PAPER PRESENTED AT VIKINGPLOP 2003 (BERGEN, NORWAY, SEPTEMBER 2003)

Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language

Michael J. Pont and Chisanga Mwelwa
Embedded Systems Laboratory, Department of Engineering, University of Leicester, University Road, LEICESTER LE1 7RH, UK.
`M.Pont@le.ac.uk, cm55@le.ac.uk`
`http://www.le.ac.uk/eg/embedded/`

Introduction

We have previously described a "language" consisting of more than eighty patterns, which will be referred to here as the "PRES Collection"*. This language is intended to support the development of reliable embedded systems using small resource-constrained microcontrollers, including - for example - devices from the 8051 family with a few hundred bytes of available memory.

The first complete set of these patterns was completed around three years ago and they have since been used in a range of industrial systems, numerous university research projects, as well as in undergraduate and postgraduate teaching on many university courses (e.g. see Pont, 2003; Pont and Banner, in press). We have also begun to develop a tool to support the development of embedded systems using these patterns (Mwelwa and Pont, 2003).

As our experience with the collection has grown, we have begun to add a number of new patterns and revised some of the existing ones (e.g. see Pont and Ong, 2003; Pont *et al.*, 2003; Key *et al.*, 2003). Inevitably, by definition, a language consists of an inter-related set of patterns: as a result, it is unlikely that it will ever be possible to refine or extend such a system without causing some side effects. However, as we have worked with this collection, we have felt that there were ways in which the overall architecture could be improved in order to reduce the impact of future changes.

This paper briefly describes some of the main alterations we have made when re-factoring our original pattern collection. It then goes on to describe one of the new patterns that has resulted from this process.

Two processor families

One of the most significant limitations of our first version of this collection was that we used only a single (8051) hardware platform to illustrate the use of the patterns. This tended to obscure the fact that, in most cases, the patterns are applicable to a wide range of different processors and systems.

In the revised patterns, we present examples for both the 8051 family of microcontrollers <u>and</u> the ARM family of devices (Figure 1). These devices are both common choices for embedded designs, but the 8051 family (of 8-bit processors) tend to be used in "low end" applications, while the ARM

---

* Our original patterns focused on "time-triggered" designs, and was known as the "PTTES" collection (after the original book title: "Patterns for Time-Triggered Embedded Systems"). Since then the collection has been expanded and broadened, and - while the focus remains on reliable designs - not all of the patterns are time-triggered. The collection has therefore been renamed the "PRES" ("Patterns for Reliable Embedded Systems").

family (of 16/32-bit processors) tend to be used in applications requiring more memory or CPU performance. Coverage of both families has helped us to focus on the "core" features of the revised patterns, and we have moved the implementation-specific details into "example" sections.

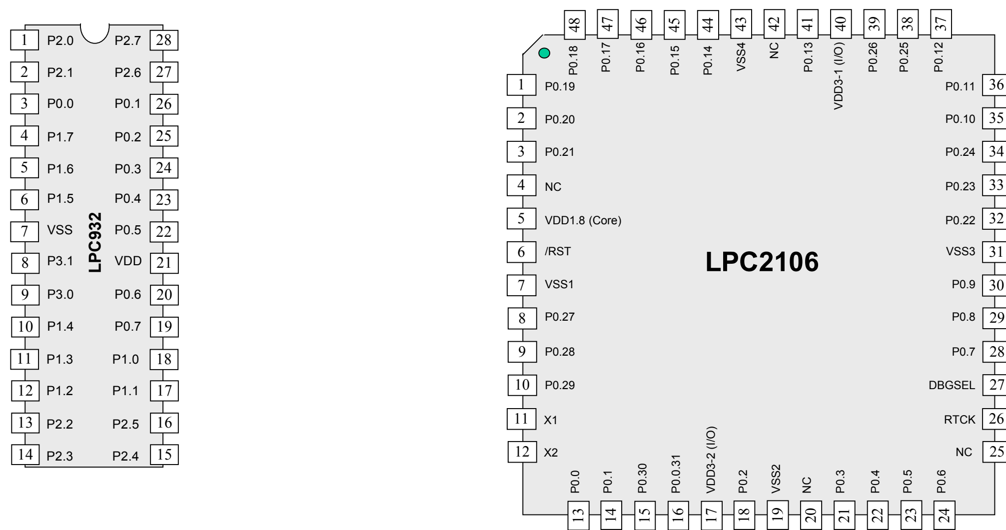

Figure 1: An external view of the two embedded processors used in the revised pattern collection. The Philips LPC932 (left) is an 8-bit device from the popular 8051 family of devices. The Philips LPC2106 (right) is based on the 16-/32-bit ARM7TDMI processor core.

Prototyping on a PC

In addition to the 8051 and ARM devices mentioned in the previous section, the revised patterns also include discussions about the creation of system prototypes using a desktop PC. We see these examples being of value primarily to developers who wish to use a PC platform to "prototype" a complex embedded design, prior to implementation using a microcontroller or similar device. They may also be of interest to companies, universities and colleges who require a cost-effective way of training people in the creation of software for embedded systems. Finally, they may be of use to "hobby" developers, who wish to gain experience with embedded software using low-cost hardware.

Something old, something new …

We present two patterns in this paper: the first is PORT WRAPPER (a new pattern); the second is PORT HEADER, which first appeared in our original pattern collection in 2001.

Please note that PORT HEADER (the pattern upon which PORT WRAPPER was based) is presented only "for reference", in order to illustrate the changes we are making (to pattern structure and content) as we revise the collection.

Tools used

To develop the microcontroller code presented in this paper, we used the Keil C51 and Keil / GNU ARM tools. Evaluation versions of these compilers are available from: `http://www.keil.com`

To develop the PC code, we used the "Open Watcom" compiler. This compiler is available (for free download) here: `http://www.openwatcom.org/`

Acknowledgements

We are very grateful to our Shepherd at VikingPLoP, Alan O'Callaghan, who provided a number of detailed (and very useful) suggestions on the drafts of this paper.

We are also grateful to the participants in our workshop at VikingPLoP (Neil Harrison, Klaus Marquardt, Bernhard Gröne and Peter Tabeling) who provided many further useful comments. In addition, after the conference, Neil Harrison was kind enough to look again at our revised paper.

We would also like to thank Chris Hills (Keil, UK) and Trevor Martin (Hitex, UK) who have provided invaluable support for the work described in this paper.

Last but not least, we'd like to thank Simon Plumtree (Addison-Wesley / Pearson Education, UK) for his continuing faith in this large project.

Copyright

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

PORT WRAPPER

Context

- You are developing a microcontroller-based embedded application using an 8051, ARM or similar processor.
- Reliability is an important design consideration.
- Your system will be developed and / or ported and / or maintained by several people. This may mean that a team of developers will be involved in the system construction. Alternatively, it may mean that your system is developed by a "one-person team" and will subsequently be maintained or ported by someone else.
- You want to minimise the memory and processor requirements, in order to allow use of cheaper hardware platforms, and therefore minimise the system unit costs.
- You want to keep the development costs (for current and future projects) as low as possible. You therefore need to make it easy to port code for part of your current system to a new project, and / or to port some or all of the system to a new hardware platform.
- You are programming in C.

Problem

How can you manage the allocation and initialisation of port pins in a non-trivial project?

Design constraints

In a typical embedded project, you may have a user interface created using an LCD, a keypad, and one or more single LEDs. There may be a serial (for example, RS-485 or CAN) link to another microcontroller board. There may be one or more high-power devices (say 3-phase industrial motors) controlled by your application.

Typically, the project will be structured (in 'C'), using a separate file for the LCD interface, another for the keypad, and so on. Each of these files will then require access to some of the microcontroller port pins, and code like that shown in Figure 2 is common.

File: "LCD.C":

```
sbit Pin_A = P3^2;
...
```

File: "Keypad.C":

```
#define Port_B = P0;
...
```

File: "LED.C"

```
sbit Pin_C = P2^7;
...
```

*Figure 2: An informal approach for controlling port pins in a simple embedded project. The various files in the project make direct access to different port pins on the processor. This approach is **NOT** recommended, for reasons discussed in the text.*

When developing this type of code, you may face the following design constraints:

- You want to ensure that there are no "pin conflicts": for example, you don't want to have the people producing the LCD code trying to use the same pins as the people producing the switch code, or the 3-phase motor driver. This can be very dangerous, if your system involves high-powered loads and inadequate testing is performed. It may also result in the need for code re-design - and expensive PCB changes - if not detected early enough in the project lifecycle.

- You want to make sure that all the port pins are initialised correctly. If this is not done, the system may not operate as required. For example, suppose that you have correctly configured "Pin X" on your microcontroller to "input" mode, for use with an emergency cut-off switch. If, late in the project, a new component is added which inadvertently sets Pin X to "output" mode, a user of the system may have no way of stopping the system in an emergency.

Solution

Creating a suitable PORT WRAPPER can help you manage port access allocations in a non-trivial project, and thereby meet the design constraints highlighted in the previous section.

A PORT WRAPPER is made up of the following components:

- A port header file that, at a minimum, contains a simple list of all of the port accesses in your project.
- A port initialiser. This is a mechanism for configuring all the pins in your system for input, output, etc.

Various possible implementations are possible: some common approaches are discussed below.

Implementation

Basic implementation

The two components identified in Solution are easily implemented, often using a pair of files "Port.H" and "Port.C".

Port.H

Port.H may simply contain a list of all the port accesses in your project, and link these to the corresponding files. For example, it may look like this:

```
/* port.h */

/ * File: lcd.c
    This uses pins 0.0 to 0.4, as follows … */

/* File: keypad.c
   This uses pins … */
```

Inevitably, such documentation can get "out of step" and - for all the usual reasons - code that is "self documenting" is generally desirable. This can be done using `#define` statements and similar mechanisms to ensure that the header file remains consistent with the evolving code.

Several examples of complete Port.H files are given below.

Port.C

Port.C will usually contain a function that initialises the various port pins, as required. For example, consider the following example:

```
/* port.c */

void PORT_Init(void)
   {
   /* Prepare for lcd (see lcd.c) */

   ...

   /* Prepare for keypad (see keypad.c) */

   ...
   }
```

Several examples of complete Port.C files are given below.

How many wrappers?

If your development is being carried out by a team of individuals, then you may face difficulties if multiple teams wish to make changes (simultaneously) to a shared pair of Port.C and Port.H files.

The simplest solution is to work, initially, with a different pair of files for each component. For example, the LCD interface may have its own Port.C and Port.H files. During the integration phase of the project, all the various port files will be combined. This is a simple solution, and avoids changing references to the wrapper files in different parts of the project after integration.

An alternative is to create distinct wrapper files for each component (for example, Port_LCD.H). These can be left as distinct files when the project is integrated. This solution has the advantage that the various wrapper files do not need to be altered at the time of system integration (that is, the final project will have a distinct pair of wrapper files for each component). This can make porting and maintenance more straightforward.

Note that in both of these cases, a degree of discipline is required in order to ensure that developers of each component are aware of port pin usage by other developers.

Reliability and safety implications

We consider some key reliability and safety implications associated with port access in this section.

Pin reset values

Whatever processor you are using, you need to pay close attention to the pin reset values.

Consider, for example, that you have connected a motorised device to a port, and that the device is activated by a 'Logic 1' output. If, by default, your microcontroller sets the pins to "1" at reset, the motorised device will move. Even if you change the port outputs to 0 at the start of your program, the motor will be 'pulsed' briefly. This can, in some systems, lead to the injury or even death of users of the system, or those in the immediate vicinity.

It is very important to ensure that your hardware design and software design assume compatible reset values. More specifically, it is generally better to try and ensure that your microcontroller pin reset values are "0" and that high-powered external devices are operated with a pin value of "1".

Pin conflicts

Despite its simplicity, PORT WRAPPER can improve system reliability, because it avoids potential conflicts between port pins, particularly during the maintenance phase of the project when developers (who may not have been involved in the original design) are required to make code changes.

Hardware requirements

Use of a PORT WRAPPER will have minimal hardware requirements.

Cost implications

Any documentation-based technique has an initial cost implication, since any form of comments or notes takes time to produce. However, any initial costs are likely to be repaid when the job of maintaining or porting the code becomes easier.

Maintenance

PORT WRAPPER is - largely - a manual documentation technique, based on coding conventions. It relies for correct operation on a disciplined approach to software development from the whole team. Sometimes, particularly as deadlines get tight, documentation can get out of step with the system.

If you are asked to maintain a system that uses a PORT WRAPPER, it can be a good idea to check that the information recorded is in fact correct before making any changes.

Portability

Use of PORT WRAPPER should help to improve portability, by making accessible - in one location - all of the port access requirements of the application. Please note that - as discussed in "Maintenance" - it is always a good idea to check that the wrapper documentation matches the system hardware before beginning a porting exercise.

Note that if you wish to port only part of the application (say the RS-485 interface) this may be easier if you have used the multi-wrapper approach (see "Implementation").

Related patterns and alternative solutions

Hardware issues

This pattern does not deal with hardware design: see Pont, 2001, Part A (particularly Chapter 7 and Chapter 8) and Part C for patterns that cover these issues.

Code libraries

In a desktop design (or a large embedded system without severe resource constraints), the "obvious" solution to the problems addressed by Port Wrapper would be to create a class or function library that fully encapsulated the port-pin control and initialisation. Such a library could be used to detect (automatically) many problems with port initialisation or port conflicts. Unfortunately, on systems with severe resource constraints, use of libraries must usually be avoided, because of the resulting overheads. If you choose to develop and use such a library, you may restrict the portability of your code framework.

Overall strengths and weaknesses

☺ PORT WRAPPER can help to reduce the chances of pin conflicts (that is, the same pins being used, for different purposes) in two or more system modules.

☺ PORT WRAPPER can help to ensure that your pins are initialised correctly.

☺ PORT WRAPPER imposes minimal CPU or memory requirements.

☹ PORT WRAPPER is - largely - a manual technique, based on coding conventions. For correct operation it relies upon a disciplined approach to software development (from the whole team).

☹ The techniques described in PORT WRAPPER may prove less practical in very large embedded projects. Such projects are rare with the resource-constrained processors considered in this pattern but - if you need to develop for such a project - you may find it more appropriate to use a code library.

Example: Flashing an LED on the LPC900 family

We illustrate how a PORT WRAPPER can be created for a device from the LPC900 family in this simple example.

```
#ifndef PORT_H
#define PORT_H 1

// Prototype for init function
void Port_Init(void);

// ------ LED_Flas.C -----------------------------------------------

// Connect LED from +5V to this pin, via appropriate resistor
sbit LED_pin = P2^0;

#endif
```

Listing 1: Part of a PORT WRAPPER example for the LPC900 family (port.h).

```
void PORT_Init(void)
   {
   // Assuming Port 2, Pin 0 used for LED
   // Set this pin to push-pull output
   // [Relevant SFRs are not bit addressable]
   P2M1 &= 0xFE;
   P2M2 |= 0x01;
   }
```

Listing 2: Part of a PORT WRAPPER example for the LPC900 family (port.c).

```
int main(void)
   {
   // Initialise the port pins as required
   Port_Init();

   // Prepare to flash LED
   LED_FLASH_Init();

   while(1)
      {
      // Change the LED state (OFF to ON, or vice versa)
      LED_FLASH_Change_State();

      // Delay for *approx* 1000 ms
      LOOP_DELAY_Wait(1000);
      }

   return 1; // Should never reach here ...
   }
```

Listing 3: Part of a PORT WRAPPER example for the LPC900 family (main.c).

Example: Controlling LEDs with the LPC2100 family

We illustrate how a PORT WRAPPER can be created for a device from the LPC2100 family in this simple example.

```
#ifndef PORT_H
#define PORT_H 1

// Prototype for init function
void PORT_Init(void);

// ------ LED_Flas.C -----------------------------------------------

// Connect LED from ground to this pin, via appropriate resistor
// LED pin is GPIO pin 0.0
#define LED_pin 0x00000001

#endif
```

Listing 4: Part of a PORT WRAPPER example for the LPC2100 family (port.h).

```
#include "main.h"
#include "port.h"

/*------------------------------------------------------------------*/

void PORT_Init(void)
   {
   // 1. Set up LED pin as GPIO
   // 2. Set to output mode

   // 1.
   // Just pin 0.0 as GPIO
   PINSEL0 &= 0xFFFFFFFE;

   // 2.
   IODIR = LED_pin;
   }
```

Listing 5: Part of a PORT WRAPPER example for the LPC2100 family (port.c).

```
#include "main.h"
#include "port.h"

#include "led_flas.h"
#include "loop_del.h"

/*..................................................................*/

int main(void)
   {
   // Initialise the port pins as required
   PORT_Init();

   // Prepare to flash LED
   LED_FLASH_Init();

   while(1)
      {
      // Change the LED state (OFF to ON, or vice versa)
      LED_FLASH_Change_State();

      // Delay for *approx* 1000 ms
      LOOP_DELAY_Wait(1000);
      }

   return 1;  // Should never reach here ...
   }
```

Listing 6: Part of a PORT WRAPPER example for the LPC2100 family (main.c).

Example: Flashing an LED on a Standard 8051

We illustrate how a PORT WRAPPER can be created for a Standard 8051 device in this simple example.

```
#ifndef PORT_H
#define PORT_H 1

// Prototype for init function
void Port_Init(void);

// ------ LED_Flas.C -----------------------------------------------

// Connect LED from +5V to this pin, via appropriate resistor
sbit LED_pin = P2^0;

#endif
```

Listing 7: Part of a PORT WRAPPER example for the "Standard 8051" (port.h).

```
#include "main.h"
#include "port.h"

/*...................................................................*/

void PORT_Init(void)
   {
   /*

   In the Standard 8051, the ports default to "output" mode.

   No initialisation is therefore required in this simple example.

   We simply record that Port 2, Pin 0 is used to control the LED.

   */
   }
```

Listing 8: Part of a PORT WRAPPER example for the "Standard 8051" (port.c).

```
int main(void)
   {
   // Initialise the port pins as required
   Port_Init();

   // Prepare to flash LED
   LED_FLASH_Init();

   while(1)
      {
      // Change the LED state (OFF to ON, or vice versa)
      LED_FLASH_Change_State();

      // Delay for *approx* 1000 ms
      LOOP_DELAY_Wait(1000);
      }

   return 1; // Should never reach here ...
   }
```

Listing 9: Part of a PORT WRAPPER example for the "Standard 8051" (main.c).

Example: Flashing an LED on a PC parallel port

We illustrate how a PORT WRAPPER can be created for a PC (prototype) in this simple example.

As before, the example simply flashes an LED. The LED should be connected to Pin 0 on the data port (Pin 2 on the parallel port), as illustrated in Figure 3.

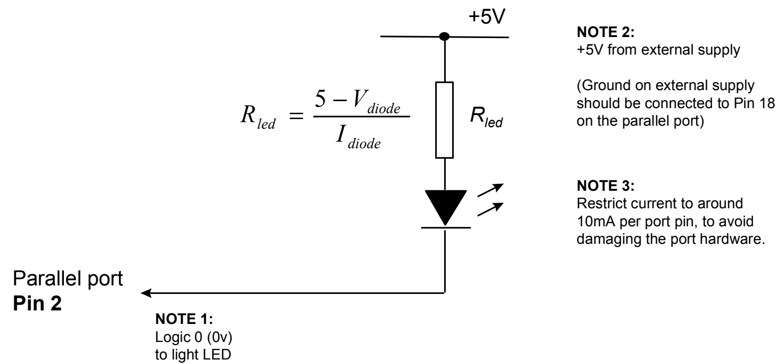

Figure 3: *Connecting an LED to the parallel port.*

```
// Setting for LPT1:
// See paper for details
#define LPT1_DATA     0x0378
#define LPT1_STATUS   0x0379
#define LPT1_CONTROL  0x037A

// ------ Public function prototypes -------------------------------

void PORT_Init(void);
void PORT_Write_Bit(const tWord PORT, const tWord PIN, const tWord VALUE);
```

Listing 10: Part of a PORT WRAPPER example for use with a PC-based system prototype (port.h).

```
#include "Main.h"
#include "Port.h"
#include <stdio.h>
#include <conio.h>

/*..................................................................*/

void PORT_Init(void)
   {
   /*
   In the PC, the ports default to "output" mode.

   No initialisation is therefore required in this simple example.

   We simply record that Pin 0 on the data port (Pin 2 on the
   parallel port) is used to control the LED.
   */
   }
```

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

void PORT_Write_Bit(const tWord PORT, const tWord PIN, const tWord VALUE)
   {
   tWord p = 0x01;
   tWord Old, New;

   p <<= PIN;

   // Read current port value
   Old = inp(PORT);

   // If we require a 1 at the pin
   if (VALUE == 1)
      {
      New = Old | p;
      }
   else
      {
      p = ~p; // Complement
      New &= p;
      }

   // Write the new value
   outp(PORT,New);
   }
```

Listing 11: Part of a PORT WRAPPER example for use with a PC-based system prototype (port.c).

```
#include "main.h"
#include "port.h"
#include "led_flas.h"
#include "loop_del.h"

/*..................................................................*/

int main(void)
   {
   // Initialise the port pins as required
   PORT_Init();

   // Prepare to flash LED
   LED_FLASH_Init();

   while(1)
      {
      // Change the LED state (OFF to ON, or vice versa)
      LED_FLASH_Change_State();

      // Delay for *approx* 1000 ms
      LOOP_DELAY_Wait(1000);
      }

   return 1; // Should never reach here ...
   }
```

Listing 12: Part of a PORT WRAPPER example for use with a PC-based system prototype (main.c).

Background material and further reading

-

PORT HEADER

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.

Problem

How do you manage port access allocations in a larger project?

Background

In a typical embedded project, you may have a user interface created using an LCD, a keypad, and one or more single LEDs. There may be a serial (RS-485) link to another microcontroller board. There may be one or more high-power devices (say 3-phase industrial motors) controlled by your application.

Each of these (software) components in your application will require exclusive access to one or more port pins. The project may include 10 - 20 different source files. How do you ensure that changes to port access in one component does not impact on another? How do you ensure that it is easy to port the application to an environment where different port pins must be used?

These issues are addressed through the simple PORT HEADER design pattern.

Solution

PORT HEADER encapsulates a simple but effective design guideline that can help you cope with the fact that many different components in a larger project will each require port access: specifically, using PORT HEADER, you will pull together the different port access features for the whole project into a single (header) file. Use of this technique can ease project development, maintenance and porting.

PORT HEADER is simple to understand and simple to apply.

Consider, for example, that we have three C files in a project (A, B, C), each of which require access to one or more port pins, or to a complete port.

File A may include the following:

```
// File A

sbit Pin_A = P3^2;

...
```

File B may include the following:

```
// File B

#define Port_B = P0;

...
```

File C may include the following:

```
// File C

sbit Pin_C = P2^7;

...
```

In this version of the code, all of the port access requirements are spread over multiple files. Instead of this, there are many advantages obtained by integrating all port access in a single `Port.H` header file:

```
// ----- Port.H -----

// Port access for File B
#define Port_B = P0;

// Port access for File A
sbit Pin_A = P3^2;

// Port access for File C
sbit Pin_C = P2^7;

...
```

Each of the remaining project files will then '`#include Port.H`'.

Listing 13 shows a complete example of a `Port.H` file from a real application.

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

   Port.H (v1.00)

  ------------------------------------------------------------------

  'Port Header' (see Chapter 10) for the project LCD_KEYP.PRJ

-*------------------------------------------------------------------*/

// ------ Sch51.C ----------------------------------------

// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS

#ifdef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif

// ------ Keypad.C -------------------------------------------------

#define KEYPAD_PORT P0

sbit C1 = KEYPAD_PORT^0;
sbit C2 = KEYPAD_PORT^1;
sbit C3 = KEYPAD_PORT^2;

sbit R1 = KEYPAD_PORT^6;
sbit R2 = KEYPAD_PORT^5;
sbit R3 = KEYPAD_PORT^4;
sbit R4 = KEYPAD_PORT^3;

// ------ LCD_A.c ---------------------------------------------------

// NOTE: Any combination of 6 pins may be used (any ports, any order)
// NOTE: Number in [] are pin numbers on *MANY* LCDs

sbit LCD_D4 = P1^0; // DB4 [11]
sbit LCD_D5 = P1^1; // DB5 [12]
sbit LCD_D6 = P1^2; // DB6 [13]
sbit LCD_D7 = P1^3; // DB7 [14]

sbit LCD_RS = P1^4; // Display register select output [4]
sbit LCD_EN = P1^5; // Display enable output [6]

// Connect Vss [1] on LCD to Gnd
// Connect Vcc [2] on LCD to +5V
// Connect Vo  [3] on LCD to Gnd
// Connect RW  [5] on LCD to Gnd

// ------ LED_Flas.C -----------------------------------------------

// Connect LED from +5V (etc) to this pin, via appropriate resistor
// [see Chapter 7 for details]
sbit LED_pin = P1^5;


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

Listing 13: An example of a real Port Header file (Port.H) from a project using an interface consisting of a keypad and liquid crystal display.

Hardware resource implications

There are no hardware resource implications.

Reliability and safety implications

Despite its simplicity, PORT HEADER can improve reliability and safety, because it avoids potential conflicts between port pins, particularly during the maintenance phase of the project when developers (who may not have been involved in the original design) are required to make code changes.

Portability

PORT HEADER is itself portable: it can be used with any microcontroller, and is not linked to the 8051 family. Use of PORT HEADER also improves portability, by making accessible, in one location, all of the port access requirements of the application.

Overall strengths and weaknesses

☺ PORT HEADER is both simple and effective - use it!

Related patterns and alternative solutions

See PROJECT HEADER [Pont, 2001, p.169].

Example: LED bargraph display

[Example omitted here.]

Further reading

-

References

Key, S.A., Pont, M.J. and Edwards, S. (2003) "Implementing low-cost TTCS systems using assembly language". Paper presented at EuroPLoP 2003 (Germany, June 2003)

Mwelwa, C. and Pont, M.J. (2003) "Towards a CASE tool to support the development of reliable embedded systems using design patterns", paper presented at the workshop *"Quality of Service in Component-Based Software Engineering"*, June 20th, 2003, Toulouse, France.

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

Pont, M.J. (2003) "Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns", *Informatica*, **27**: 81-88.

Pont, M.J. and Banner, M.P. (in press) "Designing embedded systems using patterns: A case study", to appear in: *Journal of Systems and Software*.

Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", in Hruby, P. and Soressen, K. E. [Eds.]*Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002*, pp.159-200

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T. (2003) "Prototyping time-triggered embedded systems using PC hardware". Paper presented at EuroPLoP 2003 (Germany, June 2003)