHEN KGI_BOOT =>
      IF ((SPI_PWR_OK = '0') AND (timer = '1')) THEN
         next_power_enable <= '1';
         next_timer <= '0';
```

```
                timer_en <= '0';
                next_spi_state <= KGI_BOOT2;
             ELSE
                next_power_enable <= '0';
                IF (timer = '0') THEN
                   timer_en <= '1';
                END IF;
                next_timer <= counter(22); -- 335ms
--              next_timer <= counter(10); -- simulation
             END IF;
          ------------------------------------------------------------------
          --  Wait for power to be good
          ------------------------------------------------------------------
            WHEN KGI_BOOT2 =>
              IF (SPI_PWR_OK = '1') THEN
               next_spi_state <= KGI_BOOT3;
              END IF;
          ------------------------------------------------------------------
          --  Wait for book ok signal
          ------------------------------------------------------------------
            WHEN KGI_BOOT3 =>
              IF (reg_boot_ok = '1') THEN
                 next_spi_state <= SPI_IDLE;
              END IF;
          ------------------------------------------------------------------
          --  Stay in idle until
          --  * System power cycles
          --  * System resets
          --  * CPU requests power cycle
          ------------------------------------------------------------------
            WHEN SPI_IDLE =>
              IF (SPI_PWR_OK = '0') THEN
                 next_spi_state <= START_BOOT;
              ELSIF ((RESET_COMPLETE = '0') AND (REDUNDANT_SPI_ENABLE = '1')) THEN
```

```
                rst_reg_boot_ok <= '1';
                rst_reg_watchdog <= '1';
                next_spi_state <= WAIT_BOOT;
             ELSIF (reg_pwr_cycle = '1') THEN
                next_spi_state <= PWR_CYCLE;
             END IF;
             -- In the IDLE state, the CPU can control the SPI MUX
             IF (reg_spi_status_wr_en = '1') THEN
                next_spi_mux <= sreg(1);
             END IF;
          ------------------------------------------------------------------------
          --  Pull down power for 200ms
          ------------------------------------------------------------------------
              WHEN PWR_CYCLE =>
                IF ((SPI_PWR_OK = '0') AND (timer = '1')) THEN
                   next_power_enable <= '1';
                   next_timer <= '0';
                   timer_en <= '0';
                   rst_reg_pwr_cycle <= '1';
                   next_spi_state <= START_BOOT;
                ELSE
                   next_power_enable <= '0';
                   next_timer <= counter(22); -- 335ms
--                   next_timer <= counter(10); -- simulation
                   timer_en <= '1';
                END IF;
          END CASE;
       END PROCESS spi_state_proc;
       ------------------------------------------------------------------------
       -- SOC and FPGA Clock Edges
       ------------------------------------------------------------------------
       soc_fpga_clk_rising <= soc_fpga_clk_del2 AND (NOT soc_fpga_clk_del3);
       soc_fpga_clk_falling <= (NOT soc_fpga_clk_del2) AND soc_fpga_clk_del3;
       -- When data goes low while clock is high, then reset state machine to start
```

```
position

   next_start_condition <= (NOT soc_fpga_data_in_del2) AND soc_fpga_data_in_del3
AND soc_fpga_clk_del2;

   read_address <= sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN;

   ---------------------------------------------------------------------------

   -- Serial Protocol State Machine

   ---------------------------------------------------------------------------

   reg_state_proc: PROCESS (reg_state, reg_command, bit_count, reg_addr,
                        serial_data_out, soc_fpga_clk_rising, sreg,
                        SOC_FPGA_DATA_IN, watchdog_expired, spi_mux,
                        REDUNDANT_SPI_ENABLE, reg_watchdog_enable,
                        reg_boot_ok, reg_pwr_cycle, soc_fpga_clk_falling,
                        read_address, start_condition)

   BEGIN

      next_reg_state <= reg_state;

      next_reg_command <= reg_command;

      next_bit_count <= bit_count;

      next_reg_addr <= reg_addr;

      next_serial_data_out <= serial_data_out;

      next_reg_pwr_cycle_wr_en <= '0';

      next_reg_boot_ok_wr_en <= '0';

      next_reg_watchdog_wr_en <= '0';

      next_reg_spi_status_wr_en <= '0';

      next_sreg <= sreg;

      CASE reg_state IS

         WHEN REG_IDLE =>

            next_bit_count <= (OTHERS => '0');

            IF (start_condition = '1') THEN

               next_sreg <= (OTHERS => '0');

            ELSIF (soc_fpga_clk_rising = '1') THEN

               IF ((sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN) = X"A6") THEN  -- Write

                  next_reg_command <= cWRITE;

                  next_reg_state <= REG_ADDRESS;

               ELSIF ((sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN) = X"A7") THEN -- Read
```

```
                    next_reg_command <= cREAD;
                    next_reg_state <= REG_ADDRESS;
                 ELSE
                  next_sreg <= sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN;  -- Shift data in
                 END IF;
              END IF;
           WHEN REG_ADDRESS =>
              IF (start_condition = '1') THEN
                 next_sreg <= (OTHERS => '0');
                 next_reg_state <= REG_IDLE;
              ELSIF (soc_fpga_clk_rising = '1') THEN
                 IF (bit_count = X"07") THEN
                    next_bit_count <= (OTHERS => '0');
          next_reg_addr <= (sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN);
                    IF (reg_command = cREAD) THEN
                       next_reg_state <= REG_RD_DATA;
                       CASE (read_address) IS
                          WHEN X"00" => next_sreg <= ("00000" & watchdog_expired &
spi_mux & REDUNDANT_SPI_ENABLE);
                        WHEN X"01" => next_sreg <= ("0000000" & reg_watchdog_enable);
                          WHEN X"02" => next_sreg <= ("0000000" & reg_boot_ok);
                          WHEN X"03" => next_sreg <= ("0000000" & reg_pwr_cycle);
                          WHEN OTHERS => next_sreg <= (X"FF");
                     END CASE;
                    ELSE
                       next_reg_state <= REG_WR_DATA;
                    END IF;
                 ELSE
                  next_sreg <= sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN;  -- Shift data in
                    next_bit_count <= bit_count + '1';
                 END IF;
              END IF;
           WHEN REG_RD_DATA =>
              IF (start_condition = '1') THEN
```

```
                  next_sreg <= (OTHERS => '0');
                  next_reg_state <= REG_IDLE;
               ELSIF ((bit_count = X"08") AND  (soc_fpga_clk_rising = '1')) THEN
                  next_bit_count <= (OTHERS => '0');
                  next_reg_state <= REG_IDLE;
                  next_serial_data_out <= '1';
               ELSIF (soc_fpga_clk_falling = '1') THEN
                  next_serial_data_out <= sreg(7);
                  next_sreg <= sreg(6 DOWNTO 0) & '1';
                  next_bit_count <= bit_count + '1';
               END IF;
            WHEN REG_WR_DATA =>
               IF (start_condition = '1') THEN
                  next_sreg <= (OTHERS => '0');
                  next_reg_state <= REG_IDLE;
               ELSIF (soc_fpga_clk_rising = '1') THEN
                  IF (bit_count = X"07") THEN
                     next_bit_count <= (OTHERS => '0');
                    next_sreg <= sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN;  -- Shift data in
                     next_reg_state <= REG_IDLE;
                     CASE (reg_addr) IS
                    WHEN X"00" => next_reg_spi_status_wr_en <= '1';
                       WHEN X"01" => next_reg_watchdog_wr_en <= '1';
                      WHEN X"02" => next_reg_boot_ok_wr_en <= '1';
                       WHEN X"03" => next_reg_pwr_cycle_wr_en <= '1';
                       WHEN OTHERS =>
                     END CASE;
                  ELSE
                    next_sreg <= sreg(6 DOWNTO 0) & SOC_FPGA_DATA_IN;  -- Shift data in
                     next_bit_count <= bit_count + '1';
                  END IF;
               END IF;
         END CASE;
      END PROCESS reg_state_proc;
```


```
END ARCHITECTURE rtl;

-------------------------------------------------------------------------------/
/

-------------------------------------------------------------------------------/
/

LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_unsigned.ALL;

ENTITY tb_rcc_spi IS

END tb_rcc_spi;

ARCHITECTURE tb_rcc_spi_behav OF tb_rcc_spi IS


COMPONENT INTEL_RCC_SPI

   PORT(

   CLK_IN                        : IN std_logic;

   RESET_N                       : IN std_logic;

   SOC_FPGA_CLK                  : IN std_logic;

   SOC_FPGA_DATA_IN              : IN std_logic;

   SOC_FPGA_DATA_OUT             : OUT std_logic;

   SPI_PWR_OK                    : IN std_logic;

   REDUNDANT_SPI_ENABLE          : IN std_logic;

   RESET_COMPLETE                : IN std_logic;

   SPI_CS_SELECT                 : OUT std_logic;

   PWR_ENABLE                    : OUT std_logic

 );

END COMPONENT;

   SIGNAL CLK_IN                         : std_logic;

   SIGNAL RESET_N                        : std_logic;

   SIGNAL SOC_FPGA_CLK                   : std_logic;

   SIGNAL SOC_FPGA_DATA_IN               : std_logic;

   SIGNAL SOC_FPGA_DATA_OUT              : std_logic;

   SIGNAL SPI_PWR_OK                     : std_logic;

   SIGNAL REDUNDANT_SPI_ENABLE           : std_logic;

   SIGNAL RESET_COMPLETE                 : std_logic;

   SIGNAL SPI_CS_SELECT                  : std_logic;
```

```
   SIGNAL PWR_ENABLE                    : std_logic;
BEGIN
   rcc_spi_inst : INTEL_RCC_SPI PORT MAP (
      CLK_IN                        =>  CLK_IN,
      RESET_N                       =>  RESET_N,
      SOC_FPGA_CLK                  =>  SOC_FPGA_CLK,
      SOC_FPGA_DATA_IN              =>  SOC_FPGA_DATA_IN,
      SOC_FPGA_DATA_OUT             =>  SOC_FPGA_DATA_OUT,
      SPI_PWR_OK                    =>  SPI_PWR_OK,
      REDUNDANT_SPI_ENABLE          =>  REDUNDANT_SPI_ENABLE,
      RESET_COMPLETE                =>  RESET_COMPLETE,
      SPI_CS_SELECT                 =>  SPI_CS_SELECT,
      PWR_ENABLE                    =>  PWR_ENABLE
      );
   ------------------------------------------------------------
   -- Clock - 12.5MHz
   ------------------------------------------------------------
   CLK_PROC : PROCESS
   BEGIN
     CLK_IN <= '0';
     WAIT FOR 40 ns;
     CLK_IN <= '1';
     WAIT FOR 40 ns;
   END PROCESS CLK_PROC;
   ------------------------------------------------------------
   -- Main TB Process
   ------------------------------------------------------------
   TB_MAIN_PROC : PROCESS
    procedure SERIAL_WRITE (
      constant ADDRESS: in std_logic_vector(7 DOWNTO 0);
      constant DATA: in std_logic_vector (7 DOWNTO 0)) is
    begin
      -- Start Condition
      SOC_FPGA_CLK <= '1';
```


```
SOC_FPGA_DATA_IN <= '1';

WAIT FOR 5 us;

SOC_FPGA_DATA_IN <= '0';

WAIT FOR 5 us;

-- Opcode

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '0';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '0';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '0';

WAIT FOR 5 us;
```

```
SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '0';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

-- Address

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(7);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(6);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';
```


```
WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(5);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(4);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(3);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(2);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(1);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(0);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';
```

```
-- Data
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= DATA(7);
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= DATA(6);
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= DATA(5);
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= DATA(4);
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= DATA(3);
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
```


```
  SOC_FPGA_DATA_IN <= DATA(2);
  WAIT FOR 5 us;
  SOC_FPGA_CLK <= '1';
  WAIT FOR 5 us;
  SOC_FPGA_CLK <= '0';
  WAIT FOR 1 us;
  SOC_FPGA_DATA_IN <= DATA(1);
  WAIT FOR 5 us;
  SOC_FPGA_CLK <= '1';
  WAIT FOR 5 us;
  SOC_FPGA_CLK <= '0';
  WAIT FOR 1 us;
  SOC_FPGA_DATA_IN <= DATA(0);
  WAIT FOR 5 us;
  SOC_FPGA_CLK <= '1';
  WAIT FOR 10 us;
end;
procedure SERIAL_READ (
  constant ADDRESS: in std_logic_vector(7 DOWNTO 0)) is
begin
  -- Start Condition
  SOC_FPGA_CLK <= '1';
  SOC_FPGA_DATA_IN <= '1';
  WAIT FOR 5 us;
  SOC_FPGA_DATA_IN <= '0';
  WAIT FOR 5 us;
  -- Opcode
  WAIT FOR 5 us;
  SOC_FPGA_CLK <= '0';
  WAIT FOR 1 us;
  SOC_FPGA_DATA_IN <= '1';
  WAIT FOR 5 us;
  SOC_FPGA_CLK <= '1';
  WAIT FOR 5 us;
```

```
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= '0';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= '0';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= '0';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '1';
WAIT FOR 5 us;
SOC_FPGA_CLK <= '0';
WAIT FOR 1 us;
SOC_FPGA_DATA_IN <= '1';
WAIT FOR 5 us;
```


```
SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

-- Address

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(7);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(6);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(5);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(4);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';
```

```
WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(3);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(2);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(1);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 1 us;

SOC_FPGA_DATA_IN <= ADDRESS(0);

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

-- Data

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '1';

WAIT FOR 5 us;

SOC_FPGA_CLK <= '0';

WAIT FOR 5 us;
```

```
    SOC_FPGA_CLK <= '1';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '0';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '1';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '0';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '1';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '0';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '1';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '0';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '1';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '0';
    WAIT FOR 5 us;
    SOC_FPGA_CLK <= '1';
    WAIT FOR 10 us;
 end;
BEGIN
    -- Initialize Inputs
    RESET_N <= '0';
    SOC_FPGA_CLK <= '1';
    SOC_FPGA_DATA_IN <= '1';
    SPI_PWR_OK <= '0';
    REDUNDANT_SPI_ENABLE <= '1';
    RESET_COMPLETE <= '0';
    WAIT FOR 1 ms;
    -- Resets and power good
    RESET_N <= '1';
```

```
        WAIT FOR 5 ms;
        SPI_PWR_OK <= '1';
        WAIT FOR 5 ms;
        RESET_COMPLETE <= '1';
        WAIT FOR 5 ms;
        -- Disable Timer
        Wait for 50 ms;
        SERIAL_WRITE (X"01", X"00");
        Wait for 5000 ms;
        -- Let the Watchdog Expire
        WAIT UNTIL PWR_ENABLE = '0';
        SPI_PWR_OK <= '0';
        RESET_COMPLETE <= '0';
        WAIT UNTIL PWR_ENABLE = '1';
        WAIT FOR 1 ms;
        SPI_PWR_OK <= '1';
        WAIT FOR 1 ms;
        RESET_COMPLETE <= '1';
        WAIT FOR 1 ms;
        -- Set Boot OK
        SERIAL_WRITE (X"02", X"01");
        WAIT FOR 1 ms;
        -- Read Status (should be on KGI and expired)
        SERIAL_READ (X"00");
        WAIT FOR 100 ms;
    END PROCESS TB_MAIN_PROC;
END tb_rcc_spi_behav;
```

§ §

3.0 Summary

Implementing the redundant image support incurs expense but implementing the feature can reduce system downtime. Considering the up-front system requirements are important and how these requirements are achieved.

Using a programmable logic device to implement this functionality protects against failures and allows for flexibility during development, testing, and production. Further protection and design flexibility can be achieved by adding additional signals (such as GPIO) to the PLD. If maintaining a power rail to the PLD presents a design challenge, various non-volatile solutions exist to capture the state in the event of a power down.

§ §