

Two Simple Patterns to Support the Development of Reliable Embedded Systems

Chisanga Mwelwa¹ and Michael J. Pont {cm55, M.Pont}@le.ac.uk

*Embedded Systems Laboratory, Department of Engineering, University of Leicester,
University Road, LEICESTER LE1 7RH, UK*

<http://www.le.ac.uk/eg/embedded/>

Introduction

As the title suggests, this paper is concerned with the development of software for embedded systems. Typical application areas for this type of software range from passenger cars and aircraft through to common domestic equipment, such as washing machines and microwave ovens.

We have previously described a “language” consisting of more than eighty patterns, which will be referred to in this paper as the “PRES Collection²” (see Appendix A). This language is intended to support the development of reliable embedded systems using low-cost embedded hardware with severe memory constraints. Typical implementations will employ embedded microcontrollers with a few kilobytes of available RAM.

Over the last few years, we have had the chance to observe many people use this collection when developing a range of different systems: these observations have included industrial projects and various university research projects. In this paper, we present two patterns that have resulted from these observations: HEARTBEAT LED and ERROR LED.

¹ Primary contact

² Our original patterns focused on “time-triggered” designs and were known as the “PTTES collection” (after the original book title: “Patterns for Time-Triggered Embedded Systems”). Since then, this collection has expanded and has subsequently been revised, and – while the focus remains on reliable design – not all of the patterns are time-triggered. The collection has therefore been renamed the “PRES collection” (Patterns for Reliable Embedded Systems).

The two patterns are related. HEARTBEAT LED provides a simple, low-cost mechanism for providing feedback on the overall health of your system: if the LED is flashing, the core of the system is running correctly. ERROR LED goes one step further and provides a mechanism for error reporting.

Format

The other patterns referred to in this paper are from the PRES collection and are presented in a distinctive style e.g. CO-OPERATIVE SCHEDULER. See Appendix A for a complete list of these patterns.

Acknowledgements

The authors are grateful to Alan O' Callaghan (our Shepherd at Viking PLoP 2003) for comments and suggestions on the first drafts of this paper. We are also grateful to the participants in our workshop at Viking PLoP (Neil Harrison, Klaus Marquardt, Bernhard Grone and Peter Tabeling) who all provided further useful comments.

Copyright

Copyright © 2003 Chisanga Mwelwa and Michael J. Pont. Permission is granted to copy this paper for the purposes of Viking PLoP 2003. All other rights are reserved.

HEARTBEAT LED

Context

- You are developing (or maintaining) an embedded application based on a microcontroller or microprocessor.
- You are programming in C (or a similar language).
- Your application has an architecture based on some form of scheduler.

Problem

How can you tell, at a glance, if your system is “alive”?

Design constraints

Many embedded systems have little or no user interface. There is not generally a screen on which you can display error messages or warnings to the user.

If you are working on a system prototype, or performing maintenance in the field, how can you tell that the system is “alive” – that it has power and (at least) the scheduler is running?

You could, of course, hook up a debugging link (e.g. a JTAG link), or a simpler serial link (based on RS-232), but this takes time and including suitable ports on your production system may not be practical or cost effective. Often a very simple, low-cost solution is required.

Solution

Every time we implement an embedded system, the first task we include is one that flashes a “heartbeat” LED. Wherever possible, this LED stays with the system, right into production.

We tend to use a 50% duty cycle and a frequency of 0.5 Hz (that is, the LED runs continuously, on for one second, off for one second, and so on) but this is – of course – up to you.

Use of this simple technique provides the following key benefit:

- The development team, the maintenance team and, where appropriate, the users, can tell at a glance that the system has power, and that the scheduler is operating normally.

In addition, during development, there are two less significant (but still useful) side benefits:

- After a little practice, the developer can tell “intuitively” - by watching the LED - whether the scheduler is running at the correct rate: if it is not, it may be that the timers have not been initialised correctly, or that an incorrect crystal frequency has been assumed.
- By adding the “Heartbeat” task to the scheduler array *after* all other tasks have been included, the developer can tell immediately if the task array is large enough to match the needs of the application (if the array is not large enough, the LED will never flash).

Implementation

Numerous possible implementations are possible. We give an example of one possible version at the end of this pattern.

Reliability and safety implications

Use of this simple technique may help to improve system reliability since it provides those developing the system with an indication of its health throughout the development lifecycle.

Hardware requirements

HEARTBEAT LED has minimal hardware requirements. The only requirements are a port pin connected to an appropriate LED (with a matching resistor if required).

Cost implications

As noted above, the hardware requirements are very limited. The time taken to implement this pattern is also likely to be minimal. Overall, the costs are very low.

Maintenance

HEARTBEAT LED gives a developer an indication of a systems “health” during its maintenance.

Portability

Highly portable – can be implemented on a wide range of hardware platforms.

Related patterns and alternative solutions

See NAKED LED³ for hardware details.

Overall strengths and weaknesses

- ☺ HEARTBEAT LED provides a simple, low-cost way of determining whether your system is “alive”.
- ☹ Uses a port pin and associated LED hardware.

Example: A “Heartbeat LED” task for an 8051 microcontroller

```
/*-----*  
    Heartbeat_LED.C  
-----  
    Simple 'Heartbeat LED' task for an Infineon C515C microcontroller.  
    If everything is OK, flashes at 0.5 Hz  
-----*/
```

³ Page 254 in M. J. Pont (2001), *Patterns for Time-Triggered Embedded Systems*, Addison-Wesley

```

#include "Main.H"
#include "Port.H"
#include "Heartbeat_LED.H"

// ----- Private variable definitions -----

static bit Heartbeat_led_state_G;

/*-----*

HEARTBEAT_LED_Init()
Prepare for HEARTBEAT_Update() task.

-----*/
void HEARTBEAT_LED_Init(void)
{
    Heartbeat_led_state_G = 0;
}

/*-----*

HEARTBEAT_LED_Update()

Flashes an LED on a specified port pin.

Must schedule at twice the required flash rate: thus, for 0.5 Hz
flash (on for 1 second, off for 1 second) must schedule at 1 Hz.

-----*/
void HEARTBEAT_LED_Update(void)
{
    // Change the LED from OFF to ON (or vice versa)
    if (Heartbeat_led_state_G == 1)
    {
        Heartbeat_led_state_G = 0;
        Heartbeat_led_pin = 0;
    }
    else
    {
        Heartbeat_led_state_G = 1;
        Heartbeat_led_pin = 1;
    }
}

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

```

Listing 1: The source file defining the task responsible for the flashing LED

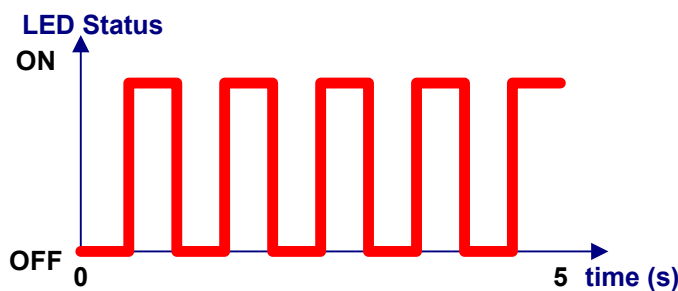


Figure 1: Graph depicting flash rate of an implemented HEARTBEAT LED task scheduled to run every 1 second

ERROR LED

Context

- You have implemented HEARTBEAT LED (see HEARTBEAT LED).

And

- You now require a means of reporting errors.

Problem

If your embedded system is not working correctly, how can you tell what is wrong?

Design constraints

See HEARTBEAT LED for the design constraints.

HEARTBEAT LED can provide a very cost-effective way of telling whether your system is “alive”. If the system is functioning, but has detected some errors, a HEARTBEAT LED may not be of great help.

How can you report errors, without significantly increasing the system (or development) costs?

Solution

To implement ERROR LED a single LED is used to report error codes to the developer or (if appropriate) the user. In most cases, we like to base the Error LED on a Heartbeat LED so that, if there are no errors, we see the usual (comforting) 0.5 Hz signal. If there is a problem, the display changes, and – by observing the different pulse rates – we can often identify the cause.

Implementation

There are many possible ways of implementing ERROR LED.

We use a (global) error variable, and maintain a list of error codes (in Main.H). In the event of an error, we adjust the output of the ERROR LED accordingly.

We give an example of a suitable implementation at the end of this pattern.

Reliability and safety implications

Most forms of error reporting – like ERROR LED – provide a means of improving system reliability.

Hardware requirements

See HEARTBEAT LED hardware requirements.

Cost implications

Implementing a basic implementation of ERROR LED will cost you little more than implementing HEARTBEAT LED. However, it takes time to include error reporting in your program code, and this may add to the development costs.

Maintenance

ERROR LED can be very valuable during system maintenance as it can be used to debug reported bugs.

Portability

Highly portable – can be implemented on a wide range of hardware platforms.

Related patterns and alternative solutions

See HEARTBEAT LED for the related patterns.

As an alternative solution one could easily substitute a buzzer for the LED, and thereby draw the attention of developers (or users) to errors using various sounds or different pulse frequencies.

Overall strengths and weaknesses

- ☺ ERROR LED provides a low-cost, non-invasive, means of error reporting.
- ☹ Uses a port pin and associated LED hardware.
- ☹ Adding error reporting takes time and hence may increase development costs.

Example: An “Error LED” task for an ARM microcontroller

```
/*-----*-
Error_LED.C
-----*

Simple 'Error LED' task for a Philips LPC2106
ARM microcontroller.

If everything is OK, flashes at 0.5 Hz

If there is an error code active, this is displayed.

-----*/
#include "Main.H"
#include "Port.H"

#include "Error_LED.H"

// see Scheduler for definition
extern int Error_code_G;

/*-----*-

ERROR_LED_Init()

Prepare for ERROR_LED_Update() function.

-----*/
void ERROR_LED_Init(void)
{
    // Set up Heartbeat_pin as GPIO
    PINSEL0 &= ~Heartbeat_pin;
```

```

// Set Heartbeat_pin to output mode
IODIR |= Heartbeat_pin;
}

/*-----*/

ERROR_LED_Update()

Flashes at 0.5 Hz if error code is zero.

Otherwise, displays error code.

Must schedule every second (soft deadline).

-----*/
void ERROR_LED_Update(void)
{
    static int LED_state = 0;
    static int Error_state = 0;

    if (Error_code_G == 0)
    {
        // No errors recorded
        // - just flash at 0.5 Hz

        // Change the LED from OFF to ON (or vice versa)
        if (LED_state == 1)
        {
            LED_state = 0;
            IOCLR = Error_pin; // Set to 0
        }
        else
        {
            LED_state = 1;
            IOSET = Error_pin; // Set to 1
        }
        return;
    }

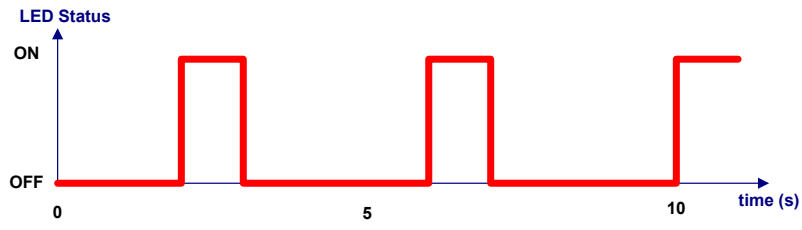
    // If we are here, there is an error code ...
    Error_state++;

    if (Error_state < Error_code_G*2)
    {
        LED_state = 0;
        IOCLR = Error_pin; // Set to 0
    }
    else
    {
        if (Error_state < Error_code_G*4)
        {
            // Change the LED from OFF to ON (or vice versa)
            if (LED_state == 1)
            {
                LED_state = 0;
                IOCLR = Error_pin; // Set to 0
            }
            else
            {
                LED_state = 1;
                IOSET = Error_pin; // Set to 1
            }
        }
        else
        {
            Error_state = 0;
        }
    }
}

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

```

Listing 2: The source file defining the task responsible for the flashing error LED



*Figure 2: Graph depicting flash rate of implemented ERROR LED when an error has been flagged for an error code set to 1 (compare this flash rate with that of HEARTBEAT LED in **Figure 1**)*

Appendix A: The PRES Collection

A complete list of the current patterns in the PRES collection is given in Table 1.

The present version of this collection consists of 71 patterns (see: M. J. Pont (2001), *"Patterns for Time-Triggered Embedded Systems"*, Addison-Wesley) plus a further 7 patterns from Pont and Ong (2002). Please note that the 2002 patterns are identified thus [VP]. Please also note that the later patterns, together, form a replacement for HARDWARE WATCHDOG (presented in *"Patterns for Time-Triggered Embedded Systems"*): HARDWARE WATCHDOG is therefore not listed in this table.

Table 1: The "PRES Collection"

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