DESIGNING AND IMPLEMENTING RELIABLE EMBEDDED SYSTEMS USING PATTERNS

Michael J. Pont

Control & Instrumentation Research Section, Department of Engineering, University of Leicester, Leicester, ENGLAND LE1 7RH, UK. [M.Pont@leicester.ac.uk]

Introduction

Four patterns intended to support the development of embedded systems are presented in this paper. These patterns are written for developers of software, and some associated hardware, for the ubiquitous 8051 family of microcontrollers. There are primarily intended to assist software developers who have experience in 'desktop' software development (in C or a related language) in the process of adapting to an embedded environment.

The patterns discussed here are as follows:

- RC RESET
- SIMPLE TIMER DELAY
- SOFTWARE SWITCH DEBOUNCER
- FUNCTION TOKEN

The four examples presented here are adapted from a larger collection of more than 50 patterns [Pont, in press], which cover techniques for interfacing (8051) microcontrollers to various input and output devices (including switches, keypads, analogue inputs). They also address various more advanced topics, such as the use of fuzzy logic and neural networks, schedulers, and various forms of serial communication technology, including the CAN bus. The overall emphasis in this collection is on the development of highly reliable embedded systems which continue to perform safely, for example, in the presence of high levels of electromagnetic interference (EMI), or following the failure of sensors, actuators or other hardware components.

For consistency, all of the code samples are written in C, for the industry-standard Keil C51 compiler: however, the patterns (and code) are readily adaptable to other software and hardware environments.

Acknowledgements

I am grateful to Ward Cunningham (my 'shepherd' at EuroPlop 1999) for comments and suggestions on the first drafts of this paper. I am also grateful to Kent Beck and others in my workshop session at this conference for detailed comments, suggestions and support.

Copyright

References

Pont, M.J. (in press) *"Patterns for reliable embedded systems: Rapid software development in C for the 8051 family of microcontrollers"*, Due for publication by Addison-Wesley Longman, June 2000.

RC RESET

Context

You are designing the hardware for an embedded application based on an 8051 microcontroller or similar device.

Problem

How do you ensure that your (8051) microcontroller operates correctly when power is supplied? How do you provide the capability to 'reset' the microcontroller while it is operating?

Background

When using personal computer (PC), we expect to 'double click' a mouse button to execute a program (in a 'Windows' environment), or to type a program name at an appropriate prompt (in a text-based environment). The popularity of PCs derives, in large part, from the ability of such devices to run an enormous range of different programs, as required by the user.

By contrast, in an embedded environment, most hardware runs only a single program, which is executed whenever power is supplied to the microcontroller. To enable this to happen, the microcontroller always begins execution at a known address (0000h in the case of the 8051 family), after it is reset. The reset process which precedes the execution of the user code is not trivial: the underlying hardware is complex and a manufacturer-defined 'reset routine' must be run to place this hardware into an appropriate initial state. Running this routine takes time, and requires that the microcontroller's oscillator is operating.

To trigger the reset operation, the original members of the 8051 family have a 'RESET' pin. When this is held at Logic 0, the chip will run normally. If, while the oscillator is running, this pin is held at Logic 1 for two (or more) machine cycles, the microcontroller will be reset.

Note that, if the reset operation is not completed correctly, the microcontroller will usually not operate at all: in rare circumstances, it may operate, but not correctly. In either event, there is usually nothing that you can do, in software, to recover control of the system. Clearly, therefore, ensuring correct reset operation is a crucial part of any application.

Solution

Various techniques may be used to ensure that - when power is applied to your 8051-based application - the reset process discussed in 'Background' is automatically carried out. The most widely used techniques are based on the use of an external capacitor and resistor: these techniques are considered in detail here.

RC reset circuits

A typical RC reset circuit is as shown in Figure 1.

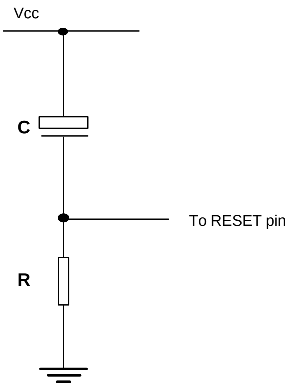

Figure 1: An (active high) RC reset circuit.

The circuit in Figure 1 operates as follows. We assume that Vcc is initially at 0V (that is, the power has not been applied to the system), and that the capacitor C is fully discharged. When power is applied, the capacitor will begin to charge. Initially the voltage across the capacitor will be 0V, and - therefore - the voltage across the resistor (and the voltage at the RESET pin) will be Vcc: this is a Logic 1 value. Gradually, the capacitor will charge and its voltage will rise, eventually to Vcc: at this time, the voltage at the reset pin will be 0V.

In the real system, the microcontroller's input voltage threshold is around 1.1 - 1.3V[1]: input voltages below this level are interpreted as Logic 0 and voltages above this level are interpreted as Logic 1. Thus, the reset operation will continue until the voltage at the RESET pin falls to a level of around 1.2V.

We can use this information to calculate the required values of R and C. To make this calculation, we use the fact that the capacitor in Figure 1 will have a voltage (*Vcap*) at time (*t*) seconds after it begins charging, given by Equation 1.

$$V_{cap} = V_{cc}(1 - e^{-t/RC})$$

Equation 1: The voltage across the capacitor in Figure 1 as a function of time. See text for details.

Note that Equation 1 assumes that the capacitor begins charging at a voltage of 0, and that the power supply voltage increases from 0V to Vcc in an instantaneous 'step' (rather than a slow ramp): these assumptions, although often made, are frequently invalid: see 'Safety and reliability issues' below for a discussion of these issues.

The Intel 8051 data sheet recommends values of 8.2K for R and 10uf for C when this form of reset circuit is used. Figure 2 substitutes these values into Equation 1 and plots the result over a period of 500ms.

[1] The data sheet for you chosen microcontroller will provide a precise value, if you require it.

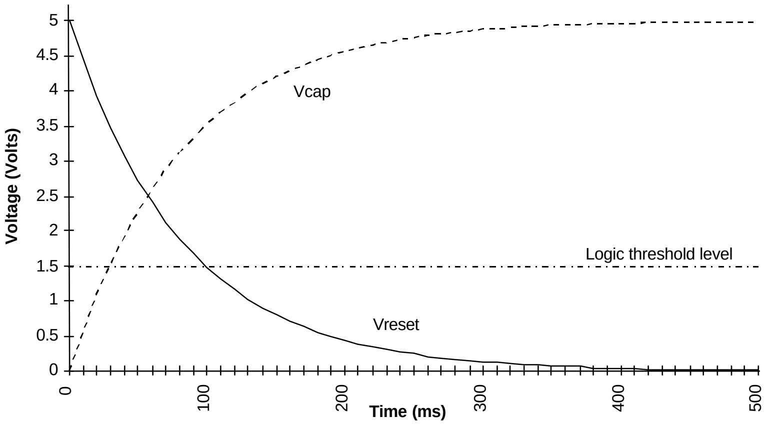

Figure 2: An example of the behaviour of an RC reset circuit, using standard component values and an ideal power supply.

When looking at Figure 2, remember that all 8051s complete their reset operation in 24 oscillator periods or less: if we use a 10 MHz oscillator, this is a maximum period of 0.0024 ms: by contrast, the recommended reset circuit takes around 100 ms to complete the reset operation. This may seem like an excessive reset period but, for reasons discussed under ‘Safety and reliability issues’, allowing approximately 100 ms for the reset is generally good practice.

Choosing values of R and C

If, having reviewed all aspects of this pattern, you have decided to use an RC-based reset circuit, what values of R and C should you use?

It is certainly not obvious how to determine suitable values of R and C directly from Equation 1. We can, however, simplify matters by noting that the product of R (in Ohms) multiplied by C (in farads) is known as the ‘time constant’ (in seconds) of this form of RC circuit. This time constant is the time taken for the capacitor to be charged to 60% of its final voltage. Thus, with a 5V supply and the circuit in Figure 1, this is the time taken for the capacitor voltage to reach 3V, and - therefore - the voltage at the reset pin to reach 2V (that is, Vcc - 3V): this is still high enough (because it is greater than 1.2V, as discussed above) to ensure that the device is in reset mode. As long as the device is still in this mode until approximately 1ms after the power supply reaches Vcc (typically around 100ms after starting: see ‘Safety and reliability issues’), the device will be reset correctly.

A basic rule of thumb, therefore, is that the RC time constant should be approximately 100 ms, and values of R and C chosen to meet this requirement will usually ensure effective reset operation (Equation 2).

$$RC \geq 100\text{ms}$$

Equation 2: A ‘rule of thumb’ for calculating appropriate RC values.

A suitable RC reset circuit satisfying these conditions is shown in Figure 3.

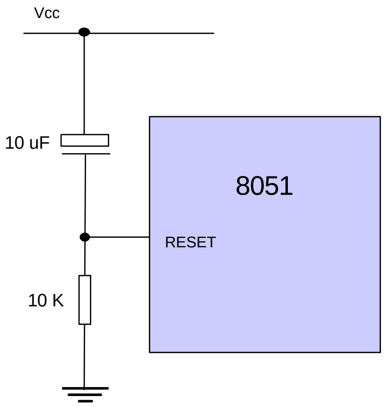

Figure 3: A suitable (active high) RC reset circuit.

We can summarise the key material in this section as follows:

- A combination of a 10K resistor and a 10uf capacitor in a RC reset circuit gives a 100 ms time constant. Bearing in mind the general limitations of RC reset circuits (see 'Safety and reliability issues'), this value is suitable for the majority of 8051-based systems.
- The standard 8K2, 10uf RC reset combination gives a time constant of 82 ms: this is generally adequate.
- Values of 1K and 10uf (which appear in some books, etc) provide a time constant of only 10ms: these values will not provide a reliable reset operation with all power supplies.

Adding a RESET button

In some systems, it is helpful to have a reset button, to force a hardware reset. This is easy to achieve. Figure 4 extends the basic shows a suitable circuit.

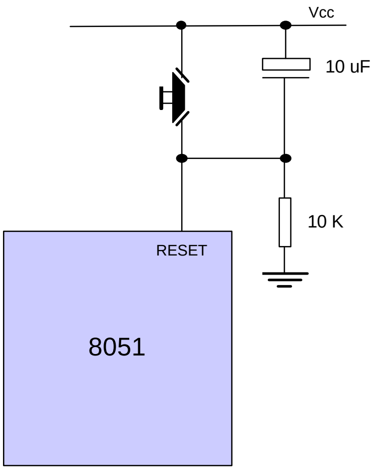

Figure 4: A reset circuit (active high) with reset switch.

Note that the reset button pulls the RESET pin (assumed to be active high: see 'Portability') to Vcc. Note also that this button also discharges the capacitor, ensuring that - when the switch is released - the proper reset process will be carried out.

Reliability and safety issues

There are a number of reliability and safety issues related to the use of RC reset circuits. The key issues are considered in this section.

Time taken for power supply to reach steady state

Suppose you are developing an embedded industrial control system, and you want to ensure that the system begins operating as soon as possible after power is applied. You note (from 'Solution') that the reset process on an 8051 microcontroller (with a 10 MHz oscillator) will take 0.0024 ms. You conclude that, allowing a reset period of 1 ms (rather than the 100 ms figure recommended above) will provide sufficient margin for error.

Suppose you adjust the values to reduce the reset period to around 1 ms. For example, Figure 5 shows the result of using a 0.1uf capacitor and a 6K7 resistor.

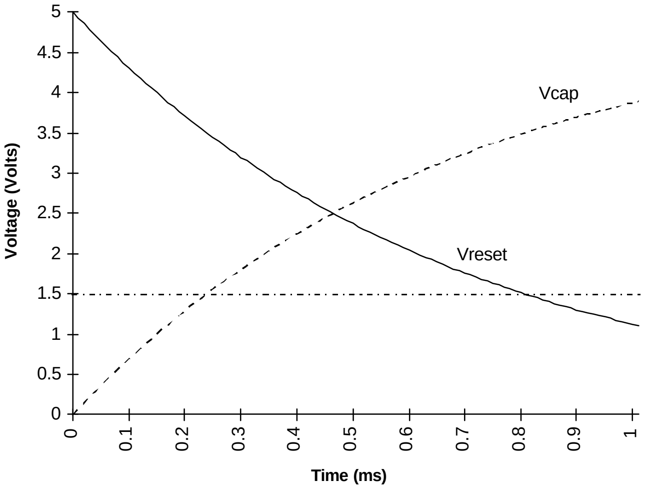

Figure 5: Using a rapid RC reset circuit: see text for details.

This combination of values may, **sometimes**, work: but in most systems it will fail. The reason is that real power supplies do not switch instantly from 0V to their specified output voltage: in reality, many supplies take 50ms or 100 ms (or more) to reach this voltage when first switched on. You need to allow for this 'ramped' voltage input in your design.

If the supply voltage increases slowly, then the capacitor in your RC reset circuit will comparatively quickly charge up, and will simply 'follow' the increasing power supply voltage. As a result, Vreset will be held at Logic 0 many milliseconds before the chip reaches its operating voltage (~5V). Therefore, the chip will only be ready to run its reset routine after the RESET signal is complete, and no reset will be performed. Your application will therefore not start correctly.

If you really must have a rapid reset, and you have control over the design of the power supply, then there are various ways of dealing with this problem. You may, for example, be able to increase the transformer capacity, or reduce the values of these filter capacitors: such issues are discussed in the pattern MAIN POWER SUPPLY (see Pont, in press). Note, however, that tying the operation of a device to a particular power supply will make your system design much less portable. If, to give a common example, your company subsequently decides to 'outsource' the power supply supply, or to use a single power supply across a range of different boards, you can quickly run into difficulties.

Time taken for oscillator to start

If the start of the (crystal) oscillator in your circuit is delayed, then the RC reset cycle may be completed before the oscillation begins. If this happens, the chip will not be reset.

Typical start-up times for crystal oscillators are 0.1 to 10 ms: however, the time taken for a crystal oscillator to start operating depends on it being mounted correctly, and having appropriate capacitors. These issues are discussed in detail in the pattern CRYSTAL OSCILLATOR (see Pont, in press).

Handling 'brownouts' and other power disruptions

Potential problems with reset circuits do not, unfortunately, only arise when embedded devices are first powered up. Consider, for example, Figure 6. This shows changes in the system supply voltage (nominally 5V) in the presence of two problems. The first of these (at time = 4 seconds) is a simple power 'glitch', where the supply voltage drops briefly to 0V. The second problem (beginning at time = 14 seconds) is a 'brownout' condition: this means that the (mains) supply voltage is reduced significantly for a period of time, but the supply does not fail completely. These types of fault are comparatively common in mains-powered systems.

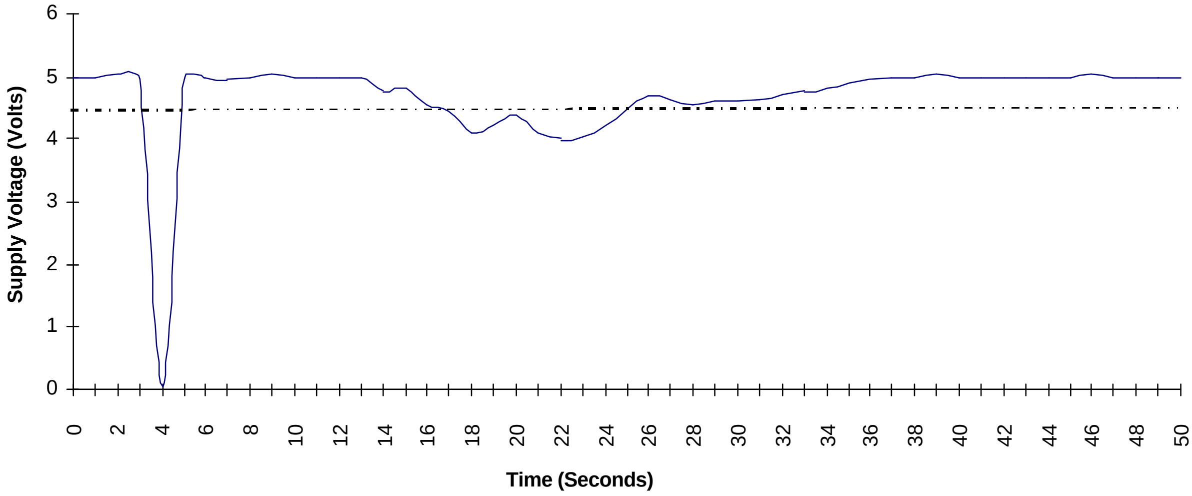

Figure 6: Examples of voltage fluctuations caused by 'glitches' and 'brownouts': see text for details.

To ensure our system operates in a predictable manner, we need to be able deal with each supply problem. In most systems, the power glitch will not pose a significant hazard. When the power fails, the voltage drops rapidly (to 0V), and the system will stop operating. When the power returns, the system will be reset in the usual way.

The ‘brownout’ is potentially more problematic. If the supply voltage drops below the minimum operating voltage (typically 4.5V for most members of the family, although this varies), the microcontroller will stop operating. If the voltage then rises again, the microcontroller will begin to operate again however, if using a simple RC reset, the device will not be reset. The results are difficult to predict, and the RC reset circuit is therefore neither reliable nor safe if brownouts are a possibility: see ‘Related patterns and alternative solutions’ for some alternative techniques.

Portability

Some portability issues, related to the different performance of various power supplies, have been considered elsewhere in this pattern. These will not be discussed further here.

Another key portability issue relates to the different reset circuits found on various 8051 devices. The reset circuits considered so far have been ‘active high’ in nature. This means that normally the RESET pin will be held at a low level (~0V): to effect a reset, the RESET pin needs to be pulled high (~Vcc), while the oscillator is running.

However, some 8051 devices have ‘active low’ inputs: these can be identified by the presence of a $\overline{\text{RESET}}$ pin. As the name suggests, these pins are held at a high level during normal operation, and must be pulled low (again usually for 24 clock cycles) to effect a reset. Examples of 8051 devices with active low inputs include the Infineon C509, C515C, and C517A. All of these are popular and widely-used devices.

Wiring an active low reset circuit is straightforward. Figure 7 shows a possible circuit for these various active low devices.

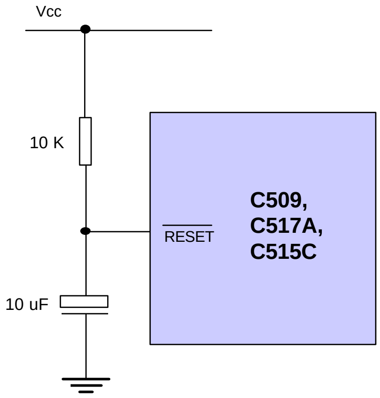

Figure 7: Creating an ‘active low’ RC reset circuit.

Note that, while the underlying principles are the same, the wiring of ‘active high’ and ‘active low’ resets are fundamentally incompatible.

Strengths and weaknesses

☺ **RC reset circuits are cheap to implement.**

☺ **RC resets are well-understood and widely used in other microprocessor and microcontroller systems.**

☺ **If your system is mains powered, and safety and reliability are not issues (and cost is), then this technique may be a good solution.**

If the system power supply characteristics are unknown or vary, or are subject to ‘brownout’, then the reset operation may not always be effective: RC resets are generally not suitable for main-powered applications which must be reliable or safe.

Related patterns and alternative solutions

The pattern 3-PIN RESET (see Pont, in press) describes a more expensive but generally much more reliable reset solution.

Example: Minimal Intel 8751 circuit with RC reset

A minimal Intel 8751 circuit, using an RC reset, is shown in Figure 8. See the pattern CRYSTAL OSCILLATOR [Pont, in press] for details of the oscillator circuit.

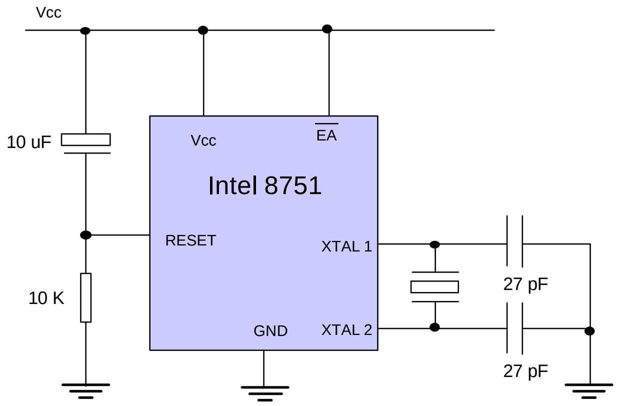

Figure 8: A minimal Intel 8751 circuit, with RC reset.

Example: Minimum Atmel 89C2051 circuit with RC reset

A minimal Atmel 89C2051 circuit, using an RC reset, is shown in Figure 9. Again, please see the pattern CRYSTAL OSCILLATOR [Pont, in press] for details of the oscillator circuit.

Note that the Atmel device does not support external memory so that - unlike Figure 8 - the EA pin is not present: see the ‘memory patterns’ [Pont, in press] for details.

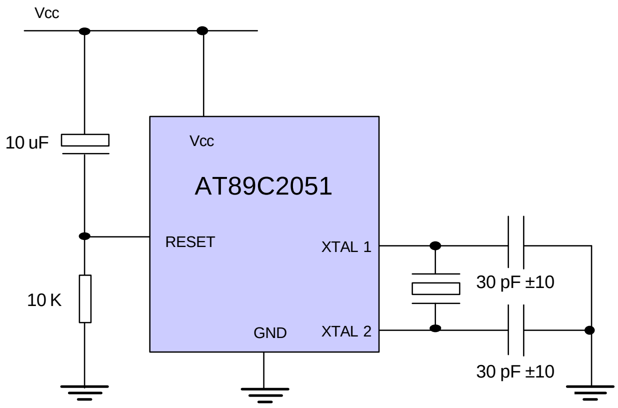

Figure 9: A minimal Atmel AT89C2051 circuit, with RC reset.

Example: Minimum Infineon C515C-8E circuit with RC reset

A minimal Infineon C515C-8E circuit is shown in Figure 10. Note that this device is has an 'active low' reset requirement.

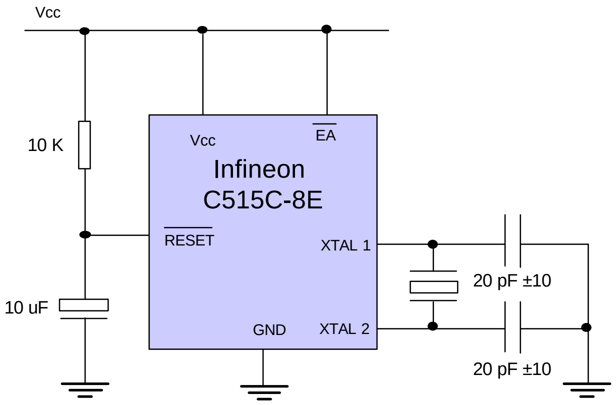

Figure 10: A minimal Infineon C515C-8E circuit, with RC reset.

Further reading

Pont, M.J. (in press) *"Patterns for reliable embedded systems: Rapid software development in C for the 8051 family of microcontrollers"*, Due for publication by Addison-Wesley Longman, June 2000.

SIMPLE TIMER DELAY

Context

You are developing software for an embedded application based on an 8051 microcontroller or similar device.

Problem

You need to wait for a fixed period of time before taking some action. For example, you want to delay 50 ms between switching on one piece of equipment, and activating a second.

Background

Suppose we want to flash some LEDs connected to Port 1 on an embedded system based on an 8051 microcontroller with a two-second cycle time (so that they are on for 1 second then off for 1 second, *ad infinitum*). The basic program structure we require could be implemented as follows:

```
...

while (1)
   {
   P1 = 0xFF;
   // Delay one second
   P1 = 0x00;
   // Delay one second
   }
```

In order to implement the delays, we could then try to create a function Delay() as follows:

```
Delay()
   {
   unsigned int x;
   unsigned int y;

   for (x=0; x<=65535; x++)
      {
      for (y=0; y<=65535; y++);
      }
   }
```

We could measure the pulse frequency we obtained, and then add or remove layers of `for` loops (and adjust the numbers) to give us the delay we required. This approach would work, and may be adequate if we simply want to flash a warning light. However, it has three main drawbacks:

1. It is difficult to produce precisely timed delays.
2. The loops are not portable, and must be re-tuned if a different processor or clock frequency is used.
3. Processor time is wasted in the loops since, while carrying out these delays, it is not generally possible to do anything else.

We can tackle all of these problems using one of the timers built in to the 8051 microcontroller. All members of the 8051 family have at least two 16-bit timer / counters, known as Timer 0 and Timer 1. Used as a timer (in 'Mode 1'), they are incremented every processor cycle[2]: that is, at the instruction cycle frequency. Assuming the timers are initially set at 0, then after 2^{16} cycles they will overflow: this overflow will cause a flag to be set (for example: TF0 for Timer 0; TF1 for Timer 1).

By varying the initial value stored in the timer, we specify the number of samples that occur before an overflow takes place, and can generate shorter delays. Longer delays can always be generated by using multiple executions of this 'wait for timer overflow' technique.

This type of mechanism forms the basis of SIMPLE TIMER DELAY routines in a wide range of embedded software systems.

[2] In the 8051 family, one processor cycle (one instruction cycle) corresponds to 12 clock cycles in many 'standard' devices (e.g. 80c51), while in recent 'extended' 8051 devices (such as the Infineon 80c505c and 80c515c) one processor cycle corresponds to 6 clock cycles. Clearly, this has an impact on the timing routines! You need to consult the data sheet for your microcontroller to ensure you use appropriate values.

Note that one consequence of the change is that the 80c515c (for example) provide the same level of basic performance at half the clock frequency. As the electromagnetic interference (EMI) generated by any digital electronic component is directly related to the clock frequency, this means that (everything else being equal) more modern versions of the 8051 family should generate less EMI than the original devices. This may be an important consideration in environments where generation of EMI is of particular concern.

Solution

Building on the material discussed under ‘Background’, SIMPLE TIMER DELAY calculations generally take the following form:

- We calculate the required starting value for the timer;
- We set a counter to this specified value;
- We wait for the counter to reach its maximum value and ‘roll over’;
- The rolling over of the timer signals the end of the delay by changing the value of a flag variable.

A detailed code example is presented below.

Reliability and safety issues

The techniques discussed in SIMPLE TIMER DELAY are not suitable for generating precisely-timed delays. They are inaccurate when used to generate short delays, and are very inaccurate when used to generate long delays.

Portability

Like the 8051 family, most microcontrollers have on-board timers: where such timers are available, this pattern may be adapted without great difficulty. If your chosen microcontroller does not have an on-board timer, the pattern LOOP DELAY [Pont, in press] offers an alternative solution.

Strengths and weaknesses

☺ **These basic time delay techniques have the great advantage that they are very simple and can be implemented in a few lines of code. As a result, they are widely applicable and are frequently used in applications where accurate timing is not of great concern.**

Because of the need to manually re-load the initial timer value, the delays obtained may not be precisely as expected. This is of particular concern where, for example, an attempt is made to delay for (say) a second by invoking a 50ms delay twenty times: this will **not** be accurate. Do not attempt to use SIMPLE TIMER DELAY to implement a real-time clock!

As implemented here, the processor is tied up waiting for the timer to overflow. Where processor power is limited, this may not be an acceptable solution.

Exclusive access to an important hardware resource (a timer) is required

The timings are not very portable: even different members of the 8051 family have different relationships between crystal frequency and instruction cycle frequency.

Related patterns and alternative solutions

SIMPLE TIMER DELAY is probably of most value if your application is ‘naked’: that is, you are not using a SCHEDULER [Pont, in press]: such (comparatively sophisticated) software environments always provide higher-level timing routines.

More generally, other patterns may provide a more elegant solution to a ‘time delay’ problem than is provided by SIMPLE TIMER DELAY. For example:

- DELAY FLAG [Pont, in press] will generate longer delays with greater accuracy. In addition, this pattern allows some ‘foreground’ processing to be carried out while waiting for a period of time to elapse. This will allow you, for example, to continually poll a switch input for a 10-second period.
- TIMER [Pont, in press] provides a means of accurately measuring elapsed time, and thus - for example - of implementing a real-time clock.
- GATED TIMER [Pont, in press] provides a means of accurately measuring the time between two external events, and thus - for example - of measuring a pulse width.
- SCHEDULER [Pont, in press] allow the use of a single hardware timer for multiple purposes.

Example: Flashing an LED

Flashing an LED can be useful as a means of drawing attention to a particular warning message or error condition. It can also be used as a means of saving power. There are, of course, numerous different ways of implementing such behaviour, some of which are entirely hardware based: here, we illustrate the use of the SIMPLE TIMER DELAY pattern.

Hardware

The single LED (or a similar device, such as a buzzer) is assumed to be connected to an 8051 microcontroller on Port 1 (P1.0), using positive logic: that is, +5v lights the LED. See DC OUTPUT (LOW POWER) [Pont, in press] for hardware details.

Software

To implement the flashing LED in software we will use an endless loop involving two ‘Simple Timer Delays’. The first of these will determine the ‘on’ period: the second will determine the LED ‘off’ period. The program is to run ‘for ever’.

We assume that we are using a ‘standard’ 8051 device. If we have a crystal frequency of 11.0592 MHz, then the corresponding instruction cycle frequency is 1/12th of the crystal frequency (see 8051 OSCILLATOR [Pont, in press] for further details and notable exceptions): in this case, 921.6 kHz. The instruction cycle period is therefore 1.085 µs, and the total time delay before the counter overflows is 2^{16} / 921600 seconds: that is, approximately 71 ms. Therefore, with this crystal, delays of up to about 70 ms can be directly obtained.

In this case, it will be convenient to have a 50ms delay. This means that the initial counter value we require is 2^{16} - 50/0.001085. This is **approximately** 19453 (decimal) or 4BFD (hexadecimal). We need to store this value in the low and high bytes of the timer used for the calculation: in this case, we will use Timer 0, which we can initialise as follows:

```
TH0 = 0x4B; // Set the high byte of Timer 0
TL0 = 0xFD; // Set the low byte of Timer 0
```

Program 1 shows a complete implementation of this example, in C.

```
/* ---------------------------------------------------------------- */
/* ---------------------------------------------------------------- */
// Example of SIMPLE TIMER DELAY pattern for 8051 microcontroller
//
// Flashes a single LED connected to P1.0 (+ve logic:
// - see DC OUTPUT (Low Power))
//
// This version does not use ISRs
//
// Compile for 8051 using Keil compiler

// Source file: Delay.C

#include <reg51.h>  // Special function register 8051

/* ................................................................ */
// Pattern: SIMPLE TIMER DELAY
void Simple_Delay_Init(void);
void Simple_Delay_Fifty_Milliseconds(void);

void Simple_Delay_N_Seconds(const int);

/* ................................................................ */
// Pattern: DC OUTPUT (LOW POWER)
#define LED_ON (1)
#define LED_OFF (0)

sbit Led_G = P1^0;

/* ---------------------------------------------------------------- */
void main(void)
   {
   Simple_Delay_Init();

   while(1) // Endless loop
     {
     Led_G = LED_ON;
     Simple_Delay_N_Seconds(1);
     Led_G = LED_OFF;
     Simple_Delay_N_Seconds(2);
     }
   }

/* ---------------------------------------------------------------- */
// Simple_Delay_N_Seconds()
//
// Provides delay of ***approximately*** N seconds
// via multiple calls to Simple_Delay_Fifty_Milliseconds()
/* ---------------------------------------------------------------- */
void Simple_Delay_N_Seconds(int N)
   {
   long i;

   for (i = 0; i < N * 20; i++)
      {
      Simple_Delay_Fifty_Milliseconds();
      }
   }
```

```
/* --------------------------------------------------------------- */
// Simple_Delay_Fifty_Milliseconds()
//
// Uses Timer 0 - must initialise TMOD correctly before using this fn
// - Simple_Delay_Init() serves this purpose
// Assumes 11.0592 MHz crystal on ‘standard’ 8051
/* --------------------------------------------------------------- */
void Simple_Delay_Fifty_Milliseconds(void)
   {
   TR0 = 0; // Stop the timer

   TH0 = 0x4B; // Set the high byte of Timer 0 (see text)
   TL0 = 0xFD; // Set the low byte of Timer 0 (see text)

   TF0 = 0; // Clear the ‘overflow’ flag
   TR0 = 1; // Start the timer

   while (TF0 == 0); // Wait for the timer to overflow

   TR0 = 0; // Stop the timer
   }

/* --------------------------------------------------------------- */
// Simple_Delay_Init(void)
//
// Prepare Timer0 for Simple Timer Delay
/* --------------------------------------------------------------- */
void Simple_Delay_Init(void)
   {
   TMOD |= 0x01;  // Place Timer 0 in Mode 1: i.e. simple 16-bit ctr
   ET0 = 0;       // Disable Timer 0 interrupt (interrupt not used)
   }
/* --------------------------------------------------------------- */
/* --------------------------------------------------------------- */
```

Program 1. A possible implementation of SIMPLE TIMER DELAY.

Further reading

Pont, M.J. (in press) *“Patterns for reliable embedded systems: Rapid software development in C for the 8051 family of microcontrollers”*, Due for publication by Addison-Wesley Longman, June 2000.

SOFTWARE SWITCH DEBOUNCER

Context

You are designing an embedded application based on an 8051 microcontroller or similar device.

Problem

How do you connect the port of an 8051 microcontroller to some form of mechanical switch (for example, a simple push-button switch, or an electromechanical relay) to allow, for example, user input, or to detect the limits of movement of a piece of equipment?

Background

Consider the simple push-button switch illustrated in Figure 11.

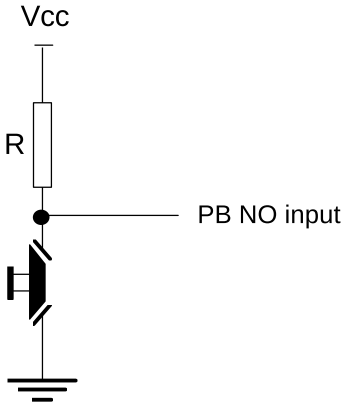

Figure 11: An example of a push-button ('normally open') switch input.

Depressing this switch will, in this arrangement, cause a voltage change from approximately 5v to 0v at the input port. In an ideal world, this change in voltage would take the form illustrated in Figure 12 (top). In practice, almost all mechanical switch contacts *bounce* (that is, turn on and off, repeatedly, for a short period of time) after the switch is closed or opened. As a result, the actual input waveform looks more like that shown in Figure 12 (bottom). Usually, switches bounce for less than 20 ms: large mechanical switches exhibit bounce behaviour for 50 ms or more.

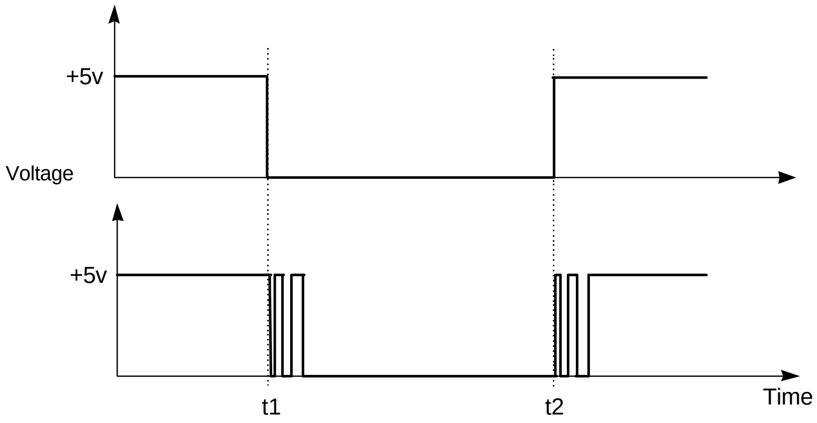

Figure 12: The voltage signal resulting from the switch shown in Figure 11. [Top] Idealised waveform resulting from a switch depressed at time t1 and released at time t2 [Bottom] Actual waveform showing leading edge bounce following switch depression and trailing edge bounce following switch release.

This bounce is equivalent to pressing an idealised switch multiple times. This causes various potential problems, not least:

- If we need to distinguish between single and multiple switch presses: for example, rather than reading 'A' from a keypad, we read 'AAAAA'
- If we wish to count the number of switch presses.
- If we need to distinguish between a switch being depressed and being released: for example, if the switch is depressed once, and then released some time later, the bounce will make it appear as if the switch has been pressed again.

Solution

Software-based switch debouncing is, in essence, straightforward:

1. We read the relevant port pin.
2. If we think we have detected a switch depression, we wait for about 20 ms, and then read the pin again.
3. If the second reading confirms the first reading, then we assume the switch really has been depressed.

Note that the figure of '20 ms' will, of course, depend on the switch used: the data sheet of the switch will provide this information. If you have no data sheet, you can either experiment with different figures, or measure directly using an oscilloscope.

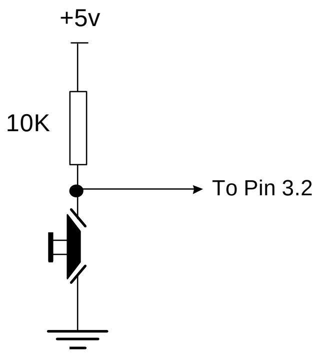

Figure 13: The hardware layout used for a simple switch input.

This basic procedure may be translated into code as follows:

- Check for input. In this case (Figure 13) input is a ‘low’ signal (port is pulled high). If no input found, return 0;

```
#define SWITCH_PORT P3
#define SWITCH_PRESSED 0  // Easy to change the logic here

sbit Switch = SWITCH_PORT^2;  // Assume switch is connected to input 3^2

...

if (!(Switch == SWITCH_PRESSED))
   {
   return 0;
   }
```

- If an input is found then wait for (say) 20ms to confirm;

```
#define SWITCH_DEBOUNCE_PERIOD 20

Delay(SWITCH_DEBOUNCE_PERIOD);  // User-defined delay routine
```

- Check for input again; return 0 if false alarm;

```
if (!(Switch == SWITCH_PRESSED))
   {
   return 0;
   }
```

- If second input found, then assume really have a keypress (return 1)

```
return 1;
```

Code Sample 1 illustrates a possible implementation of this technique.

```
// Basic switch input
char Switch_Get_Input1()
   {
   char Switch_status, Switch_status2, Countdown;

   // First check for switch input
   // We assume Switch_Scan() returns 1 if switch was depressed
   // - see Program 2 for possible implementation
   Switch_status = Switch_Scan();

   if (Switch_status == 0)
      {
```

```
      // No input found - return 0
      return 0;
      }

   // Possible input detected - wait for debounce
   // We assume Switch_Debounce() gives an appropriate delay
   // - see Program 2 for possible implementation
   Switch_Debounce();

   // Now see if switch is still depressed
   Switch_status2 = Switch_Scan();

   if (Switch_status2 != Switch_status)
      {
      // Pre-debounce and post-debounce switch presses are different
      // Assume no valid switch input found - return 0
      return 0;
      }

   // If we got this far, switch was pressed - return 1
   return 1;
   }
```

Code Sample 1. A possible implementation of SOFTWARE SWITCH DEBOUNCER.

Reliability and safety issues

Reading a switch input is more complicated than it initially appears, and there are several potential problems that can reduce the reliability and safety of any application with switch inputs.

In this section, we consider first some specific problems arising from sustained switch depressions. We then address some more general problems.

Dealing with sustained switch depressions

A possible problem with Code Sample 1 is that one sustained switch depression may be interpreted as multiple switch depressions. This will happen because, if the switch is pressed for a period longer than the debounce period, the function `Switch_Get_Input1()` will return. If this function is called again before the switch is released, then it may appear that the switch has been pressed again. This may not be what you require.

The simplest way of dealing with this situation is to wait until the key is released before returning from the function. Thus, before the final step above, add a line:

```
while (Switch == SWITCH_PRESSED);  // Wait for switch to be released
```

This approach is certainly not ‘real time’ (because we may spend a lot of time waiting for a key release); however, it only slows down the background processing, and foreground (that is, interrupt) processing can still continue.

Code Sample 2 illustrates this technique.

```
// Switch input - this version waits for key release
char Switch_Get_Input2()
   {
   char Switch_status, Switch_status2, Countdown;

   // First check for switch input
   // We assume Switch_Scan() returns 1 if switch was depressed
   // - see Program 2 for possible implementation
   Switch_status = Switch_Scan();

   if (Switch_status == 0)
      {
      // No input found - return 0
      return 0;
      }

   // Possible input detected - wait for debounce
   // We assume Switch_Debounce() gives an appropriate delay
   // - see Program 2 for possible implementation
   Switch_Debounce();

   // Now see if switch is still depressed
   Switch_status2 = Switch_Scan();

   if (Switch_status2 != Switch_status)
      {
      // Pre-debounce and post-debounce switch presses are different
      // Assume no valid switch input found - return 0
      return 0;
      }

   // Wait for switch to be released
   do
      {
      Switch_status2 = Switch_Scan();
      } while (Switch_status2 == Switch_status);

   // *May* wish to debounce again here
   Switch_Debounce();

   // If we got this far, switch was pressed - return 1
   return 1;
   }
```

Code Sample 2. A possible implementation of SOFTWARE SWITCH DEBOUNCER*. This version waits for key release.*

Code Sample 2 still has problems. If you choose to wait for a key release before returning, you run the risk that damage to the switch, or even a deliberately sustained key press by a user, will cause the system to 'hang'. As a result, some form of 'time out' facility will often be required.

Code Sample 3 illustrates one way of implementing the required behaviour:

```
// Switch input - this version interprets sustained depressions as faults
char Switch_Get_Input()
   {
   char Switch_status, Switch_status2, Countdown;

   // First check for switch input
   Switch_status = Switch_Scan();

   if (Switch_status == 0)
      {
```

```
      // No input found - return
      return 0;
      }

   // Possible input detected - wait for debounce
   Switch_Debounce();

   // Now see if switch is still depressed
   Switch_status2 = Switch_Scan();

   if (Switch_status2 != Switch_status)
      {
      // Pre-debounce and post-debounce switch presses are different
      // Assume no valid switch input found - return
      return 0;
      }

   // Got valid input - now wait until switch is released
   // Also provided a 'time out' here (basis of auto repeat)
   // Omit this if not required.

   // Set timeout of 1 seconds using Simple Timer Delay (NOT ACCURATE!)
   Countdown = 20;
   do
      {
      Switch_status2 = Switch_Scan();
      Simple_Delay_Fifty_Milliseconds();
      } while ((Switch_status2 == Switch_status) && (--Countdown > 0));

   if (Switch_status2 == Switch_status)
      {
      // Switch is still depressed
      // We assume this indicates a fault
      return SWITCH_FAULT;
      }

   // The switch was pressed and released
   // We will debounce again
   Switch_Debounce();

   // Now (finally) return switch value
   return 1;
   }
```

Code Sample 3. A possible implementation of SOFTWARE SWITCH DEBOUNCER. *This version interprets sustained depressions as faults.*

Finally, it must be emphasised that sustained switch depressions do not always indicate a fault: in fact, sustained switch depressions can be used to turn this two-state input device into a three-state (or more) input device.

For example, if we have a real-time clock, we may have just two buttons (‘forward’ and ‘backward’) to set the time. To avoid this process becoming unduly tedious, we might decide that a brief depression of the ‘Forward’ button should slowly increment the displayed time, while a sustained depression (longer than five seconds), should advance the display more rapidly.

To implement this type of behaviour, we can use a code architecture very similar to that described above: this time, however, we assume that if the switch remains depressed, we will assume the user is indicating that he or she wishes to carry out some different activity.

Code Sample 4 illustrates one way of implementing this ‘three state’ behaviour:

```
// Switch input - this version implements a trinary input routine
char Switch_Get_Input()
   {
   char Switch_status, Switch_status2, Countdown;

   // First check for switch input
   Switch_status = Switch_Scan();

   if (Switch_status == 0)
      {
      // No input found - return
      return 0;
      }

   // Possible input detected - wait for debounce
   Switch_Debounce();

   // Now see if switch is still depressed
   Switch_status2 = Switch_Scan();

   if (Switch_status2 != Switch_status)
      {
      // Pre-debounce and post-debounce switch presses are different
      // Assume no valid switch input found - return
      return 0;
      }

   // Got valid input - now wait until switch is released
   // Also provided a 'time out' here (basis of auto repeat)
   // Omit this if not required.

   // Set timeout of 1 seconds using Simple Timer Delay (NOT ACCURATE!)
   Countdown = 20;
   do
      {
      Switch_status2 = Switch_Scan();
      Simple_Delay_Fifty_Milliseconds();
      } while ((Switch_status2 == Switch_status) && (--Countdown > 0));

   if (Switch_status2 == Switch_status)
      {
      // Switch is still depressed
      // We assume this is because user wants to carry out
      // some ‘sustained depression’ activity
      return SWITCH_SUSTAINED;
      }

   // The switch was pressed (briefly) and released
   // We will debounce again
   Switch_Debounce();

   // User pressed switch briefly
   return SWITCH_BRIEF;
   }
```

Code Sample 4. A possible implementation of SOFTWARE SWITCH DEBOUNCER*. This version implements a ‘trinary’ input routine.*

General problems with switch inputs

A general rule of thumb in a microcontroller-based system (and one which we return to in many patterns) is that the software should be ‘in control’ of the hardware, rather than vice versa. A switch input is a simple example of this. Consider, for example, the use of a latching (or 'toggle') switch: this type of switch leaves the hardware in control because (in almost all circumstances) the software cannot alter the switch position once it has been set. This means, among other things, that the user of the device receives visual feedback from the switch position saying that (for example) the device is on, and the software cannot control this feedback even in the event of an error, etc. If, however, a push button (PB) switch is used (for example) to switch on a device, then the software is in control and may be able to save power by deactivating some of the system during idle periods: in these circumstances, an LED or audible warning device can be used to inform the user of the change of state. The general rule is: do not use latching switches as input devices.

However, simply using a PB switch is not enough to ensure reliability. For example, consider again the switch in Figure 11: this type of switch is often referred to as 'normally open' (NO). If this form of NO switch is removed or damaged, leaving the connection permanently open, we cannot detect this fact in software. This may have safety or reliability implications.

In some circumstances, there may be advantages to using a 'normally closed' (NC) PB switch (Figure 14). Using an PB-NC switch, we can detect the removal of the switch or damage to the wiring, although we cannot generally distinguish this from a normal (sustained) switch depression.

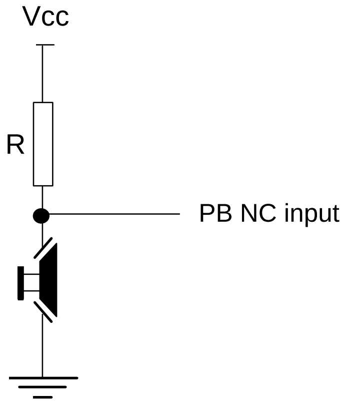

Figure 14: An example of a push-button ('normally closed') switch input.

All of the above techniques have considered 'single pole' switches. The most reliable solution is often to use a PB 'double-pole, double-throw' (PB-DPDT) switch (Figure 15). This generates two inputs, which will always have opposite logic if the switch is undamaged and wired correctly. Using such an input device, we can detect various types of switch faults (including switch removal) and wiring faults (including wire cutting). Note that such a switch requires two input pins and more software than a single-pole switch input.

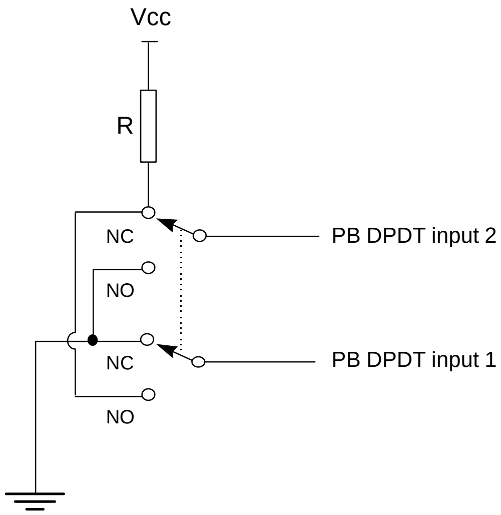

Figure 15: An example of a push-button DPDT switch input. With the arrangement shown, Input 1 will have a voltage of 0V and Input 2 Vcc with the switch open: these values will be reversed when the switch is closed.

Even using PB-DPDT inputs is not enough to guarantee safety. Another general rule is that a safety-related input should not rely on a single hardware or software component. Thus, for example, a switch limiting the movement of a robotic device should have an additional, and independent, backup circuit available.

Portability

This pattern may be adapted for use with other microcontrollers without difficulty.

Strengths and weaknesses

☺ **SOFTWARE SWITCH DEBOUNCER requires a minimum of external hardware.**

☺ **It does not require an operating system or scheduler.**

☺ **It does not use an interrupt input and is therefore, in some circumstances, safer than equivalent interrupt-based solutions.**

☺ **It is very flexible: for example, the programmer can incorporate 'auto repeat' functions with few code changes.**

☺ **Overall: it is simple and cheap to implement.**

A complete implementation may require a substantial amount of code to ensure reliable operation.

Like all software-only input techniques, SOFTWARE SWITCH DEBOUNCER provides no protection against out-of-range inputs: if someone applies +/-20V to your switch (by mistake, or deliberately), they may 'fry' the microcontroller. There is little you can do, in software, to guard again this. For the same reasons, SOFTWARE SWITCH DEBOUNCER provides limited immunity to electrostatic discharge (ESD). ESD can present a significant problem in harsh environments (e.g. industrial systems, automotive applications), and compliance with international standards in this area (e.g. IEC 1000-4-2) is a requirement for some applications. See HARDWARE SWITCH DEBOUNCER [Pont, in press] for a discussion of this issue, and a solution. See also ‘Reliability and safety issues’ for further comments on the safe use of switch inputs.

Related patterns and alternative solutions

Reading a switch input is a classic example of a situation where we can trade hardware cost against software complexity. In this case, the superficially trivial process of responding to a mechanical switch input can involve what at first sight seems a rather complex software framework with a large number of parameters to deal with issues such as switch debounce (how long does the switch bounce?) and sustained switch depressions (do we require an ‘auto repeat’ facility?).

For a simpler software-based switch interface, consider REGULAR SWITCH CHECK [Pont, in press].

For an alternative approach using hardware, consider HARDWARE SWITCH DEBOUNCER [Pont, in press].

Where the switch (or other input device) does not suffer from bounce (for example, where your input is derived from a solid-state relay or some other form of ‘electronic switch’) then much of the complexity of these input strategies can be avoided through the use of PORT I/O [Pont, in press].

Example: Counting key presses

This example illustrates the use of SOFTWARE SWITCH DEBOUNCER to count the number of times a switch is depressed. The count is displayed on a bank of LEDs.

This example assumes that the hardware consists of a single-pole NC or NO switch.

```
/* -------------------------------------------------------------- */
/* -------------------------------------------------------------- */

// Example of Software Switch Debouncer pattern
// for 8051 microcontroller
//
// Also illustrates: SIMPLE TIMER DELAY; DC OUTPUT (Low Power)
//
// Counts number of depressions of a SPST pushbutton switch (P1^0)
// Displays result on bank of LEDs (P3)
//
// Compile for 8051 using Keil compiler

// Source file : SwitchP.C

#include <reg51.h>  // Special function register 8051

/* .............................................................. */
// Pattern: Software Switch Debouncer
#define SWITCH_PRESSED (0) // defines switch pressed value

char Switch_Get_Input();   // Main switch input function
char Switch_Scan();        // Scans the switch
```

```
void Switch_Debounce();    // Debounce (delay) function

sbit Switch_pin_G = P1^0;  // The switch connection

/* .............................................................. */
// Pattern: SIMPLE TIMER DELAY
void Simple_Delay_Init(void);
void Simple_Delay_Fifty_Milliseconds(void);

/* .............................................................. */
// Pattern: LED OUTPUT
#define LED_PORT (P3)
void LED_Port_Display(unsigned char);

/* -------------------------------------------------------------- */
void main(void)
   {
   int Switch_presses = 0;

   Simple_Delay_Init();

   while(1) // Run forever
      {
      Switch_presses += Switch_Get_Input();

      LED_Port_Display((unsigned char) Switch_presses);
      }
   }

/* -------------------------------------------------------------- */
// Switch_Get_Input()
//
// The main Software Switch Debouncer function
// Note - includes 'auto repeat'
// Note - duration varies - not real time
/* -------------------------------------------------------------- */
char Switch_Get_Input()
   {
   char Switch_status, Switch_status2, Countdown;

   // First check for switch input
   Switch_status = Switch_Scan();

   if (Switch_status == 0)
      {
      // No input found - return
      return 0;
      }

   // Possible input detected - wait for debounce
   Switch_Debounce();

   // Now see if switch is still depressed
   Switch_status2 = Switch_Scan();

   if (Switch_status2 != Switch_status)
      {
      // Pre-debounce and post-debounce switch presses are different
      // Assume no valid switch input found - return
      return 0;
      }

   // Got valid input - now wait until switch is released
   // Also provided a 'time out' here (basis of auto repeat)
   // Omit this if not required.

   // Set timeout of 1 seconds using Simple Timer Delay (NOT ACCURATE!)
   Countdown = 20;
   do
      {
      Switch_status2 = Switch_Scan();
      Simple_Delay_Fifty_Milliseconds();
      } while ((Switch_status2 == Switch_status) && (--Countdown > 0));

   if (Switch_status2 == Switch_status)  // Switch still depressed
      {
      // Key is still depressed - start auto-repeat function
      // Return a different value here if you need to know that
      // you are in auto-repeat mode (e.g. -1).
      return 1;
      }

   // The switch was pressed and released
   // Need to debounce again
   Switch_Debounce();

   // Now (finally) return switch value
   return 1;
   }

/* --------------------------------------------------------------- */
// Switch_Debounce()
//
// Here, debounce using Simple Timer Delay - numerous other possibilities
/* --------------------------------------------------------------- */
void Switch_Debounce()
   {
   Simple_Delay_Fifty_Milliseconds();
   }

/* --------------------------------------------------------------- */
// Switch_Scan()
//
// Checks to see if switch is currently depressed.
// Note: if no possibility of ‘bounce’ this is all you need to do.
/* --------------------------------------------------------------- */
char Switch_Scan()
  {
  if (Switch_pin_G == SWITCH_PRESSED)
     {
     return 1;
     }
  else
     {
     return 0;
     }
  }

/* --------------------------------------------------------------- */
// Simple_Delay_Fifty_Milliseconds()
//
// Uses Timer 0 - must initialise TMOD correctly before using this fn
// - Simple_Delay_Init() serves this purpose
// Assumes 11.0592 MHz crystal on ‘standard’ 8051
/* --------------------------------------------------------------- */
void Simple_Delay_Fifty_Milliseconds(void)
   {
   TR0 = 0; // Stop the timer

   TH0 = 0x4c; // Set up the  initial values (11.0592 MHz xtal)
   TL0 = 0x00; // - this sets the 50ms delay (adjust as required)

   TF0 = 0; // Clear the ‘overflow’ flag
   TR0 = 1; // Start the timer

   while (TF0 == 0); // Wait for the timer to overflow
   TR0 = 0; // Stop the timer
   }

/* --------------------------------------------------------------- */
// Simple_Delay_Init(void)
//
// Prepare Timer0 for Simple Timer Delay
/* --------------------------------------------------------------- */
void Simple_Delay_Init(void)
   {
   TMOD |= 0x01;  // Place Timer 0 in Mode 1: i.e. simple 16-bit ctr
   ET0 = 0;       // Disable Timer 0 interrupt (interrupt not used)
   }

/* --------------------------------------------------------------- */
// LED_Port_Display()

// Simple function to display char on LEDs connected to port.
/* --------------------------------------------------------------- */
void LED_Port_Display(unsigned char Switch)
   {
   LED_PORT = Switch;
   }
/* --------------------------------------------------------------- */
/* --------------------------------------------------------------- */
```

Program 2. A possible implementation of SOFTWARE SWITCH DEBOUNCER.

Further reading

Pont, M.J. (in press) *"Patterns for reliable embedded systems: Rapid software development in C for the 8051 family of microcontrollers"*, Due for publication by Addison-Wesley Longman, June 2000.

FUNCTION TOKEN

Context

You are developing a cost-effective embedded application for use in an environment which may be subject to high levels of high levels of electromagnetic interference (EMI).

Problem

How can you reduce the impact of EMI on your application using software-based techniques?

Background

Electromagnetic interference (EMI) can be a major source of problems in many embedded systems. While systems such as military aircraft may have to be designed to withstand extreme levels of EMI (typically direct lightning strikes), more mundane applications for automotive environments, and even domestic washing machines, are also examples of environments where EMI is common. If not handled correctly, then EMI can lead to erratic, unreliable and even dangerous systems.

In general, EMI can have several different impacts on microcontroller-based systems, including the following:

- It can disrupt the program flow (the ‘program counter’): the effect is similar to inserting one or more ‘goto X’ instructions in the executable code, where X is a ‘random’ address, anywhere in system memory;
- It can change the value of data (or instructions) in memory;
- It can corrupt the values from sensors;
- It can corrupt instructions fed to actuators or other output devices.

The techniques discussed in FUNCTION TOKEN are primarily aimed at tackling corruption to the program counter: other patterns (such as DUPLICATE DATA, CRC DATA, SINGLE-SENSOR INPUT and MULTI-SENSOR INPUT [see Pont, in press]) tackle the other aspects of EMI behaviour.

Solution

The basic idea implemented in FUNCTION TOKEN is very simple:

- Before we call a function, we set the value of one or more (global) variables to a known value.
- During the function execution, we check that the global variables have the required values: if they do, we are reassured: if they do not, we assume that (through EMI or some other fault) we have been ‘thrown’ into the function, and we take ‘appropriate action’.
- Depending on the system, and the function call, this appropriate action may involve (for example) re-starting the system, or returning the system to an appropriate recovery state. Note that, in most cases, the only effective course of action in the event of EMI corruption is to restart the main program loop.

We give an example of the implementation of FUNCTION TOKEN below.

Reliability and safety issues

In an ideal world, it is better to limit the impact of EMI through shielding or other hardware-based design techniques (see Pont, in press). However, software-based measures against EMI -such as those implemented in FUNCTION TOKEN - are also effective, and should be considered where:

- The cost of the application is a primary design consideration, and failure of the system will cause only inconvenience, and will not result in any risk of injury. A possible example of such an application is a domestic washing machine.
- Failure of the system may result in injury (or worse), and hardware shielding, etc, has already been included in the design: software-based measures may then be included as a ‘second layer of defence’, primarily to allow the system to continue to operate in the event of failure of, or damage to, the hardware mechanisms. A possible example of such an application is an automotive system.

Portability

This pattern may be adapted for use with other microcontrollers without difficulty.

Strengths and weaknesses

☺ **The techniques presented here as ‘FUNCTION TOKEN’ have been shown to be effective in tackling one of the main consequences of EMI: disruption in the flow of control (IEE, 1998; Pont *et al.,* 1999).**

FUNCTION TOKEN is not without a cost, and can add 10 - 20% to the code size (Coulson, 1998).

Related patterns and alternative solutions

FUNCTION TOKEN is not a complete solution to the problems caused by EMI. The techniques discussed in FUNCTION TOKEN are primarily aimed at tackling corruption to the program counter (see ‘Background’): other (software-based) patterns (such as DUPLICATE DATA, CRC

DATA, SINGLE-SENSOR INPUT and MULTI-SENSOR INPUT [see Pont, in press]) tackle the other aspects of EMI behaviour. In addition, the more general pattern WATCHDOG [Pont, in press] can also provide another layer of protection.

Example: Using Function Token

In this example, we illustrate - using a simple simulation of EMI - the effectiveness of this token-passing approach.

We begin by illustrating (in Program 3) the effects of EMI. What the program does is to simulate the impact of EMI on the processor's program counter (PC): in effect, as we go around the main program loop, the program may be (at 'random') thrown back into one of the functions.

```
#include <stdlib.h>
#include <time.h>

void Function1();
void Function2();
void Function3();

void main(void)
   {
   int x,i;

   // Different results each run...
   srand((unsigned) time(NULL));

   for (i=0; i<=20; i++)
      {
      printf("Test %2d - ",i);

      one:
      Function1();

      two:
      Function2();

      three:
      Function3();

      // Simulate EMI
      x = rand() % 10;
      switch(x)
         {
         case 1: goto one;
         case 2: goto two;
         case 3: goto three;
         }
      }
   }

void Function1()
   {
   printf("1");
   }

void Function2()
   {
   printf("2");
   }

void Function3()
```

```
{
printf("3\n");
}
```

Program 3: A simulation of the effects of EMI in an embedded application.

Here is an example of the output of Program 3:

```
Test  0 - 123
Test  1 - 123
Test  2 - 123
Test  3 - 123
Test  4 - 123
23
Test  5 - 123
Test  6 - 123
Test  7 - 123
Test  8 - 123
23
Test  9 - 123
Test 10 - 123
Test 11 - 123
Test 12 - 123
Test 13 - 123
Test 14 - 123
Test 15 - 123
3
Test 16 - 123
Test 17 - 123
Test 18 - 123
Test 19 - 123
123
Test 20 - 123
23
```

The outputs generated in addition to the normal ‘Test’ lines shown the effect of this simulated EMI interference.

Program 4 illustrates a simple application of FUNCTION TOKEN to address this problem.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int Function1();
int Function2();
int Function3();

int Token = 1;

void main(void)
   {
   int x,i;

   // Different results each run...
   srand((unsigned) time(NULL));

   for (i=0; i<=20; i++)
      {
      printf("Test %2d - ",i);

      one:
```

```
      if (Function1())
         {
         // Take appropriate action...
         }

      two:
      if (Function2())
         {
         // Take appropriate action...
         }

      three:
      if (Function3())
         {
         // Take appropriate action...
         }

      // Simulate EMI
      x = rand() % 100;
      switch(x)
         {
         case 1: goto one;
         case 2: goto two;
         case 3: goto three;
         }
      }
   }

int Function1()
   {
   if (Token == 1)
      {
      printf("1");
      Token = 2;
      return 0;
      }

   // Incorrect token
   // Take appropriate action here ...

   return 1; // ...also inform calling function
   }

int Function2()
   {
   if (Token == 2)
      {
      printf("2");
      Token = 3;
      return 0;
      }

   // Incorrect token
   // Take appropriate action here...

   return 1; // ...also inform calling function
   }
```

```
int Function3()
   {
   if (Token == 3)
      {
      printf("3\n");
      Token = 1;
      return 0;
      }

   // Incorrect token
   // Take appropriate action here...

   return 1; // ...also inform calling function
   }
```

Program 4: Applying FUNCTION TOKEN *to the simulated EMI problem. See text for details.*

Here is an example of the output of Program 4:

```
Test  0 - 123
Test  1 - 123
Test  2 - 123
Test  3 - 123
Test  4 - 123
Test  5 - 123
Test  6 - 123
Test  7 - 123
Test  8 - 123
Test  9 - 123
Test 10 - 123
Test 11 - 123
Test 12 - 123
Test 13 - 123
Test 14 - 123
Test 15 - 123
Test 16 - 123
Test 17 - 123
Test 18 - 123
Test 19 - 123
Test 20 - 123
```

In this case, through the simple use of FUNCTION TOKEN, the corruption of the program has been removed (in this case completely).

Further reading

Coulson, D.R. (1998) "EMC techniques for microprocessor software", in: Proceedings of IEE Colloquium on Electromagnetic Compatibility of Software", November 1998, Savoy Place, London, UK.

IEE (1998) "Proceedings of IEE Colloquium on Electromagnetic Compatibility of Software", November 1998, Savoy Place, London, UK.

IEE (1999) "Proceedings of IEE Colloquium on Electromagnetic Compatibility for Automotive Electronics", September 1999, Birmingham, UK.

Pont, M.J. (in press) *"Patterns for reliable embedded systems: Rapid software development in C for the 8051 family of microcontrollers"*, Due for publication by Addison-Wesley Longman, June 2000.

Pont, M.J., Kureemun, R., Ong, H.L.R., and Peasgood, W. (1999) *"Increasing the reliability of embedded automotive applications in the presence of EMI: A pilot study"*, in: Proceedings of IEE Colloquium on Electromagnetic Compatibility for Automotive Electronics", September 1999, Midland Engineering Centre, Birmingham, UK.

Storey, N. (1996) *"Safety-critical computer systems"*, Addison-Wesley, UK.