

Using watchdog timers to improve the reliability of TTCS embedded systems: Seven new patterns and a case study

Michael J. Pont¹ and Royan H.L. Ong

Department of Engineering, University of Leicester, University Road, LEICESTER LE1 7RH, UK.

Introduction

We have recently described a “language” consisting of more than seventy patterns which will be referred to here as the “PTTES Collection” (Table 1). These patterns are intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered, co-operatively scheduled (TTCS) system architecture.

STANDARD 8051	SMALL 8051	EXTENDED 8051
CRYSTAL OSCILLATOR	CERAMIC OSCILLATOR	RC RESET
ROBUST RESET	ON-CHIP MEMORY	OFF-CHIP DATA MEMORY
OFF-CHIP CODE MEMORY	NAKED LED	NAKED LOAD
IC BUFFER	BJT DRIVER	IC DRIVER
MOSFET DRIVER	SSR DRIVER (DC)	EMR DRIVER
SSR DRIVER (AC)	SUPER LOOP	PROJECT HEADER
PORT I/O	PORT HEADER	HARDWARE DELAY
SOFTWARE DELAY	HARDWARE WATCHDOG	CO-OPERATIVE SCHEDULER
HARDWARE TIMEOUT	LOOP TIMEOUT	MULTI-STAGE TASK
MULTI-STATE TASK	HYBRID SCHEDULER	PC LINK (RS232)
SWITCH INTERFACE (SOFTWARE)	SWITCH INTERFACE (HARDWARE)	ON-OFF SWITCH
MULTI-STATE SWITCH	KEYPAD INTERFACE	MX LED DISPLAY
LCD CHARACTER PANEL	I ² C PERIPHERAL	SPI PERIPHERAL
SCI SCHEDULER (TICK)	SCI SCHEDULER (DATA)	SCU SCHEDULER (LOCAL)
SCU SCHEDULER (RS-232)	SCU SCHEDULER (RS-485)	SCC SCHEDULER
DATA UNION	LONG TASK	DOMINO TASK
HARDWARE PULSE-COUNT	SOFTWARE PULSE-COUNT	HARDWARE PRM
SOFTWARE PRM	ONE-SHOT ADC	ADC PRE-AMP
SEQUENTIAL ADC	A-A FILTER	CURRENT SENSOR
HARDWARE PWM	PWM SMOOTHER	3-LEVEL PWM
SOFTWARE PWM	DAC OUTPUT	DAC SMOOTHER
DAC DRIVER	PID CONTROLLER	255-TICK SCHEDULER
ONE-TASK SCHEDULER	ONE-YEAR SCHEDULER	STABLE SCHEDULER

Table 1: The 72 patterns we have assembled in order to support the development of embedded systems. From: Pont, M.J. (2001) “Patterns for time-triggered embedded systems”, Addison-Wesley.

¹ To whom correspondence should be addressed: M.Pont@le.ac.uk.

This paper presents seven new patterns for embedded systems. We introduce them here.

WATCHDOG RECOVERY

In the PTTES Collection, the pattern **HARDWARE WATCHDOG** was presented: this pattern was intended to describe how to use watchdog timers (such as the ubiquitous ‘1232’ chip) in any embedded application.

At the time the PTTES Collection was assembled, **HARDWARE WATCHDOG** was viewed as a very minor pattern, and it was presented in an introductory part of the book. In recent months, we have helped other people use the PTTES Collection on a number of large and small projects. In the course of these projects, it quickly became clear that the range of ways in which watchdog timers can be used in TTCS applications was not adequately described in **HARDWARE WATCHDOG**.

The patterns described in this paper - together - form a replacement for **HARDWARE WATCHDOG**. In this new collection of watchdog-related patterns, **WATCHDOG RECOVERY** forms the entry point. This pattern describes, in general terms, how to use a watchdog with TTCS applications, and provides numerous links to the other patterns in this paper (where more detailed solutions are provided).

SCHEDULER WATCHDOG

This pattern describes how you can use a watchdog timer to ensure that the scheduler in your TTCS application is operating correctly.

PROGRAM-FLOW WATCHDOG

This pattern provides a description of a popular technique for dealing with program-flow errors in embedded systems: such errors are often thought to arise from electromagnetic interference.

OSCILLATOR WATCHDOG

We have met many developers (some with considerable experience) who believe that general-purpose watchdog timers can form the basis of techniques for detecting oscillator failure. One reason for including **OSCILLATOR WATCHDOG** in this paper is to help dispel this (sometimes dangerous) myth.

RESET RECOVERY, FAIL-SILENT RECOVERY, LIMP-HOME RECOVERY

These patterns describe strategies which may be used to recover your system in the event of a watchdog-induced reset.

Case study

A short case study is presented at the end of the paper. This employs an example (an automotive cruise-control system) to illustrate how many of the patterns presented in this paper can be used in a realistic embedded application.

Acknowledgement

We are very grateful to Bob Hanmer (our Shepherd at VikingPlop), who provided numerous useful suggestions during the evolution of this paper.

WATCHDOG RECOVERY

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. **Co-OPERATIVE SCHEDULER** [Pont, 2001, page 254]).

Problem

How you make best use of a watchdog timer in your TTCS application?

Background

Suppose there is a hungry dog guarding a house (Figure 1), and someone wishes to break in . If, during the burglary, an accomplice repeatedly throws the guard dog small pieces of meat, then the animal will be so busy concentrating on the food that he will ignore his guard duties, and will not bark. However, if the accomplice run out of meat or forgets to feed the dog for some other reason, the animal will start barking, thereby alerting the neighbours, property occupants or police.



Figure 1: The origins of the ‘watchdog’ analogy. See text for details.

This type of canine behaviour is mirrored (to an extent) in computerised “watchdog timers” used in microcontroller-based systems. More specifically, the watchdog timers used to implement **WATCHDOG RECOVERY** will - usually - have the following two features:

- The timer must be refreshed at regular, well-defined, intervals.
If the timer is not refreshed at the required time it will overflow, an process which will usually cause the associated microcontroller to be reset.

- When starting up, the microcontroller can determine the cause of the reset.

That is, it can determine if it has been started ‘normally’, or re-started as a result of a watchdog overflow. This means that, in the latter case, the programmer can ensure that the system will try to handle the error that caused the watchdog overflow.

As we will see, the features of watchdog hardware are a good match for the needs of TTCS systems, and - with a little care - a watchdog timer can form the basis of a simple but effective way of improving your system's ability to handle a range of different faults.

Solution

Understanding the basic operation of watchdog timer hardware is not difficult. However, making good use of this hardware in a TTCS application requires some care. As we will see in this section, there are three main issues which need to be considered:

- Choice of hardware;
- The watchdog-induced reset;
- The recovery process.

We begin by considering the choice of hardware.

Choice of hardware

We have seen in many previous cases (in Pont, 2001) that, where available, the use of on-chip components is to be preferred to the use of equivalent off-chip components. Specifically, on-chip components tend to offer the following benefits:

- Reduced hardware complexity, which tends to result in increased system reliability.
- Reduced application cost.
- Reduced application size.

These factors also apply when selecting a watchdog timer. In addition, when implementing **WATCHDOG RECOVERY**, it is usually important that the system is able to determine - as it begins operation - whether it was reset as a result of normal power cycling, or because of a watchdog timeout. **In most cases, only on-chip watchdogs allow you to determine the cause of the reset in a simple and reliable manner.**

With appropriate on-chip hardware, determining the cause of a reset is usually straightforward: we give an example at the end of this pattern to illustrate this.

The watchdog-induced reset

We consider time-based error detection, handling program-flow errors, and other - more general - uses for watchdog resets in this section.

(a) Time-based error detection

A key requirement in applications using a co-operative scheduler is that, for all tasks, under all circumstances, the following condition must be adhered to:

$$Duration_{Task} < Interval_{Tick}$$

Where: $Duration_{Task}$ is the task duration, and $Interval_{Tick}$ is the system 'tick interval'.

The pattern **SCHEDULER WATCHDOG** [this paper] describes techniques that will help you to meet this condition.

(b) Responding to program-flow errors

Timer-based error detection requires the watchdog timer to do two things:

1. Detect time-related errors;
2. Cause a system reset (and, thereby, invoke an error-recovery process).

Time-based error detection is not the only possibility. When the system uses a watchdog timer, we can use this timer to force a system reset at any time, through the use of an endless loop:

```
// One way of forcing a watchdog-induced reset
while(1);
```

Use of a watchdog in this way is particularly appropriate in situations where you have detected an error, and the nature of this error means that you cannot be sure what state the system is currently in.

One form of error that gives rise to such concerns is the program-flow error, which can occur as a result of electromagnetic interference. When such errors occur the program flow may be diverted to a “random” address in code memory. By the time you manage to detect that such a random jump has taken place, it is generally impossible to predict what damage has been done, and you therefore cannot be sure that, if you call an error-handling function, it will operate as intended.

In these circumstance, we can use the watchdog timer to perform a system reset, after which we call the error handler. In doing this, we assume that the system is more likely to operate correctly after it is reset, and that the error-handling function will therefore work more effectively when called in this way.

This technique for dealing with program-flow errors is discussed in detail in **PROGRAM-FLOW WATCHDOG** [this paper].

(c) Other uses for watchdog-induced resets

If your system uses watchdog-induced resets to handle program-flow errors, and / or it uses timer-based error detection techniques, then it can make sense to also use watchdog-induced resets to handle other errors. Doing this means that you can integrate some or all of your error-handling mechanisms in a single place (usually in some form of system initialisation function). This can - in many systems - provide a very “clean” and approach to error handling that is easy to understand (and maintain).

Note that this combined approach is only appropriate where the recovery behaviour you will implement is the **same** for the different errors you are trying to detect: an examination of the possible error-recovery mechanisms (which are summarised in the next section) may help you to decide if this is the case for your system.

Here are some suggestions for the types of errors that can be effectively handled in this way:

- Failure of on-chip hardware (e.g. analogue-to-digital converters, ports).
- Failure of external actuators (e.g. DC motors in an industrial robot; stepper motors in a printer).
- Failure of external sensors (e.g. ultraviolet sensor in an art gallery; vibration sensor in an automotive system).
- Temporary reduction in power-supply voltage.

We illustrate the use of this approach to error handling in the case study at the end of this paper.

Recovery behaviour

Before we decide whether we need to carry out recovery behaviour, we assume that the system has been reset. If the reset was “normal” we simply start the scheduler and run the standard system configuration.

If, instead, the cause of the reset was a watchdog overflow, then there are three main options:

- We can simply continue **as if** the processor had undergone an “ordinary” reset. This option is discussed in the pattern **RESET RECOVERY** [this paper].
- We can try to “freeze” the system in the reset state. This option is discussed in the pattern **FAIL-SILENT RECOVERY** [this paper].
- We can try to have the system run a different algorithm (typically, a very simple version of the original algorithm, often without using the scheduler). This option is discussed in the pattern **LIMP-HOME RECOVERY** [this paper].

Hardware resource implications

The main resource implication is that a suitable watchdog timer is required.

Reliability and safety implications

We consider a number of key features in this section.

Risk assessment

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

The limitations of single-processor designs

It is important to appreciate that there is a limit to the extent to which reliability of a single-processor embedded system can be improved using a watchdog timer.

For example, **LIMP-HOME RECOVERY** is the most sophisticated recovery strategy considered in this paper. If implemented with due care, it can prove very effective. However, it relies for its operation on the fact that - even in the presence of an error - the processor itself (and key support circuitry, such as the oscillator, power supply, etc) still continues to function. If the processor or oscillator suffer physical damage, or power is removed, **LIMP-HOME RECOVERY** cannot help your system to recover.

In the event of physical damage to your “main” processor (or its support hardware), you may need to have some means of engaging another processor to take over the required computational task. One way to perform this type of activity is to use **WATCHDOG SLAVE**².

Time, time, time ...

Suppose that the braking system in an automotive application uses a 500 ms watchdog and the vehicle encounters a problem when it is travelling at 70 miles per hour (110 km per hour). In these circumstances, the vehicle and its passengers will have travelled some 15 metres / 16 yards - right into the car in front - before the vehicle even begins to switch to a “limp-home” braking system.

In some circumstances, the programmer can reduce the delays involved with watchdog-induced resets, and thereby improve the system reliability. For example, many systems force a watchdog reset using code like this:

```
// One way of forcing a watchdog-induced reset
while(1);
```

Some hardware allows you to adjust the watchdog delay while the watchdog is active. This can be a useful means of reducing the delays involved in the watchdog-induced reset. For example, using the Infineon C515C, the watchdog reload register can be changed at any time, thereby altering the overflow period. This allows the programmer to do the following:

```
// Set up the watchdog for “normal” use
// - overflow period = ~39 ms
WDTREL = 0x00;

...

// Adjust watchdog timer for faster reset
// - overflow set to ~300 µs
WDTREL = 0x7F;

// Now force watchdog-induced reset
while(1);
```

On-chip watchdogs and ‘idle’ mode

In most applications based on **CO-OPERATIVE SCHEDULER** [Pont, 2001, page 254], the microcontroller enters ‘idle’ mode between scheduler ticks, after executing the Dispatcher function.

² **WATCHDOG SLAVE** is based on the “shared-clock scheduler” architecture (see Pont, 2001, Part F). The pattern is still under development, and details will be released at a future PLoP conference.

You need to be aware that - when entering idle mode - some microcontrollers **disable the (on-chip) watchdog timer**. If this happens, then none of your watchdog-based error-handling mechanisms will operate correctly.

If your chosen microcontroller disables the watchdog timer in idle mode, it may be necessary to avoid using this mode. Please note that the scheduler will still operate correctly in these circumstances; however, the power consumption of your system will increase.

Portability

This pattern does not rely (in any way) on features which are unique to the 8051 family: it can be applied in systems based on any microcontroller (e.g. PIC, AVR, HC08, C16x, ARM, etc).

Overall strengths and weaknesses

☺ **Watchdogs can provide a ‘last resort’ form of error recovery. If you think of the use of watchdogs in terms of ‘if all else fails, then we’ll let the watchdog reset the system’, you are taking a realistic view of the capabilities of this approach.**

Use of this technique usually requires an on-chip watchdog.

Used without due care at the design phase and / or adequate testing, watchdogs can reduce the system reliability dramatically. In particular, in the presence of sustained faults, badly-designed watchdog “recovery” mechanisms can cause your system to repeatedly reset itself. **This can be very dangerous.**

Watchdogs with long timeout periods are unsuitable for many applications.

Related patterns and alternative solutions

We consider a number of related patterns and alternative solutions in this section.

Related pattern: **SCHEDULER WATCHDOG** [this paper]

This pattern describes how you can use a watchdog timer to ensure that the scheduler in your TTCS application is operating correctly.

Related pattern: **PROGRAM-FLOW WATCHDOG** [this paper]

This pattern provides a description of a popular technique for dealing with program-flow errors in embedded systems: such errors are often thought to arise from electromagnetic interference.

Related pattern: **OSCILLATOR WATCHDOG** [this paper]

This pattern describes how to deal with oscillator failures in a single-processor embedded system.

Related pattern: **RESET RECOVERY** [this paper]

This pattern describes a very simple recovery strategy that can be used after a watchdog-induced reset.

Related pattern: **FAIL-SILENT RECOVERY** [this paper]

This pattern describes how to shut down your system after a watchdog-induced reset.

Related pattern: **LIMP-HOME RECOVERY** [this paper]

This pattern describes how you can re-start your system (and run a different - usually very simple - algorithm), after a watchdog-induced reset.

Other simple watchdog solutions

Bruce Powel Douglass has described an alternative watchdog pattern (**WATCHDOG** [Douglass, 1999, p.646]): note that this pattern is not tailored for use with TTCS applications.

A software watchdog?

In certain restricted circumstances, a software watchdog may also be useful. This can be created from two components:

- A Timer ISR;
- A refresh function.

Essentially, we set a timer to overflow in (say) 60 ms. Under normal circumstances, this timer will never overflow, because we will call the “refresh” function regularly, and - thereby - restart the timer. If, however, the program is ‘jammed’, the refresh function will not be called. When the timer overflows, the ISR will be called: this can be used to implement an ‘appropriate’ error recovery strategy.

The main advantage of a software watchdogs is that different forms of error recovery (not necessarily involving a complete chip reset) are possible.

The main concern with this approach is that some errors (for example, those induced by EMI) may disrupt the “software” timer as well as the main application code: hardware watchdogs **appear** to be more robust in these circumstances.

Note that some hardware provides a way of obtaining a combination of “software” and “hardware” watchdogs. Specifically, the DS87C520 (and similar family members) allow the programmer to invoke an interrupt service routine (ISR) a short time before the chip undergoes a full reset. This provides a mechanism for trying to deal with the source of the error in an ISR and - if unsuccessful - allowing a full reset to take place.

When one processor is not enough

As noted in “solution”, there is a limit to the extent to which reliability of a single-processor embedded system can be improved using any form of watchdog timer.

Using a shared-clock scheduler (see Pont, 2001, Part F) can sometimes be a useful alternative to the techniques discussed in this pattern.

Other patterns

Some alternative patterns for fault tolerance and error recovery which may be of interested were presented recently by Saridakis (2002).

Example: Automotive cruise control.

Use of **WATCHDOG RECOVERY** is illustrated in the case study at the end of this paper.

Example: Determining the cause of a watchdog reset

As noted in “Solution”, most implementations of **WATCHDOG RECOVERY** rely on an ability to determine the cause of a system reset. Fortunately, this is usually easy to do. For example, in the Infineon C515C, the WDTS flag (bit 6 in the register IP0) is set if the reset was caused by a watchdog timer overflow. Having determined the status of this bit, it should be cleared in software:

```
// Determine if reset was caused by watchdog overflow (C515C)
if (IP0 & 0x40)
{
    // WDTS flag is set - reset *was* caused by watchdog
    Watchdog_reset_G = 1;

    // Clear the IP0 flag
    IP0 &= 0xBF;
}
else
{
    Watchdog_reset_G = 0;
}
```

Further reading

Douglass, B.P. (1999) “*Doing hard time: Developing real-time system with UML, objects, frameworks, and patterns*”, Addison-Wesley. ISBN: 0-201-498375.

Pont, M.J. (2001) “*Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*”, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

Saridakis, T. (2002) “A system of patterns for fault tolerance”, paper presented at EuroPloP 2002, July 2002, Germany.

SCHEDULER WATCHDOG

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. **Co-OPERATIVE SCHEDULER** [Pont, 2001, page 254]).

Problem

How can you use a watchdog timer to ensure that the scheduler in your TTCS application is operating correctly?

Background

SCHEDULER WATCHDOG can be seen as an implementation of the more general pattern **WATCHDOG RECOVERY** [this paper]: please refer to **WATCHDOG RECOVERY** for background information that will assist in the understanding of the present pattern.

Solution

We will start by considering time-based error detection techniques.

Time-based error detection

A key requirement in applications using a co-operative scheduler is that, for all tasks, under all circumstances, the following condition must be adhered to:

$$Duration_{Task} < Interval_{Tick} \quad - Eq. 1$$

Where: $Duration_{Task}$ is the task duration, and $Interval_{Tick}$ is the system ‘tick interval’.

We have previously discussed (Pont, 2001) a number of techniques which can help you meet this condition³. In many systems, the designers apply **SCHEDULER WATCHDOG** in order to develop a final safety net for their system.

In order to implement such a safety net, we need to examine the condition summarised in **Equation 1** more carefully. While **Equation 1** describes a key condition which must be satisfied in order to guarantee precise scheduling behaviour, we also need to take into account the CPU overheads imposed by the running of the scheduler itself. We can represent this as follows:

$$Duration_{Task} < \left(1 - \frac{CPU_{Scheduler}}{100}\right) \times Interval_{Tick} \quad - Eq. 2$$

³ A summary of these techniques is given in “Related patterns and alternative solutions”.

Where: $CPU_{Scheduler}$ is the percentage of the available CPU time consumed by the scheduler itself.

Equation 2 will be applicable where (only) one task is scheduled to execute at any tick interval. If this condition is not satisfied, then we need to take into account the duration of *all* tasks that are scheduled to run in the same tick interval. Thus - at every tick interval - we need to ensure that:

$$\sum_{i=1}^N Duration_{task\ i} < \left(1 - \frac{CPU_{Scheduler}}{100}\right) \times Interval_{tick} \quad - Eq. 3$$

Where: $\sum_{i=1}^N Duration_{task\ i}$ is the sum of the duration of all the tasks scheduled to run at a particular tick interval.

It is possible to test the condition summarised in **Equation 3** using a watchdog timer, as follows:

- Set the watchdog timer to overflow at a period greater than the tick interval.
- Create a task that will update the watchdog timer shortly before it overflows.
- Start the watchdog.

Under normal circumstances, the watchdog timer will never overflow, and your system will operate as normal. However, if the duration of a task (or the duration of a sequence of tasks, scheduled to execute in the same tick interval) cause the scheduling to be significantly disrupted, the watchdog timer will reset the system.

Selecting the overflow period

Selecting the watchdog overflow period requires some care, since the choice of the overflow period will depend on the system characteristics.

(a) Systems with ‘hard’ timing constraints

For systems with “hard” timing constraints for one or more tasks, it is usually appropriate to set the watchdog overflow period to a value slightly greater than the tick interval (e.g. 1.1 ms overflow in a system with 1 ms ticks). In this way, you will very rapidly detect scheduling problems.

Please note that to do this, the watchdog timer will usually need to be driven by a crystal oscillator (or the timing will not be sufficiently accurate). In addition, the watchdog timer will need to give you enough control over the timer settings, so that the required overflow period can be set.

(b) Systems with ‘soft’ timing constraints

The ‘hard timing’ approach is very effective, but before deciding on this option, you should bear in mind the fact that many (‘soft’) TTCS systems continue to operate safely and effectively, even if - **at times** - the duration of the task(s) that are scheduled to run at a particular time exceeds the tick interval.

To give a simple example, a scheduler with a 1 ms tick interval can - **without problems** - schedule a single task with a duration of 10 ms that is called every 20 ms.

Of course, if the same system is also trying to schedule a task of duration 0.1 ms every 5 ms, then - at times - the 0.1 ms task will be blocked. Often careful design will avoid this blockage but - even if it occurs - it still may not matter because, although the 0.1 ms will not always run on time, it will always run (that is, it will run 200 times every second, as required).

For some tasks - with soft deadlines - this type of behaviour may be acceptable. If it is, then it is appropriate to use a watchdog timer with a longer time-out period.

Typically, this will be done as follows:

- Set the watchdog to overflow after a period of around 100 ms.
- Feed the watchdog every millisecond, using an appropriate task.
- Only if the scheduling is blocked for more than 100 ms will the system be reset.

Recovery strategies

In the event that the watchdog timer has overflowed, we know that something has disrupted the scheduling.

A range of suitable recovery strategies are discussed in [RESET RECOVERY](#) [this paper], [FAIL-SILENT RECOVERY](#) [this paper] and [LIMP-HOME RECOVERY](#) [this paper].

Hardware resource implications

Using [SCHEDULER WATCHDOG](#) requires an appropriate watchdog timer: please see [WATCHDOG RECOVERY](#) [this paper] for details.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

Please see [WATCHDOG RECOVERY](#) [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

Portability

The approach to error detection a recovery described in [SCHEDULER WATCHDOG](#) is not in any way specific to the 8051 microcontroller family: it can be used with any device.

Overall strengths and weaknesses

- ☺ **SCHEDULER WATCHDOG** provides a useful “safety net” in the event that problems in the system disrupt the scheduling.

Use of this technique usually requires an on-chip watchdog.

Used without due care at the design phase and / or adequate testing, watchdogs can reduce the system reliability dramatically. In particular, in the presence of sustained faults, badly-designed watchdog “recovery” mechanisms can cause your system to repeatedly reset itself. **This can be very dangerous.**

Watchdogs with long timeout periods are unsuitable for many applications.

Related patterns and alternative solutions

Please refer to **WATCHDOG RECOVERY** [this paper] for references to other, general, watchdog patterns.

In this section, some patterns directly related to **SCHEDULER WATCHDOG** are mentioned.

Other mechanisms for detecting (or avoiding) time-based errors

As noted in “Solution” there are a number of other patterns in the PTTES Collection that can help you satisfy the time constraints described in Equation 1, Equation 2 and Equation 3. For example:

- The processor patterns (**STANDARD 8051**, **SMALL 8051**, **EXTENDED 8051**) allow selection of a processor with performance levels appropriate for the application.
- The oscillator patterns (**CRYSTAL OSCILLATOR** and **CERAMIC RESONATOR**) allow an appropriate choice of oscillator type, and oscillator frequency to be made, taking into account system performance (and, hence, task duration), power-supply requirements, and other relevant factors.
- The various Shared-Clock schedulers (**SCC SCHEDULER**, **SCI SCHEDULER (DATA)**, **SCI SCHEDULER (TICK)**, **SCU SCHEDULER (LOCAL)**, **SCU SCHEDULER (RS-232)**, **SCU SCHEDULER (RS-485)**) describe how to schedule tasks on multiple processors, while still maintaining a time-triggered system architecture. Using one of these schedulers as a foundation, the pattern **LONG TASK** describes how to migrate longer tasks onto another processor without compromising the basic time-triggered architecture.
- **LOOP TIMEOUT** and **HARDWARE TIMEOUT** describe the design of timeout mechanisms which may be used to ensure that tasks complete within their allotted time.
- **MULTI-STAGE TASK** discusses how to split up a long, infrequently-triggered task into a short task, which will be called more frequently. **PC LINK (RS232)** and **LCD CHARACTER PANEL** both implement this architecture.
- **HYBRID SCHEDULER** describes a scheduler that has most of the desirable features of the (pure) co-operative scheduler, but allows a single long (pre-emptible) task to be executed.

Before implementing **SCHEDULER WATCHDOG**, you should consider whether these patterns meet the needs of your application.

Example: Automotive cruise control.

Use of **SCHEDULER WATCHDOG** is illustrated in the case study at the end of this paper.

Example: A library for the watchdog timer on the Infineon C515C

A simple code “library” supporting the use of the watchdog timer on the Infineon C515C is presented in Listing 1, Listing 2 and Listing 3.

```
/*-----*-  
  
WATCHDOG_C515C_Init()  
  
This function sets up the watchdog timer.  
  
-----*/  
void WATCHDOG_C515C_Init(void)  
{  
    // Watchdog timer prescaler (1/16) enabled  
    // Watchdog timer reload value is 0x6B  
    // Oscillator is 10 MHz -> watchdog period is ~103 ms  
    WDTREL = 0xEB;  
  
    // Start watchdog timer  
    WDT = 1;  
    SWDT = 1;  
}
```

Listing 1: Part of a small “watchdog” library for the Infineon C515C.

```
/*-----*-  
  
WATCHDOG_C515C_Refresh()  
  
Feed the internal C515C watchdog.  
  
-----*/  
void WATCHDOG_C515C_Refresh(void)  
{  
    WDT = 1;  
    SWDT = 1;  
}
```

Listing 2: Part of a small “watchdog” library for the Infineon C515C.

```

/*-----*/

WATCHDOG_C515C_Cause_of_Reset()

Returns 1 if last reset was caused by watchdog (and clears flag)
Returns 0 if last reset was "normal".

/*-----*/
int WATCHDOG_C515C_Cause_of_Reset(void)
{
    // Determine if reset was caused by watchdog overflow (C515C)
    if (IP0 & 0x40)
    {
        // Clear the IP0 flag
        IP0 &= 0xBF;

        return 1;
    }

    return 0;
}

```

Listing 3: Part of a small “watchdog” library for the Infineon C515C.

Further reading

Pont, M.J. (2001) *“Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers”*, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

PROGRAM-FLOW WATCHDOG

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. **Co-OPERATIVE SCHEDULER** [Pont, 2001, page 254]).

Problem

How can you recover from program-flow errors on an embedded processor?

Background

This pattern is concerned with reducing the impact of ‘program flow’ errors on embedded applications. Such errors can occur as a result of electromagnetic interference.

Arguably, the most serious form of program-flow error in an embedded microcontroller is corruption of the program counter (PC), also known as the instruction pointer⁴. Since the PC of the 8051 is a 16-bit wide register, we make the reasonable assumption that – in response to PC corruption – the PC may take on any value in the range 0 to 65535. In these circumstances, the 8051 processor will fetch and execute the next instruction from the code memory location pointed to by the corrupted PC register. This code memory location may contain:

- program code,
- data constants, or,
- “nothing” (that is, it is unprogrammed, and contains neither code nor meaningful data).

We discuss each of these possibilities in the sections that follow.

Vectoring to program-code locations

Clearly, corruption of the instruction pointer that causes the program flow to be diverted to a “random” address is likely to cause severe side effects. However, the precise impact of such diversions can be very difficult to predict.

A particular problem arises because in the 8051 (and many other processors), more than half of the instructions are “multibyte instructions”, such as “POP” and “ACALL” that occupy two or three memory locations, respectively. PC corruption may cause the program flow to be diverted to any of these locations.

⁴ The PC is only one of many registers in an embedded processor and there is no evidence to suggest that this particular register is any more or less susceptible to EMI than the others. However, the impact of corruption to the PC is arguably the most serious result of EMI, as it can result in disruption to the program flow.

To illustrate the nature of the resulting “multibyte instruction trap” (MIT), consider the assembly code shown below:

```
0100 759850    MOV    98H,#050H
0103 438920    ORL    89H,#020H
0106 758DFD    MOV    8DH,#0FDH
```

If the PC is corrupted and takes on the value 0x0101, then the code above will be interpreted as follows:

```
0101 98        SUBB    A,R0
0102 5043       JNC     #43H
0104 8920       MOV     20H,R1
0106 758DFD    MOV     8DH,#0FDH
```

In this example, the first three instructions of the original program code have been misinterpreted while the rest remain unchanged. Of course, the number of instructions that will be misinterpreted depends on the instruction sequence, the corrupted PC value and the state of the processor at the time of PC corruption. In short, the precise impact **is impossible to predict** in most practical situations.

Vectoring to data locations

The problems with misinterpretation of instructions also apply to data values stored in the code area since, to the processor, data constants - such as digital filter coefficients stored in the code area - are indistinguishable from program code.

Again, the results of this are - again - very difficult to predict.

Vectoring to unprogrammed memory locations

Unprogrammed memory locations will usually (by default) have the contents 0xFF, which corresponds to the “MOV R7,A” in the 8051 instruction set (“Copy the contents of the accumulator to register R7”).

In many applications, the program code will occupy the lower code memory addresses, and the remainder of the memory will be unprogrammed. In these circumstances the processor will execute “MOV R7,A” instructions until the PC reaches the end of the physical code memory. The processor will then continue executing program code at location 0x0000. This can have an impact similar to a processor reset⁵.

In other applications there may be unprogrammed “gaps” in the memory maps, followed by constant data or program code. Execution of this code (or data, treated as code) is likely to have less predictable side effects, as we discussed earlier.

⁵ It is important to appreciate that this is NOT the same as a processor reset. For example, when the processor is reset, register values take on well-defined values: this has important implications for several aspects of system behaviour, such as initial port settings.

Solution

The technique we discuss here has previously been described and assessed in a number of studies (Campbell, 1995; Campbell, 1988; Niauxsat, 1998; Ong and Pont, 2001; Ong *et al.*, 2001; Ong and Pont, 2002).

An overview of the approach

The most straightforward implementation of **PROGRAM-FLOW WATCHDOG** involves two stages:

- We fill unused locations at the end of the program code memory with single-byte “No Operation” (NOP), or equivalent, instructions.
- We place a “PC Error Handler” (PCEH) at the end of code memory to deal with the error⁶.

The operation of Program-Flow Watchdogs may be easily predicted. When an PC error occurs **and the PC points to a memory location within the Program-Flow Watchdog area**, the processor will repeatedly execute NOP instructions until the PC points to the start of the PCEH. The error handler will then carry out its intended recovery function.

Dealing with errors

Here, we will assume that the PCEH will consist mainly of a loop:

```
// Force watchdog timeout
while(1);
```

This means that, as discussed in **WATCHDOG RECOVERY** [this paper] the watchdog timer will force a clean system reset.

Please note that, as also discussed in **WATCHDOG RECOVERY**, we may be able to reduce the time taken to reset the processor by adapting the watchdog timing. For example:

```
// Set up the watchdog for “normal” use
// - overflow period = ~39 ms
WDTREL = 0x00;

...

// Adjust watchdog timer for faster reset
// - overflow set to ~300 µs
WDTREL = 0x7F;

// Now force watchdog-induced reset
while(1);
```

After the watchdog-induced reset, we need to implement a suitable recovery strategy. A range of different options are discussed in **RESET RECOVERY** [this paper], **FAIL-SILENT RECOVERY** [this paper] and **LIMP-HOME RECOVERY** [this paper].

⁶ Note that, except in the event of PC corruption, the PCEH is unreachable.

Hardware resource implications

As noted above, **PROGRAM-FLOW WATCHDOG** can only be guaranteed to work where the corrupted PC points to an “empty” memory location. Maximum effectiveness will therefore be obtained with comparatively small programs (a few kilobytes of code memory), and larger areas of empty memory.

If devices with less than 64kB of code memory are used, a problem known as “memory aliasing can occur (see Figure 2).

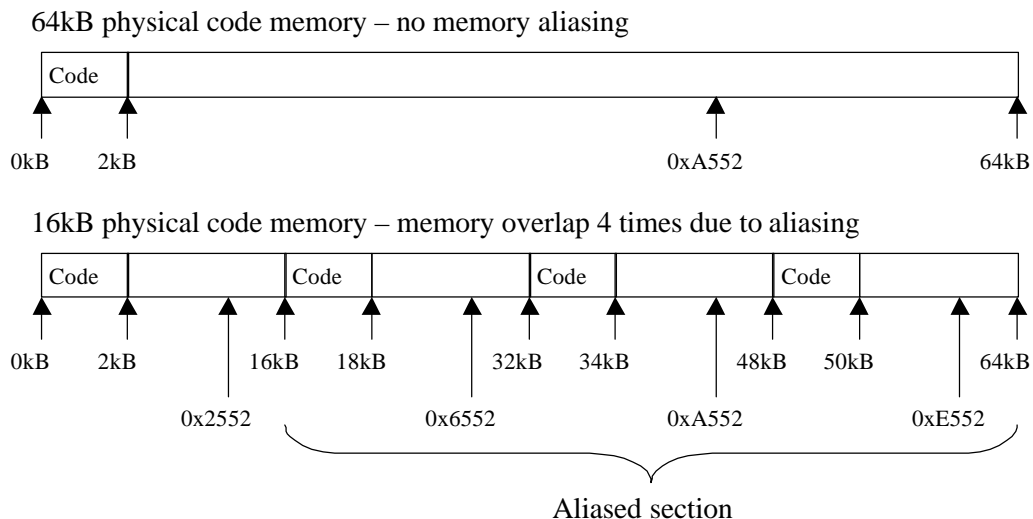


Figure 2: Problems caused by memory aliasing. See text for details.

Figure 2 shows a 2kB program on microcontrollers with 16kB and 64kB of physical code memory. In this example, the PC is addressing location 0xA552 on the 64kB device. **In the 16kB device, the lack of the upper two address lines means that addresses 0x2552, 0x6552, 0xA552 and 0xE552 will all address the same physical code location.**

Memory aliasing has an important impact on the effectiveness of Program-Flow Watchdog. For example, suppose your program code is 2 kB in size:

- If you use a 64 kB code memory to store your program, you are likely to trap (on average) $62/64 * 100\%$ (= 96.9%) of program-flow errors.
- If you use a 4 kB code memory to store your program, you are likely to trap (on average) $2/4 * 100\%$ (= 50%) of program-flow errors.

The implications are clear. If you want to increase the chances of detecting program-flow errors using this approach, you need to use the maximum amount of (code) memory that is supported by your processor. In the case of the 8051 family, this generally means selecting a device with 64 kB of memory. **Clearly, this choice will have cost implications.**

Reliability and safety implications

Use of **PROGRAM-FLOW WATCHDOG** may help to improve reliability of your system in the presence of program-flow errors (which may, in turn, result from EMI).

Under normal conditions, neither the filling of unused memory locations with NOP instructions, nor the addition of an error handler will have any impact on your program.

Please note, however, that - if a watchdog timer is used as part of your error-recovery strategy - you need to ensure that you have a full understanding of the implications this can have for the reliability of your system. Please see [WATCHDOG RECOVERY](#) [this paper] for further details.

Portability

The technique used in this pattern is applicable with any microcontroller family (however, the particular instructions used for the “NOP” behaviour will - obviously - need to match the hardware).

Overall strengths and weaknesses

☺ **A low-cost technique that can be effective in the presence of program-flow errors.**

For maximum effectiveness, a significant amount of “empty” code memory is required.

Related patterns and alternative solutions

We consider a number of alternative solutions and related patterns in this section.

Speeding up the response

We stated in “Solution” that the most straightforward implementation of **PROGRAM-FLOW WATCHDOG** involves two stages:

- We fill unused locations at the end of the program code memory with single-byte “No Operation” (NOP), or equivalent, instructions.
- Second, a small amount of program code, in the form of an “PC Error Handler” (PCEH), is placed at the end of code memory to deal with the error.

Suppose that we implement this solution, and that our microcontroller therefore has a large amount of ROM, filled with NOP instructions. Further suppose that a program-flow error throws the PC to the start of this NOP area. It may then take an appreciable period of time for the processor to reach the error handler. In addition, the time taken to recover from an error is highly variable (since it depends on the value of the corrupted PC).

An alternative is to fill the memory not with “NOP” instructions but with “jump” instructions. In effect, we want to fill each location with “Jump to address X” instructions, and then place the error handler at address X.

In practice, such a jump instruction will occupy more than one byte, but this problem is not insurmountable. In the 8051, the simplest implementation is to fill the empty memory with “long jump” instructions (0x02). As a result (almost) every time the PC lands in this area, the processor will execute the instruction: “Jump to 0x0202”. The error handler will then be located at address 0x0202.

We give an example of this process below.

The recovery operation

A range of suitable recovery strategies are discussed in [RESET RECOVERY](#) [this paper], [FAIL-SILENT RECOVERY](#) [this paper] and [LIMP-HOME RECOVERY](#) [this paper].

Hardware-based alternatives

To deal with EMI-related problems, hardware solutions, including device shielding, wiring screening and input/output filtering are widely used. However, hardware solutions are expensive, can suffer physical damage, and – in applications such as Hall-effect sensors – can interfere with normal device operation.

Despite this, in most cases, it does not make sense to abandon hardware protection completely. Software-based techniques can be effective as an adjunct to hardware-based techniques.

Example: Automotive cruise control.

Use of [PROGRAM-FLOW WATCHDOG](#) is illustrated in the case study at the end of this paper.

Example: Implementing Program-Flow Watchdog (NOP fill)

We summarise how to implement a basic [PROGRAM-FLOW WATCHDOG](#) on the 8051 microcontroller, using the Keil compiler, below:

1. Compile and link the program, as normal, with an error handler.
2. From the .M51 file, determine the length of the error function (e.g. 45 bytes, 0x2D bytes).
3. Determine the size of the code memory you will use (e.g. 0x2000 = 8K); ideally, this will be 64 kbytes.
4. Subtract the size of the error function from the code-memory size (e.g. 0x2000 - 0x2D = 0x1FD3)
5. Use the compiler / linker options to move the error handler to this location.
6. EITHER: Use your device programmer to fill the memory with NOP instructions.
OR: Use the .M51 file to determine the required size, and use the startup.A51 file to set the values to “NOP”.
7. Re-compile and link the code, and program the chip.

Example: Implementing Program-Flow Watchdog (Jump version)

We summarise how to implement a “jump” version of [PROGRAM-FLOW WATCHDOG](#) on the 8051 microcontroller, using the Keil compiler, below:

1. Write the program (including error handler).
2. Use the compiler / linker options to move the error handler to location 0x0202.
3. Use your programmer to fill the memory with 0x02 instructions - or use the .M51 file (again) to determine the required size, and use the startup.A51 file to set the values to “0x02”.
4. Compile and link the code, and program the chip.

Further reading

- Campbell, D. (1995) “*Designing for Electromagnetic compatibility with Single-Chip Microcontrollers*”, Motorola Application Note AN1263.
- Campbell, D. (1998) “Defensive Software Programming with Embedded Microcontrollers”, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, Nov 1998 (Conference code 98/471).
- Niaussat, A. (1998) “*Software techniques for improving ST6 EMC performance*”, ST Application Note AN1015/0398.
- Ong, H.L.R and Pont, M.J. (2001) “Empirical comparison of software-based error detection and correction techniques for embedded systems”, Proceedings of the 9th International Symposium on Hardware / Software Codesign, April 25-27 2001, Copenhagen, Denmark. Pp.230-235. Published by ACM Press, New York. ISBN: 1-58113-364-2.
- Ong, H.L.R and Pont, M.J. (2002) “The impact of instruction pointer corruption on program flow: a computational modelling study”, *Microprocessors and Microsystems*, **25**: 409-419.
- Ong, H.L.R, Pont, M.J. and Peasgood, W. (2001) “Do software-based techniques increase the reliability of embedded applications in the presence of EMI?” *Microprocessors and Microsystems*, **24**: 481-491.
- Pont, M.J. (2001) “*Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*”, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

OSCILLATOR WATCHDOG

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- You are designing an appropriate hardware foundation for your application.

Problem

How do you detect the failure of the oscillator in your embedded application (and what should you do in these circumstances)?

Background

What is a watchdog timer

To understand this pattern, you'll need to understand what a watchdog timer is: [WATCHDOG RECOVERY](#) [this paper] provides the necessary background.

If we have watchdog timers, why do we need oscillator watchdogs?

People sometimes assume that watchdog timer is a good way of detecting oscillator failure. However, a few moments thought quickly reveals that this is very rarely the case.

When the oscillator fails, the associated microcontroller will stop. **Even if (by using a watchdog timer, or some other technique) you detect that the oscillator has failed, you cannot execute any code to deal with the situation.**

In these circumstances, you may be able to improve the reliability of your system by using an *oscillator watchdog*.

Solution

Software-based techniques can be used to solve many problems in embedded applications which are traditionally handled by adding hardware: for example, switch debouncing can be carried out using external hardware components or through software (see [SWITCH INTERFACE \(SOFTWARE\)](#) [Pont, 2001, page 399]). In general, where it is possible to use software, this results in a more flexible and lower-cost solution.

In some cases, software is not an option and hardware **is** required: dealing with oscillator failure is such a case. Specifically, to implement Oscillator Watchdog, you need to select a microcontroller with on-chip 'oscillator watchdog' hardware.

Oscillator-watchdog hardware is not part of the 8051 core, and implementations vary slightly. However, the behaviour is always the same: if an oscillator failure is detected, the microcontroller is forced into a reset state: this means that port pins take on their reset values.

The state of the port pins can be crucial, since it means that the developer has a better chance of ensuring that hardware devices controlled by the processor (for example, dangerous machinery) will be shut down if the processor's oscillator fails. Please note that - as discussed in **PORT I/O** [Pont, 2001, page 174] - the port reset values must be taken into account when making use of an oscillator watchdog: failure to do so will render the use of such a watchdog meaningless.

In most cases, the microcontroller will be held in a reset state "for ever". However, most oscillator watchdogs will continue to monitor the clock input to the chip: if the main oscillator is restored, the system will leave reset and will begin operating again.

Hardware resource implications

Use of an oscillator watchdog requires no hardware resources (apart, of course, from the watchdog hardware itself).

Reliability and safety implications

Quartz-based oscillators

As discussed in **CRYSTAL OSCILLATOR** [Pont, 2001, page 54], quartz crystals form the basis of almost all stable oscillator circuits. A common source of failure for such components is physical damage (for example, through sustained vibration or from a sudden sharp impact). Use of **OSCILLATOR WATCHDOG** can be particularly effective in systems employing crystal oscillators where physical damage can occur.

In addition to their fragility, crystal oscillators also have one other feature: they can take a comparatively long time to start up (up to around 10 ms). Until this oscillator starts, most 8051 devices are "in limbo: they cannot enter their normal reset state - and the ports will be at an undefined value. 10 ms may seem a long period of time if high-power machinery is connected to a port pin.

Use of an oscillator watchdog can assist in these circumstances too. In order to drive the processor into a reset state, the oscillator watchdog needs to contain its own oscillator. This will usually be a low-frequency (and low stability) RC oscillator. Under normal circumstances, the "watchdog" behaviour will be invoked if the frequency of the main oscillator is lower than that of the RC device.

RC oscillators usually have a very rapid start time. As a result, when an oscillator watchdog is available, the RC oscillator will start first and will (with many microcontrollers) very rapidly detect that the main oscillator is not operating. The oscillator watchdog will then force the system into a reset state. The system will only leave this state when the main oscillator has become active.

The impact of such an oscillator can be very significant. For example, the time to reach a reset state in a standard 8051 can be around 10 ms. Using an Infineon C515C, with oscillator watchdog, the

maximum time to reach reset state becomes 34 μ s. This is a very significant speed improvement (approximately 300x faster).

The limitations of single-processor designs

Use of an oscillator watchdog will simply leave your system “frozen”, albeit in a well-defined state. This is much better than leaving the system in an indeterminate state. However, it may not be enough.

For example, suppose your system is controlling the brakes or steering in a moving vehicle. In these circumstances “freezing” the system may be highly undesirable. Instead, you may wish to switch in a ‘backup’ microcontroller, in order to try and return control of the vehicle to the driver.

WATCHDOG SLAVE⁷ describes one way in which you can achieve this.

Portability

Oscillator Watchdog hardware is available in only a comparatively small number of microcontrollers. Assuming the presence of such hardware will restrict the portability of your design.

Overall strengths and weaknesses

- ☺ **Can improve reliability in situations where oscillator failure occurs (for example, due to system vibration).**

Can only be used where processors have appropriate hardware support.

Related patterns and alternative solutions

Please see “Reliability and Safety Implications” for information about **WATCHDOG SLAVE**.

Example: Automotive cruise control.

Use of **OSCILLATOR WATCHDOG** is illustrated in the case study at the end of this paper.

Example: The Philips P87LPC760 family

The Philips P87LPC760 family of (8051) microcontrollers have on-chip oscillator watchdogs. Please refer to the data sheets for these devices for further details.

Example: The Infineon C868 family

The Infineon C868 (8051) microcontrollers have on-chip oscillator watchdogs. Please refer to the data sheets for these devices for further details.

Further reading

Pont, M.J. (2001) “*Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*”, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

⁷ **WATCHDOG SLAVE** is based on the “shared-clock scheduler” architecture (see Pont, 2001, Part F). The pattern is still under development, and details will be released at a future PLoP conference.

RESET RECOVERY

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. **Co-OPERATIVE SCHEDULER** [Pont, 2001, page 254]).

and:

- You are using techniques described in **WATCHDOG RECOVERY** [this paper] - or similar approaches - in order to improve the fault-tolerance of your system.
- A watchdog-induced reset has occurred.

Problem

How can you ensure that your processor re-starts safely after a watchdog-induced reset?

Background

Please see **WATCHDOG RECOVERY** [this paper] for background information on watchdog timers.

Solution

As we discussed in **WATCHDOG RECOVERY**, all of the error-recovery strategies discussed in this paper begin with a system reset, which has been caused by the overflow of a watchdog timer.

What are we trying to achieve?

Using **RESET RECOVERY** we assume that the best way to deal with an error (the presence of which is indicated by a watchdog-induced reset) is to re-start the system, in its normal configuration.

Implementation

RESET RECOVERY is very easy to implement. We require a basic watchdog timer, such as the common “1232” external device, available from various manufacturers (we show how to use this device in an example below).

Using such a device, the cause of a system reset cannot be easily determined. However, this does not present a problem when implementing **RESET RECOVERY**. After any reset, we simply start (or re-start) the scheduler and **try** to carry out the normal system operations.

Hardware resource implications

As noted in “Solution”, it is not necessary to distinguish between a ‘normal’ system reset, and a reset caused by a watchdog overflow when implementing **RESET RECOVERY**. One consequence of this is that any type of watchdog hardware (internal or external) can be used.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application. Please see [WATCHDOG RECOVERY](#) [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

The particular problem with [RESET RECOVERY](#) is that, if the error that gave rise to the watchdog reset is permanent (or long-lived), then you are likely to lose control of your system as it enters an endless loop (reset, watchdog overflow, reset, watchdog overflow, ...).

This lack of control can have disastrous consequences in many systems.

Portability

This approach can be used with any processor or microcontroller.

Overall strengths and weaknesses

☺ Very easy to implement.

MUST BE HANDLED WITH EXTREME CARE (see “Reliability and safety issues”).

Related patterns and alternative solutions

Two alternative recovery strategies are discussed in [FAIL-SILENT RECOVERY](#) [this paper] and [LIMP-HOME RECOVERY](#) [this paper].

Example: A library for the ‘1232’ external watchdog timer

In this example we present a very simple library which will allow the use of an external ‘1232’ watchdog chip.

The use of the 1232 is very straightforward:

- We wire up the watchdog to the microcontroller reset pin, as illustrated in Figure 3.
- We choose from one of three (nominal) possible timeout periods, and connect the TD pin on the 1232 to select an appropriate period (see Table 2).
- We pulse the /ST0 line on the 1232 regularly, with a pulse interval less than the timeout period.

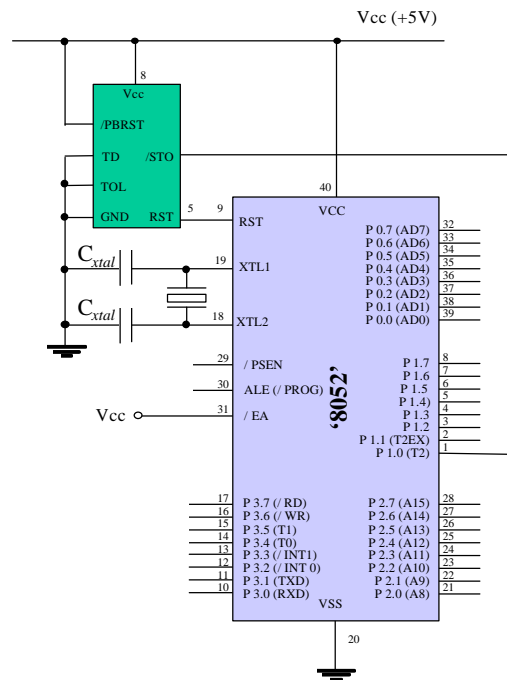


Figure 3: Simple demonstration circuit for 1232 watchdog.

	Minimum timeout	Typical timeout	Maximum timeout
TD to GND	62.5 ms	150 ms	250 ms
TD floating	250 ms	600 ms	1000 ms
TD to Vcc	500 ms	1200 ms	2000 ms

Table 2: Timings for the ubiquitous '1232; watchdog.

The key part of a suitable code library is given in Listing 4.

```

/*-----*
Dog_1232. C
-----

Watchdog timer library for external 1232 WD.
/*-----*/

#include "Dog_1232. h"

// ----- Port pins -----

// Connect 1232 (pin /ST) to the WATCHDOG_pin
sbit WATCHDOG_pin = P1^0;

// ----- Private variables -----

// Current state of the watchdog pin
static bit WATCHDOG_state_G = 0;

```

```

/*-----*-
WATCHDOG_Feed()

'Feed' the external 1232-type watchdog chip.

-----*/
void WATCHDOG_Feed(void)
{
    // Change the state of the watchdog pin
    if (WATCHDOG_state_G == 1)
    {
        WATCHDOG_state_G = 0;
        WATCHDOG_pin = 0;
    }
    else
    {
        WATCHDOG_state_G = 1;
        WATCHDOG_pin = 1;
    }
}

/*-----*-
----- END OF FILE -----
-----*/

```

Listing 4: Part of a central heating demo using 'Super Loop' and 'Hardware Watchdog'.

As noted above, most TTCS applications will use this library by feeding the watchdog at the start of the dispatcher function, as follows:

```

/*-----*-

SCH_Dispatch_Tasks()

This is the 'dispatcher' function. When a task (function)
is due to run, SCH_Dispatch_Tasks() will run it.
This function must be called (repeatedly) from the main loop.

-----*/
void SCH_Dispatch_Tasks(void)
{
    tByte Index;

    // Feed the watchdog here
    WATCHDOG_Feed();

    // Dispatches (runs) the next task (if one is ready)
    for (Index = 0; Index < SCH_MAX_TASKS; Index++)
    {
        ...
    }
}

```

Further reading

Pont, M.J. (2001) *"Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers"*, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

FAIL-SILENT RECOVERY

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. [Co-OPERATIVE SCHEDULER](#) [Pont, 2001, page 254]).

and:

- You are using techniques described in [WATCHDOG RECOVERY](#) [this paper] - or similar approaches - to improve the fault-tolerance of your system.
- A watchdog-induced reset has occurred.

and:

- Simply re-starting the system in the event of an error (as discussed in [RESET RECOVERY](#) [this paper]) is not an appropriate response, since there is a significant risk that the error is either “permanent”, or that it will re-occur. This could leave your system stuck, out of control, in an endless “reset, watchdog overflow, reset, watchdog overflow, ...” loop.

Problem

How can you ensure that your processor re-starts safely after a watchdog-induced reset?

Background

Please see [WATCHDOG RECOVERY](#) [this paper] for background information on watchdog timers.

Solution

When using Fail-Silent Watchdog, our aim is to shut the system down after a watchdog-induced reset. This type of response is referred to as “fail silent” behaviour because the processor becomes “silent” in the event of an error⁸.

As indicated in “Context”, we assume that simply re-starting the system in the event of an error (as discussed in [RESET RECOVERY](#) [this paper]) is not an appropriate response, since there is a significant risk that the error is either “permanent”, or that it will re-occur. We also assume that “freezing” the system is a known (safe) state in the event of an error will increase (rather than decrease) the reliability of our system.

⁸ This type of behaviour is often contrasted with what is known as “babbling idiot” failure (particularly in multi-processor systems), where a damaged or faulty processor remains active (and continues to interfere with the rest of the system, particularly by transmitting “noise” to other nodes). For example, use of [RESET RECOVERY](#) in the presence of sustained faults can give rise to such a “babbling idiot”.

Software architecture

FAIL-SILENT RECOVERY is implemented after every “Normal” reset as follows:

- The scheduler is started and program execution is normal.

By contrast, after a watchdog-induced reset, **FAIL-SILENT RECOVERY** will typically be implemented as follows:

- Any necessary port pins will be set to appropriate levels (for example, levels which will shut down any attached machinery).
- Where required, an error port will be set to report the cause of the error,
- All interrupts will be disabled, and,
- The system will be stopped, either by entering an endless loop or (preferably) by entering power-down or idle mode.

This effectively freezes the processor, in a known - safe - state.

Please note that power-down or idle mode is used because, in the event that the problems were caused by EMI or ESD, this is thought likely to make the system more robust in the event of another interference burst.

Hardware resource implications

The main resource implication is that a suitable watchdog timer is required.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

Please see **WATCHDOG RECOVERY** [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

Portability

The main constraint on portability is that the watchdog timer used to implement this pattern must allow the programmer to determine the cause of a reset.

Overall strengths and weaknesses

- ☺ If implemented in appropriate circumstances and with care, this pattern can help to improve the reliability of your system.

Increases the system complexity.

Related patterns and alternative solutions

Two alternative recovery strategies are discussed in [RESET RECOVERY](#) [this paper] and [LIMP-HOME RECOVERY](#) [this paper].

Example: Automotive cruise control.

Use of [FAIL-SILENT RECOVERY](#) is illustrated in the case study at the end of this paper.

Example: Fail-Silent behaviour in the Airbus A320

In the A310 Airbus, the slat and flap control computers form an ‘intelligent’ actuator sub-system. If an error is detected during landing, the wings are set to a safe state and then the actuator sub-system shuts itself down (Burns and Wellings, 1997, p.102).

Please note that the mechanisms underlying this “fail silent” behaviour are unknown: they may not be the same as the techniques described in this paper.

Further reading

Burns, A. and Wellings, A. (1997) “*Real-time systems and programming languages*”, Addison-Wesley.

Pont, M.J. (2001) “*Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*”, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

LIMP-HOME RECOVERY

Context

- You are developing a single-processor embedded application a member of the 8051 family of microcontrollers (or similar hardware).
- The application has a time-triggered architecture, constructed using a scheduler (e.g. **Co-OPERATIVE SCHEDULER** [Pont, 2001, page 254]).

and:

- You are using techniques described in **WATCHDOG RECOVERY** [this paper] - or similar approaches - to improve the fault-tolerance of your system.
- A watchdog-induced reset has occurred.

and:

- Simply re-starting the system in the event of an error (as discussed in **RESET RECOVERY** [this paper]) is not an appropriate response, since there is a significant risk that the error is either “permanent”, or that it will re-occur. This could leave your system stuck, out of control, in an endless “reset, watchdog overflow, reset, watchdog overflow, ...” loop.
- “Freezing” the system in a known state in the event of an error (as discussed in **FAIL-SILENT RECOVERY** [this paper]) is not appropriate behaviour, as it is likely to decrease (rather than increase) the reliability of our system. For example, completely shutting down a piece of essential medical equipment is something we would wish to do only as a last resort. Similarly, shutting down an aircraft control system during takeoff is (highly) undesirable.

Problem

How can you ensure that your processor re-starts safely after a watchdog-induced reset?

Background

Please see **WATCHDOG RECOVERY** [this paper] for background information on watchdog timers.

Solution

As we discussed in **WATCHDOG RECOVERY**, all of the error-recovery strategies presented in this paper begin with a system reset, which has been caused by the overflow of a watchdog timer.

What are we trying to achieve?

In using **LIMP-HOME RECOVERY**, we make four assumptions about our system:

- A watchdog-induced reset indicates that a significant error has occurred.
- Although a full (normal) re-start is considered too risky, it may still be possible to let the system “limp home” by running a simple version of the original algorithm.

Overall, in using this pattern, we are looking for ways of ensuring that the system continues to function - even in a very limited way - in the event of an error.

Software architecture

LIMP-HOME RECOVERY is implemented after every “Normal” reset as follows:

- The scheduler is started and program execution is normal.

By contrast, after a watchdog-induced reset, **LIMP-HOME RECOVERY** will typically be implemented as follows:

- The scheduler will not be started.
- A simple version of the original algorithm will be executed.

Hardware resource implications

At the processor level, the main resource implication is that a suitable watchdog timer is required.

However (rather more substantial) hardware costs may also arise if “backup” hardware (such as sensors or actuators) is required in the limp-home system.

Reliability and safety implications

In safety-related or safety-critical systems, this pattern should not be implemented before a complete risk-assessment study has been conducted (by suitably-qualified individuals).

Successful use of this pattern requires a full understanding of the errors that are likely to be detected by your error-detection strategies (and those that will be missed), plus an equal understanding of the recovery strategy that you have chosen to implement. Without a complete investigation of these issues, you cannot be sure that implementation of the pattern you will increase (rather than decrease) the reliability of your application.

Please see **WATCHDOG RECOVERY** [this paper] for further discussion of the reliability and safety implications associated with watchdog timers.

Portability

The main constraint on portability is that the watchdog timer used to implement this pattern must allow the programmer to determine the cause of a reset.

Overall strengths and weaknesses

- ☺ **If implemented in appropriate circumstances and with care, this pattern can help to improve the reliability of your system.**

Increases the system complexity.

Related patterns and alternative solutions

Two alternative recovery strategies are discussed in [RESET RECOVERY](#) [this paper] and [FAIL-SILENT RECOVERY](#) [this paper].

Some alternative patterns for fault tolerance and error recovery which may be of interest were presented recently by Saridakis (2002).

Example: An outline design for a reliable steer-by-wire application

Suppose that an automotive steer-by-wire system has been created that runs a single task, every 10 ms. We will assume that the system is being monitored to check for task over-runs (see [SCHEDULER WATCHDOG](#) [this paper]). We will also assume that the system has been well designed, and has appropriate timeout code, etc, implemented.

Suppose that a passenger car using this system is being driven on a motorway, and that an error is detected, resulting in a watchdog reset. What recovery behaviour should be implemented?

We could simply re-start the scheduler and “hope for the best”. However, this form of “reset recovery” is probably not appropriate. In this case, if we simply perform a reset, we may leave the driver without control of their vehicle (see [RESET RECOVERY](#) [this paper]).

Instead, we could implement a fail-silent strategy. However, this - again - would be likely to leave the driver unable to control the steering, which would probably be left in a fixed position.

In these circumstances, we may wish to assess a limp-home control strategy. In this case, a suitable strategy might involve a code structure like this:

```
while(1)
{
    Update_basic_steering_control ();
    Software_delay_10ms();
}
```

This is a basic software architecture (based on [SUPER LOOP](#) [Pont, 2001, p.162]).

In creating this version, we have avoided use of the scheduler code. We might also wish to use a different (simpler) control algorithm at the heart of this system. For example, the main control algorithm may use measurements of the current speed, in order to ensure a smooth response even when the vehicle is moving rapidly. We could omit this feature in the “limp home” version.

Of course, simply using a different software implementation may still not be enough. For example, in our steer-by-wire application, we may have a position sensor (attached to the steering column) and an appropriate form of DC motor (attached to the steering rack). Both the sensor and the actuator would then be linked to the processor.

When designing the limp-home controller, we would like to have an additional sensor and actuator, which are - as far as possible - independent of the components used in the main (scheduled) system.

This option makes sense because it is likely to maximise the chances that the Slave node will operate correctly when it takes over (for example, it could have been a failure of the DC motor that made it necessary to shut down the original processing).

This approach has two main implications:

1. The hardware **must** ‘fail silently’: for example, if we did add a backup motor to the steering rack, this would be little use if the main motor ‘seized’ when the scheduler task was shut down.

Note that there may be costs associated with obtaining this behaviour. For example, we may need to add some kind of clutch assembly to the motor output, to ensure that it could be disconnected in the event of a motor jam. However, such a decision would need to be made only after a full risk assessment. For example, it would not make sense to add a clutch unit if a failure of this unit (leading to a loss of control of steering) was more likely than a motor seizure.

2. The cost of hardware duplication can be significant, and will often be considerably higher than the cost of a duplicated processor: this may make this approach economically unfeasible.

When costs are too high, sometimes a compromise can prove effective. For example, in the steering system, we might consider adding a second set of windings to the motor for use by the Slave (rather than adding a complete new motor assembly). Again, such a decision should be made only after a full risk assessment.

Further reading

Pont, M.J. (2001) “*Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers*”, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

Saridakis, T. (2002) “A system of patterns for fault tolerance”, paper presented at EuroPloP 2002, July 2002, Germany.

CASE STUDY:

Implementing an automotive cruise-control system

We consider the use of many of the patterns presented in this paper in an automotive cruise-control system (CCS)⁹. The CCS will (it is assumed) be required to take over the task of maintaining the vehicle at a constant speed even while negotiating a varying terrain involving, for example, hills or corners in the road.

The CCS will be implemented using a single Infineon C515C microcontroller (an example of an **EXTENDED 8051** [Pont, 2001, p.46]).

To operate the CCS, the driver will do the following:

- Enter the required speed on a rotary speed-control dial¹⁰. This will be read, via a potentiometer and analogue-to-digital converter: see **ONE-SHOT ADC** [Pont, 2001, p.757].
- Press a “cruise” switch: see **SWITCH INTERFACE (SOFTWARE)** [Pont, 2001, p.399].

The CCS will stop cruising “on demand” (and return control to the driver), if:

- The driver presses the cruise switch again, or,
- The driver touches the brake pedal (which is also assumed to have a switch attached).

While cruising, the CCS will then repeatedly perform the following tasks:

- It will check the status of the brake pedal and the cruise switch, to ensure that the driver wishes to keep cruising: if not, it will return control to the driver.
- It will check for system errors (see below): if an error is detected, it will “fail silently” and return control to the driver.
- It will read the desired cruise speed (entered by the driver).
- It will measure the current vehicle speed by counting pulses from magnetic sensors attached to two wheels (see **HARDWARE PULSE COUNT** [Pont, 2001, p.728]).
- It will use **PID CONTROLLER** [Pont, 2001, p.861] to determine the required throttle setting.
- It will use an appropriate hardware interface to control the throttle (possibly using **HARDWARE PWM** [Pont, 2001, p.808] and **MOSFET DRIVER** [Pont, 2001, p.139]).

The error-handling functions will use many of the patterns presented in this paper:

- **WATCHDOG RECOVERY** [this paper] will provide the basic framework for the error handling.
- **SCHEDULER WATCHDOG** [this paper] will provide a means of detecting scheduling errors.
- **PROGRAM-FLOW WATCHDOG** [this paper] may be appropriate here. Program-flow errors can occur as a result of electromagnetic interference, and the typical passenger car contains numerous

⁹ This application is builds on the simple PID-based CCS described in **PID CONTROLLER** [Pont, 2001, p.861].

¹⁰ Note that the driver is assumed to be able to alter the cruise speed while cruising, by means of this dial.

electromechanical devices such as contact breakers, alternators, relays and ignition coils that are excellent sources of high-energy, wideband, electromagnetic noise that is capable of corrupting electronic circuits.

- **OSCILLATOR WATCHDOG** [this paper] will provide a means of dealing with oscillator failures.
- We need to ensure that the speed sensors, throttle actuator, speed-control dial, cruise switch and brake switch all operate correctly while cruising. If any of these devices fail then, using the approach discussed in **WATCHDOG RECOVERY** [this paper], we will force a watchdog reset.

There are various checks we can perform. For example: (1) the speed sensors can be tested by looking for sudden (impossibly rapid) changes in speed, and checking that neither sensor reads zero while cruising. (2) The throttle actuators can be checked by including a sensor to detect that the throttle setting actually changes on demand. (3) The speed-control dial can employ a dual-gang potentiometer to provide two measures of the desired speed: if they don't agree, or are out of range, we assume there is an error. (4) The switches can be "doubled up", so that both a "normally open" and "normally closed" switch is used in each case: see **SWITCH INTERFACE (SOFTWARE)** [Pont, 2001, p.399] for further details.

- In terms of recovery behaviour, we assume that the driver is able to take over control of the vehicle in the event of an error. This means that neither **RESET RECOVERY** [this paper] nor **LIMP-HOME RECOVERY** [this paper] will be appropriate in this system and that, instead **FAIL-SILENT RECOVERY** [this paper] will be used.

The basic system architecture will be based on **CO-OPERATIVE SCHEDULER** [Pont, 2001, p.255]: the core of the resulting main function is shown in Listing 5.

```
void main(void)
{
    // Determine cause of system reset
    // (***) IF RESET CAUSED BY WATCHDOG, THEN FAIL SILENTLY HERE (***)
    CCS_Check_Cause_Of_Reset();

    // Set up the scheduler
    SCH_Init_T2();

    // Prepare the CCS
    CCS_Init();

    // Add the tasks
    // TIMING IS IN TICKS (10 ms tick interval)
    SCH_Add_Task(CCS_Read_Cruise_and_Brake_Switches, 1, 2);
    SCH_Add_Task(CCS_Get_Required_Speed, 2, 10);
    SCH_Add_Task(CCS_Calculate_Current_Speed, 3, 10);
    SCH_Add_Task(CCS_Compute_and_Apply_Throttle, 4, 10);

    // All tasks added: start running the scheduler
    SCH_Start();

    while(1)
    {
        SCH_Dispatch_Tasks();
    }
}
```

Listing 5: The function main() for the CCS demo.

At the start of the program, the function `CCS_Check_Cause_Of_Reset ()` is called: this function is key to the watchdog-based error handling. A possible implementation of this function (assuming the use of Infineon C515C hardware and the Keil compiler) is given in Listing 6.

```
void CCS_Check_Cause_Of_Reset(void)
{
    // Determine if reset was caused by watchdog overflow (C515C)
    if (IPO & 0x40)
    {
        // WDTS flag is set - reset *was* caused by watchdog
        Watchdog_reset_G = 1;

        // We shut the system down here - fail silently

        // Try to tell the driver
        CCS_Beep()

        // Disable any / all interrupts
        EA = 0;

        // Enter power-down mode
        PCON |= 0x02;
        PCON |= 0x40;

        // Safety net - just in case we are thrown out of power-down
        // (Should NEVER be needed)
        while (1);
    }

    // -----
    // If we get this far, then this was a "normal" reset
    // - carry on and start the scheduler, etc.
    // -----
}
```

Listing 6: A function for handling watchdog-related errors in the CCS system. See text for details.