Si8250 Real-Time Kernel Overview

1. Introduction

The Si8250 is a fully-programmable digital power controller/manager that is useful over a wide range of end applications. Behavior of this controller is determined by firmware; so, each end-system requires an application program containing the control and protection algorithms. Design of such an application program could be a significant user development if written from start-to-finish. The Si825xK2.0 (*K2.0*) is a royalty-free application software kernel for the Si825x family of digital power controllers that greatly reduces application program development time, effort, and engineering risk. The K2.0 installed kernel (Figure 1) supports isolated and non-isolated Si8250-based systems and includes verified algorithms that perform analog data acquisition, system initialization, soft-start, steady state regulation, fault detection/recovery, network interface communications, and system shutdown supporting function library and other application-specific routines. The user needs only to port this kernel to the end application and add specialized (user-proprietary) source-code-level functions. Porting the kernel involves editing the source code files using the Application Builder tools included in the Si8250DK development kit. It is also available for download at www.silabs.com. These tools simplify kernel porting through a series of graphical user interfaces (although it is strongly recommended the user have a background in C-Language programming). The integrated development environment (included with the tool set) allows the user to create and edit source-level C-code.

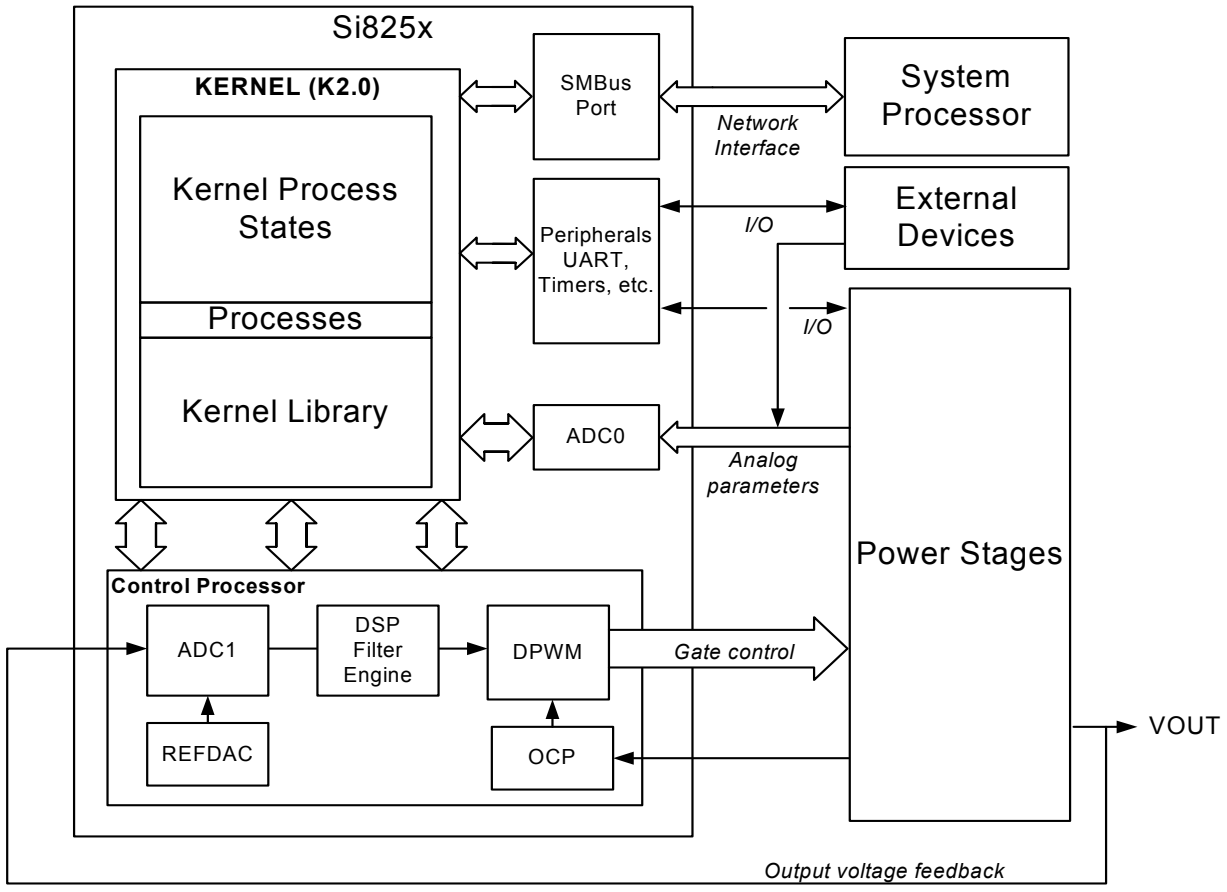

Figure 1. Si825x System with K2.0 Installed

This document is useful as both a tutorial and a reference. A system-level discussion is presented in the next section, followed by a top-level kernel description and description of each kernel process state. "Appendix A—Library Functions" on page 44 is a summary of kernel library functions, and "Appendix B—Header File" on page 50 is a description of the kernel set-up parameters contained in the header (.h) file.

Note: Throughout this document, the names of application software variables and program blocks are in *italicized* print. For example: *Kernel_Regulation_Process*, *mFilterProcess(TEMP)*, etc.

1.1. Differences from Kernel Version 1.0

- K2.0 uses a cooperative architecture that readily accommodates the addition of user-proprietary software. All user-added code can reside in a single file (main.c), simplifying code additions and system debug.
- Synchronous design—All kernel processes are synchronized by a programmable hardware timer for deterministic operation.
- Optimized library functions make K2.0 source code easy to understand and simplify user code additions.
- Added robustness—All code is non-blocking (no looping instructions) and is resistant to “infinite looping” regardless of system status. Added safeguards ensure more robust system operation.
- Newly-supported system functions—Efficiency optimization using dynamic dead time control, half-bridge input node dc balance, V_{IN} feed-forward, additional network interface commands, input and output current measurement, dead time control for improved efficiency, and advanced nonlinear control algorithms for improved transient response.
- Fast, software-based ADC0 multiplexer and fault detection management (with software-filtered, 16-bit variables) have been implemented for superior noise rejection and greater fault detection programmability via the network interface.
- K2.0 typically occupies approximately 20% less code space than V1.0 and executes up to 50% faster, depending on system configuration. Many system applications will operate at a lower CPU clock frequency as a result. K2.0 can compile to code sizes as small as 2 kB for systems without network interface and 10 kB for systems with network interface.

1.2. System States of a Typical Power Systems

Typical power system states and the corresponding support provided by K2.0 are summarized in Table 1. A state diagram showing system operation is shown in Figure 2.

Table 1. Typical Power System States and K2.0 Support Summary

End System Operating State	Summary of Functional Support Provided by K2.0
Initialization	Initializes all Si825x hardware configuration registers: V_{DD} monitor, MCU watchdog timer, on-board oscillator, processor I/O port type and functions, 12-bit ADC, kernel scheduler (Timer2), UART (for isolated applications), ENABLE input, REFDAC, communications watchdog timer, DPWM timing set, DSP filter engine coefficients, CPU interrupts, peak current detector, and hardware OCP. Kernel software parameter initialization clears internal status and control flags and loads system parameters including protection parameters and system set points (e.g., V_{OUT}).
Soft-Start	Validates startup permissives: tests VIN thresholds for voltage above UVLO level, checks internal and external enable states, tests for faults. Initiates soft-start; adjusts for pre-bias start (if any), preloads software filters (ensures smooth departure from 0 V); ramps REFDAC on a specified trajectory in closed-loop mode with protection enabled. Updates internal status and control flags. Manages network interface communications.
Transition	Manages transition from soft-start to steady-state regulation without V_{OUT} overshoot. Updates internal status and control flags.
Steady-State Regulation	Executes protection fault tasks, manages network interface communications. Executes low-bandwidth loop optimization: dead time control (improved efficiency), nonlinear control (faster transient response), dc balance (half-bridge input node balance at $V_{IN}/2$), V_{IN} feed-forward. Frame skipping at low loads. Updates internal status and control flags.
Fault Recovery	Takes action in response to a detected fault. Possible responses include ignoring the fault, correcting the fault, logging the fault, or proceeding to shutdown.
System Shut Down	Shuts system off: soft or hard stop, asynchronous stop or stopping on a switching frame boundary, all or some DPWM outputs in parking state (excluded outputs continue to follow DPWM timing). Attempt restart for a specified number of tries, or latch off (requires input voltage to be cycled to restart).

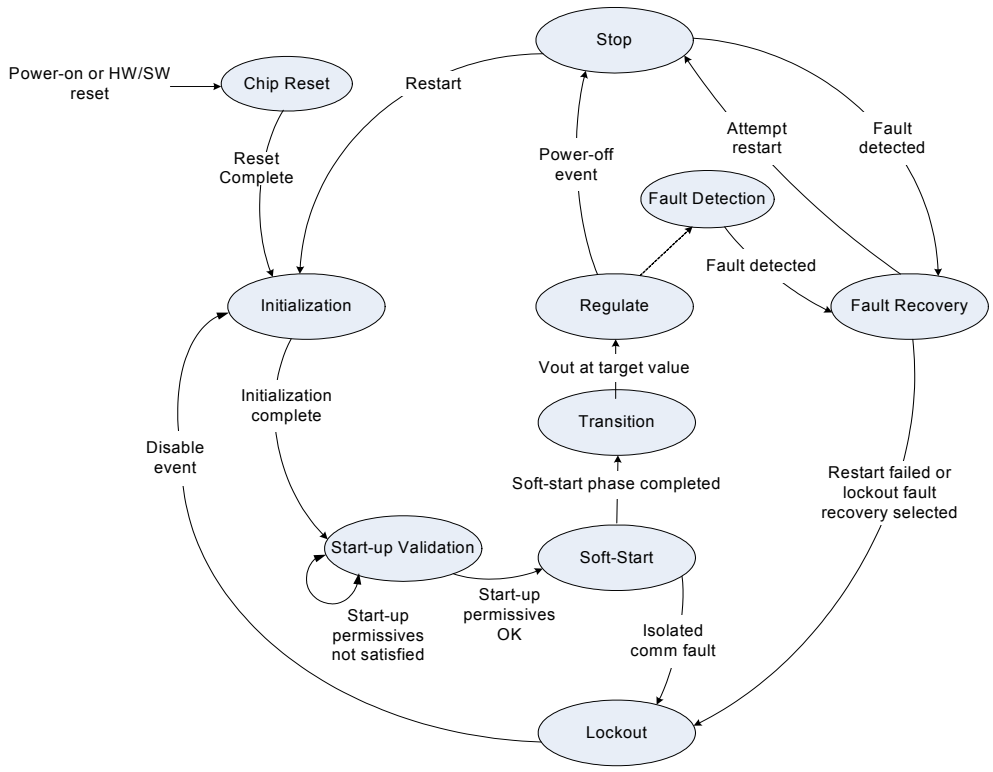

Figure 2. Top-Level System State Diagram

Referring to Figure 2 and assuming a Si825x-based power system with the K2.0 kernel application software installed, the initialization state is entered immediately after the Si825x hardware reset. In this state, on-chip hardware peripherals are configured, and all kernel software parameters (status flag states, variable values, etc.) are initialized.

After initialization, system startup permissives are evaluated in the startup validation state. For example, input voltage is measured, filtered to remove spurious noise, and compared to the under-voltage lockout threshold. The enable input (if used) is checked for the ON state; fault status is evaluated, and so on. If all conditions are met, kernel execution moves to the soft-start state where the output voltage is ramped to a target value under closed-loop control and with system fault protection fully enabled.

When the output voltage is just below the minimum specified value at the end of soft-start, the kernel execution enters the transition state. The transition state gently brings the output voltage into the regulation band without overshoot, after which program execution vectors to the regulation state. During regulation, all previously activated operations (network interface, fault detection/recovery, etc.) continue to be executed, and additional low-bandwidth loop optimization operations (such as nonlinear control for faster transient response and dead time adjustment for greater efficiency) are enabled. The regulation state is exited on a stop event, such as turning the ENABLE input off or the presence of a fault condition. The next operating state is determined by the nature of the stop event and the fault recovery rules for the system. In some cases, an automatic restart may be attempted, and in other cases, the system is locked-out and requires input power to be cycled to restart the system.

2. Kernel Description

2.1. Structure

The K2.0 kernel is optimized for small size, fast execution, ease of user-modification, and robust operation. To meet these goals, a cooperative programming structure based on a hierarchical state machine is used (Figure 3).

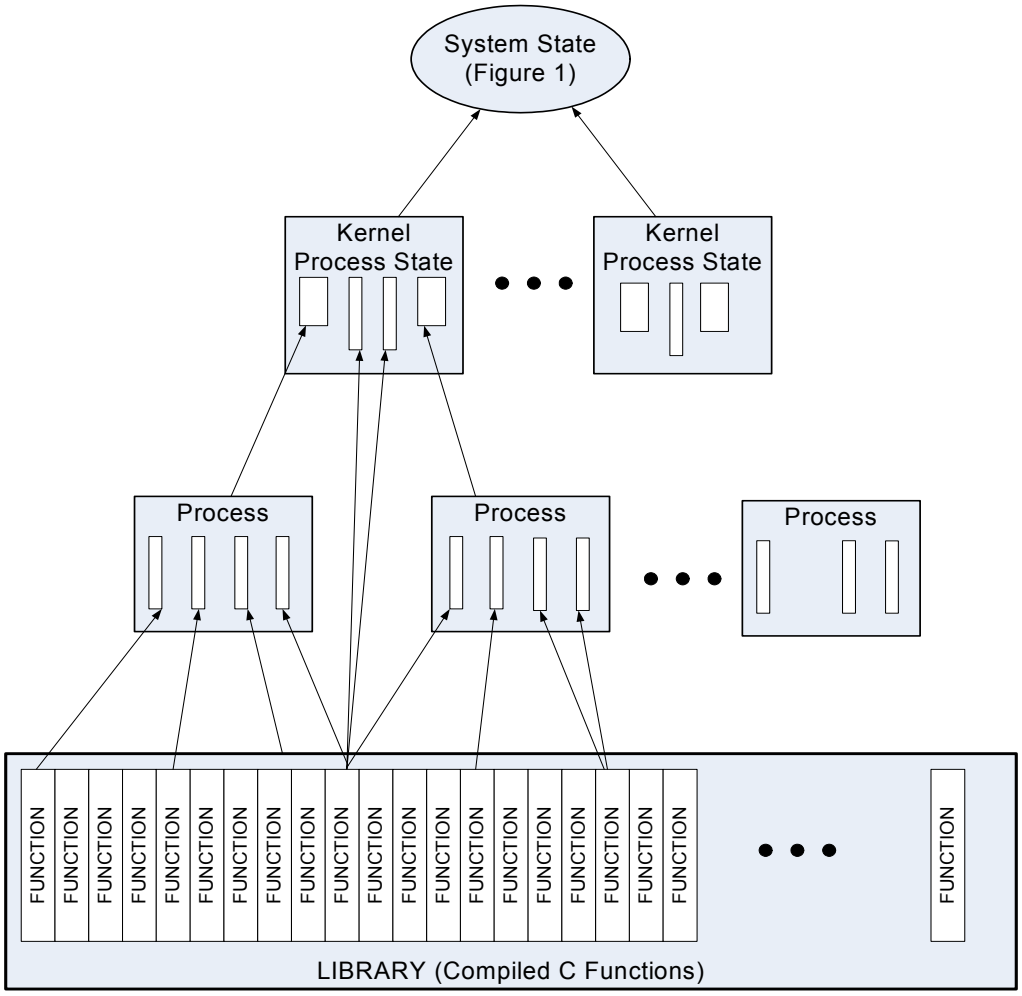

Figure 3. Kernel Software Structure

Low-level C-language *functions* (analogous to "*subroutines*" in Basic) are stored in the kernel library. These functions perform elementary operations, such as testing bit states, filtering analog data, and so on. Each function has a "fall-through" code architecture (also referred to as "non-blocking code") that does not make use of conditional "looping" code. As a result, each function executes quickly and without vulnerability to getting stuck in an "infinite loop" when unforeseen circumstances occur. Combinations of library functions are connected with "glue code" to form individual power system processes, such as the input voltage feed-forward process. Like the library functions from which they are built, these base processes execute quickly, consume little code space, and exhibit robust operation.

Functions and processes are used to create large-scale system code building blocks called kernel process states, such as the *Kernel_Regulation_State.* The system operating states shown in Figure 2 are implemented by sequentially executing one or more individual kernel process states. When invoked, each kernel process state executes its task, then updates software status flags for future reference and specifies the next kernel process state to be executed based upon data and results gathered in the current kernel process state.

2.1.1. K2.0 File Organization

The K2.0 kernel file organization is shown in Figure 4. The “main.c” program is very simple; it initializes the kernel and calls the kernel process (compiled from the kernel program file). The kernel executes and returns to main.c where proprietary user-generated code functions may be placed. This code might include control of external devices, such as fans, and displays or may contain special control or other proprietary algorithms. The process repeats upon exit from any user code in Main.

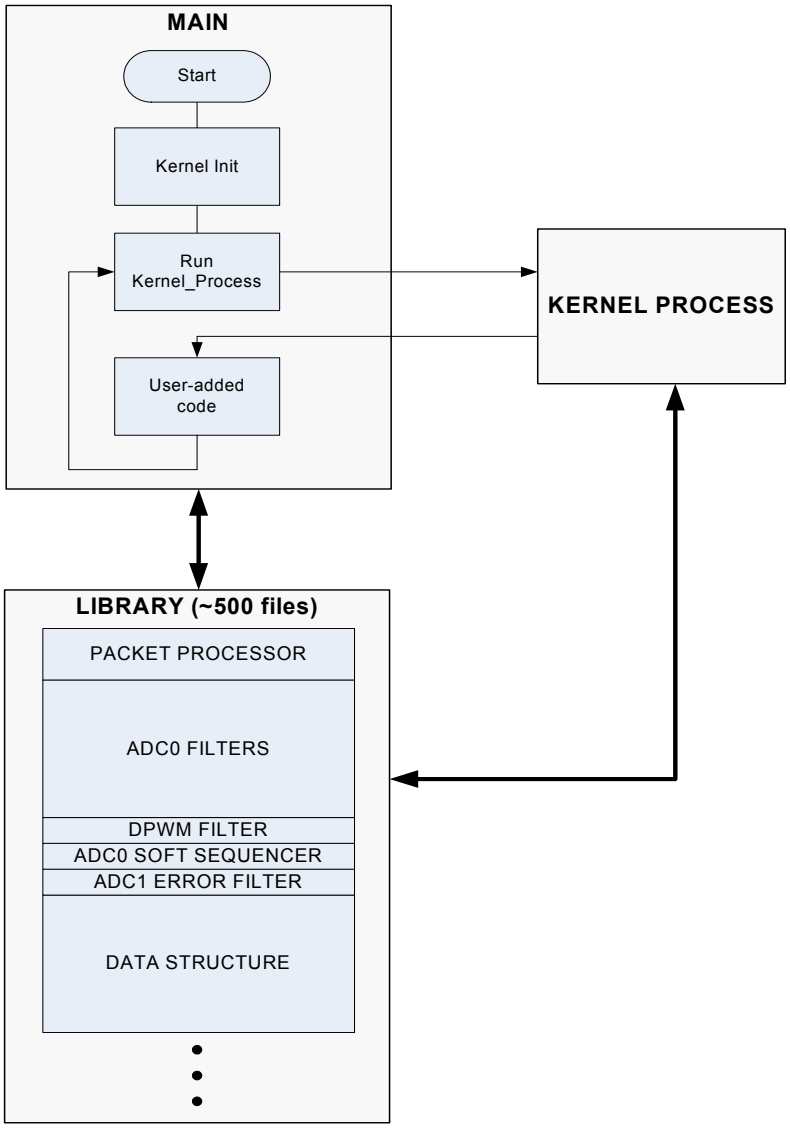

Figure 4. Kernel File Organization

Each C-Language library function is contained in its own file (there are over 500 function files in the library). Individual files ensure that each function is completely optimized with no unrelated code for the smallest program memory footprint possible. Example library functions include the isolated communications packet processor, individual filters for each ADC0 input, the ADC0 soft-sequencer, DPWM averaging filter, ADC1 error filter, and others. Library functions may be called from any program making them usable in main.c, the kernel, or any other project file.

A summary of the kernel process states and their functional descriptions appears in Table 2. Each kernel process state is detailed in a later section of this document.

Table 2. Kernel Process State Definitions

State Name	Function	Operations
Kernel_Process	Dispatcher	Passes program control to the appropriate process when called. As each process completes execution, it specifies the next process to be called based on its own operating results.
Kernel_Init	Chip HW initialization	Initializes all of the hardware and software settings onboard the Si8250 in preparation for startup.
Kernel_High_Level_Init	Power converter initialization	This process is essentially a “converter reset”. That is, various power converter-specific functions and parameters are initialized.
Kernel_Validation_Init_Process	Initializes startup variables	Prepares the system for “validation”, which is system parameter testing to determine if it is permissible to initiate soft-start.
Kernel_Validation_Process	Determines if startup is permissible	Tests system parameters to determine if it is permissible to initiate soft-start.
Kernel_Soft_Start_Init_Process	Soft-start initialization	Prepares system for soft-start.
Kernel_Soft_Start_WarmUp_Process	Manages startup delay (t_{ON}).	This process provides an adjustable time delay prior to the beginning of the soft-start ramp (t_{ON}) per network interface specifications.
Kernel_Soft_Start_Process	Executes soft-start	Executes soft-start output voltage ramp.
Kernel_Regulation_Init_Process	Initialization for steady-state regulation	This process prepares the system to enter steady-state regulation.
Kernel_Transition_Process	Completes soft-start ramp without overshoot	Gently completes transition of soft-start to nominal steady-state output voltage setting without overshoot.
Kernel_Regulation_Process	Manages steady-state regulation	Performs steady-state regulation, control optimization, and system maintenance.
Kernel_Stop_Init_Process	Selects stop mode	Performs a hard stop on fault; otherwise, vectors to Kernel_Soft_Stop_Init for a soft-stop.
Kernal_Soft_Stop_Process	Manages soft-stop	Maintains a controlled, linear, declining output voltage ramp during converter turn-off.
Kernel_Disable_Process	Halts power conversion, decides next operating state.	Terminates power system operation; checks for faults, and updates status flags accordingly.
Kernel_Lockout_Process	Manages background tasks during converter shutdown	Maintains key system functions, such as monitoring enable signals for change and servicing network interface communications, while the system power converter is shut down.
Kernel_Fault_Recovery_Init_Process	Initializes fault recovery process	Manages fault restart counters and fault flags.

Table 2. Kernel Process State Definitions (Continued)

State Name	Function	Operations
Kernel_Fault_Recovery_Process	Resolves faults.	Tests for under/over voltage on V_{OUT}, V_{IN}, under/over temperature, on-time fault, and overcurrent. Directs execution to next kernel process state based on fault type.
Kernel_State_Control_Process	Directs program execution to fault recovery state (attempt restart), or to an off state.	Executes the fault action; for example, system shut-down and/or process state change.
Kernel_Threshold_Process	Tests for fault thresholds.	Tests specified variable against stored warning and fault thresholds. Updates appropriate status flags accordingly.

Table 3 shows the order in which kernel process states are executed. The kernel process states in the first column typically move to the process states in the second column unless an “event” (interrupt) is present, in which case, a switch is made to the process state listed in the last column on the right. For example, the *Kernel_Process* is always followed by the *Kernel_Init_Process*.

Table 3. Process States

Kernel Process State	Next Kernel Process State	Event	Next Kernel Process State
Kernel_Process	Kernel_Init		
Kernel_Init	Kernel_High_Level_Init		
Kernel_High_Level_Init_Process	Kernel_Validation_Init_Process		
Kernel_Validation_Init_Process	Kernel_Validation_Process		
Kernel_Validation_Process	Kernel_Soft_Start_Init_Process	Isolated communi-cations fault	Kernel_Lockout_Process
Kernel_Soft_Start_Init_Process	Kernel_Soft_Start_WarmUp_Process		
Kernel_Soft_Start_WarmUp_Process	Kernel_State_Control_Process		
Kernel_Soft_Start_Process	Kernel_Regulation_Init_Process	On-time fault	Kernel_State_Control_ Process
Kernel_Regulation_Init_Process	Kernel_Transition_Process		
Kernel_Transition_Process	Kernel_State_Control_Process		
Kernel_Regulation_Process	Kernel_State_Control_Process		
Kernel_Stop_Init_Process	Kernel_Disabled_Process	Fault present	Kernel_Fault_Recovery_ Init_Process
Kernal_Soft_Stop_Process	Kernel_Disabled_Process		
Kernel_Disabled_Process	Kernel_Lockout_Process	ENABLE input OFF and not masked	Kernel_High_Level_Init
		Isolated communi-cation fault	Kernel_Lockout_Process
Kernel_Lockout_Process	Kernel_Fault_Recovery_Init_Process	ENABLE input OFF and not masked	Kernel_Disabled_ Process
		Isolated communi-cation fault	Kernel_Lockout_Process
Kernel_Fault_Recovery_Init_Process	Kernel_Fault_Recovery_Process		

Table 3. Process States (Continued)

Kernel Process State	Next Kernel Process State	Event	Next Kernel Process State
Kernel_Fault_Recovery_Process		V_{OUT} under/over voltage fault	Kernel_Lockout_Process
		V_{IN} under/over voltage fault	
		Over/under temperature fault	
		On-time fault	
		Input over current	
		Fault condition removed—enable OFF and not masked	Kernel_Stop_Init_Process
		Isolated communications fault	Kernel_Lockout_Process
Kernel_State_Control_Process	If no events are present, return to previous process.	Isolated communications fault	Kernel_Lockout_Process
		Peak current event (ICYC)	Kernel_Fault_Recovery_Process
		Input over current fault	
		V_{IN} under/over voltage fault	
		Over/under temperature fault	
		V_{OUT} under/over voltage fault	
		Undervoltage lockout fault	Kernel_Stop_Init_Process
		ENABLE input OFF and not masked	

However, the *Kernel_Validation_Process* moves to the *Kernel_Soft_Start_Init_Process* unless an isolated communications fault is present, in which case, execution moves to the *Kernel_Lockout_Process*. From this state table, one can see that all events are initiated by a fault, and, as a result, program execution is vectored to either a lockout, recovery, or stop process state.

All files contained in the K2.0 kernel are summarized in Table 4.

Table 4. Si8250K2.0 Kernel Files

File Name	File Type	Description
DPSK_Balance.c	Process	Half bridge transformer dc balance algorithm.
DPSK_Balance.h	Header file	Header file for half bridge transformer dc balance.
DPSK_Deadtime.c	Process	Dead time (efficiency optimization) control.
DPSK_Deadtime.h	Header file	Header file for dead time (efficiency optimization) control.
DPSK_Forward.c	Process	V_{IN} feed forward algorithm.
DPSK_Forward.h	Header file	Header file for V_{IN} feed forward algorithm.
DPSK_kernel.c	Kernel State Process	Kernel state control processes.
DPSK_kernel.h	Header file	Header file for kernel state control processes.
DPSK_main.c	Kernel Main	Main.c program from which DPSK_kernel.c is called.
DPSK_Setup.h	Header file	Kernel configuration file for DPSK_kernel.c
DPSK_Transient.c	Process	Nonlinear control algorithm for load and unload output voltage transients.
DPSK_Transient.h	Header file	Header file for nonlinear control algorithm.
PMB_Cmds.c	Network Interface Support	Network interface command interpreter.
PMB_Drv.c	Network Interface Support	Network interface low-level SMBus driver.
PMB_Drv.h	Header file	Header file for network interface low-level SMBus driver.
PMB_Mem.c	Network Interface Support	Network interface memory manager.
PMB_Mem.h	Header file	Header file for network interface memory manager.
PMBus.c	Network Interface Support	Network interface process (manages command interpreter, memory manager, and SMBus driver).
PMBus.h	Header file	Header file for network interface process.
PMBus_setup.h	Header file	Header file for network interface process.
Si8250.h	Header file	Header file for Si8250 digital power controller.
Silabs_power.h	Header file	Header file for library functions.
Silabs_power.lib	Library	Library file (contains over 500 individual function files).

3. Porting the K2.0 Kernel to the End Application

Porting the kernel to the end application requires the following operations (all tools are included in Si8250DK development kit and are available for download at www.silabs.com):

1. Loop compensator design using the Compensator tool (located in the Application Builder).
2. Switch timing design using the Timing Editor tool (located in the Application Builder).
3. System management processor configuration using the System Wizard tool (located in the Application Builder).
4. Add kernel code for user-specific system functions using the IDE editor.
5. Compile and download the code into the Si8250 using the IDE editor and C-compiler (Note: full compiler version is required for object code files larger than 4 kB).
6. Debug the system using the IDE Debugger; modify code as needed, and iterate between Steps 5 and 6 until the system is debugged.

3.1. Compensator Design

The buck regulator compensation design tool is useful for isolated and non-isolated buck converters including single and multi-phase POLs, half-bridge, full bridge, and other buck configurations. System parameter values are entered into the appropriate fields along with the desired pole/zero frequencies. The compensator tool then generates frequency response curves for the system, controller, and plant while generating the required DSP Filter Engine coefficients and writing them into the source code. See “AN259: Designing with the Si825x Family of Digital Power Controllers” for more details on using this and other tools referenced in this document.

3.2. Timing Design

The timing editor tool automatically generates timing initialization code for the Si8250 based on the user’s graphical input. The user specifies the switching cycle length and then draws the required timing, specifying the interphase and phase-to-phase timing relationships. A simulation function allows the user to quickly verify operation. Timing parameters are automatically generated and written to the source code upon exit from this tool.

3.3. System Processor Configuration

The system configuration wizard tool allows the user to select and configure peripherals using drop-down lists and check boxes. Initialization code is automatically generated and written to the source file upon exit from this tool.

3.4. Adding Proprietary Code

Code is added by typing the C-language code into the editor. Be sure to maximize use of the kernel library functions for maximum code efficiency and reliability.

Note: Adding new (unproven) code can complicate system debug. It is therefore recommended that new code added in this step be initially disabled (commented out or bypassed) until the basic kernel functions have been verified. The proprietary code added in this step can then be enabled and debugged one segment at a time.

3.5. Compiling, Loading and Debugging Code

Compile and load the program using the C-compiler and debugger. Note that any object code file (machine code) greater than 4 kB will require the full compiler version. Code is debugged using the real-time debugger contained in the Silicon labs IDE. In cases where the source code is new, debugging may be an iterative process of finding code bugs, modifying the code, and compiling and running the new code version until the system operates properly.

4. Kernel Process State Descriptions

4.1. Kernel Process (*Kernel_Process*)

4.1.1. Description and Operation

This kernel process state passes program control to the appropriate kernel process state when called. As each kernel process state completes execution, it specifies the next kernel process state to be called based on its own internal operating results. For example, a given kernel process state may detect a fault and specify the next fault recovery initialization process state to be executed by writing the appropriate process value to the variable, *UkernelState. Kernel_Process* contains a *switch* instruction with multiple *case* statements, each associated with a specific kernel process. Each *case* statement compares the value in *UkernelState* to its associated kernel process state. Program execution is immediately vectored to the matching kernel process state.

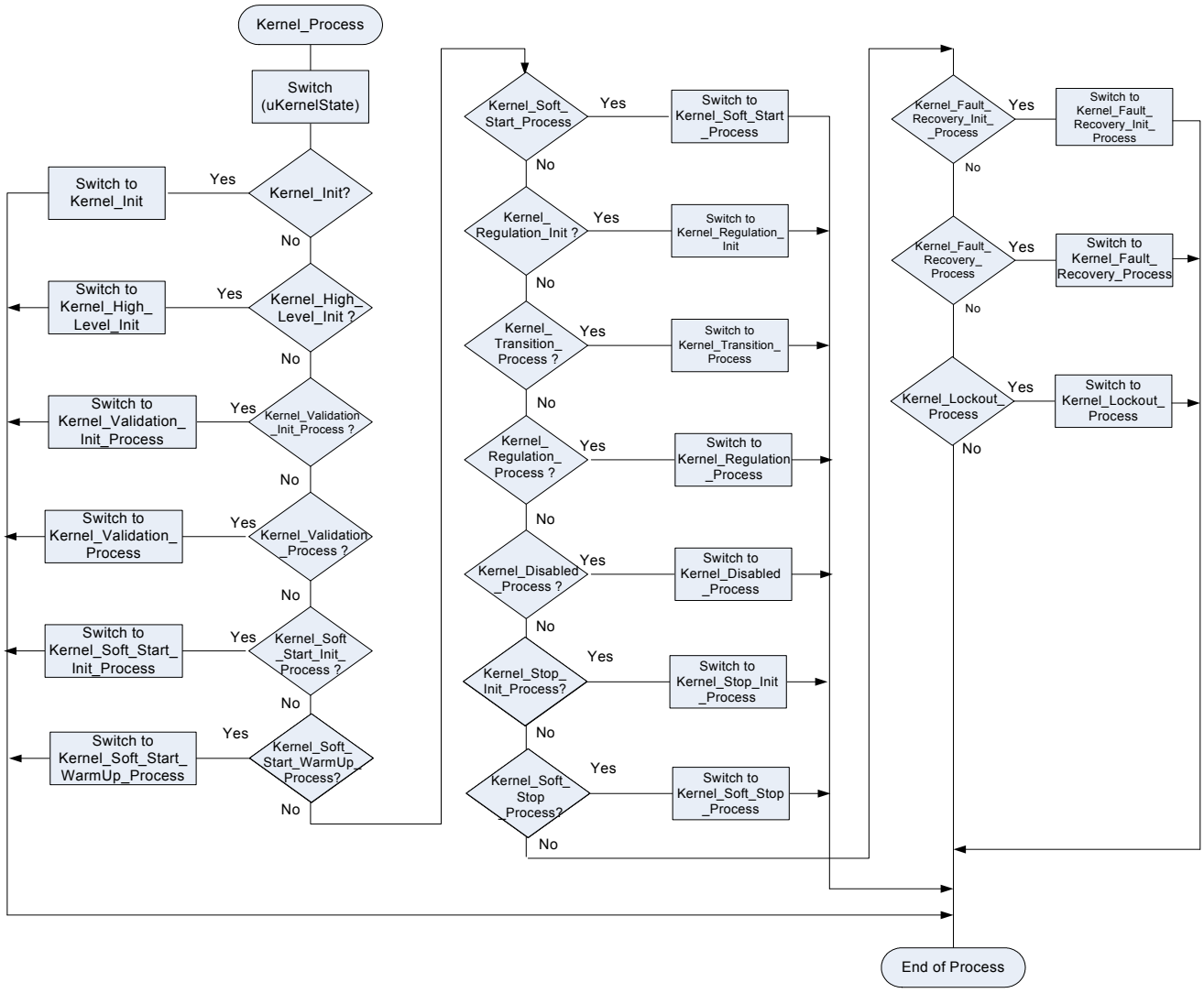

Figure 5. Kernel_Process Flowchart

4.2. Kernel Initialization (*Kernel_Init*)

4.2.1. Description and Operation

This kernel process state initializes all of the hardware and software settings onboard the Si825x in preparation for startup. This is accomplished by retrieving stored constants in Flash memory and writing them to the appropriate special function register (SFR) and memory locations within the Si825x. As shown in Figure 6, *Kernel_Init* begins by enabling the V_{DD} monitor, which resets to the Si825x when its bias supply falls below a preset minimum voltage. The MCU watchdog timer is then enabled and the oscillator frequency set.

The remaining on-chip hardware is initialized in a similar way: the port I/O mode (open drain, push/pull/weak pullup) is configured, and pin functions are assigned (e.g., assignment of SMBus port lines, ADC0 inputs, etc.). Timer2, which supplies the kernel 100 µs time tick, is then initialized and enabled. This is followed by initialization of the UART, external ENABLE input, and REFDAC. When a network interface is built into the project, the kernel parameter table pointer defaults to the *scKernelCtlDefault* table only when the Si825x is run for the first time after programming. Thereafter, the pointer defaults to the network interface settings structure. For a non-network interface build, the pointer unconditionally defaults to *scKernelCtlDefault*. The library function, *sxKernelCtl*, block copies the contents of the selected parameter table from Flash memory into the appropriate RAM locations. The remaining hardware initialization (DPWM, DSP Filter current limit, and OCP) are then initialized, and interrupts are enabled. Variable *uKernelState* is then set to *KSTATE_HIGH_INIT*, which is decoded by *Kernel_Process* and results in a switch to the *Kernel_High_Level_Init_Process*.

Kernel_Init also services the ADC0 analog multiplexer (AMUX). As shown in the lower right side of Figure 6, an ADC0 end-of-conversion interrupt causes software to increment the AMUX channel and initiate another conversion cycle.

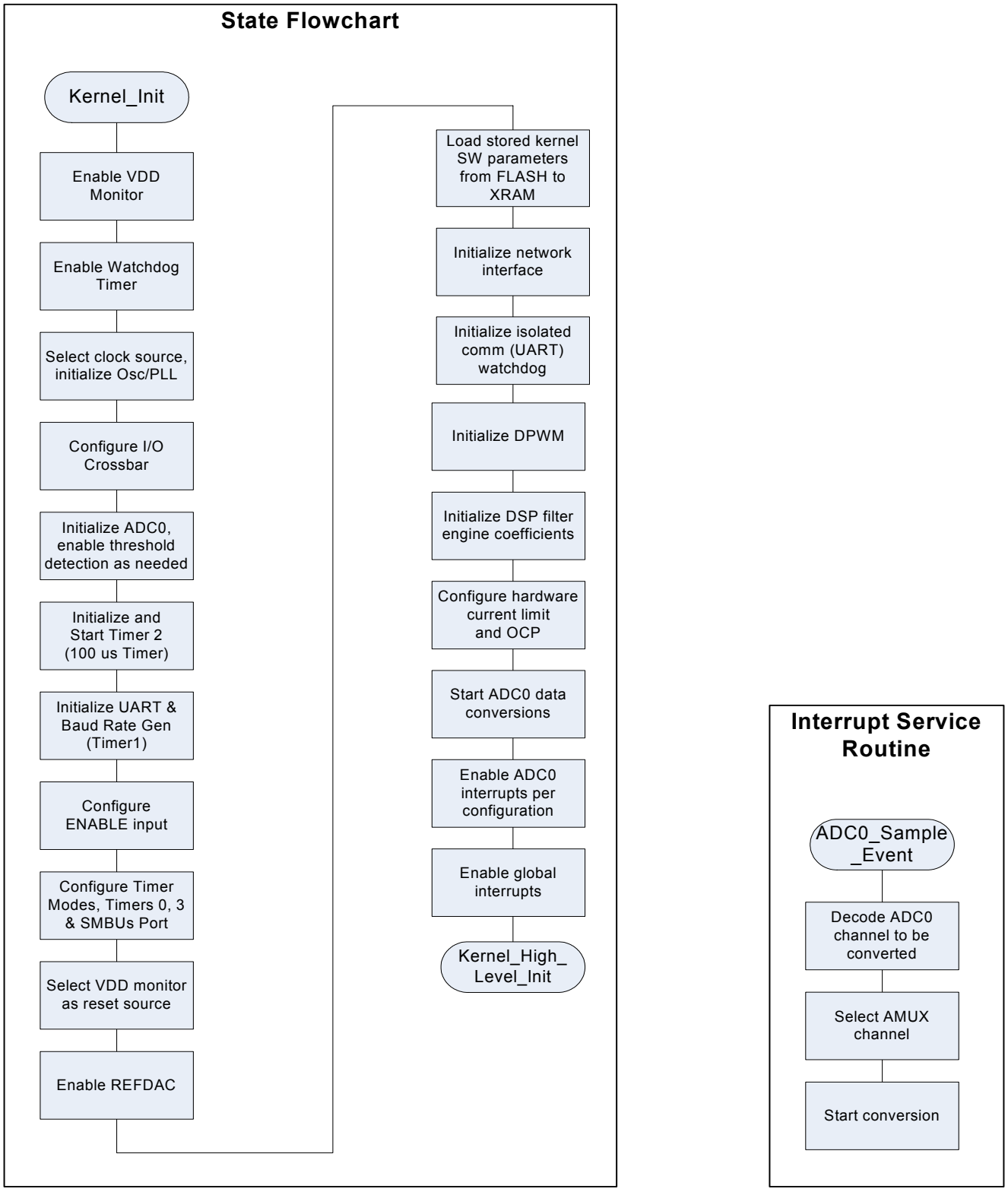

Figure 6. Kernel_Init Flowchart

4.3. Kernel High Level Initialization (*Kernel_High_Level_Init_Process*)

4.3.1. Description and Operation

This kernel process state is essentially a “power converter reset” state. That is, various power-converter-specific functions and parameters, such as status and control flags, internal and external enable controls, stop modes, and restart counters, are initialized. No low-level Si825x hardware initialization, such as I/O port configuration, is performed.

Program execution begins by resetting the communications watchdog timer. This timer asserts its output when UART communication (typically used to transmit data across an isolation barrier) ceases for a specified time period indicating a communications fault. For example, an Si8250 located on the secondary side of a power supply might use a primary-side microcontroller to digitize and transmit V_{IN} data across an isolation barrier using its UART. A failure in the primary-side controller would cause the communications across the isolation barrier to cease, which, in turn, would cause the communications watchdog to time out and assert its output. The Si8250 recognizes this action as a communications fault and halts operation as a result.

The next program operates reset status and control flags and initializes the internal and external “*enables*” signals (if used). The output voltage margins are set, and the counters and restart timers are reset. Program execution then exits to the validation phase in preparation for soft-start.

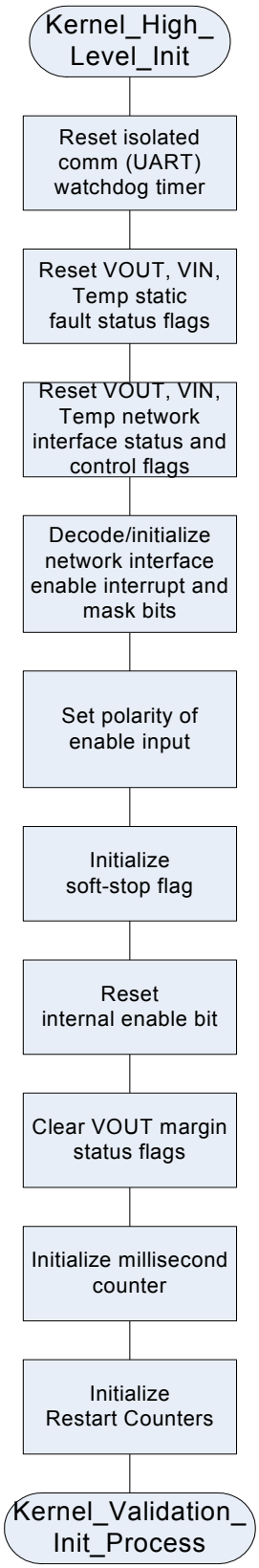

Figure 7. Kernel_High_Level_Init Flowchart

4.4. Kernel Validation Initialization (*Kernel_Validation_Init_Process*)

4.4.1. Description and Operation

This kernel process state (Figure 8) prepares the system for "validation"—a sequence of system parameter tests to determine if it is permissible to initiate soft-start. Program execution begins by updating status and control flags. To ensure that fault and warning flag states are correct, they are reset to zero during this kernel process state in preparation for updating during the validation process state that immediately follows.

Startup delay (t_{ON}) is defined as the time period between the end of validation and the beginning of the actual soft-start ramp (startup delay is a programmable variable specified in the network interface spec). This delay is implemented using a software startup delay counter synchronized to the 100 µs event timer, and the time-out value is stored in the variable, *sxKernelCtl.ton_delay*. This process state is then complete; the next kernel process state is validation.

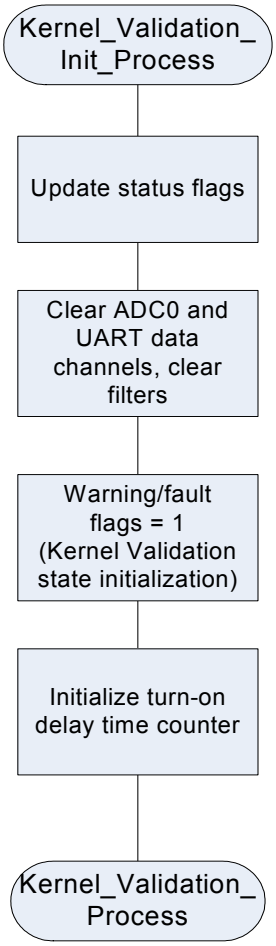

Figure 8. Kernel_Validation_Init_Process Flowchart

4.5. Kernel Validation Process (*Kernel_Validation_Process*)

4.5.1. Description and Operation

This kernel process state unconditionally tests system parameters to determine if it is permissible to initiate soft-start. In addition, it manages a software 1 ms startup timer and validation counter when the 100 µs time tick is asserted.

Program execution begins with initialization of the validation counter, which keeps track of the validation status by decrementing each time a startup permissive is satisfied. As a result, a validation counter will equal zero when all permissives have been satisfied, indicating soft-start can begin. If the end application is isolated, the Si825x is assumed to be on the secondary side of the supply. A small MCU with on-board ADC located on the primary-side is recommended to digitize local analog parameters, such as supply input voltage, and transmit them to the secondary-side Si825x in packet form via UART. (Application software for the primary-side C8051F300 MCU is also included in the Si8250DK development kit or available for download at www.silabs.com.)

The function, mIsUINxRdy(), checks for the presence of new primary-side data from UART, and the mUINPacketProcess() packet processor is called if new data is available. The packet processor “unbundles” the individual parameters received. Fault status is updated as follows: fault flags are cleared, and the Kernel_UIN_Threshold_Process() is called to test the value of each parameter against upper and/or lower limits. The packet communication is finalized by the UART done function, mUINxDone(). If the end application is non-isolated, supply input voltage (and all other parameters) are digitized ADC0 and the thresholds checked in software. Analog parameters converted by ADC0 are digitized and filtered (averaged) to minimize error due to noise. The resulting values are then threshold-checked to ensure they are within limits. The read temperature sensor routine is a typical example of conversion and conditioning of an analog process. The function, mIsInputRdy(TEMP), checks for the presence of converted temperature data. If new data is available and all temperature-related warning and fault flags are cleared, the converted data is filtered by the *mFilterProcess(TEMP)* function and limit-checked by *Kernel_TEMP_Threshold_Process*.

The result of these operations is an updated, filtered temperature measurement and an updated set of temperature warning and fault flags. (Other analog parameters are handled in much the same way; for example, the sequence, *mIsInputRdy(AIN3), mFilterProcess(AIN3), Kernel_AIN3_Threshold_Process*, processes ADC0 input AIN3 in the same manner as the temperature sensor output signal described above.) The feedback parameter, VSENSE, is digitized in the same way, except filtering is provided by the *mAdvFilterProcess (VSENSE,6)* function. This function is an advanced filter with a programmable cutoff frequency determined by the number in parentheses (valid cutoff frequency range is 0 to 8).

Kernel operations are synchronized by a 100_µs timetick supplied by Timer2. *If a 100 µs time tick is asserted*, interrupting source Timer2 is reset, and the communications watchdog, delay counter (used in the t_{ON} delay algorithm), and startup counter (used in soft-start algorithm) are all updated. *If 100 µs time tick is not asserted*, system parameters are tested by the validation tree to determine if it is permissible to initiate soft-start. Each decision in the validation tree tests the state of each parameter status flag—if the state is false (no fault present), the validation counter is decremented; if the state is true, the validation counter is left unchanged. As shown in Figure 9, the input voltage, internal and external enable, temperature, and communication fault flags are checked, and if all are false, the validation count is driven to zero, and startup initialization begins.

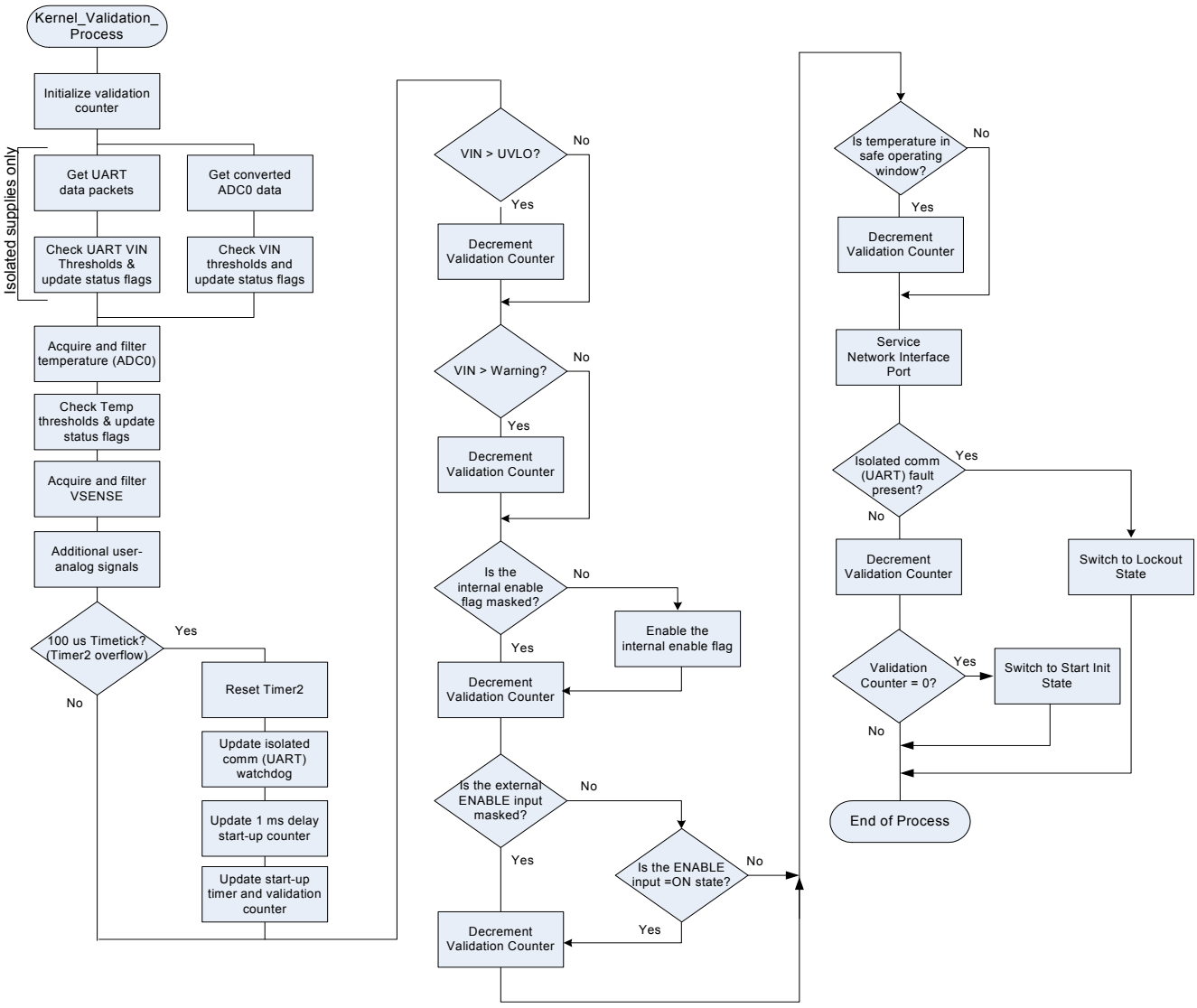

Figure 9. Kernel_Validation_Process Flowchart

4.6. Kernel Soft Start Initialization Process (*Kernel_Soft_Start_Init_Process*)

4.6.1. Description and Operation

This kernel process state (Figure 10) prepares the system for a soft-start. It begins by clearing all fault and warning status flags. The *bDeviceOff* status flag is cleared to indicate the power system is no longer in the OFF state. Next, the REFDAC is set to zero output in preparation for a zero pre-bias start (startup into pre-bias is automatically handled by the *Kernel_Soft_Start_Process* state). The integrator term in the PID filter is enabled (it was previously disabled to prevent integrator wind-out), and ADC1 is enabled. The output voltage target value is set next. To avoid overshoot at the end of transition, the output voltage target is set to a value that is slightly less than the specified regulated output voltage. This is accomplished by stopping the voltage ramp two 100 µs time ticks *before* the rated output voltage level is achieved (the last part of the voltage ramp will be completed by the transition process). Finally, the startup warm-up timer is initialized, and control execution passes to the soft-start warm-up process state.

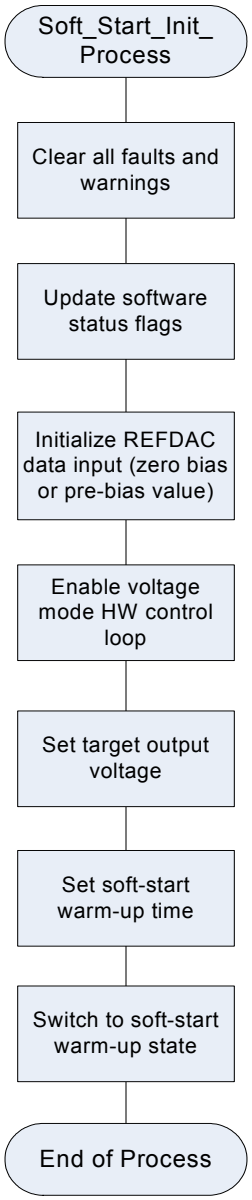

Figure 10. Soft_Start_Init_Process Flowchart

4.7. Kernel Soft Start Warm-up Process (*Kernel_Soft_Start_WarmUp_Process*)

4.7.1. Description and Operation

This kernel process state provides an adjustable time delay (t_{ON}) prior to the beginning of the soft-start ramp (t_{ON}). This delay time is beneficial in that it allows the software filter outputs to settle. This process state unconditionally performs routine kernel maintenance operations (averaging and threshold-testing parameters, etc.). In addition, soft-start warm-up-specific operations are performed when the 100_µs time tick is asserted.

If the 100 µs time tick is asserted, interrupting source Timer2 is reset, and startup delay timer service begins. If the startup delay counter is a zero, delay time has expired, and the startup delay counter is reset followed by initialization of soft-start parameters as follows:

- Parameter soft_ref determines the step size of each update to the soft-start ramp, that is, the number of REFDAC LSBs changed by software on each soft-start ramp update. Soft_ref will be = 1 for all but the fastest ramp times; increasing soft_ref increases soft-start slope but decreases step resolution.
- The soft scale parameter (*soft_scale_val*) determines the rate at which the soft-start ramp is updated—the higher the value, the longer the time between updates (and consequently the slower the soft-start ramp).
- Following these operations, a RETURN statement is executed, and program control exits this process state and returns to the calling process state.

If the startup delay counter is not zero, startup delay is still in progress. In this case, the startup timer is updated (decremented), and the average duty cycle is calculated by the *mAdvFilterProcess(DPWM,8).* The isolated communications watchdog timer is then updated, and program execution is moved to maintenance operations starting with network interface service (merge point A in Figure 11).

If 100 µs time tick is not asserted, program execution moves to maintenance operations starting with network interface service (merge point A in Figure 11). Analog parameters are converted averaged and their limits threshold-checked (UART for primary-side parameter measurements in isolated systems). The state control processor is then called to manage possible fault conditions.

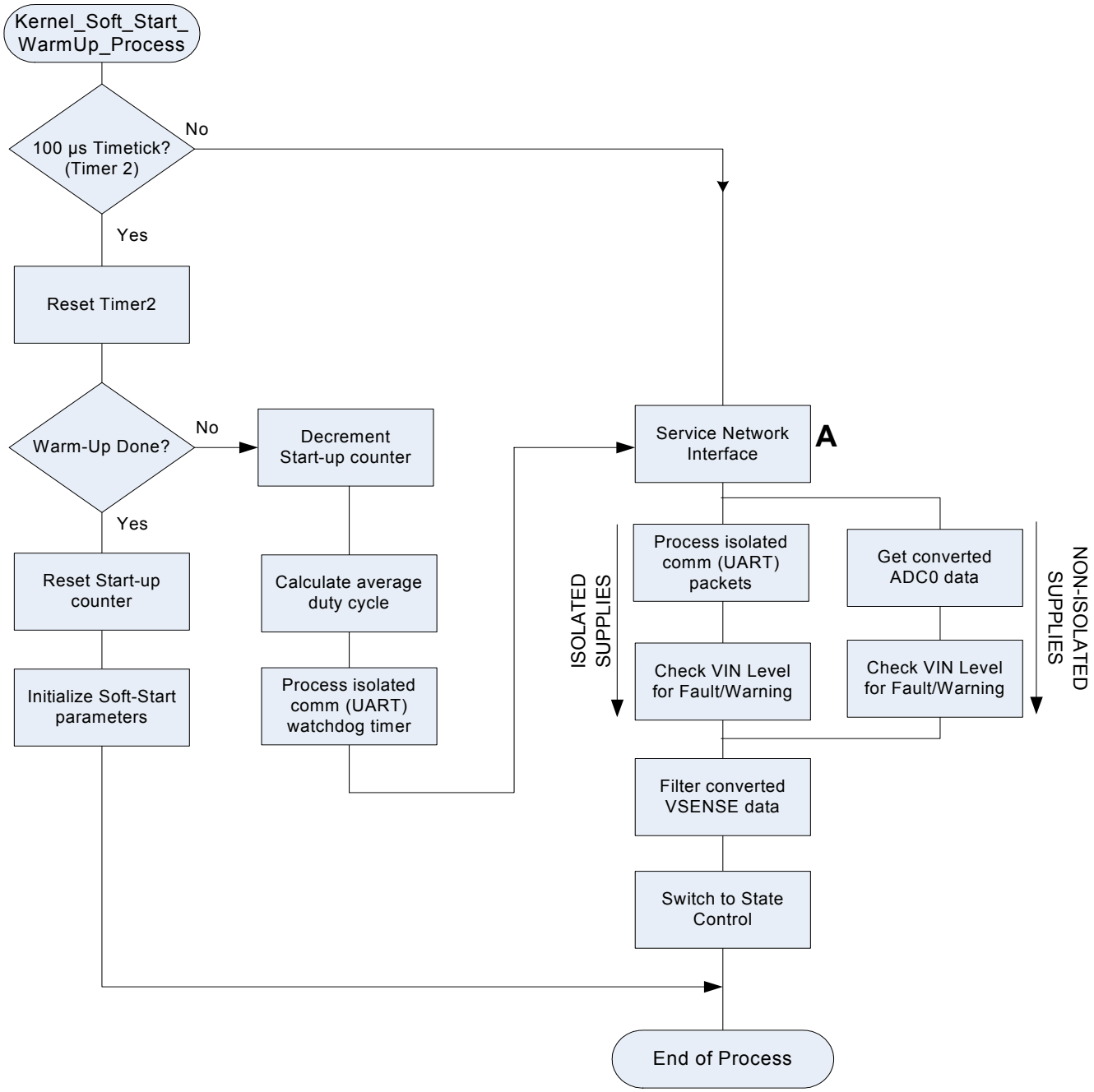

Figure 11. Kernel_Soft_Start_WarmUp_Process Flowchart

4.8. Kernel Soft Start Process (*Kernel_Soft_Start_Process*)

4.8.1. Description and Operation

This kernel process state generates a fixed rate output voltage ramp. The digitized VSENSE parameter is retrieved when the result of the *mIsInputRdy(VSENSE)* is logic 1 and averaged by the function, *mAdvFilterProcess (VSENSE,6)*. This function is an advanced filter with a programmable cutoff frequency determined by the number in parentheses (valid cutoff frequency range is 0 to 8—the higher the number the lower the cutoff frequency). The network interface is then serviced.

If the end application is isolated, the Si825x is assumed to be on the secondary side of the supply. A small MCU with onboard ADC located on the primary side is recommended to digitize local analog parameters, such as supply input voltage, and transmit them to the secondary-side Si825x in packet form via UART. (Application software for the primary-side MCU is also included in the Si8250DK development kit or available for download at **www.silabs.com**). The function, *mIsUINxRdy()*, checks for the presence of new primary-side data from UART, and the *mUINPacketProcess()* packet processor is called if new data is available. The packet processor "unbundles" the individual parameters received. Fault status is updated as follows: fault flags are cleared, and the threshold process, *Kernel_UIN_Threshold_Process(),* is called to test the value of each parameter against upper and/or lower limits. The packet communication is finalized by the UART done function, *mUINxDone()*.

The converted V_{SENSE} and temperature values are retrieved when the results of the *mIsInputRdy(VSENSE)* and *mIsInputRdy(TEMP)* functions are logic 1, respectively. Filtering is provided by the *mAdvFilterProcess (VSENSE,6)* function. The *mFilterProcess(TEMP)* function averages the temperature sensor output signal, followed by limit checking by the *Kernel_TEMP_Threshold_Process*.

Kernel operations are synchronized by a 100 µs timetick supplied by Timer2. When a timetick occurs, Timer2 is reset and the supply output voltage increased by the soft-start slope regulation algorithm *Indirect_Soft_Transition_Process()*. This algorithm regulates the linearity of the soft-start ramp by automatically adjusting the REFDAC to maintain a constant output voltage slope. Upon exit from this function, the REFDAC data value is checked for a magnitude greater than the output voltage target. If the results of this test are true, the soft-start kernel process state is complete, and program control is passed to the regulation initialization process state (*Kernel_Regulation_Init_Process*). If the REFDAC data value is below the target output voltage level, the soft-start kernel process state is not yet complete; as a result, soft-start kernel process state execution continues. The soft-start kernel process state has a built-in 1 ms software timer that both acts as a prescaler for the startup timer; it also ensures that the isolated communication watchdog timer and t_{ONMAX} fault are periodically serviced. The 1 ms timer is serviced as follows: if the 1 ms timer value is not equal to zero, the 1 ms timer is updated (decremented); then, the startup (t_{ONMAX}) software timer is updated. The startup timer is then checked for a zero value (expiration), in which case the t_{ONMAX} fault flag (*bTonMaxFault*) is set indicating a t_{ONMAX} fault. If the timer value is zero (1 ms timeout expired), the timer is reset, and program execution moves immediately to process exit service routines (isolated communications watchdog service, calculation of average duty cycle and service of t_{ONMAX} fault). The final step in this kernel process state is servicing the t_{ONMAX} fault; if the t_{ONMAX} fault flag is set indicating a fault is present, program control is transferred to the fault recovery initialization state. Otherwise, the state control process state is called to resolve any pending faults.

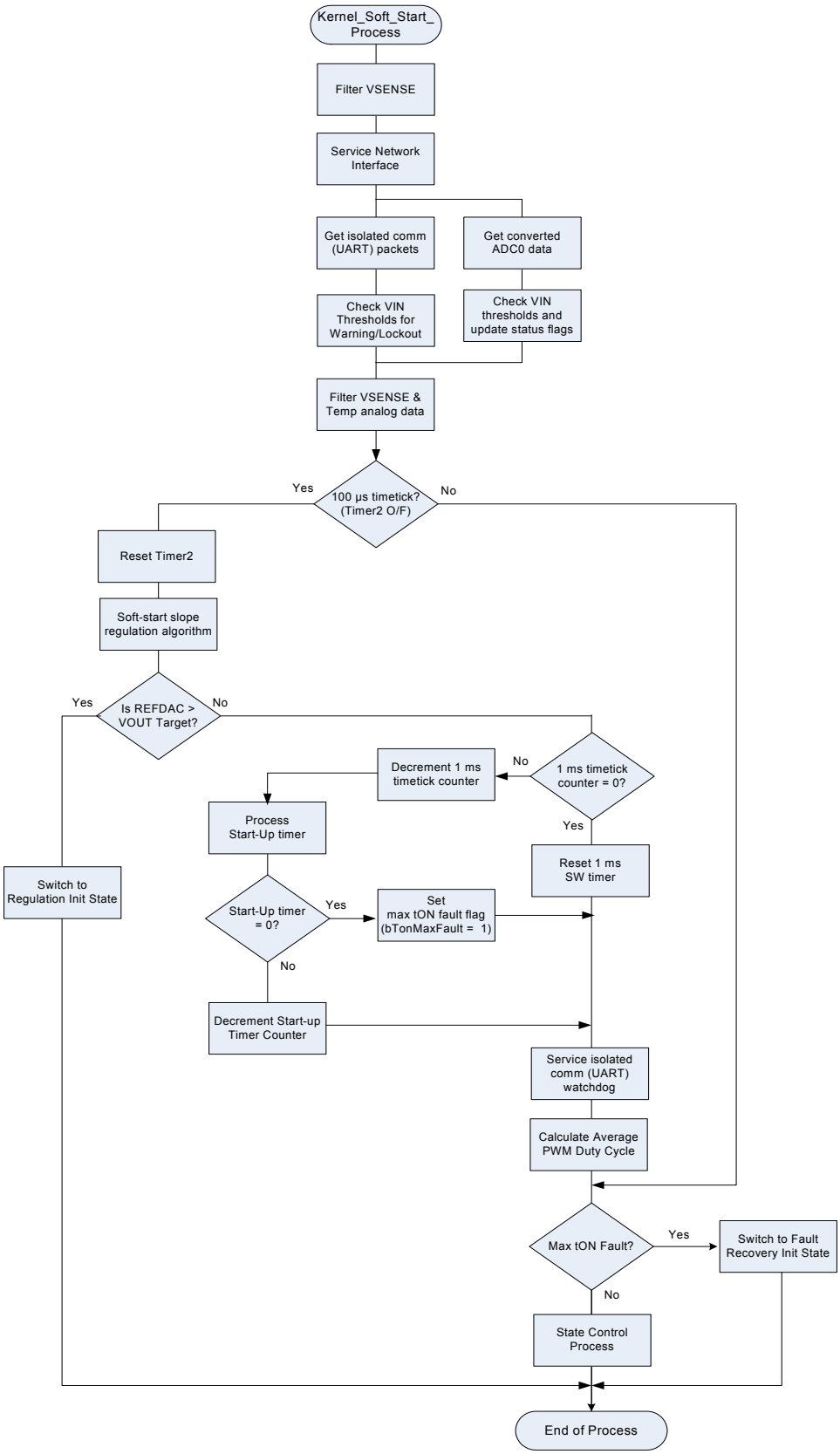

Figure 12. Kernel_Soft_Start_Process Flowchart

4.9. Kernel Regulation Initialization Process (*Kernel_Regulation_Init_Process*)

4.9.1. Description and Operation

This process prepares the system to enter steady-state regulation. Process execution begins with updating software status flags followed by initialization of the following network interface parameters: transition rate, V_{OUT} margins, and trim and cal settings. All fault flags are then cleared (to remove any residual status information that has already been serviced). ADC1 LSB size is then set, and program control switches to the kernel transition process state.

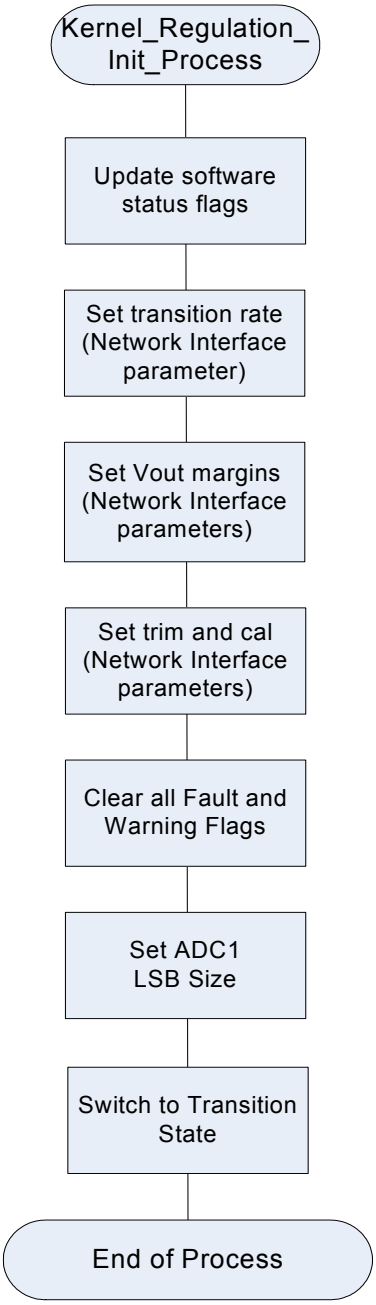

Figure 13. Kernel_Regulation_Init_Process Flowchart

4.10. Kernel Transition Process (*Kernel_Transition_Process*)

4.10.1. Description and Operation

The kernel transition process state completes the soft-start process by gently ramping the output voltage into its specified range without overshoot. This process state unconditionally performs routine kernel maintenance operations (averaging and threshold-testing parameters, etc.). In addition, transition-specific operations are performed when the 100 µs time tick is asserted. (Transition operations consist of decrementing a transition rate timer (counter) on each 100 µs timetick and updating the output voltage value when the timer reaches zero.) An output voltage update operation consists of servicing the transition rate counter, updating the REFDAC control variable, and writing the updated control value to the REFDAC. This sequence of operations results in an incremental change in output voltage toward its nominal steady-state value.

If the 100 µs time tick is asserted, interrupting source Timer2 is reset, and a transition operation is executed as follows:

- **If the transition rate timer is zero**, it is time for an output voltage update. In this case, the transition rate counter is reset to its initial value, and the output voltage is updated as follows:
  - The REFDAC data value is retrieved and compared to the target output voltage value. If the REFDAC data value equals the target output voltage value, transition is complete, and execution switches to the kernel regulation process state.
  - The REFDAC data value is incremented if its value is less than the target output voltage and decremented if it is greater than the target output voltage value.
  - The updated REFDAC data value is written to the REFDAC.
- **If the transition rate timer is not zero**: the transition rate counter is decremented and maintenance tasks are performed (service isolated communications (UART) watchdog timer, calculate average duty cycle). Program execution then moves to the maintenance operations at program merge point A in Figure 14 (the same routine maintenance sequence executed when the 100 µs time tick is not asserted as described above).

If the 100 µs time tick is not asserted, program execution moves to program merge point A of Figure 14. (It is also possible to arrive at this process program location if the 100 µs was executed without completing the transition process.) The digitized VSENSE parameter is filtered (averaged) by the *mFilterProcess (VSENSE)* function, and the converted temperature sensor output signal is filtered by the *mFilterProcess (TEMP)* function. The resulting V_{SENSE} and temperature values are then threshold checked for warnings and faults by the *Kernel_VSENSE_Threshold_Process* and *Kernel_TEMP_Threshold_Process,* respectively. Service to the network interface port is then performed, followed by an input voltage measurement and threshold check.

If the end application is isolated, the Si825x is assumed to be on the secondary side of the supply. A small MCU with onboard ADC located on the primary side is recommended to digitize local analog parameters, such as supply input voltage, and transmit them to the secondary-side Si825x in packet form via UART. (Application software for the primary-side MCU is also included in the Si8250DK development kit or available for download at **www.silabs.com**.) The function, *mIsUINxRdy()*, checks for the presence of new primary side data from UART, and the *mUINPacketProcess()* packet processor is called if new data is available. The packet processor “unbundles” the individual parameters received. Fault status is updated as follows: fault flags are cleared, and the Threshold Process *Kernel_UIN_Threshold_Process()* is called to test the value of each parameter against upper and/or lower limits. The packet communication is finalized by the UART done function, *mUINxDone()*. The kernel state control process state is then called to process any pending fault conditions.

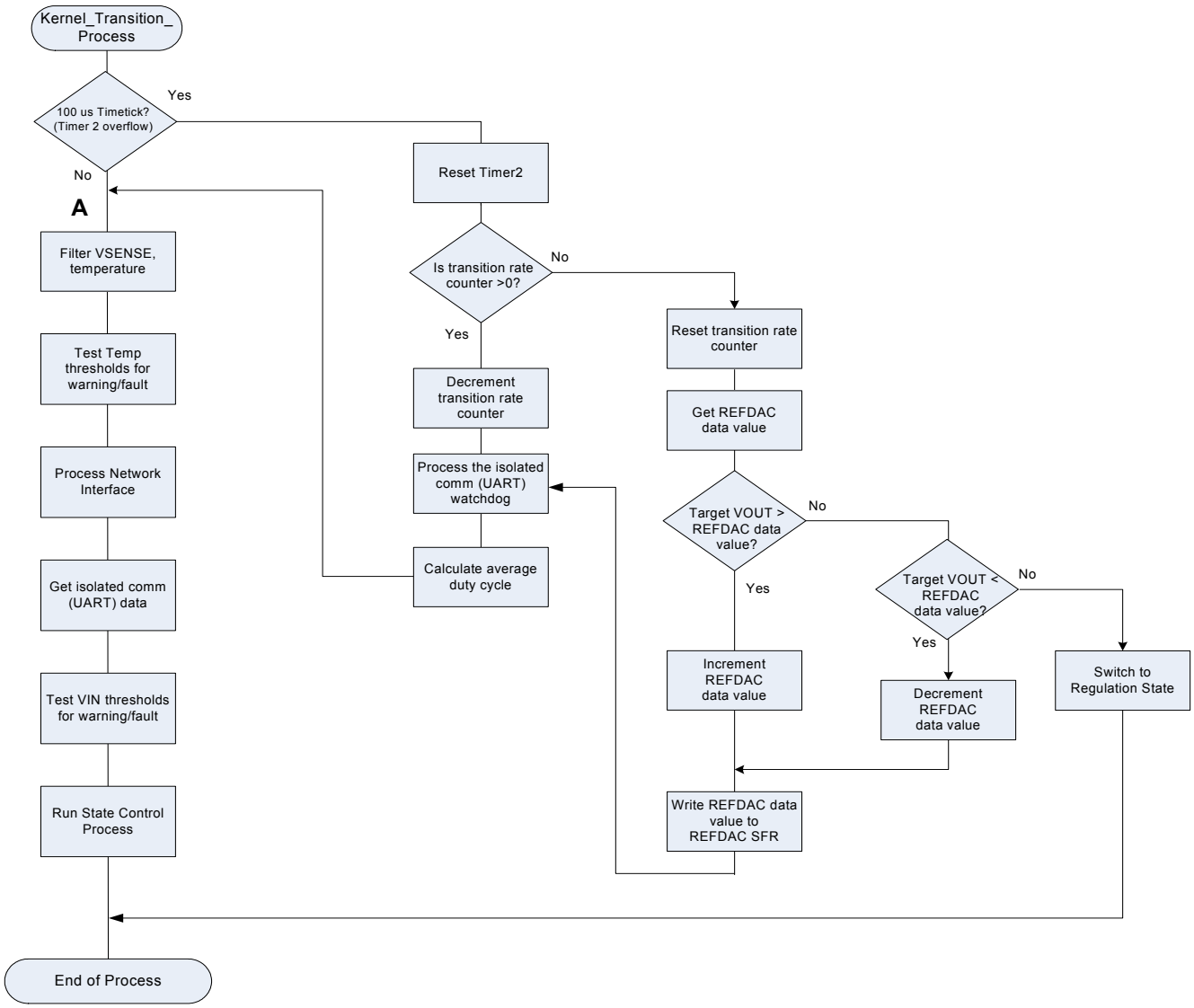

Figure 14. Kernel_Transition_Process Flowchart

4.11. Kernel Regulation Process (*Kernel_Regulation_Process*)

4.11.1. Description and Operation

This kernel process state handles all system operations during steady-state regulation including nonlinear control, input voltage feed-forward, fault detection, network interface service, and others. This process state unconditionally performs routine kernel maintenance operations (averaging and threshold-testing parameters, etc.). In addition, steady-state regulation-specific operations are performed when the 100 µs time tick is asserted.

The kernel regulation process state begins with the analog-to-digital conversion of the V_{SENSE} and the onboard temperature sensor signal. The converted V_{SENSE} result is filtered by the *mFilterProcess (VSENSE,6)* function, and the temperature value is filtered by the *mFilterProcess(TEMP)* function. Each resulting value is threshold-checked by the *Kernel_VSENSE_Threshold_Process* and *Kernel_TEMP_Threshold_Process,* respectively.

Kernel operations are synchronized by a 100 µs time tick event generated by Timer2.

If a 100 µs time tick is asserted, interrupting source Timer2 is reset, the isolated communications watchdog timer is then updated by the function, *mUINWtDogProcess()*, and the average duty cycle is calculated by the function, *mAdvFilterProcess (DPWM, 8)*—an advanced filter with a programmable cutoff frequency determined by the number in parentheses (valid cutoff frequency range is 0 to 8). The transient detector threshold is also fine-tuned. (The nonlinear control algorithm is on a separate processing thread and is invoked by an interrupt generated by the transient detector.) Program execution then moves to program merge point A (Figure 15) to begin execution of maintenance operations.

If a 100 µs time tick is not asserted, routine maintenance updates are performed. (These same maintenance operations (merge point A on Figure 15) are also executed after the completion of the 100 µs interval soft-stop processing.)

The remainder of this process state consists of processing and threshold testing the input voltage level and possibly addressing related faults. If the end application is isolated, the Si825x is assumed to be on the secondary side of the supply. A small MCU with on-board ADC located on the primary-side is recommended to digitize local analog parameters, such as supply input voltage, and transmit them to the secondary-side Si825x in packet form via UART. (Application software for the primary-side MCU is also included in the Si8250DK development kit and is available for download at **www.silabs.com**.)

The function, *mIsUINxRdy()*, checks for the presence of new primary-side data from UART, and the *mUINPacketProcess()* packet processor is called if new data is available. The packet processor “unbundles” the individual parameters received. Fault status is updated as follows: fault flags are cleared, and the threshold process, *Kernel_UIN_Threshold_Process(),* is called to test the value of each parameter against upper and/or lower limits. (If the end application is non-isolated, supply input voltage (and all other parameters) are digitized ADC0 and the thresholds checked in software.

Analog parameters converted by ADC0 are digitized, then filtered (averaged) to minimize error due to noise. The resulting values are then threshold-checked to ensure they are within limits. The next operations are low-bandwidth loop optimization functions. The function, *Balance_Primary_Process()*, adjusts the DPWM trim registers to change the pulse width offset of the PH1 output to maintain the voltage on the input capacitive divider at measured $V_{IN}/2$. Voltage mode control loop gain varies with input voltage; so, analog systems typically apply PWM slope compensation to counteract these changes by varying the slope of the PWM ramp. The function, *Feed_Forward_Process()*, achieves the same result in the digital domain by varying gain term A3 in proportion with the value of V_{IN} to maintain a constant loop gain over the specified input voltage range.

The function, *Active_Dead_Time_Process()*, continuously adjusts the dead time between the main buck switch and the synchronous rectifier switch to minimize the freewheeling time of the synchronous rectifier body diode.

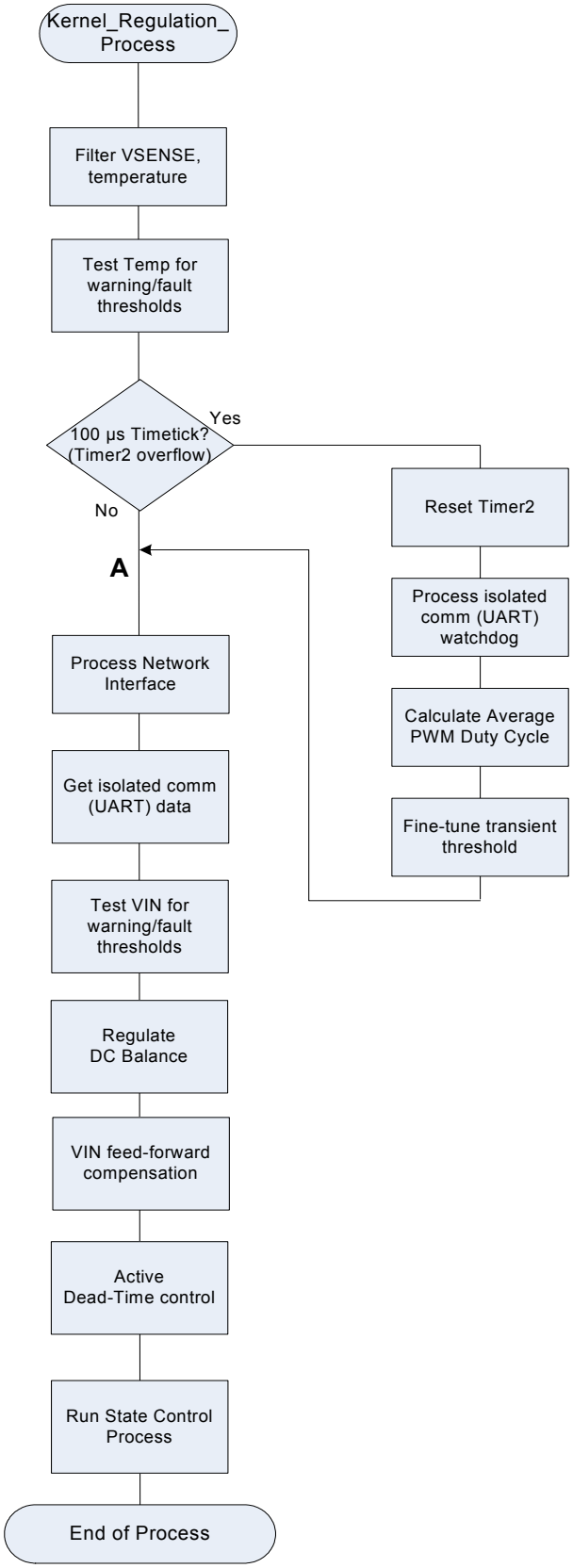

Figure 15. Kernel_Regulation_Process Flowchart

4.12. Kernel Stop Initialization Process (*Kernel_Stop_Init*)

4.12.1. Description and Operation

The kernel stop initialization process state prepares the system for shutdown. Default operation is as follows:

- A “hard stop” (DPWM outputs immediately bypassed to programmed stop states) if the shutdown is caused by a fault. In this case, the next kernel process state is fault recovery initialization.
- A “soft stop” (controlled, negative-going output voltage ramp from nominal output voltage to zero volts) if soft-stop mode is selected, and if the source of the shutdown is **not** a fault. In this case, execution moves to the disabled kernel process state.

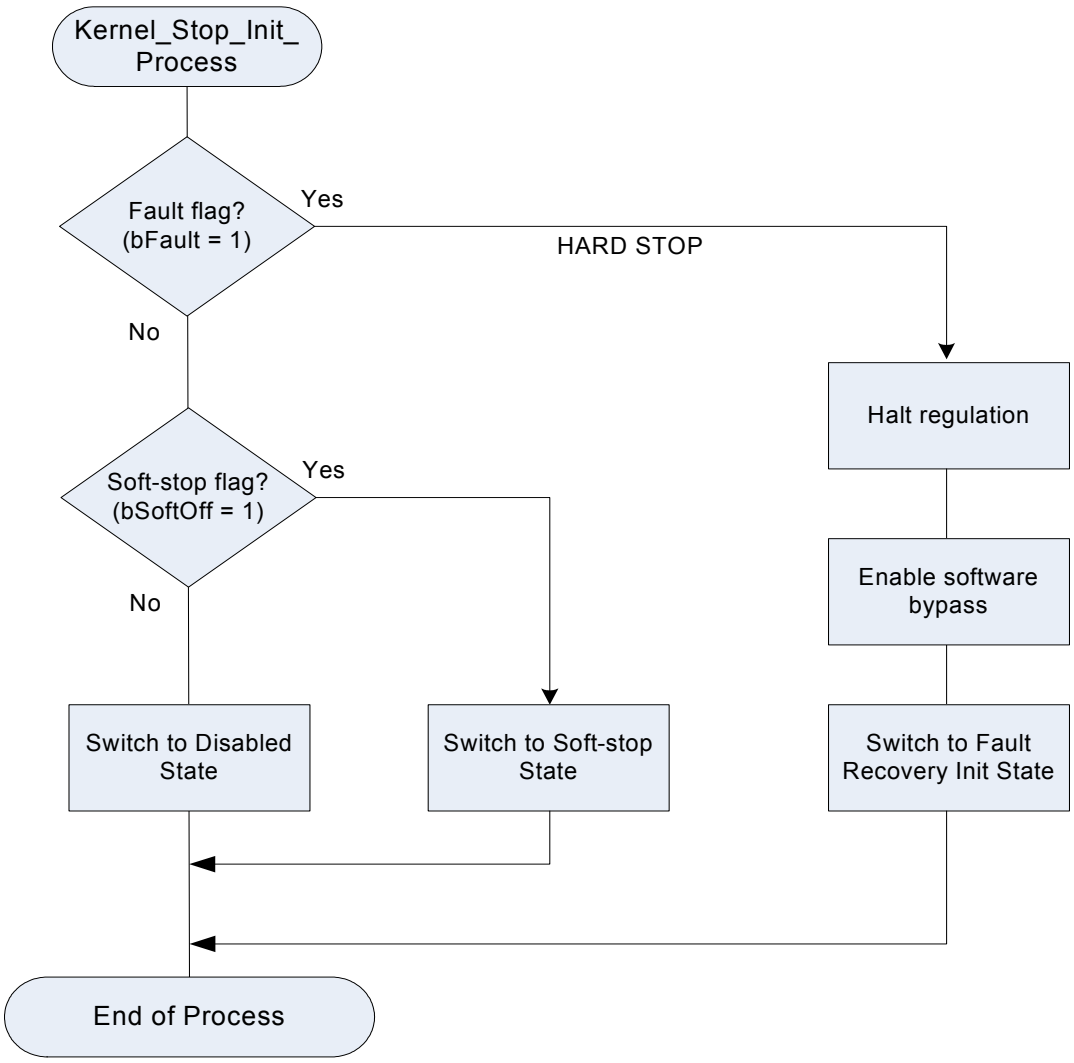

Figure 16. Kernel_Stop_Init_Process Flowchart

4.13. Kernel Soft Stop Process (*Kernel_Soft_Stop*)

4.13.1. Description and Operation

This kernel process state ramps the output voltage from its nominal value to zero at a fixed rate (it is essentially "soft-start in reverse"). This process state unconditionally performs routine kernel maintenance operations (averaging and threshold-testing parameters, etc.). In addition, soft-stop-specific operations are performed when the 100 µs time tick is asserted.

Note: System shutdown automatically bypasses the kernel soft-stop process state and invoices a hard (immediate) stop if the event causing the shutdown is a fault.

If a 100 µs time tick is asserted, interrupting source Timer2 is reset, and the soft-stop counter is checked for a non-zero value. If the value is non-zero, the soft-stop timeout is still in progress; so, the counter is updated (decremented), and program execution moves to the isolated communications (UART) watchdog timer processing routine located at merge point A in Figure 17. However, if the soft-stop counter is equal to zero (expiration), the REFDAC must be decremented to continue the downward output voltage ramp. In this case, the soft-stop counter is initialized, and the REFDAC data value is compared to the output voltage target value. If the REFDAC data value is greater than the output voltage target value, the REFDAC value is reduced and written to the REFDAC. If the REFDAC value is less than the output voltage value, the output voltage must be at zero, at which point soft stop is complete, and program execution switches to the disabled kernel process state.

If a 100 µs time tick is not asserted, only routine updates are performed. These updates consist of acquiring and processing analog parameters and threshold testing input voltage for faults. The converted V_{SENSE} result is filtered by the *mAdvFilterProcess (VSENSE,6)* function, an advanced filter with a programmable cutoff frequency determined by the number in parentheses (valid cutoff frequency range is 0 to 8).

The *mFilterProcess(TEMP)* function averages the temperature sensor output signal, followed by limit checking by the *Kernel_TEMP_Threshold_Process.* The network interface is then processed, followed by measurement of the supply input voltage. In an isolated system, primary-side data is obtained via UART by the kernel packet processor function *mUINPacketProcess()*. The values received are tested against upper and/or lower limits by the threshold process function, *Kernel_UIN_Threshold_Process*. The packet communication is completed by the UART done function, *mUINxDone()*. If the end application is non-isolated, supply input voltage (and other parameters) are digitized ADC0, then filtered (averaged) to minimize error due to noise. The resulting values are then threshold-checked to ensure they are within limits. The kernel state control process state is then called to manage any fault conditions.

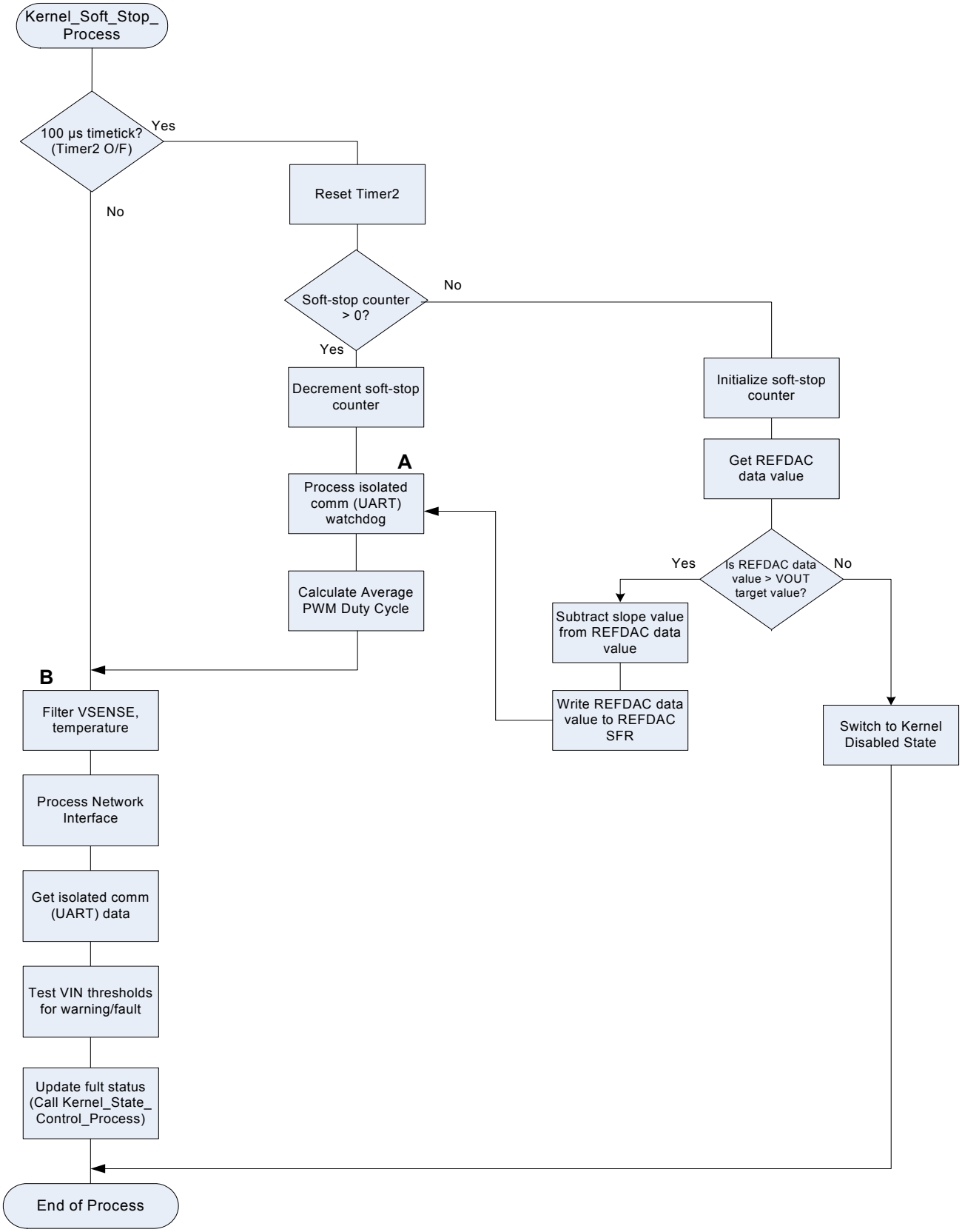

Figure 17. Kernel_Soft_Stop_Process Flowchart

4.14. Kernel Disabled Process (*Kernel_Disabled_Process*)

4.14.1. Description and Operation

This kernel process state terminates power system operation, checks for faults, and updates status flags accordingly. This kernel state will also pass control to the high level init state for system restart attempts if the appropriate conditions are met. Program execution begins with a software flag update to indicate the system has been turned off. ADC1 is then disabled and the software bypass enabled, effectively forcing all DPWM outputs to their OFF (safe) states. This is followed by routine service to the network interface. The V_{SENSE} and temperature sensor signals are then digitized by ADC0 and filtered by the functions, *mFilterProcess (VSENSE)* and *mFilterProcess (TEMP*), respectively. System input voltage values for network interface reporting are acquired next. In an isolated system, primary-side data is obtained via UART by the kernel packet processor function, *mUINPacketProcess()*. Packet communication is completed by the UART done function, *mUINxDone()*. If the end application is non-isolated, supply input voltage (and other parameters) are digitized ADC0, then filtered (averaged) to minimize error due to noise. The next operations performed are synchronized to the 100 µs timetick.

If a 100 µs time tick is asserted, interrupting source Timer2 is reset, and the isolated communications (UART) watchdog is processed. Next, the running-average duty cycle data accumulated during operation is cleared and execution branches to reentry point A in Figure 18.

If 100 µs time tick is not asserted, the current kernel process state decides the next process state to be called based on examination of the presence or absence of an isolated communications fault and the states of the internal and external enable signals. A switch to the high level init process state (to attempt restart) will be made only if the following three conditions are true: first, there is no isolated communications (UART) fault (if the fault is present, program execution is switched to the lockout process state); second, the internal enable bit is unmasked (enabled) and in the ON state; third, the external enable (if used) is unmasked and in the ON state.

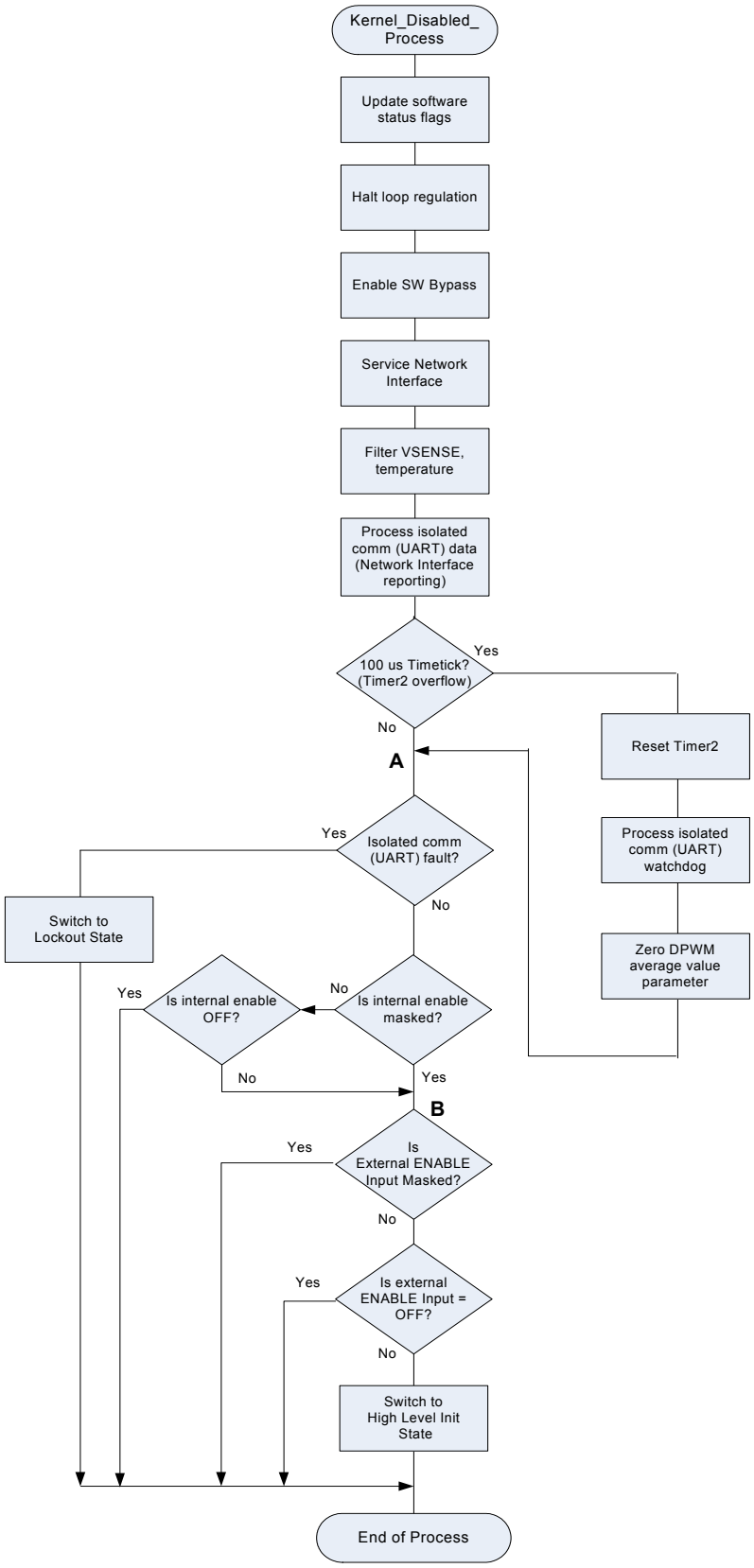

Figure 18. Kernel_Disabled_Process Flowchart

4.15. Kernel Lockout Process (*Kernel_Lockout_Process*)

4.15.1. Description and Operation

This kernel process state maintains key system functions, such as monitoring enable signals for change and servicing network interface communications, while the system power converter is shut down. The start of this kernel process state is identical to the start of the disabled process state allowing either of these two process states to halt converter operation. Program execution begins with termination power system operation. A software flag update is performed to indicate the system has been turned off. ADC1 is then disabled and the software bypass enabled, forcing all DPWM outputs to their OFF (safe) states. This is followed by routine service to the network interface. The V_{SENSE} and temperature sensor signals are then digitized by ADC0 and filtered by the functions, *mFilterProcess (VSENSE)* and *mFilterProcess (TEMP*), respectively. System input voltage values for network interface reporting are acquired next. In an isolated system, primary-side data is obtained via UART by the kernel packet processor function, *mUINPacketProcess(),* after receive data is available as checked by the *mIsUARTRxRdy()* function. The packet communication is completed by the UART done function, *mUINxDone()*. If the end application is non-isolated, supply input voltage (and other parameters) are digitized ADC0, then filtered (averaged) to minimize error due to noise. The next operations performed are synchronized to the 100 µs timetick.

If a 100 µs time tick is asserted, interrupting source Timer2 is reset, and the isolated communications (UART) watchdog is processed. Next, the running-average duty cycle data accumulated during operation are cleared and execution branches to reentry point A in Figure 19.

If 100 µs time tick is not asserted, program execution falls through to common program merge point A in Figure 19. Here, internal and external enable signals are tested to determine if they are unmasked and in the ON state. If either signal tests true, a switch to the disabled state is made to allow converter restart attempts.

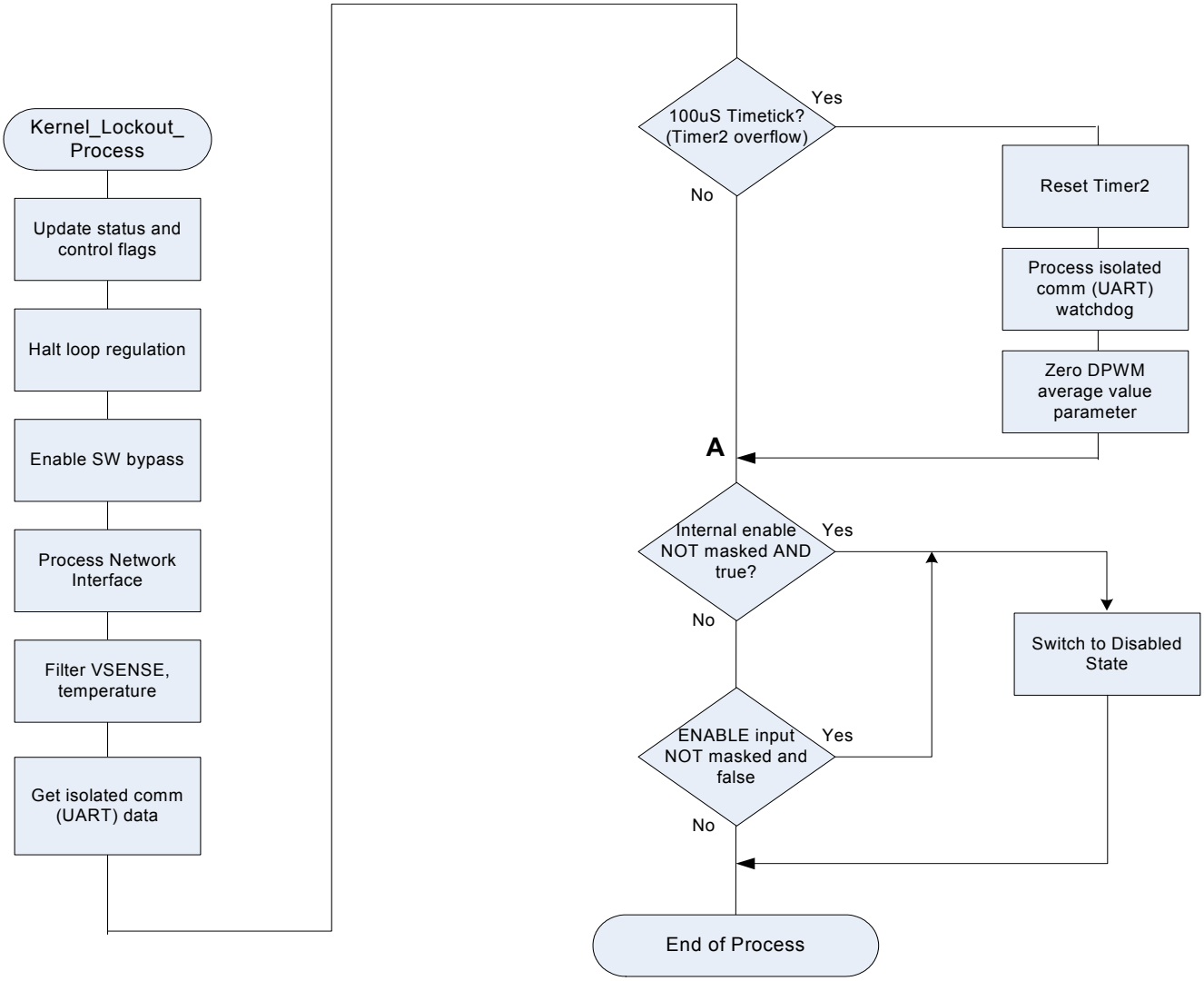

Figure 19. Kernel_Lockout_Process Flowchart

4.16. Kernel Fault Recovery Initialization Process (*Kernel_Recovery_Init_Process*)

4.16.1. Description and Operation

This kernel process state manages restart counters and flags for detected faults in preparation for fault recovery. The flowchart shown in Figure 21 is a decision tree composed of seven individual decision cells each having the basic flow shown in the example of Figure 20. There is one decision cell for each of the following faults: V_{OUT} overvoltage, V_{OUT} undervoltage, V_{IN} overvoltage, V_{IN} undervoltage, over temperature, under temperature, and t_{ONMAX}. Per the network interface specification, any given fault is allowed seven restart attempts before action is taken to implement fault recovery.

The example decision cell shows the overvoltage fault Figure 20. Program execution can bypasses service to the fault bit (i.e., ignore the fault) if so programmed via the network interface. When so programmed, execution branches to the next fault service routine in the tree. If not bypassed, the fault flag (e.g., V_{OUT} over voltage fault) is tested for logic 1. If the results of this test are true, the fault restart counter is checked for a value between 1 and 7 inclusive, per network interface specs. If it is in this range, the restart counter is updated (decremented), and program execution advances to the next fault in the tree. However, if the fault bit is still true *and* the restart counter reaches the count of zero, program execution vectors to the lockout process state where the system is shut down.

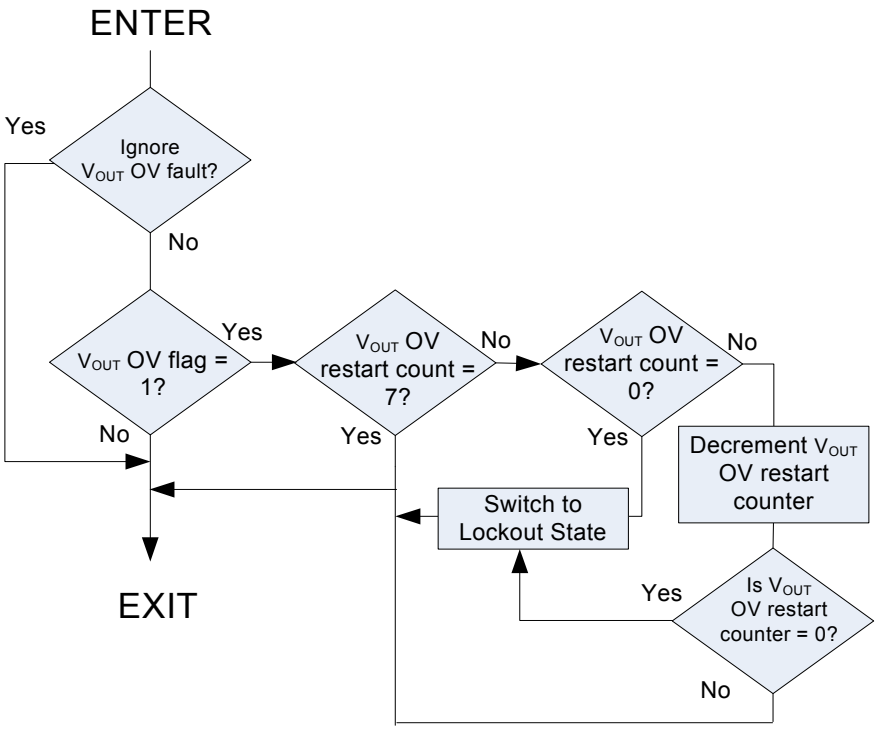

Figure 20. Fault Recovery Decision Cell

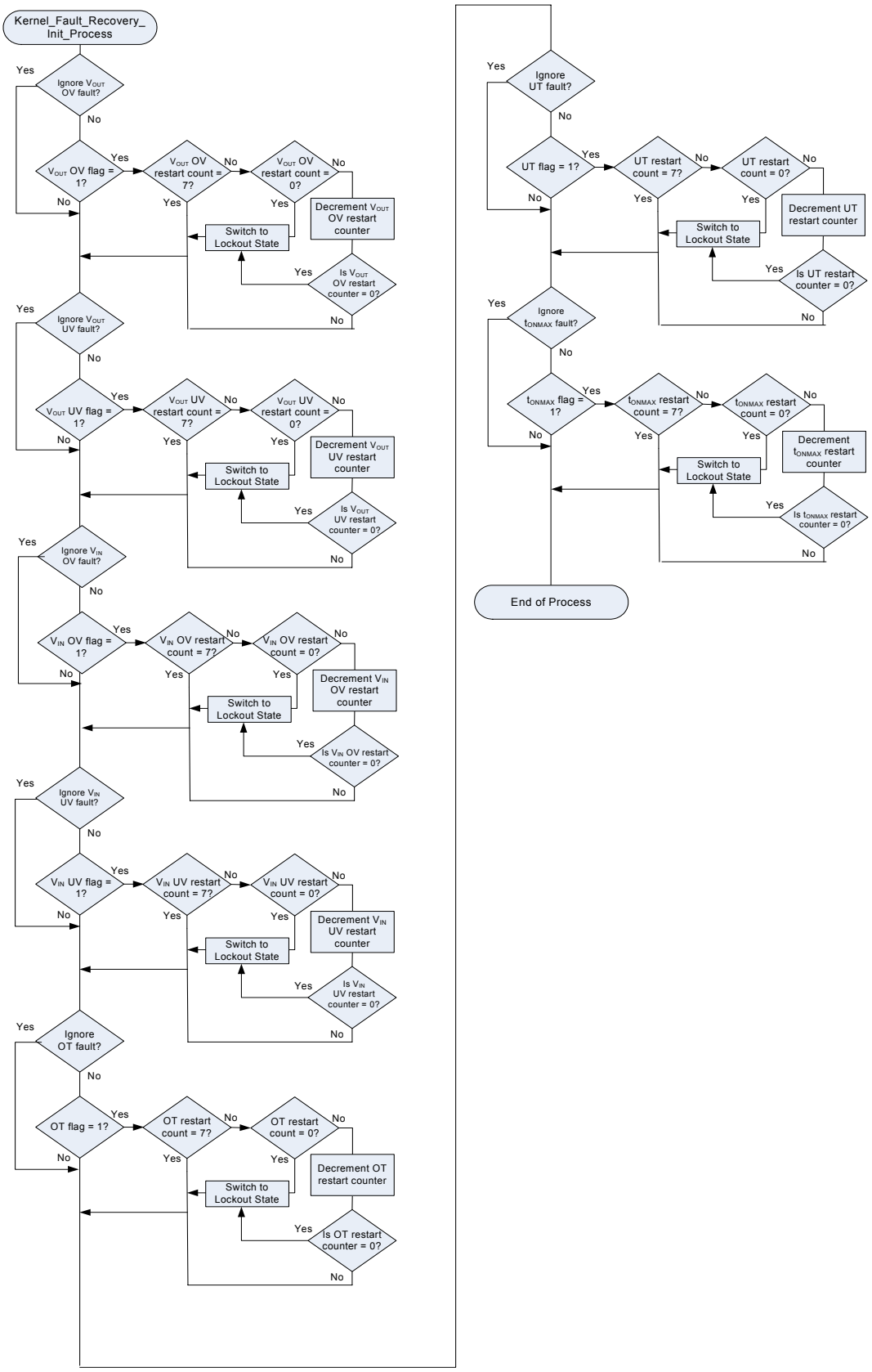

Figure 21. Kernel_Fault_Recovery_Init_Process Flowchart

4.17. Kernel Fault Recovery Process (*Kernel_Recovery_Process*)

4.17.1. Description and Operation

This kernel process state verifies the fault, then directs program execution to the appropriate kernel process state. A switch is made to the lockout or stop init process states if the fault requires system shutdown. A switch is made to the validation init process state if the fault type specifies restart attempts. It also performs routine system maintenance (network interface service, threshold-testing, etc.).

Program execution begins with the conversion of the V_{SENSE} and onboard temp sensor signal filtering the results from each and resetting their associated fault flags, then performing a threshold operation on both to determine if either is at a warning or fault level. The next operations performed are synchronized to the 100 µs timetick.

If a 100 µs time tick is asserted, interrupting source Timer2 is reset, and the isolated communications (UART) watchdog is processed. Next, the running-average duty cycle data accumulated during operation are cleared, and main loop execution resumes with routine network interface maintenance.

If 100 µs time tick is not asserted, Program execution falls through to the network interface maintenance reentry point.

Next, input voltage is converted and threshold-checked. In an isolated system, primary-side data is obtained via UART by the kernel packet processor function, *mUINPacketProcess()*. The packet communication is completed by the UART done function, *mUINxDone()*. If the end application is non-isolated, supply input voltage (and other parameters) are digitized ADC0, then filtered (averaged) to minimize error due to noise.

If no faults arise from the above processing, startup is permissible, and execution switches to the kernel validation process state. If faults are present *and* the internal *or* external enable signals are unmasked and true (i.e., “activated” (unmasked) and in their ON state), the system must be shut down, and program execution vectors to the stop init process state.

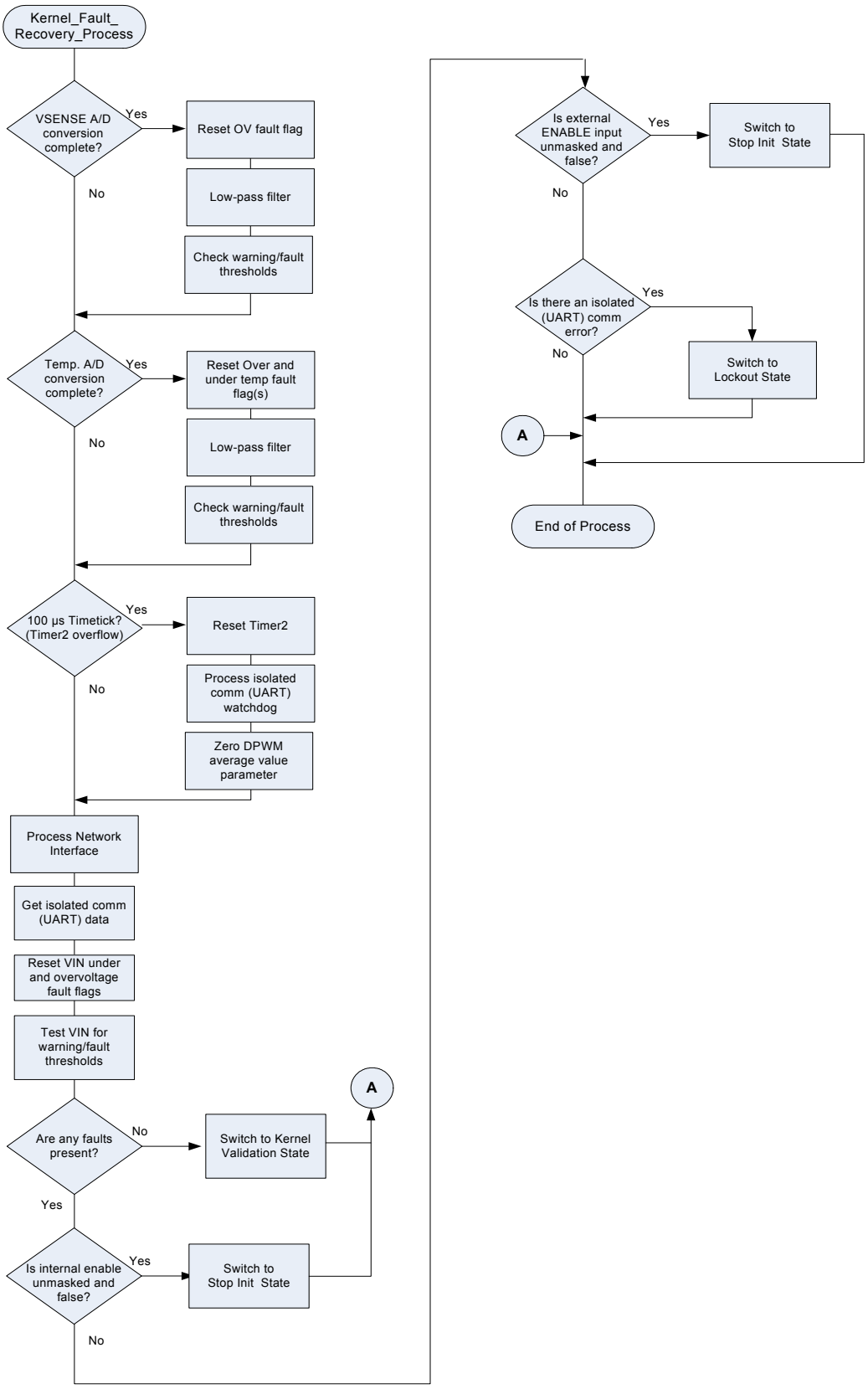

Figure 22. Kernel_Fault_Recovery_Process Flowchart

4.18. Kernel State Control Process (*Kernel_State_Control_Process*)

4.18.1. Description and Operation

This kernel process state executes the action associated with a fault. It checks all fault status flags and takes the appropriate action (e.g., system shutdown and/or process state change). This process state is called by other process states as a service function.

Starting at the top of Figure 23, the first seven decision cells service the isolated communications watchdog, input overcurrent, input overvoltage, input undervoltage, over-temperature, and under-temperature faults. Output overvoltage and undervoltage faults are also processed in the same way but only when the system is not in startup. As in the fault recovery init process state of “4.17. Kernel Fault Recovery Process (Kernel_Recovery_Process)”, each of these faults can be optionally ignored if so programmed by the network interface. In each case and if not ignored, the presence of any of these faults will enable output bypass forcing the DPWM outputs into their OFF (safe) states, update the system OFF status flag, and switch program execution to the fault recovery init process state.

Execution is immediately switched to the stop init state when the input voltage is at or below the undervoltage lockout (UVLO) threshold or when either the external or internal enable signals are unmasked and false (OFF). The last decision cell in the tree allows the output voltage to be programmed over the network interface, except when the system is in startup.

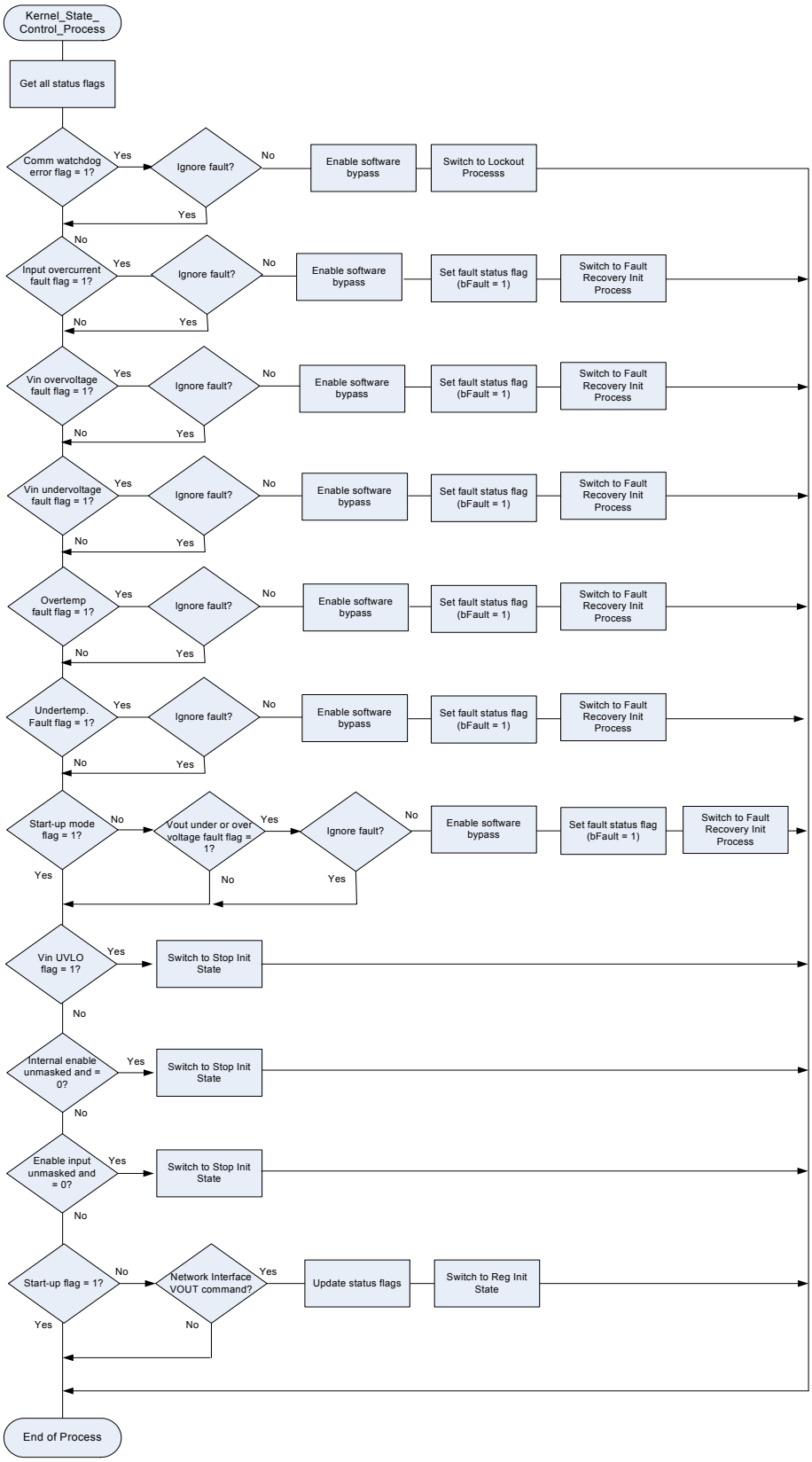

Figure 23. Kernel_State_Control_Process Flowchart

4.19. Kernel Threshold Process (Kernel_Parameter_Threshold_Process)

4.19.1. Description and Operation

This group of service processes supports fault detection for all other kernel process states. There are separate threshold processes for each parameter source (i.e., each UART packet and ADC0 AMUX channel). Threshold processes compare the value of the specified parameter to one or more static thresholds and update the associated software status flag(s) with the comparison results.

A typical over/under fault/warning threshold process decision cell is shown in Figure 24. The parameter to have thresholds checked is specified as part of the function call. For example, the correct function call to threshold-check the temperature sensor is: *Kernel_**TEMP**_Threshold_Process*.

As shown, the specified parameter is first compared to static fault settings provided by the user's software, followed by the same operation to check warning thresholds. Each comparison sets the appropriate status flag, then returns to the calling function.

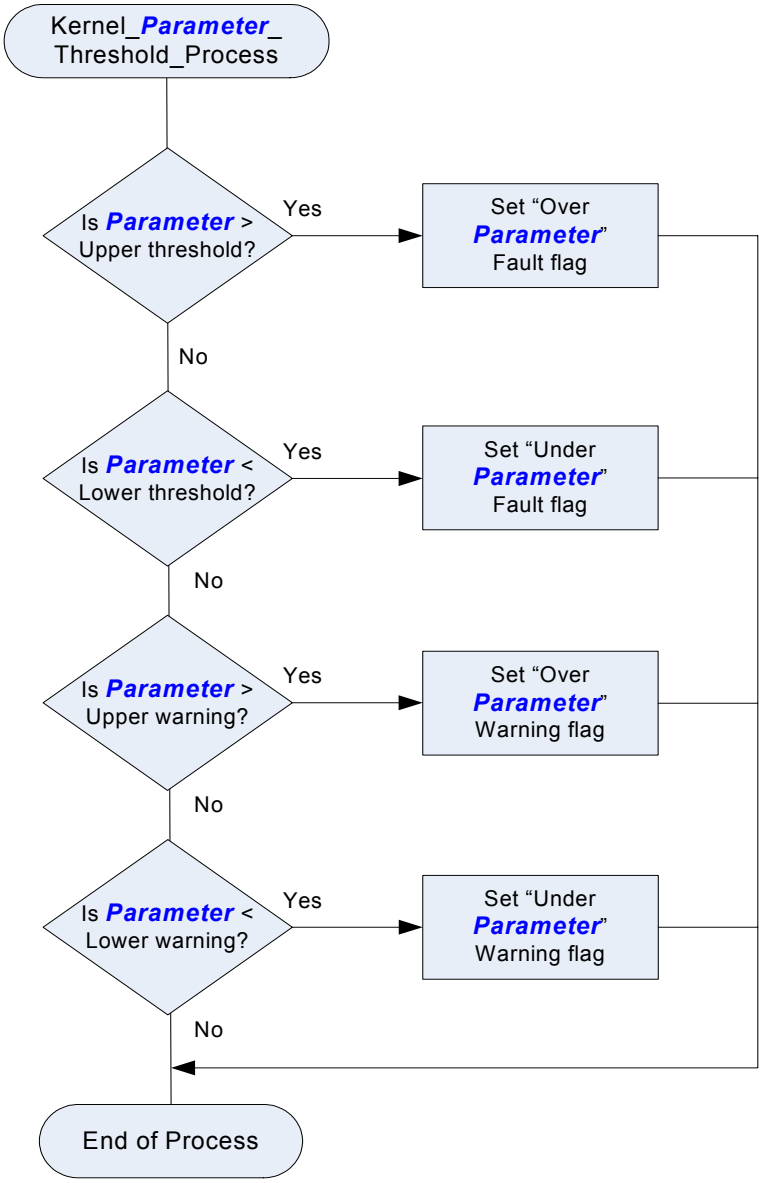

Figure 24. Example Threshold_Process Parameter Flowchart

APPENDIX A—LIBRARY FUNCTIONS

Functions

Table 5. Kernel Functions Summary

Function	Description
Balance_Primary_Process	For half-bridge applications. A closed-loop feedback loop function that maintains the voltage at the capacitive input divider node at a value of $V_{IN}/2$ by adjusting the pulse width offset of one of the primary switching control phases.
Active_Dead_Time_Process	Optimizes efficiency by minimizing the value of the averaged u(n) for any given load.
Feed_Forward_Process	Implements V_{IN} feed-forward operation. Maintains a constant loop gain by adjusting DSP Filter Engine coefficient A3 with changes in input voltage.
Indirect_Soft_Transition_Process()	Provides closed-loop soft start ramp linearity regulation.
mAdvFilterProcess**(operand, n)**	Advanced filter function. The operand is processed by the filter the time constant (cutoff frequency) determined by the value of n, where $0 \leq n \leq 8$.
mFilterProcess**(operand)**	Software filters (averages) the operand (digitized analog parameter).
mIsInputRdy**(operand)**	Checks ADC0 for operand data conversion—returns a "1" if conversion is complete.
mIsUARTRxRdy()	Checks the UART—returns a "1" if receive data is ready.
mResetInput**(operand)**	Clears the software filter associated with operand.
mUINClearWtDog	Resets the isolated communication watchdog timeout period.
mUINDone()	Checks the UART for end-of-packet processing—returns a "1" if processing is finished.
mUINIsWtDogErr()	Checks for an isolated communications watchdog error and returns a "1" if error is present.
mUINPacketProcess()	Extracts and stores parameters (packets) from the UART receive data channel.
mUINWtDogProcess()	Updates the isolated communications (UART) watchdog timer decrementing the internal software counter.
PMBus_Process()	Processes network interface requests and executes actions (if required). Also manages the network interface-associated memory, I/O and other resources.
Transient_Threshold_Process	Fine-tunes the threshold detector threshold setting with increasing input voltage.

As previously mentioned, there are approximately 500 individual functions in the kernel library, all of which are based on the functions listed in Table 5. The large number of functions is the result of multiple instantiations of the functions listed above. For example, there are eight external analog inputs on ADC0, each channel of which has eight different filters for a total of 64 filters. The following pages provide an overview of the functions listed in Table 5.

Balance_Primary_Process()

Description: *Balance_Primary_Process()* is a feedback algorithm that regulates the capacitive input node of a half-bridge dc-dc converter to a value of $V_{IN}/2$. It regulates by sampling the average capacitive node voltage and comparing it to a reference value of the averaged input voltage divided by two. The difference between these two terms is used to adjust duty cycle offset using the trim registers of primary switching phase PH1 until the node voltage achieves a value of $V_{IN}/2$.

Entry Parameters: Average V_{IN}, Average capacitive node voltage

Exit Parameters: Increment/decrement or hold of trim variable TLCD1

Functional Flowchart:

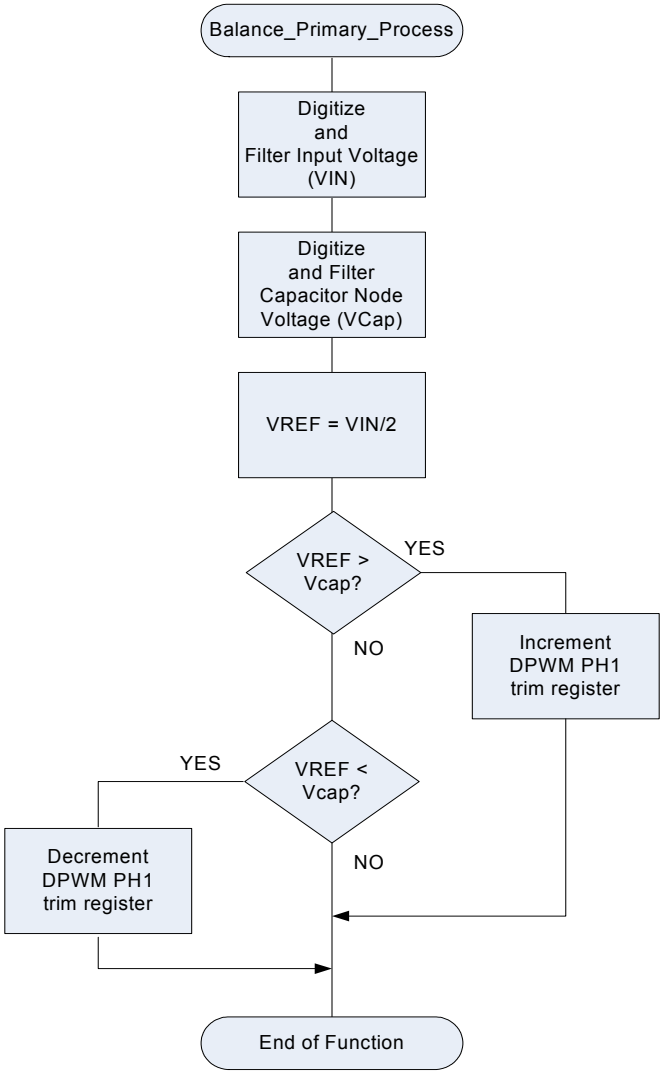

Figure 25. Balance_Primary_Process Flowchart

Dead_Time_Control_Process()

Description: The body diode of a synchronous rectifier (synchro) acts as a conventional diode when the synchro is off. During this time, the power dissipated by the body diode is approximately 0.7 V x I_{OUT}. For example, at an output current of 20 A, the power dissipated in the diode is approximately 14 W. This is in stark contrast to the power dissipated by the synchro when on; assuming a 10 mΩ R_{DSON}, the synchro power dissipation is 10 mΩ x (20 A)^2 = 4 W. For high-efficiency operation, the "free-wheeling" time of this body diode must be minimized. The Dead_Time_Control_Process() is a feedback algorithm for a half-bridge converter that adjusts synchronous rectifier "free-wheeling" (dead) time by writing to the PH timing control registers to minimize the average value of compensated duty cycle ratio u(n) for any given load. This is a valid method to optimize dead time since u(n) increases with losses at a given load point.

Entry Parameters: Average u(n)

Exit Parameters: PH3, PH4 trailing-edge time registers

Indirect_Soft_Transition_Process()

Description: This process regulates the linearity of the soft-start ramp. Closed-loop linearity control is implemented by measuring the average slope of the soft-start ramp, then comparing it to a slope reference. Software adjusts the REFDAC to drive the difference between the two control terms to zero.

Entry Parameters: Average V_{OUT}, slope reference term (single byte)

Exit Parameters: REFDAC variable update

Functional Flowchart:

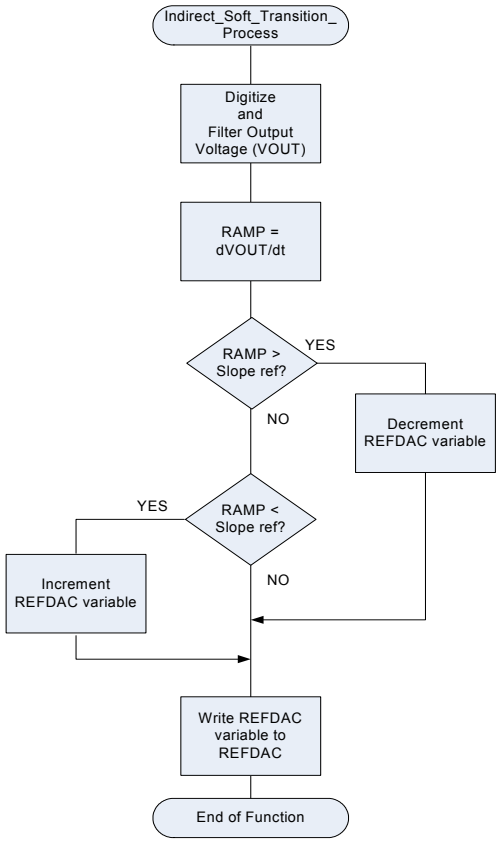

Figure 26. Soft_Transition_Process Flowchart

mAdvFilterProcess(*operand, n*)

Description: The mAdvFilterProcess() is an advanced discrete time filter algorithm that processes the operand with cutoff frequency determined by n, where n is a value between 0 and 8 inclusive.

Entry Parameters: Operand, n

Exit Parameters: Filtered operand

mFilterProcess(*operand*)

Description: *mFilterProcess()* is an advanced discrete time filter algorithm that processes the operand with fixed cutoff frequency determined by the associated default macro (#define statement).

Entry Parameters: Operand

Exit Parameters: Filtered operand

mIsInputReady(*operand*)

Description: This single-bit test checks for ADC0 data conversion complete for the specified channel (operand). This function returns a “1” if new data is available or a “0” if conversion is not yet complete. This is not a C-language function call; it is a single-bit test.

Entry Parameters: None

Exit Parameters: 1 if new data is available, 0 if new data is not available.

mIsUARTRxRdy()

Description: This single-bit test checks the Si825x UART receive buffer status soft flag for new receive data available. This function returns a “1” if new data is available or a “0” if conversion is not yet complete. This is not a C-language function call; it is a single-bit test.

Entry Parameters: None

Exit Parameters: 1 if new data is available, 0 if new data is not available.

mResetInput(*operand*)

Description: This function clears the data accumulation in the filter associated with the operand. It is electrically equivalent to shorting the capacitor in a simple RC low-pass filter.

Entry Parameters: Operand

Exit Parameters: None

mUINClearWtDog()

Description: The isolated communications (UART) timer runs as a background task. Its function is to assert a fault flag if isolated communication ceases for a time period greater than or equal to the watchdog timeout period. This function effectively “strobes” the watchdog, resetting its timeout period.

Entry Parameters: None

Exit Parameters: Watchdog timer period reloaded

mUINDone()

Description: This function is a handshake (internal state flag) indicating that the current packet processing has been completed and processing of the next packet can commence.

Entry Parameters: None

Exit Parameters: None

mUINIsWtDogErr()

Description: This function tests the isolated communications (UART) watchdog error flag and returns a 0 if the watchdog has not timed-out (indicating no error); or a “1” indicating the watchdog has timed-out and an error is present.

Entry Parameters: None

Exit Parameters: Fault flag update: “0” if no watchdog error (normal operation) or “1” if the watchdog has timed-out (error condition).

mUINPacketProcess()

Description: This function is the isolated communication (UART) data packet processor to guarantee data delivery and integrity and guarantees that data communication faults are recoverable within reason.

- For the purposes of the kernel, the Si825x is operated as a client, and the primary-side MCU is operated as the server. Both sides of the communication channel are watchdog-protected. These watchdogs ensures that data is transmitted in a timely manner.
- Data integrity is ensured by the 1s complement checksum, which calculates the sum of the data in the string.
- Data recoverability is provided by packet stuffing based on a robust protocol that provides deterministic start and stop states.

Entry Parameters: None

Exit Parameters: None

Packet Format: Contact Silicon Labs Power Applications support for information.

mUINWtDogProcess()

Description: This function performs the background processing tasks for the isolated communications (UART) watchdog timer. It updates internal counters based on data received by the UART to ensure all packets have been received within the appropriate time. Counters are managed such that no overflow occurs if serial communications are operating normally, but that overflow does occur if a communication fault is present. This is an internal process that should be called periodically.

Entry Parameters: None

Exit Parameters: None

Transient_Process()

Description: This function closes a second software loop around the DSP Filter Engine coefficients to aggressively control transient events. During a transient, the coefficients will be modified to reduce the transient magnitude and pull in the transient settling time.

Entry Parameters: ADC1DAT

Exit Parameters: None

APPENDIX B—HEADER FILE

The header (.h) file contains initialization data for kernel hardware and software parameters (H = hardware S = software).

Parameter	H/S	Description
KERNEL_POWER_CONFIG	H	Network interface power configuration byte
KERNEL_WRITE_PROTECT	H	Network interface write-protect byte
KERNEL_MODE	S	Network interface exponent for V_{OUT} format command
KERNEL_VOUT	S	Network interface V_{OUT} command
KERNEL_VOUT_TRIM	S	Network interface V_{OUT} trim command
KERNEL_VOUT_CAL	S	Network interface V_{OUT} calibration command
KERNEL_VOUT_MAX	S	Network interface V_{OUT} max command
KERNEL_VOUT_HIGH	S	Network interface V_{OUT} margin high command
KERNEL_VOUT_LOW	S	Network interface V_{OUT} margin low command
KERNEL_TRANS_RATE	S	Network interface soft-start transition phase rate command*
KERNEL_VOUT_DROOP	S	Network interface droop command*
KERNEL_VOUT_SCALE	S	Network interface V_{OUT} scaling (gain calibration) command
KERNEL_VOUT_OFFSET	S	Network interface V_{OUT} offset command*
KERNEL_POUT_MAX	S	Network interface maximum output power command*
KERNEL_MAX_DUTY	S	Network interface maximum duty cycle power command*
KERNEL_FREQUENCY	H	Network interface DPWM frequency (implemented as read-only)
KERNEL_VIN_ON	S	Network interface input voltage ON threshold command
KERNEL_VIN_OFF	S	Network interface input voltage OFF threshold command
KERNEL_INTERLEAVE	S	Network interface interleave command*
KERNEL_IOUT_SCALE	S	Network interface I_{OUT} scaling (gain calibration) command
KERNEL_IOUT_OFFSET	S	Network interface I_{OUT} offset command*
KERNEL_VFAN1	S	Network interface fan#1 command–*
KERNEL_VFAN2	S	Network interface fan#2 command*
KERNEL_VOUT_OV_FAULT	S	Network interface V_{OUT} overvoltage fault command
KERNEL_VOUT_OV_CTL	S	Network interface V_{OUT} overvoltage control parameter command
***Note:** Provision, but not implemented.		

Parameter	H/S	Description
KERNEL_VOUT_OV_WARN	S	Network interface V_{OUT} overvoltage warning command
KERNEL_VOUT_UV_WARN	S	Network interface V_{OUT} undervoltage warning command
KERNEL_VOUT_UV_FAULT	S	Network interface V_{OUT} undervoltage fault command
KERNEL_VOUT_UV_CTL	S	Network interface V_{OUT} undervoltage control parameter command
KERNEL_IOUT_OC_FAULT	S	Network interface I_{OUT} overcurrent fault command
KERNEL_IOUT_OC_CTL	S	Network interface I_{OUT} overcurrent control parameter command
KERNEL_IOUT_OC_LV_FAULT	S	Network interface I_{OUT} overcurrent and V_{OUT} low voltage fault command*
KERNEL_IOUT_OC_LV_CTL	S	Network interface I_{OUT} over current and V_{OUT} low voltage control parameter command*
KERNEL_IOUT_OC_WARN	S	Network interface I_{OUT} over current warning command
KERNEL_IOUT_UC_FAULT	S	Not implemented
KERNEL_IOUT_UC_CTL	S	Not implemented
KERNEL_POUT_FAULT	S	Not implemented
KERNEL_POUT_CTL	S	Not implemented
KERNEL_TEMP_OT_FAULT	S	Network interface over temperature fault command
KERNEL_TEMP_OT_CTL	S	Network interface over temperature control parameter command
KERNEL_TEMP_OT_WARN	S	Network interface over temperature warning command
KERNEL_TEMP_UT_WARN	S	Network interface under temperature warning command
KERNEL_TEMP_UT_FAULT	S	Network interface under temperature fault command
KERNEL_TEMP_UT_CTL	S	Network interface under temperature control parameter command
KERNEL_VIN_OV_FAULT	S	Network interface V_{IN} over voltage fault command
***Note:** Provision, but not implemented.		

Variable	H/S	Description
KERNEL_VIN_OV_CTL	S	Network interface VIN over voltage control parameter command
KERNEL_VIN_OV_WARN	S	Network interface VIN over voltage warning command
KERNEL_VIN_UV_WARN	S	Network interface VIN under voltage warning command
KERNEL_VIN_UV_FAULT	S	Network interface VIN under voltage fault command
KERNEL_VIN_UV_CTL	S	Network interface VIN under voltage control parameter command
KERNEL_IIN_OC_FAULT	S	Not implemented
KERNEL_IIN_OC_CTL	S	Not implemented
KERNEL_IIN_OC_WARN	S	Not implemented
KERNEL_TON_DELAY	S	Network interface t_{ON} Delay command
KERNEL_TON_RISE	S	Network interface t_{ON} Rise command
KERNEL_TON_MAX	S	Network interface t_{ON} Max command
KERNEL_TON_MAX_CTL	S	Network interface t_{ON} Max control parameter command
KERNEL_TOFF_DELAY	S	Network interface t_{OFF} Delay command
KERNEL_TOFF_FALL	S	Network interface t_{OFF} Fall command
KERNEL_TOFF_MAX	S	Network interface t_{OFF} Max command
KERNEL_SOFT_SLOPE	S	Network interface Soft Slope command
KERNEL_SOFT_SCALE	S	Network interface Soft Scale command
KERNEL_FILT_KP	H	DSP Filter kP (Proportional) coefficient variable
KERNEL_FILT_KI	H	DSP Filter kI (Integral) coefficient variable
KERNEL_FILT_KD	H	DSP Filter kD (Differentiator) coefficient variable
KERNEL_FILT_A0	H	DSP Filter A0 (SINC filter gain) coefficient variable
KERNEL_FILT_A1	H	DSP Filter A1 (LPF Pole 1) coefficient variable
KERNEL_FILT_A2	H	DSP Filter A2 (LPF Pole 2) coefficient variable
KERNEL_FILT_A3	H	DSP Filter A3 (LPF filter gain term) coefficient variable
KERNEL_FILT_DEC	H	DSP Filter Decimation coefficient variable
KERNEL_PIDCN	H	DSP Filter configuration byte
KERNEL_DPWM_DPWMCN	H	DPWM configuration byte
KERNEL_DPWM_SW_CYC	H	DPWM switching cycle length byte
KERNEL_DPWM_PH_POL	H	DPWM phase initial polarity byte

Variable	H/S	Description
KERNEL_DPWM_ENABX_OUT	H	DPWM ENABLE bypass data byte
KERNEL_DPWM_OCP_OUT	H	DPWM over current protection bypass data byte
KERNEL_DPWM_SWBP_OUT	H	DPWM software bypass data byte
KERNEL_DPWM_SWBP_OUTEN	H	DPWM software bypass enable byte
KERNEL_DPWM_PH1_CNTL0	H	DPWM PH1 control register 0 byte
KERNEL_DPWM_PH1_CNTL1	H	DPWM PH1 control register 1 byte
KERNEL_DPWM_PH1_CNTL2	H	DPWM PH1 control register 2 byte
KERNEL_DPWM_PH1_CNTL3	H	DPWM PH1 control register 3 byte
KERNEL_DPWM_PH2_CNTL0	H	DPWM PH2 control register 0 byte
KERNEL_DPWM_PH2_CNTL1	H	DPWM PH2 control register 1 byte
KERNEL_DPWM_PH2_CNTL2	H	DPWM PH2 control register 2 byte
KERNEL_DPWM_PH2_CNTL3	H	DPWM PH2 control register 3 byte
KERNEL_DPWM_PH3_CNTL0	H	DPWM PH3 control register 0 byte
KERNEL_DPWM_PH3_CNTL1	H	DPWM PH3 control register 1 byte
KERNEL_DPWM_PH3_CNTL2	H	DPWM PH3 control register 2 byte
KERNEL_DPWM_PH3_CNTL3	H	DPWM PH3 control register 3 byte
KERNEL_DPWM_PH4_CNTL0	H	DPWM PH4 control register 0 byte
KERNEL_DPWM_PH4_CNTL1	H	DPWM PH4 control register 1 byte
KERNEL_DPWM_PH4_CNTL2	H	DPWM PH4 control register 2 byte
KERNEL_DPWM_PH4_CNTL3	H	DPWM PH4 control register 3 byte
KERNEL_DPWM_PH5_CNTL0	H	DPWM PH5 control register 0 byte
KERNEL_DPWM_PH5_CNTL1	H	DPWM PH5 control register 1 byte
KERNEL_DPWM_PH5_CNTL2	H	DPWM PH5 control register 2 byte
KERNEL_DPWM_PH5_CNTL3	H	DPWM PH5 control register 3 byte
KERNEL_DPWM_PH6_CNTL0	H	DPWM PH6 control register 0 byte
KERNEL_DPWM_PH6_CNTL1	H	DPWM PH6 control register 1 byte
KERNEL_DPWM_PH6_CNTL2	H	DPWM PH6 control register 2 byte
KERNEL_DPWM_PH6_CNTL3	H	DPWM PH6 control register 3 byte
KERNEL_DPWM_DPWMTLLT0	H	DPWM u(n) lower limit control byte0

Variable	H/S	Description
KERNEL_DPWM_DPWMTLGT0	H	DPWM u(n) upper limit control byte0
KERNEL_DPWM_DPWMTLLT1	H	DPWM u(n) lower limit control byte1
KERNEL_DPWM_DPWMTLGT1	H	DPWM u(n) upper limit control byte1
KERNEL_DPWM_DPWMTLLT2	H	DPWM u(n) lower limit control byte2
KERNEL_DPWM_DPWMTLGT2	H	DPWM u(n) upper limit control byte2
KERNEL_DPWM_DPWMTLLT3	H	DPWM u(n) lower limit control byte3
KERNEL_DPWM_DPWMTLGT3	H	DPWM u(n) upper limit control byte3
KERNEL_DPWM_DPWMULOCK	H	DPWM symmetry lock control byte
KERNEL_DPWM_DPWMTLCD0	H	DPWM u(n) trim correction data byte0
KERNEL_DPWM_DPWMTLCD1	H	DPWM u(n) trim correction data byte1
KERNEL_DPWM_DPWMTLCD2	H	DPWM u(n) trim correction data byte2
KERNEL_DPWM_DPWMTLCD3	H	DPWM u(n) trim correction data byte3
KERNEL_OCP_IPKCN	H	Peak current detector control byte
KERNEL_OCP_LEBCN	H	Peak current leading-edge blanking time control byte
KERNEL_OCP_OCPCN	H	Peak current detector over-current protection counter control byte
KERNEL_GAIN_RD_CAL_IIN	S	ADC0 gain supply input current read calibration
KERNEL_OFFSET_RD_CAL_IIN	S	ADC0 offset supply input current read calibration
KERNEL_GAIN_WR_CAL_IIN	S	ADC0 gain supply input current write calibration
KERNEL_OFFSET_WR_CAL_IIN	S	ADC0 offset supply input current write calibration
KERNEL_GAIN_RD_CAL_IOUT	S	ADC0 gain supply output current read calibration
KERNEL_OFFSET_RD_CAL_IOUT	S	ADC0 offset supply output current read calibration
KERNEL_GAIN_WR_CAL_IOUT	S	ADC0 gain supply output current write calibration
KERNEL_OFFSET_WR_CAL_IOUT	S	ADC0 offset supply output current write calibration
KERNEL_GAIN_RD_CAL_TEMP1	S	ADC0 gain supply temperature read calibration
KERNEL_OFFSET_RD_CAL_TEMP1	S	ADC0 offset temperature read calibration
KERNEL_GAIN_WR_CAL_TEMP1	S	ADC0 gain temperature write calibration
KERNEL_OFFSET_WR_CAL_TEMP1	S	ADC0 offset temperature write calibration

APPENDIX C—RECOMMENDED C-CODE LEARNING RESOURCES

- VTC C Programming DVD course ($99 complete—buy online at **http://www.vtc.com/**)
- The Complete C Reference 4th Edition, Herbert Schildt, Osborne/McGraw Hill ISBN 0-07-212124-6

APPENDIX D—SYSTEM MANAGEMENT PROCESSOR PERIPHERAL INITIALIZATION

Peripheral	Initialization State
V_{DD} Monitor	Enabled
Watchdog Timer	Disabled
Oscillator	Internal Oscillator enabled/selected. F = 24.5 MHz
PLL	Enabled
P0.0	SCL
P0.1	SDA
P0.2	SMB Alert
P0.3	General purpose digital I/O*
P0.4	UART TX
P0.5	UART RX
P0.6	General purpose digital I/O*
P0.7	ENABLE Input
P1.0	General purpose analog/digital I/O*
P1.1	General purpose analog/digital I/O*
P1.2	ADC0 input (average output current)
P1.3	General purpose analog/digital I/O*
P1.4	General purpose analog/digital I/O*
P1.5	General purpose analog/digital I/O*
P1.6	General purpose analog/digital I/O*
P1.7	C2D
Port0 I/O	Bits 7, 6, 4 = push/pull. Others = open drain
Port0 Skip	Bits 7, 6 skipped
Port0 Data	0xFF
Port1 I/O	Bit 6 = push/pull. Others = open drain
Port1 Skip	Bits 7 through 0 skipped
Port1 Data	0xFF

Peripheral	Initialization State
ADC0	Enabled, dual tracking mode (2 cycles), Auto-sequencing disabled, VREF enabled, AIN2 threshold detector enabled.
Timer2	Enabled, split (dual 8-bit) mode, interrupts disabled. Low byte = reload value for ADC0, high byte = reload value for general purpose counter. FCLK = SYSCLK/12
UART0	RX enabled. RX routed to P0.5, Tx routed to P0.7, UART clk = TIMER1 (clocked by SYSCLK).
ENABLE Input	Disabled
Timer0	(SMBUs clock) 8-bit auto reload mode
SMBus	Enabled, slave mode, clock from Timer0
Reset Source	Power-on RESET flag
***Note:** Connected to switches, LEDs, and external connector on Si825x target board.	

System Management Processor Peripheral Initialization Code (From File DPSK_kernel.c)

```
// Init the kernel for operation
void Kernel_Init(void)
{
        unsigned char _index;

        //*********************************************************************
        VDM0CN |= 0x80;                                         // Enable the monitor
        //*********************************************************************


        //*********************************************************************
        PCA0MD = PCA0MD & 0xBF;                                 // Disable the WDT
        //*********************************************************************


        //*********************************************************************
        // Oscillator related init
        OSCICN = 0x87;                                          // Init the clocks, 24.5MHz
//      PLLCN = 0xC0;
        PLLCN = 0xC4;
        CLKSEL = 0x00;
        //*********************************************************************

        //*********************************************************************
        // General port init

        // P0.0         SCL
        // P0.1         SDA
        // P0.2         SMBA
        // P0.3         J8, pin 1
        // P0.4         UART (TX)
        // P0.5         UART (RX)
        // P0.6         Green LED
        // P0.7         Enable

        // P1.0         J8, pin 5
        // P1.1         J8, pin 6
        // P1.2         Average Iout
        // P1.3         J8, pin 2
        // P1.4         Switch (S4)
        // P1.5         Switch (S5)
        // P1.6         Red LED
        // P1.7         C2D

        P0MDOUT = 0xD0;                                         // 1101 0000
        P1MDOUT = 0x40;                                         // 0100 0000
        P0SKIP = 0xCC;                                          // 1100 1100
        P1SKIP = 0xFF;                                          // 1111 1111
        P1MDIN = 0xFB;                                          // 1111 1011
        P0 = 0xFF;                                                      // 1111 1111
        P1 = 0xFF;                                                      // 1111 1111
        XBR0 = 0x05;                                            // 0000 0101
        XBR1 = 0xC0;                                            // 1100 0000
        //*********************************************************************

        //*********************************************************************
        // Init ADC0 dependencies

        ADC0CN = 0x00;
        ADC0TK = 0xFF;                                          // Dual-Tracking (2 cycles)
//      ADC0TK = 0xF8;
        ADC0CF = ((24500000/2500000) << 3);     // Fclk/CLKSAR - 1 << 3

        REF0CN = 0x07;                                          // Enable Voltage Reference and Temp Sensor

        ADC0CN = 0x84;                                          // Enable ADC0,

        ADC0STA0 = 0;                                           // Remove any pending conversions
        ADC0STA1 = 0;

        mAINxDone();                                            // Clear the end of sample status
```

```
//      EIE2 |= 0x02;                                   // Enable threshold detection for AIN2
        ADC0LM0 &= ~0x04;
        bPeakCurrent = 0;
        //*********************************************************************

        //*********************************************************************
        // Init Timer 2
        TMR2CN = 0x08;                                  // Stop Timer 2; Clear TF2H and TF2L; disable low ;enable split mode;

        CKCON &= ~0x30;                                 //      Timer2 low and high bytes use SYSCLK/12 as their timebase.


        TMR2RLL = -(24500000/63912/12);         // Initialize the low byte reload value for ADC0
        TMR2RLH = -(24500000/10000/12);         // Initialize the high byte reload value for a general counter

        TMR2L = 0xFE;                                   // Set to reload
        TMR2H = 0xFE;                                   // Set to reload

        EIE2 &= ~0x20;                                  // Disable Timer2 interrupts (should not be enabled anyway)

        TMR2CN |= 0x04;                                 // Start Timer2
        //*********************************************************************

        //*********************************************************************
        // Init UART dependencies

        XBR0 |= 0x01;                                   // Enable the UART in the crossbar

        SCON0 = 0x10;                                   // ignor stop, clear flags

        TMOD &= ~0xF0;                                  // TMOD: timer 1 in 8-bit autoreload
        TMOD |= 0x20;

        TH1 = -(24500000/115200/2);             // Set Timer1 reload value

        CKCON |= 0x08;                                  // Timer 1 clocked from system clock (prescaler used by Timer 0 only)

        TL1 = 0xFF;                                     // initialize timer1 to reload value

        TR1 = 1;                                        // START Timer1

        mUINReset();                                    // Set the initial start state
        //*********************************************************************

        //*********************************************************************
        // Disable the ENABLE signal
        IT01CF = 0x88;
        TCON &= 0xF0;
        TCON |= 0x04;
        IE1 = 0;
        //*********************************************************************


        //*********************************************************************
        TMOD &= ~0x0F;                                  // TMOD: timer 0 in 8-bit autoreload
        TMOD |= 0x02;

        TH0 = -(24500000/100000);               // Set Timer0 reload value
        CKCON |= 0x04;                                  // Timer 0 clocked from system clock (prescaler used by Timer 0 only)
        TL0 = 0xFF;                                     // initialize timer1 to reload value
        TR0 = 1;                                        // START Timer1

        SMB0CF = 0x1C;                          // Enable slave mode, Tmr0 is the clock
        SI = 0;                                                         // Clear SMBus Interrupt Flag
        SMB0CF |= 0x80;                                 // Enable SMBus;
        //*********************************************************************

        //*********************************************************************
        RSTSRC = 0x02;                                  // Set power monitor as a reset
        //*********************************************************************
```

DOCUMENT CHANGE LIST

Revision 0.1 to Revision 0.2

- Added "Appendix D—System Management Processor Peripheral Initialization" on page 56.

Revision 0.2 to Revision 0.3

- Updated "Contact Information" on page 62.
  - Updated disclaimer.

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: MCUinfo@silabs.com
Internet: **www.silabs.com**

SILICON LABORATORIES