WHITEPAPER

Breaking up is hard to do

Dennis Kelly
Field Engineering Specialist

QNX Software Systems

2/24/16

Breaking up is hard to do

Overview

Contrary to the title, this paper is not about romance. It is about embedded software design and how to avoid professional heartache.

It's a fact of life – embedded systems need human interaction. With some systems, so much work is invested in the HMI that it's easy to lose sight of the fact that the system exists for the purpose of performing work!

This paper will deal with life-cycle issues of embedded systems requiring an HMI.

Case study

> *XYZ Corp makes a controller for an industrial widget maker. The design document calls for three main menu options, each with additional submenus and functional screens. All told there are 30+ screens to be implemented.*
>
> *Engineering is leaning towards a Qt implementation. The project lead favors C++ because of its strict type-checking. Qt provides the necessary support for building menus and displaying the required graphics screens. Qt includes libraries for accessing required serial ports and networking. Plus, Qt code can create threads and directly make other system calls – so it is easy to directly access the controller from the HMI.*

Sounds like the perfect approach, right? Not exactly. This monolithic design will prove problematic during initial implementation, and even more so later in the lifecycle. We will begin by dealing with initial development issues. Later we will discuss issues as the product matures.

Looking at the case study, you have to wonder, what is the purpose of the design? Remember - it is the back-end controller software that is the heart of your product – not the HMI. For reasons that will become apparent as the discussion progresses, the best design keeps this back-end independent of the HMI. Your HMI strategy should not be forcing the development path for the back-end.

Different tasks, different skills

There are multiple reasons for keeping the back-end truly independent. The first is efficient division of labor.

- Developers of back-end, low-level software may not have the training to work on an HMI.
- Likewise, developers of HMI software typically don't have the skills required for low-level work.

In the proposed all-in-one design, you are unnecessarily forcing back-end developers out of their efficiency zone. For example, the back-end programmer is being forced to:

1) Conform his back-end design to somehow fit inside the HMI binary

2) Use the HMI toolchain rather than a traditional familiar makefile

3) Always build the complete HMI package without error before any testing of the back-end

4) Have all features of the HMI completed before he can efficiently test the back-end

The third and fourth reasons have a major impact on the development cycle.

- It is typical that the HMI lags on the development timeline. HMI often has more people involved and is more subject to change due to the subjective nature of an HMI. However with a monolithic design, the HMI's slow progress and regressions make it difficult for the back-end developers and testers.

- Experience shows that there are frequent periods during development when a project codebase cannot be successfully built due to pending changes and developer check-in errors. When the HMI baseline contains the back-end, this will unnecessarily waste developer time.

- Further, how can back-end features be tested when there is no HMI to control them? Again, typically the back-end team wastes time waiting for the HMI to catch up.

An additional problem as implementation progresses, is that given a monolithic codebase, it is too easy for HMI developers to take shortcuts by modifying the back-end code. When this is allowed to happen without the knowledge of the back-end designer, it can easily lead to deviations from critical design specifications. Such issues can be resolved during testing, but this leads to unnecessary delays while corrections are made.

Interprocess communication

So far, we have illustrated some advantages of decoupling the HMI from the back-end logic. However, two questions remain:

- How do the two programs communicate as a finished product?

- How can testing proceed independently for each program?

Operating systems provide various methods known as inter-process communication (IPC) for exchanging data between independent processes. Each process will have its own protected memory space, so there can be no direct contact. This is an advantage over monolithic designs, since one process cannot corrupt the memory of the other.

We will only consider methods which are event-driven without the need for any polling – an absolute requirement for embedded systems.

One method of communicating between processes is to setup a shared memory region with a synchronization object like a mutex or semaphore. Shared memory is not the cleanest of methods since both processes typically have write access to the same memory and programming discipline is required to avoid corruption. However, this may be the only practical method when very large blocks of memory must be shared.

Other methods of IPC may require a connection. This would include message-passing (as in Windows and QNX) or a socket connection (via localhost). These both typically imply a client-server model.

Connection-oriented methods have the requirement that both client and server know the connection status of its counterpart. This adds extra code and complexity to the software – especially error handling.

Any of these methods will allow communication between processes, but they require you to write and maintain throw-away code to independently test the server without the client, or vice versa.

Publish–subscribe paradigm

A third type of inter-process communication which satisfies the requirement for independent testing of HMI and back-end modules is known as publish-subscribe (or pub/sub). This paradigm is considered connectionless because there is no direct connection between the publisher (producer) of data and the subscriber (consumer) of data. Rather, a broker program maintains a set of published data objects and allows any number of subscribers to request access to the data objects.

With this loosely-coupled paradigm, servers do not have to handle connect and disconnect of their clients. Connections are the responsibility of the standard broker program. The server need not know (or care) whether or not any client is connected – it simply faithfully publishes the data!

When a server, (such as the back-end process) is publishing data to a pub/sub object, it is possible to monitor the process by subscribing to the data object. That may sound like it requires API calls (programming), but a goal of pub/sub is to provide a very simple interface so that virtually any type of client may become a subscriber.

In the QNX embedded operating system, the pub/sub broker is a daemon called pps. Pps maintains published data objects as filesystem objects having text-based contents. This methodology makes subscribing very easy – just open and read! In fact, the shell utility cat can become a subscriber, receiving data only when the object is updated by the publisher. An example is in order.

Let's imagine a simple case where the server publishes the time-of-day once per second. The shell command to make cat a pps subscriber is shown below with the output showing the published data.

```
# cat /pps/myDevice/serverData?wait | grep time::

time::Tue Aug 25 11:12:00 EDT 2015

time::Tue Aug 25 11:12:01 EDT 2015

time::Tue Aug 25 11:12:02 EDT 2015

^C
```

Note the special characters “?wait” appended to the filename of the data object. A pps pub/sub object opened in this way will return data whenever the object changes. In other words, the subscription is event-driven and requires no polling.

It is important to note that the filesystem object /pps/myDevice/serverData is not a regular file! It does not grow in length when the publisher writes to it. Rather, the pps implementation is a data overwrite, which triggers all subscribers to wake with the latest data.

Likewise, utility echo can be used to generate publisher data as well. The following simple shell script simulates the behavior of the server. (Note, it is required that directory /pps/myDevice pre-exist.)

```
# while true
> do
> sleep 1
> mytime=$(date)
> echo time::$mytime  >>/pps/myDevice/serverData
> done
```

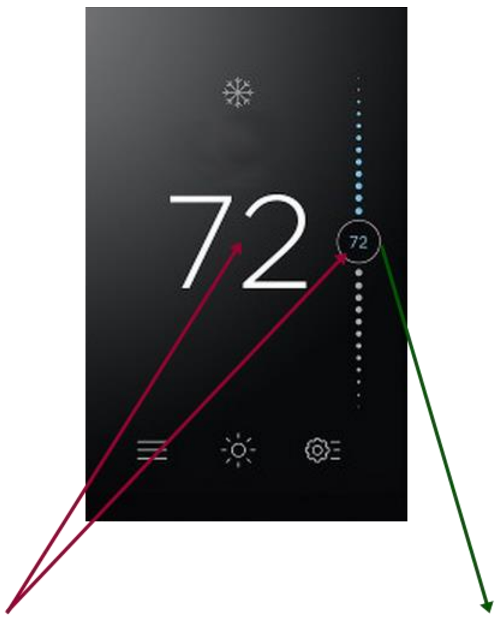
With the above simple shell scripts, we have simulated publishing data, and created an event-driven subscriber. To try the example, run the publisher first to create file object /pps/myDevice/serverData. Then, from a second shell (perhaps telnet or ssh), run the subscriber script. If you open additional ttys, the data in all the subscriber sessions will be in sync. There is no practical limit to the number of subscribers.

For testing purposes, if the HMI maintained an output field for time of day, the publisher script above could be used to verify a proper HMI update. In this way we have removed the dependency on the HMI of the back-end server. Of course this only simulates one piece of the data set used by the back-end, but a program with simple printf’s can generate other data published by the back-end.

The question now arises, how does the back-end receive user input from the HMI? And, how is it easily simulated without actually having a completed HMI?

The answer is that the back-end, in addition to publishing data for the HMI, also subscribes to data published by the HMI. Likewise, the HMI is not just a subscriber, but also a publisher of user actions.

In the case study above, one of the screens is a very simple thermostat. There are two numbers displayed by the HMI (measured temperature and target temperature), plus a slider for the user to operate.



Current temperature and target temperature user are published by back-end

Slider position changes due to published by HMI

So, to generate slider movement for the back-end to test its response, a simulated publisher script might contain:

```
# echo sliderPosition::73  >>/pps/myDevice/hmiData
# sleep 1
# echo sliderPosition::76  >>/pps/myDevice/hmiData
# sleep 1
# echo sliderPosition::75  >>/pps/myDevice/hmiData
# sleep 1
```

Effectively, the back-end would perform processing equivalent to this script, watching for changes in **sliderPosition** within the **hmiData** object.

```
# cat /pps/myDevice/hmiData?wait | grep sliderPosition::
sliderPosition::73
sliderPosition::76
sliderPosition::75
```

We now have isolated the back-end from the HMI, provided a strict interface specification via the pps objects, demonstrated completely event-driven interprocess communication, and provided a simple method of simulating a datastream from the opposite component.

It should be noted that while this example used a simple name::value text format, we could just as easily have published the data as a standard JSON object or XML document. Using one of these standard formats in your publisher may facilitate the development of your HMI subscriber.

Assigning blame

Not only does the isolation provided by pub/sub make development easier, it is also important when debugging issues. Quickly isolating issues to a particular module and development team can reduce schedule impact and friction between developers.

1) It is pretty easy to establish whether or not correct data is being published by a component. Once you can determine that, the problem has been bisected – you know to look for the issue with the publisher or the subscriber .

2) Excess CPU utilization is much easier to track compared to a monolithic approach. The problem is readily bisected by observing CPU utilization for each independent process.

Avoiding mid-life crisis

It is a certainty that successful products eventually require updating. This typically means some incremental changes in function, plus ease-of-use improvements. In today's terms, the latter implies

- a fresh new face, and
- "access anywhere" connectivity

Both of these can be especially problematic for a monolithic design. If your back-end logic is tightly coupled to your HMI, changing to a different graphic system to achieve a newer look-and-feel can involve a total rewrite!

Additionally, many HMI systems (including Qt) do not readily support web viewing.

Your company has invested man-years of effort in developing and verifying your monolithic product. Is management willing to re-invest a similar amount of effort just to add modern connectivity options? Or, will your product be forced to do without changes required to remain competitive… and eventually wither away?

It is at this stage of the product life-cycle where the clean delineation of back-end/HMI again pays big dividends! At this point, the product engineers have faith in the pub/sub solution and realize that changes in the HMI "face" can be implemented without impacting product functionality. Further, the incremental back-end functionality changes are understood to be truly incremental.

Let's further consider the case of adding a mobile interface to your product. Obviously, the primary HMI cannot be run on a phone or tablet – but your product has no dependence on that

HMI! You will need to produce a new HMI for the mobile app plus a socket server on your embedded target. Fortunately,

- pub/sub lets you freely add an app server as a subscriber to back-end published data, and
- pub/sub also allows multiple processes (two differing HMI's) to publish to data objects

It will be the job of the mobile app server to subscribe to the back-end and forward all status changes to the mobile app, and conversely to publish received mobile app user events to the back-end. Since the pub/sub interfaces are well known, this server is not hard to write. In fact, the verbatim subscriber data from the object can be forwarded on the socket, improving the likelihood that the local and remote HMI's can share some source code.

In practical terms, this means that

- both the local interface and mobile interface will always present the same data
- both the local interface and the mobile interface can control the target, and
- multiple mobile apps will be able to connect simultaneously (when the server is so written)

Conclusion

The case for creating separate processes for HMI and back-end of embedded systems has been presented. The pub/sub paradigm has been shown to be good method for maintaining a strict, well-documented interface between components. Pub/sub is particularly well suited when HMI technologies change and mobile apps become a requirement.