Design Patterns for Device Driver Design

Sachin Bammi
Software/Firmware Engineer
sbammi@slb.com
Schlumberger Technology Corporation (www.slb.com)

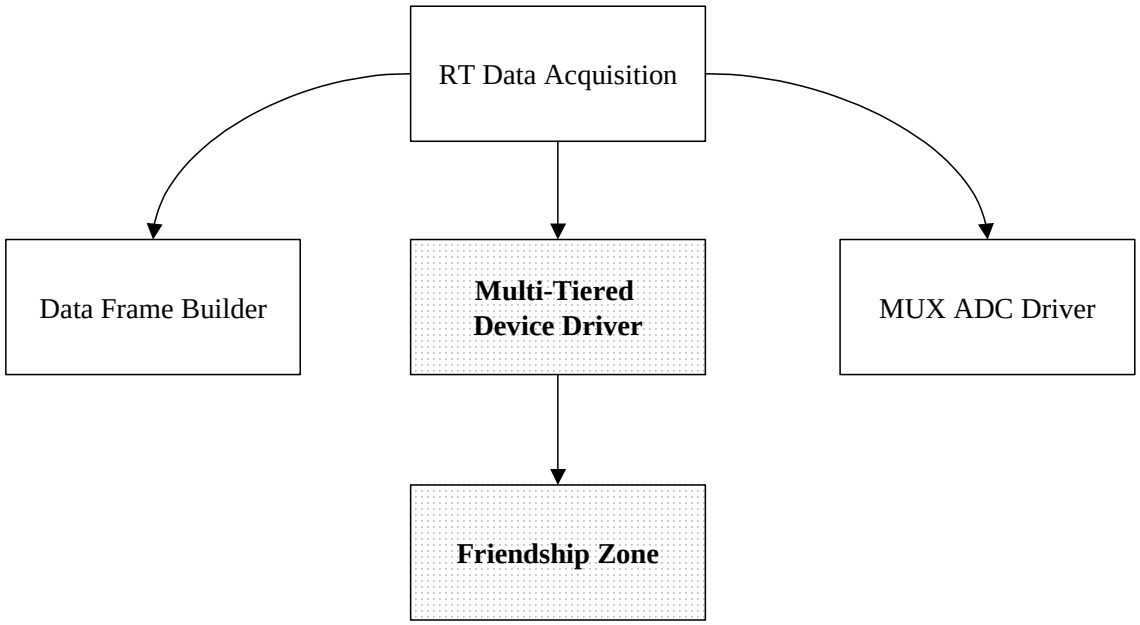
Abstract:
This paper presents two design patterns on designing and developing device drivers which balance the opposing forces of data encapsulation, system efficiency and managing change in software due to change in business and technical requirements over the course of a project. It ends by providing a real life example of how to apply them to a serial communication protocol driver.

Introduction

Device drivers are all pervasive in the embedded software/firmware world. They form a critical part of the low-level code on which typically most of the embedded real time applications are based on. Hence getting them done right is of paramount importance.

The patterns presented in this paper aim at providing general architecture specific guidelines for developing device drivers. The patterns would eventually form a part of a pattern language being developed by the author for developing real time applications, which drive drilling electronics in harsh environmental conditions. While the pattern language that develops due to this effort will be rather specific in nature, these individual patterns would have a more general appeal. The following figure presents the most current vision of the author for the aforementioned pattern language henceforth called "Down Hole Firmware Pattern Language".

Figure 1: Down Hole Firmware Pattern Language version 1.0



Only the shaded design patterns i.e. "Multi-Tiered Device Driver" and "Friendship Zone" in Figure 1 are introduced in this paper. Others are work in progress. The paper also presents a real life sample implementation in C++ for both of them as applied to a serial communication protocol driver.

Pattern: Multi Tiered Device Driver

Context
In the world of embedded systems, device drivers have been written for a long time now. This is an architectural pattern for designing a device driver by giving an example of a serial communication driver.

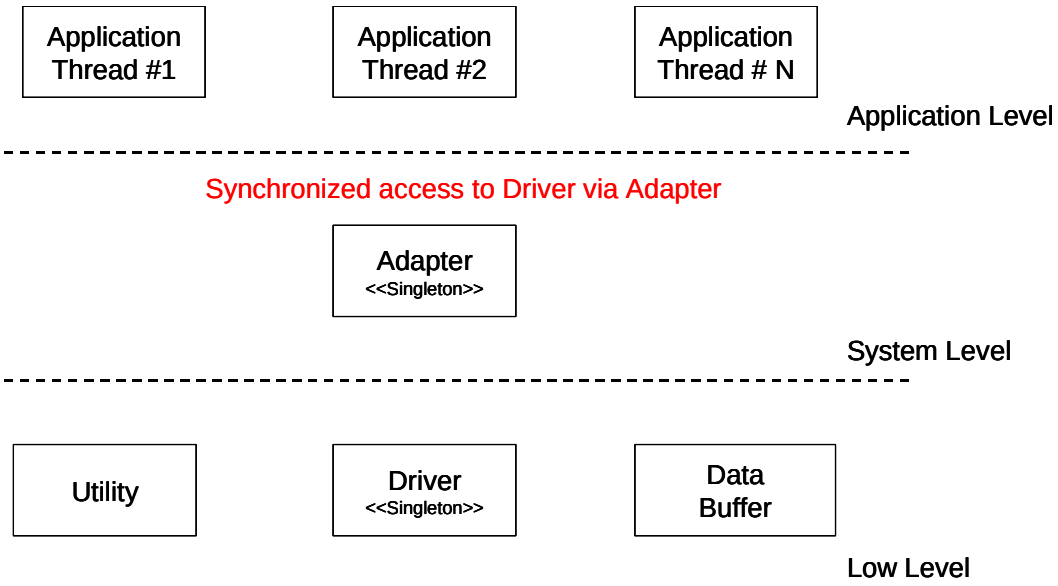
Problem
An important challenge in developing device drivers is to keep the design flexible. This helps in making any future changes/upgrades in hardware or the business logic in the real-time application, which uses the driver, easily without affecting too much the components of the driver. However this flexibility comes at the price of code bloat and performance efficiency. Hence the problem is to find the right trade-off.

Forces
During the development phase of a project there is always a chance of requirements getting changed on the business logic side and the need to make the code generic enough so that it can be ported to other future hardware upgrades. This presents a challenge for the software/firmware engineer to accommodate for these possibilities in the design on one hand by grouping things that could change together while avoiding code complexity, code bloat and system inefficiency on the other. For greater flexibility in design one has to group things that typically change together by creating different layers of abstraction, but this in turn can slow down the system because of increased number of function calls through different layers. Hence an optimum number of abstractions need to provided so that a balance is reached between design flexibility and system efficiency in real time systems.

Solution
Design a multi tiered architecture that divides the device driver code into the three abstractions or groups: Application level, System level and Low level. If the hardware changes then the code should be modified only at the Low level or conversely if the business requirements change then only the application code changes. The system level code provides access functions to the low level code for the application level code. The application level code cannot directly call the low level code. This way we can achieve the aim of grouping code that typically changes together. This architecture is shown in the Figure 2.

Resulting Context
The code is divided into three layers so that the business logic is separated from the low level hardware specific code and with System level providing the necessary bridge in between. However the design can still be re-factored to make it more efficient especially at the Low level where the time sensitive interrupt handling occurs. The following pattern aims to address this deficiency.

Figure 2: Multi Tiered Architectural pattern for device driver design



Pattern: Friendship Zone

Context
The previous pattern showed how the architecture of the device driver could be designed so that we could have flexible design to accommodate potential future changes in business requirements and hardware. However, there are still system inefficiencies related to fast data access especially for code at the System level and Low level. This is especially a concern for parts of the code that service interrupts. Due to data encapsulation at the Low level additional function calls have to be made to access data members of other classes. This brings in additional latency and code bloat into the system and may be unacceptable for several real time systems.

Problem
How to make the three tiered architectural pattern presented earlier more efficient by providing faster data access at the System level and Low level.

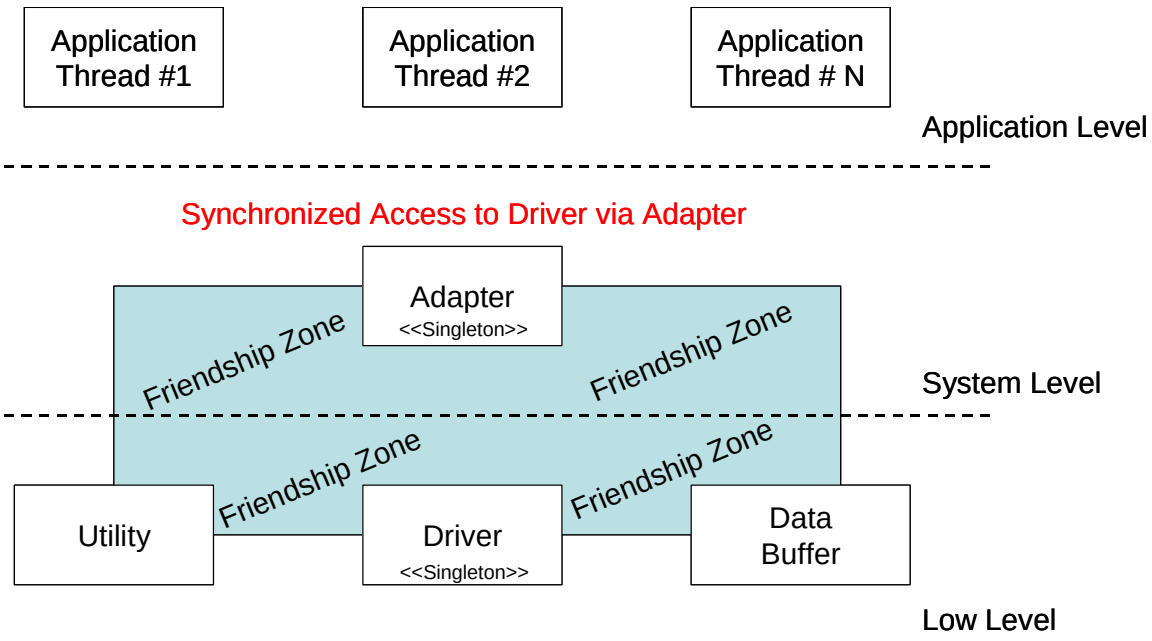
Forces
From a truly data encapsulation point of view each class/module should protect its data by either keeping it private or providing the appropriate access functions. However for time-critical and space starved real time embedded systems this could be a concern because of additional time taken to make a function call and the code bloat due to additional data access functions.

Hence we need a pattern that addresses all the above issues for it to be successfully applied to the design of a device driver which has to be efficient, not take too much code space and at the same time be modularized enough that future changes to the code in one component of it can be made easily without affecting the other parts.

Solution
Balance the opposing forces of data encapsulation and system efficiency. This can be achieved by using the “Friend” feature in C++, which allows one class to access the private data of the other if the latter declares the former to be its “Friend”. This removes the need of having additional function calls and at the same time keeps the data of the class concerned hidden from all the other classes except its friends. In the pattern the author prescribes a “Friendship Zone” between the system level and Low level abstractions of the driver code. It is up to the individual firmware engineer to decide how exactly the friendships have to be established between the classes in these two levels to find an effective balance between the various competing forces mentioned in the earlier section. An example is presented in the section “Sample Implementation”. In C, one could use globals in the Friendship Zone for faster data access.

Resulting Context
The remaining inefficiencies that persist after implementing the multi tiered device driver architectural pattern are addresses while balancing the opposing forces of data encapsulation.

Figure 3: Friendship Zone pattern overlaid on the Multi tiered Device Driver pattern



Sample Real Life Implementation

Figure 4 presents an object-oriented design for a serial communication Driver which implements this pattern. The LTBDriver is a Singleton class, which declares LTB Adapter to be its "Friend". Since LTBDriver class implements the Singleton pattern, it guarantees that there will be one and only instance of it. Hence for the LTB hardware resource there is only one driver and access to that driver is through the LTBAdapter class. The LTBAdapter class controls all access to the LTB Hardware resource and implements both the Adapter and Singleton pattern. Other application classes like the tFrameBuilder class, tLTBAcquisition class and the tToolscopeComm class use it to access the LTB resource. They do not have direct access to the LTB driver. All methods of the LTB Driver are private and can only be accessed by its "Friend" the LTBAdapter. This guarantees that if by any chance some piece of code maliciously tries to call a function on the LTBDriver, it would cause a compile-time error, which is better than a run-time error. The LTBAdapter class uses a semaphore to synchronize access to the shared LTB resource by all the application threads. The LTB driver uses a Utility class Endian and a Data Buffer class LTBCommBuffer to perform its task.

LTBCommBuffer class encapsulates the buffer used to hold sent and received messages. Endian class encapsulates the various utility functions to convert from Big Endian to Little Endian format, compute checksum, and compute CRC etc.

Figures 4 and 5 were made using Rhapsody 6.0 for C++ by Ilogix (www.ilogix.com). As is evident from these UML diagrams, the low-level implementation details of the serial communication protocol like sending and receiving messages with predefined timeouts, retires, inter-character delays and checks for the message quality are completely transparent to the application level classes. Hence they do not know any more than they need to. However, at the low-level quick access to data is more important and hence "Friend" classes are used to save a time taken to make function call to access another class's data.

The Sequence Diagram in Figure 5 shows the sequence of events that happen in a typical function call made by the tToolscopeComm application thread on the LTBAdapter.

A different approach to this issue could be to add another class called LTBProtocol which has all the LTB protocol specific information encapsulated in it and making the LTBDriver more generic by changing it to just send and receive bytes. Strictly from an Object Oriented Analysis and Design (OOAD) point of view, that would be a better approach. However from a more practical point of view we would not be having several protocols that are going to be supported by the system being developed. There are only two protocols being supported and there is a very slim probability that there are going to be several more in future. Hence the otherwise valid concern of code duplication since each protocol has its own driver is really not that critical in this case. Also at the end of the day by adding another class to encapsulate the Serial Communication Protocol definition is akin to just adding another layer of abstraction between the system and the low level code. There is theoretically no limit to how abstract and generic we may make

our code and the decision to stop at a particular level of abstraction is typically governed by practical project related considerations. In this case having three layers of abstraction i.e. Application level code, System level code and Low-level code was considered appropriate by the author.

Figure 4: Class Diagram to show the design pattern for the Serial Communication Driver

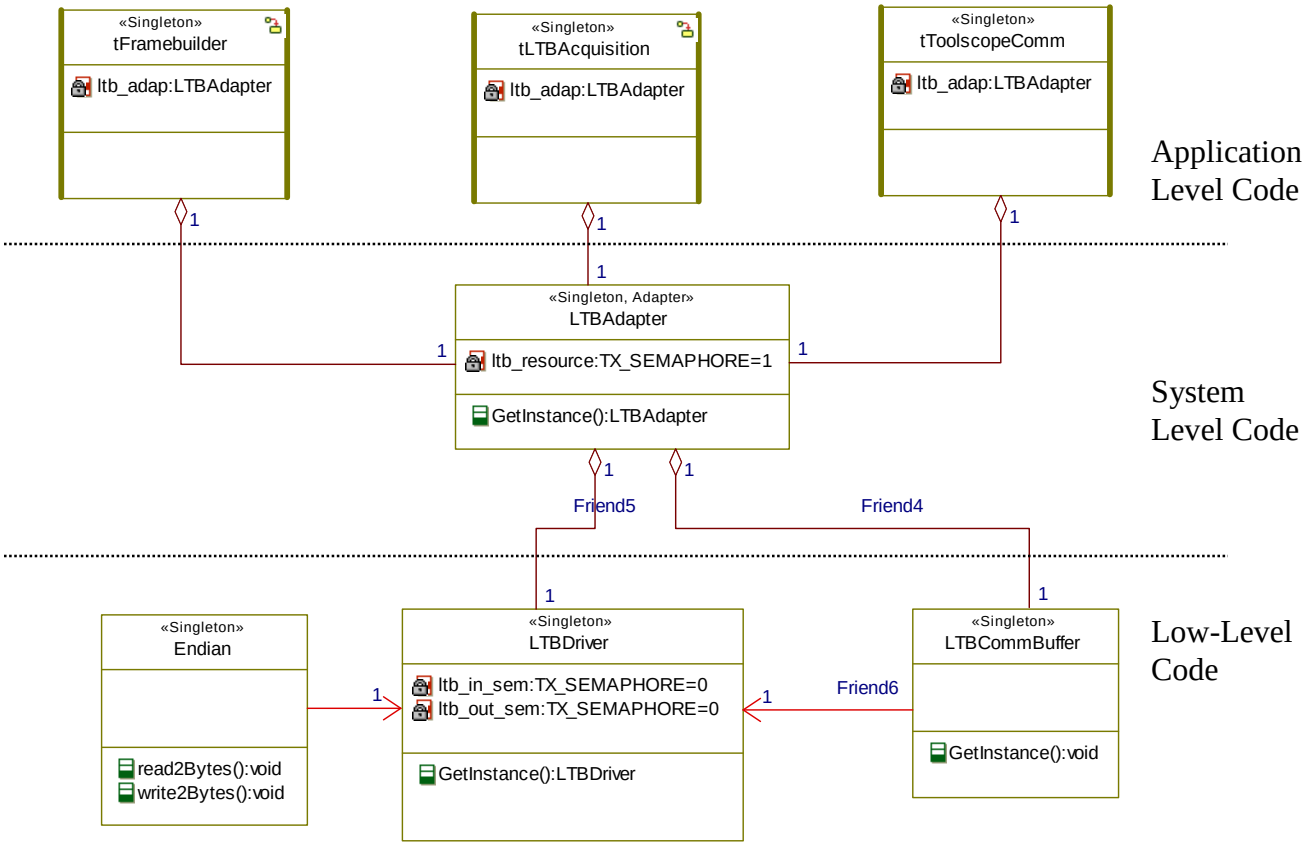
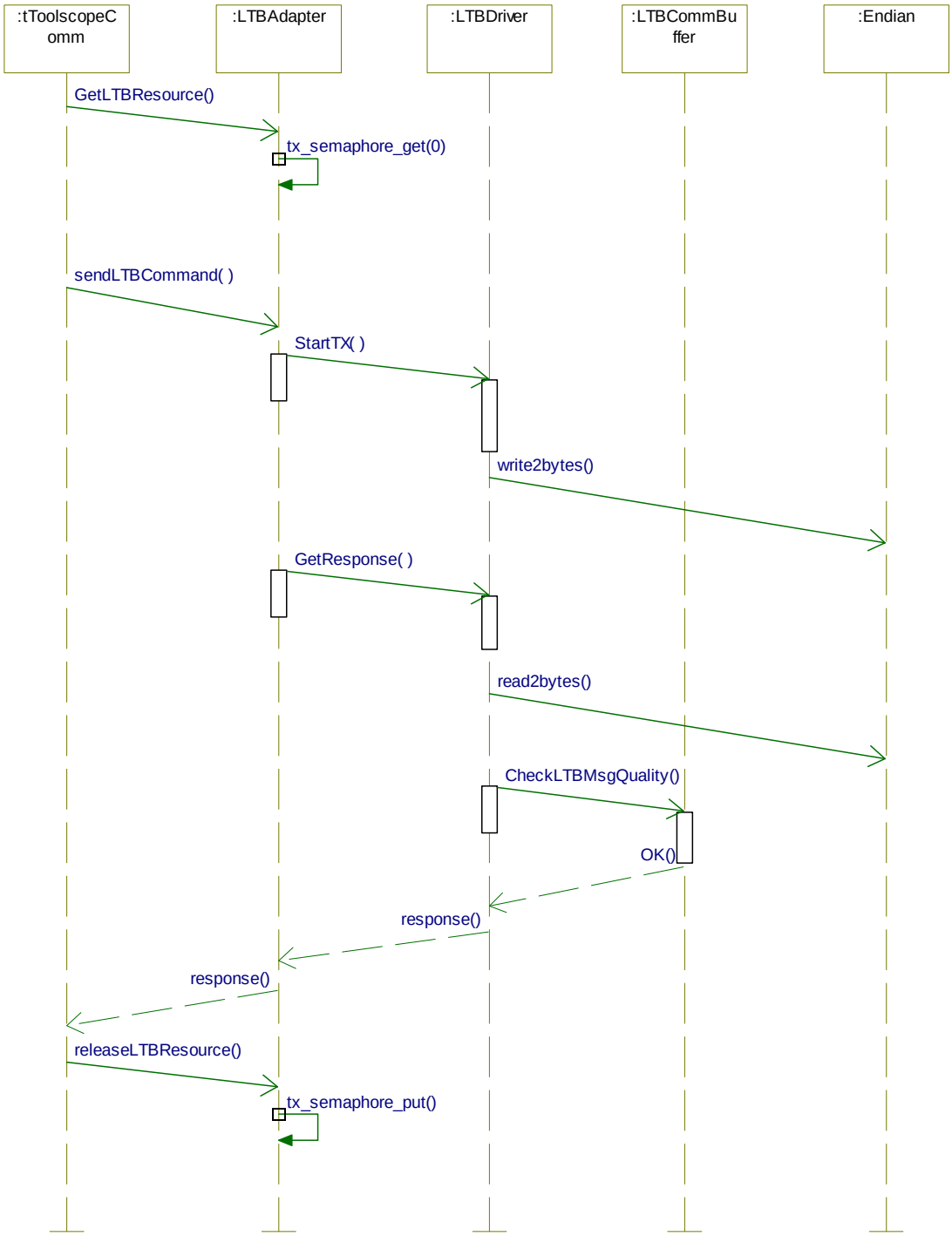


Figure 5: Sequence Diagram to show the working for the Serial Communication



NOTE: LTB is Schlumberger's proprietary serial communication protocol.

Acknowledgements
The author would first of all like to thank Lise Hvatum for introducing him to the world of Pattern Languages and PLoP and for providing general advice on writing papers for the same. It is safe to say that without her guidance this paper would not have been written. The author would also like to express his gratitude for the help he received from his shepherd, James O. Coplien, in preparing the manuscript and for providing specific advice on improving the presentation of the material. That was very helpful in getting the paper in its present form.

References

1. Internal Schlumberger technical literature.
2. E. Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”
3. “Singleton Pattern & its implementation with C++” by vsrajeshvs, Link: http://www.codeproject.com/cpp/singletonrvs.asp (accessed on 1st June 2006)