
A SELECTION OF PATTERNS FOR RELIABLE EMBEDDED SYSTEMS

**Michael J. Pont¹, Royan H.L. Ong, Chinmay R. Parikh, Ridwan Kureemun,
Chen Pang Wong, William Peasgood and Yuhua Li**

Control & Instrumentation Research Section,
Department of Engineering, University of Leicester, Leicester, ENGLAND LE1 7RH, UK.

Introduction

We present here a small selection of patterns intended to support the development of embedded systems: specifically, the patterns are written for developers of software (and some associated hardware) for the ubiquitous 8051 family of microcontrollers.

These introductory patterns 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.

Please note that these patterns are taken from a larger collection:

Pont, M.J. (in preparation) “*Patterns for reliable embedded systems*”, due for publication January 2000 by Addison-Wesley.

The patterns as a whole cover techniques for interfacing (8051) microcontrollers to various I/O devices (including switches, keypads, analogue inputs), the use of schedulers and real-time operating systems, and various forms of serial communications (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 Keil C51 compiler: however, the patterns (and code) are readily adaptable to other software and hardware environments.

¹ M.Pont@le.ac.uk - <http://www.leicester.ac.uk/engineering/mjp9/>

LED OUTPUT

Context

You are developing an embedded application for an 8051 microcontroller (see Chapter 3) or similar device.

Problem

You need to drive a single light-emitting diode (LED) from the microcontroller port.

Background

To make use of LEDs, it is useful to have a basic understanding of these devices. As the use of LEDs frequently involves the use of resistors, it can also be helpful to know something about the range of available resistor values. We consider both of these topics here.

Basic features of LEDs

Given the name it is hardly surprising that, electrically, an individual LED operates as a diode. LEDs have a forward voltage drop of about 2 volts, and typically require a current of around 5 - 15 mA for a bright display (Figure 1). Note that the forward voltage required to 'switch on' a conventional (silicon) diode is around 0.7 V: the difference arises because LEDs are generally manufactured from gallium arsenide phosphide (see Horowitz and Hill, 1989, for further details).

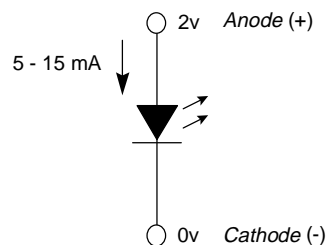


Figure 1: Lighting a single LED.

Using resistors

Many resistors are available only in a limited range of values (Figure 2).

	10	100	1.0K	10K	100K	1.0M
	11	110	1.1K	11K	110K	1.1M
	12	120	1.2K	12K	120K	1.2M
	13	130	1.3K	13K	130K	1.3M
	15	150	1.5K	15K	150K	1.5M
	16	160	1.6K	16K	160K	1.6M
	18	180	1.8K	18K	180K	1.8M
	20	200	2.0K	20K	200K	2.0M
2.2	22	220	2.2K	22K	220K	2.2M
2.4	24	240	2.4K	24K	240K	2.4M
2.7	27	270	2.7K	27K	270K	2.7M
3.0	30	300	3.0K	30K	300K	3.0M
3.3	33	330	3.3K	33K	330K	3.3M
3.6	36	360	3.6K	36K	360K	3.6M
3.9	39	390	3.9K	39K	390K	3.9M
4.3	43	430	4.3K	43K	430K	4.3M
4.7	47	470	4.7K	47K	470K	4.7M
5.1	51	510	5.1K	51K	510K	5.1M
5.6	56	560	5.6K	56K	560K	5.6M
6.2	62	620	6.2K	62K	620K	6.2M
6.8	68	680	6.8K	68K	680K	6.8M
7.5	75	750	7.5K	75K	750K	7.5M
8.2	82	820	8.2K	82K	820K	8.2M
9.1	91	910	9.1K	91K	910K	9.1M
						10M

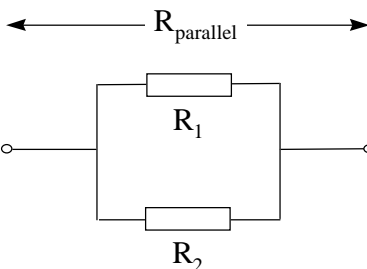
Figure 2: Many resistors are only generally available in these 161 ‘standard’ values (the ‘E24 series’), or in a subset of this range. Note that many cheaper resistors have a tolerance of 5%: that is, values may vary by this amount. More accurate resistors (1% tolerance, or less) are also available, at higher cost. For the type of application considered here, 5% resistors will usually be ideal.

Where this limited range does not provide a component suitable for your task, the available resistors may be assembled in series or parallel (or some combination of the two) to give more appropriate values.

The resulting resistance two resistors (R_1 and R_2) in series (R_{series}) is given by:

$$R_{series} = R_1 + R_2$$


The resulting resistance two resistors in parallel is calculated as follows:

$$R_{parallel} = \frac{R_1 R_2}{R_1 + R_2}$$


Note that, by placing two resistors in parallel, we always obtain a smaller resistance: if, for example, we place two 91Ω resistors in parallel we obtain a combined resistance of 45.5Ω .

It is also worth noting that ‘packages’ of identical resistors (typically eight) are also available, in ‘dual in line’ (DIL) casings. In some applications (such as multi-segment LEDs) these can be useful.

Reliability and safety issues

LEDs (particularly flashing LEDs) are frequently used as warning devices. Bear in mind that in bright sunlight, such warnings will be barely visible, and that blind or partially sighted people will never be able to see them. Adding an additional or alternative audible output may be appropriate in some systems.

As we discuss below (‘Strengths and weaknesses’), LEDs consume large amounts of power (compared, for example, to liquid crystal displays), and need to be used with care in many battery-powered designs.

Solution

Both hardware and software are required to implement this pattern.

Hardware

The ports on typical microcontroller hardware, such as most members of the 8051 family, can be set at values of 0V and 5V² under software control, and can source (or sink) about 20 mA of current. As a result it is usually possible to drive single LEDs directly from the port pins (Figure 3).

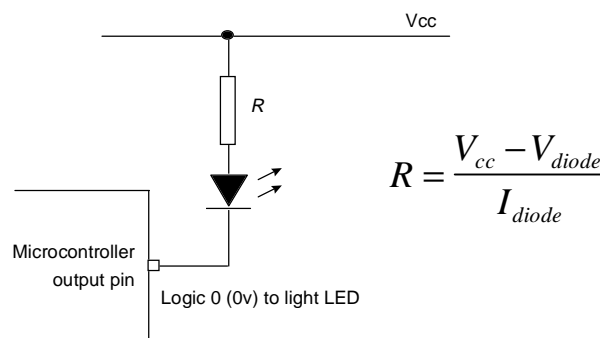


Figure 3: Connecting a single LED directly to a microcomputer port (as illustrated here) is usually possible, but not always recommended: see text for details. Note: when calculating the required resistance, the resistance values are in Ohms, voltages in Volts and current in Amps.

In this case, typical values are:

- Supply voltage, $V_{cc} = 5\text{v}$,
- LED forward voltage, $V_{diode} = 2\text{v}$,
- Required diode current, $I_{diode} = 15\text{ mA}$ (note that the data sheet for your chosen LED will provide this information).

This gives a required resistor value of 200Ω: this value is available in most ranges (see ‘Background’).

² Note that 3V versions of the 8051 family are now available: the techniques here can be applied to such controllers without difficulty, as we demonstrate below (‘Example: Driving an LED from a 3V microcontroller’)

Why 'sink' the current?

Most of the circuits we present for controlling LEDs or for switching higher-powered DC and AC loads involve some form of current 'sink', as in Figure 3. Here, the current flows 'in to' the processor port. This is despite the fact that most microcontroller ports, manufactured using some form of MOS technology, can also 'source' current, so that the LED could be arranged as illustrated in Figure 4.

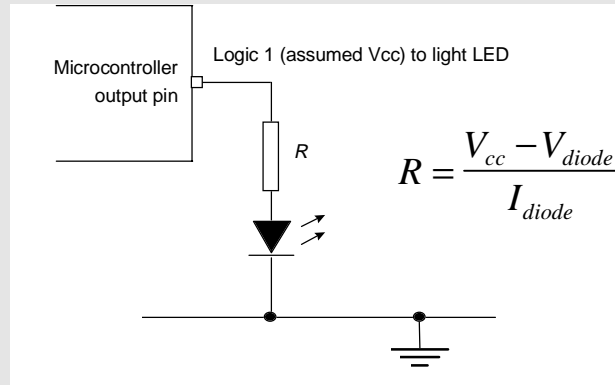


Figure 4: Connecting a single LED directly to a processor port. Here the port acts as a current source. See text for details, and for a discussion of the drawbacks of this approach.

However, some of the other circuits used as buffers or drivers (see, for example, SWITCHED DC OUTPUT [page >>>]) may use different manufacturing processes, including TTL technology. TTL devices can sink current in much the same way as MOS devices, but are poor current sources. As a result, hardware designs based on current sinking are generally more portable (across different logic families) than designs based on current sourcing and, for this reason, will be used throughout this pattern collection.

Buffering the outputs

Even where the ports can drive LEDs directly, we can both reduce the risk of damage to the ports, and ensure we have sufficient current available to light the LED brightly, by using a standard inverter, or other driver hardware, as a 'buffer' between the microcomputer and the LED.

Suitable inverters are readily, and very cheaply available: six are packaged together, for example, in the ubiquitous 74LS04 IC (or equivalent). Figure 5 illustrates one way in which such an inverter may be used.

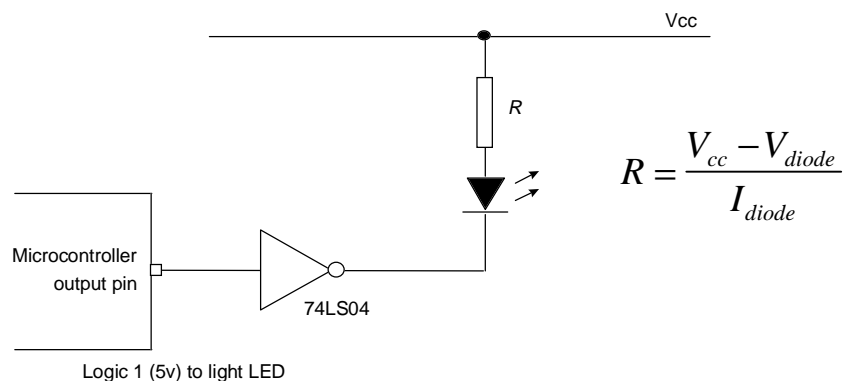
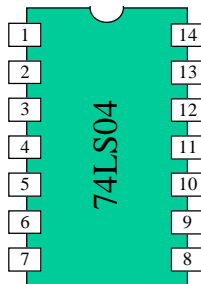


Figure 5: Driving an LED via a 74LS04 inverter (see Figure 6).

- 01 - Input 1
- 02 - Output 1
- 03 - Input 2
- 04 - Output 2
- 05 - Input 3
- 06 - Output 3
- 07 - Gnd

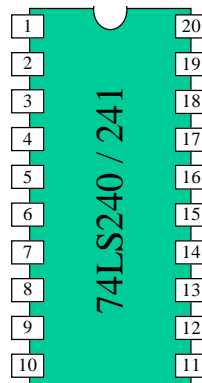


- 14 - Vcc
- 13 - Output 6
- 12 - Input 6
- 11 - Output 5
- 10 - Input 5
- 09 - Output 4
- 08 - Input 4

Figure 6: Pinout of the 74ls04 inverter. Note that each chip contains 6 independent inverters.

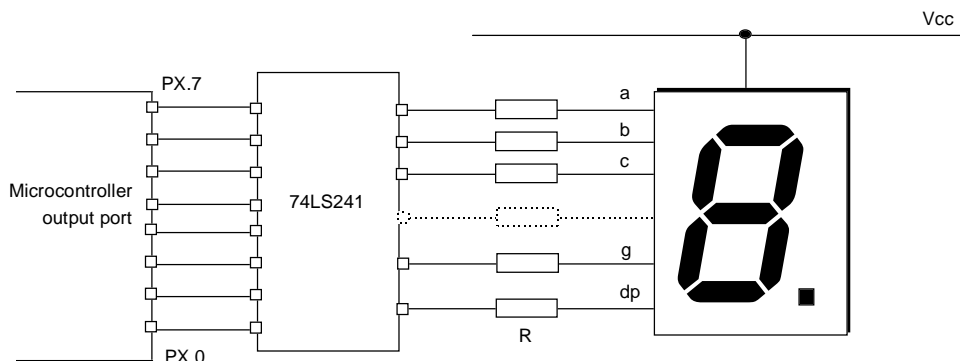
Another alternative is the simple inverting (74LS240) and non-inverting (74LS241) buffers that come in packages of eight (Figure 7). These are particularly useful when driving small numbers of multi-segment LED displays (Figure 8).

- 01 - Gate 1 (inverted)
- 02 - Input 1A1
- 03 - Output 2Y4
- 04 - Input 1A2
- 05 - Output 2Y3
- 06 - Input 1A3
- 07 - Output 2Y2
- 08 - Input 1A4
- 09 - Output 2Y1
- 10 - GND



- 20 - Vcc
- 19 - Gate 2 (non-inverted)
- 18 - Output 1Y1
- 17 - Input 2A4
- 16 - Output 1Y2
- 15 - Input 2A3
- 14 - Output 1Y3
- 13 - Input 2A2
- 12 - Output 1Y4
- 11 - Input 2A1

Figure 7: Pinout of the 74ls240 and 241 buffers. Note that each chip contains 8 buffers, arranged in two groups of 4 (Group 1, Group 2). The buffers are all tri-state devices, and - for use as a simple buffer - the gates must be enabled (in the case of Group 1) by applying a low (~0V) input to Gate 1, or (in the case of Group 2), by applying a high (~5V) input to Gate 2.



$$R = \frac{V_{cc} - V_{diode}}{I_{diode}}$$

Note: All resistor values are the same

Figure 8: Using a 74LS241 to drive a multi-segment LED display.

The simple 74LS240/241 and 74LS04 can sink a current of around 20 mA. This may not always be enough: for example, some blue LEDs require currents of up to 50mA to operate correctly. Where you need a higher current, a form of simple transistor driver is a cost-effective solution when driving a single or small number of LEDs: Figure 9 summarises one possibility.

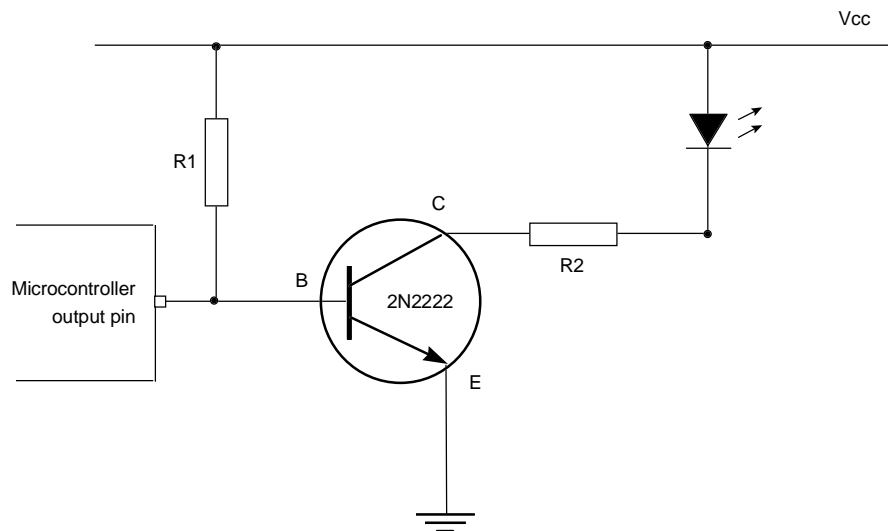


Figure 9: Driving an LED via a transistor (NPN) driver. A logic 1 output is required to light the LED. See text for details of resistor values.

On the 8051 family, R1 is only usually required when using Port 0, which has no internal pull-up resistors, and can be omitted when using the other ports. When R1 is used, any value between 1K and 10K will allow correct operation.

An appropriate value of R2 should be chosen depending on the LED used. Assuming that we have a supply V_{cc} , that the voltage drop across the LED is 2V, and that the transistor is in saturation (we will assume 0.4V voltage drop, V_{ce-sat}), then Ohm's law allows us to calculate the required value of R2 (Equation 1).

$$R2 = \left[\frac{V_{cc} - V_{LED} - V_{ce-sat}}{I_{LED}} \right]$$

Equation 1: Calculating resistor values for transistor switches.

In this equation, I_{LED} is the current required to light the LED at an appropriate brightness: 15mA is a common value, but the data sheet for the LED used will provide accurate information.

Note that, when working with 3V microcontrollers, the range of available buffer ICs is more restricted: in these situations, use of transistors may be a very useful alternative: we give an example of this below.

Further details information about this circuit, and some alternative approaches that may be used with high-powered LEDs, are given in SWITCHED DC OUTPUT [page >>>].

Software

We assume that, as described above, an LED is connected to a single output pin. Using the C51 compiler, or equivalent, (see Chapter 4), we can control the LED in software on a 'port-by-port' or

'pin-by-pin' basis. For example, assuming that the LED is connected to Pin 0 on Port 3 of an 8051-family microcontroller, we can flash the diode by controlling the whole port, as follows:

```
P3 = 0xFF;
... // delay
P3 = 0x00;
... // delay
P3 = 0xFF;
... // etc
```

Alternatively, we can usually make use of an sbit variable (see Chapter 4) to provide a finer level of control. At the same time, we consider the fact that - depending on the hardware (see below) - the LED may be lit using a logic 1 or a logic 0 output on the port pin, and we make the software flexible enough to deal easily with subsequent hardware changes:

```
#define LED_PORT P3

#define LED_ON 0          // Easy to change the logic here
#define LED_OFF 1

sbit Warning_led = LED_PORT^0; // LED is connected to 4.0

...

Warning_led = LED_ON;
... // delay
Warning_led = LED_OFF;
... // delay
Warning_led = LED_ON;
... // etc
```

Strengths and weaknesses

Single LEDs, controlled in the manner described here, are widely used to indicate that embedded equipment is operational or activated; flashing LEDs are frequently used as a warning. LED OUTPUT can be applied successfully to a range of LED-based display devices.

The main consequences of using the pattern are that the LEDs themselves are comparatively high-power devices. For example, a typical (small) single LED requires 2v at 10 mA to produce a reasonable display, and thereby consumes a power of about 20 mW: by contrast, a complete LCD panel able to display some 30 to 60 characters will consume around the same amount of power (**excluding** any backlighting): see LCD PANEL OUTPUT [page >>>]. This relatively high power consumption can make the use of LEDs in battery-powered equipment impractical. Note that when using single LEDs in such applications, power consumption can be reduced by 'flashing' the LED: an example of this is given in association with SIMPLE DELAY [Page 11].

The main decision to be made when driving a small LED (current requirement ~15mA) is whether or not to use a buffer. In these circumstances, use of a buffer may increase production costs by around \$0.30 per LED in medium volumes. However, if there is any possibility of the LED suffering damage while the product is in use, a buffer may be a good option: it is almost always cheaper to replace a blown buffer (~\$0.10) than it is to replace a blown microcontroller (~\$1.00+). Overall, in very low cost (or high volume) products, or in situations where repair is not a practical proposition, then use of a buffer will simply add to production costs. For low-volume and / or high-cost products where repairs may be required, then buffers are a good solution.

Related patterns and alternative solutions

The main alternative to the techniques described here for driving single LEDs are high-power buffers described in SWITCHED DC OUTPUT [page >>>].

Techniques for driving multi-segment LEDs are discussed in MULTI-SEGMENT LED OUTPUT [page >>>] and LED PANEL OUTPUT [page >>>]. Where more information (for example, complete words or lines of text) must be displayed, and particularly where power consumption is a concern, then LCD panels are the main alternative to LEDs: see LCD PANEL OUTPUT [page >>>].

Example: Driving an LED from a 3V microcontroller

Some modern microcontrollers (such as the C501 GV member of the 8051 family from Siemens) are 3V devices. Such devices have the benefit of lower power consumption, making them ideal in battery-powered applications. However, the range of available 3 V peripheral devices, such as buffers, is more restricted than the corresponding range of 'traditional' 5 V devices. Use of transistor-based buffer can be particularly valuable in such circumstances.

The required resistor values may be easily calculated (see Figure 9). If, for example, we are connecting a small LED with a voltage drop (V_{LED}) of 2V and a required current (I_{LED}) of 15 mA to a transistor in saturation ($V_{ce-sat} = 0.4V$) using a 3 V supply (V_{cc}) on Port 1 of a C501GV, then the required value of R2 (in Figure 9) is 40 Ω . If we are using 'standard' resistors, then the nearest available value is 39 Ω : this will be fine. Because we are using Port 1, which has internal pull-up resistors, R1 may be omitted.

Example: Driving a high-power IR LED transmitter

Infra-red (IR) LEDs are widely used in domestic applications for remote control. They are also a component in many security systems. IR LEDs often have higher current requirements than conventional LEDs, but are otherwise connected in the same way.

For example, the Siemens SFH485 IR LED requires a current of 100mA, at a forward voltage of 1.5V. If we wish to drive this LED using (say) Port 1 of an 8051 microcontroller, then the circuit from Figure 9 can be used, as the 2N2222 transistor can handle a maximum collector current of up to 800mA.

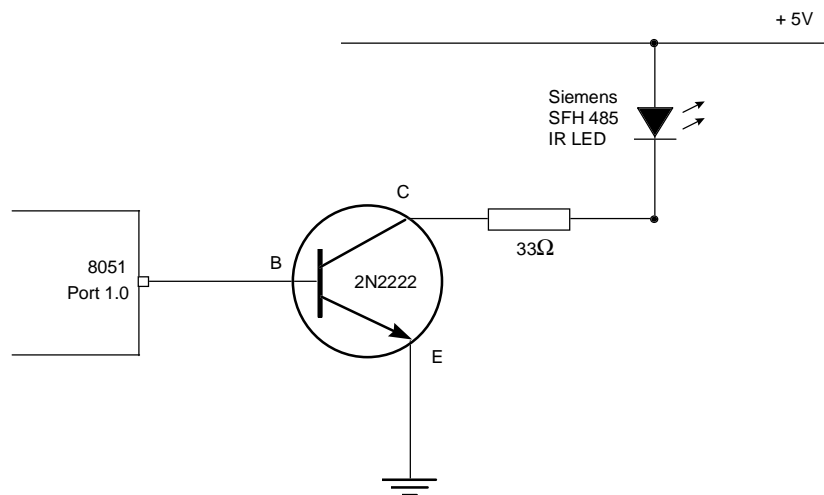
With I_{LED} at 100mA, a supply voltage of 5V and an LED forward voltage of 1.5V, the required value of R2 is:

$$R2 = \left[\frac{5.0V - 1.5V - 0.4V}{0.100A} \right] = 31\Omega$$

Again, we assume a saturation voltage (V_{CE}) for the transistor of 0.4V.

Here we would use the next standard resistor value, in this case 33 Ω . Note that the nearest value is 30 Ω : however, by using this we run the slight risk of running the LED at too high a forward voltage, and reducing the life of this component.

The resulting circuit is shown in Figure 10.



*Figure 10: Driving an IR LED via a transistor (NPN) driver.
A logic 1 output is required to generate an IR output.*

Additional examples

For further examples of the use of this pattern, see:

- SIMPLE DELAY [page 11];
- SP SWITCH (POLLED) [page 16];
- >>>

Further reading

Horowitz, P. and Hill, W. (1989) “The art of electronics”, Second edition, Cambridge University Press, Cambridge, UK.

SIMPLE DELAY

Context

You are developing an embedded application for an 8051 microcontroller (see Chapter 3) 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

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 cycle³: 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 Delay' routines in a wide range of embedded software systems.

Reliability and safety issues

The techniques discussed in SIMPLE 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.

³ In the 8051 family, one processor cycle (one instruction cycle) corresponds to 12 clock cycles in many 'traditional' devices (e.g. 80c51), while in recent devices (such as the Siemens 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 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.

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.

The main weaknesses are that:

- 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 DELAY to implement a real-time clock!
- 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 DELAY is probably of most value if your application is ‘naked’ (see NAKED [page >>>]): that is, you are not using a SCHEDULER [page >>>] or a real-time operating system (see RTOS [page >>>]): 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 DELAY. For example:

- DELAY FLAG [page >>>] 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 [page >>>] provides a means of accurately measuring elapsed time, and thus - for example - of implementing a real-time clock.
- GATED TIMER [page >>>] provides a means of accurately measuring the time between two external events, and thus - for example - of measuring a pulse width.
- SCHEDULER [page >>>] and RTOS [page >>>] each 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 DELAY pattern.

Hardware

The single LED (or similar device - e.g. 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 LED OUTPUT [Page 2] for hardware details.

Note: the example program may be simulated effectively using dScope without any hardware, as discussed in Chapter 4.

Software

To implement the flashing LED in software we will use an endless loop involving two ‘Simple 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 $1/12^{\text{th}}$ of the crystal frequency (see 8051 OSCILLATOR [page >>>] for further details and notable exceptions): in this case, 921.6 kHz. The instruction cycle period is therefore $1.085 \mu\text{s}$, 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 (in 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 DELAY pattern for 8051 microcontroller
//
// Also illustrates: LED OUTPUT
//
// Flashes a single LED connected to P1.0 (+ve logic: see LED OUTPUT)
// This version does not use ISRs
//
// Compile for 8051 using Keil compiler
// May be simulated (slowly!) on dScope without hardware
// - use 8051.dll (see Ch.4)

// Source file: Delay.C

#include <reg51.h> // Special function register 8051 (see Ch.4)

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

void Simple_Delay_N_Seconds(const int);

/* ..... */
// Pattern: LED OUTPUT
#define LED_ON (1)
#define LED_OFF (0)

sbit Led_G = P1^0;

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

    while(1) // Endless loop (see Ch.4)
    {
        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 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 DELAY.

Additional examples

For further examples of the use of this pattern, see:

- SP SWITCH (POLLED) [page 16];
- >>>

Further reading

-

SP SWITCH (POLLED)

Context

You are developing an embedded application for an 8051 microcontroller (see Chapter 3) or similar device.

Problem

You need to connect the port of a 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.

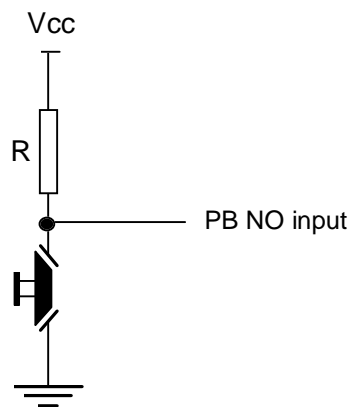


Figure 11: An example of a push-button ('normally open') switch input.

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) 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.

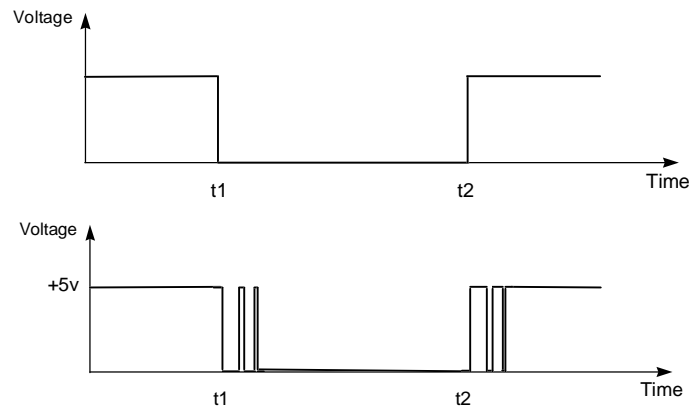


Figure 12: The voltage signal resulting from the switch shown in Figure 11. [Top] Idealised waveform resulting from a switch depressed at time t_1 and released at time t_2 [Bottom] Actual waveform showing leading edge bounce following switch depression and trailing edge bounce following switch release.

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.

Reliability and safety issues

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 13). 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.

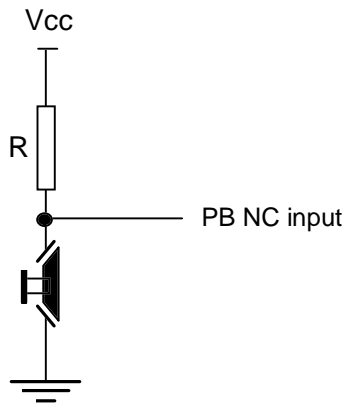


Figure 13: An example of a push-button ('normally closed') switch input.

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 14). 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. The use of double-pole switches is discussed in the pattern DP SWITCH [page >>>].

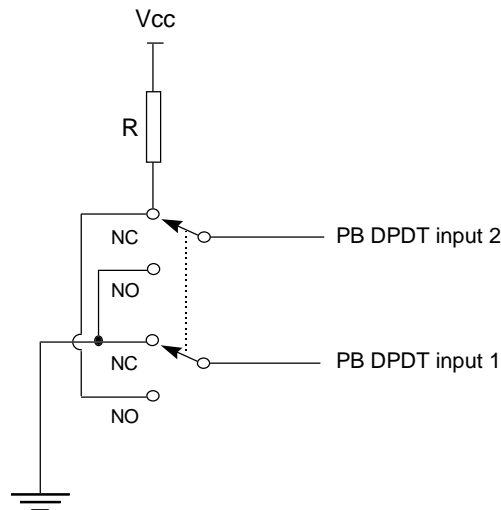


Figure 14: An example of a push-button DPDT switch input. With the arrangement shown, Input 1 will have a voltage of 0V and Input 2 Vcc with the switch open: these values will be reversed when the switch is closed.

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.

Solution

The polled software solution for single-pole switches is straightforward. We poll the input port regularly (approximately every 100 ms is usually sufficient). If we think we have detected a switch depression (or release), we keep monitoring the input for about 20 ms. If, during the monitoring period, the output becomes stable, we have detected a closure.

Note that the figure of '20 ms' will 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.

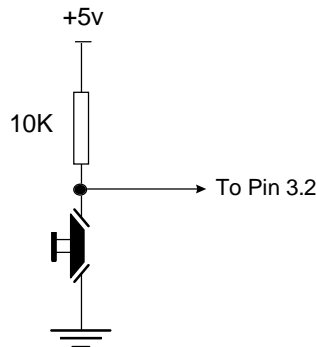


Figure 15: The hardware layout used for a simple switch input.

The basic algorithm is as follows:

- Check for input. In this case (Figure 15) 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
    PONT ET AL., "A SELECTION OF PATTERNS FOR RELIABLE EMBEDDED SYSTEMS" [EUROPLOP 1999]
    PAGE 18, LAST EDITED: 26 MAY, 1999
```

```

// - 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 SP SWITCH (POLLED).

Dealing with sustained switch depressions - 1

A possible problem with the above code is that one sustained switch depression will 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 will be called 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 SP SWITCH (POLLED). This version waits for key release.

Dealing with sustained switch depressions - 2

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 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 SP SWITCH (POLLED). This version interprets sustained depressions as faults.

Dealing with sustained switch depressions - 4

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 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 SP SWITCH (POLLED). This version implements a 'trinary' input routine.

Strengths and weaknesses

The main strengths of SP SWITCH (POLLED) are as follows:

- It 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 (see Chapter 5).
- 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.

The main weaknesses of SP SWITCH (POLLED) are as follows:

- A complete implementation may require a substantial amount of code to ensure reliable operation.
- Like all software-only input techniques, SP SWITCH (POLLED) 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 against this. See HARDWARE-DEBOUNCED SWITCH [page >>>] for a discussion of this issue.
- For the same reasons, SP SWITCH (POLLED) 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. Again, HARDWARE-DEBOUNCED SWITCH [page >>>] for a discussion of this issue, and a solution.
- See ‘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 an alternative software-based solution to the switch debouncing problem, consider SOFTWARE-DEBOUNCED SWITCH (INTERRUPT). For an alternative approach using hardware, consider HARDWARE-DEBOUNCED SWITCH.

Where the switch (or other input device) does not suffer from bounce (e.g. 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 DISCRETE INPUT [page >>>].

Example: Counting key presses

This example illustrates the use of SP SWITCH (POLLED) 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 SP Switch (polled) pattern
// for 8051 microcontroller
//
// Also illustrates: SIMPLE DELAY; LED OUTPUT
//
// Counts number of depressions of a SPST pushbutton switch (P1^0)
// Displays result on bank of LEDs (P3)
//
// Compile for 8051 using Keil compiler
// May be simulated (slowly!) on dScope without hardware
// - use 8051.dll (see Ch.4)

// Source file : SwitchP.C

#include <reg51.h> // Special function register 8051 (see Ch.4)

/* ..... */
// Pattern: SP Switch (polled)
#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 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 SP Switch (polled) 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 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 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 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 SP SWITCH (POLLED).

Additional examples

For further examples of the use of this pattern, see:

- >>>

Further reading

FUNCTION TOKEN

Context

You are developing an embedded application for an 8051 microcontroller (see Chapter 3) or similar device.

Problem

You need to develop a highly-reliable software system in an environment where incorrect (or inappropriate) function calls may be made. Such calls may result, for example, through the presence of high levels of electromagnetic interference (EMI).

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’): this 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.

In many situations, it is better to limit the impact of EMI through appropriate hardware design (see Chapter 5): however, various software-based measures are also effective.

Reliability and safety issues

The aim of this pattern is to ensure the reliability and safety of embedded applications.

Solution

To tackle some of the problems caused by EMI, 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.

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

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).

However, 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. Two key patterns that are frequently used in association with token-based protection schemes are FULL MEMORY [page >>>], and WATCHDOG PROTECTION [page >>>].

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).

Additional examples

For further examples of the use of this pattern, see:

- SIMPLE DELAY [page 11];
- SP SWITCH (POLLED) [page 16];
- LED OUTPUT [page 2].
- >>>

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.

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