New ALT for Application Timers and Synchronisation Point Scheduling

(Two excerpts from a small channel based scheduler)

Øyvind TEIG and Per Johan VANNEBO
Autronica Fire and Security[1], *Trondheim, Norway*
{oyvind.teig,per.vannebo}@autronicafire.no

Abstract. During the design of a small channel-based concurrency runtime system (**ChanSched**, written in **ANSI C**), we saw that *application timers* (which we call *egg* and *repeat* timers) could be part of its supported **ALT** construct, even if their states live through several **ALT**s. There are no side effects into the **ALT** semantics, which enable waiting for channels, channel timeout and, now, the new application timers. Application timers are no longer busy polled for timeout by the process. We show how the classical occam language may benefit from a spin-off of this same idea. Secondly, we wanted application programmers to be freed from their earlier practice of explicitly coding communication states at channel synchronisation points, which was needed by a layered in-house scheduler. This led us to develop an alternative to the non-**ANSI C** "computed **goto**" (found in **gcc**). Instead, we use a **switch/case** with **goto** *line-number-tags* in a *synch-point-table* for scheduling. We call this table, one for each process, a *proctor table*. The programmer does not need to manage this table, which is generated with a script, and hidden within an **#include** file.

Keywords. application timers, alternative, synch-point scheduling

Introduction

This paper describes two ideas that have been implemented as part of a small runtime system (**ChanSched**), to be used in a forthcoming new product. **ChanSched** relates to processes, synchronous, zero-buffered, rendezvous-type, one-way *data* channels and asynchronous *signal / timeout* data-free channels.

It started with the first author showing the second author a private "designer's note" [1], where the problem of mixing communication states and application states is discussed. Channel state machines are visible in the code, on par with application states. This state mix had been incurred by an earlier runtime system, where synchronous channels were running on top of an asynchronous runtime system. With **ChanSched**, we now wanted to make it simpler for new programmers to understand and use these channels as a fundamental design and implementation paradigm. Even if the earlier system, described in [2] and [3], has worked fluently for years, the reasonable critique by readers was that the code was somewhat difficult to understand.

1 A UTC Fire & Security Company.
NO-7483 Trondheim, Norway. See http://www.autronicafire.no

So, the second author was inspired to build **ChanSched** from scratch, based on this earlier experience. Added goals were to use it in systems with low power consumption and high formal product "approval level".

During this development, which ping-ponged between the two of us, we solved two problems present with the previous system: *how do we manage application timers without busy-polling* and *how do we schedule to synchronisation points using only* ***ANSI C****?*

1. Application Timers in a New ALT

1.1 The Problem and our Problem

The problem was that with the earlier system we had "busy-polled" *application timers* to trigger actions. The actions could be status checks once per hour or lights to blink in certain patterns. On porting this to **ChanSched**, we wanted to avoid polling. We had written this channel-based runtime system in **C**, where its **ALT** could either have channel components, or a single[2] timer that could timeout on silent channels and nothing more. We called the latter an **ALTTIMER**, defined as *non-application timer.* We also wanted several timers and channel handling to go on concurrently in the same **ALT**. Earlier experience with occam [4] was of some help when designing this, but when going through the referees' comments to this paper, we understood that knowledge had withered. We were asked to show examples in occam-like syntax. Obeying this, we surprisingly (and problematically for the paper, we thought) saw that occam in fact does not need to build application timers by polling – its designers had included them in the **ALT**! Something had been lost in our translation to **C**. However, the surprise persisted when we saw that this exercise showed that even classical occam could benefit from our thoughts. Looking over our shoulders, maybe there was a reason why the occam timers had timed out for us.

1.2 Our ***ANSI C*** *based* ***ChanSched*** *System and the New Flora of Timers*

We start by showing a **ChanSched** process. Then, we proceed with a classic occam example and finally to a suggestion for a new timer mechanism for occam. Then, we will come back and do a more thorough discussion of our implementation.

The code (Listing 1) is an "extended" version of the **Prefix** process in **Commstime**[3] ([5] and Figure 1). Our cooperative, non-preemptive scheduler runs processes by calling the process by name, via its resolved for once address. So, on every scheduling and rescheduling, line 3 is entered. There, the process's context pointer is restored from the parameter **g_CP** (pointing to a place in the heap), filled by the scheduler. In the second half of this paper we shall see how the **PROCTOR_PREFIX** causes cooperative synchronisation (or blocking) points in the code to be reached, by jumping over lines, even into the **while** loop.

The consequence of this is that lines 5-7 are initialisation code. Knowing this one could try to understand the code as one would occam: **CHAN_OUT** and **gALT_END** are synchronisation points, meaning that line 9 will run only when a communication has taken place, and line 18 when a communication or a timeout has happened. We will not go through every detail here, but concentrate on the timers.

[2] Our implementation handled only a single timer in its **ALT**.

[3] When we developed **ChanSched**, we used **Commstime** as our natural test case. Since it has no **ALT** timer handling in any of its processes, we decided to insert our timers in **P_Prefix**, for no concrete reason.

```
01  Void P_Prefix (void)                  // extended "Prefix"
02  {
03    Prefix_CP_a CP = (Prefix_CP_a)g_CP; // get process Context from Scheduler
04    PROCTOR_PREFIX()                    // jump table (see Section 2)
05    ...  some initialisation
06    SET_EGGTIMER (CHAN_EGGTIMER, CP->LED_Timeout_Tick);
07    SET_REPTIMER (CHAN_REPTIMER, ADC_TIME_TICKS);
08    CHAN_OUT (CHAN_DATA_0, &CP->Data_0, sizeof(CP->Data_0));  // first output
09    while (TRUE)
10    {
11      ALT();                            // this is the needed "PRI_ALT"
12        ALT_EGGREPTIMER_IN  (CHAN_EGGTIMER);
13        ALT_EGGREPTIMER_IN  (CHAN_REPTIMER);
14        gALT_SIGNAL_CHAN_IN (CHAN_SIGNAL_AD_READY);
15        ALT_CHAN_IN         (CHAN_DATA_2, &CP->Data_2, sizeof (CP->Data_2));
16        ALT_ALTTIMER_IN     (CHAN_ALTTIMER, TIME_TICKS_100_MSECS);
17      gALT_END();
18      switch (g_ThisChannelId)
19      {
20        ...  process the guard that has been taken, e.g. CHAN_DATA_2
21        CHAN_OUT (CHAN_DATA_0, &CP->Data_0, sizeof (CP->Data_0));
22      };
23    }
24  }
```

Listing 1. **`EGGTIMER`**, **`REPTIMER`** and **`PROCTOR_PREFIX`** (**`ANSI C`** and macros).
(See Figure 1 for process data-flow diagram)

Note that **`ALT`** guard *preconditions* are hidden – they are controlled with **`SET`** and **`CLEAR`** macros (none shown). Only the input macros beginning with '**`g`**' check preconditions; the others do not waste time testing a constant **`TRUE`** value.

As one may understand from the above, timers are seen as channels – one channel per timer. Listing 1 is discussed in more detail throughout this paper.

1.2.1 Flora of Timers: **`ALTTIMER`**

Line 16 is our **`ALTTIMER`**. As mentioned, when neither channel **`CHAN_SIGNAL_AD_READY`**[4] nor **`CHAN_DATA_2`** have communicated for the last 100 ms, the **`CHAN_ALTTIMER`** guard causes the **`ALT`** to be taken. (The **`ALT`** structure is said to be "taken" by the first ready guard.) When a *channel* guard is taken, the underlying timer associated with the **`ALTTIMER`** is stopped. It is restarted again every time the **`ALT`** is entered.

1.2.2 **`ALTTIMER`** *and Very Long Application Timers*

The **`ALTTIMER`** was the only form of timer we had in our first port [2, 3]. Now we wanted to get further. On any scheduling from that timeout or any channel input, function calls had been made to handle application timers. An application timer is an object in local process context, used as parameter to a timer library. We needed to start, stop or poll for timeout, every so often. Since channels were often silent, the scheduling caused by **`ALTTIMER`**s bounded the resolution of our application timers. If a timeout were 100 ms, then a 10 seconds timeout would be accurate to 100 ms.

By polling with a library call, we could handle timeouts longer that the global system timer word. The 100 ms could easily build timeouts of days – out of reach of a shorter system timer, which increments a 10 ms tick into a 16 bits integer. It is the system timer that is the basic mechanism of the **`ALT`** timer; the same is true for occam. The main rationale for application timer polling in smaller real-time systems may be this extended flexibility.

[4] In order to also test non-timer interrupts, we included an analogue input ready channel. The potentiometer value thus read was used to pulse control a blinking LED. This way our example became rather complete.

1.2.3 Flora of Timers: **`EGGTIMER`** *and* **`REPTIMER`**

In order to avoid application timers by polling, we decided to implement two new timer types and make them usable directly in the **`ALT`**. An **`EGGTIMER`** times out once (at a pre-defined time) and an **`REPTIMER`** times out repeatedly (at a pre-defined sequence of equally spaced times). Generically, we will call them **`EGGREPTIMER`**s here.

These timers are initialised in application code *before* the **`ALT`**. After initialisation, they will be started the first time they are seen in an **`ALT`** – the next time the **`ALT`** is reached, they will continue to run (discussed later). They are *not* stopped when their **`ALT`** is taken by some other guard, only when they have timed out. So long as their **`ALT`** remains on the process execution path (e.g. in a loop), sooner or later they will time out and be taken. Even then, the **`REPTIMER`** will already have continued, with no skew and low jitter handling. However, **`EGGREPTIMER`**s may be stopped by application code before they have timed out. In this respect, the semantics of **`ALTTIMER`** and **`EGGREPTIMER`** differ – since an **`ALTTIMER`** has no meaning outside an **`ALT`**.

1.2.4 Arithmetic of Time

Observe that no use of our timers makes reference to *system time*, or any derived value used to store some previous or future time. Therefore, no time arithmetic with values derived from system time may be done. We have in fact not *yet* seen any need for time arithmetic at process level[5]. If this for some reason is needed, we could easily add a function to read system time.

1.3 Timers in Classical occam

As we were forced to rediscover: occam is able to handle *any* timer, including our application timers. In listing 2, there is an example of an **`ALTTIMER`** and a **`REPTIMER`**[6].

```
PROC P_Listing2 (VAL INT n, CHAN INT InChan? OutChan!)  -- extended “Prefix”
  INT Timeout_ALTTIMER, Timeout_REPTIMER:
  TIMER Clock_ALTTIMER, Clock_REPTIMER:
  SEQ
    OutChan ! n
    Clock_REPTIMER ? Timeout_REPTIMER
    Timeout_REPTIMER := Timeout_REPTIMER PLUS half.an.hour
    WHILE TRUE
      Clock_ALTTIMER ? Timeout_ALTTIMER
      PRI ALT
        Clock_REPTIMER ? AFTER Timeout_REPTIMER
          ...  process every 30 minutes
          Timeout_REPTIMER := Timeout_REPTIMER PLUS half.an.hour
          -- no skew, only jitter
        INT Data:
        InChan ? Data
          ...  process Data
        Clock_ALTTIMER ? AFTER Timeout_ALTTIMER PLUS hundred.ms
          ...  MyChan pause do background task (starvation possible)
          -- skew and jitter
:
```

Listing 2. General timers in occam.

[5] Note that the **`Consume`** process in a proper **`Commstime`** implementation in fact does use time arithmetic (for performance measurement). We measured consumed time with the debugger.

[6] To free ourselves from the **`ChanSched ANSI C`** extended **`Commstime`**, the next two examples stand for themselves, reflecting only the additional *timer* aspects of **`P_Prefix`**.

Scheduling is enabled **AFTER** the specified timeouts. Therefore both timer types will cause jitter, but **REPTIMER** will be without skew, provided a timeout is handled before the next one is due. This is the same as for the code in Listing 1.

In occam, it is the process code that does the necessary time arithmetic. Inputting system time from any timer into an **INT** (lines 30 and 33) happens immediately – no synchronisation is involved that may cause blocking. These *absolute* time values are used to calculate the next *absolute* timeout value (lines 31 and 42) and used by the **ALT** guards (lines 35 and 42). The language requires the programmer to manage these values. We remember working with occam code. Should the **TIMER** or the **INT** have **clock** or **time** in its name (or neither)? And when reading other people's code: is **now** a **TIMER** or an **INT**? The **INT** holding those absolute time values obfuscated the thinking process.

1.4 New Timers for occam

```
46  PROC P_Listing3 (VAL INT n, CHAN INT InChan? OutChan!)  -- extended “Prefix”
47    TIMER My_ALTTIMER, My_REPTIMER:   -- only timers, no variables
48    SEQ
49      OutChan ! n
50      SET_TIMER (REPTIMER, My_REPTIMER, 30, MINUTE, 24H)
51      SET_TIMER (ALTTIMER, My_ALTTIMER, 0, MILLISEC, 32BIT)
52      WHILE TRUE
53        PRI ALT
54          My_REPTIMER ? AFTER ()
55            ...  process every 30 minutes (no timeout value to compute)
56            -- no skew, only jitter
57          INT Data:
58          InChan ? Data
59            ...  process Data
60          My_ALTTIMER ? AFTER (100)
61            ...  MyChan pause do background task (starvation possible)
62            -- skew and jitter
63  :
```

Listing 3. Concrete configurable timers in a new occam.

Listing 3 shows a suggestion of how to rectify. Here, we don't need to do time arithmetic. No declaration like Line 26 is needed – we never see those **INT** values. We just have **TIMER**s, which now behave more like abstract data types: **SET_TIMER** is parameterised with unit (granularity) and length (max time). Line 50 sets a granularity of 1 minute and orders a tick every 30 minutes – thus needs 60 x 24 = 1440 (i.e. **INT16** is enough) to count up to 24 hours. Line 51 enables timeouts up to 2^{32} milliseconds – some 49 days (**INT32**).

Lines 50 and 51 also define the *type* of timer: **ALTTIMER** or **REPTIMER**.

So, we now have a set of configurable timers, mixable within the same **ALT**.

A restriction is that there may be only one **ALTTIMER**, since it defines the handling of silent channels.

The semantics of **EGGREPTIMER**s are that they are treated like channels: associated content is not touched when the **ALT** is taken. In this case it means that a stopped or running timer will stay stopped or running.

Observe that we have not banned the possibility to read system time and do arithmetic with time.

We are not concerned about precise language syntax here; enough to say that we are uncertain about parameters to **AFTER** in Lines 54 and 60. Neither have usage rules and/or runtime checks for these timers been considered.

Another point is that we suggest to *initialise* an **EGGREPTIMER** with the **SET**-command, and *start* it in the **ALT** – as we do in **ChanSched** (see below). It may start to run straight away. However, in any case it will timeout in the **ALT**. It is outside the scope of this paper to discuss the precise semantics or semantic differences of these two possibilities, or the enforcement of usage rules by a compiler. Our choice compares to calculating a new timeout value (*initialise*) and then using (*start*) that timeout in a classical occam **AFTER**.

1.5 More on our **ANSI C**-based **ChanSched** System

1.5.1 Some Implementation Issues

Our implementation relies on a separate timer server process **P_Timers_Handler**, handling any number of timeouts (Figure 1). It delivers timeouts through asynchronous send type channels carrying no data, called **TIMER** channels, one per **EGGREPTIMER**.

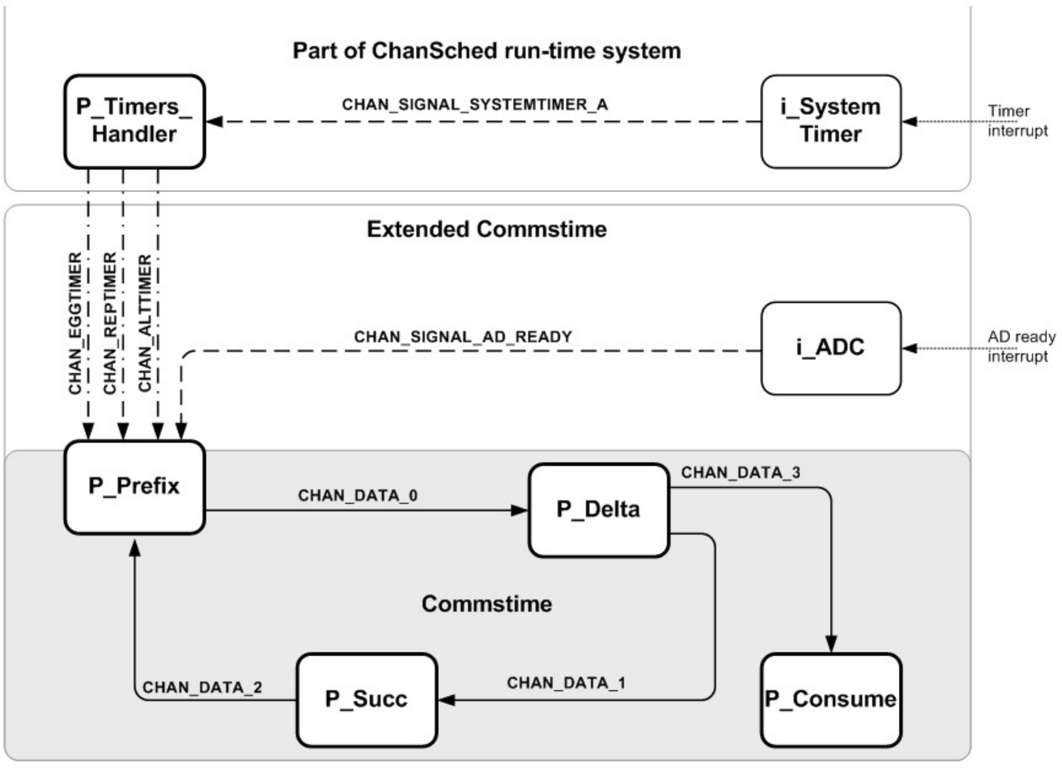

Figure 1. Our test system, called "extended Commstime".
(See Listing 1 for **P_Prefix** code)

An individual asynchronous non-blocking timer channel thus represents each timer, so any number of these may be signalled simultaneously. When a new **EGGREPTIMER** is added, the timer process is recompiled with a larger set of timers, controlled by a constant drawn from a table. So, once written and tested, the timer process requires no further editing.

Initially, we did parameter handling of **EGGREPTIMER**s in the **ALT**, modeled by how the **ALTTIMER** is handled, where **AFTER** has a parameter. However, we soon realised that this was impractical, since *start*, *restart* and preemptive *stop* often would be easiest coded (and consequently, understood) away from the **ALT**. However, the macros/functions used to handle the *start*, *restart* and *stop* do fill the timer data structure, including the value for next timeout. So, there should not be synchronising points between timer set and the **ALT**, as this could cause a wanted timeout to have passed before the **ALT** was reached. Section 1.4 discusses this in more detail for the new occam.

We use a function call and no real channel communication to set the `EGGREPTIMER`s parameters, used by the concurrent `P_Timers_Handler`. This does not violate shared value exclusive usage, due to careful coding hidden from the user and the "run-to-completion" semantics of our runtime system. Therefore, any buffer process is not needed.

`P_Timers_Handler` takes its "tick updated" signal from a signal channel, sent directly from the system timer interrupt. Processor sleep continues until the next timeout, if there is no other pending work.

1.6 Discussion and Conclusion

The `EGGTIMER` and `REPTIMER` do not seem to interfere with the `ALTTIMER`, even if their states outlive the `ALT`. To outlive the `ALT` is not as unique as one may think: any channel would in a way outlive the `ALT`, since it is possible for a process to be first on a channel when the input `ALT` is not active. This is, in fact, a bearing paradigm.

We have not done any research to find existing uses of this concept in different programming environments. We certainly suspect this could exist.

Raising timers from "link level" `ALTTIMER` to "application level" `EGGREPTIMER`s we feel is a substantial move towards a more flexible programming paradigm for timers, in our `ANSI C` based system. Now, none of the processes that need application timers need to do any busy poll. It improves understanding, coding and battery life.

Configurability of the timers has been shown. A new occam may benefit from these ideas as well. We have done no formal verification of `EGG` or `REPTIMER`s.

2. Cooperative Scheduling via the Proctor Table

2.1 The Scheduler is Not as Transparent to User Code as we Thought

This section only considers Listing 1. As described in the introduction, the non-preemptive scheduler controlling an asynchronous message system with processes that have run-to-completion semantics had never been designed to reschedule to synchronisation points, since there are no synchronisation points in the paradigm. So we built a layer on top of it ([2] and [3]), looking heavily to the `SPoC` [6] occam to C translator.

We had learnt that the asynchronous scheduler worked like the synchronous scheduler of `SPoC`, which indeed had a rich set of synchronisation points. However, that had occam source code on top. The `SPoC` scheduler had unique states for channel communication (i.e. synchronisation points) and the compiler *flattened* application states and communication states. This is the model we had used, where we did flattening of the two state spaces (in `ANSI C`) by hand.

Discovering the obvious – to make channel visibility be like many *channel-based* libraries – has been a long way to go [1]. The goal described in that note was to *"send and send and then receive, including `ALT`"* sequentially in code, with no visible communication states in between.

So we decided to make a new cooperative scheduler from scratch, also motivated by the Safety Integrity Level (`SIL`) requirements as defined by `IEC 61508`, where arguing along the `CSP` line of thinking is appreciated [7].

Our main criterion for a new scheduler was that, in some way or another, it should be able to reschedule a process to the code immediately following the synchronisation point. This is when the process had been *first* on a channel (or set of input channels) and should not proceed until the second contender arrived at the other end. This is the same functionality as described in [2] and [3] – but with invisible synchronisation points.

2.2 The Proctor Scheduling Table

Our solution was the “proctor” jump table: a name invented by us, illustrating that it takes care of scheduling and acts on behalf of the scheduler. It is generated by standard **`ANSI C`** pre-processor constructs, by hand coding or by a script. Errors in the table would cause the compiler either to issue an error about a missing label, or to warn about an unused label. We raised that warning to become an error, to make the scheme bullet proof.

Listing 4 shows how the **`CHAN_OUT`** macro first stores the actual line number, then makes a label like **`SYNCH_8_L`**, which is the rescheduling point (in Listing 1). Observe that a **`C`** macro, no matter how many lines it may look, is laid out as a single line by the pre-processor. Now, the system has a legal label to which it can reschedule; so a **`goto`** (if automatically generated) is a viable mechanism to use.

```
#define SCHEDULE_AT goto

#define CAT(a,b,c,d,e) a##b##c##d##e // Concatenate to f.ex. “SYNCH_8_L”

#define SYNCH_LABEL(a,b,c,d,e) CAT(a,b,c,d,e) // Label for Proctor-table

#define PROC_DESCHEDULE_AND_LABEL() \
        CP->LineNo = __LINE__; \
        return; \
        SYNCH_LABEL(SYNCH,_,__LINE__,_,L):

#define CHAN_OUT(chan,dataptr,len) \
        if (ChanSched_ChanOut(chan,dataptr,len) == FALSE) \
        { \
            PROC_DESCHEDULE_AND_LABEL(); \
        } \
        g_ThisAltTaken = FALSE
```

Listing 4. Some macros used to build, and usage of line number labels.

The proctor table takes us there. A **`goto`** line number (**`SCHEDULE_AT`**) taking **`CP->LineNo`** as parameter (which is not on the stack but in process context) has survived the return:

```
#define PROCTOR_PREFIX()\
        switch (CP->LineNo)\
        {\
            case 0: break;\
            case 8: SCHEDULE_AT SYNCH_8_L;\
            case 17: SCHEDULE_AT SYNCH_17_L;\
            case 21: SCHEDULE_AT SYNCH_21_L;\
            DEFAULT_EXIT\
        }
```

Listing 5. The proctor-table.

This is standard **`ANSI C`**. We avoid the extension called “computed **`goto`**” (address) that is available in **`gcc`**, a compiler we do not use for these applications [8].

We could call our solution “scripted **`goto`**” (label), just to differentiate. Listing 6 shows the output of our script, which generates the proctor table file for us:

```
In P_Commstime.c there were 4 processes, and 10 synchronisation points
In P_Timers_Handler.c there was 1 process, and 1 synchronisation point
There were a total of 2 files, 5 processes and 11 syncronisation points
```

Listing 6. Log from the ProctorPreprocessor script.

When the scheduler always schedules the process to the function start in Listing 1, the proctor table macro causes the process to re-schedule to the correct line. The code is truly invisible but available, since the macro body is contained in a separate `#include`d file.

Initially, a dummy `CP->LineNo`, set up by the run-time system, is set to zero. This takes the process through its initialising code: from the proctor table to the first synchronisation point, Line 8 of Listing 1.

2.3 Discussion and Conclusion

The complexities of a preemptive scheduler – and the fact that we do not need one – makes this solution quite usable. It is safe and invisible to the user, who do not need to relate to link level states (also called communication states or synchronisation points). So, the user needs to relate only to application states. The code is portable, standard **`ANSI C`**. Local process variables that reside on the stack will not survive a synchronisation point, so the programmer has to place these in process context. The overhead of the proctor jump table also includes storing the next line number at run-time, but this is small and acceptable for us. These points are less frequent than function calls, but are comparable in cycle count.

3. Conclusions

Section 1 shows that differentiating configurable types of timers in the **`ALT`** may raise timers to a higher and more portable level.

Section 2 displays the use of a standard **`ANSI C`** feature wrapped into a jump (proctor) table, in service for a cooperative scheduler.

Making **`ANSI C`** process scheduling with invisible channel communication and synchronisation states is a step forward for us. With **`EGGTIMER`**s and **`REPTIMER`**s, process application code is now easier to write, read and understand.

We have also noted that there may be a need to show timer handling into an "extended `Commstime`", so that implementors could have a common platform also for this.

Acknowledgement

We thank the management at the Autronica Fire and Security's development department in Trondheim for allowing us to publish this work.

*[**Øyvind Teig** is Senior Development Engineer at Autronica Fire and Security. He has worked with embedded systems for more than 30 years, and is especially interested in real-time language issues (see* `http://www.teigfam.net/oyvind/pub` *for publications and contact information).* ***Per Johan Vannebo*** *is Technical Expert at Autronica Fire and Security. He has worked with embedded systems for 13 years.]*

References

[1] Ø. Teig, *A scheduler is not as transparent as I thought (Why CSP-type blocking channel state machines were visible, and how to make them disappear),* in 'Designer's Notes' #18, at author's home page, `http://www.teigfam.net/oyvind/pub/notes/18_A_scheduler_is_not_so_transparent.html`

[2] Ø. Teig, *From message queue to ready queue (Case study of a small, dependable synchronous blocking channels API – Ship & forget rather than send & forget),* in 'ERCIM Workshop on Dependable Software Intensive Embedded Systems', in cooperation with *31st. EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA),* Porto, Portugal, 2005. IEEE Computer Press, ISBN 2-912335-15-9. Also at `http://www.teigfam.net/oyvind/pub/pub_details.html#Ercim05`

[3] Ø. Teig, *No Blocking on Yesterday's Embedded CSP Implementation (The Rubber Band of Getting it Right and Simple)*, in 'Communicating Process Architectures 2006', P.H. Welch, J. Kerridge, and F.R.M. Barnes (Eds.), pp. 331-338, IOS Press, 2006.
Also at `http://www.teigfam.net/oyvind/pub/pub_details.html#NoBlocking`

[4] Inmos Limited, *The* occam *programming language*. Prentice Hall, 1984.
Also see `http://en.wikipedia.org/wiki/occam_(programming_language)`

[5] P.H. Welch and F.R.M. Barnes, *Prioritised Dynamic Communicating Processes - Part I.*
In 'Communicating Process Architectures 2002', J. Pascoe, R. Loader and V. Sunderam (Eds.), pp. 321–352, IOS Press, 2002.

[6] M. Debbage, M. Hill, S. Wykes, D. Nicole, *Southampton's Portable occam Compiler (`SPoC`).*
In: R. Miles, A. Chalmers (eds.), in 'Progress in Transputer and occam Research', WoTUG 17, pp. 40-55. IOS Press, Amsterdam, 1994.

[7] IEC 61508, *SIL: Safety Integrity Level (SIL level), a safety-related metric used to quantify a system's safety level.* See `http://en.wikipedia.org/wiki/Safety_Integrity_Level`

[8] Wikipedia, *Computed goto.* See `http://en.wikipedia.org/wiki/Goto_(command)#Computed_GOTO`