Design Decoupling in a Message-Passing Environment

Authored by: **Rob Krten**
Updated by: **Thomas Fletcher**

That QNX Neutrino is a message-passing microkernel is one thing. Recognizing the true benefit of this when it comes time to design your system is quite another. For a standard UNIX-like application environment, like an HTTP web server or application logger, this understanding is irrelevant. The value of message-passing architecture becomes important in the design of an entire system. A common issue that arises is that of design decoupling, and it becomes apparent when you ask the following questions:

How much work should one single process do?, i.e. where do I draw the line in terms of functionality?

1. How do I structure drivers for my hardware?
2. How do I create a large system in a modular manner?
3. Is this design future proof, i.e. will it work two years from now when my requirements change?

Here are the short answers, in order:

1. A process should focus on the task at hand, leaving everything else to other processes
2. Drivers should be structured to present an abstraction of the hardware
3. To create a large system, start with several small, well-defined systems and 'glue' them together

If you've followed the above, you'll have a system consisting of reusable 'building block' components that you should be able to rearrange, reuse, and add to in the future

Applying this to the real world

The goals

Let's look at each of these issues using a hypothetical situation. You are the software architect for a security company, and you've been tasked with creating the software design for a new security system. The hardware consists of swipe-card readers, door-lock actuators, and various sensors (smoke, fire, motion, glass-break, and so on).

Your company has decided that they want to build products for an entire range of markets - a small security system suitable for a home or a small office, up to gigantic systems for large, multi-site customers. Furthermore, they want their systems to be upgradeable so that as a customer's site grows, they can simply add on more devices without having to throw out the original system. Finally, whether the configuration is small or large, the security system should be able to support any of the devices. For example, a small configuration should be able to support devices more typically used in a larger configuration, and vice versa.

The plan

Your first job is to sketch out a high-level architectural overview of the system and decide how to structure the software. Working from your stated goals, the implied requirements are that the system must be capable of supporting various numbers of each of the above devices, distributed across a range of physical areas, and it must be 'future proof' so that you can add new types of sensors, output devices, and so on in the future as your requirements change or as new types of hardware become available (e.g. retinal scanners).

So, the first step is to define the functional breakdown, or determine how much work one single process should do.

If you step back from the security example for a moment and consider a database program, you see some of the same concepts. A database program manages a database - it doesn't worry about the type of medium that the data lives on, nor is it (generally) worried about the organization of that medium, or the partitions on the hard disk, etc. It certainly doesn't care about the SCSI or EIDE disk-driver hardware. In such a system, the database makes use of a set of abstract services supplied by the file system component. As far as the database is concerned, everything else is opaque; it doesn't need to see further down the abstraction chain. The next level, the file system component, makes use of a set of abstract services from the disk driver. Finally, the disk driver controls the hardware. The obvious advantage of this approach is that because the higher levels don't know the details of the lower levels, you can substitute the lower levels with different implementations, provided that the well-defined interfaces are maintained.

If you consider this in light of the security system, you can immediately start at the bottom - the hardware. You know that you have different types of hardware devices, and you know that the next higher level up in the software hierarchy probably doesn't want to know the details of the hardware. So, your first step is to 'draw the line' at the hardware interfaces, meaning that you'll create a set of device drivers for the hardware, and provide a well-defined API for the higher levels of software to use.

One such higher level of software will most likely be some kind of control application. For example, it'll need to verify that Mr. Smith actually has access to door number 76 at 06:30, and, if that's the case, will allow him to enter. Already you can see that the control software will want to have access to a database (for verification and, perhaps, logging), will need to be able to read the data from the swipe-card readers (to find out who's at the door), and will need the ability to control the door locks (to open the door).

Once you have these two levels defined, you can sort out the interface. Since you've analyzed the requirements from the higher level, you now know what shape the interface will take on the lower level. For the swipe-card readers, you're going to need to know that a) someone has swiped their card at the card reader, and b) which card reader that was, so that you can determine the person's location. For the door-lock actuator hardware, you're going to need the ability to allow the door to be opened, and prevent the door from being opened (i.e. so that Mr. Smith can get in, but the door locks after him).

This system should scale - that is, it should operate in a small business environment with just a few devices, right up to a large 'campus' environment with hundreds (if not thousands) of devices.

When you need to analyze a system for scalability, you're looking at the following issues:

- Is there enough CPU power to handle a 'maximum' system?
- Is there enough hardware capacity (PCI slots, etc.)?
- How do you distribute your system?
- Are there any bottlenecks? Where are they, and how can you get around them?

As you'll see, these scalability concerns are very closely tied to the way you've chosen to break up your design work. For example, if you put too much functionality into a given process (say you had one process that controlled all the door locks), you're limiting how you can distribute that process across multiple CPUs. At the other end of the spectrum, if you put too little functionality into a process, say, for instance, you had one process per function per door lock, you run into the problem of excessive communications and overhead.

So, keeping these goals in mind, let's look at the design for each of your hardware drivers.

Door-lock actuators

Let's start with the door-lock actuators since they're the simplest to deal with. The only thing that you want this driver to do is to allow a door to be opened, or prevent one from being opened. This naturally dictates the form that the commands to the driver will take. You need to tell the driver which door lock it should manipulate and its new state (locked or unlocked). For the initial prototype of the software, those would be the only two commands that I'd provide. Later, to offload processing and to give you the ability to support new hardware, you might consider adding more commands. However, suppose you have two different types of door-lock actuators. One is simply a door-release mechanism that when 'active' means the door can be opened and when 'inactive' means the door can't be opened. The second is a motor-driven door mechanism that when 'active' means the motor turns on, causing the door to swing open and when 'inactive' means the motor releases the door, allowing it to swing closed. Initially, these might look like they're two different drivers with two different interfaces. However, they can be handled by the same interface (but not, however, the same driver). All you really want to do (at a high level) is allow someone to go through the door. Naturally, this means that you'd like to have some kind of timer associated with the door opening. When access has been granted, you'll allow the door to remain unlocked for, say, 20 seconds. After that point, you lock the door. For certain kinds of doors, you might wish to change the time - longer or shorter - as required.

A key question that arises is, where to put this timer functionality. There are three possible places where it can go:

- The door-lock driver itself
- A separate timing-manager driver that then talks to the door-lock driver
- The control program itself

This comes back to issues of design decoupling (and is related to scalability).

At one end of the spectrum, if you put the timing functionality into the control program itself, you're putting additional workload into the control program. Not only does the control program have to handle its normal duties (database verification, operator displays, etc.), it now also has to manage timers. For a small, simple system, this probably won't make much difference. However, once you start to scale your design up into a campus- wide security system, you'll be incurring additional processing in one central location. Whenever you see the phrase 'one central location,' you should immediately be looking for scalability problems. There are a number of significant problems with putting the functionality into the control program:

- **scalability** - the control program must maintain a timer for each door; the more doors, the more timers
- **security** - if the communications system fails after the control program has opened the door, you're left with an unlocked door
- **upgradeability** - if a new type of door requires different timing parameters (for example, instead of just a simple 'lock' and 'unlock' command, it might require multiple timing parameters to sequence various hardware), you now have to upgrade the control program

The short answer here is that the control program really shouldn't have to manage the timers. This is a low-level detail that's ideally suited to being offloaded.

Let's consider the next point. Should you have a process that manages the timers, which then communicates with the door-lock actuators? Again, I'd answer, no. The scalability and security aspects of the points raised above don't apply in this case (you're assuming that this timer manager would be distributed amongst the various CPUs, so it scales well. And, since it's on the same CPU, you can eliminate the communications failure component). The upgradeability aspect still applies, however. But there's a new issue - functional clustering. What I mean by this is that the timing function is tied to the hardware. You might have dumb hardware where you have to do the timing yourself, or you might have smart hardware that has timers built directly into it. In addition, you might have complicated hardware that might require multi-phase sequencing. By having a separate manager performing the timing function, it now has to be aware of the details of the hardware. The problem here is that you've split the knowledge about the hardware across two processes, without gaining any advantages. On a file-system disk driver, for example, this might be similar to having one process responsible for reading blocks from the disk while another one is responsible for writing. You're certainly not gaining anything and, in fact, you're complicating the driver because now the two processes must coordinate with each other to get access to the hardware.

That said, there are cases where having a process that's between the control program and the individual door locks makes sense. Suppose that you wanted to create a 'meta' door- lock driver, for some kind of complicated, sequenced door access (for example, for a high-security application, where you want one door to open, and then completely close before allowing access to another door, and so on). In this case, the meta driver would actually be responsible for cycling the individual door-lock drivers. The nice thing about the way that you've structured your devices is that as far as the control software is concerned, this meta driver looks just like a standard door-lock driver; the details are hidden by the interface.

You're not finished yet!

Your security system is by no means complete. You've only looked at the door locks and illustrated some of the common design decisions that must be made. Your goals for the door-lock driver were for the process to do as much work as was necessary to provide a clean, simple, abstract interface for higher levels of software to use. You used functional clustering (guided by the capabilities of the hardware) when deciding which functions were appropriate to put into the

driver, and which functions should be left to other processes. By implication, you've seen some of the issues associated with constructing a large system as a set of small, well-defined building blocks which could all be tested and developed individually, leading naturally to a system design that will allow new types of modules to be added seamlessly.