ERCIM Workshop on Dependable Software Intensive Embedded systems
In cooperation with EUROMICRO 2005
Porto, Portrugal

From message queue to ready queue

Case study of a small, dependable synchronous blocking channels API
"*Ship & forget* rather than *send & forget*"

Øyvind Teig
Autronica Fire and Security, Trondheim (A UTC Fire and Security company)
http:\\home.no.net\oyvteig

Abstract

This case study shows CSP style synchronous interprocess communication on top of a run-time system supporting SDL asynchronous messaging, in an embedded system. Unidirectional, blocking channels are supplied. Benefits are no runtime system message buffer overflow and "access control" from inside of a process of client processes in need of service. A pattern to avoid deadlocks is provided with an added asynchronous data-less channel. Even if still present here, the message buffer is obsoleted, and a ready queue only could be asked for. An architecture built this way may be formally verified with the CSP process algebra.

1. Introduction

This "industrial" paper assumes that asynchronous interprocess communication is known by the reader. It describes a case study where the "opposite" – synchronous interprocess communication – was a viable solution, even for a small embedded system. A reader should hopefully be triggered to investigate further, as this short six page format implies.

The starting point for this now "work done" case was an in-house process/task non-preemptive run-time system written in C, compiled for an Atmel AVR processor, which contained 128 KB FLASH for program code and 32 KB external RAM. Several products built on this architecture had been successfully shipped. Messages were always asynchronous, meaning that any sender process would "send & forget" and go on.

But there were aspects where that design could be enhanced, like 1.) the system message buffer could in theory overflow, 2.) pointer movement between processes (and possible race conditions) is difficult to handle and 3.) incoming messages could arrive in a process unprotected: regardless of its internal state.

However, we did close these cases by careful design and field trials.

Still, in another product, we decided to build a layer of synchronous, blocking and unidirectional *channels* on top of the asynchronous system. Having also shipped a product with this paradigm and implementation, with no new software release after a year's use (also thanks to stable functional requirements), we decided to continue and use it in a second product.

The concern here had initially been to select a dependable software pattern to avoid deadlocks. Interestingly, the selected pattern includes data-less asynchronous signal channels (later).

2. SDL and CSP

The two "competing" paradigms here are SDL ("the asynchronous") and CSP ("the synchronous" in this context). The edit-by-anyone *Wikipedia* dictionary [1] (also pointing to more academic sources) has entries of both:

SDL: "SDL (short for Specification and Description Language) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems. It is defined by the ITU-T (Recommendation Z.100.) Originally focused on telecommunication systems, its current areas of application include process control and real-time applications in general."

CSP [2]: "'Communicating Sequential Processes' which was published in 1985. In May 2003, that book was the third-most cited computer science reference of all time according to Citeseer (albeit a very unreliable source due to the nature of it sampling). … As its name suggests, CSP allows us to describe systems as a number of components (processes), which operate independently and communicate with each other solely over well-defined channels. CSP introduces a process algebra which is used to describe a process' communications with its environment."

Some languages influenced by CSP are occam [3] and Ada [4].

Synchronous systems may be built with asynchronous components, by adding some kind of handshake (like, waiting for a reply or building mechanisms into the run-time system). Likewise, asynchronous systems may be built with synchronous components, by adding some kind of overflow handling (like, overflow buffer processes.)

Most embedded system would probably need to use both paradigms. If we build with solely synchronous primitives, no input should allow the software or system to malfunction (therefore, have control on input, so that loosing data is a conscious action). If we build solely on asynchronous primitives, no interprocess communication should be allowed to crash the system (therefore, have control on it and introduce some kind of handshake or synchronism).

3. Blocking

When a process waits for a reply or access to other processes (below), it *blocks* "on the synchronous channel". Think of blocking as equal to what a calling function does when it waits for a called subroutine to return. There is no other thing to do than "wait".

Opposite, an asynchronous sending will *not* block.

Starting new processes from within a process may have "blocking fork/join sematics", depending on the operating system (occam blocks). In our system processes are only spawned from "main", before the scheduler is entered (the spawning in itself will not block, so that more than one process may be started).

With blocking semantics, *parallel slackness* becomes an issue – that we have enough processes to get things done, with a goal to handle all necessary I/O activity. Total work done should not be less.

Observe that passive waiting is indeed used when "delay" is wanted.

4. Access control of other processes

With blocking semantics, we will also be able to have others hang while waiting for *this* process. This could be busy processing or busy with another session. Then, in due course, we would open access to this from others, and process their messages, one in turn.

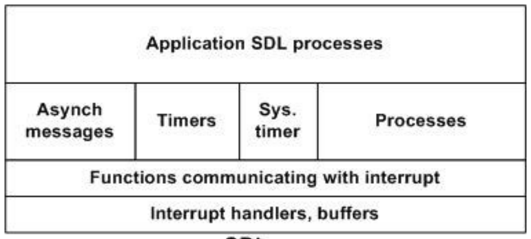
A side effect of this is that we may *choose whom we want to serve*, so that the server (this) could serve the clients (them) in a *fair* manner – or in the manner it chooses.

The term most often used for this is *selective waiting*. Both occam and Ada (and UML 2.0 [5]) have structures for it, and we have implemented it in the channel layer. It is called ALT for ALTernative.

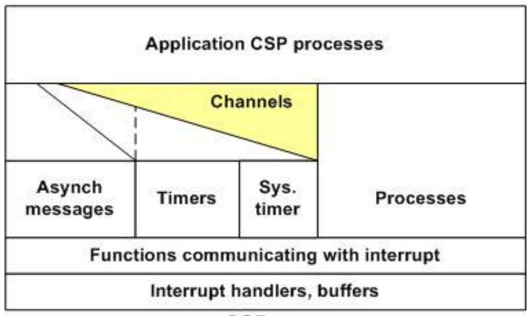
Another facet is that processes built with this scheme certainly need to obey the protocol semantics in interchange with other processes, but they do not need to know the semantics of the other processes' internal behaviour. As some times with asynchronous systems, *when*, and perhaps even *if* a message is put in a common message queue, need not be known. Not needing to know another party's semantics has been referred to as WYSIWYG semantics [6]. This also makes processes become dependable software components. It also shows the compositional semantics of CSP.

Observe that this is not the same as the *monitor* mechanism implemented in Java, where some sort of queuing of the blocking threads is done. However, one can build channels with monitors as building blocks in Java, .NET and Posix [7]. At least one major operating system bases interprocess communication on synchronous, blocking, rendezvous (later) type communication from bottom and up, and the vendor argue strongly about the safety perspective of this solution [8].

5. The layered architecture


Application SDL processes
Asynch messages
Timers
Sys. timer
Processes
Functions communicating with interrupt
Interrupt handlers, buffers

SDL


Application CSP processes
Channels
Asynch messages
Timers
Sys. timer
Processes
Functions communicating with interrupt
Interrupt handlers, buffers

CSP

The channel abstraction API resides on top of the SDL run-time system. Only one SDL function is not abstracted, the _FSM_Init_, which initialises a process. "SDL processes" may coexist with "CSP processes", but any communication between the two worlds would have to be done as asynchronous messages, with channels not used by the CSP process in charge during such a session.

6. The channels C API abstraction

All C code use of this API is by macros. A channel is a global data structure containing id of the *first* process, one half of the memcpy parameters, and states in the channel. For debug purposes we optionally insert intended sender and receiver identification.

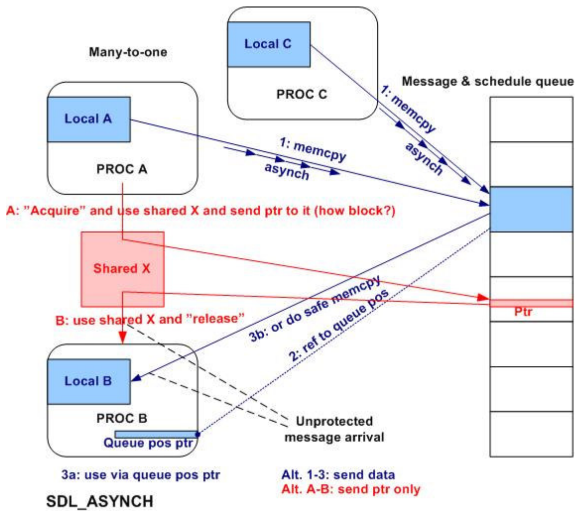
```
#define CHAN_INIT_F
        (CHAN,SENDER,RECEIVER,ALTTAKEN)
#define CHAN_IN_F
        (CHAN,DATA,EVENT)
#define CHAN_IN_VARLEN_F
        (CHAN,LEN,DATA,EVENT)
#define ALT_CHAN_IN_F
        (GUARD,CHAN,DATA,EVENT,ALTTAKEN)
#define ALT_CHAN_IN_VARLEN_F
        (GUARD,CHAN,LEN,DATA,EVENT,ALTTAKEN
#define ALT_CHAN_IN_ASYNC_SIGNAL_F
        (GUARD,CHAN,EVENT,ALTTAKEN)
#define CHAN_OUT_F
        (CHAN,DATA,EVENT)
#define CHAN_OUT_VARLEN_F
        (CHAN,LEN,DATA,EVENT)
#define CHAN_OUT_ASYNC_SIGNAL_F
        (CHAN,RESCHEDULEME)
#define CHAN_IN_ASYNC_SIGNAL_F
        (CHAN,EVENT)
#define ALT_TIMER_IN_F
        (GUARD,TIME,UNIT,EVENT,ALTSTATE,ALTTAKEN)
#define FSM_RESCHEDULE_F
        (EVENT)
```

There are channel initialisation, inputs (for use in ALT constructs and not) and outputs (outputs do not know whether they are part of any ALT). An input may also have a timeout attached. And the local communicating state machines (processes) may need to want themselves to be rescheduled in some cases.

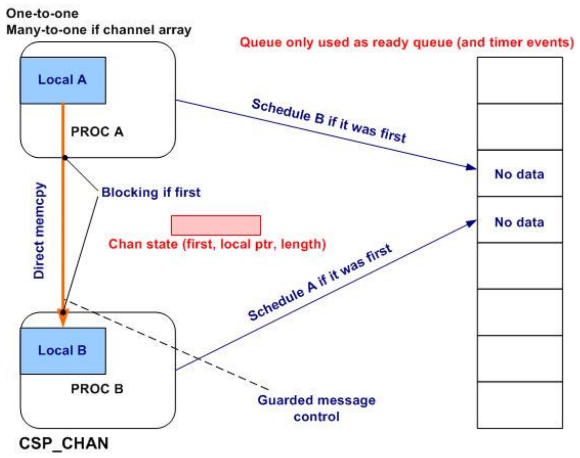
All these macros define state changes that are visible from the outside of a process.

7. Semantics of asynchronous messages

A message is sent (*memcpy*'ed) into a message queue of individual elements, all with the size of the largest element. Proc A and B may proceed to send other messages immediately, before they are descheduled, i.e. they do not block. A receiver must handle the incoming message queue (and *memcpy* the data again if it wants to keep it) on a first come first serve basis. Should any of the messages not confirm with some process inner state, it is discarded or set aside for later processing. The process becomes another scheduler above the run-time scheduler. If one needs to send larger chunks of data than one could afford to put into the message queue, a pointer to the message is sent instead. This requires access mechanisms to be built into the data segment, and some kind of feedback to the sender to inform that data has been read and is free (i.e. some synchronisation mechanism). In effect, one invents part of the channel concept anew every time. Also, with asynchronous messages, a (fast) producer and a (slow) consumer may be out of phase, and there is no mechanism to avoid queue overflow (..than to insert synchronousity). In most systems an overflowed interprocess message queue is detected by the run-time system, which then often have no other way out than to restart the system.



8. Semantics of synchronous channels


One-to-one
Many-to-one if channel array
Queue only used as ready queue (and timer events)
Local A
PROC A
Schedule B if it was first
No data
No data
Blocking if first
Direct memcpy
Chan state (first, local ptr, length)
Schedule A if it was first
Local B
PROC B
Guarded message control
CSP_CHAN

A *channel* is a named entity. You send or receive “on a channel”, not with a named process. It acts as a handle to a "protected" region of the code (or rather to a state-controlled protected phase), where the two interacting processes meet and do a *memcpy* from sender's internal to receiver's internal data structures. The memcpy is invisible in process code. In our case the channel is one-way. This meeting and memcpy'ing phase is often called a *rendezvous*, an Ada term. The part arriving first (sender or receiver) blocks and is descheduled until the second part (receiver or sender) arrives. Then memcpy and continuation of last part, and rescheduling of first.

Observe that an input *timeout* or simply a *wait* (for polling) is implemented as a channel with a timer on the sender side. The run-time system inserts the rescheduling message when a timer has timed out. It also hinders timeouts to enter a process incorrectly if the optional channel(s) attached have already been taken.

Observe that we have not implemented timeout on sending. To implement this on on-chip interprocess communication with non-preemptive scheduling would have been straighforward, had we seen any need for it.

The *asynchronous* channel implemented here does not cause the message buffer to overflow if one rule is obeyed: That no new asynchronous signal is sent before contact with the receiver has been achieved.

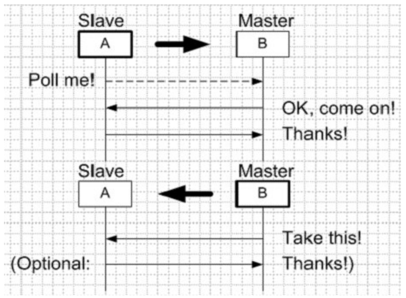
9. “From message queue to ready queue”

This paper’s title implies that the message queue is now *not* used for anything else than for sending dummy signals to a process, with the only purpose of scheduling it. So, we could have dropped the message queue for a process ready queue. But the asynchronous run-time system is well tested and we wanted to keep it.

Such a ready queue would also need to contain the cause of the rescheduling. The place this is needed is in an ALT, where we must know which clause that caused the rescheduling (i.e. channel number or timeout).

10. Deadlock avoidance

Where two processes spontaneously need to send data to each other, a blocking communication scheme might cause deadlock. This is a state where processes try to communicate – in a circular pattern – *blocking* for each other to become ready, and therefore unable to proceed. This is pathological and *must not* happen. All processes run at the same priority, so any priority inversion problem is ruled out in the system.



The pattern we chose to avoid this was to give the processes clear roles: slave and master. Master may send on the blocking channel any time (solid arrows) when it has something. Slave would never block on any spontaneous message, since the asynchronous “poll me” message (stippled arrow) lets the slave go on and not block, and then instead hang in an INPUT or ALT on the input channel. When the “OK, come on” message arrives from the master, the agreed upon protocol assures no deadly embrace.

11. Coding examples

(The text folds starting with “... “ in the code excerpt here contain code, or other folds.)

```
P_BS100_Sim (void)
{
    switch (ContextPtr->State)
    {
        ...  ST_INITIALIZING_A (Initialize)
        ...  ST_STATE_IN_ALT_030_032_104_A
        ...  --> ST_STATE_IN_ALT_030_032_104_END_A
        ...  ST_STATE_OUT_031_A
        ...  ST_STATE_OUT_110_A
        ...  ST_STATE_OUT_END_A
        ...  ST_STATE_STOPPED_A, (crash)
        ...  default, (crash)
    }
    ...  Common action: get more to do from workpool
}
```

All process data is kept in a local "Context" in each process. It is allocated on the heap in the initialising phase. The scheduler schedules a process when it *calls* it. A process keeps track of its own state and does a switch/case on this state. It must run to completion on every scheduling, since there is non-preemptive scheduling. Therefore a process function return goes back to the scheduler.

This gives the nice side effect of one common stack for all processes.

Asynchronous i/o is pulled in/out by interrupts that use queues, which are polled by driver type processes.

```
case ST_STATE_IN_ALT_033_093_A:
{
    bool_a alt_taken = FALSE;
    g_ALT_AL_ComP_Driver = CHAN_ALT_ENABLED_ON_A;

    ALT_CHAN_IN_F  (ContextPtr->Guard_033_093, g_chan_033,
                    ContextPtr->ALCP_033i_093i,
                    S_EVENT_ALT_033_093_A, &alt_taken);
    ALT_CHAN_IN_F  (ContextPtr->Guard_033_093, g_chan_093,
                    ContextPtr->ALCP_033i_093i,
                    S_EVENT_ALT_033_093_A, &alt_taken);
    ALT_TIMER_IN_F (ContextPtr->Guard_Timer_ENTX, MS_TIMEOUT_A,
                    TU_MS_A, S_EVENT_TIMEOUT_A,
                    &g_ALT_AL_ComP_Driver, &alt_taken);

    ContextPtr->State = ST_STATE_IN_ALT_033_093_120_END_A;
    break;
}
```

Above, is an example of an input ALT construct, where we see two channels with a timeout. A component of the ALT will be skipped if its “Guard_” becomes FALSE. This is the way to control client's access.

Not shown is the hand-coded test to verify that all guards are not FALSE (equal to the occam STOP, where a process cannot proceed). If so, it is a programming or design error, suited to "crash" – for further investigation and required program update.

Observe that there is no busy waiting by the process to facilitate waiting on a channel. This is because the second contender "pulls" the rescheduling of the first. This is the usual *monitor & condition variable* solution (also [7]).

12. Formal basis of the architecture

CSP, on which this scheme is based, has not been much discussed here, but it is possible to model and verify any system with this process algebra [9]. This would be out of reach (expensive), and not very interesting for us (small system and use of known software patterns). (Admittedly, this author knows CSP through occam and has had no hands-on experience with it.) However, other process algebras, like FSP, analysed with the free LTSA tool, may also be used [10] (it has been tried). Modelling asynchronous systems (albeit with finite size buffers, which makes them synchronous when buffers are empty/full) is also possible with Promela and the free SPIN tool [11].

The channel layer API discussed here was modelled on macros used by the code generator of the *SPoC* occam-to-C compiler [12]. SPoC also was a non-preemptive run to completion system, with appropriate scheduling queues and a timer queue (but no message queue). All the communication states and process start & stop that we would have to code by hand in the project was done automatically by SPoC, since occam supports channel input, output, ALT, input timeout and wait, as well as compositional prioritised processes.

13. Discussion

The system adds approximately 2 KB of program memory. Execution time overhead (as compared to the asynchronous system) depends on whether the asynchronous system uses any synchronization to make it "safe" (this would make the systems about equal), and how the asynchronous system uses the received data. If it needs to keep the data past next descheduling, the asynchronous system needs *two* memcpy's.

Setting up a communication or an ALT obviosuly takes more cycles than just returning from the process, which is the asynchronous behaviour. But these cycles are only a small percentage of burnt cycles in our processes anyhow, so the overhead has been insignificant for our applications.

We would not have used this system had we had, say, 8 KB of code space (the SDL runtime system is about 20 KB, and we need that anyhow). But with 128 KB of code space (or, soon 256 KB – nice for an 8 bit machine) and 16 MHz clock, the added well-being of knowing that the system never overflows the message queue or sends unwanted messages into a processes, outweighs the overhead.

Using this methodology (with occam, SPoC and a C CSP library) has proven valuable for about 15 years, where (provided functional requirements are stable), "ship & forget" was more the rule than the exception. This was in embedded systems (then with Transputers and later DSPs and Intel 386ex machines) and on host Windows machines.

When the communication states have been set up, using them is straightforward: fill local structs in the context and set a state variable to send, or just set a state variable to input or wait.

But, there is more complexity to this system than I like to admit, also when it comes to personal engineers’ preferences and background. If OO has had its way, the CSP kind of thinking certainly also has [13].

Grasping the communicating state machines, which are not in the application domain, but constitute the skeleton of the process/data-flow architecture, is individual. A channel most probably seems as belonging to OSI *network* (3) or *transport* (4) layer, and certainly not the application layer (7). Some programmers learn this methodology easily; let them handle it. Some resist or do not bother about these technicalities, let them concentrate on the product proper and *application* communication layer. The infrastructure person(s) should then set up the necessary construct for the application people to just use.

However, when the communication infrastructure code once has been set up, it tends to stay stable and work.

Setting the size of the present ready queue is a matter of finding the maximum scheduling incidence volume. When this is found, even the producer-consumer problem will not cause further queue usage. To find this value we let the scheduler catch any overflow and then increase, with a margin. This is the same procedure as with a pure asynchronous system. However, when maximum has been found, there is no room for further surpises, since the value is a function of the number of channels and processes, not the communication pattern.

A future dream is to have (a subset of?) Ada available for microcontrollers of this type, or Java (where CSP libraries [14][15] are available). Or hope that result of ongoing occam research will hit industry some day [16]. In the meantime, we could use solutions as the one discussed here, which really is quite dependable, even if it is based on hand-written C.

14. References

Ref. [17] has been added since it is a good starting point for both theory and practice of this field of computer science. Use this list as hands-on and academic starting points, not especially for direct referencing of origins.

[1] *Wikipedia* at http://en.wikipedia.org/wiki/
SDL at #Specification_and_Design_Language
CSP at #Communicating_sequential_processes

[2] C.A.R. Hoare, *Communicating Sequential Processes,* Prentice Hall, 1985

[3] Wikipedia [1] at #occam_programming_language

[4] Wikipedia [1] at #Ada_programming_language

[5] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual, 2.ed.*, Addison Wesley, 2004

[6] P.H. Welch, University of Kent at Canterbury, UK in mail list, at http://www.wotug.org/lists/occam/1142.txt

[7] P.B. Hansen and C.A.R. Hoare, monitors, see Wikipedia [1] at #Monitor_%28synchronization%29

[8] Integrity from Green Hills Software Inc.

[9] The tool FDR2 by Formal Systems

[10] J. Magee and J. Kramer, *Concurrency, State models and Java programs*, Wiley, 1999.
Free tool at http://www-dse.doc.ic.ac.uk/concurrency/

[11] G.J. Holzmann, *The Spin Model Checker*, *Primer and reference manual*, Addison-Wesley, 2003
Free tool at http://spinroot.com/spin/whatispin.html

[12] M. Debbage, M. Hill, S. Wykes, D. Nicole, *Southampton's Portable Occam Compiler (SPOC*", In: Miles, Chalmers (ed.), "Progress in Transputer and occam Research", IOS Press, Amsterdam, 1994 (WoTUG 17 proceedings), pp. 40-55.
Free tool at: http://gales.ecs.soton.ac.uk/software/spoc/

[13] Ø. Teig, *CSP: arriving at the CHANnel island - Industrial practitioner's diary: In search of a new fairway*, in "Communicating Process Architectures", P.H. Welch and A.W.P. Bakkers (Eds.), IOS Press, NL, 2000, Pages 251-262, at http://home.no.net/oyvteig/pub

[14] P.H. Welch and P.D.Austin. *Communicating Sequential Processes for Java (JCSP)*, 1999-
At http://www.cs.kent.ac.uk/projects/ofa/jcsp/

[15] Communicating Threads for Java (CTJ),
G. Hilderink, J. Broenink, W. Vervoort, A. Bakkers *Communicating Java Threads*, in the Proceedings of the 20th World Occam and Transputer User Group Technical Meeting, pp. 48-76, ISBN 90 5199 336 6, IOS Press, The Netherlands. At http://www.ce.utwente.nl/javapp/

[16] P.H.Welch, F.R.M. Barnes (University of Kent at Canterbury), *occam-pi: blending the best of CSP and the pi-calculus*,
at http://www.cs.kent.ac.uk/projects/ofa/kroc/

[17] WoTUG home page: http://www.wotug.org/