


EDN[®]

THE DESIGN MAGAZINE OF THE ELECTRONICS INDUSTRY

JUNE 6, 1996

Table of Contents	pg 4
Out in Front	pg 19
Powerline communication	pg 71
3-D audio	pg 87
Design Ideas	pg 103
Whose fault is it anyway?	pg 127
Capacitor amplifier reduces ripple without dc loss	pg 137
DCT scaling enables universal MPEG decoder	pg 147
An event scheduler for control applications	pg 155
Ping-Pong scheme uses semaphores to pass dual-port-memory privileges	pg 179
Power Sources Showcase	pg 191
David Brubaker	pg 257
Jack Ganssle	pg 261
 DILBERT	pg 20

COVER STORY

**ANTENNAS:
CRITICAL LINKS IN
THE WIRELESS
SIGNAL CHAIN** pg 52

A CAHNERS PUBLICATION

<http://www.ednmag.com>
*Check us out
on the
internet*

Ping-Pong scheme uses semaphores to pass dual-port-memory privileges

OYVIND TEIG, AUTRONICA AS

Numerous schemes for passing data between processors and dual-port memory are possible. However, a Ping-Pong scheme uses just three semaphores, uses no state information inside the dual-port memory, and does not depend on time.

Each processor has the privilege three times, resulting in a maximum of six synchronous transmissions. A processor that owns two semaphores has the privilege. The processor that wants to pass the privilege to the other processor does so by freeing the oldest owned semaphore.

You can use any register-

During the last few years, the price of dual-port memory has dropped to a level that makes it feasible for use in embedded systems. Using a dual-port RAM seems attractive, but you must know how to use this RAM properly. The two processors, one on each side of the dual-port RAM, cannot just read and write to the dual-port RAM

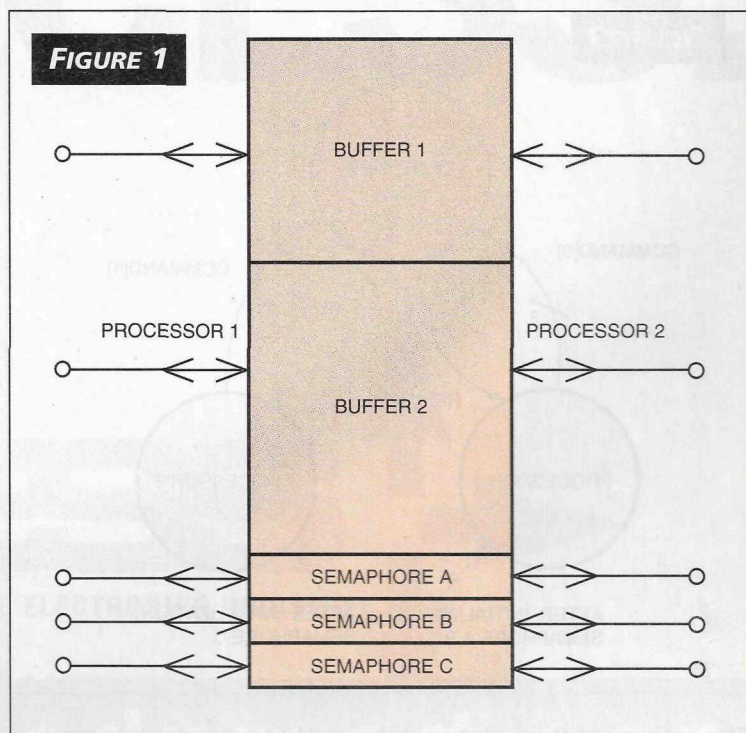
at any time. To help the designers handle this problem, most dual-port RAMs have internal semaphores. Semaphores are flags that only one processor at a time can own.

These semaphores are only basic building blocks. You must also implement a scheme that allows data to safely pass from one processor to the other. Several schemes are possible. In one scheme, the dual-port RAM itself could hold a state variable for use during the processors' arbitration. Another scheme is to guarantee that the other processor reads and modifies the data within a tight time limit.

However, a third possible scheme uses no state information inside the dual-port RAM area and does not depend on time. This scheme uses three of the usual eight hardware semaphores, and the two processors can differ in processing power and speed. The processors on each side pass through simple state machines with only one possible next state. This "Ping-Pong" approach lets a privilege continuously pass back and forth between the processors.

Ping-Pong scheme passes privileges

This scheme describes a privilege that passes between the processors (Figure 1). A processor can hold the privilege as long as it wants to. When a processor has the privilege, the processor is free to do whatever it wants with the buffers. The scheme uses three semaphores (A, B, and C) to pass the privilege. Figure 2 shows a complete privilege-passing sequence.



A Ping-Pong scheme uses three semaphores (A, B, and C) to pass dual-port-memory privileges between processors 1 and 2.

SEMAPHORES AND DUAL-PORT MEMORY

or protocol-based scheme to interpret the buffers, and you can or cannot overwrite data. Also, the scheme protects any number of buffers. The scheme is fast, because a processor has only to free one semaphore and poll for the next between any communication. The communication does not deadlock, because the scheme acquires and releases the semaphores in correct order.

This simple solution was not easy to derive. You cannot use a single semaphore alone, because it only protects the data and gives no synchronization or direction indication. In a single-semaphore scheme, any processor, including the processor that just released a semaphore, could acquire and receive a released semaphore. Similarly, you cannot implement the communication with only two semaphores. However, a scheme with three semaphores does not let a processor acquire and then release the same semaphore, acquire any two semaphores after each other, or release any two semaphores after each other. The sequence of processor 1 is "free-get-free-get-free-get" of semaphores A, B, and C.

You must assure power-up consistency between the two processors. You can accomplish this using a simultaneous acquire of the needed semaphores and a time-out before the sequence starts.

The following analogy helps to explain the scheme. Consider two people wanting to share an ice-cream cone. They

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

FIGURE 2

This sequence shows the passing of privileges between processors 1 and 2 using the Ping-Pong scheme.

could use three balls to help them share it, one red, one blue, and one green, all initially lying on a table. They have to agree on the ball color sequence (r-b-g-r-b-g, etc), and who gets the first lick. Whoever licks the ice-cream cone must hold two balls.

A simulation example

The following Occam program simulates this Ping-Pong scheme. (Occam is a registered trademark of SGS-Thomson Microelectronics, formerly, Inmos.) Occam-2 is a language that supports parallel processes, making the real-time scheduler invisible and unreachable. The strongly typed language has a set of rules that, with the lack of pointers and dynamic memory handling, make programming virtually fool-proof. This language is small and easy to learn. Like the real-time parts of Ada, Occam-2 is based on the CSP-notation (Communicating Sequential Processes, a formal theory developed by CAR Hoare). The following is the main part of the program:

```
PROC TestDualPort (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:

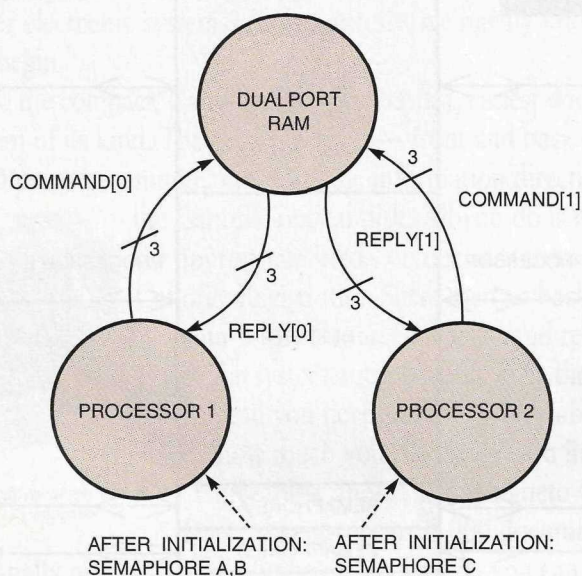
... PROC Delay
... PROC DualPortRam
... PROC Processor

[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 of the bold-faced text lines beginning with three dots are folds. This fold crease repeats as a heading at the place where the contents of the fold are present. Occam uses strict indenting of two spaces to define blocks of code.

A single INT, whose privilege to own passes between the two processors, simulates the dual-port RAM's data space. Occam supports channels (using the CHAN construct) and protocols (using the PROTOCOL construct). All communication between parallel processes (using the PAR construct) occurs over synchronous, unbuffered, unidirectional chan-

FIGURE 3



The command-flow diagram shows that each processor communicates with the dual-port RAM using three command channels. The memory replies to each processor using three reply channels.

SEMAPHORES AND DUAL-PORT MEMORY

nels. Occam has no semaphores, because process encapsulation in servers share resources. Yet, the purpose of this program is to simulate a shared buffer and semaphores. Thus, to create the shared buffer, you must break an Occam rule with the #PRAGMA SHARED compiler directive.

Figure 3 shows a command-flow diagram of the main program listing. Each processor communicates with the dual-port RAM through three command channels (one for each semaphore), and the RAM replies over three reply channels (one for each semaphore). This scheme corresponds to having a separate address for each query to a real dual-port RAM.

The following code defines the Occam protocols:

```
... 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:
```

Both are simple protocols, but Occam also supports variant protocols, which are user-defined protocol formats.

The following code shows the time aspect of Occam:

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

TIMER is a primitive data type, and the basic unit is a tick (1 μ sec on high-priority processes and 64 μ sec on low-priority processes). This procedure is necessary for the optional time delay.

The DualPortRam code is as follows:

```
... 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
;
```

The most interesting thing in this code is the CHAN parameters. Both command and reply are 2-D arrays of channels. The dimensions represent the two processors and three semaphores. The Occam compiler assures that there is only one sender and one receiver per channel.

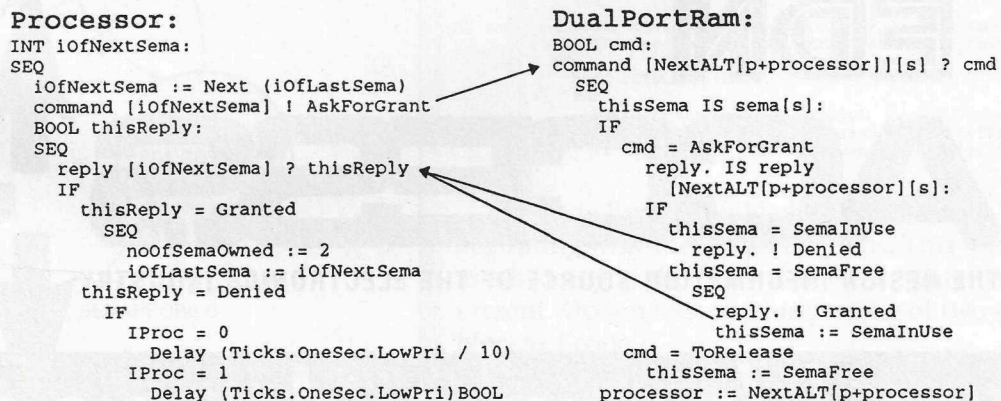
The following code handles the processor queries. Observe that the question mark (?) passively waits for data on a channel, and the exclamation mark (!) sends data over a channel whenever a receiver is ready for the data:

```
... 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
```

The above code implements a typical server, one that sits idly waiting for a command coming from any processor (PRI ALT p=0 FOR NoOfProcessors) and going to any semaphore (PRI ALT s=0 FOR NoOfSema). The code actually implements waiting for six channels (2 \times 3). The following command sets up six times: command [NextALT[p+processor]][s] ? cmd. The code processes the first received command. If the semaphore is in use, the DualPortRam replies a denial. If the semaphore is free, the DualPortRam grants the semaphore and relocks it. The DualPortRam does not know which processor is using the semaphore; it knows only the binary state. Note that decimal points in Occam names mean nothing more than an underscore in C names. For example, "reply" and "reply." are two distinct names.

Whenever one processor has been served, the other

FIGURE 4



Side-by-side Processor and DualPortRam code show the essence of the Ping-Pong communication scheme.

SEMAPHORES AND DUAL-PORT MEMORY

processor is placed first in the ALT queue of passive waiting. Without this explicit control of the ALT fairness, you must introduce a delay in the processors, so that they can't immediately ask again for a semaphore. This repeat query 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.

The following is 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 following 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
```

The buffer value now needs to be initialized:

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

The real processor code comes next:

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

The following code repeatedly asks for a second semaphore. If the DualPortRam denies, the code goes into a waiting mode. This waiting is unnecessary. However, you can look upon this time as time when the processor can do things other than Ping-Pong the data back and forth.

```
... Acquire 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 you can see in this code, one processor has time to serve the dual-port RAM once/sec; the other, 10 times/sec. This means that the fastest processor will perform nine queries with a denial for each success. Full speed with no delay causes the buffer value to increment to 10,000 in 3 sec, including the original 1-sec delay.

Whenever a processor owns two semaphores, the processor can do whatever it wants with the buffer. A system could handle several buffers through this three-semaphore scheme and could also assign directions to the semaphores. With three buffers, there could be one semaphore 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 buffer's value by 1 and then 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
```

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
```

Figure 4, which shows adjacent Processor and DualPort-Ram code, illustrates some of the communication elements.

All of the code is complete, is fully tested, and is working. (Code to report to the screen has been stripped off.) The Occam code was tested on an SGS-Thomson transputer PC plug-in board. Occam is now also available to nontransputer 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 a Digital Equipment Alpha running OSF 3.0 and a SPARC running SunOS/Solaris system. You can also run Occam on PCs under a DOS extender. For further information, try the following www sites: <url:http://www.hensa.ac.uk/parallel/occam/documentation/> or <url:http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/kroc/>. EDN

Author's biography

Oyvind Teig is a senior development engineer at Autronica As (Trondheim, Norway). He works on the design and programming of real-time systems and holds an MSC degree from the Norwegian Institute of Technology.

VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest
570

Medium Interest
571

Low Interest
572