Application Note

Single and Multistage Watchdog Timers

by Jim Lamberson ~ Copyright © 2012 Sensoray

Introduction

A **watchdog timer** is an electronic circuit that detects, and 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

As its name implies, a watchdog timer performs a timing function. The timing function is performed by a circuit that produces a delayed output signal in response to an input trigger signal. The circuit may be implemented as an analog delay circuit, which typically employs a monostable multivibrator, or as a digital delay circuit, which uses a digital counter to control the delay time. This application note focuses on digital watchdog timers, though many of the principles discussed here also apply to analog delay circuits.

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 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; this is known as a watchdog “event.” The timeout signal is connected to external circuitry so that it can initiate corrective action.

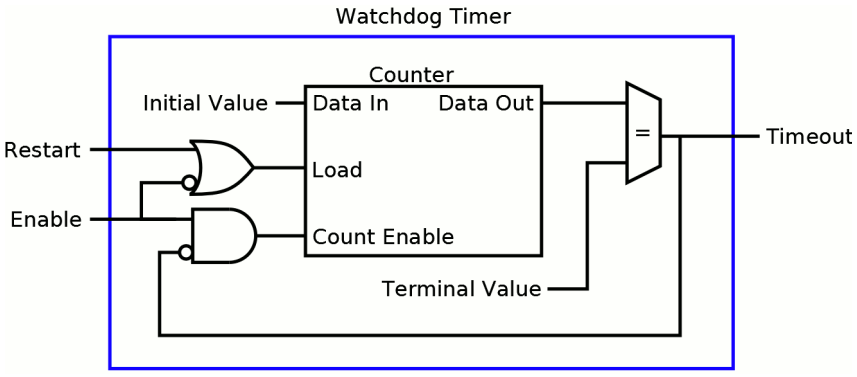

Figure 1: A basic digital watchdog timer

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 “kick” port. 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.

In many watchdog designs, the timer will be restarted regardless of the value written to the kick port, so that any arbitrary value will cause a timer restart. In other designs, however, a specific value must be written to the kick port; the timer will not be restarted unless the correct value is written to the port. This is useful in various situations. For example, in a multi-threaded program it may be necessary to detect faults in more than one thread. In such cases, each thread can be designed to contribute a part of the kick port value, so that the correct value will be formed only if all of the threads are operating normally.

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 begins 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 first sets all outputs to safe states and ceases output control and then it disables 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 state and 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.

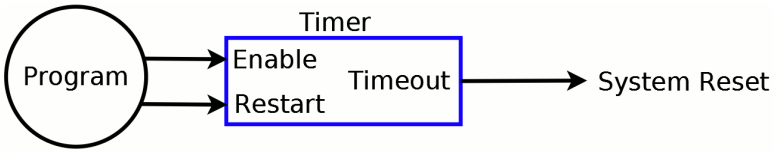

Figure 2: This circuit asserts a continuous reset upon timeout

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 may be used to satisfy this requirement.

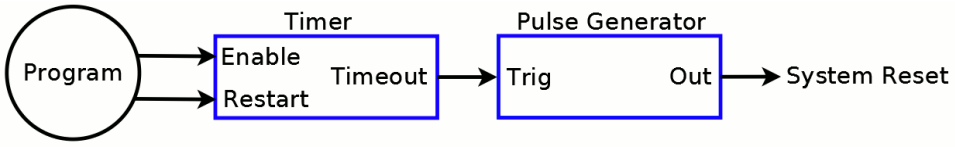

Figure 3: This circuit generates reset pulses upon timeout

Fail-Safe Systems in Multistage Watchdogs

It's been emphasized that a watchdog event 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 has two or more timers that signal events at different times, and the first event doesn't immediately restart the computer; it merely schedules a computer restart that will occur at a future time. Consequently, a multistage watchdog must work in concert with special circuitry that can switch outputs to safe states upon the first watchdog event, prior to the computer restart.

One way to do this is to employ a separate “control reset” signal that resets only the control circuitry (but not the computer) upon the first watchdog event. This is easily implemented, but it presents some complications and shortcomings. For example, although a control reset will restore outputs to their default power-up states, those states may not always be ideal. In fact, the ideal safe states may change as as a function of the overall system state. Also, a control reset can cause the loss of important state information that may be needed for fault recovery, and it may also interfere with the operation of interfaces that might otherwise continue to function normally during a fault condition.

A better 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.

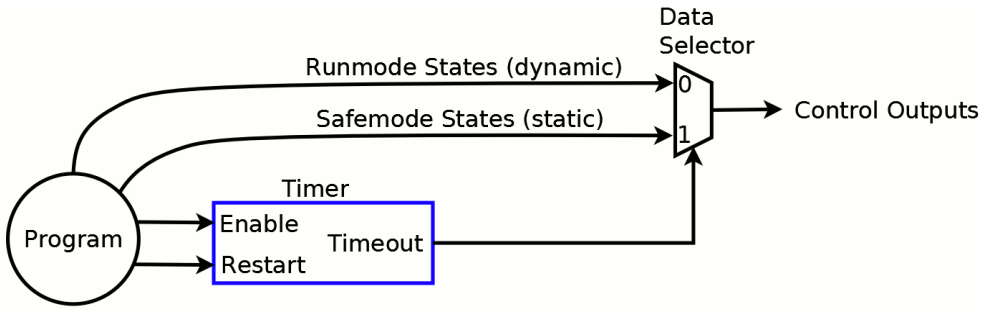

Figure 4: A flexible, watchdog-controlled, fail-safe system

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 hardware events, hardware configuration) that may be needed for fault recovery.

Two-stage Watchdog

Depending on the type of application, an abrupt system restart can be very expensive in terms of downtime, loss of important state information, or both. A multistage watchdog reduces the probability of incurring these expenses. The two-stage watchdog shown below will immediately switch the control outputs to safe states upon Timer1 timeout, 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, record state or fault information, or a combination of these. If the program successfully recovers, the scheduled system reset will be canceled and the system will have avoided an expensive restart.

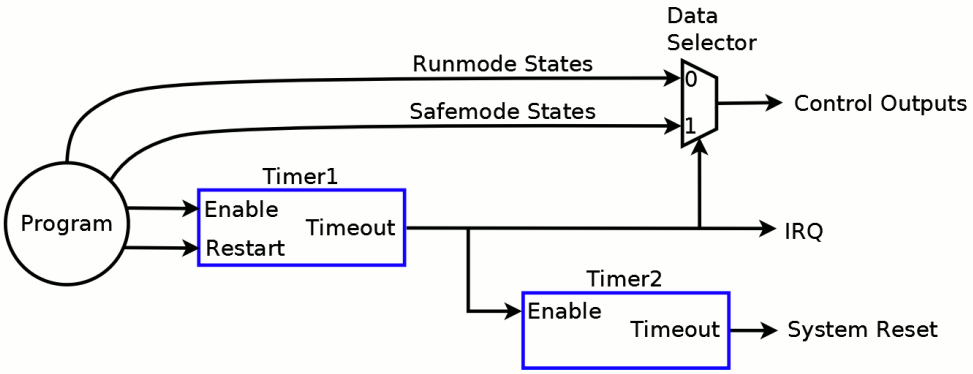

Figure 5: Two-stage watchdog

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 fail-safe states, start Timer2 running, and request interrupt service via a maskable interrupt request (IRQ). If the computer is able to respond to the IRQ, the program will have a limited-time opportunity to try to recover from the fault condition or, if the fault is uncorrectable, the program can try to 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

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

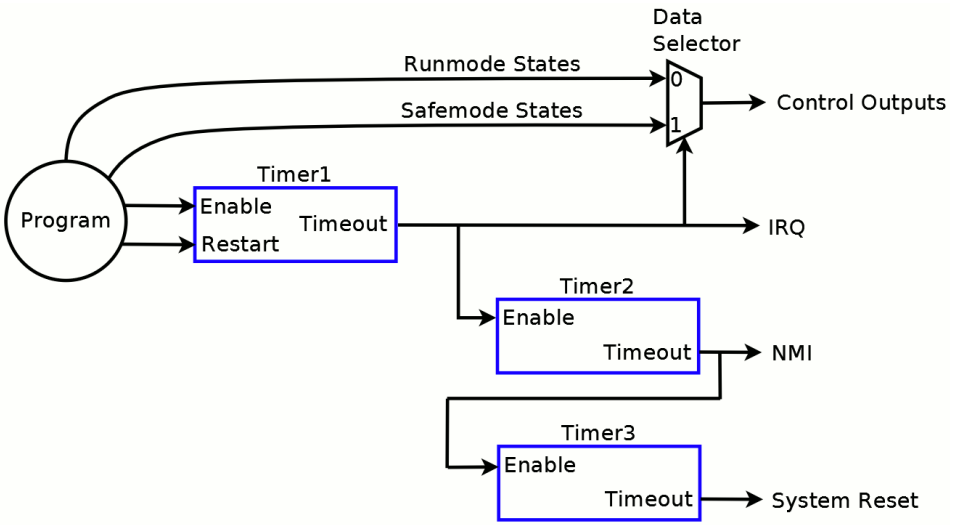

Figure 6: Three-stage watchdog

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 uses Sensoray's model 826 PCI Express board to implement a basic real-time control system with a two-stage watchdog and dual-triggered fail-safe operation. In this implementation (see Figure 7), all control outputs will switch to safe states in response to either a watchdog Stage1 timeout or upon closure of an emergency stop (E-Stop) switch. If the fail-safe condition is caused by the watchdog, the program will be notified and Stage2 will be started. If the program fails to clear the fault in the allotted time, Stage2 will trigger a system reset. If the fail-safe condition is caused by an E-Stop switch closure, the program will be notified, but Stage2 will not be started and no system reset will occur.

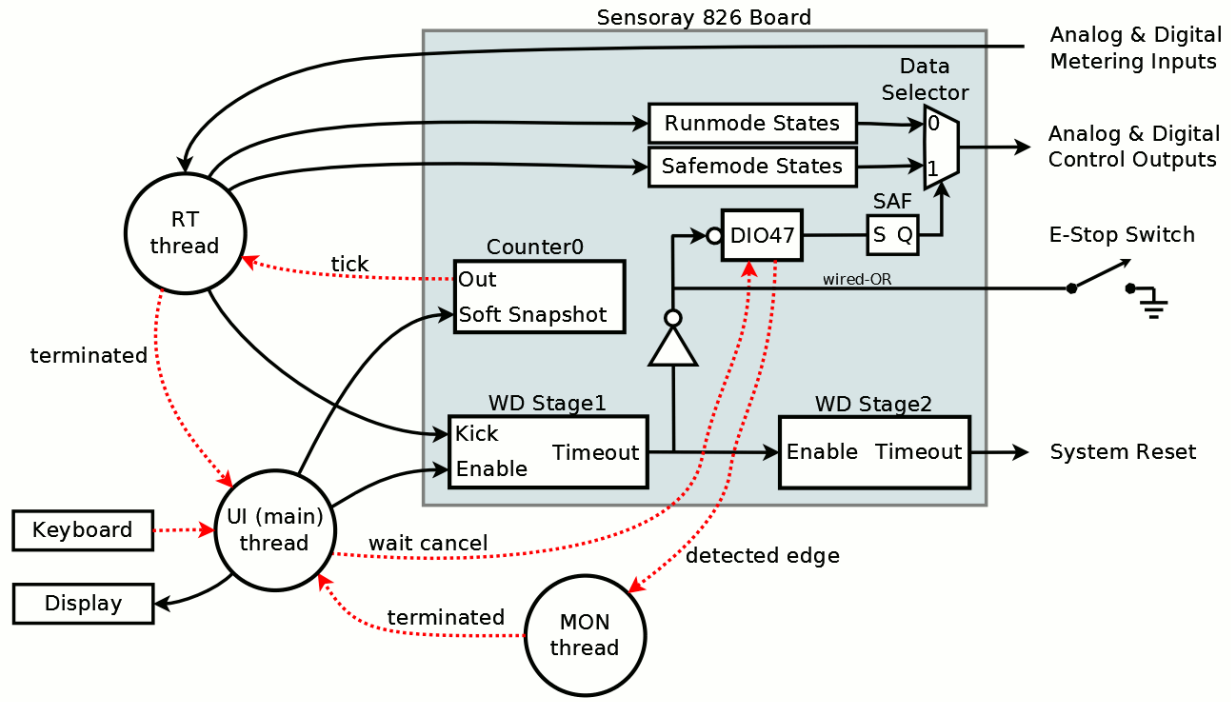

Figure 7: A real-time program working together with a two-stage watchdog and fail-safe controller

Hardware Resources

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

In this application, one of the general-purpose counters (Counter0) is used as a real-time clock; this generates periodic ticks that pace real-time program execution. Two watchdog stages are used, one to activate fail-safe mode and the other to generate a system reset signal. One of the DIOs (DIO47) is connected to an external, normally-open E-Stop switch. DIO47 is configured for wired-OR operation so that the DIO pin can be driven by a watchdog Stage0 timeout or by the E-Stop switch. The fail-safe system is configured so that fail-safe mode will be activated when DIO47 is driven low. In Figure 7, all signal routing within the gray rectangle is configured at run time 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 enables the watchdog timer and creates the RT and MON threads. It then blocks until a user keypress or a termination signal from MON. When it starts, MON simply blocks while it waits for an edge event to be captured on DIO47 or a wait cancellation signal from UI. When RT starts, it establishes the initial safemode and runmode states and kicks the watchdog, then it 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 control output states, and writes new runmode states as required by the application. At the end of the loop, RT kicks the watchdog and jumps to the top of its main loop to wait for the next tick.

If a fault condition prevents RT from running, RT will neglect to kick the watchdog and watchdog Stage1 will eventually timeout. The timeout signal propagates onto DIO47, thus setting the SAF flag and switching the control outputs to their fail-safe states. The timeout also starts watchdog Stage2, thereby scheduling a system restart. The leading edge of the timeout signal is detected and captured by DIO47; this unblocks MON, which in turn self-terminates. The MON termination signal notifies UI that fail-safe mode has been activated. If it is able to run, UI determines that a watchdog timeout caused fail-safe activation. It may respond to this by logging state and debug information, and then it can simply wait for Stage2 to reset the system or it may attempt to restore normal operation itself, thus avoiding a system reset.

An operator may activate the E-Stop switch at any time. 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; this will unblock MON, which will then self-terminate and signal UI. After determining that the fail-safe condition was activated by the E-Stop switch, UI can take action as appropriate (e.g., it can notify RT). Note that this will not cause a system restart because watchdog Stage1 has not timed out, and therefore Stage2 remains disabled.

The program is terminated by pressing any key. When a key is pressed, UI is unblocked and will send a wait cancel to MON and invoke a soft snapshot on Counter0, and then block until both MON and RT have terminated. Upon receiving the wait cancel, MON unblocks and self-terminates. RT is unblocked by the Counter0 snapshot. Recognizing this to be a soft snapshot (vs. periodic real-time tick), RT will clean up all real-time processing – including programming the control outputs to their shutdown states – and then self-terminate. When both MON and RT have terminated, UI will unblock and disable the watchdog (since RT no longer needs its services), perform any other required cleanup, and then terminate.

Source Code

The following source code has been simplified for clarity. In particular, no error checking or handling is performed and header files, support functions, and required function arguments have been omitted.

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

    WatchdogStart();                    // Configure and 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())    // Block until MON terminates (sig=0) or console keypress (sig=1)
        {
        case 0:        // MON thread terminated because safemode has been activated

            S826_WatchdogStatusRead(board, &status);  // Determine what caused safemode activation.

            if (status)      // Watchdog timeout activated safemode
            {
                             //   TODO: HANDLE WATCHDOG TIMEOUT
            }
            else             // Watchdog didn't timeout, so e-stop switch must have activated safemode
            {
                             //   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 ////////////////////////////////////////////////
{
    unsigned int 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 ////////////////////////////////////
{
    unsigned int 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
        }
    }
}
```