Trondheim, 26 October 1995
Øyvind Teig
Autronica

AUTRONICA

Semaphores and Dual-Port Memory

Introduction

During the last years, the price of dual-port memory has dropped to a level where design-ins in embedded systems has become feasible. A dual-port ram seems attractive, but how are we actually going to use the thing? The two processors, one on each side of the dual-port ram, cannot just read and write to the dual-port ram at any time. In order to help the designers with this problem, the dual-port ram these days are often produced with internal *semaphores*, flags that may be owned by only one processor at any time.

These semaphores are only basic building blocks, a scheme has to be built on top of them to allow data to be safely passed from one processor to the other. Using the semaphores, several solutions could be implemented. The dual-port ram itself could hold a state variable that is used during the processors' arbitration. Another solution would be to guarantee that the other processor reads and modifies the data within a 'hard' time limit.

This article describes a third scheme where no state information inside the dual-port ram area is used, and no dependence on time exists. Three of the usual 8 hardware-semaphores are used. The two processors may differ in processing power and speed. The processors on each side pass through very simple state-machines with only one possible next-state. We could call the proposed solution 'ping-pong', since a privilege is passed back and forth between the processors continuously.

The solution is simulated with an occam program (see Appendix).

PING-PONG

The scheme describes a privilege that is passed between the processors (fig.1). A processor may hold the privilege as long as it wants to. When a processor has the privilege, it is free to do whatever it wants with the buffers. The privilege is passed with the three semaphores which need to be used for the scheme. The semaphores are named 'A', 'B' and 'C'. Fig. 2 shows a complete sequence.

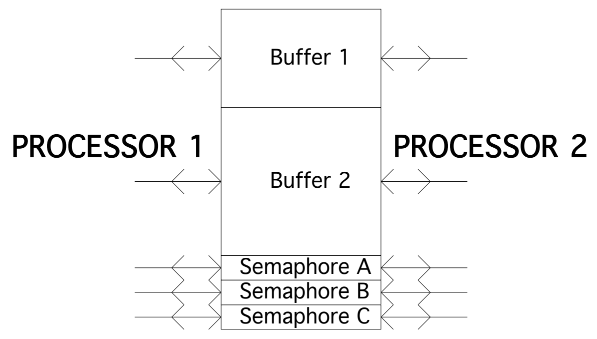


Fig.1 - Dual-port RAM

Power-up

Processor 1:	A,B,C free	get A and B	wait 1 second
Processor 2:	-”-	get C	wait 1 second

Repeated forever:

Processor 1:	owns A, B:	read, write buffer	free A		get C (poll)
Processor 2:	owns C:		get A (poll)	read, write buffer	free C
Processor 1:	owns B, C:	read, write buffer	free B		get A (poll)
Processor 2:	owns A:		get B (poll)	read, write buffer	free A
Processor 1:	owns C,A:	read, write buffer	free C		get B (poll)
Processor 2:	owns B:		get C (poll)	read, write buffer	free B

Fig. 2 - Processor states and data-flow

Each processor has the privilege 3 times, causing a maximum of 6 synchronous transmissions. Whenever a processor owns two semaphores it has the privilege. Whenever it wants to pass the privilege to the other processor, it frees the oldest owned semaphore.

Any scheme may be used for interpretation of the buffers. They may be ‘register-based’ or ‘protocol-based’, and data may be overwritten or not. Also, the scheme would protect any number of buffers.

The scheme is quite fast, as a processor will only have to free one semaphore and poll for the next between any communication. Alternatively there may be an interrupt when the next semaphore arrives. The communication will not deadlock because semaphores are acquired and released in correct order.

It was not as easy as we thought to come up with this very simple solution. A single semaphore alone cannot be used for this, as it only protects the data, it does not give any synchronization or direction-indication. A scheme with one semaphore only, once having released the semaphore, it could be acquired and received by *any* processor, including the processor that just released it. We found no way to implement the communication with two semaphores only. However, with three semaphores it is possible *not* to acquire and then release the same semaphore, nor acquire any two semaphores after each other, nor release any two semaphores after each other. The sequence of processor 1 is 'free-get-free-get-free-get' of semaphores A,B,C.

Power-up consistency between the two processors must be assured. This could be done by a simultaneous acquire of the needed semaphores and a time-out before the sequence starts.

It is easy to understand the system with an example of 2 persons wanting to share an ice-cream. They could use three balls to help them share it, one red, one blue and one green, all lying on the table initially. They have to agree on the ball color sequence (r-b-g-r-b-g..), and who gets the first lick. In order to lick the ice-cream, one has to hold two balls. Try it!

APPENDIX: Occam-2 code to simulate dual-port RAM and two processors

Occam-2 is a language that supports parallel processes through a PAR construct, making the real-time scheduler invisible and unreachable. The language is strongly typed and has a set of rules that, together with the lack of pointers and (not compile-time known) dynamic memory handling, make programming virtually waterproof. It is a small language which is easy to learn. It is based on the CSP-notation (Communicating Sequential Processes, a formal theory developed by C.A.R. Hoare) - as is the real-time parts of Ada. The main part of the program is seen below:

```
PROC Test.DualPort (CHAN OF SP fs, ts, []INT mem)
  ...  PROTOCOLS
  VAL Ticks.OneSec.LowPri IS 15625:

  INT            bufferOfDualPort:
  #PRAGMA SHARED bufferOfDualPort -- This breaks an occam rule

  VAL NoOfProcessors IS 2:
  VAL NoOfSema       IS 3:

  ...  PROC Delay
  ...  PROC DualPortRam
  ...  PROC Processor

  [NoOfProcessors][NoOfSema]CHAN OF Command command:
  [NoOfProcessors][NoOfSema]CHAN OF Reply   reply:
  SEQ
    bufferOfDualPort := 0
    PAR
      DualPortRam (command, reply)
      Processor  (0, [0,1], command[0], reply[0])
      Processor  (1, [2],   command[1], reply[1])
:
```

The code listing is *folded*. All (bold text) lines beginning with 3 dots are folds. Later on this fold *crease* is repeated as a heading at the place where the contents of the fold is present.

The dual-port ram's data space is simulated with a single INT, whose privilege to own is passed between the two processors. Occam already supports channels (CHAN) and protocols (PROTOCOL), all communication between parallel processes (PAR) is done via synchronous, unbuffered uni-directional channels. Occam does not have semaphores because shared resources are handled by process encapsulation in servers. Yet the whole purpose of *this* program is to simulate a shared buffer and semaphores. In order to make the shared buffer we had to break an occam rule with the `#PRAGMA SHARED` compiler directive.

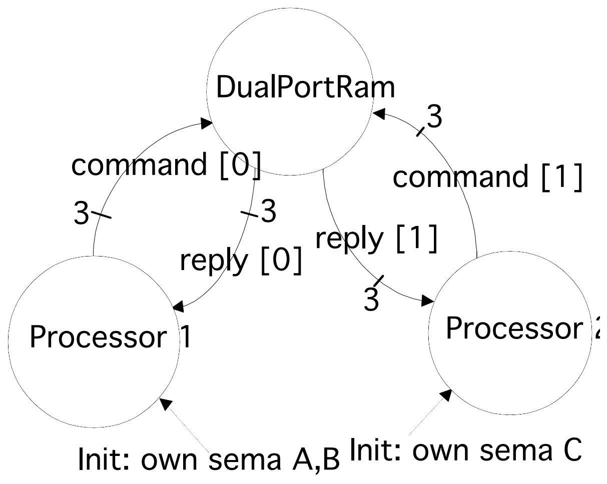


Fig. 3 - Command flow diagram

What we have already seen in the program listing is drawn as a command-flow diagram above. Each `processor` communicates with the `DualPortRam` via three command channels (one for each semaphore), and `DualPortRam` replies over three reply channels (one for each semaphore). This would correspond to a separate address for each query to a real dual port ram.

```
...  PROTOCOLS
PROTOCOL Command     IS BOOL:
VAL      AskForGrant IS TRUE:
VAL      ToRelease   IS FALSE:

PROTOCOL Reply   IS BOOL:
VAL      Granted IS TRUE:
VAL      Denied  IS FALSE:
```

The occam protocols are defined above. Both are *simple* protocols, but occam also supports *variant* protocols which are user-defined protocol formats.

Now let us look at the time aspect of occam. `TIMER` is a primitive data type, and the basic unit is a tick (1μs on high priority processes and 64μs on low priority processes). This procedure is needed for the optional time-delay.

```
...  PROC Delay
PROC Delay (VAL INT Ticks)
  INT   time:
  TIMER clock:
  SEQ
    clock ? time
    clock ? AFTER time PLUS Ticks
:
```

Now let us look at the `DualPortRam` code. The most interesting thing here is the `CHAN` parameters. Both command and reply are two-dimensional arrays of channels. The dimensions represent `processors` (2) and semaphores (3). The occam compiler assures that there is only one sender and one receiver per channel.

```
...  PROC DualPortRam
PROC DualPortRam ([][]CHAN OF Command command,
  [][]CHAN OF Reply reply)

  [NoOfSema]BOOL sema:
  VAL SemaFree  IS FALSE:
  VAL SemaInUse IS TRUE:
  SEQ
    SEQ i = 0 FOR SIZE sema
      sema [i] := SemaFree
    ...  Process processor commands and reply
:
```

Processing of `processor` queries is done below. Observe that the question mark (**?**) passively waits for data on a channel, and the exclamation mark (**!**) sends data over a channel whenever there is a receiver ready to receive the data (also see fig. 4):

```
...  Process processor commands and reply
VAL NextALT IS [1,0,1]:
INT processor:
SEQ
  processor := 0
  WHILE TRUE
    change := FALSE
    PRI ALT p = 0 FOR NoOfProcessors
      PRI ALT s = 0 FOR NoOfSema
        BOOL cmd:
        command [NextALT[p+processor]][s] ? cmd
          SEQ
            thisSema IS sema[s]:
            IF
              cmd = AskForGrant
                reply. IS reply [NextALT[p+processor]][s]:
                IF
                  thisSema = SemaInUse
                    reply. ! Denied
                  thisSema = SemaFree
                    SEQ
                      reply. ! Granted
                      thisSema := SemaInUse
              cmd = ToRelease
                thisSema := SemaFree
            processor := NextALT[p+processor] -- Fair scheduling of processors
```

This code implements a typical server, that sits idly waiting for a command coming from any `processor` (`PRI ALT p = 0 FOR NoOfProcessors`) for any semaphore (`PRI ALT s = 0 FOR NoOfSema`). The code actually implements waiting for 6 channels (2 by 3). It is the "`command [Next[p+processor]][s] ? cmd`" - line that is set up 6 times. The first command to be received is processed; if the semaphore is in use, a denial is replied; if it is free, it is granted and locked again. `DualPortRam` does not know which `processor` is using the semaphore, it only knows its binary state. Observe that decimal points in occam names mean nothing more than underscore in C names. (“`reply`” and “`reply.`” are two distinct names)

Whenever one `processor` has been served, the other `processor` is placed first in the `ALT`-queue of passive waiting. Without this explicit control of the `ALT` *fairness*, we had to introduce a delay in the `processors` so that they should not re-ask for a semaphore immediately. This would cause the *releasing* semaphore query never to be served. With the fair scheduling, the `processors` do not need this delay. No good system design should rely on inserted repeated delays.

Now let us look at the `processor` code:

```
...  PROC Processor
PROC Processor  (VAL INT IProc,
  VAL []INT Init,
  []CHAN OF Command command,
  []CHAN OF Reply reply)

  INT FUNCTION Prev (VAL INT This) IS ((This + (NoOfSema-1)) REM (SIZE command)):
  INT FUNCTION Next (VAL INT This) IS ((This + 1) REM (SIZE command)):

  INT iOfLastSema, noOfSemaOwned, myLastBufferValue:
  SEQ
    ...  Ask for initial semaphores
    ...  Init myLastBufferValue
    Delay (Ticks.OneSec.LowPri)
    ...  Repeatedly hold and release buffer
:
```

The semaphores are initialized according to the Init array:

```
...  Ask for initial semaphores
SEQ i = 0 FOR SIZE Init
  VAL I IS Init [i]:
  BOOL thisReply:
  SEQ
    command [I] ! AskForGrant
    reply [I] ? thisReply
    IF
      thisReply = Granted
        SKIP
      thisReply = Denied
        CAUSEERROR()
noOfSemaOwned := SIZE Init
```

After this the buffer value needs to be initialized:

```
...  Init myLastBufferValue
IF
  noOfSemaOwned = 2
    myLastBufferValue := 0
  noOfSemaOwned = 1
    myLastBufferValue := 1
```

And then comes the real processor code:

```
...  Repeatedly hold and release buffer
iOfLastSema := Init [(SIZE Init) - 1]
WHILE TRUE
  IF
    noOfSemaOwned = 1
      ...  Ackquire a second semaphore = receive buffer
    noOfSemaOwned = 2
      SEQ
        ...  Owns buffer: Read, increment, write and test buffer
        ...  Release first of two semaphores = send buffer
```

Below is the code that repeatedly asks for a second semaphore. If it is denied, it waits. As described earlier, this waiting is not needed. However, this time could be looked upon as the time when the processor is able to do other things than ping-pong the data back and forth.

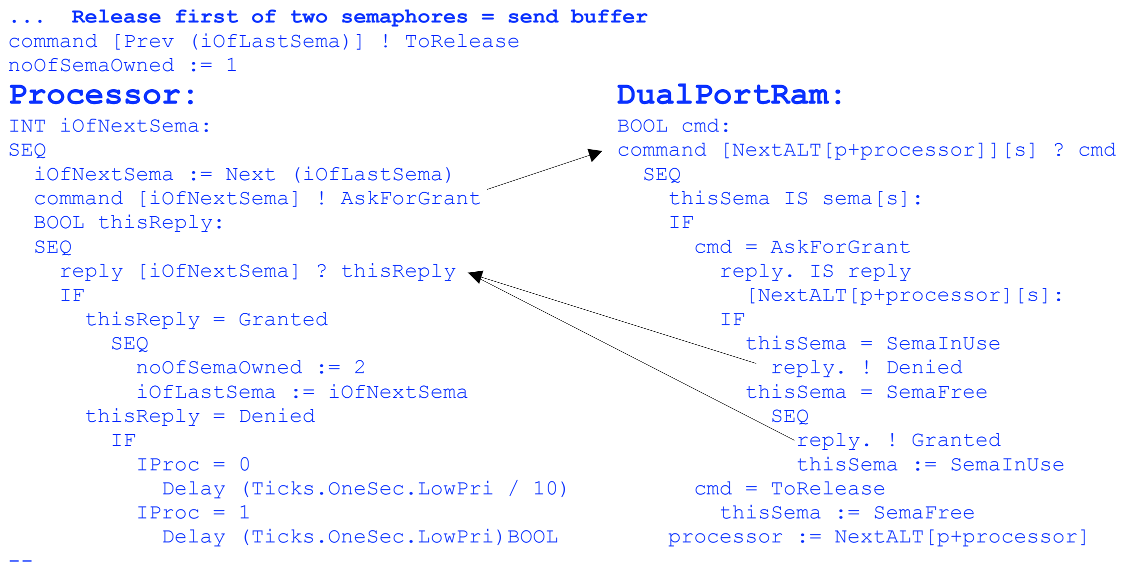
```
...  Ackquire a second semaphore = receive buffer
INT iOfNextSema:
SEQ
  iOfNextSema := Next (iOfLastSema)
  command [iOfNextSema] ! AskForGrant
  BOOL thisReply:
  SEQ
    reply [iOfNextSema] ? thisReply
    IF
      thisReply = Granted
        SEQ
          noOfSemaOwned := 2
          iOfLastSema := iOfNextSema
      thisReply = Denied
        IF
          IProc = 0
            Delay (Ticks.OneSec.LowPri / 10)
          IProc = 1
            Delay (Ticks.OneSec.LowPri)
```

As we can see in the code above, one processor has been given time to serve the DualPortRam once per second, the other 10 times per second. This means that the fastest processor will do 9 queries with a denial for each success. Full speed with no delay caused the buffer value to be incremented to 10000 in 3 seconds, including the original one second delay.

Whenever it owns 2 semaphores, the processor is able to do whatever it wants with the buffer. A system could handle several buffers through this (3-semaphore) scheme, and they could be assigned directions as well. With three buffers, there could be one for each direction (for command/reply) and one for bidirectional data ("register"-based). Our test program tests to see whether the other processor has incremented the value by 1, then it increments the value and sends it on.

```
...  Owns buffer: Read, increment, write and test buffer
IF
  bufferOfDualPort = myLastBufferValue
    SKIP
  bufferOfDualPort <> myLastBufferValue
    CAUSEERROR()
myLastBufferValue := bufferOfDualPort + 2
bufferOfDualPort  := bufferOfDualPort + 1
```

After this the program sends the buffer by releasing the oldest of the two semaphores:



```
...  Release first of two semaphores = send buffer
command [Prev (iOfLastSema)] ! ToRelease
noOfSemaOwned := 1
```

Processor:

```
INT iOfNextSema:
SEQ
  iOfNextSema := Next (iOfLastSema)
  command [iOfNextSema] ! AskForGrant
  BOOL thisReply:
  SEQ
    reply [iOfNextSema] ? thisReply
    IF
      thisReply = Granted
        SEQ
          noOfSemaOwned := 2
          iOfLastSema := iOfNextSema
      thisReply = Denied
        IF
          IProc = 0
            Delay (Ticks.OneSec.LowPri / 10)
          IProc = 1
            Delay (Ticks.OneSec.LowPri)BOOL
--
```

DualPortRam:

```
BOOL cmd:
command [NextALT[p+processor]][s] ? cmd
  SEQ
    thisSema IS sema[s]:
    IF
      cmd = AskForGrant
        reply. IS reply
          [NextALT[p+processor][s]:
        IF
          thisSema = SemaInUse
            reply. ! Denied
          thisSema = SemaFree
            SEQ
              reply. ! Granted
              thisSema := SemaInUse
      cmd = ToRelease
        thisSema := SemaFree
    processor := NextALT[p+processor]
```

Fig. 4 - Communication - an example

Some of the communication parts are illustrated above, where a part from `processor` and a part of `DualPortRam` have been placed adjacent to each other.

The code described is complete, has been fully tested and is working. Code to report to the screen has been stripped off.

The occam[1] code was tested on an SGS-Thomson transputer PC plug-in board. Occam is now also available to non-transputer users. A system called SPOC (Southampton Portable Occam Compiler) generates ANSI-C. Also, a compiler called KROC (Kent Retargetable Occam Compiler) now generates code that runs on DEC Alpha running O.S.F. 3.0 and SPARC running SunOS/Solaris systems. Occam may also run on PC's under a DOS extender. For further information try the www-sites below:

`<url:http://www.hensa.ac.uk/parallel/occam/documentation/>`
`<url:http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/kroc/>`

[1] Occam is a registered trademark of SGS-Thomson Microelectronics (previously Inmos)