Patterns for Designing a Generic Device Driver for Interrupt Driven I/O

Sachin Bammi
Senior Software Engineer
sbammi@slb.com
Schlumberger Technology Corporation

Abstract:
This paper presents a few design patterns on designing and developing generic device drivers for interrupt driven I/O, 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 sample implementation showing how to apply them to a serial communication protocol driver.

Copyright © 2006 by Sachin Bammi. All rights reserved.

1.0 Introduction

Device drivers are all pervasive in the embedded software/firmware world. Microsoft's Windows operating system alone supports thousands of devices with more than 30,000 drivers already released and more being introduced daily [WDF06]. They form a critical part of the low-level code on which all the embedded real time applications are based upon. Hence getting them done right is of paramount importance.

Device driver development involves consideration of many different features, which include synchronization, asynchronous I/O, driver layering, plug and play, power management, etc [WDF06]. Each of these features can potentially have its own set of a pattern language, which describes the best practices to implement it. However there are some generic patterns that can potentially act as references for writing drivers with any of the aforementioned features.

The patterns presented in this paper aim at providing general architecture specific guidelines for developing device drivers for interrupt driven I/O on proprietary hardware. 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 taking several measurement at the same time. While the pattern language that develops due to this effort will be rather specific in nature, it is the author's belief that 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 "Measurement While Drilling (MWD) Firmware Pattern Language".

Only the shaded design patterns i.e. "Multi-Tiered Device Driver", "Synchronous Managed Access", "Asynchronous Managed Access" and "Friendship Zone" in Figure 1 are introduced in this paper. Others are work in progress and some more information about them can be found in the pattern thumbnails at the end of this paper. The paper also presents a real life sample implementation in C++ for some of them as applied to a serial communication protocol driver.

2.0 Intended Audience and Scope

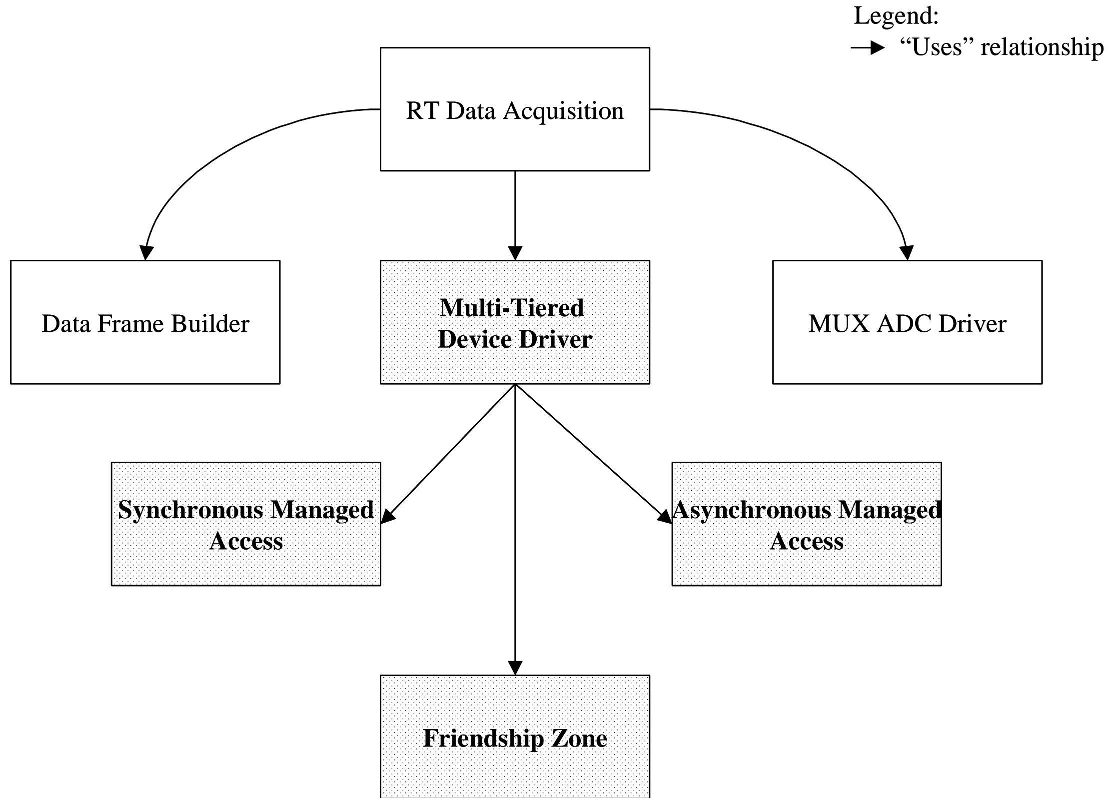
This paper is not intended to be a tutorial for writing device drivers. There are several available on the web. It is also not intended to address the area of device driver development for common operating systems like Windows (NT, 2000, XP), Sun Solaris, Linux and Unix. There is considerable support in terms of technical literature and documentation available for developing device drivers for aforementioned common operating systems [WDD05, WDT05, VM06, Cant99, Pajari91].

The intended audience of this paper is engineers developing custom device drivers for interrupt driven I/O based embedded applications on proprietary hardware using either homegrown or commercially available real time operating systems (RTOS). The scope of this work is limited to general design issues related to device driver development for

Copyright © 2006 by Sachin Bammi. All rights reserved.

proprietary embedded applications that are responsible for data acquisition, data processing, data transmission and data logging in real time.

Figure 1: Measurement While Drilling Firmware Pattern Language version 1.0



It presents some key concepts to keep in mind while designing a generic device driver for interrupt driven I/O. The patterns presented here are by themselves not enough for a good design since a good design requires deep knowledge of the device under consideration and the specific hardware and RTOS on which the device driver will run. What this paper tries to provide are some generic characteristics of a good device driver design, which the author believes are independent of more specific hardware and RTOS issues.

Some of the patterns in MWD Firmware patterns language provide the most benefit when they are applied together. This is because some provide a more specific refinement to the others in the language but still have enough intrinsic merit in author's judgment to stand on their own as a pattern. For example the "Multi-Tiered Device Driver" and the "Friendship Zone" patterns can be applied together along with the "synchronous Managed Access" pattern to write a serial communication driver as described later in this paper in section 7.0.

Copyright © 2006 by Sachin Bammi. All rights reserved.

3.0 Pattern: Multi Tiered Device Driver

3.1 Context

Device driver code has several parts. A generic driver would have code that works directly with the real time operating system and accesses the hardware registers, code that provides an interface to the rest of the embedded application to use the device driver and code that provides for other utility and house keeping needs of the device driver. Organizing this code into meaningful blocks can make the design flexible and the code easy to maintain.

3.2 Problem

An significant 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, easy, without affecting too much the other components of the code. However this flexibility comes at the price of code bloat and performance efficiency. Hence the problem is to find the right trade-off.

3.3 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 be provided so that a balance is reached between design flexibility and system efficiency in real time systems.

3.4 Solution

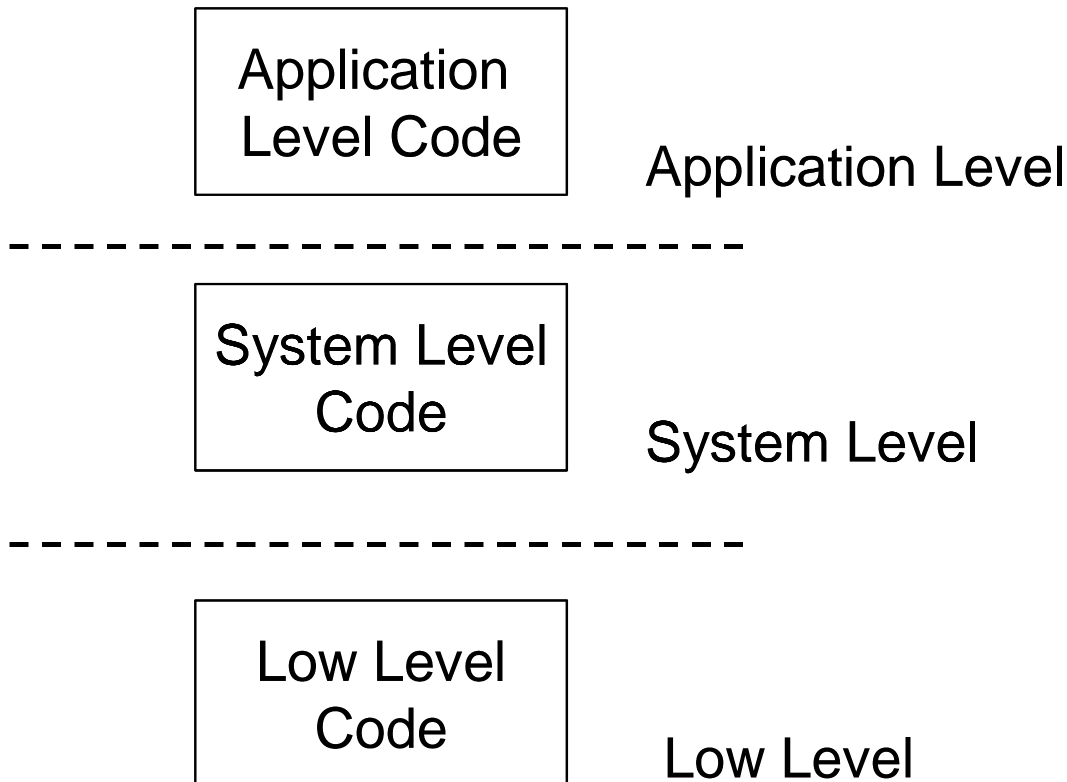
Design a multi tiered architecture that divides the device driver code into the three abstractions or groups: Application level, System level and Low level. If the hardware changes then the code should be modified only at the Low level or conversely if the business requirements change then only the application 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 architectural pattern is shown in the Figure 2.

3.5 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. There should be only one object that represents the driver for a particular device and as such should provide a synchronized way for application level objects to access the device.

Copyright © 2006 by Sachin Bammi. All rights reserved.

Figure 2: Multi Tiered Architectural pattern for device driver design



3.6 Related Patterns
The device driver code uses the Singleton pattern [GHJV94] to guarantee that that there is only one instance of it. The system level code can use the Adapter or the Facade patterns [GHJV94] to hide the low level details of the driver from the application level objects. The adapter/facade for the device driver's low level code is also a singleton.

3.7 Known Uses
Barry Rubel [Rubel95] discusses the use of layered architecture in decomposing system requirements for mechanical control systems.

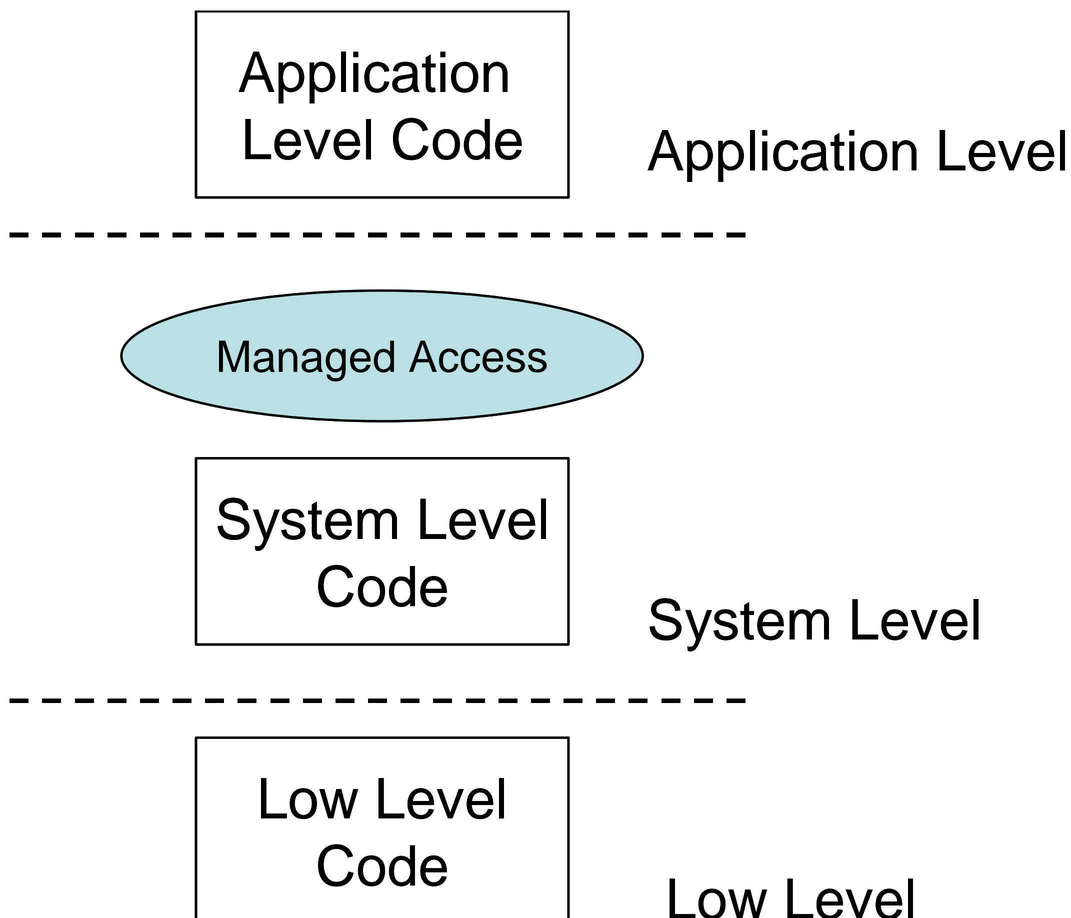
Some device drivers developed in Schlumberger for real time applications have used a layer approach to organizing and architecting the driver code [SLB].

Copyright © 2006 by Sachin Bammi. All rights reserved.

4.0 Managing Device Access

Access to hardware device that is being used in real time needs to be managed. If let alone it can be made to do more than one thing at the same time by application level objects which in turn can lead to undesirable functioning of the device. An example of this is if a printer is made to print two documents at the same time without proper access management then the result is undesirable. Access to the device can be managed in two ways: synchronously or asynchronously. Synchronous managed access can be used for interrupt driven I/O between various slave sub systems and the master system where a response/acknowledgment is necessary for data acquisition/communication in real time. Asynchronous managed access could be used for one-way communication where either a response/acknowledgement is not necessary or its simply too inefficient to wait for response/acknowledgement. Common examples of this are sending a broadcast message, or sending a command to a printer etc. The next two patterns present an implementation for these two approaches to having a managed device access.

Figure 3: Multi Tiered Architectural pattern with Managed Access



Copyright © 2006 by Sachin Bammi. All rights reserved.

4.0 Pattern: Synchronous Managed Access

4.1 Context

Application level objects need to access the device to perform their functions and receive a response/acknowledgment back. They use the Driver class that encapsulates the device to access it. The device cannot handle multiple requests at the same time. The device should be able to carry out the work for one application object without any interruptions from other application objects as this can lead to undesirable effects/results. The application object waits for the response.

4.2 Problem

Different application level objects may try to access the device at the same time and hence can adversely affect the function of the device.

4.3 Forces

Synchronization adds latency into the system by making the other application objects wait for a chance to get access to the device. If not implemented right it can lead high priority tasks to starve due to priority inversion [SRL90, KB02, Kalinsky03, Kalinsky06]. Improperly chosen synchronization techniques can lead to severe problems as exemplified by the software glitch that was discovered during NASA's Mars mission [Jones06, Reeves98]. On the other hand a synchronously managed access is very easy and straightforward to implement.

4.4 Solution

Use synchronization but judiciously. If the application objects can wait for a response then using a synchronously managed access approach to driver resources is a way to go. One has to choose carefully between the various types of synchronization mechanisms available like semaphores, mutexs, critical sections etc and decide on what fits best for their implementation. If there are multiple devices that need access to them being synchronized then using semaphore is good, but for one device it is better to use a mutex of type - priority inheritance [Kalinsky03, Kalinsky06]. Using re-entrant function calls can help reduce the need to add synchronization.In synchronously managed access the application level object waits for the response to a request and after getting it or timing out releases the hardware resource.

Figure 4 shows synchronously managed access for I/O where a mutex is used to provide for task synchronization since there are multiple tasks but only one device driver to share.

4.5 Resulting Context

As shown in the following figure, implementing this pattern guarantees that the device driver will handle only one request at a time and that various application objects will not stomp over each other in trying to get access to the device driver. This approach is very straight forward to implement as long as the developer keeps in mind the various pitfalls possible in implementing a synchronously managed access as described above.

Copyright © 2006 by Sachin Bammi. All rights reserved.

4.6 Related Patterns

Schmidt and Cranor in their a pattern called “Half-Sync/Half-Async” propose to simplify concurrent programming effort by decoupling synchronous I/O form asynchronous I/O without compromising on execution efficiency [VCK96, SSRB00]. They propose synchronous managed access for application level tasks to a queue of messages, which is being filled up asynchronously.

4.7 Know Uses

Kalisky has talked about the uses of synchronously managed access pattern in his course titled “Architectural design of device drivers “ at the Embedded systems conference in 2006 [Kalinsky06].

In Schlumberger drivers for proprietary serial communication protocols in the MWD firmware implement this pattern [SLB].

Schmidt and Cranor in their a pattern called “Half-Sync/Half-Async” present examples from BSD Unix [LMKQ84], the original System V UNIX STREAMS communication framework [Ritchie84], Multi threaded version of Orbix 1.3 [Horn93], Motorola Iridium system [Schmidt96] and the Conduit communication framework [Zweig90] from the Choices OS project [CIRM93] as examples of places where synchronous managed access pattern is applied in conjunction with asynchronous managed access pattern [VCK96, SSRB00].

5.0 Pattern: Asynchronous Managed Access

5.1 Context

Application level objects need to access the device to perform their functions but either do not expect to receive a response/acknowledgment back or it is very inefficient if they wait while blocking the hardware resource. They can be notified or can check the status of the I/O by themselves at a later stage. The device cannot handle multiple requests at the same time. The device should be able to carry out the work for one application object without any interruptions from other application objects as this can lead to undesirable results.

5.2 Problem

Different application level objects may try to access the device at the same time and hence can adversely affect the function of the device. The application objects do not need to wait for a response/acknowledgement from the device. How can we implement managed access which involves no waiting for response by application objects and no multiple requests to handle at the same time for the driver?

5.3 Forces

While the various application objects do not have to wait for I/O the driver can still handle only one I/O request at a time. Hence a synchronously managed access to the driver as described in the previous pattern is not necessary. Asynchronous

Copyright © 2006 by Sachin Bammi. All rights reserved.

implementation can be used to improve efficiency but on the other hand can make the programming logic very complex.

5.4 Solution

The solution is to apply asynchronously managed access judiciously. The implementation involves splitting I/O into two separate asynchronous parts where the application objects access the device synchronously and after submitting the I/O request release access to the driver. The driver implements a queue in which its keeps the accumulated I/O requests. The application objects are informed or they can check themselves about the I/O status at a later stage. Figure 5 presents a sequence diagram showing asynchronous managed access for driver output.

The asynchronous input can be in turn implemented in two ways. The first approach is to let the device adapter periodically poll the driver for new messages. The driver would have to maintain a message queue/buffer to handle overflow of incoming messages if the polling frequency is not high enough. The second approach is where the device driver informs the device adapter of a new message every time it receives one. The first approach is useful in cases where the interrupt frequency is very high and hence the device adapter tries to get the messages from the driver in bulk at a frequency that it can manage. Figure 6 presents a typical sequence of events for this scenario. The second approach can be applied when the interrupt frequency is erratic and not very high. In this case the device adapter does not poll for messages at some predefined interval but instead gets a notification from the driver when a message comes in. Figure 7 presents a typical sequence of events for this scenario.

5.5 Resulting Context

The application objects gets to access the driver in a way so that driver does not have to handle multiple requests at the same time and they get to do so without having to wait for a response to their I/O request.

5.6 Related Patterns

Schmidt and Cranor in their a pattern called “Half-Sync/Half-Async” propose to simplify concurrent programming effort by decoupling synchronous I/O form asynchronous I/O without compromising on execution efficiency [VCK96, SSRB00]. For the low-level threads they propose using asynchronously managed access where the driver creates a notification on receiving a message which is then handled by the system and the message is put in a queue.

5.7 Know Uses

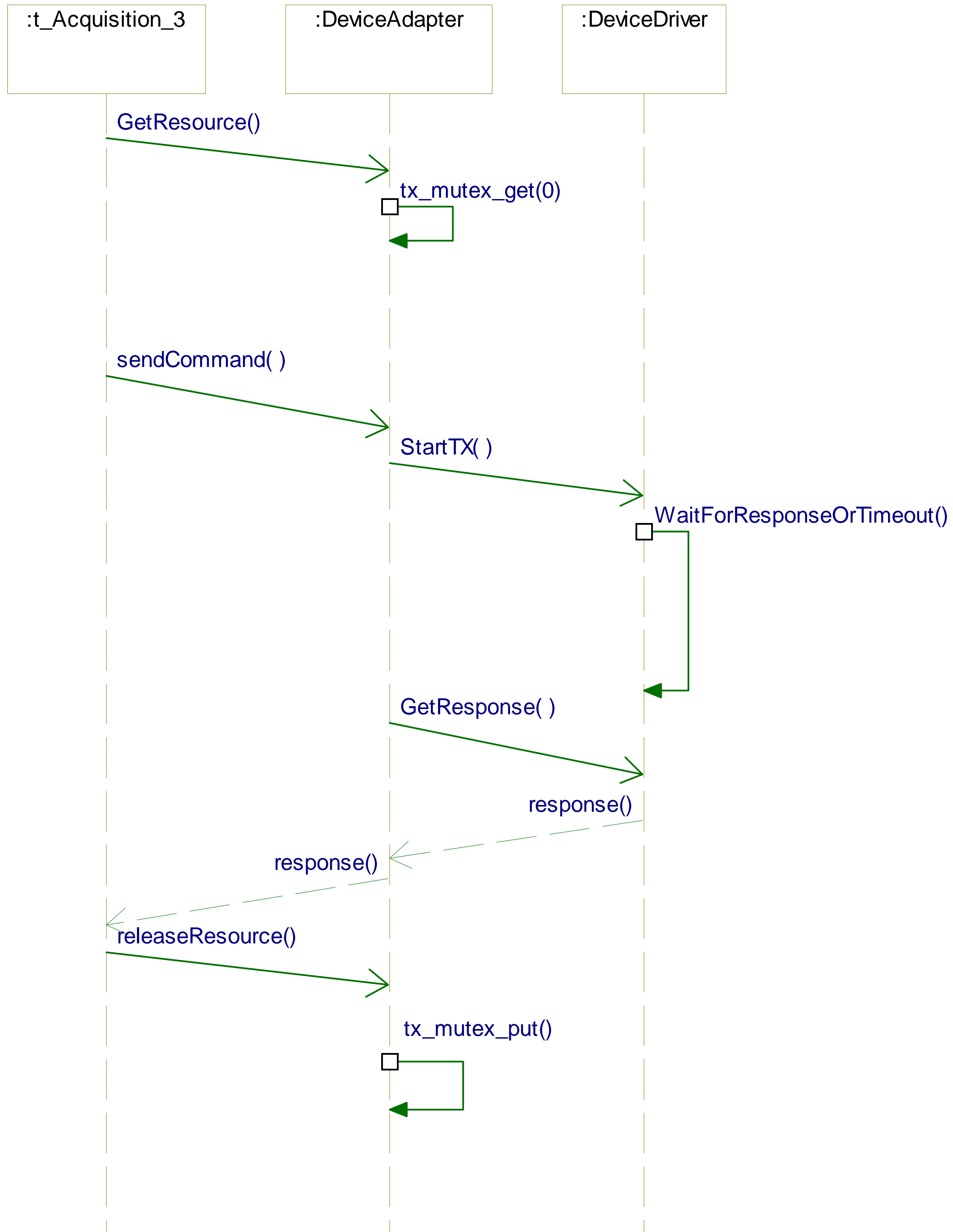
D Kalisky talked about this in his course titled “Architectural design of device drivers “ at the Embedded systems conference in 2006 [Kalinsky06].

Schmidt and Cranor in their a pattern called “Half-Sync/Half-Async” present examples from BSD Unix [LMKQ84], the original System V UNIX STREAMS communication framework [Ritchie84], Multi threaded version of Orbix 1.3 [Horn93], Motorola Iridium system [Schmidt96] and the Conduit communication framework [Zweig90] from the

Copyright © 2006 by Sachin Bammi. All rights reserved.

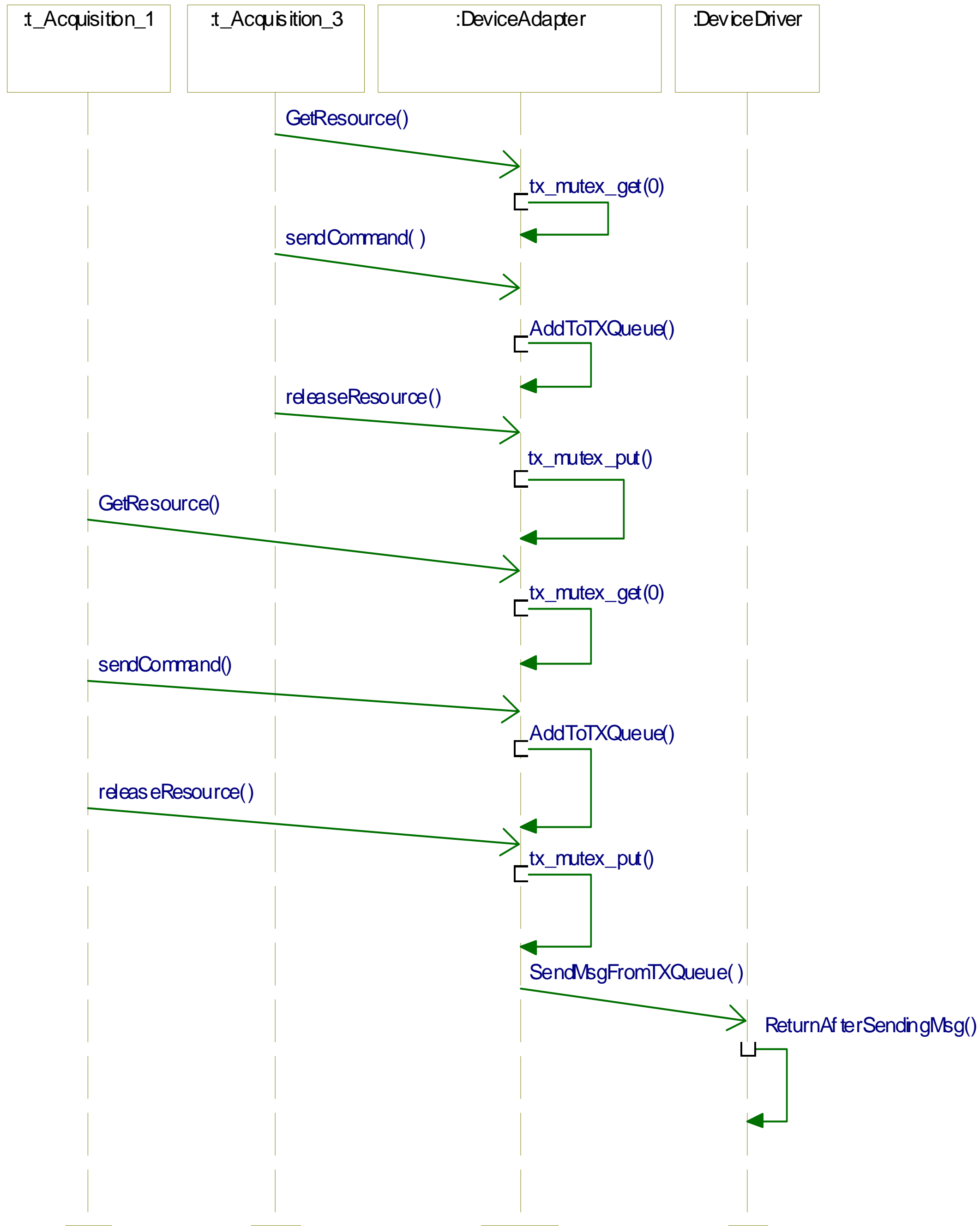
Choices OS project [CIRM93] as examples of places where asynchronous managed access pattern is applied in conjunction with synchronous managed access pattern [VCK96, SSRB00].

Figure 4: Synchronous Managed Access for Input/Output



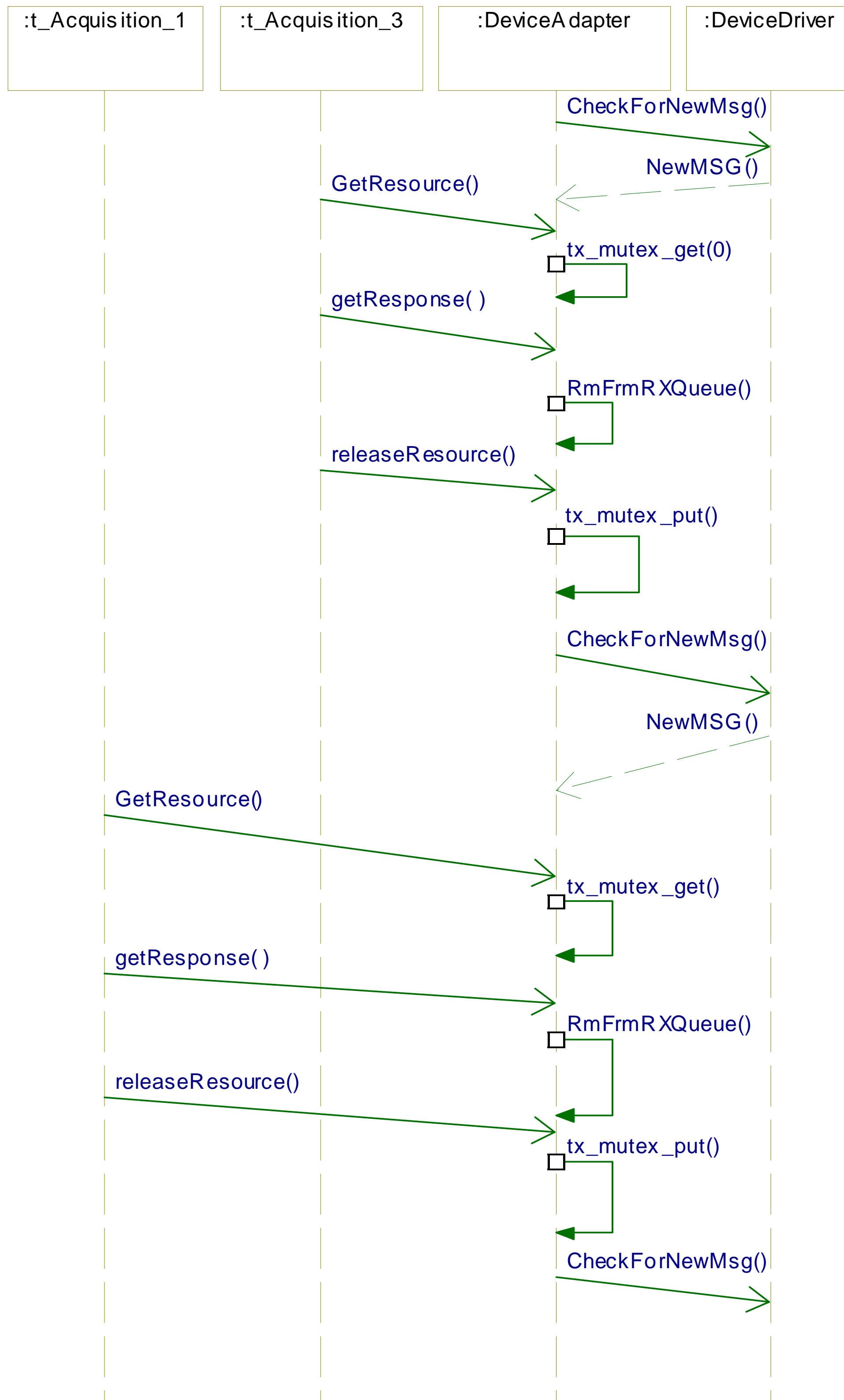
Copyright © 2006 by Sachin Bammi. All rights reserved.

Figure 5: Asynchronous Managed Access for Output



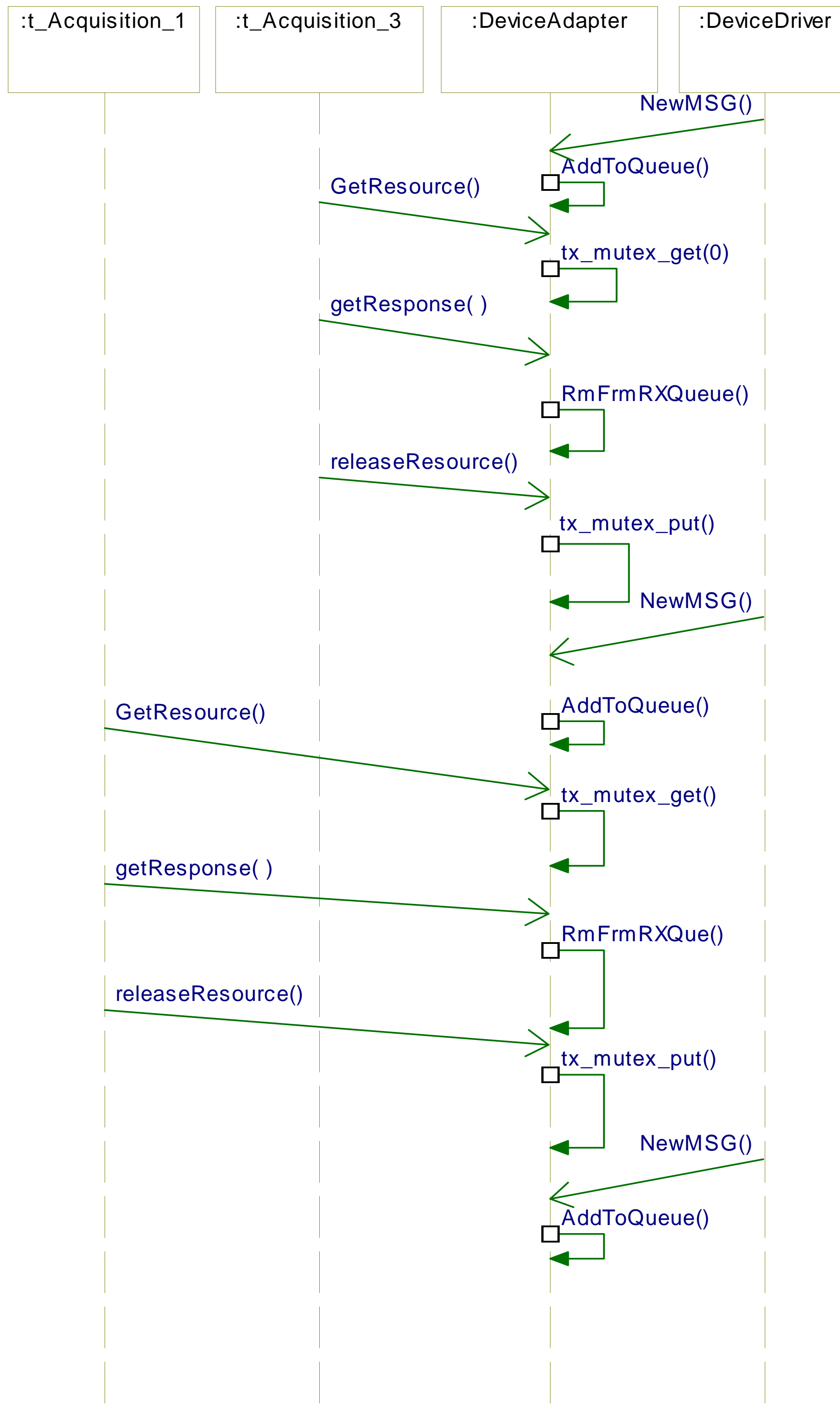
Copyright © 2006 by Sachin Bammi. All rights reserved.

Figure 6: Asynchronous Managed Access for Input: Polling version



Copyright © 2006 by Sachin Bammi. All rights reserved.

Figure 7: Asynchronous Managed Access for Input: Push version



Copyright © 2006 by Sachin Bammi. All rights reserved.

6.0 Pattern: Friendship Zone

6.1 Context

Restricting access to the internal data buffers of the device driver is critical to prevent any malicious code from accidentally corrupting them and consequently degrading the performance of the device. Encapsulating and having a separate abstraction layer for low-level driver code is the first step in this direction. However encapsulation and layering in certain time-critical embedded systems can have a negative effect on system efficiency due to the use of access member functions instead of direct data access. This is especially a concern for parts of the code that service interrupts as it can potentially lead to increase in interrupt latency [Ganssle01]. Another drawback of encapsulation is the additional code bloat, which for some embedded systems may not be acceptable.

6.2 Problem

How to balance the need of data security with system efficiency especially in the low-level code where interrupt latency can be a major concern.

6.3 Forces

From a truly data encapsulation and security point of view each class/module should protect its data by either keeping it private or providing the appropriate access control 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 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.

6.4 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 section 6.3. This is because depending on the specific system requirements, proprietary hardware and communication protocol details, the relationships between the objects in the friendship zone can vary quite a bit. An example is presented in the section 7.0 - “Sample Implementation”. In C, one could use global variables in the Friendship Zone for faster data access.

Copyright © 2006 by Sachin Bammi. All rights reserved.

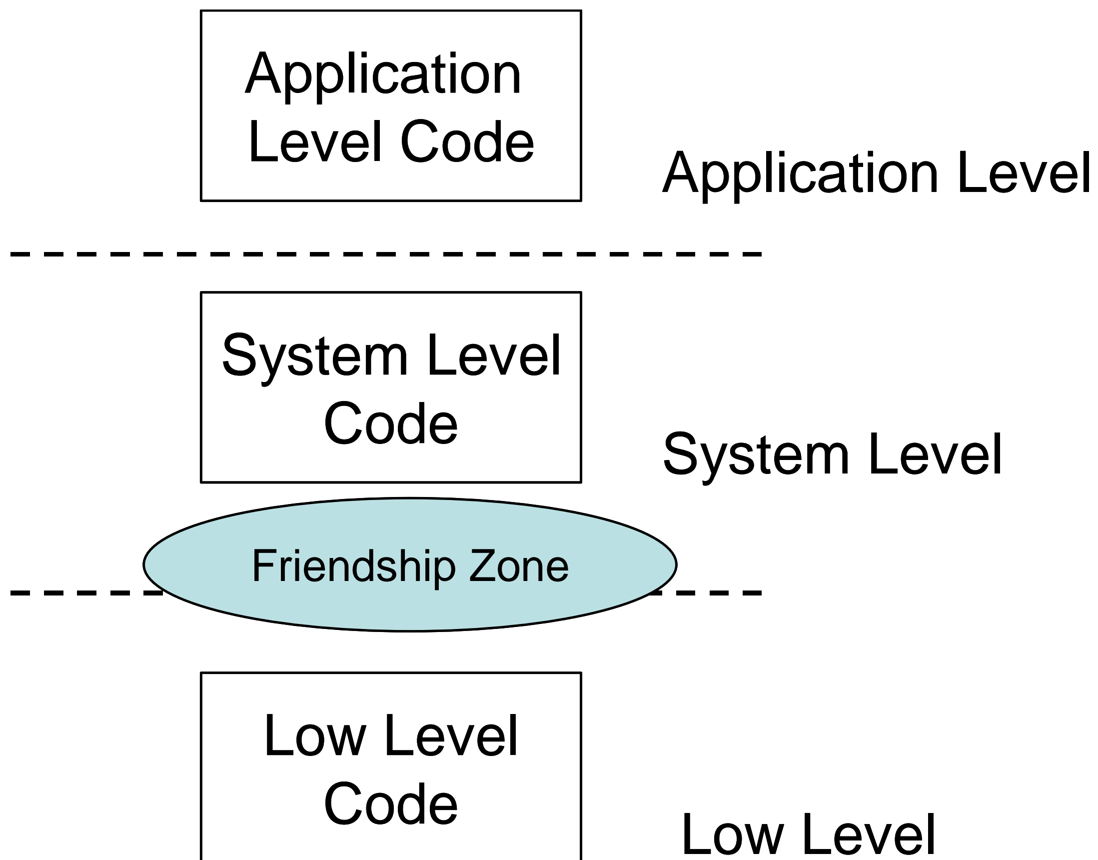
Some other things that can be considered to speedup things in the low level code are using in-lining functions, no virtual member functions and using constant references in parameter passing so that copy constructor does not get called.

6.5 Resulting Context
The inefficiencies that can happen due to data encapsulation are addressed without having to compromise on data security.

6.6 Related Patterns
Gamma et. al. in their landmark design patterns book present the Memento pattern in which an object uses the "Friend" feature in C++ to effectively have two interfaces – 'narrow' and 'wide' so that it could allow access to its private data while "Preserving encapsulation boundaries" [GHJV94].

6.7 Known Uses
In Schlumberger drivers for proprietary serial communication protocols in the MWD firmware implement this pattern [SLB].

Figure 8: Multi Tiered Architectural pattern with Friendship Zone



Copyright © 2006 by Sachin Bammi. All rights reserved.

7.0 Sample Implementation: A Multi-Tiered, Synchronous Serial Communication Driver using Friendship Zone Pattern

Figure 9 presents an object-oriented design for a serial communication Driver which implements the “Multi-Tiered Device Driver”, “Synchronous Managed Access” and “Friendship Zone” patterns. The DeviceDriver is a Singleton class, which declares DeviceAdapter to be its “Friend”. Since DeviceDriver class implements the Singleton pattern, it guarantees that there will be one and only one instance of it. Hence for one device there is only one driver and access to that driver is through the DeviceAdapter class. The DeviceAdapter class controls all access to the Device and implements both the Adapter and Singleton pattern. Other application classes like the t_Acquisition_1 class, t_Acquisition_2 class and the t_Acquisition_3 class use it to access the device. They do not have direct access to the device driver. All methods of the DeviceDriver are private and can only be accessed by its “Friend” the DeviceAdapter. This guarantees that if by any chance some piece of code maliciously tries to call a function on the DeviceDriver, it would cause a compile-time error, which is better than a run-time error. The DeviceAdapter class uses a mutex to synchronize access to the shared device by all the application threads. The device driver uses a Utility class and a Data Buffer class to perform its task.

CommBuffer 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. depending upon the specific details of underlying communication protocol.

As is evident from the UML sequence diagrams 9 and 10, 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 without breaking encapsulation boundaries. 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.

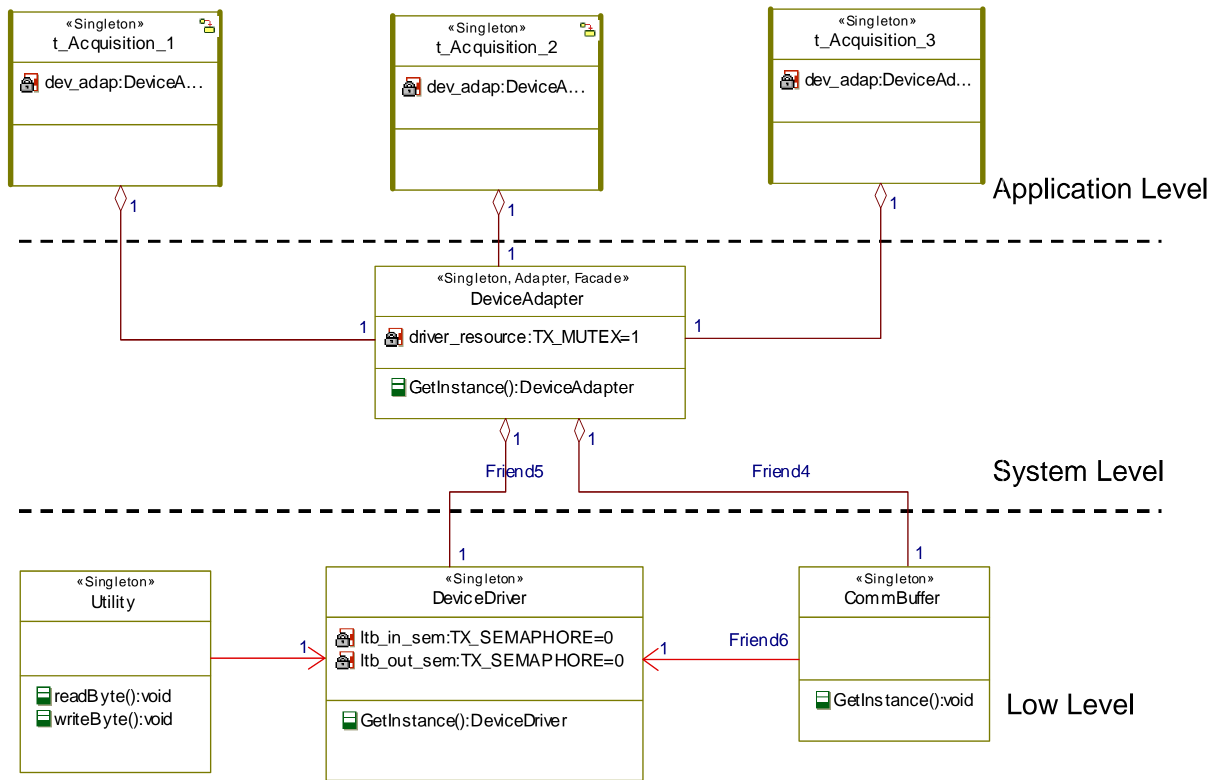
The sequence diagram in Figure 10 shows the sequence of events that happen in a typical function call made by the t_Acquisition_3 application thread on the DeviceAdapter.

A different approach to this issue could be to add another class called Protocol which has all the communication protocol specific information encapsulated in it and making the DeviceDriver 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 there were not going to be several protocols 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

Copyright © 2006 by Sachin Bammi. All rights reserved.

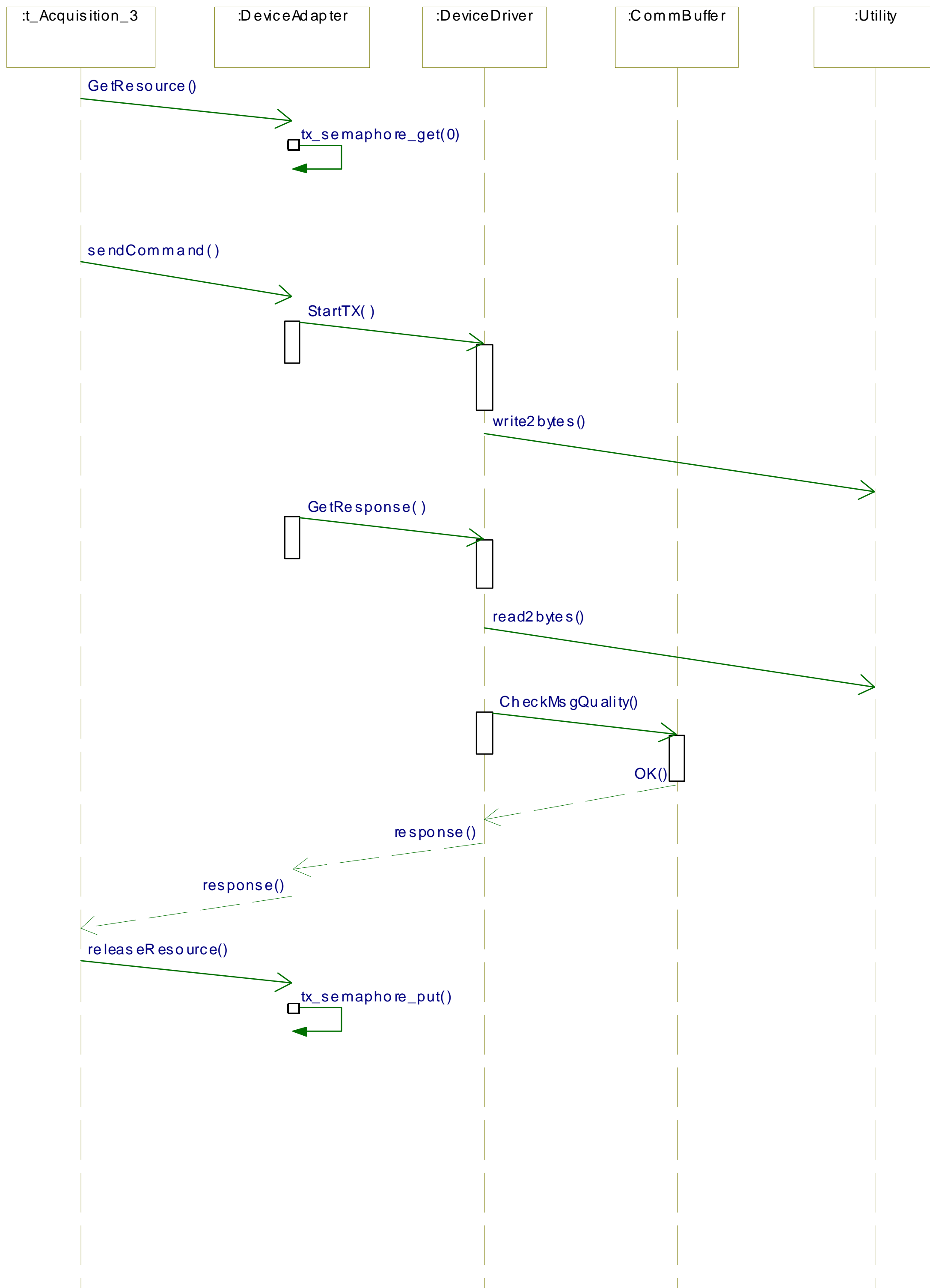
to just adding another layer of abstraction between the system and the low level code. There is theoretically no limit to how abstract and generic we may make our code and the decision to stop at a particular level of abstraction is typically governed by practical project related considerations. In this 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 9: Class Diagram to show the design pattern for the Serial Communication Driver



Copyright © 2006 by Sachin Bammi. All rights reserved.

Figure10: Sequence Diagram to show the working for the Serial Communication



Copyright © 2006 by Sachin Bammi. All rights reserved.

8.0 Pattern Thumbnails

Pattern	Intent
RT Data Acquisition	Divide the system into "Data providers", "Data Consumers" and "Data Brokers"
Multi Tiered Device Driver	Divide the code into "Application Level Code", "System Level Code" and "Low Level Code"
Synchronous Managed Access	Provide Application level code with synchronous access to Low level code.
Asynchronous Managed Access	Provide Application level code with asynchronous access to Low level code. It includes sub-patterns for Async output, Async input (polling version) and Async input (Push version)
Friendship Zone	Form relationships – "friendships" between low level and system level classes that promote faster data access without breaking data encapsulation boundaries.
Data Frame Builder	Modularize the Data frame building process so that future changes in business logic can be incorporated easily
MUX ADC Driver	A common approach to sample Analog to Digital Converter (ADC) data from a multiplexed data acquisition channel.

9.0 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 and the members of the writers workshop – "Intimacy Gradient" at PLoP 2006, for providing specific advice on improving the presentation of the material. That was very helpful in getting the paper in its present form.

10.0 References

1. [WDF06] Introduction to Windows Driver Foundation. Link: http://www.microsoft.com/whdc/driver/wdf/wdf-intro.mspx (accessed on 1st June 2006)
2. [WDD05] *Writing Device Drivers*. Sun Microsystems, Inc., 2005.
3. [WDT05] *Device Driver Tutorial*. Sun Microsystems, Inc., 2005. Link: http://192.18.109.11/817-5789/817-5789.pdf (accessed on 21st November 2006)
4. [VM06] Windows NT Device Driver Development (OSR Classic Reprints) by Viscarola, P., G., and Mason, W., A., OSR Press (2006)
5. [Cant99] Writing Windows WDM Device Drivers by Cant, C., CMP; Book & CD Rom edition (January 7, 1999)

Copyright © 2006 by Sachin Bammi. All rights reserved.

6. [Pajari91] Writing UNIX Device Drivers by Pajari, G. Addison-Wesley Professional; 1st edition (November 25, 1991)
7. [GHJV94] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Boston, 1994.
8. [Rubel95] Rubel, B., "Patterns for Generating a Layered Architecture", Chapter 7, Pattern Languages of Program Design, edited by Coplien, J. and Schmidt, D., Addison-Wesley, 1995.
9. [VCK96] Vlissides, J., Coplien, J. and Kerth, N., eds. Pattern Languages of Program Design-2, Addison-Wesley, 1996.
10. [SSRB00] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects, John Wiley & Sons; 1 edition, 2000.
11. [SLB] Internal Schlumberger technical literature.
12. [Jones06] Jones, M. B. "What really happened on Mars?" an email communication sent by M. B. Jones. Link: http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html (accessed on 21st November 2006)
13. [Reeves98] Reeves, G. "Re: What Really Happened on Mars?," Risks-Forum Digest, Volume 19: Issue 58, January 1998. Link: http://catless.ncl.ac.uk/Risks/19.54.html#subj6 (accessed on 21st November 2006)
14. [SRL90] Sha L., Rajkumar, R., and Lehoczky, J.P. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization,," *IEEE Transactions on Computers,* September 1990, p. 1175
15. [KB02] Introduction to Priority Inversion, Kalinsky, D. and Barr, M., Embedded Systems Programming, VOL. 15 NO. 4, April 2002. Link: http://www.embedded.com/story/OEG20020321S0023 (accessed on 21st November 2006)
16. [Kalinsky03] Kalinsky, D., Introduction to Real-Time Operating Systems, Introductory Course for Real-Time Software Development using an RTOS, Courseware Version 2.1, 3-05-03, D. Kalinsky Associates, 2003.
17. [Kalinsky06] Kalinsky, D., Architectural Design of Device Drivers, Tutorial # ESC-505, Embedded Systems Conference 2006 San Jose – Silicon Valley, D. Kalinsky Associates, 2006.
18. [Ganssle01] Interrupt Latency, Ganssle, J. G. Embedded Systems Programming, VOL. 14 NO.12, October 2001. Link: http://www.embedded.com/story/OEG20010918S0052 (accessed on 21st November 2006).
19. [LMKQ84] Leffler, S. J., M.McKusick, M., Karels, M. and Quarterman, J. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
20. [Ritchie84] Ritchie, D. "A Stream Input–Output System,"*AT&TBell Labs Technical Journal*, vol. 63, pp. 311–324,Oct. 1984.
21. [Horn93] Horn, C. "The Orbix Architecture," tech. rep., IONA Technologies, August 1993.

Copyright © 2006 by Sachin Bammi. All rights reserved.

22. [Schmidt96] Schmidt, D. C. “A Family of Design Patterns for Application level Gateways,” *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.
23. [Zweig90] Zweig, J. M. “The Conduit: a Communication Abstraction in C++,” in *Proceedings of the* 2nd *USENIX C++ Conference*, pp. 191–203,USENIX Association, April 1990.
24. [CIRM93] Campbell, R., Islam, N., Raila, D., and Madany, P. “Designing and Implementing Choices: an Object-Oriented System in C++,” *Communications of the ACM*, vol. 36, pp. 117–126, Sept. 1993.

Copyright © 2006 by Sachin Bammi. All rights reserved.