Prototyping time-triggered embedded systems using PC hardware

Michael J. Pont[1], Andrew J. Norman, Chisanga Mwelwa and Tim Edwards

Embedded Systems Laboratory[2], University of Leicester, LEICESTER LE1 7RH, UK.

Introduction

We have previously described a "language" consisting of nearly eighty patterns (see Appendix 1). These patterns are aimed primarily at applications based on small microcontrollers, particularly those from the ubiquitous 8051 family.

In this paper, we present two new patterns (see Table 1) which are intended to demonstrate how the time-triggered techniques described in our earlier studies can be used effectively in embedded systems based on PC hardware. We see these patterns 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.

Pattern	Problem addressed
TTCO SCHEDULER (DOS)	How can you create a co-operative scheduler for DOS-based PC hardware?
PORT WRAPPER (PC)	How can you make use of the PC's parallel port to interact with the outside world in your embedded design?

Table 1: *The patterns introduced in this paper.*

Compiler used

The code presented in this article was all compiled and tested using the "Open Watcom" compiler. This compiler is available (for free download) here: `www.openwatcom.org`

Acknowledgements

Copyright

---

[1] To whom correspondence should be addressed: `M.Pont _at_ le.ac.uk`

[2] `http://www.le.ac.uk/eg/embedded/`

TTCo Scheduler (DOS)

Context

- You are developing a complex embedded application.

- You are programming in C (or a similar language).

- Your application has a time-triggered (software) architecture, based on a co-operative scheduler (e.g. CO-OPERATIVE SCHEDULER [Pont, 2001, page 255]).

- You are using a desktop PC for system prototyping.

Or

- You are involved in providing training in a company setting (or teaching in a university / college environment), and need a cost-effective platform with which you can introduce key techniques used in the development of embedded software.

Or

- You are a hobby developer who wishes to gain experience with embedded software.

Problem

How can you create a co-operative scheduler for DOS-based PC hardware?

Background

This pattern is concerned with the creation of a scheduler for a desktop PC, in order to allow prototyping of embedded software.

What is a scheduler?

We have previously sought to demonstrate (Pont, 2001) that time-triggered software architectures, implemented using a co-operative scheduler, form a simple and highly predictable platform that serves the needs of a wide range of embedded designs. There are two ways of viewing such a scheduler:

- At one level, a scheduler can be viewed as a simple operating system that allows tasks to be called periodically, or (less commonly) on a one-shot basis.

- At a lower level, a scheduler can be viewed as a single timer interrupt service routine that is shared between many different tasks. As a result, only one timer needs to be initialised, and any changes to the timing requires only one function to be altered. Furthermore, we can generally use the same scheduler whether we need to execute 1, 10 or 100 different tasks.

We have described, in detail, how to create such a scheduler for use in a microcontroller-based embedded system (see CO-OPERATIVE SCHEDULER [Pont, 2001, p.255]). Some familiarity with this previous pattern will be helpful when considering the rest of the material presented here.

Why build a PC prototype?

Most embedded systems are implemented using specialised processors (such as the 8051, PIC, HC08, C167, ARM, etc). For example, in our previous patterns, we have focused on the popular 8051 family. 8051 microcontrollers - like most embedded processors - require a minimum of external components in order to operate. Prices for 8051 devices start at less than $1.00 (US). At this price, you get a performance of around 1 million instructions per second, and 256 **bytes** of on-chip RAM. The 8051's profile (price, performance, available memory) matches the needs of many embedded systems very well.

Although the cost of embedded hardware is very low, the cost of the required development tools is rather higher. For example, most embedded code - for whatever target processor - is cross-compiled (or assembled) on a PC. Unfortunately, such cross-compilers are expensive (around $2000.00 per seat is not uncommon; $10,000 per seat is not unheard of), and there are regular "maintenance" charges to be paid.

One way of reducing the number of cross-compilers required in an organisation is to use the PC hardware (in place of your target hardware) to develop an initial prototype of your system[3]. As we will demonstrate later in this pattern, it is possible to use the PC in this way to develop software that will have an architecture virtually identical to the one used on a standard embedded board. As a result, it is possible for large teams to carry out the initial development of a system using very low-cost tools. This prototype code can then be converted (by a smaller team, using a reduced number of expensive tools) into a form suitable for use on your chosen embedded platform.

If you decide to begin development of an embedded system in this way, the PC hardware is powerful, and provides easy access to large amounts of memory. This can make it easy to rapidly test and compare different design solutions. The screen and disk - not generally available on embedded boards - can also help with debugging. In addition, if your team members develop a PC prototype and they want to test out designs on a "black box" platform (without screen, keyboard, mouse, etc), you probably have access to older PCs that can be used for this purpose.

We also note that, even when not used for system prototyping, a desktop PC can allow developers to experiment with realistic software architectures for embedded systems without having to invest in expensive cross-compilers or hardware. This is of value not just for company training programmes, but also in many university and college environments (where funds are often limited). Working in this way, large classes can gain experience with the different software design techniques required in embedded systems while using ordinary desktop PCs.

---

[3]   To do this you will, of course, need a compiler for the PC platform. However, there are many high-quality compilers available for this platform available for very low cost (often for no cost).

In a similar way, the "hobby" developer can also benefit. If you want to experiment with a robot that will mow your grass, or a home security system, an old PC can be an excellent hardware platform.

Why build a scheduler "on top of DOS"?

When working with embedded systems, particularly safety-critical or safety-related applications where we need to be able to completely predict the system behaviour, we prefer to program for "naked" hardware. Specifically, the only "operating system" we use is a co-operative scheduler (our own, or an alternative for which we have access to all source code).

When developing **prototypes** using PC hardware, we follow a slightly different approach. Because of the complexity of the PC platform, the cost of developing our own OS (and BIOS) from scratch would be considerable. In addition, one reason for using the PC platform is to gain access to features such as hard disks or graphics screens. Low-cost desktop compilers make it very easy to use such facilities: however, such tools assume the presence of an operating system and / or standard BIOS on the PC. If we build a completely new OS, we may also need a completely new set of development tools. In the process, we will undermine some of the key reasons for using the PC platform in the first place.

The alternative is to find a very simple OS that is readily available, and widely supported by existing tools. We can then construct our embedded architecture on this foundation (Figure 1). The simplest operating system that is available for the PC is a form of "DOS" (Disk Operating System). The original Microsoft DOS (MS-DOS 1.0) was released in 1981, along with the original IBM 5150 PC. The latest Microsoft version (MS-DOS 6.22) was released in 1994. More recent versions of DOS are also available[4]. All versions have a command-line (rather than graphical) user interface.
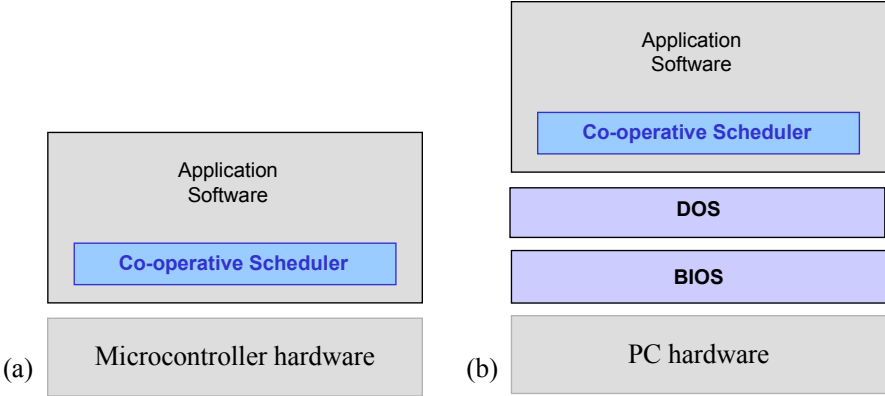

Figure 1: *The difference between the scheduler environment on target hardware and the environment used here for prototyping on a PC. See text for details.*

---

[4]   Of particular interest is the FreeDOS project, which is a complete - open source - version of DOS. For further details please see: `http://www.freedos.org/`

Building a scheduler "on top of DOS" is straightforward because, unlike more complex operating systems, DOS allows user programs to gain direct access to the underlying PC timers[5]. We describe how to achieve this in the "Solution" section.

Solution

When developing a scheduler for a DOS-based PC platform, the main thing we need to identify is a source of timer ticks.

In the PC architecture, the system timer runs at 1.193 MHz. From this, DOS uses a system tick of approximately 55 ms. This system tick arises as follows:

- An external (crystal) oscillator provides a square wave of frequency 1.19318 MHz.

- This square wave is fed into the Counter 0 channel of an 8254[6] programmable interval timer (PIT).

- The PIT (Channel 0) is loaded with a "divider" value of 2^{16}, which means that it generates an output square wave of 18.206 Hz (= 1193180 / 65536).

- The 18.206 Hz square wave is fed, via an 8259A programmable interrupt controller (PIC), into the IR0 interrupt pin on the 80x86 (or equivalent) processor at the heart of the PC.

- The IR0 input gives rise to an interrupt number 0x08. This in turn (explicitly) calls the 0x1C "user" interrupt.

This process may sound rather complicated (and somewhat circuitous), but the key implications are as follows:

- The programmer can connect the 0x1C interrupt source to an interrupt service routine which will be called periodically, at a rate of 18.206 Hz (approximately every 55 ms).

- The programmer can adjust the divider settings in the PIT, to give a higher tick rate (e.g. 1000Hz). Note that, if the timing is adjusted, the programmer should "chain" the DOS interrupt (that is, call it every 55 ms), to retain normal DOS behaviour in the system.

We illustrate how to achieve these aims in the example below.

Reliability and safety implications

We must issue the following warning about this pattern:

This pattern is intended for use only in system prototyping on a desktop PC. **IT IS NOT INTENDED FOR USE IN ANY APPLICATIONS WHICH ARE SAFETY-RELATED OR SAFETY-CRITICAL.**

---

[5] We briefly consider some alternative OS platforms (such as Linux) in the section "Related patterns and alternative solutions".

[6] An 8253 PIT is used in some older PCs. The behaviour is, as far as we are concerned, the same.

There are two main reasons for issuing such a warning:

- As noted in "Solution" (and illustrated in Figure 1), the scheduler described in this pattern is built "on top of DOS". Because of the complexity of this OS / BIOS combination, and the fact that not all of the behaviour can be completed predicted, this is not an appropriate platform for any safety-related or safety-critical design.

- The desktop PC was - of course - originally designed for use in an office environment. System components are, where possible, designed to be easily removed, for upgrade or repair. There are large numbers of components, and large numbers of connectors. If we subject such a device to even moderate vibration, to high or low temperatures, or to high humidity, we should not be surprised to experience reliability problems.

Hardware resource implications

This pattern is intended to support system prototyping. It does not - generally - have resource implications for the final embedded design. However, it may allow you to determine the likely resource implications in advance. This can, for example, mean that you use the desktop prototype to determine (a) likely CPU performance requirements, and (b) likely memory requirements. This may, in turn, allow you to select an appropriate embedded processor.

That being said, it should be noted that the use of DOS in this pattern has resource implications: in particular, it has an impact on the available RAM in your system. There are two reasons for this:

1. DOS itself will - of course - require some RAM. For example, the latest version of MS-DOS (6.22) will require around 64k or RAM.

2. Secondly - and more importantly in modern designs - DOS operates in the processor's "real" mode. As a result, by default, your application size will be restricted to a maximum of approximately 640k. Most embedded systems require much less than 640k of memory, and for such systems this does not represent a practical limitation when prototyping.

   If you wish to use more of the available memory on your PC, you will need a form of "DOS extender". We do not cover this issue here: please start by investigating "DPMI" (the DOS Protected Mode Interface) if you wish to explore this issue further.

   If your application requires memory that is around the 640k limit, you need to bear in mind that, on a PC, code is not "executed place" (as happens in most embedded environments). Instead, the application code is copied from disk (or other storage) and executed in RAM. This means that both your executable code and data required during the program run must fit into the available memory.

Portability

The co-operative scheduler architecture used here is highly portable, and can be used on a range of different processor platforms.

Related patterns and alternative solutions

Related patterns

Please refer to CO-OPERATIVE SCHEDULER [Pont, 2001, p.255] and TTCO SCHEDULER (ASSEMBLY LANGUAGE) [Key *et al.*, this conference] for detailed discussions about alternative scheduler designs. Please refer to PORT WRAPPER (PC) [this paper] for further information about the use of PC platforms for prototyping.

Using Linux

DOS is a widely-available OS for PCs. Another low-cost alternative is Linux.

Compared with DOS, the main drawback with Linux (from our perspective) is that gaining access to the underlying system "tick" is not as straightforward, and requires the creation and use of an appropriate device driver. Creation of such a driver is well beyond the scope of this publication: further information can be found in Rubini and Corbet (2001).

Other PC platforms

We have assumed in this pattern that you will be using a desktop PC to prototype embedded systems. Other versions of the PC platform are also available, including various forms of "embedded PC" with limited numbers of components. Such boards are, generally, much more robust than desktop (or notebook) PCs, and are much more compact. This can allow the use of very complete prototypes. In addition, while we do not recommend use of the techniques presented here for use in any form of safety-related or safety-critical design[7], use of a DOS-based scheduler on embedded PC hardware can be an effective solution for low-volume applications that are not in any way safety related.

Overall strengths and weaknesses

☺   This pattern describes a scheduler environment which is essentially the same as that employed on most embedded hardware: this allows effective prototyping of system software on a desktop PC, prior to "porting" to a suitable embedded platform.

☹   There are some differences between the scheduler described here and that available on most embedded hardware. These differences include the lack of "idle" or "sleep" modes of operation, and the lack of a "watchdog" facility.

☹   The techniques described here are intended for use in system prototyping: the code will generally need to be "ported" for use on the final target hardware.

---

[7]   For the reasons discussed in "Reliability and safety implications".

Example: Design of an advanced CMFD application

We give an example of the type of application for which we have found PC prototyping useful in this section.

We recently developed a four-stage condition monitoring and fault diagnosis (CMFD) framework (Parikh *et al.*, 2003; Figure 2). This has been applied a range of different problems, including fault diagnosis in diesel engines and medical diagnosis.
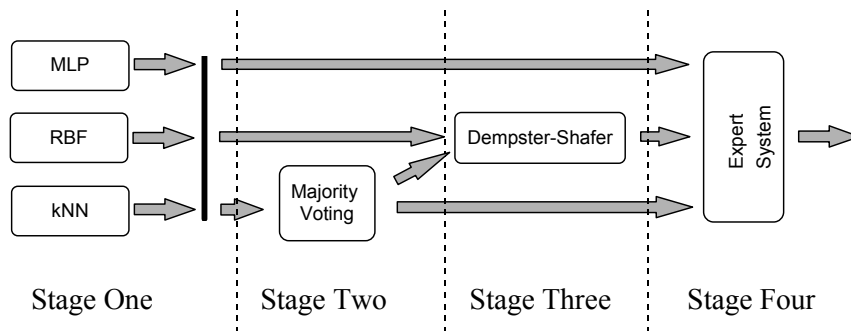
Figure 2: *A schematic representation of the four-stage classifier framework (described in detail in Parikh et al.,2003).*

In this case, system development was carried out on a desktop PC, making extensive use of MATLAB and Simulink, plus some in-house C programs. Subsequent development of the system was carried out also on a desktop PC: in this case, all of the system components were gradually converted into in-house C code. Only after this design was fully tested was the porting to a fully embedded platform carried out.

Example: Scheduler with 1 ms tick

This example describes a complete scheduler for use on the PC platform. The scheduler is a slightly simplified version of the design described in detail in CO-OPERATIVE SCHEDULER [Pont, 2001, p.255]. Please refer to the original pattern for a detailed description of the various scheduler functions.

As discussed in "Solution" (in this pattern), this implementation of the scheduler uses DOS interrupt 0x1C as the source of timer interrupts, and adjusts the system timing to generate these "ticks" at a 1 ms intervals. The original DOS ISR is chained (every 55 ms).

Please note that this scheduler simply displays some text on the PC screen: in PORT WRAPPER (PC) [this paper], the scheduler is used to control some port pins.

Please also note that, in keeping with the "prototype" nature of this system, the program will terminate if the user touches a key on the keyboard.

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

   Main.c (v1.00)

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

   Demonstration program for PC Scheduler.

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

#include "main.h"
#include "sch_dos.h"

#include "led_flas.h"

#include <conio.h>
#include <stdio.h>

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

int main(void)
   {
   tByte Abort = 0;

   // Set up the scheduler
   // Timings are in ticks (~1 ms tick interval)
   // (Max interval / delay is 65535 ticks)
   SCH_Init();

   // Prepare for the 'Flash_LED' task
   LED_Flash_Init();

   // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
   SCH_Add_Task(LED_Flash_Update, 0, 1000);

   // Start the scheduler
   SCH_Start();

   while (!Abort)
      {
      // Hitting any key will abort the scheduling
      Abort = SCH_Dispatch_Tasks();
      }

   // Stop the scheduler
   SCH_Stop();

   return 0;
   }

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

Listing 1: *Part of the PC scheduler implementation.*

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

   sch_dos.c (v1.00)

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

   *** THIS IS A SIMPLIFIED SCHEDULER FOR DOS ***

   *** 1 ms tick interval ***

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

#include "sch_dos.h"

#include "conio.h"
#include "stdio.h"

// ------ Private data type declarations ---------------------------

// Task data type
typedef struct
   {
   // Pointer to the task (must be a 'void (void)' function)
   void (__far * pTask)(void);

   // Delay (ticks) until the function will (next) be run
   // - see SCH_Add_Task() for further details
   tWord Delay;

   // Interval (ticks) between subsequent runs.
   // - see SCH_Add_Task() for further details
   tWord Period;

   // Incremented (by scheduler) when task is due to execute
   tByte RunMe;
   } sTask;


// ------ Public constants -----------------------------------------

// The maximum number of tasks required at any one time
// during the execution of the program
//
// MUST BE ADJUSTED FOR EACH NEW PROJECT
#define SCH_MAX_TASKS   (1)

// ------ Private variable declarations ----------------------------

// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];

// Pointer to original timer interrupt function
void (__interrupt __far *Orig_int_1c)();

// ------ Private function prototypes -------------------------------

void __interrupt __far SCH_Update(void);
```

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

   SCH_Init()

   Scheduler initialisation function.  Prepares scheduler
   data structures and sets up timer interrupts at required rate.

   You must call this function before using the scheduler.

-*------------------------------------------------------------------*/
void SCH_Init(void)
   {
   tByte i;

   for (i = 0; i < SCH_MAX_TASKS; i++)
      {
      SCH_tasks_G[i].pTask  = 0;
      }


   // Timer is already set up
   // Default is ~ 18 ticks per second (~55 ms tick interval)

   // Adjust to 1ms ticks
   outp(0x43,0x34);  // Control word
   outp(0x40,0xA9);  // Low byte
   outp(0x40,0x04);  // High byte
   }


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

   SCH_Start()

   Start the scheduler.

-*------------------------------------------------------------------*/
void SCH_Start(void)
   {
   // Store the information about the current timer interrupt handler
   Orig_int_1c = _dos_getvect(0x1c);

   // Link the PC timer to the scheduler update function
   _dos_setvect(0x1c, SCH_Update);
   }

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

   SCH_Update()

   This is the scheduler ISR.  It is called at a rate
   determined by the timer settings in the 'init' function.

-*------------------------------------------------------------------*/
void __interrupt __far SCH_Update(void)
   {
   tByte Index;
   static tByte Tick_count;

   // Assumes 1ms ticks
   if (++Tick_count == 55)
      {
      // Call the original DOS interrupt handler
      // [To maintain normal DOS behaviour]
      Orig_int_1c();
      }
```

```c
      // NOTE: calculations are in *TICKS* (not milliseconds)
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
         {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
               // The task is due to run
               SCH_tasks_G[Index].RunMe += 1;  // Inc. the 'RunMe' flag

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule regular tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               }
            }
         }
      }


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

   SCH_Stop()

   In this simplified scheduler, we have the ability to return to
   DOS.  When we do this, we need to restore the timer settings,
   and the ISR link.

-*-------------------------------------------------------------------*/
void SCH_Stop(void)
   {
   _dos_setvect(0x1c, Orig_int_1c);
   outp(0x43,0x34);  // Set to standard 55ms ticks again
   outp(0x40,0x00);  // Low byte
   outp(0x40,0x00);  // High byte
   printf("Finished\n");
   }

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

   SCH_Dispatch_Tasks()

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

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

   // Dispatches (runs) the next task (if one is ready)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      if (SCH_tasks_G[Index].RunMe > 0)
         {
         (*SCH_tasks_G[Index].pTask)();  // Run the task

         SCH_tasks_G[Index].RunMe -= 1;   // Reset / reduce RunMe flag

         // Periodic tasks will automatically run again
         // - if this is a 'one shot' task, remove it from the array
         if (SCH_tasks_G[Index].Period == 0)
            {
            SCH_tasks_G[Index].pTask = 0;
            }
         }
      }
```

```
      // Allow user to abort by pressing a key ...
      if (kbhit())
         {
         return 1;
         }

      return 0;
      }

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

   SCH_Add_Task()

   Causes a task (function) to be executed at regular intervals
   or after a user-defined delay

   Fn_P   - The name of the function which is to be scheduled.

   DELAY  - The interval (TICKS) before the task is first executed

   PERIOD - If 'PERIOD' is 0, the function is only called once,
            at the time determined by 'DELAY'.  If PERIOD is non-zero,
            then the function is called repeatedly at an interval
            determined by the value of PERIOD (see below for examples
            which should help clarify this).

   RETURN VALUE: None (no way of deleting tasks in this version...)

-*-------------------------------------------------------------------*/
void SCH_Add_Task(void (__far * pFunction)(void),
                  const tWord DELAY,
                  const tWord PERIOD)
   {
   tByte Index = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
      {
      Index++;
      }

   // Have we reached the end of the list?
   if (Index == SCH_MAX_TASKS)
      {
      // Task list is full - in this version, we simply abort...
      return;
      }

   // If we're here, there is a space in the task array
   SCH_tasks_G[Index].pTask  = pFunction;

   SCH_tasks_G[Index].Delay  = DELAY + 1;
   SCH_tasks_G[Index].Period = PERIOD;

   SCH_tasks_G[Index].RunMe  = 0;
   }


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

Listing 2: *Part of the PC scheduler implementation.*

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

   LED_flas.C (v1.00)

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

   Simple 'Flash LED' test function for scheduler.

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

#include "Main.h"
#include "Port.h"
#include "LED_flas.h"

#include <stdio.h>
#include <conio.h>

#define LED_OFF 1
#define LED_ON  0

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

static tByte LED_state_G;

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

  LED_Flash_Init()

  - See below.

-*-------------------------------------------------------------------*/
void LED_Flash_Init(void)
   {
   LED_state_G = 0;
   }


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

  LED_Flash_Update()

  Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

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

-*-------------------------------------------------------------------*/
void LED_Flash_Update(void)
   {
   // Change the LED from OFF to ON (or vice versa)
   if (LED_state_G == 1)
      {
      LED_state_G = 0;
      printf("+ LED is OFF\n");  // Illustrative purposes only ...
      }
   else
      {
      LED_state_G = 1;
      printf("  LED is ON\n");   // Illustrative purposes only ...
      }
   }

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

Listing 3: *Part of the PC scheduler implementation.*
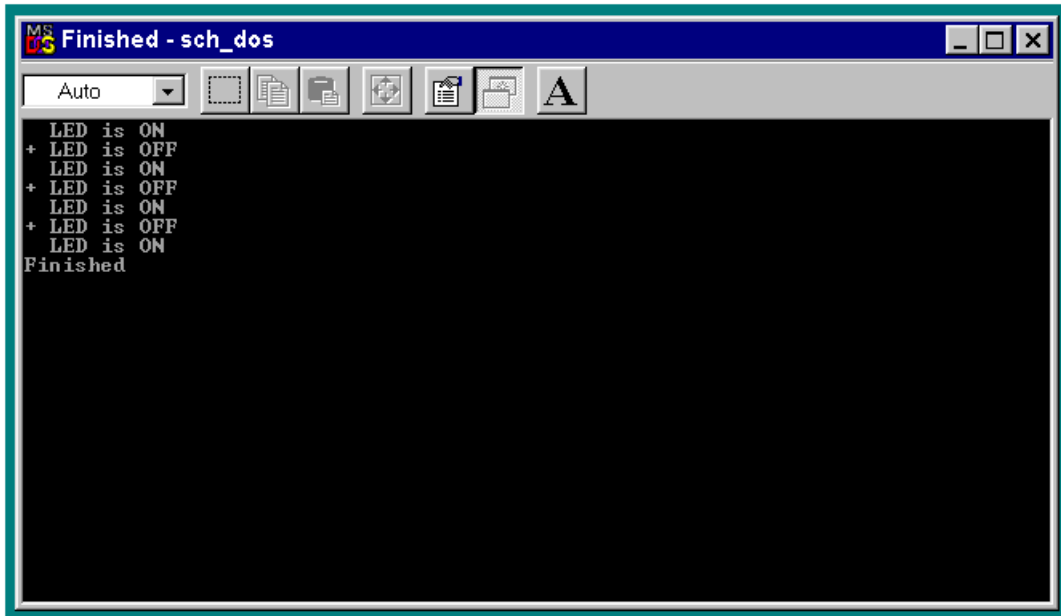
Figure 3: *Output from the PC scheduler system. Note that the scheduling is terminated after a few seconds, when the user presses a key on the keyboard.*

PORT WRAPPER (PC)

Context

- You have implemented TTCO SCHEDULER (DOS) [this paper].

Problem

How can you make use of the PC's parallel port to interact with the outside world in your prototype embedded system?

Background

Since 1981, most PCs have had a 25-pin D-type connector on the rear panel (Table 2). This provides a connection to the "parallel port" (or "printer port") and is usually used to connect to a printer, using a Centronics protocol.
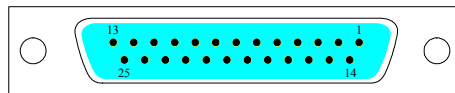
Pin	Label	Input / Output Pin	Pin	Label	Input / Output Pin
1	/Strobe	Input / Output	10	/Ack	Input
2	Data 0	Input / Output	11	Busy	Input
3	Data 1	Input / Output	12	Paper	Input
4	Data 2	Input / Output	13	Online	Input
5	Data 3	Input / Output	14	/AF	Input / Output
6	Data 4	Input / Output	15	/Error	Input
7	Data 5	Input / Output	16	Initialise	Input / Output
8	Data 6	Input / Output	17	/Select	Input / Output
9	Data 7	Input / Output	18-25	Ground	-

Table 2: *Pin connections for the PC's parallel port (diagram shows the 25-pin DB25 socket, viewed from the outside).*

In most (desktop) situations, the parallel port is used for printing. In this pattern, we use this flexible port for general-purpose I/O.

Solution

We consider how to read from and write to the various pins on the parallel port in this section.

Note that, in most (desktop) PCs, additional parallel ports can be added, if required (up to a maximum of four ports). We will assume the use of a single port in most of the examples: however, the techniques will work on PCs with up to four such ports.

We begin by considering the various registers that provide an interface to the parallel port.

Parallel port registers

We communicate with each the parallel port via a data register (Table 3), a status register (Table 4) and a control register (Table 5). Note that, with the exception of the data register, the link between registers and port pins is somewhat convoluted…

7	6	5	4	3	2	1	0	Bit
D7	D6	D5	D4	D3	D2	D1	D0	Signal
9	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*Pin*

Table 3: *The parallel port data register. This is a bi-directional register. Offset is 0x00 (see section "Parallel port addresses" for an explanation of this offset figure)*

7	6	5	4	3	2	1	0	Bit
/Busy	/Ack	Paper	Online	/Error	X	X	X	Signal
11	*10*	*12*	*13*	*15*	*-*	*-*	*-*	*Pin*

Table 4: *The parallel port status register. This is a read-only register. Offset is 0x01. X = unused (usually have value 1).*

7	6	5	4	3	2	1	0	Bit
X	X	X	IRQ	/Select	Init	/AF	/Strobe	Signal
-	*-*	*-*	*-*	*17*	*16*	*14*	*1*	*Pin*

Table 5: *The parallel port control register. This is a bi-directional register. Offset is 0x02. X = unused (usually have value 1). Note that the IRQ pin is usually set by a jumper or DIP switch on the interface card (this pin is not used in any of the examples in this pattern).*

Parallel port pin addresses

Most PCs can have up to four printer ports connected: these are referred to as LPT1 - LPT4. In most cases, only one port (LPT1) is available.

To connect to these ports, we need to know the addresses that the BIOS has assigned to the port(s). These addresses are assigned when the PC is booted.

In early PCs (where the parallel interface was located on the mono graphics card), the starting address (for a single port) was typically 0x03BC. In more recent PCs, the address of a single port is usually 0x0378.

To locate the port assignments for you hardware, you can use the DOS "debug" program to display memory locations 0040:0008. Here is a typical result:

```
c:\> debug
-d 0040:0008 L8
0040:0008     78 03 78 02 00 00 00 00
```

This result gives the addresses of all the ports (LPT1, LPT2, LPT3, LPT4). In this case, only the first two ports are installed: these are at addresses 0x0378 and 0x0278, respectively (note the swapped byte order).

More specifically, the above addresses refer to the data registers for these ports. As discussed in the previous section, each port is controlled via a data, status and control register. The addresses of the associated status register is found by adding 1 to the data-register address; the address of the control register is found by adding 2 to the data-register address.

Reading and setting pins on the data port

The data port (pins 2 - 9) are bi-directional and very easy to use.

We present here a small code library for reading and setting pins on this part of the parallel port.

Note that "positive logic" applies: if you write a 1 to a data-port register bit, the corresponding pin output will be +5V; if you write a 0 to the register bit, the corresponding output will be 0V.

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

   port_dat.c (v1.00)

   Simple library providing I/O capabilities
   via the "data" section of the PC parallel port.

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

#include "main.h"
#include "port_dat.h"

#include <conio.h>

// Setting for LPT1:
#define LPT1_DATA      0x0378

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

static tByte Last_output_G;


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

void PORT_DATA_Init(void)
   {
   // Set all data-port pins to 0
   outp(LPT1_DATA, 0);

   // Store last output
   Last_output_G = 0;
   }
```

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

void PORT_DATA_Write_Byte(const tByte VALUE)
    {
    // Write the new value
    outp(LPT1_DATA, VALUE);

    // Store last output
    Last_data_output_G = VALUE;
    }

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

void PORT_DATA_Write_Bit(const tByte PIN, const tByte VALUE)
    {
    tByte p = 0x01;
    tByte New;

    // Set the appropriate bit
    p <<= PIN;

    if (VALUE == 1)        // If we require a 1 at the pin
        {
        New = Last_data_output_G | p;
        }
    else                   // If we require a 0 at the pin
        {
        p = ~p; // Complement
        New = Last_data_output_G & p;
        }

    // Write the new value
    outp(LPT1_DATA, New);

    // Store last output
    Last_data_output_G = New;
    }

/*-------------------------------------------------------------------*/
tByte PORT_DATA_Read_Byte(void)
    {
    tByte Raw_input;

    // Read the "raw" port value
    Raw_input = inp(LPT1_DATA);

    // Need to take into account the register values
    return Raw_input | Last_data_output_G;
    }

/*-------------------------------------------------------------------*/
tByte PORT_DATA_Read_Bit(const tByte PIN)
    {
    tByte p = 0x01;
    tByte Byte;

    // Read the port value
    Byte = PORT_DATA_Read_Byte();

    // Set the appropriate bit
    p <<= PIN;

    return (Byte | p);
    }

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

Listing 4: *Reading and setting pins on the "data" port (see text for details).*

Working with the rest of the parallel port (the StatCon virtual port)

As we noted earlier, the remaining (useful) pins on the parallel port are accessed via two registers: the control register and the status register. In total, there are nine pins available in this way.

We prefer to make use of eight of these pins and to treat these as a single "virtual input port", which we will call here the "StatCon" port (Table 6).

7	6	5	4	3	2	1	0	StatCon Bit
D7	D6	D5	D4	D3	D2	D1	D0	Signal
17	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*Port pin*
C3	C2	S3	C1	S4	S5	S7	S6	C/S register bit
Y	*N*	*N*	*Y*	*N*	*N*	*Y*	*N*	*Inverted?*

Table 6: *The "StatCon" port. This is a virtual 8-bit input port, accessed through the status and control registers. The corresponding bits in the control and status registers are identified in the table.*
Note that some of these pins are inverted (in hardware), as indicated.

A code library for working with this port is presented in Listing 5. Please note that this library takes care of the fact that some of the signals from the status and control registers are inverted (in hardware).

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

   port_sc.c (v1.00)

   Simple library providing input capabilities
   via the "status" and "control" sections of the PC parallel port.

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

#include "main.h"
#include "port_sc.h"

#include <conio.h>

// Settings for LPT1:
#define LPT1_DATA 0x0378
#define LPT1_STAT 0x0379
#define LPT1_CONT 0x0380

// Standard bit constants
#define  BIT0     0x01
#define  BIT1     0x02
#define  BIT2     0x04
#define  BIT3     0x08
#define  BIT4     0x10
#define  BIT5     0x20
#define  BIT6     0x40
#define  BIT7     0x80
```

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

tByte PORT_SC_Read_Byte(void)
   {
   tByte Status, Control;
   tByte Stat_con; // The "virtual port" value

   // Read the status register
   Status = inp(LPT1_STAT);

   // Read the control register
   Control = inp(LPT1_CONT);

   // Now start to assemble a single input byte ...

   // Stat_con bits 0 and 1 are from Status bit 6 and 7
   Stat_con = Status >> 6;

   // Stat_con bit 2 is from Status bit 5
   Stat_con |= ((Status & BIT5) >> 3);

   // Stat_con bit 3 is from Status bit 4
   Stat_con |= ((Status & BIT4) >> 1);

   // Stat_con bit 4 is from Control bit 1
   Stat_con |= ((Control & BIT1) << 3);

   // Stat_con bit 5 is from Status bit 3
   Stat_con |= ((Status & BIT3) << 2);

   // Stat_con bit 6 is from Control bit 2
   Stat_con |= ((Control & BIT2) << 4);

   // Stat_con bit 7 is from Control bit 3
   Stat_con |= ((Control & BIT3) << 4);

   // Finally, Stat_con bits 7, 4 and 1 must be inverted
   // (10010010b -> 0x92)
   return Stat_con ^ 0x92;
   }
/*-------------------------------------------------------------------*/
tByte PORT_SC_Read_Bit(const tByte PIN)
   {
   tByte p = 0x01;
   tByte Byte;

   // Read the port value
   Byte = PORT_SC_Read_Byte();

   // Set the appropriate bit
   p <<= PIN;

   return (Byte | p);
   }
/*------------------------------------------------------------------*-
  ---- END OF FILE ----------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 5: *Accessing the "StatCon" virtual port. See text for details.*

Reliability and safety implications

Please refer to TTCₒ Sᴄʜᴇᴅᴜʟᴇʀ (DOS) [this paper] for a discussion of the risks involved in working with the PC platform (and of working "over" DOS in particular).

As a result of these risks:

This pattern is intended for use only in system prototyping on a desktop PC. **IT IS NOT INTENDED FOR USE IN ANY APPLICATIONS WHICH ARE SAFETY-RELATED OR SAFETY-CRITICAL.**

Hardware resource implications
The main implication is that you cannot use the port for printing.

Portability
This pattern is highly portable and can be used with different compilers, and different PC platforms, without difficulty.

Related patterns and alternative solutions
See Pᴏʀᴛ I/O [Pont, 2001, p.174] for a discussion of port control in a microcontroller.

As far as any external devices are concerned, the parallel-port interface is the same as a microcontroller port. See IC Bᴜꜰꜰᴇʀ [Pont, 2001, p.118], BJT Dʀɪᴠᴇʀ [Pont, 2001, p.124], IC Dʀɪᴠᴇʀ [Pont, 2001, p.134], MOSFET Dʀɪᴠᴇʀ [Pont, 2001, p.139], SSR Dʀɪᴠᴇʀ (DC) [Pont, 2001, p.144], EMR Dʀɪᴠᴇʀ [Pont, 2001, p.149] and SSR Dʀɪᴠᴇʀ (AC) [Pont, 2001, p.156] for information about control of DC and AC loads using port pins.

Overall strengths and weaknesses
☺   A simple way of reading or controlling port pins in a design based on an embedded PC.

☺   Additional parallel ports can be added to expand capabilities, if required.

☹   Even a basic 8051 microcontroller has 32 port pins available. Compared with this, there are a limited number of pins are available on each parallel port.

Example: Scheduler task for controlling a "heartbeat" LED

A simple task for controlling a flashing LED is shown in Listing 6. The LED should be connected to Pin 0 on the data port (Pin 2 on the parallel port; see Table 2), as illustrated in Figure 4.
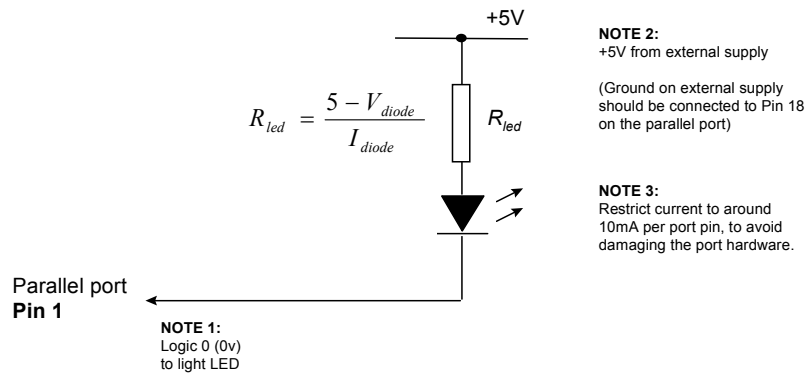

Figure 4: *Connecting an LED to the parallel port.*

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

   LED_flas.C (v1.00)

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

   Simple 'Flash LED' test function for scheduler
   with parallel-port control.

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

#include "main.h"
#include "led_flas.h"
#include "port_dat.h"

// For printf() [Demo purposes only]
#include <stdio.h>

#define LED_OFF 1
#define LED_ON  0

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

static tByte LED_state_G;

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

   LED_Flash_Init()

   - See below.

-*-------------------------------------------------------------------*/
void LED_Flash_Init(void)
   {
   LED_state_G = 0;

   // Set up the parallel port
   PORT_DATA_Init();

   // Set the LED pin (pin 0) to LED OFF
   PORT_DATA_Write_Bit(0, LED_OFF);
   }
```

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

  LED_Flash_Update()

  Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

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

-*-------------------------------------------------------------------*/
void LED_Flash_Update(void)
   {
   // Change the LED from OFF to ON (or vice versa)
   if (LED_state_G == 1)
      {
      LED_state_G = 0;

      // Set the LED pin (pin 0) to 1 (LED OFF)
      PORT_DATA_Write_Bit(0, LED_OFF);

      printf("+ LED is OFF\n");  // Illustrative purposes only ...
      }
   else
      {
      LED_state_G = 1;

      // Set the LED pin (pin 0) to 0 (LED ON)
      PORT_DATA_Write_Bit(0, LED_ON);

      printf("  LED is ON\n");   // Illustrative purposes only ...
      }
   }

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

Listing 6: *A task for controlling a "heartbeat" LED (see text for details).*

Appendix 1:
The PTTES Collection

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

The present version of this collection consists of 71 patterns from Pont (2001), plus a further 7 patterns from Pont and Ong (2003). Please note that the 2003 patterns were originally presented at the VikingPLoP conference (in 2002), and are identified thus [VP]. Please also note that the later patterns, <u>together</u>, form a replacement for HARDWARE WATCHDOG (presented in Pont, 2001): HARDWARE WATCHDOG is therefore not listed in this table.

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

Table 7: *The "PTTES Collection".*

References and further reading

Cooper, J. (2002) *"Using MS-DOS 6.22"*, Que.

Key, S.A., Pont, M.J. and Edwards, S. (this conference) "Implementing low-cost TTCS systems using assembly language", to appear in the proceedings of EuroPLoP 2003. [8]

Kopetz, H. (1997) *"Real-time systems: Design principles for distributed embedded applications"*, Kluwer Academic.

Messmer, H-P (2001) *"The indispensable PC hardware book"* (4th Edition). Addison-Wesley.

Parikh C.R., Pont, M.J., Jones, N.B. and Schlindwein, F.S. (2003) "Improving the performance of CMFD applications using multiple classifiers and a fusion framework", *Transactions of the Institute of Measurement and Control* , **25**(2): 123-144.

Pont, M.J. (2001) *"Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers"*, Addison-Wesley. [9]

Pont, M.J. (2002) *"Embedded C"*, Addison-Wesley. [10]

Pont, M.J. and Banner, M.P. (2004) "Designing embedded systems using patterns: A case study", *Journal of Systems and Software,* 71(3): 201-213. [11]

Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", Proceedings of VikingPLoP 2002 (Denmark, September 2002). [12]

Rubini, A. and Corbet, J. (2001) *"Linux Device Drivers"* (2nd Edition). O'Reilly UK.

---

[8]   A copy of this paper is available here: `http://www.le.ac.uk/eg/sak15/`

[9]   Further information about this book is available here: `http://www.engg.le.ac.uk/books/pont/pttes.htm`

[10]   Further information about this book is available here: `http://www.engg.le.ac.uk/books/pont/ec51.htm`

[11]   A copy of this paper is available here: `http://www.engg.le.ac.uk/books/pont/downloads.htm`

[12]   A copy of this paper is available here: `http://www.engg.le.ac.uk/books/pont/downloads.htm`