

Re: Shared Memory for Application/Communication decoupling

Source: <http://coding.derkeiler.com/Archive/General/comp.arch.embedded/2006-12/msg00573.html>

- *From:* "Steve at fivetrees" <steve@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Mon, 11 Dec 2006 20:49:35 -0000
-

"ChrisQuayle" <nospam@xxxxxxxxxxxxx> wrote in message
[news:bLgeh.1064\\$KT2.493@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:news:bLgeh.1064$KT2.493@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx)

Christian Walter wrote:

Of course each IPC method brings in its own piece of code and using different IPC methods within a single task will introduce a lot of complexity.

Thanks a lot for sharing. Still I can't agree on this point because I think the interface requirements are quite different. For example look at other systems where different IPC mechanisms are used when appropriate (EJB – Enterprise Java Beans with Locale and Remote Interface to name only one). Maybe this would be a interesting poll for the ESD magazine – "Type of IPC mechanisms used today".

Kind regards,
Christian Walter

I don't want to get into arguments about semantics, but in general, shared memory techniques are just a nice way of saying 'global data'. If you have > 1 writer, you have to do all the synchronisation and mutual exclusion at low level, where it's difficult to maintain visibility and keep track of.

A messaging based interface is generally more elegant and arguably more reliable, since for any single message, only one task writes and one task reads the data. This means that you can elevate synchronisation issues to a higher, task based level. Message based ipc reinforces an object view of the design at global level.

On modern micros, the overhead is probably insignificant as well. Shared memory / global data is a thing of the past, from the days when micros

Re: Shared Memory for Application/Communication decoupling

were not very powerfull...

I agree fully with Chris's remarks – I'd avoid global data like the plague.

However: I've also done fieldbus work, and I'm aware that one way of mirroring the states of the numerous parameters on numerous controllers is to use shared memory. I don't like this method myself – see above re global data. I much prefer a set of routines for accessing or changing parameter(PARAMETER_NAME) on controller(index). The usual arguments against this include a) different data types and b) overhead of the access methods – personally I prefer to solve these issues rather than fall back on global data. That way be dragons.

However – if you're *really* stuck for memory, you might consider a block of shared memory as a part of the messaging interface – so long as you forbid your coders from bypassing the messaging interface on pain of death. Having distinct set/read routines is still essential.

BTW: I'm assuming that your comms code builds a mirror of the states of the remote controllers, and your application layer uses the mirrored data. If you interrogate on demand, then the memory issues go away – but the application is kinda slow...

HTH,

Steve

<http://www.fivetrees.com>

.