Journal of Embedded Systems, 2014, Vol. 2, No. 3, 39-52
Available online at http://pubs.sciepub.com/jes/2/3/2
© Science and Education Publishing
DOI:10.12691/jes-2-3-2

Studying the Impact of Scheduler Implementation on Task Jitter in Real-Time Resource-Constrained Embedded Systems

Mouaaz Nahas*

Department of Electrical Engineering, College of Engineering and Islamic Architecture, Umm Al-Qura University, Makkah, KSA
*Corresponding author: mmnahas@uqu.edu.sa

Received March 15, 2014; Revised August 05, 2014; Accepted August 20, 2014

Abstract Over recent decades, many studies have considered the development, assessment and refinement of scheduling algorithms for use in real-time embedded applications. Various studies have also considered the impact of variations in the interval between the executions of periodic tasks (i.e. jitter) on the behaviour of such systems. Despite interest in both of these areas, there has been comparatively little attention paid to the impact of scheduler implementation techniques on jitter behaviour. This is unfortunate because – as we demonstrate in the course of this paper – there is a 'one-to-many' mapping between scheduler algorithms and scheduler implementations, and even comparatively small changes in the scheduler implementation can have a significant impact on jitter behaviour. Throughout this paper, our focus is on implementations of a form of "cyclic executive" which is one of the simplest scheduling algorithms in widespread use. The results presented demonstrate that – even for this very simple scheduling algorithm – implementation decisions can have a significant impact on both jitter behaviour and on resource requirements. We would expect that the results obtained would also apply to more complicated algorithms: indeed, as the algorithms grow more complicated, we would expect that the number of implementation options would increase, with a corresponding increase in the jitter variation.

Keywords: *real-time, scheduling algorithm, scheduler implementation, jitter, time-triggered co-operative, cyclic executive, on-line schedule, off-line schedule*

Cite This Article: Mouaaz Nahas, "Studying the Impact of Scheduler Implementation on Task Jitter in Real-Time Resource-Constrained Embedded Systems." *Journal of Embedded Systems*, vol. 2, no. 3 (2014): 39-52. doi: 10.12691/jes-2-3-2.

1. Introduction

In this paper, we are concerned with the implementation of schedulers for use in real-time resource-constrained embedded systems. Such systems usually require high degrees of reliability and predictability while having severe resource constraints. Our particular focus is on the impact of scheduler implementation decisions on the timing behaviour of the system: in particular, variations in the interval between the release times of periodic tasks (namely, the task jitter).

There have been many previous studies which looked at the topic of jitter. Jitter has been found to arise due to clock drift, branching in the code, the scheduling algorithm employed, or as a consequence of using specific hardware [1]. In real-time systems, the jitter is mainly considered at task level (e.g. release time), and most concern about task jitter has been in the context of scheduling [2]. For example, standard scheduling algorithms based on fixed timing constraints (e.g. fixed periods and deadlines) can induce jitter if a task is blocked in a high-load situation: to deal with such issues, a range of flexible solutions have been proposed for use at run-time [3]. In distributed systems, reducing the variations in message transmission delays can help to reduce the jitter levels [4,5,6]

The presence of jitter can have a detrimental impact on the performance of many real-time systems where particular tasks must be executed at precise timing. For example, [7] show that – during data acquisition tasks – jitter rates of 10% or more may result in a meaningless interpretation of the sampled signal. Serious impacts of jitter on a wide range of applications have been discussed in [8,9,10,11].

While jitter has been widely investigated, the impact of scheduler implementation on jitter behaviour has not received widespread attention. This is unfortunate because – as we demonstrate in the course of this paper – there is a 'one-to-many' mapping between scheduler algorithms and scheduler implementations, and even comparatively small changes in the scheduler implementation can have a significant impact on the jitter behaviour [6,12-18].

Our focus in the paper is on implementations of a form of "cyclic executive" [12,19]. This algorithm is also called "time-triggered co-operative" (TTC) scheduling algorithm and it is one of the simplest schedulers that is in widespread use. The paper discusses a wide range of TTC scheduler implementations for use in both single- and multi-processor designs. It provides a systematic method for comparing between the various implementations and demonstrates how – with a little modification during the implementation stage of a scheduler (in the source code) – the jitter at the task level can be significantly reduced or entirely eliminated. Also, it will be shown how manipulating the scheduler implementation can have impacts on the computational as well as memory resources.

We would expect that the results obtained from this algorithm would also apply to more complicated algorithms (e.g. "Earliest Deadline First" algorithms: [20]). Indeed, as the algorithms grow more complicated, we would expect that the number of implementation options would increase, with a corresponding increase in the jitter variation.

The remainder of the paper is organised as follows. Section 2 presents a review of previous work in the area of scheduler implementation. In Section 3, we discuss possible sources of jitter in the TTC scheduling algorithm which is considered in this study. In Section 4, we assess the jitter levels in the original implementation of the TTC scheduler that forms the benchmark against which later implementations are evaluated. We then, in Section 5, explore several new TTC implementations which produce lower levels of jitter in single-processor systems. A similar study is repeated in Section 6 and Section 7, this time using a multi-processor TTC algorithm as the benchmark. In Section 8, we discuss the findings and present the overall paper conclusions.

2. Previous work on scheduler Implementations

The implementation of schedulers is a major problem which faces designers of real-time scheduling systems [21]. In their useful publication, Cho and colleges clarified that the well-known term *scheduling* is used to describe the process of finding the optimal schedule for a set of real-time tasks, while the term *scheduler implementation* refers to the process of implementing a physical (software or hardware) scheduler that enforces – at run-time – the task sequencing determined by the designed schedule. While there has been a great deal of interest in development, assessment and refinement of scheduling algorithms [19,20,22,23,24], we have found evidence of only a limited amount of work on scheduler implementation.

Some early work concerning the implementation of cyclic executives (in the Ada programming language) was carried out by [12]. Later on, [25] looked at the problems of implementing forms of cyclic executive in assembly language. Phatrapornnant and Pont [17] have also looked at implementation of a form of cyclic executive: they have described techniques to maintain low jitter behaviour when dynamic voltage scaling is employed (in order to reduce system power consumption). Hughes and Pont [26], [27] described an implementation of TTC schedulers with a wide range of "task guardian" mechanisms to reduce the impact of a task-overrun problem on the real-time performance of the system. In our previous publication [28], we described a low-jitter TTC scheduler framework and compared it with an early scheduler implementation (as in [14]). Two jitter-reduction techniques were later on developed and integrated with the TTC algorithm to enhance its timing predictability [29]. Such techniques form the basis of some of the TTC implementations described in this paper.

Looking more generally at scheduler implementation techniques, Katcher et al. [30] argue that that there is a wide gap between scheduling theory and its implementation in operating system kernels running on specific hardware platforms. They also note that the implementation of a particular algorithm can introduce costs which must be taken into account when validating the timing correctness properties of a real-time system. Moreover, they consider four generic scheduler implementations for a fixed priority scheduling algorithm: two time-triggered-based and two event-triggered-based. When applied to two realistic task sets – corresponding to avionics and inertial navigation applications – the different implementations demonstrated different levels of schedulability degradation. In [13], it was reported that the choice of particular implementation can have a major impact on the critical success factors for a real-time system. Xu [31] emphasised that "the simplified high-level abstraction of code" is only an approximation of "the actual real-time software implementation" which does not take into account all the implementation details that may affect timing. Therefore, the performance of the real-time system would critically depend on implementation details of the task scheduler [32]. In addition, as the system expands, the scheduler design and implementation processes will increase in complexity and, consequently, the impact on the entire system performance becomes more significant [21].

Our discussions in this paper will also concern multi-processor architectures. In this case, as we will discuss in Section 7.2, we need to implement techniques which maintain synchronisation of the clocks on the different CPUs. In this case, the impact of the underlying network protocol must be considered. For example when – for example – the Controller Area Network (CAN: [33] protocol is used to link the various processor nodes, jitter can be caused by variations in the lengths of transmitted messages. To reduce such variations, [34] proposed a technique by which transmitted data are masked before transmission. An alternative (and, it is claimed, more general) approach to reducing the variation in CAN message durations has been described in [5,6].

3. Sources of Jitter in Single-processor TTC Implementations

In order to reduce levels of task jitter in any scheduler, it is necessary to identify possible sources of timing variations. As outlined in the introduction, our focus in this paper is on time-triggered co-operative schedulers. In such schedulers, possible sources of task jitter (in single-processor implementations) can be divided into three categories:

- Scheduling overhead variation.
- Task placement.
- Tick drift.

We consider each of these categories in turn in this section.

3.1. Scheduling overhead variation

The overhead of a conventional scheduler arises mainly from context switching. In some systems, the scheduling overhead is comparatively large and may have a highly variable duration. As an example, Figure 1 illustrates how a TTC system can suffer release jitter as a result of variations in the scheduler overhead.

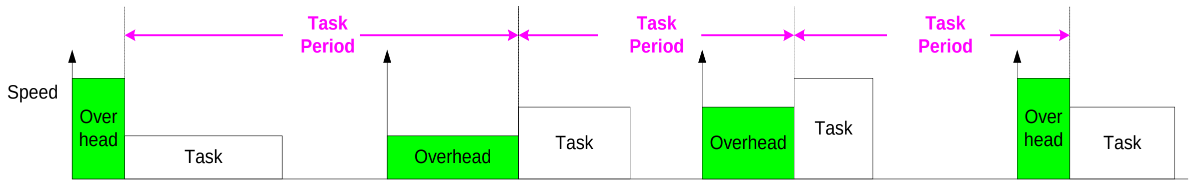

Figure 1. Release jitter caused by variation of scheduling overhead

3.2. Task Placement

Even if variations in the scheduler overhead are avoided, we may still have problems with jitter in a TTC design as a result of the task placement.

To illustrate this, consider Figure 2. In this schedule, Task C runs sometimes after A, sometimes after A and B, and sometimes alone. Therefore, the period between every two successive runs of Task C is highly variable. Moreover, if Task A and B have variable execution durations, then the jitter levels of Task C will be even larger.

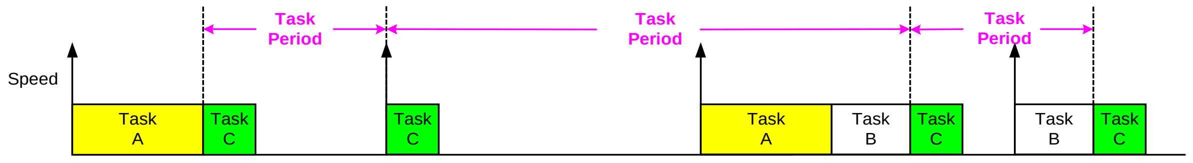

Figure 2. Release jitter caused by task placement in TTC schedulers

3.3. Tick Drift

For completeness, we also consider tick drift as a source of task jitter. In the TTC designs considered in this paper, a clock tick is generated by a hardware timer that is linked to an interrupt service routine (see Section 4). This mechanism relies on the presence of a timer that runs at a fixed frequency: in these circumstances, any jitter will arise from variations at the hardware level (e.g. through the use of a low-cost frequency source, such as a ceramic resonator, to drive the on-chip oscillator: see [14]).

In the scheduler implementations considered in this paper, the software developer has no control over the clock source. However, in some circumstances, those implementing a scheduler must take such factors into account. For example, in situations where "dynamic voltage scaling" (DVS) is employed to reduce CPU power consumption, it may take a variable amount of time for the processor's "phase-locked loop" (PLL) to stabilise after the clock frequency is changed (see Figure 3). As discussed elsewhere, it is possible to compensate for such changes in software and thereby reduce jitter (see [17]). Such techniques are not considered further in this paper.

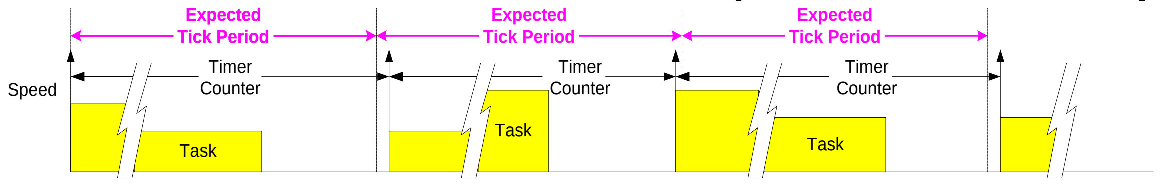

Figure 3. Tick drift in DVS systems

4. Assessing a Single-processor TTC Implementation

Having considered three possible sources of jitter in TTC algorithm, we explore the operation of a particular TTC scheduler which was originally described and fully documented [14]. We will refer to this implementation here as "Original TTC-Dispatch" [1] scheduler implementation.

4.1. The Original TTC-Dispatch Implementation

The Original TTC-Dispatch scheduler is driven by periodic interrupts generated from an on-chip timer. When an interrupt occurs, the processor executes an "Update" function (see Listing 1). In the Update function, the scheduler checks to see if any tasks are due to run and sets appropriate flags. After these checks are complete, a Dispatch function (Listing 2) will be called, and the identified tasks (if any) will be executed. When not executing the Update and Dispatch functions, the system will usually enter a low-power ("idle") mode.

1 The name is derived from the way the scheduler is implemented in software. This was to distinguish it from the earlier simpler implementations: e.g. schedulers based on super loop (TTC-SL) and interrupt service routine (TTC-ISR) (see [14] for more details).

```
void SCH_Update(void) interrupt INTERRUPT_Timer_6_Overflow
  {
  tByte Index;

  // Clear T6 interrupt request flag
  T6IR = 0;

  // 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;  // Incr. the 'Run Me' flag

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

Listing 1. "Update" ISR of Original TTC-Dispatch scheduler

```
void 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_Delete_Task(Index);
           }
        }
     }
  // Report system status
  SCH_Report_Status();

  // The processor enters idle mode at this point
  SCH_Go_To_Sleep();
  }
```

Listing 2. Dispatch function of Original TTC-Dispatch scheduler

4.2. Measuring the Task Jitter

The experimental methodology used to obtain jitter results from the TTC implementations on a single-processor system is outlined here.

4.2.1. Hardware Platform

In order to explore the impact of the TTC implementations in practical designs, we used a Phytec board supporting a 16-bit C167 microcontroller with an oscillation frequency of 20 MHz. The Keil C166 compiler was used [35].

4.2.2. Tasks

In the single processor design considered in this study, we used four different tasks with randomly-varying durations. The task schedule was chosen here in such a way that the jitter for each task was maximised. More specifically, all tasks begin to execute at time delay equals to 0 (i.e. in the first tick interval). However, the first task (called task A) runs every 2 scheduler ticks, the second task (called task B) runs every 3 scheduler ticks, the third task (called task C) runs every 4 scheduler ticks, and the fourth task (called task D) runs every scheduler tick. In this design, the Major Cycle(The Major Cycle is a period equal to the lowest common multiple of the periods of the scheduled tasks: see, for example, [12].) for the schedule is equal to 12 ticks (Figure 4). The scheduler tick interval used was 10 ms.

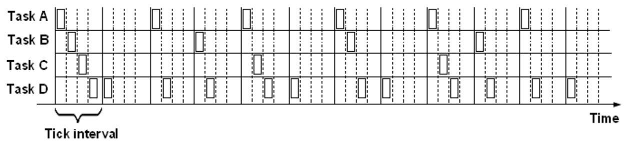

Figure 4. Scheduler tasks used in this study for measuring the jitter levels

4.2.3. Jitter Measurements

To measure the jitter on each task, we set a pin high at the beginning of the task (for a short time) and then measure the periods between every two successive rising edges. We recorded 25,000 samples in each experiment. The periods were measured using a National Instruments data acquisition card 'NI PCI-6035E' [36], used in conjunction with appropriate software LabVIEW 7.1 [37].

To assess the jitter levels, we report two values: the average jitter and the difference jitter. The difference jitter is obtained by subtracting the minimum period from the maximum period from the measurements in the sample set. The average jitter is represented by the standard deviation in the measure of average periods. Note that there are many other measures that can be used to represent the levels of task jitter, but these measures were felt to be appropriate for this study.

4.3. Results from Original TTC-Dispatch Scheduler

Table 1 shows the periods and jitter measurements for all tasks when the Original TTC-Dispatch scheduler is used.

Table 1. Task jitter from the Original TTC-Dispatch scheduler (all values in µs)

Task	A	B	C	D
Min	19994.3	29598.5	39240.6	8894
Max	20003.1	30397.7	40738.9	11073.2
Average	19999	30033.1	39999	9948.2
Diff. jitter	8.8	799.2	1498.4	2179.1
Avg. jitter	2.7	189.7	249.7	360.1

We can clearly see from the results shown in the table that all tasks in the system – including the first one – have a jitter in their release times. Moreover, we notice that the jitter levels increase as the task order increases (that is, the jitter is higher for Task B than Task A, and higher for Task C than Task B, etc.). Despite that the Original TTC-Dispatch scheduler (developed by [14] was made so simple, reliable and cost-effective, it suffers high jitter at the task release times as jitter levels in this implementation largely depend on the task schedule itself. In more severe cases this would degrade the overall real-time system performance.

5. Reducing Jitter in Single-processor TTC Implementations

In this section, we explore different ways in which a TTC scheduler can be implemented. In each case we base our solution on the Original TTC-Dispatch implementation. As we will see, different implementations provide different levels of task jitter.

5.1. Reducing Variations in the Scheduler Overhead (Modified TTC-Dispatch)

In the Original TTC-Dispatch scheduler implementation, the scheduler first determines – in the Update function (Listing 1) – which tasks are due to execute and then executes the tasks from the Dispatch function (Listing 2). A consequence of this arrangement is that the scheduler overhead varies depending on the number of tasks that are to be implemented in a given tick interval. This means that even the first task to be executed (which is, implicitly, the task with the highest priority) can suffer from release jitter.

In order to reduce the jitter in the first task, we can re-arrange the activities performed in the Update and Dispatch functions, as illustrated in Listing 3 and Listing 4, respectively. In this modified implementation (which we will call Modified TTC-Dispatch), the Update (Listing 3) function simply keeps track of the number of Ticks and all scheduling and dispatch activities are now carried out in the Dispatch function (Listing 4).

```
void SCH_Update(void) interrupt INTERRUPT_Timer_6_Overflow
   {
   // Clear T6 interrupt request flag
   T6IR = 0;
   Tick_count_G++;
   }
```

Listing 3. "Update" ISR of the Modified TTC-Dispatch scheduler

```
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;
   bit Update_again = 0;

   do {
      // 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].pTask)();  // Run the task

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
               }
            }
         }

      // Disable Timer 6 interrupt
      T6IE = 0;

      if (--Tick_count_G > 0)
         {
         Update_again = 1;
         }
      else
         {
         Update_again = 0;
         }

      // Re-enable Timer 6 interrupt
      T6IE = 1;

      } while (Update_again);

   // Report system status
   SCH_Report_Status();

   // The processor enters idle mode at this point
   SCH_Go_To_Sleep();
   }
```

Listing 4. Dispatch function in the Modified TTC-Dispatch scheduler

5.2. Results from Modified TTC-Dispatch Scheduler

Table 2 shows the impact of the scheduler modifications. The task set is identical to that used to produce the results shown in Table 1.

Table 2. Task jitter from the Modified TTC-Dispatch scheduler (all values in µs)

Task	A	B	C	D
Min	19999.4	29603.9	39245.2	8888.3
Max	19999.5	30394.4	40738.7	11103.6
Average	19999.4	29966.5	40000.2	9989.6
Diff. jitter	0.1	790.5	1493.5	2215.3
Avg. jitter	0	199.8	248	373.2

As can be seen from the table, the changes to the scheduler implementation have been successful in reducing the jitter in Task A (almost to 0). Remember that Task A is implicitly the highest-priority task in the cyclic executive scheduling algorithm. This task in many applications requires high degrees of predictability (which can be manifested by low jitter characteristics).

5.3. Reducing Variations in the Scheduler Overhead (Offline TTC-Dispatch)

One way of classifying scheduling algorithms is by the time at which scheduling decisions are made. If the scheduling decisions for the entire task set are made before the system activation, then the algorithm is called an "off-line" scheduler: if, instead, the decisions are made at run-time, we have an "on-line" scheduler [24,38]. In many cases, scheduling implementations may be on-line, off-line or some combination of the two.

The Original and Modified TTC-Dispatch schedulers make the scheduling decisions online. One approach to reducing variations in the scheduler overhead (at run time) is to make most or all of the scheduling decisions offline. We explore this option in what we will call Offline TTC-Dispatch scheduler.

In Offline TTC-Dispatch, we use an array to store the task schedule. Since all tasks are periodic, this array needs to be able to store information for the whole Major Cycle. In the implementation of Offline TTC-Dispatch, we used a desktop C program to calculate the schedule information for all tasks and stored the results in a two-dimensional array. The size of the schedule array A_s is given as:

$$A_s = N_{task} \cdot N_{tick} \quad (1)$$

Where N_{task} is the total number of tasks in the scheduler and N_{tick} is the total number of ticks in the major cycle.

Listing 5 shows the Dispatch function for Offline TTC-Dispatch scheduler.

```
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;
   static tByte Ticks = 0;   // Current tick number

   // 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 (Task_Schedule_G[Index][Ticks] == 1)
         {
         // Run the task
         (*SCH_tasks_G[Index].pTask)();
         }
      }

   // Check if the major cycle is complete. If so, re-start the ticks count
   if (++Ticks == SCH_MAJOR_CYCLE)
      {
      Ticks = 0;
      }
   }
```

Listing 5. Dispatch function in the Offline TTC-Dispatch scheduler

5.4. Results from Offline TTC-Dispatch scheduler

Table 3 presents the results obtained with Offline TTC-Dispatch scheduler.

Table 3. Task jitter from the Offline TTC-Dispatch scheduler (all values in μs)

Task	A	B	C	D
Min	19999.4	29607.4	39266.4	8920.7
Max	19999.5	30390.9	40731.5	11079.6
Average	19999.4	29974.8	39998.3	9987.4
Diff. jitter	0.1	783.5	1465.1	2158.9
Avg. jitter	0	193.1	245.2	367.7

Looking at Table 3, we can see that – although the use of an offline scheduler has produced a slight reduction in the levels of task jitter for Task B, Task C and Task D – there is still considerable room for improvement. More clearly, special techniques may need to be implemented with the scheduler to take care of the task jitter.

5.5. Reducing the Impact of Task Placement (Offline TTC-SD)

In Section 3.2, we considered the impact of task placement on "low-priority" tasks running in TTC schedulers. In order to reduce the variation in the starting times of such tasks, the Offline TTC-SD scheduler implementation places a "Sandwich Delay" (SD: [39]) around tasks which execute prior to other tasks in the same tick interval.

Briefly, a Sandwich Delay (SD) is a mechanism – based on a hardware timer – which can be used to ensure that a particular code section always takes approximately the same period of time to execute. The SD operates as follows: 1) we set a timer running; 2) we perform an activity; 3) we wait until the timer reaches a predetermined count value. In these circumstances – as long as the timer count is set to a duration that corresponds to the WCET of our sandwiched activity – we have the potential to fix the execution period.

In the Offline TTC-SD scheduler, we use SDs to provide execution "windows" of fixed sizes in situations where there is more than one task in a tick interval. To clarify this, consider again the set of tasks shown in Figure 2 and compare this with same set of tasks executed by an Offline TTC-SD scheduler, as shown in Figure 5. In Figure 5, the required SD prior to Task C is equal to the WCET of Task A plus the WCET of Task B. This implies that in the first tick, the scheduler runs Task A and then waits for the period equals to the WCET of Task B before running Task C. The figure shows that when a SD is used (as part of the TTC scheduler) prior to Task C, the periods between the successive runs of this task become equal. Note that this scheduler is based on the TTC-SD scheduler implementation we presented elsewhere [29]. The only differences are that here the whole task schedule (including estimates of tasks' WCETs) is made offline and that the hardware platform used is based on the 16-bit C167 microcontroller.

```
void SCH_Dispatch_Tasks(void)
   {
   tByte Index, ;
   bit Update_again = 0;
   tWord Req_Rls_tm;     // Required release time
   static tWord Ticks = 0;

   // Start timer 3 to count from zero
   T3=0;
   T3R=1;

   // 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 (Task_Schedule_G[Index][Ticks] == 1)
         {
         // Check the required SD for the current task
         if (Max_SD_G[Index] > 0)
            {
            Req_Rls_tm = Max_SD_G[Index] + (20*Index);  // 20*Index is a margin
            while(T3 < Req_Rls_tm);
            }

         // Run the task
         (*SCH_tasks_G[Index].pTask)();
         }
      }|
   // Stop timer 3
   T3R = 0;

   if (++Ticks == SCH_MAJOR_CYCLE)
      {
      Ticks = 0;
      }
   }
```

Listing 6. Dispatch function in the Offline TTC-SD scheduler

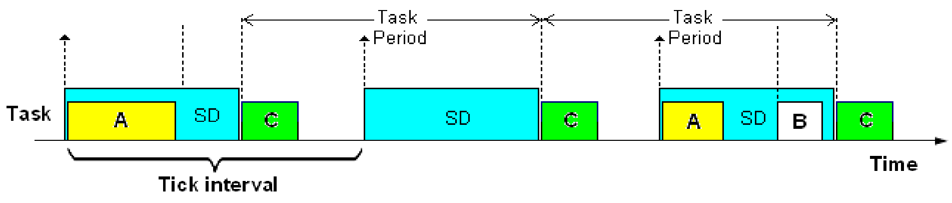

Figure 5. Using SDs to reduce release jitter in TTC schedulers

Note that – in this study – the WCET for each task is calculated offline by picking the maximum duration out of thousands of runs of the task. Dispatch code for the Offline TTC-SD scheduler is shown in Listing 6.

5.6. Results from the Offline TTC-SD Scheduler

Table 4 shows the periods and jitter measurements for all tasks when the Offline TTC-SD scheduler implementation is employed.

Table 4. Task jitter from the Offline TTC-SD scheduler (all values in µs)

Task	A	B	C	D
Min	19999.5	29998.8	39998.5	9999.2
Max	19999.6	29999.9	39999.6	10000.3
Average	19999.6	29999.3	39999.1	9999.8
Diff. jitter	0.1	1.1	1.1	1.1
Avg. jitter	0	0.3	0.2	0.2

The table shows how the use of sandwich delays prior to the execution of each task helped to remove most of the jitter in the release time of low-priority tasks. However, there is still some jitter in these tasks which is resulted from the use of software delay [29].

5.7. Reducing the Impact of Task Placement (Offline TTC-MTI)

Although the SD technique can help to fix the release time of low-priority tasks, the use of software loop – to check if the required SD for the concerned task is complete – can still result in a low level of jitter since the time taken to leave the loop and run the task is not fixed. In addition, we are forced to run the processor in normal operating mode while the SD is executing: this is likely to result in increased power consumption. In order to address both of these issues, we can use a modified sandwich mechanism employing "multiple timer interrupts" (MTIs). We will call this scheduler implementation Offline TTC-MTI. This is to distinguish it from the TTC-MTI scheduler we presented elsewhere [29] that is based on online scheduling rather than offline.

As with the Offline TTC-SD, the tasks in the Offline TTC-MTI scheduler execute in predetermined time intervals (set to match the WCET of the task concerned). In this implementation, multiple timer interrupts are used to generate the execution slots: this allows more precise control of timing. The use of interrupts also allows the processor to enter an "idle" mode after completion of each task, allowing a reduction of power consumption.

In order to achieve this, we require two timers: 1) a "Tick timer": which is used to generate the scheduler periodic tick interrupts and trigger the execution of the first task in the interval (as normal) and, 2) a "Task timer": which is used – within Tick intervals – to trigger the execution of any further tasks which are due to run in the Tick interval. The process is illustrated in Figure 6.

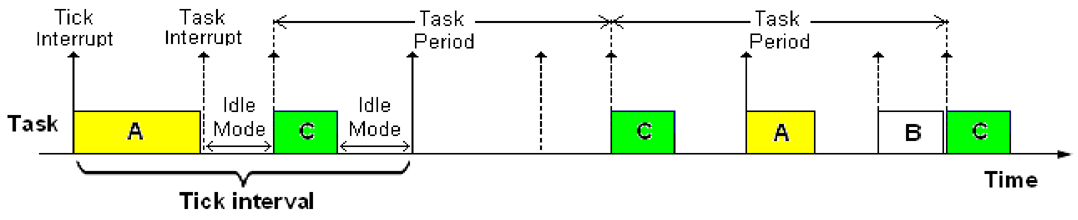

Figure 6. Using MTIs to reduce release jitter in TTC schedulers

```
void SCH_Update_T6(void) interrupt INTERRUPT_Timer_6_Overflow
   {
   // Clear T6 interrupt request flag
   T6IR = 0;

   // Reset the tick count if the end of the major cycle is reached
   Ticks_Cnt_G %= SCH_MAJOR_CYCLE;

   // Reset the task count to start from 0
   Task_Cnt_G = 0;

   T0R = 0;      // Stop T0

   // Set T0 for the next micro-tick interval
   T01CON = 0;                                              // Load CAPCOM1 T0 ctrl reg
   T0 = (0xFFFF - (Micro_Tick_G[Task_Cnt_G][Ticks_Cnt_G])); // Load CAPCOM1 T0 reg
   T0R = 1;                                                 // Start T0

   // Increment the tick count
   Ticks_Cnt_G++;
   }
```

Listing 7. "Update" ISR of the Tick-Timer-Interrupt in the Offline TTC-MTI scheduler

```
void SCC_Update_T0(void) interrupt INTERRUPT_Timer_0_Overflow
   {
   // Clear T0 interrupt request flag
   T0IR = 0;

   // Reset the dummy tick count if end of major cycle reached (jitter compensation)
   Dummy_Ticks_Cnt_G %= SCH_MAJOR_CYCLE;

   T0R = 0;      // Stop T0

   // Set T0 for the next micro-tick interval
   T01CON = 0;                                              // Load CAPCOM1 T0 ctrl reg
   T0 = (0xFFFF - (Micro_Tick_G[Task_Cnt_G][Ticks_Cnt_G])); // Load CAPCOM1 T0 reg
   T0R = 1;                                                 // Start T0

   // Dummy action for reducing jitter
   Dummy_Ticks_Cnt_G++;
   }
```

Listing 8. "Update" ISR of the Task-Timer-Interrupt in the Offline TTC-MTI scheduler

Code for the Offline TTC-MTI scheduler is shown in Listing 7 to Listing 9.

```
void SCH_Dispatch_Tasks(void)
  {
  tWord Index = 0;

  // Check the schedule array to know which task is ready at this location
  Index = Final_Task_Schedule_G[Task_Cnt_G][Ticks_Cnt_G-1];
  if(Index < SCH_MAX_TASKS)
    {
    // Run the task
    (*SCH_tasks_G[Index].pTask)();
    }
  // Increment the task count
  Task_Cnt_G++;

  // The processor enters idle mode at this point
  SCH_Go_To_Sleep();
  }
```

Listing 9. Dispatch function in the Offline TTC-MTI scheduler

5.8. Results from the Offline TTC-MTI Scheduler

Table 5 shows the periods and jitter measurements for all tasks when the Offline TTC-MTI scheduler implementation is employed.

Table 5. Task jitter from the Offline TTC-MTI scheduler (all values in µs)

Task	A	B	C	D
Min	19999.4	29999.2	39998.9	9999.7
Max	19999.5	29999.3	39999	9999.8
Average	19999.4	29999.3	39998.9	9999.8
Diff. jitter	0.1	0.1	0.1	0.1
Avg. jitter	0	0	0	0

By looking at the results presented in

Table 5, we can see that the use of the "idle" mode prior to tasks in the multiple timer interrupts method helped to further reduce the jitter in the release times of all tasks running in the system (almost to 0).

5.9. Comparing CPU overheads

In Table 6, the CPU overheads for all of the considered TTC schedulers are presented. This reflects the processor utilisation by each scheduler implementation.

To make these measurements, we set a pin high at the start of each tick and then low just before the processor goes to sleep. We report the readings for 1000 scheduler cycles. In order to compare the scheduler overhead of each model, we fixed the task durations in this experiment to make sure that the task overhead is identical in all programs.

Table 6. CPU overhead in all considered scheduler methods

Method	CPU overhead per cycle (ms)	CPU overhead per tick (ms)
Original TTC-Dispatch	4.55	0.38
Modified TTC-Dispatch	4.49	0.37
Offline TTC-Dispatch	4.34	0.36
Offline TTC-SD	20.49	1.71
Offline TTC-MTI	5.02	0.42

We can see from the table that the Offline TTC-Dispatch scheduler implementation has the lowest CPU overhead. However, since our TTC design is so simple, we see very little difference between the CPU time in on-line and off-line implementations. In more sophisticated designs, we would expect to see significant reduction in CPU processing when off-line schedulers are employed (see [40]). The difference in CPU overhead between Offline TTC-SD and Offline TTC-MTI is around 15.5 ms per cycle (1.3 ms per tick) which is so significant: this is – as expected – caused by the presence of the SDs in the Offline TTC-SD scheduler.

5.10. Comparing Memory Requirements

Table 7 presents the memory requirements for implementing the described schedulers on the CPU hardware platform considered in this study (i.e. 16-bit C167 processor).

Table 7. Data and code memory requirements in all considered scheduler implementations

Method	RAM requirements (Bytes)	ROM requirements (Bytes)
Original TTC-Dispatch	46	1016
Modified TTC-Dispatch	36	1022
Offline TTC-Dispatch	70	988
Offline TTC-SD	78	1070
Offline TTC-MTI	176	1194

Please note that the Original TTC-Dispatch implementation requires more RAM memory than the Modified TTC-Dispatch implementation. Also note that the off-line designs considered in our study required additional data memory (RAM) to store the schedule table. However, due to the simplicity of the off-line scheduler, the code memory (ROM) requirements for these schedulers are lower than the on-line equivalents.

Table 7 also shows that the Offline TTC-MTI implementation requires more data and code memory as compared to the Offline TTC-SD implementation. When choosing between these schedulers, a developer would need to weigh up jitter and resource requirements.

6. Assessing a Multi-processor TTC Implementation

Having considered possible implementation options for single-processor TTC schedulers, we now consider multi-processor designs. We begin with describing the hardware and software platforms used to implement a multi-processor testbed.

6.1. A S-C TTC Scheduler Implementation

We have previously sought to demonstrate that a "Shared-Clock" (S-C) architecture provides a simple and low-cost software framework for multi-processor TTC systems, without requiring specialized hardware [14,41]. We will employ such an architecture here, with nodes connected using a CAN protocol [33].

The scheduler used in the examples in this paper is based on the "shared-clock CAN scheduler" described previously [14]. For consistency with our previous studies [42,43], we will refer to this scheduler implementation as "TTC-SCC1".

In the multi-processor study, we investigate the levels of jitter (and implementation costs) in a simple testbed which contains two nodes: Master and Slave. To consider the impact of the transmission protocol as well as the scheduler implementation, we implement the described TTC models (original and low-jitter) in the Slave node.

Listing 10 shows the Update function for the TTC-SCC1 scheduler. The Dispatch function, used in single-processor Original TTC-Dispatch design (Listing 2), is again used in this multi-processor design.

```
void SCC_A_SLAVE_Update(void) interrupt INTERRUPT_CAN_C167CR
   {
   tByte Index;

   // Reset this when tick is received
   Network_error_pin = NO_NETWORK_ERROR;
   // Check tick data - send ack if necessary
   // NOTE: 'START' message will only be sent after a 'time out'
   if (SCC_A_SLAVE_Process_Tick_Message() == SLAVE_ID)
      {
      SCC_A_SLAVE_Send_Ack_Message_To_Master();
      // Feed the watchdog ONLY when a *relevant* message is received
      // (noise on the bus, etc, will not stop the watchdog...)
      //
      // START messages will NOT refresh the slave
      // - Must talk to every slave at regular intervals
      SCC_A_SLAVE_Watchdog_Refresh();
      }
  // Check the last error codes on the CAN bus via the status register
  if ((C1CSR & 0x0700) != 0)
     {
     Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
     Network_error_pin = NETWORK_ERROR;
     // See Infineon C167CR manual for error code details
     CAN_error_pin0 = ((C1CSR & 0x0100) == 0);
     CAN_error_pin1 = ((C1CSR & 0x0200) == 0);
     CAN_error_pin2 = ((C1CSR & 0x0400) == 0);
     }
  else
     {
     CAN_error_pin0 = 1;
     CAN_error_pin1 = 1;
     CAN_error_pin2 = 1;
     }

  // 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;  // Set the run flag

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

Listing 10. "Update" ISR in the TTC-SCC1 scheduler (Slave node)

6.2. Measuring the Task Jitter

To measure the timing, computational costs and memory requirements in the multi-processor designs considered in this paper, we used an experimental methodology based on that outlined in Section 4.2. A summary of the methodology is presented here.

6.2.1. Hardware Platform

In the multiple processor design, two Phytec boards supporting C167 microcontroller were used to implement the Master and Slave nodes. Both processors ran at 20 MHz. The Master and Slave were connected using CAN bus running at 1 Mbit/sec data rate.

6.2.2. Software Setup

On the Master node, we only set one task to run. On the Slave node, we scheduled four different tasks with different periods as outlined in Section 4.2: this is, again, to maximise the software jitter levels on the Slave tasks.

A TTC-SCC1 scheduling algorithm was implemented over the CAN protocol as follows (based on [14]): the first byte of the 8 transmitted bytes in the CAN data segment was reserved for the Slave identifier (ID) to which tick message – sent from Master – is addressed. Only the addressed Slave will reply an acknowledgement message to the Master where this message must be sent back within the same tick interval in which the tick message is received. The remaining 7 bytes of the CAN data segment contained pseudo-random values, in order to maximise the jitter caused by bit stuffing mechanism in CAN hardware [34].

Please note that since we need to incorporate a data-coding technique (see Section 7.2) – which use two bytes in each Tick message – all results include values from 8-bytes and 6-bytes models. This helps to obtain meaningful comparisons.

Please also note that a Keil C166 compiler was used for software development in all studies.

6.2.3. Jitter Measurements

The jitter in the multi-processor case is represented by measuring the interval between the release time of the (single) Master task and the release times of the tasks on the Slaves. The jitter measured by this method involves both the transmission jitter (i.e. any jitter caused by variations in the time taken to transmit a CAN message) and the software jitter (i.e. any jitter caused by the scheduler architectures on the Master and Slave nodes).

To make these measurements, a pin on the Master node was set high (for a short period) at the start of the Master task. Another pin on the Slave (initially high) was set low at the start of the Slave task we wished to study. The signals from these two pins were then AND-ed (using a 74LS08N chip: [44]), to give the transmission delays between Master and Slave. In all cases, the widths of the resulting pulses were measured using a National Instruments data acquisition card 'NI PCI-6035E', used in conjunction with LabVIEW 7.1 software.

To represent the results, maximum, minimum and average message transmission times are reported here. To assess the jitter levels, average jitter and the difference jitter were reported. The difference jitter is obtained by subtracting the best-case (minimum) transmission time from the worst-case (maximum) transmission time from the measurements in the sample set. The average jitter is represented by the standard deviation in the measure of average message transmission time.

6.3. Results from the TTC-SCC1 scheduler

Table 8 shows the transmission delays and jitter measurements for all Slave tasks when the TTC-SCC1 scheduler implementation is employed.

Table 8. Task jitter from the TTC-SCC1 scheduler (all values in μs)

Slave Task	No. of data bytes	A	B	C	D
Min	8	181	183.4	201	59.6
	6	162.7	165.1	182.4	57.3
Max	8	194	589.1	979.4	1214.5
	6	176.1	568	960.2	1177.7
Average	8	186.1	278.5	465.1	257
	6	167.3	259.9	445.5	255.2
Diff. jitter	8	13	405.7	778.4	1154.9
	6	13.4	402.9	777.8	1120.4
Avg. jitter	8	2.1	116.4	160.9	213.7
	6	2	116.6	160.7	213.6

The results in the table show that all tasks have measurable levels of jitter.

7. Reducing Task Jitter in Multi-Processor Implementations

In this section we describe how task jitter, on Slave node, can be reduced by employing the low-jitter TTC schedulers in the multi-processor design considered. In this case, we consider only multi-processor designs based on Offline TTC-MTI scheduler (described in Section 5.7) as the best TTC implementation in terms of jitter behaviour.

7.1. The TTCj-SCC1 Implementation

The described scheduler here is referred to as TTCj-SCC1 scheduler. This is to denote that the TTC scheduler used in the Slave node is with minimum jitter. Thus, the "multiple timer interrupts" (MTI) method described in Section 5.7 is used here (with the Slave scheduler). This is again to minimise the task jitter which arises from the original implementation of the TTC scheduler.

Note that when implementing the MTI method in the Slave scheduler using S-C protocol, the "Tick Interrupt" is generated by the arrival of tick messages sent periodically from the Master node where the "Task Interrupt" is generated by a Slave's on-chip timer (Listing 11 and Listing 12). The Dispatch function for this scheduler implementation is illustrated in Listing 13. Note that after the last task completes execution, the scheduler checks the network status before entering the "idle" mode.

```
void SCC_A_SLAVE_Update(void) interrupt INTERRUPT_CAN_C167CR
   {
   // Reset the timer 3 interrupt flag
   T3IR=0;

   // Stop timer 3
   T3R = 0;

   // Reset this when tick is received
   Network_error_pin = NO_NETWORK_ERROR;

   // Check tick data - send ack if necessary
   // NOTE: 'START' message will only be sent after a 'time out'
   if (SCC_A_SLAVE_Process_Tick_Message() == SLAVE_ID)
      {
      SCC_A_SLAVE_Send_Ack_Message_To_Master();
      SCC_A_SLAVE_Watchdog_Refresh();
      }

   // Reset the tick count if the major cycle is reached
   Ticks_Cnt_G %= SCH_MAJOR_CYCLE;

    // Reset the task count to start from zero
   Task_Cnt_G=0;

   // Set T3 for the next micro-tick interval
    T01CON = 0;                                           // load CAPCOM1 timer 0 control register
    T0 = (0xFFFF - (Micro_Tick_G[Task_Cnt_G][Ticks_Cnt_G])); // load CAPCOM1 timer 0 register
    T0R = 1;                                              // Start timer 0 run bit

    // Increment the tick count
   Ticks_Cnt_G++;
   }
```

Listing 11. "Update" ISR of the Tick-Timer-Interrupt in the TTCj-SCC1 scheduler (Slave node)

```
void SCC_Update_T0(void) interrupt INTERRUPT_Timer_0_Overflow
   {
   // Reset the timer 0 interrupt flag
   T0IR=0;

   // Stop timer 0
   T0R = 0;

   // Reset this when tick is received
   Network_error_pin = NO_NETWORK_ERROR;

   // Check tick data - send ack if necessary
   // NOTE: 'START' message will only be sent after a 'time out'
   if (SCC_A_SLAVE_Process_Tick_Message() == SLAVE_ID)
      {
      SCC_A_SLAVE_Send_Ack_Message_To_Master();
      SCC_A_SLAVE_Watchdog_Refresh();
      }

   // Reset the dummy tick count if the major cycle is reached (for jitter compensation)
   Dummy_Ticks_Cnt_G %= SCH_MAJOR_CYCLE;

    // Reset the dummy task count (for jitter compensation)
   Dummy_Task_Cnt_G=0;

   // Set T0 for the next micro-tick interval using offline-computed micro-tick array
    T01CON = 0;                                             // load CAPCOM1 timer 0 control register
    T0 = (0xFFFF - (Micro_Tick_G[Task_Cnt_G][Ticks_Cnt_G-1]));   // load CAPCOM1 timer 0 register
    T0R = 1;                                                // Start timer 0 run bit

    // Increment the dummy tick count (for jitter compensation)
   Dummy_Ticks_Cnt_G++;
   }
```

Listing 12. "Update" ISR of the Task-Timer-Interrupt in the TTCj-SCC1 scheduler (Slave node)

```
void SCH_Dispatch_Tasks(void)
  {
  tWord Index = 0;

  // Check the schedule array to know which task is ready at this location
  Index = Final_Task_Schedule_G[Task_Cnt_G][Ticks_Cnt_G-1];
  if(Index < SCH_MAX_TASKS)
    {
    // Run the task
    (*SCH_tasks_G[Index].pTask)();
     }

   // Increment the task count
  Task_Cnt_G++;

  // After the last task executed or checked, do these operations
  if(Task_Cnt_G == SCH_MAX_TASKS)
  {
  // Check the last error codes on the CAN bus via the status register
  if ((C1CSR & 0x0700) != 0)
     {
     Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
     Network_error_pin = NETWORK_ERROR;
     // See Infineon C167CR manual for error code details
     CAN_error_pin0 = ((C1CSR & 0x0100) == 0);
     CAN_error_pin1 = ((C1CSR & 0x0200) == 0);
     CAN_error_pin2 = ((C1CSR & 0x0400) == 0);
     }
  else
     {
     CAN_error_pin0 = 1;
     CAN_error_pin1 = 1;
     CAN_error_pin2 = 1;
     }
  }

  // Report system status
  SCH_Report_Status();

  // The scheduler enters idle mode at this point
  SCH_Go_To_Sleep();
  }
```

Listing 13. Dispatch function in the TTCj-SCC1 scheduler (Slave node)

Table 9 shows the transmission delays and jitter measurements for all Slave tasks when the TTC-SCC1-MTI scheduler implementation is employed. The results show that the Slave tasks still suffer from jitter (in this case, caused by the CAN communication protocol).

Table 9. Task jitter from the TTCj-SCC1 scheduler (all values in μs)

Slave Task	No. of data bytes	A	B	C	D
Min	8	167.8	744.5	1320	1894.4
	6	149.6	723.6	1296.8	1868.8
Max	8	178	754.6	1330.1	1904.6
	6	157.9	731.9	1305.1	1877.1
Average	8	171.2	747.9	1323.5	1897.9
	6	152.4	726.6	1299.7	1871.7
Diff. jitter	8	10.2	10.1	10.1	10.2
	6	8.3	8.3	8.3	8.3
Avg. jitter	8	1.5	1.5	1.5	1.5
	6	1.3	1.4	1.3	1.3

7.2. The TTCj-SCC1j implementation

The use of low-jitter schedulers can only compensate for the jitter caused by the software implementation of the TTC algorithm. In a multi-processor system, jitter can also arise from the characteristics of the communication protocol used (in this case CAN).

CAN uses 'non-return to zero' (NRZ) coding for bit representation. Under this scheme, a drift in the receiver's clock may occur if a long sequence of identical bits is transmitted on the bus. To overcome such a problem, CAN – at its physical layer – employs a bit stuffing mechanism in which the sending controller stuffs the opposite-polarity bit after each sequence of five identical bits detected in the data stream [45]. The bit stuffing causes the CAN frame length, and hence the transmission period, to become (in part) a complex function of the data contents. In S-C TTC schedulers, where Slaves are triggered by arrivals of messages sent from the Master, variations in the transmission time can cause variations in the release times of tasks running on the Slave nodes (Figure 7).

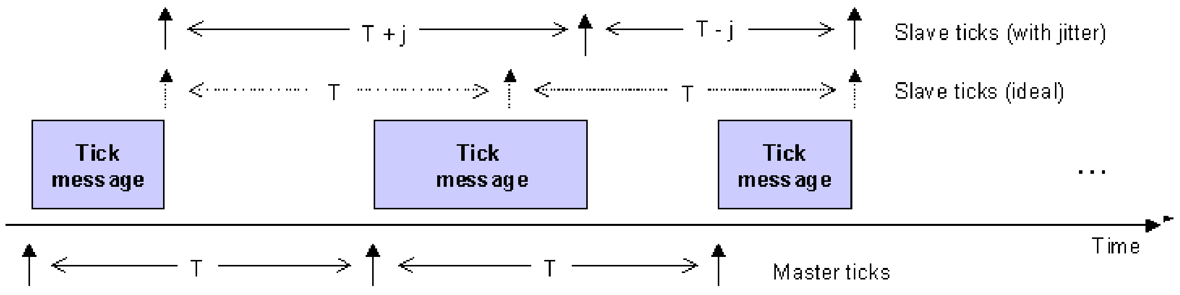

Figure 7. Impact of message-length variations on the Slave ticks in the TTC-SCC system

Researchers have previously described techniques which aim to compensate for jitter caused by "bit-stuffing" in CAN networks. For example, Nolte and colleagues [4,34] have proposed techniques which have the potential to reduce the number of stuff-bits in particular set of CAN data without imposing large computational or memory overheads. In a previous study [6], an alternative method based on "software bit stuffing" (SBS), was developed and applied to a wide range of low-cost microcontroller families. SBS uses two bytes – in the CAN data segment – for stuff coding. This helps to fix the data length in all transmitted frames on the CAN bus. As a result, SBS had the capability to reduce the message-length variation (i.e. jitter) by around 40% when the technique is incorporated in practical implementations.

In the study presented in this paper, SBS is incorporated in the low-jitter TTC schedulers (TTCj-SCC1) in order to minimise the jitter caused by the CAN hardware. The resulting scheduler is referred to here as "TTCj-SCC1j". The reason for using this name is to denote that such a scheduler has minimum jitter in the TTC scheduler employed in the Slave node as well as minimum jitter caused by the CAN messages transmission in the S-C scheduling protocol.

Table 10 shows the transmission delays and jitter measurements for all Slave tasks when the TTCj-SCC1j scheduler is employed. Note that only 6-bytes are used here for data since two bytes are required for message decoding.

Table 10. Task jitter from the TTCj-SCC1j scheduler (all values in µs)

Slave Task	A	B	C	D
Min	167.8	744.8	1320	1894.4
Max	173	750	1325.2	1899.6
Average	169.3	746.4	1321.7	1896.1
Diff. jitter	5.2	5.2	5.2	5.2
Avg. jitter	0.8	0.8	0.9	0.8

The jitter values in the table demonstrate that when the TTCj-SCC1j scheduler is implemented with SBS technique, the overall jitter on each task in the Slave node can be reduced down to approximately 5 µs on the used hardware. Remember that, in our shared-clock design, the data segment of the transmitted CAN frames contained random bytes. In practical CAN implementations, jitter levels cannot be entirely eliminated by removing all bit stuffing from the CAN data field since other fields can still induce some jitter (for more details, see [6]).

7.3. CPU Overheads

In Table 11, the CPU overhead (on the Slave) for the various TTC-SC schedulers considered in this paper are presented. To obtain these measurements, we used the methodology outlined in Section 5.9.

Table 11. CPU overhead in all considered scheduler methods

Method	No. of data bytes	CPU overhead per cycle (ms)	CPU overhead per tick (ms)
TTC-SCC1	8	4.84	0.40
	6	4.81	0.40
TTCj-SCC1	8	6.08	0.51
	6	5.96	0.50
TTCj-SCC1j	6 + 2 stuff coding	11.16	0.93

The results in the table show that – as in the single-processor systems – the scheduler employing multiple timer interrupts requires more CPU time (than the original scheduler) to set the timing for execution slots and keep tracking of the major cycle. The results also show that when SBS coding method is applied, an additional CPU load is imposed. The increase in the CPU overhead is approximately 0.4 ms / tick which is equal to the time required to perform the decoding process in SBS method when the microprocessor platform considered in this study is used [6].

7.4. Memory Requirements

Table 12 the memory requirements (on the Slave) for the various TTC-SCC schedulers considered in this paper are presented.

Table 12. Data and code memory requirements in the considered scheduler methods

Method	No. of data bytes	RAM requirements (Bytes)	ROM requirements (Bytes)
TTC-SCC1	8	64	1586
	6	58	1552
TTCj-SCC1	8	240	1916
	6	236	1882
TTCj-SCC1j	6 + 2 stuff coding	243	2070

From the table, we can see that when the SBS technique is employed in the TTCj-SCC1 scheduler, additional RAM and ROM overheads are imposed. However, it should be noted that each C167 processor used in this study has 2 Kbytes of on-chip RAM and 32 Kbytes of on-chip ROM [46]: as a consequence, the increase in memory requirements may not prove significant in most applications.

8. Conclusions

While there has been a great deal of interest in the development, assessment and refinement of real-time scheduling algorithms, we have found evidence of only limited amount of work on scheduler implementation. For example, although the impact of jitter has been widely investigated in real-time embedded systems, the impact of scheduler implementation on jitter behaviour has not received widespread attention. In order to begin to address this issue, we have explored some possible implementations of a simple TTC scheduler in this paper and reported the jitter behaviour (and resource requirements) for each implementation. It is clear from the results that even a small (and by no means exhaustive) selection of TTC scheduler implementations demonstrated a wide range of different patterns of timing behaviour.

More generally, it might be argued that – despite an enormous effort in the theoretical studies of scheduling algorithms – the results of such studies are often incomplete. The first reason for making such a claim is that (as we have demonstrated in this paper) it is not enough to say that “the system implements an XYZ scheduling algorithm”, because there is a ‘one-to-many’ mapping between scheduler algorithms and scheduler implementations. As a consequence, even under normal operating conditions, we can only define the scheduler behaviour through the source code or (given potential ambiguities in the translation of the source code: [47]) from the binary code. The second reason for arguing that many scheduling algorithms are incomplete is that they do not take into account the system behaviour when something goes wrong. For example, the most basic TTC scheduling algorithm assumes that – if a task overruns – all subsequent tasks will be delayed. This is fine, in theory, but it seems unlikely that any practical TTC implementation can ever achieve this. For example, if a task overruns for a week – or a year – then, in theory, the TTC scheduler should keep track of all “missing” tasks and execute them “immediately” when the overrunning task completes. Providing full support for such a mechanism requires a large memory capacity (potentially an infinite memory capacity).

In practice, users of a TTC scheduler will provide some mechanism for dealing with task overruns (even if these are not complete, or not completely defined). For example, general mechanisms for dealing with task overruns (in software) are discussed by [48] while a complete software implementation is discussed by [27]. Whether or not such mechanisms are incorporated in the scheduler implementation, a scheduler description is not complete if – like all aspects of the system behaviour – the recovery mechanisms are not explicitly defined.

The results (and discussions) in this paper do not simply apply to TTC schedulers. Indeed, we would expect that – as the algorithms grow more complicated – the number of implementation options would increase with a corresponding increase in the jitter variations. We would also expect that the jitter-reduction techniques described in this study can be applied, in full or in part, when implementing other scheduling algorithms.

Acknowledgement

The work presented in this paper was conducted in the Embedded Systems Laboratory (ESL) at University of Leicester, UK, under the supervision of Professor Michael Pont, to whom the author is thankful.

References

[1] M. Sanfridson, "Timing problems in distributed real-time computer control systems," *Mechatronics Lab, Dept. of Machine Design, Royal Inst. of Technology, Stockholm*, 2000.

[2] K.-J. Lin and A. Herkert, "Jitter control in time-triggered systems," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on ,*, 1996, vol. 1, pp. 451-459.

[3] P. Marti, J. M. Fuertes, G. Fohler, and K. Ramamritham, "Jitter compensation for real-time control systems," in *22nd IEEE Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings*, 2001, pp. 39-48.

[4] T. Nolte, H. Hansson, and C. Norstrom, "Minimizing CAN response-time jitter by message manipulation," in *Eighth IEEE Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings*, 2002, pp. 197-206.

[5] M. Nahas and M. J. Pont, "Using XOR operations to reduce variations in the transmission time of CAN messages: A pilot study," in *Proceedings of the Second UK Embedded Forum*, Birmingham, UK, 2005, pp. 4-17.

[6] M. Nahas, M. J. Pont, and M. Short, "Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol," *Journal of Systems Architecture*, vol. 55, no. 5-6, pp. 344-354, May 2009.

[7] F. Cottet and L. David, "A Solution to the Time Jitter Removal in Deadline Based Scheduling of Real-time Applications," presented at the 5th IEEE Real-Time Technology and Applications Symposium-WIP, Vancouver, Canada, 1999, pp. 33-38.

[8] A. J. Jerri, "The Shannon sampling theorem #8212; Its various extensions and applications: A tutorial review," *Proceedings of the IEEE*, vol. 65, no. 11, pp. 1565-1596, Nov. 1977.

[9] S. H. Hong, "Scheduling algorithm of data sampling times in the integrated communication and control systems," *IEEE Transactions on Control Systems Technology*, vol. 3, no. 2, pp. 225-230, Jun. 1995.

[10] A. Stothert and I. M. Macleod, "Effect of Timing Jitter on Distributed Computer Control System Performance," in *Proceedings of the 15th IFAC Workshop on Distributed Computer Control Systems (DCCS'98)*, 1998.

[11] M. Nahas, M. Short, and M. J. Pont, "The impact of bit stuffing on the real-time performance of a distributed control system," presented at the Proceeding of the 10th International CAN conference iCC, Rome, Italy, 2005, pp. 10-1-10-7.

[12] T. P. Baker and A. Shaw, "The cyclic executive model and Ada," *Real-Time Syst*, vol. 1, no. 1, pp. 7-25, Jun. 1989.

[13] B. Koch, "The Theory of Task Scheduling in Real-Time Systems: Compilation and Systematization of the Main Results," Studies Thesis, University of Hamburg, 1999.

[14] M. J. Pont, *Patterns for time-triggered embedded systems: building reliable applications with the 8051 family of microcontrollers*. Harlow: Addison-Wesley, 2001.

[15] S. K. Baruah, "The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors," *Real-Time Systems*, vol. 32, no. 1-2, pp. 9-20, Feb. 2006.

[16] C. Mwelwa, "Development and Assessment of a Tool to Support Pattern-Based Code Generation of Time-Triggered (TT) Embedded Systems," PhD Thesis, University of Leicester, Leicester, UK, 2006.

[17] T. Phatrapornnant and M. J. Pont, "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 113-124, Feb. 2006.

[18] M. J. Pont, S. Kurian, H. Wang, and T. Phatrapornnant, "Selecting an appropriate scheduler for use with time-triggered embedded systems.," in *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP '2007)*, Irsee, Germany, 2007, pp. 595-618.

[19] C. D. Locke, "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives," *The Journal of Real-Time Systems*, vol. 4, no. 1, pp. 37-53, Mar. 1992.

[20] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[21] Y. Cho, S. Yoo, K. Choi, N.-E. Zergainoh, and A. A. Jerraya, "Scheduler implementation in MP SoC design," in *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2005, vol. 1, pp. 151-156 Vol. 1.

[22] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237-250, Dec. 1982.

[23] A. K.-L. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Thesis, Massachusetts Institute of Technology, 1983.

[24] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. New York: Springer, 2005.

[25] S. Key, M. J. Pon, and S. Edwards, "Implementing Low-cost TTCS Systems using Assembly Language.," in *Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 2003)*, Germany, 2003, pp. 667-690.

[26] Z. H. Hughes and M. J. Pont, "Design and test of a task guardian for use in TTCS embedded systems," in *Proceedings of the UK Embedded Forum 2004*, Birmingham, UK, 2004, pp. 16-25.

[27] Z. M. Hughes and M. J. Pont, "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed," *Transactions of the Institute of Measurement and Control*, vol. 30, no. 5, pp. 427-450, Dec. 2008.

[28] M. Nahas, M. J. Pont, and A. Jain, "Reducing task jitter in shared-clock embedded systems using CAN," in *Proceedings of the UK Embedded Forum 2004*, Birmingham, UK, 2004, pp. 184-194.

[29] M. Nahas, "Employing Two 'Sandwich Delay' Mechanisms to Enhance Predictability of Embedded Systems Which Use Time-Triggered Co-Operative Architectures," *Journal of Software Engineering and Applications*, vol. 04, no. 07, pp. 417-425, 2011.

[30] D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 920-934, Sep. 1993.

[31] J. Xu, "On inspection and verification of software with timing requirements," *IEEE Transactions on Software Engineering*, vol. 29, no. 8, pp. 705-720, Aug. 2003.

[32] G. S. Avrunin, J. C. Corbett, and L. K. Dillon, "Analyzing partially-implemented real-time systems," *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 602-614, Aug. 1998.

[33] Bosch, *CAN Specification Version 2.0*. Bosch, 1991.

[34] T. Nolte, H. Hansson, C. Norström, and S. Punnekkat, "Using bit-stuffing distributions in CAN analysis," presented at the IEEE Real-Time Embedded Systems Workshop, London, 2001.

[35] Keil Software, "C166 Compiler, Optimizing 166/167 C Compiler and Library Reference, User Guide." Keil Elektronik GmbH., and Keil Software, Inc., 1998.

[36] National Instruments, "Low-Cost E Series Multifunction DAQ 12 or 16-Bit, 200 kS/s, 16 Analog Inputs." [Online]. Available: http://www.ni.com/pdf/products/us/4daqsc202-204_ETCx2_212_213.pdf. [Accessed: 08-Mar-2014].

[37] "LabVIEW System Design Software," *National Instruments*. [Online]. Available: http://www.ni.com/labview/. [Accessed: 08-Mar-2014].

[38] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob, "The performance and energy consumption of embedded real-time operating systems," *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1454-1469, Nov. 2003.

[39] M. J. Pont, S. Kurian, and R. Bautista-Quintero, "Meeting Real-Time Constraints Using 'Sandwich Delays,'" in *Transactions on Pattern Languages of Programming I*, J. Noble and R. Johnson, Eds. Springer Berlin Heidelberg, 2009, pp. 94-102.

[40] J. Xu and D. L. Parnas, "Priority Scheduling Versus Pre-Run-Time Scheduling," *Real-Time Systems*, vol. 18, no. 1, pp. 7-23, Jan. 2000.

[41] D. Ayavoo, M. J. Pont, M. Short, and S. Parker, “Two novel shared-clock scheduling algorithms for use with ‘Controller Area Network’ and related protocols,” *Microprocessors and Microsystems*, vol. 31, no. 5, pp. 326-334, Aug. 2007.

[42] M. Nahas, “Estimating Message Latencies in Time-Triggered Shared-Clock Scheduling Protocols Built on CAN Network,” *Journal of Embedded Systems*, vol. 2, no. 1, pp. 1-10, 2014.

[43] M. Nahas, “Developing a Novel Shared-Clock Scheduling Protocol for Highly-Predictable Distributed Real-Time Embedded Systems,” *American Journal of Intelligent Systems*, vol. 2, no. 5, pp. 118-128, Dec. 2012.

[44] Texas Instruments, “74LS08 Datasheet.” [Online]. Available: http://www.cs.amherst.edu/~sfl<aplan/courses/spring-2002/cs14/74LS08-datasheet. pdf. [Accessed: 08-Mar-2014].

[45] M. Farsi and M. B. M. Barbosa, *CANopen implementation: applications to industrial networks*. Baldock, Hertfordshire, England; Philadelphia, PA: Research Studies Press, 1999.

[46] Infineon Technologies, “C167CR Derivatives: 16-Bit Single-chip Microcontroller; Microcontrollers. User’s manual V 3.1.” Mar-2000.

[47] L. Hatton, “Programming Languages and Safety-Related Systems,” in *Achievement and Assurance of Safety*, F. Redmill and T. Anderson, Eds. Springer London, 1995, pp. 48-64.

[48] J. A. de la Puente and J. Zamorano, “Execution-time Clocks and Ravenscar Kernels,” in *Proceedings of the 12th International Workshop on Real-time Ada*, New York, NY, USA, 2003, pp. 82-86.