Single and Multistage Watchdog Timers

by PJim Lamberson |

SENSORAY | embedded electronics

Designed and Manufactured in the U.S.A.

SENSORAY | p. 503.684.8005 | email: info@SENSORAY.com | www.SENSORAY.com

7313 SW Tech Center Drive | Portland, OR 97223

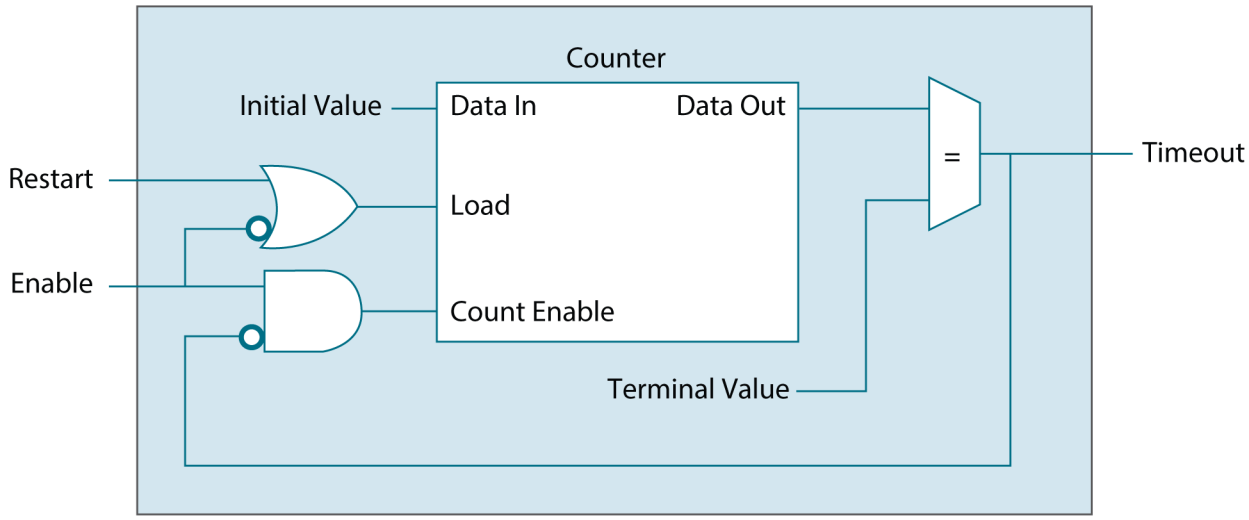
INTRODUCTION

A watchdog timer is an electronic circuit that initiates corrective action in response to a computer hardware malfunction or program error. It is an essential component of systems that are difficult or impossible to physically access because it provides a way to automatically recover from transient faults. Also, a watchdog timer can respond to faults more quickly than a human operator, making it invaluable in cases where a human operator would be too slow to react to a fault condition. Watchdog timers are widely used in embedded and remote systems, in equipment ranging from microwave ovens to Mars rovers.

Every watchdog timer, however simple or sophisticated, must initiate two corrective actions. First, it must set the computer's control outputs to safe levels so that potentially dangerous devices such as motors and heaters will not pose threats to people or equipment. This is a high priority action that must occur as soon as a fault is detected. After setting the outputs to safe levels, the next order of business is to restore normal system operation. This can be as simple as restarting the computer, as if a human operator has pressed the computer's reset pushbutton, or it may involve a sequence of actions that ultimately ends with a computer restart.

STRUCTURE AND OPERATION

In general, a watchdog timer (or just "watchdog") consists of a digital counter that counts from an initial value to a terminal value at a rate determined by a fixed-frequency clock. Typically the counter counts down from the initial value to zero, and the initial value is programmable so that the program can configure how long it takes to count to zero. The watchdog will "timeout" when the counter reaches zero, causing it to assert its timeout signal and halt counting. The timeout signal is connected to external circuitry so that it can initiate corrective action.



A program can restart the timer at any time by loading the initial value into the counter; this is commonly called "kicking" the watchdog. The watchdog is kicked by momentarily asserting its restart input (usually by writing to a watchdog control register). During normal operation, the application program regularly kicks the timer to keep it from reaching zero, typically as part of a control loop. If a fault condition prevents the program from kicking the timer, the watchdog will timeout and initiate corrective action.

Some watchdogs can be enabled and disabled by software, making it possible for a program to enable the watchdog only when its services are needed. Other watchdogs are automatically enabled upon system boot and cannot be disabled at all; these are typically used to detect and recover from boot faults. In some cases, a computer may employ both types of watchdogs. In theory, there is no upper limit to the number of watchdogs used in a computer.

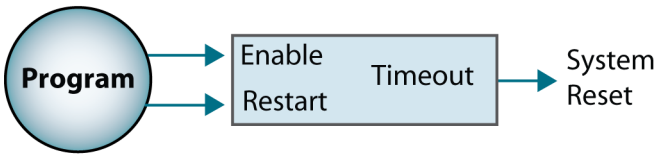
Software-enabled watchdogs are typically disabled upon system reset. When a control program starts executing, it enables its watchdog before it starts to control the output signals. Once the watchdog is enabled, the program must regularly kick the watchdog to prevent timeouts. When the application is preparing to terminate, it sets all outputs to safe states and ceases output control before disabling the watchdog; the application can terminate after disabling the watchdog.

WATCHDOG ARCHITECTURES

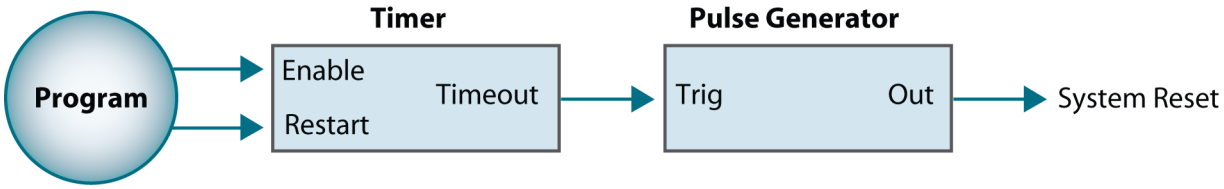
Three watchdog architectures are discussed here, each with progressively greater complexity and capabilities. The first is a simple, single-stage watchdog that unconditionally triggers a system reset. Next is a two-stage watchdog that provides an opportunity for program-managed recovery. Finally, a three-stage watchdog is described that allows for program-managed recovery and, failing that, it provides an opportunity to log debug information.

SIMPLE WATCHDOG

The most basic watchdog circuit has a single timer that invokes an immediate computer restart upon timeout. The timeout signal is connected to the computer's system reset input, either directly or through a conditioning circuit, so that a computer restart will occur when the watchdog times out. This architecture depends on the system reset to force control outputs to their safe states.



Some computers will power-down if a continuous timeout signal is applied to the system reset input. In such cases, a pulse may be required to initiate a system restart. A pulse generator can often be used to satisfy this requirement.

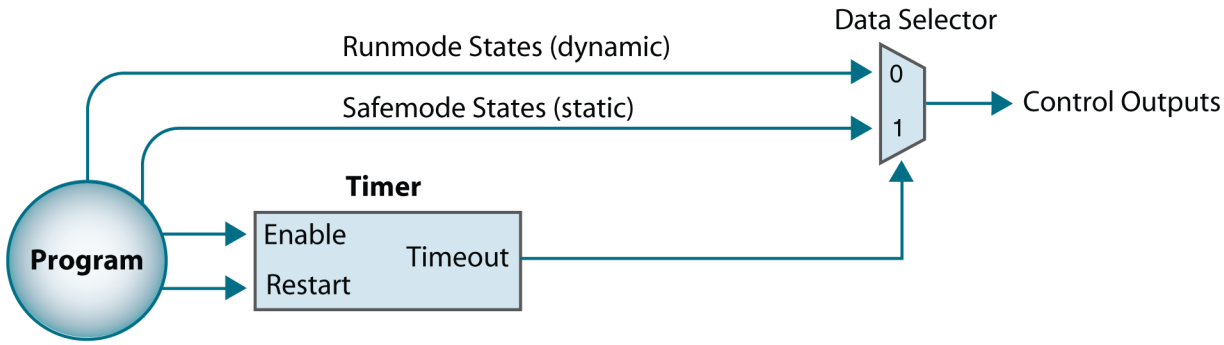


FAIL-SAFE SYSTEMS IN MULTISTAGE WATCHDOGS

It's been emphasized that a watchdog timeout must cause the computer's control outputs to promptly switch to safe states. This will happen automatically if the watchdog triggers an immediate computer restart (as in the previous example). However, a multistage watchdog timeout doesn't immediately restart the computer; it merely schedules a restart to occur at a future time. Consequently, a multistage watchdog must work in concert with special circuitry that will switch outputs to safe states upon timeout, prior to the computer restart.

One way to do this is to employ a dedicated control reset signal that resets the control circuitry (but not the computer) upon watchdog timeout. This is easily implemented but it presents some complications and shortcomings. For example, a control reset will restore outputs to their default power-up states, but the default states may be sub-optimal safe states, and the ideal safe states may even change as as a function of the overall system state. Also, a reset can cause the loss of important state information needed for fault recovery, and it may interfere with the operation of interfaces that could otherwise continue to function normally during a fault condition.

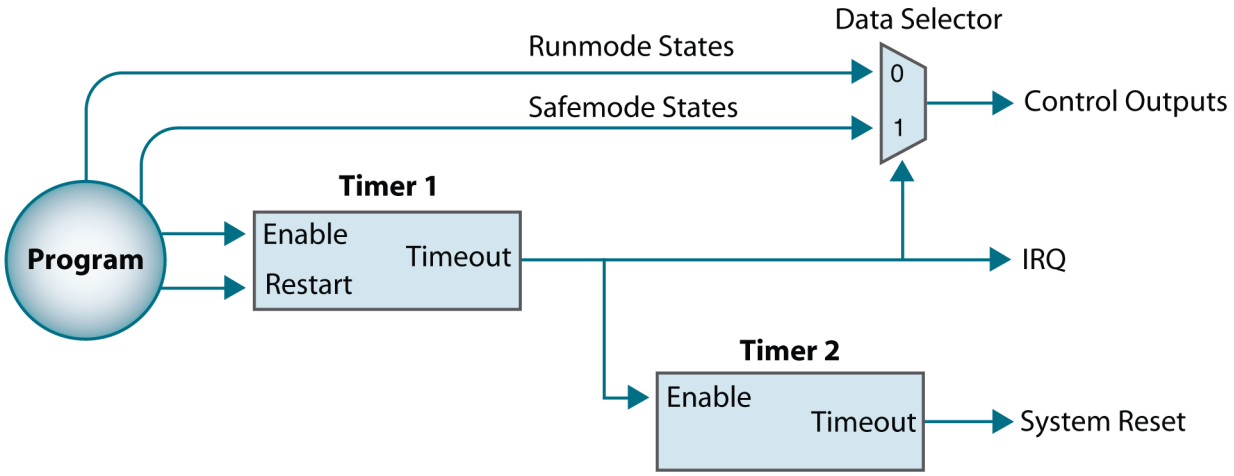
A superior alternative is to define two different sets of control states, "Runmode" and "Safemode," and employ a data selector to route the state sets to the outputs under watchdog control. The program can modify Runmode states at any time, but it can change Safemode states only when permitted by a special write-protect mechanism. Typically, the program will begin to control the Runmode states after it establishes Safemode states, which comprise a complete, customized set of safe states for all outputs.



During normal operation, the Runmode states are routed through the data selector to the outputs. Upon watchdog timeout, the data selector switches input sets so that Safemode states are applied to the outputs in place of Runmode states. Since the control circuitry has not been reset, it will continue to function normally (to the extent possible) and it will retain interface state information (e.g., incremental encoder counts, captured events, hardware configuration) that may be needed for fault recovery.

TWO-STAGE WATCHDOG

An abrupt system restart can be very expensive in terms of both downtime and loss of important state information. One or both of these costs can be mitigated with a multistage watchdog. This two-stage watchdog immediately switches the control outputs to safe states, but instead of triggering an immediate system restart, it schedules a deferred system restart and signals the computer, thus allowing time for the program to attempt to recover from the fault or log state information. If the program successfully recovers, the scheduled system reset will be canceled and the system will have avoided an expensive restart.



During normal operation, the program kicks Timer1 at regular intervals to prevent it from timing out. As long as Timer1 has not timed out, Timer2 is disabled and held at its initial value, and the control outputs are allowed to change under program control.

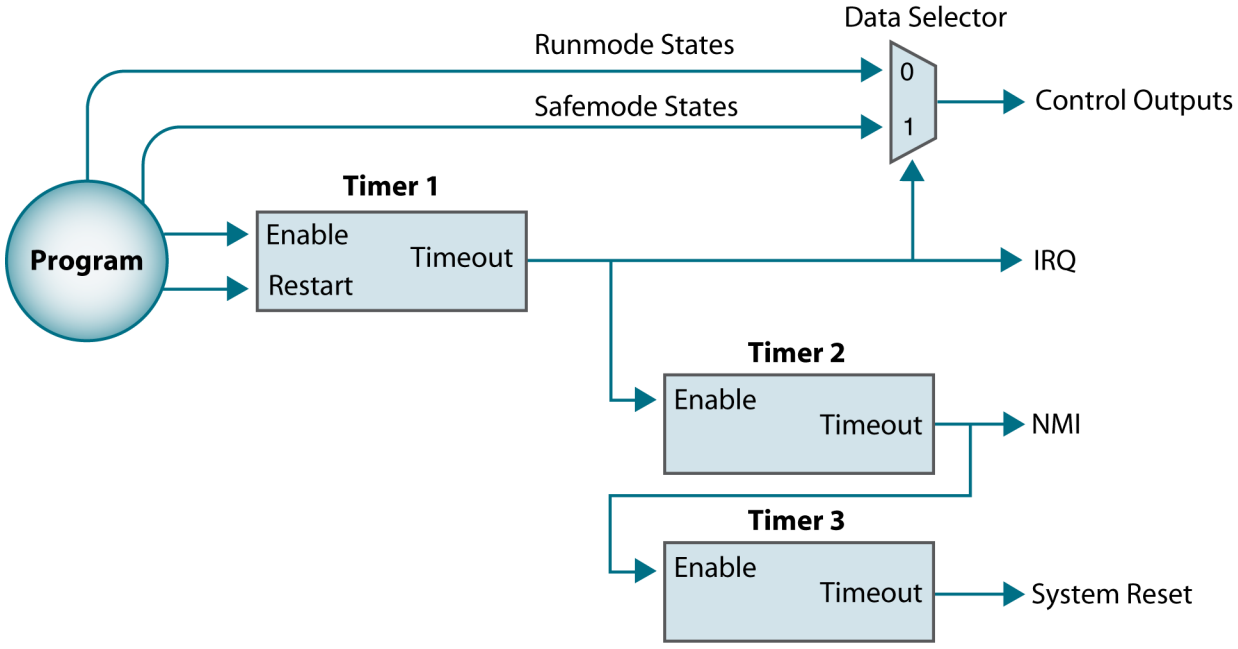
If a fault causes Timer1 to timeout, its Timeout signal will simultaneously switch the outputs to safe states, start Timer2 running, and request interrupt service. If the computer is able to respond to the IRQ, the program will have an opportunity to try to recover from the fault condition or, if the fault is uncorrectable, the program can save state and fault information. Upon successful recovery, the program will disable Timer1 (and by extension, Timer2), thus canceling the system reset. If the computer can't respond to the IRQ or recovery is impossible, a system reset will be invoked when Timer2 times out.

If the program will never take advantage of the IRQ, Timer1's timeout signal can be routed to the computer's NMI (non-maskable interrupt) input instead of a maskable IRQ input. In this case, a NMI notifies the computer that a system restart is imminent, and Timer2 will give it time to record fault information before the restart is triggered.

THREE-STAGE WATCHDOG

This multistage watchdog extends the two-stage watchdog by adding a third timer, making it possible to record debug information if graceful recovery fails.

During normal operation, the program kicks Timer1 at regular intervals to prevent a timeout. While Timer1 is running, Timer2 and Timer3 are disabled and held at their initial values and the control outputs are allowed to change under program control.



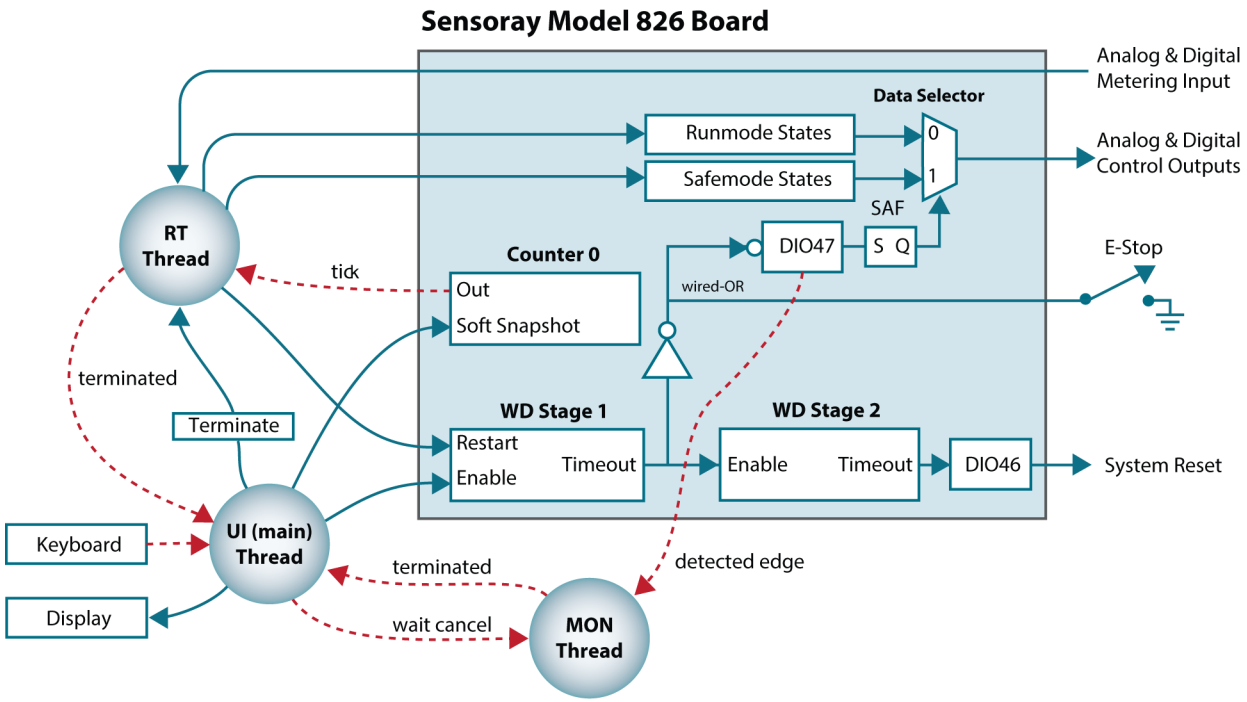
As in the previous example, a Timer1 timeout will switch the outputs to safe states, start Timer2, and request interrupt service. If the computer is able to respond to the IRQ, the program will attempt to recover from the fault condition and, if successful, the program will disable Timer1 (and by extension, Timer2), thus canceling further corrective actions.

If the computer cannot respond to the IRQ, or it takes too long to respond, or graceful recovery is impossible, Timer2 will timeout, which in turn will start Timer3 and assert a non-maskable interrupt request to indicate that a system restart is imminent. If the computer is appropriately configured and able to respond to the NMI, it will do so by logging important fault information (e.g., crash dump). The system will restart when Timer3 times out.

EXAMPLE

This example shows how to use Sensoray's model 826 PCI Express board to implement a basic real-time system with a two-stage watchdog and external fail-safe triggering. In this implementation, the control outputs will switch to safe states in response to a watchdog timeout or a signal from an emergency stop (E-Stop) switch. If the fail-safe condition is caused by a timeout, the watchdog timer (Timer1) will initiate

program-managed fault recovery; if the recovery fails, the reset timer (Timer2) will trigger a system restart by issuing pulses on the system reset output. If the fail-safe condition is caused by an E-Stop switch closure, the program will be notified but no system restart will occur.



HARDWARE RESOURCES

The model 826 board has 56 fail-safe control outputs consisting of eight analog outputs and 48 digital I/O (DIO) channels. Each DIO can operate as an input, a wired-OR I/O, or a fail-safe output, with edge detection and capture capability. The board also has six counters, sixteen analog inputs, and a three-stage watchdog timer.

One general purpose counter is used as a timer, leaving the other five available for general application use; this generates periodic ticks to pace real-time execution. Watchdog stages 1 and 2 are used as watchdog and reset timers.

Two DIOs are used by the fail-safe and watchdog mechanisms (46 remain available for general application use). DIO46 is the system reset output, and DIO47 serves as a wired-OR that combines the internal watchdog timeout signal with an external fail-safe trigger. In this example, the external trigger is sourced from a normally-open E-Stop switch. In the diagram shown above, all signal routing within the gray rectangle is configured by programming the board's internal signal routing matrix; no external wiring is required.

SOFTWARE STRUCTURE AND OPERATION

The example program is a skeletal console application with three threads: UI (user interface), RT (real-time), and MON (fail-safe monitor). When the program starts, the main (UI) thread creates the RT and MON threads and enables the watchdog timer. It then blocks until a user keypress or a termination signal from MON. Any keypress signals the program to terminate. When MON starts, it simply blocks while it waits for an edge event to be captured on DIO47. When RT starts, it establishes the safemode and initial runmode states, starts its tick timer, and kicks the watchdog; it then blocks until the first tick.

During normal operation, each tick unblocks RT, causing it to make one pass through its control loop. Within the control loop, it reads metering inputs, computes output states, and writes new runmode states as required by the application. At the end of the loop, RT kicks the watchdog and then it returns to the top of the loop to wait for the next tick.

If a fault condition prevents RT from running its control loop, RT will not kick the watchdog and watchdog Stage1 will timeout. The timeout signal propagates through DIO47, setting the SAF flag and switching the outputs to their safemode states. The timeout also enables watchdog Stage2, effectively scheduling a system restart. The leading edge of the timeout signal is detected and captured by DIO47; this unblocks MON and MON self-terminates, thus notifying UI that fail-safe mode has been activated. UI may respond to this by logging state and debug information, and then it can simply wait for Stage2 to restart the system or it may attempt to restore normal operation itself, thus avoiding a system restart.

During normal operation, an operator may activate the E-Stop switch. The E-Stop signal propagates through DIO47, setting the SAF flag and switching the outputs to their fail-safe states. The signal's leading edge is detected and captured by DIO47, thus terminating MON and signaling UI. UI can then take action as appropriate. Note that this will not cause a system restart because watchdog Stage1 has not timed out and thus Stage2 remains disabled.

SOURCE CODE

The following source code listing has been simplified for clarity. In particular, no error checking or handling is performed and header files, supporting functions, and required function arguments have been omitted. A functional version of this code can be downloaded from Sensoray's website.

```
int UI_thread(void)   // USER INTERFACE (MAIN) THREAD ////////////////////////////////
//
{
    const uint dio47[] = {0x800000, 0};

    WatchdogStart();                    // Enable watchdog timer
    ThreadStart(RT_thread);             // Launch realtime control thread
    ThreadStart(MON_thread);            // Launch safemode monitor thread

    printf("\nWATCHDOG DEMO\nHit any key to exit ...");

    while (1)    // MAIN LOOP
    {
        switch (WaitForSafemodeOrKeypress())    // Wait for signal from MON (sig=0) or
console keypress (sig=1)
```

```
            {
        case 0:         // Signal from MON termination -- safemode has been activated

            switch (S826_CounterSnapshotRead(board, 5, NULL, NULL, NULL, 0))  // Find out
why safemode activated ..
            {
            case S826_ERR_OK:          // Watchdog timeout activated safemode
                break;                 //   TODO: HANDLE WATCHDOG TIMEOUT
            case S826_ERR_NOTREADY:    // Watchdog didn't timeout, so e-stop switch closure
activated safemode
                break;                 //   TODO: HANDLE EMERGENCY STOP
            }
                    break;

        case 1:         // Keypress -- user is terminating application

            S826_DioWaitCancel(board, dio47);     // Signal MON thread to terminate
            S826_CounterSnapshot(board, 0);       // Signal RT thread to terminate
            WaitForThreadTerminate(RT_thread);    // Wait for rt to terminate
            WatchdogStop();                       // Deactivate the watchdog timer
            WaitForThreadTerminate(MON_thread);   // Wait for MON to terminate
            return 0;                             // Terminate program
           }
    }
}

int MON_thread(void)    // SAFEMODE MONITOR THREAD ////////////////////////////////////
///////
{
    uint diolist[] = {0x00800000, 0};
    S826_DioCapRead(board, diolist, 0, INFINITE);    // wait for captured edge on dio47
    return 0;                                        // terminate thread
}

int RT_thread(void)    // REAL-TIME OUTPUT CONTROL THREAD ////////////////////////////////
////
{
    uint snapshotReason;

    SafemodeDataWrite();     // Establish fail-safe output states
    PeriodicTimerStart();    // Start the tick timer

    while (1)    // MAIN LOOP -------
    {
        S826_CounterSnapshotRead(board, 0, NULL, NULL, &snapshotReason, INFINITE);  //
Wait for next tick
        switch (snapshotReason)
        {
        case S826_SSR_ZERO:                // If a tick occurred
                                           //   TODO: CONTROL LOOP WORK IS DONE HERE
            WatchdogKick(board);           //   Kick the watchdog
            break;                         //   Loop
        case S826_SSR_SOFT:                // If main thread is terminating
            PeriodicTimerStop(board, 0);   //   Halt the tick timer
            return 0;                      //   Terminate thread
        }
    }
}
```