Towards a Pattern Language which Supports the Migration of Systems from Event-Triggered Pre-emptive (ETP) to Time-Triggered Co-operative (TTC) Software Architectures

Farah N. Lakhani[1], Anjali Das[2] and Michael J. Pont[1]

[1]*Embedded Systems Laboratory, University of Leicester,*
University Road, LEICESTER LE1 7RH, UK.

[2]*TTE Systems Ltd,*
106 New Walk, LEICESTER LE1 7EA,, UK.

`fnl1@le.ac.uk; a.das@tte-systems.com; M.Pont@le.ac.uk`

Abstract

We have previously described a “language” consisting of more than seventy patterns. This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered (TT) system architecture.

The present paper has a focus on techniques for converting an event-triggered (ET) system to an equivalent TT system.

The introduction to this paper describes the approach that we have taken to migrate from an ET to a TT architecture, and the motivation for making such a change. The core of the paper then goes on to describe some new patterns which represent the first parts of what is intended to be a small pattern collection.

Acknowledgements

We are grateful Bob Hanmer, to our Shepherd at EuroPLoP 2009, for his numerous helpful comments on this paper (and for his patience).

This work is supported by an HEC scholarship awarded to Farah N Lakhani from the Pakistan government. Additional support is provided by TTE Systems Ltd.

Copyright

Introduction

We have previously described a “language” consisting of more than seventy patterns, which will be referred to here as the “PTTES Collection” (see Pont, 2001). This language is intended to support the development of reliable embedded systems: the particular focus of the collection is on systems with a time triggered (TT) system architecture. Work began on these patterns in 1996, and they have since been used in a range of industrial systems, numerous university research projects (e.g. see Kurian and Pont, 2005; Phatrapornnant and Pont, 2006; Short and Pont, 2007; Bautista-Quintero and Pont, 2008; Gendy and Pont, 2008; Hughes and Pont, 2008) as well as in undergraduate and postgraduate teaching on many university courses.

Our focus throughout this work has been on systems with a “time-triggered architecture”, the main alternative to which is an “event triggered architecture”. We distinguish between these two alternatives in as shown in Figure 1 below:

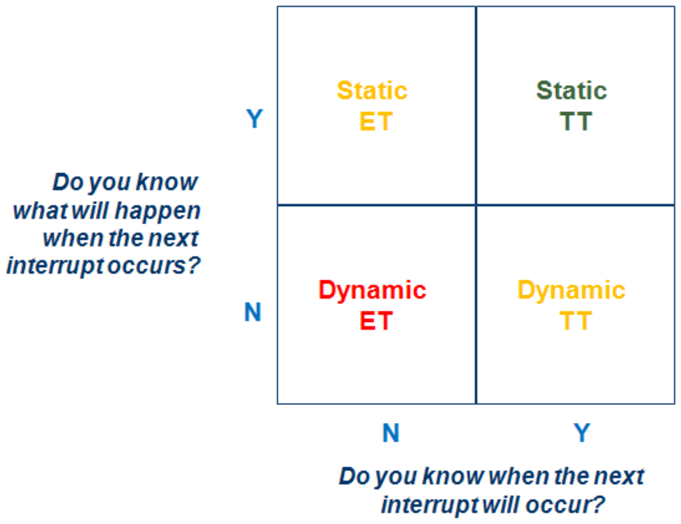
Static ET
Static TT
Dynamic ET
Dynamic TT
Y
N
Do you know what will happen when the next interrupt occurs?
N
Y
Do you know when the next interrupt will occur?

Figure 1: One way of distinguishing between time-triggered (TT) and event-triggered (ET) software architectures in a system design.

We assume that - in a static TT system - we always know (i) when the next interrupt will occur, and (ii) exactly what the system will do in response to this interrupt. At the other extreme, we have dynamic ET systems: in such designs we assume that (i) we never know when the next interrupt will occur, and (ii) that we do not know exactly what the system will do in response to this interrupt. In reality, many systems lie somewhere between these two extremes.

When and why should you migrate?

Our underlying assumption is that, in most cases, making a system “more TT” is likely to make it easier to predict how the system will behave and will – therefore – improve reliability.

If you'd like further information about the differences in behaviour between ET and TT systems, Short et al, 2008 describe, in detail, the results from a 4-year study which compared ET and TT architectures when used in a multi-processor automotive system. In summary, the results demonstrated that ET systems had a greater failure rate. For systems which are not safety related (for example, simple consumer products), we find that the greatest single benefit obtained through the use of TT architectures is a reduction in testing times.

Organisation of this paper

We have structured this paper in the form of a new (abstract) pattern (EVENTS TO TIME (TTC)), two new design patterns (BUFFERED OUTPUT and POLLED INPUT), plus two pattern implementation examples (RS 232 DATA TRANSFER and SWITCH INTERFACE).

Please note that EVENTS TO TIME (TTC) describes conversion to one possible TT architecture ("TTC") provides a single tasking system architecture. In the future we will explore conversions to a range of different TT architectures.

References

Bautista-Quintero, R. and Pont, M.J. (2008) "Implementation of H-infinity control algorithms for sensor-constrained mechatronic systems using low-cost microcontrollers", *IEEE Transactions on Industrial Informatics*, 16(4): 175-184.

Gendy, A.K. and Pont, M.J. (2008) "Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems", *IEEE Transactions on Industrial Informatics*, 4(1): 37-46.

Hughes, Z.M. and Pont, M.J. (2008) "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed", *Trans Institute of Measurement and Control*, 30(5): 427-450.

Kurian,S and Pont, M.J. (2005) "Building Reliable embedded systems using abstract patterns, patterns and pattern implementation examples", Proceedings of the second UK embedded forum (Birmingham UK 2005) pp. 36-59. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Phatrapornnant, T. and Pont, M.J. (2006) "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling", IEEE Transactions on Computers, 55(2): 113-124.

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.

Short, M.J and Pont, M.J. (2007) "Fault-tolerant time-triggered communication using CAN" IEEE Transactions on Industrial Informatics, 3(2):131-142

Short, M.J, Pont, M.J and Fang,Jiangzhong. (2008) "Assessment of performance and dependability in embedded control systems: methodology and case study" Control Engineering Practice, Vol. 16, pp. 1293–1307, July 2008

EVENTS TO TIME (TTC)

{abstract pattern}

Context

- You and / or your development team have programming or design experience with "event-triggered and / or pre-emptive" (ET/P) system architectures: that is, architectures which may involve use of conventional real-time operating system (RTOS) and / or multiple interrupt-service routines (linked to different interrupt sources) and / or task pre-emption.
- You are in the process of creating or upgrading an embedded system, based on a single processor.
- You already have at least a design or prototype for your system based on some form of ET/P architecture.
- Because predictable and highly-reliable system operation is a key design requirement, you have opted to employ a "time-triggered co-operative" (TTC) system architecture in your system, if this proves practical.

Problem

How can you convert event triggered / pre-emptive designs and code (and mindsets) to allow effective use of a TTC scheduler as the basis of your embedded system?

Background

If we were forced to sum up the difference between "embedded" and "desktop" systems in a single word we'd say "interrupts".

Event triggered behaviour in systems is usually achieved through the use of such interrupts. The system is designed to handle interrupts associated with a range of sources (e.g. switch inputs, CAN interface, RS-232, analogue inputs, etc). Each interrupt source will have an associated priority. Each interrupt source will also require the creation of a corresponding "interrupt service routine" (ISR): this can be viewed as a short task which is triggered "immediately" when the corresponding interrupt is generated.

Creating such (ET/P) systems is – on the surface at least – straightforward. However, challenges often begin to arise (in non-trivial designs) at the testing stage. It is generally impossible to determine what state the system will be in when any interrupt occurs, which makes comprehensive testing almost impossible.

A time-triggered system also requires an understanding of interrupts, but the operation is fundamentally different. Of particular concern in this pattern is a "time-triggered co-operative" (TTC) architecture which supports single- tasking system. In such systems only one task is active at any point in time: this task runs to completion and then return control to the scheduler. At the heart of a TTC system is a cooperative scheduler which determines when the tasks in the system will be executed: once the tasks start running they "run to completion": they cannot be pre-empted by another task. In such a system, there is only a single interrupt source (usually a periodic timer "tick"): this is used to drive the scheduler.

Solution

Here's what you need to do to migrate to a TTC design:

- You need to ensure that only a single – periodic - timer interrupt is enabled (all other interrupt sources will be converted to flags, which will be polled as required).
- You have to determine an appropriate "tick interval" for your system (that is, you need to determine how frequently the timer interrupts need to take place).
- You have to convert any ET ISRs into periodic tasks and add these to the schedule.
- You need to deal – cleanly – with any "long tasks" (that is, tasks which may have an execution time greater than your chosen tick interval.

To illustrate part of the translation process consider a simple ET system running two tasks, X and Y. These tasks are invoked by separate interrupts and implemented by associated ISRs. Table 1 illustrates one alternative TTC system, implemented using a standard TTC scheduler (Pont, 2001).

Events ⇨ **Time**

Events

```
void main(void)
   {
   X_init();
   Y_init();
   EA = 1 ; // Enable all interrupts

   while(1)
      {
      PCON |= 0x01;
      }
   }

void X_ISR(void) interrupt IEIndex
   {
   }

void Y_ISR(void) interrupt IEIndex
   {
   }
```

Time

```
void main(void)
   {
   SCH_Init(); // Set up the scheduler
   X_Init();
   Y_Init();

   // Add tasks to scheduler
   SCH_Add_Task(X_Update(), 0, 100);
   SCH_Add_Task(Y_Update(), 20, 200);

   // Start the scheduler
   SCH_Start();
   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }
```

Table 1: Converting a system from Event triggered pre-emptive to Time triggered Cooperative

Dealing with long tasks
To deal with the problem of long task, we need to find an elegant way of splitting up long tasks (which are called infrequently) into a series of much shorter tasks (called frequently). The pattern BUFFERED OUTPUT [later in this paper] describes a solution to this problem.

As "TTC" allows only single interrupt to be enabled in the system , we need to convert all other interrupt sources to flags , and we need to "poll" (that is, check periodically) to see if these flags have been set. The pattern POLLED INPUT [later in this paper] describes how we can achieve this.

Related patterns and alternative solutions

The PTTES collection

The PTTES collection (Pont, 2001) describes, in detail, a range of techniques which can be used to implement embedded systems with TTC architecture. This book can now be downloaded (free of charge) from the following WWW site:
http://www.tte-systems.com/books/pttes

TT Schedulers

The pattern TT SCHEDULER[*] provides relevant background information on tasks, basic scheduling concepts and the situations in which it may be appropriate to use a TTC scheduler in your application.

Reliability and safety implications

When compared to pre-emptive schedulers, co-operative schedulers have a number of desirable features, particularly for use in safety-related systems (Allworth, 1981; Ward, 1991; Nissanke, 1997; Bate, 2000).

For example, Nissanke (1997, p. 237) notes: "[Pre-emptive] schedules carry greater runtime overheads because of the need for context switching—storage and retrieval of partially computed results. [Co-operative] algorithms do not incur such overheads. Other advantages of [co-operative] algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for guaranteeing exclusive access to any shared resource or data".

Allworth (1981, pp. 53–54) also notes: "Significant advantages are obtained when using this [co-operative] technique. Since the processes are not interruptable, poor synchronisation does not give rise to the problem of shared data. Shared subroutines can be implemented without producing re-entrant code or implementing lock and unlock mechanisms".

[*] Wang, H., M. J. Pont, et al. (2007). Patterns which help to avoid conflicts over share resources in time-triggered embedded systems which employ a pre-emptive schedule the 12th European Conference on Pattern Languages of Programs (EuoPLoP 2007),Irsee, Germany

Although not the main focus of this pattern, the advantages of a TT(C) approach also apply in distributed systems: see, for example, Scarlett and Brennan (2006).

Overall strengths and weaknesses

☺ Use of TTC architecture tends to result in a system with highly predictable patterns of behaviour.

☹ Inappropriate system design using this approach can result in applications which have a comparatively slow response to external events.

Examples

In applications where systems have been designed using event-triggered architecture it is occasionally later found to be necessary for higher reliability to migrate to time-triggered architectures. One such example is found in a recently published article (Turley, 2009). The article has mentioned the changes in the architecture design Sony Electronics have done to improve the performance. It states

"Given high packet rates Sony was hoping for, frequent interrupts turned from being a necessity to being a problem. In their experience, most network stacks are interrupt driven, especially from the hardware interface when it needs servicing. As data rates climb, these interrupts (and their attendant context switching) become so frequent that the overhead overwhelms the actual task. To fix this, the team has decided to switch from an interrupt driven to a software polled design. The resulting efficiency was dramatic".

This pattern can be applied to a variety of systems from drive by x-wire systems to simple control systems where performance and reliability is an important issue.

BUFFERED OUTPUT

{design pattern}

Context

- You are applying the pattern EVENTS TO TIME (TTC)
- You need to deal – cleanly – with a "long task" (that is, a task which may have an execution time greater than your chosen tick interval.
- You need to send a significant amount of data between your processor / system and an external device: the data transfer process will take some time.

Problem

How can you structure the data-transfer tasks in your application in a manner which is compatible with TTC architecture?

Background

We illustrate the need for the present pattern with an example.

Suppose we wish to transfer data to a PC at a standard 9600 baud. Transmitting each byte of data, plus stop and start bits, involves the transmission of 10 bits of information (assuming a single stop bit is used). As a result, each byte takes approximately 1 ms to transmit.

Now, suppose we wish to send this information to the PC:

```
Current core temperature is 36.678 degrees
```

If we use a standard function (such as some form of `printf()`) the task sending these 42 characters will take more than 40 milliseconds to complete. In a system supporting task pre-emption, we may be able to treat this as a low-priority task and let it run as required. This approach is not without difficulties (for example, if a high-priority task requires access to the same communication interface while the low-priority task is running). However, with appropriate system design we will be able to make this operate correctly under most circumstances.

Now consider the equivalent TTC design. We can't support task pre-emption and a long data-transmission task (around 40 ms) is likely to cause significant problems. More specifically, if this time is greater than the system tick interval (often 1 ms, rarely greater than 10 ms) then this is likely to present a problem as shown in Figure 2. The RS-232 task is a "long task" has duration greater than the system tick and so is missing the next tick intervals.

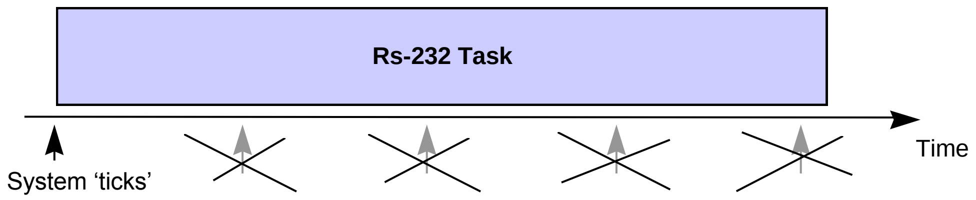

Figure 2: A schematic representation of the problems caused by sending a long character string on an embedded system. In this case, sending the massage takes 42 ms while the System tick interval is 10 ms.

Perhaps the most obvious way of addressing this issue is to increase the baud rate; however, this is not always possible, and - even with very high baud rates - long messages or irregular bursts of data can still cause difficulties.

More generally, the underlying problem here is that the data transfer operation has a duration which depends on the length of the string which we wish to submit. As such, the worst-case execution time (WCET) of the data transfer task is highly variable (and, in a general case, may vary depending on conditions at run time). In a TTC design, we need to know all WCET data for all tasks at design time. We require a different system design. As Gergeleit and Nett (2002) have noted " Nearly all known real-time scheduling approaches rely on the knowledge of WCETs for all tasks of the system." The known WCET of tasks will be helpful for developers in designing the offline schedule and preventing task overrun.

Solution

Convert a long data-transfer task (which is called infrequently and may have a variable duration) into a periodic task (which is called comparatively frequently and which has a very short – and known – duration).

A BUFFERED OUTPUT consists of three key components:

- A buffer (usually just an array, implemented in software)
- A function (or small set of functions) which can be used by the tasks in your system to write data to the array.
- A periodic (scheduled) task which checks the buffer and sends a block of data to the receiving device (when there are data to send).

Figure 3 below provides an overview of this system architecture. All data to be sent are first moved to a software buffer (a very fast operation). The data is then shifted – one block at a time – to the relevant hardware buffer in the microcontroller (e.g. 1 byte at a time for a UART, 8 bytes at a time for CAN, etc): this software-to-hardware transfer is carried out every 1ms (for example), using a (short) periodic task.

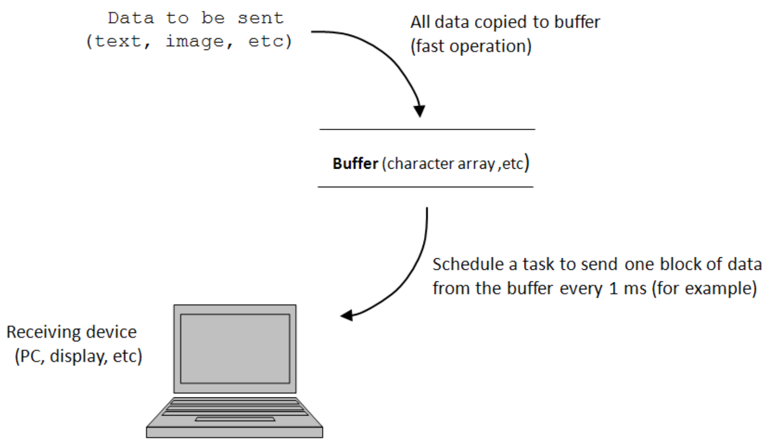
Data to be sent
(text, image, etc)
All data copied to buffer
(fast operation)
Buffer (character array,etc)
Schedule a task to send one block of data
from the buffer every 1 ms (for example)
Receiving device
(PC, display, etc)

Figure 3: An overview of the BUFFERED OUTPUT architecture.

From Figure 3, it should be noted that the long task "RS-232 task " in Figure 2 has been replaced by two short (high frequency) tasks, as shown in Figure 4:

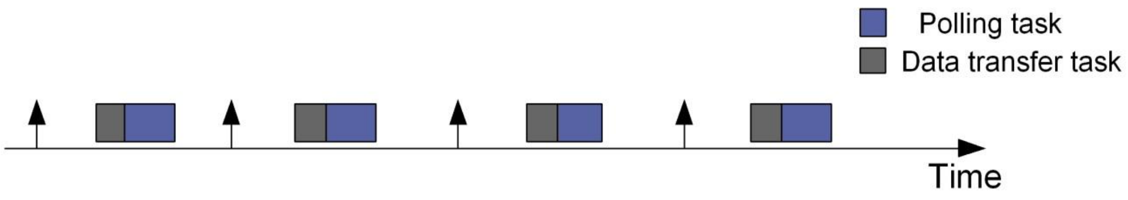

Figure 4: Long "RS-232" task is now transformed into two short tasks

Hardware resource implications

In most cases, the CPU requirements for BUFFERED OUTPUT are very limited, provided we take reasonable care at the design stage. For example, if we are sending message over a CAN bus and we know that each message takes approximately 0.15 ms to transmit; we should schedule the data transmission task to check the buffer at an interval > 0.15 ms. If we do this, the process of copying data from the software buffer to the (CAN) hardware will take very little time (usually a small fraction of a millisecond).

For very small designs (e.g. 8-bit systems) the memory requirements for the software buffer can prove significant. If you can't add external memory in these circumstances, you will need to use a small buffer and send data as frequently as possible (but see the comment above).

In some cases, hardware support can help to reduce both memory requirements and processor load. For example, if using UART-based data transmission, UARTs often have 16-byte hardware buffers: if you have these available, it makes sense to employ them.

Portability

This technique is generic and highly portable.

Reliability and Safety Issues

- Special care must be taken while defining buffer length, the data transfer should not cause any buffer overflow
- Applications that involve high amount of data transfer like video and DSP applications or data acquisition systems the use of buffer might not be a viable solution.

Overall strengths and weaknesses

☺ Use of buffered output is an easy solution for faster data transfer from a task running in an embedded application

☹ One has to be very careful while defining the buffer length, inappropriate buffer definitions may cause buffer overflow and data loss.

RS-232 DATA TRANSFER (BUFFERED OUTPUT)

{pattern implementation example}

Context

- You need to transmit data from your embedded processor using “RS-232”.
- You wish to transmit the data using a BUFFERED OUTPUT [this paper].
- Your chosen implementation language is C.
- Your chosen implementation platform is 8051 family of microcontrollers.

Problem

How can you implement a BUFFERED OUTPUT in C for the 8051 family of microcontrollers?

Background

RS232 is a standard asynchronous protocol used for serial communication between the processor and the peripherals; it is character oriented and is intended to be used with single 8-bit blocks of data. To transmit a byte of data over a serial link the data frame consists of a start bit to indicate start of transmission, the data itself(5 to 8 bits) and one or more stop bits to indicate the end of data block. RS232 can operate at different baud rates ranging from value of 75 to 330,000, however a baud rate of 9600 (a value lies in mid of the range) is recommended for safe use in our context.

Solution

Use of Buffered output can greatly reduce the time for data transfer task as compared to printf() function. The printf() function sends data immediately to UART. As a result the duration of the transmission is usually too long. The pattern buffered output makes use of an intermediate software buffer (character array) in between. Sending data to buffer is a fast operation. Our code library of RS232 data transfer as shown in Listing 1 replaces the printf() function. A brief functionality of each function is shown in Table 2:

Function	*Functionality*
PC_LINK_IO_Write_String_To_Buffer()	This function copies a null terminated string to the character buffer. The contents of the buffer then passed over a serial link.
PC_LINK_IO_Write_Char_To_Buffer()	Stores a character in the 'write' buffer, ready for later transmission.
PC_LINK_IO_Get_Char_From_Buffer()	Retrieves a character from the (software) buffer, if one is available.
PC_LINK_IO_Update()	Checks for characters in the UART (hardware) receive buffer and sends next character from the software transmit buffer.

Table 2: RS-232 code library functions descriptions

```
void PC_LINK_IO_Write_String_To_Buffer(const char* const STR_PTR)
      {
      tByte i = 0;

      while (STR_PTR[i] != '\0')
         {
         PC_LINK_IO_Write_Char_To_Buffer(STR_PTR[i]);
         i++;
         }
      }

   void PC_LINK_IO_Write_Char_To_Buffer(const char CHARACTER)
      {
      // Write to the buffer *only* if there is space
      if (Out_waiting_index_G < TRAN_BUFFER_LENGTH)
         {
         Tran_buffer[Out_waiting_index_G] = CHARACTER;
         Out_waiting_index_G++;
         }
      else
         {
         // Write buffer is full
         // Increase the size of TRAN_BUFFER_LENGTH
         // or increase the rate at which UART 'update' function is called
         // or reduce the amount of data sent to PC
         Error_code_G = ERROR_USART_WRITE_CHAR;
         }
      }

   char PC_LINK_IO_Get_Char_From_Buffer(void)
      {
      char Ch = PC_LINK_IO_NO_CHAR;

      // If there is new data in the buffer
      if (In_read_index_G < In_waiting_index_G)
         {
         Ch = Recv_buffer[In_read_index_G];

         if (In_read_index_G < RECV_BUFFER_LENGTH)
            {
            In_read_index_G++;
            }
         }

      return Ch;
      }

   void PC_LINK_IO_Update(void)
      {

      // Deal with transmit bytes here

      // Is there any data ready to send?
      if (Out_written_index_G < Out_waiting_index_G)
         {
         PC_LINK_IO_Send_Char(Tran_buffer[Out_written_index_G]);

         Out_written_index_G++;
         }
      else
         {
         // No data to send - just reset the buffer index
         Out_waiting_index_G = 0;
         Out_written_index_G = 0;
         }

      // Only dealing with received bytes here
      // -> Just check the RI flag
      if (RI == 1)
         {
         // Flag only set when a valid stop bit is received,
         // -> data ready to be read into the received buffer

         // Want to read into index 0, if old data has been read
         // (simple ~circular buffer)
         if (In_waiting_index_G == In_read_index_G)
            {
            In_waiting_index_G = 0;
            In_read_index_G = 0;
            }

         // Read the data from USART buffer
         Recv_buffer[In_waiting_index_G] = SBUF;

         if (In_waiting_index_G < RECV_BUFFER_LENGTH)
            {
            // Increment without overflowing buffer
            In_waiting_index_G++;
            }

         RI = 0;  // Clear RT flag
         }

      }

   void PC_LINK_IO_Send_Char(const char CHARACTER)
      {

      tLong Timeout1 = 0;
      tLong Timeout2 = 0;

      if (CHARACTER == '\n')
         {
         if (RI)
            {
            if (SBUF == XOFF)
               {
               Timeout2 = 0;
               do {
                  RI = 0;

                  // Wait for uart (with simple timeout)
                  Timeout1 = 0;
                  while ((++Timeout1) && (RI == 0));

                  if (Timeout1 == 0)
                     {
                     // USART did not respond - error
                     Error_code_G = ERROR_USART_TI;
                     return;
                     }

                  } while ((++Timeout2) && (SBUF != XON));

               if (Timeout2 == 0)
                  {
                  // uart did not respond - error
                  Error_code_G = ERROR_USART_TI;
                  return;
                  }
```

```
          RI = 0;
          }
       }

   Timeout1 = 0;
   while ((++Timeout1) && (TI == 0));

   if (Timeout1 == 0)
      {
      // uart did not respond - error
      Error_code_G = ERROR_USART_TI;
      return;
      }

   TI = 0;
   SBUF = 0x0d;  // output CR
   }

if (RI)
   {
   if (SBUF == XOFF)
      {
      Timeout2 = 0;

      do {
         RI = 0;

         // Wait for USART (with simple timeout)
         Timeout1 = 0;
         while ((++Timeout1) && (RI == 0));

         if (Timeout1 == 0)
            {
            // USART did not respond - error
            Error_code_G = ERROR_USART_TI;
            return;
            }

         } while ((++Timeout2) && (SBUF != XON));

      RI = 0;
      }
  }

Timeout1 = 0;
while ((++Timeout1) && (TI == 0));

if (Timeout1 == 0)
   {
   // USART did not respond - error
   Error_code_G = ERROR_USART_TI;
   return;
   }

TI = 0;

SBUF = CHARACTER;
}
```

Listing 1: RS232 Data transfer code Library using Buffered output

POLLED INPUT

{design pattern}

Context

- You are applying the pattern EVENTS TO TIME(TTC)
- You need to poll inputs from the available interfaces (switches, keypads, sensors, ADCs) etc

Problem

How do I build a TT system which is equivalent of my ET system such that it can respond to all (external/internal) input interfaces?

Background

Designing a TT system requires more planning efforts. In a "TTC" design the possible occurrence and the execution times of all the tasks needs to be known in advance. The designer has to plan a task schedule which must execute all the tasks periodically at their allocated time intervals. This effort makes the system more predictable. In contrast to this, in an event triggered system the schedule executes the tasks dynamically as the events arrive thus no guarantee that they meet any timeliness constraints. This is the reason that ET designs are not recommended for safety critical applications. The event triggered behaviour in systems is achieved through the use of interrupts. To support these interrupts, Interrupt Service Routines (ISRs) are provided. Whenever an interrupt occurs it stops the currently running task and ISR executes to respond to the interrupt. This "context switching" is an overhead that sometimes raised serious complications in systems.

The abstract pattern EVENTS TO TIME [this paper] provides more relevant background information.

Solution

A POLLED INPUT should meet the following specification:

- It should include a periodic task which polls for the occurrence of the event.
- The period of the above task should be set to some value less than or equal to minimum inter-arrival time[*] of the event in question.
- The interrupt associated with this event should not be enabled. In fact only one interrupt associated with the timer responsible for generating system "ticks" should be enabled.

[*] In ET systems the exact arrival time of events is not known so we assume a minimum distance between the arrivals of two consecutive events.

Hardware resource implications

Different interfaces have different implications under various circumstances. Reading a switch input imposes minimal loads on CPU and memory resources whereas scanning the keypad interface imposes both a CPU and memory load.

Reliability and safety issues

One major concern here in migrating from event triggered to time triggered is to make systems more predictable. Characteristic for the time triggered architecture is the treatment of (physical) real time as a first order quantity (Kopetz and Bauer 2002) this implies to the fact that time triggered systems must be very carefully designed, the task activation rates must be fixed according to the system dynamics i.e. how frequent an input needs to be polled.

Portability

This technique is generic and highly portable.

Overall strengths and weaknesses

☺ A flexible technique, programmer can easily do changes in code for example if auto repeat is required in case of SWITCH INTERFACE

☺ It is simple and cheap to implement.

☹ Provides no protection against out of range inputs or electrostatic discharge (ESD)

☹ More processor utilization in polling for tasks which are unlikely to occur, because the tasks are always tested for readiness whether actually enable or not.

SWITCH INTERFACE (POLLED INPUT)

{pattern implementation example}

Context

- You need to respond to a switch press from your embedded application
- Your chosen implementation language is C
- Your chosen implementation platform is NXP LPC2000 family of ARM7-based microcontrollers

Problem

How can you implement POLLED INPUT for a simple switch press in C for NXP LPC2000 family of microcontrollers?

Background

Simple push button switches as given in Figure 5 are very common in embedded applications. Pressing them causes a voltage change from Vcc to 0 volts at the input port. (For a detailed explanation see pattern SWITCH INTERFACE in PTTES (Pont 2001)

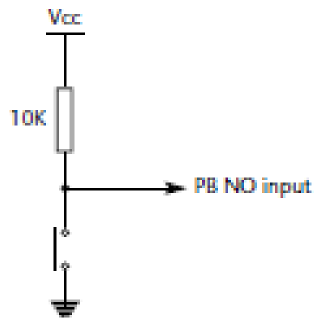

Figure 5: A simple push button switch with no internal pull-up

Note: There could be various types of switches (reset, on-off, multistate) for simplicity we are considering here only the simple interface push button switch with debounce support.

In an ideal world the change in voltage would take the form as shown in Figure 6:

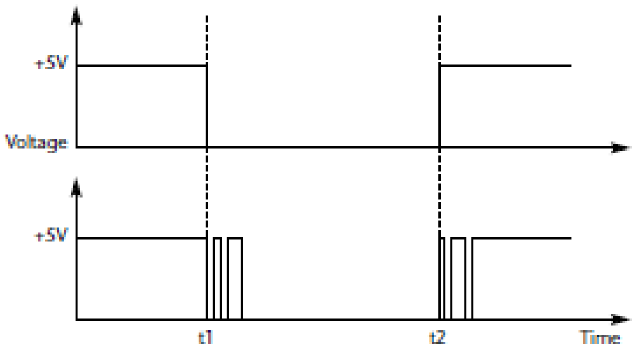

Figure 6: The voltage signal resulting from switch

In practical all mechanical switch contacts bounce (that is, turn on and off, repeatedly, for a short period of time) after the switch is closed or opened. As a result the actual waveform looks more like that shown in Figure 6 (bottom).

Solution

Polling a switch for an input involves the following steps:

1. A relevant port pin is read and if a switch depression is detected it will be read again after their debounce period (provided in datasheets) to confirm the detection.
2. If it is confirmed that switch is pressed a task is run to respond to the switch press.

A simple code example illustrating simple switch detection and debounce system is given in Listing 2. This example demonstrates a simple switch interface with debounce support. The interface is implemented as a task which periodically checks a switch pin to see if it is pressed or not.

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

   switch.c (v1.00)

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

   Simple switch detection and debounce system.

#include "main.h"
#include "switch.h"
#include "port.h"
#define PORTS (1)


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

static uint32_t gSwitchListeners[PORTS],gLastState[PORTS],gCurState[PORTS];

// ------ Private constants ---------------------------------------

//Allows NO or NC switch to be used (or other wiring variations)

#define SW_PRESSED (0)

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

  Switch_Pin_Read()

  Checks the read register bits locally to avoid excess overhead.

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

#define Switch_Pin_Read(l,m,n) \
   (((l) ? (IOPIN1) : (IOPIN0)) & n) ? (SW_PRESSED) : (1 - SW_PRESSED))


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

  Switch_Init()

  Initialising the switches.
```

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

void Switch_Init()
   {
   uint8_t port;

   for (port = 0; port < PORTS; port++)
      {
      gSwitchListeners[port] = 0;
      gLastState[port] = 0;
      gCurState[port] = 0;
      }

   //Initialise the output LED pin (LED_pin2) to indicate
   //switch is pressed
   PORT_Pin_GPIO_Set_Direction(LED_pin2, 1);
   PORT_Pin_Write(LED_pin2, 1);


   //Add Button_Pin to the switch listener
   Switch_AddListener(Button_Pin);
}
/*------------------------------------------------------------------*-

  Switch_AddListener()

  Adds a switch to the listen list for pins.

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

void Switch_AddListener(uint16_t pin)
   {
   // Get the port
   uint8_t
      port = (uint8_t)(pin / 100);
   if (port < PORTS)
      {
      pin %= 100;
      gSwitchListeners[port] |= 1 << pin;
      PORT_Pin_Set_Mode(pin, 0, 0);

      PORT_Pin_GPIO_Set_Direction(pin, 0);

      }
   }

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

  Switch_RemoveListener()

  Removes a switch from the listen list for pins. Does not remove the pin's mode.

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

void Switch_RemoveListener(uint16_t pin)
   {
   // Get the port
   uint8_t
      port = (uint8_t)(pin / 100), bit;
   if (port < PORTS)
      {
      pin %= 100;
      bit = ~(1 << pin);
      gSwitchListeners[port] &= bit;
      gLastState[port] &= bit;
      gCurState[port] &= bit;
      }
   }
*------------------------------------------------------------------*-

  Switch_Update()

Detects and debounces switch presses on hooked pins. Could be recorded VERY
efficiently with direct HW access as this already has the bits to check in the pin
registers.

-*------------------------------------------------------------------*/
void Switch_Update()
   {
   // Used for debugging with RapidiTTy
   T0TCR = 0;
   T0TCR = 1;

   uint8_t  port, pin;
   uint32_t temp, bit;
     for (port = 0; port < PORTS; port++)
      {
      //Check for switches on this port

      if ((temp = gSwitchListeners[port]))
         {
         pin = port * 100;
         bit = 1;
         do
            {
            if (temp & bit)
               {
               if (Switch_Pin_Read(port, pin, bit))
                  {
                  if (!(gCurState[port] & bit) && (gLastState[port] & bit))
                     {
                     // Was held before, debounced
                     // Call the callback
                     gCurState[port] |= bit;
                     Switch_Pressed(pin);
                     }
                  else
                     {
                     // Start debouncing
                     gLastState[port] |= bit;
                     }
                  }
               else
                  {
                  if ((gCurState[port] & bit) && !(gLastState[port] & bit))
                     {
                     // Was released before, debounced
                     // Call the callback
                     gCurState[port] &= ~bit;
                     Switch_Released(pin);
                     }
                  else
                     {
                     // Start debouncing
                     gLastState[port] &= ~bit;
                     }            }
               temp &= ~bit;
               }
            pin++;
            bit <<= 1;
            }
         while (temp);
         }
      }
   // Used for debugging with RapidiTTy
   T0TCR = 0;
   T0TCR = 1;
   }

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

  Switch_Pressed()

  Called when a switch is first pressed and debounced.
  Passes the full pin id of the activated input.

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

void Switch_Pressed(uint16_t pin)
   {
   if (pin == Button_Pin)
      {
      // Set to 0 to turn LED on
       PORT_Pin_Write(LED_pin2, 0);

      }
   }
/*------------------------------------------------------------------*-

  Switch_Released()

  Called when a switch is released. Passes the full pin id of the activated input.

-*------------------------------------------------------------------*/
void Switch_Released(uint16_t pin)
   {
   if (pin == Button_Pin)
      {
      // Set to 1 to turn LED off
       PORT_Pin_Write(LED_pin2, 1);

      }
   }
/*------------------------------------------------------------------*-

  Switch_IsPressed()

  Checks if a given pin is currently pressed and debounced.

-*------------------------------------------------------------------*/
uint8_t Switch_IsPressed(uint16_t pin)
   {
   uint8_t
      port = pin / 100;
   if (port > PORTS)
      {
      return 0;
      }
   return (gCurState[port] & (1 << (pin % 100))) ? (1) : (0);
   }
/*------------------------------------------------------------------*-
  ---- END OF FILE -------------------------------------------------
-*------------------------------------------------------------------*/
```

Listing 2: Simple switch press and debounce system using Polled Input

Further reading

Albert, A. and R. Bosch GmbH,(2004) “Comparison of Event-Triggered and Time-Triggered Concepts with regard to Distributed Control Systems”, in Embedded World. : Nurnberg. p. 235-252.

Allworth, S.T., 1981. An Introduction to Real-Time Software Design. , Macmillan, London.

Audsley, N., Tindell, K. and Burns, A. (1993), *"The end of the line for static cyclic scheduling?"* Proceedings of the 5th Euromicro Workshop on Real-time Systems, Finland, pp. 36-41.

Baker, T.P. and Shaw, A. (1989) “The cyclic executive model and Ada”, Real-Time Systems, 1(1): 7-25.

Bate, I.J. (1998) “Scheduling and timing analysis for safety critical real-time systems”, PhD thesis, University of York, UK.

Bate,I.J.(2000) “Introduction to scheduling and timing analysis”, in *“The Use of Ada in Real Time Systems”* (6 April, 2000). IEE Conference Publication 00/034

Bennett, S. (1994) *“Real-Time Computer Control”* (Second Edition) Prentice-Hall.

Buschmann, F., Henney, K. and Schmidt, D.C. (2007) “Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing” (Volume 4). Wiley. ISBN: 978-0-470-05902-9

Buttazzo, G. C. (2004), *"Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications",* 2nd ed, Springer.

Cottet, F. and David, L. (1999) “A solution to the time jitter removal in deadline based scheduling of real-time applications”, 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, pp. 33-38.

Edwards, T., Pont, M.J., Scotson, P. and Crumpler, S. (2004) “A test-bed for evaluating and comparing designs for embedded control systems”. In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.

Fohler, G. (1999) “Time Triggered vs. Event Triggered - Towards Predictably Flexible Real-Time Systems”, Keynote Address, Brazilian Workshop on Real-Time Systems, May 1999.

Gergeleit, M. and Nett, E. 2002: SchedulingTRANSIENT OVERLOAD with the TAFT Scheduler. GI/ITG specialized group of operating systems.

Hartwich F., Muller B., Fuhrer T., Hugel R., Bosh R. GmbH, (2002), Timing in the TTCAN Network, Proceedings 8th International CAN Conference.

Herzner, W., Kubinger, W. and Gruber, M. (2006) “Triple-T - A system of patterns for reliable communication in hard real-time systems”; in D. Manolescu, M. Völter, J. Noble (eds): Pattern Languages of Program Design 5 (PLOPD5); pp.89-126; Software Engineering/Patterns Series, Addison-Wesley, Boston. ISBN 0-321-32194-4

Hong, S.H. (1995) “Scheduling algorithm of data sampling times in the integrated communication and control systems”. IEEE Transactions on Control Systems Technology, 3(2): 225-230

Kalinsky, D., 2001. Context switch, Embedded Systems Programming, 14(1), 94-105.

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

Kopetz, H. and Bauer,G (2002) “The Time-Triggered Architecture”, IEEE Special Issue on Modeling and Design of Embedded Software pp.112-126.

Kurian, S. and Pont, M.J. (2007) “Maintenance and evolution of resource-constrained embedded systems created using design patterns”, *Journal of Systems and Software,* **80**(1): 32-41.

Liu, C. L. and Layland, J. W. (1973), “Scheduling algorithms for multi-programming in a hard real-time environment”, *Journal of the ACM,* **20**(1): 40-61.

Locke, C.D. (1992) “Software architecture for hard real-time systems: Cyclic executives vs. Fixed priority executives”, *The Journal of Real-Time Systems,* 4: 37-53.

Maaita, A. and Pont, M.J. (2005) “Using ‘planned pre-emption’ to reduce levels of task jitter in a time-triggered hybrid scheduler”. In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.18-35. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Massa, A.J. (2003) “Embedded Software Development with eCOS”, Prentice Hall. ISBN: 0-13-035473-2.

Nissanke, N., 1997. Realtime Systems. , Prentice-Hall.

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. (2002) “Embedded C”, Addison-Wesley. ISBN: 0-201-79523-X.

Pont, M.J. (2008) “Applying time-triggered architectures in reliable embedded systems: Challenges and solutions", *Elektrotechnik & Informationstechnik*

Shaw, A.C. (2001) “Real-time systems and software” John Wiley, New York. [ISBN 0-471-35490-2]

Scarlett, J.J. and Brennan,R.W(2006) “Re-evaluating Event-Triggerd and Time-Triggered Systems”, in *IEEE conference on Emerging technologies and factory automation*. p. 655-661.

Torngren, M. (1998) “Fundamentals of implementing real-time control applications in distributed computer systems”, Real-Time Systems, vol.14, pp.219-250.

Turley,J (2009) “Gaming the systems –high end networking on the cell processor” Embedded.com Issue September 2009.

Ward, N. J. (1991) “The static analysis of a safety-critical avionics control system”, in Corbyn, D.E. and Bray, N. P. (Eds.) “*Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, 1991”* Published by SaRS, Ltd.

Xu , J. (1993) “Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations,’’ IEEE Transactions on Software Engineering, 19(2), pp. 139-154.

Xu , J. and Parnas, D.L. (1990) “Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations,’’ IEEE Transactions on Software Engineering, 16(3), pp. 360-369.

Xu , J. and Parnas, D.L. (1993) “On Satisfying Timing Constraints in Hard-Real-Time Systems,” IEEE Transactions on Software Engineering, 19(1), pp. 70-84.

Xu , J. and Parnas, D.L. (2000) “Priority Scheduling Versus Pre-Run-Time Scheduling,’’ International Journal of Time-Critical Systems, 18, 7-23, Kluwer Academic Publishers.