

# **A generic real time data acquisition pattern language for embedded applications involving interrupt driven I/O**

**Sachin Bammi**  
**Senior Software Engineer**  
[sbammi@slb.com](mailto:sbammi@slb.com)  
**Schlumberger Technology Corporation**

## Abstract:

This paper presents six design patterns on designing and developing generic real time embedded applications, which involve interrupt driven I/O. The pattern language includes patterns for real time data acquisition and developing device drivers for analog to digital converters and serial communication. These patterns 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. They provide general guidelines for developing embedded applications on custom hardware.

## **1.0 Introduction**

The patterns presented in this paper aim at providing general architecture specific guidelines for developing real time embedded data acquisition systems involving interrupt driven I/O on proprietary hardware. The patterns are elements 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 measurements 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 "Real Time Data Acquisition (RTDA) firmware pattern language".

## **2.0 Intended Audience and Scope**

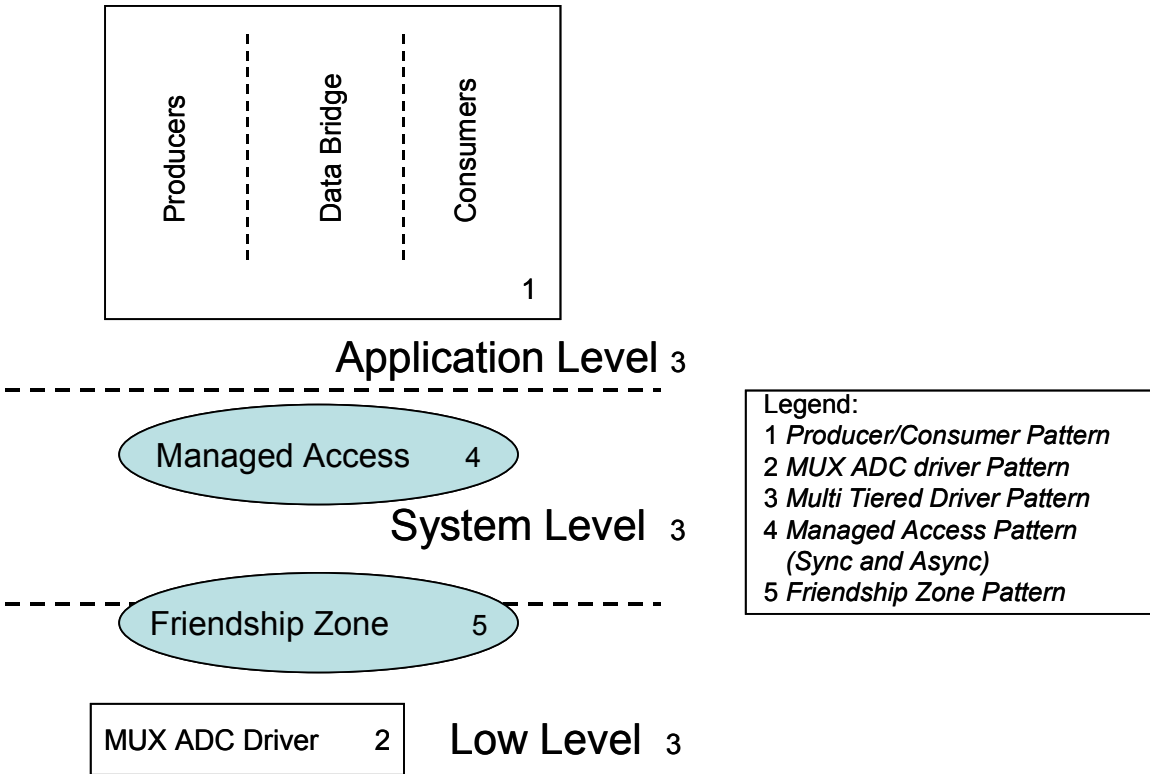
The intended audience of this paper is beginning to intermediate level embedded software engineers developing custom real time applications for interrupt driven I/O based embedded systems on proprietary hardware using either homegrown or commercially available real time operating systems (RTOS). The author expects many senior or advanced level embedded software engineers to be familiar with most if not all of the concepts discussed in this paper. The technical scope of this work is limited to general design issues related to system development for proprietary embedded applications that are responsible for data acquisition, data processing, data transmission and data logging in real time.

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 design, which the author believes are independent of more specific hardware and RTOS issues.

Some of the patterns in RTDA 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. Similarly the "Producer/Consumer" pattern can be applied with "Multi-Tiered Device Driver" pattern to define the overall architecture of the real time application.

Figure 1 presents these patterns in a mosaic fashion to show how they fit with each other in the big picture. The following sections discuss the individual patterns in more detail and the reader of this paper may be well served by referring back to this diagram after reading each of the patterns.

Figure 1: Pattern Mosaic for Architecture of a Real time data acquisition embedded application



### **3.0 Pattern: Producer/Consumer or Publish/Subscribe**

#### **3.1 Context**

In any real time data acquisition there are typically more than one source of data each producing some information that needs to be captured at a different rate and possibly format. Acquiring data is just one part of the picture; there can be several different processes/threads that consume the acquired data either directly as it comes from the source or after it has been pre-processed by another consumer process/thread. The data consuming processes/threads are typically governed by business requirements, which in turn can be time-constrained. Some of these may not be directly impacted by a business need but need the data from the source to monitor the system and perform diagnostics to ensure it keeps running flawlessly. There also needs to be mechanism by which the various processes/threads can share data while being able to do their job. In a commercial real time system, hence it is easy to see, that things can quickly get very complex if there are varied and/or multiple data sources or varied and/or changing business needs or both.

#### **3.2 Problem**

How to design a real time system that can handle the current and future mix of data sources with their varying data production rates and the constantly changing business needs. This problem is especially exacerbated when the time from concept to market is less and the project stakeholders are banging on the door.

#### **3.3 Forces**

To make a flexible design that can handle future addition or change in the nature of data sources along with changing business needs requires more thought, more code and more modularization of the various software functions. It also requires more patience from the project stakeholders. This creates the need to identify and develop best practices that can provide designers with specific advice to tackle the problem.

#### **3.4 Solution**

A very widely used solution (see the 'Known Uses' section 3.7) is to create separate threads/processes for data producers and data consumers, which have a mechanism like a queue or a synchronously accessed shared data buffer to communicate and share data with each other. The mechanism used by the producers and the consumers to communicate with each other is termed as the data bridge figure illustrating the pattern.

If the data bridge is made using queues the one typical concern is queues are bound to a particular data type. Hence if the various data sources are producing different types of data then this could lead to a proliferation of queues through out the system and thus making it complex. This issue can be easily circumvented by grouping data into object collections/structures with a unique message id to determine its source.

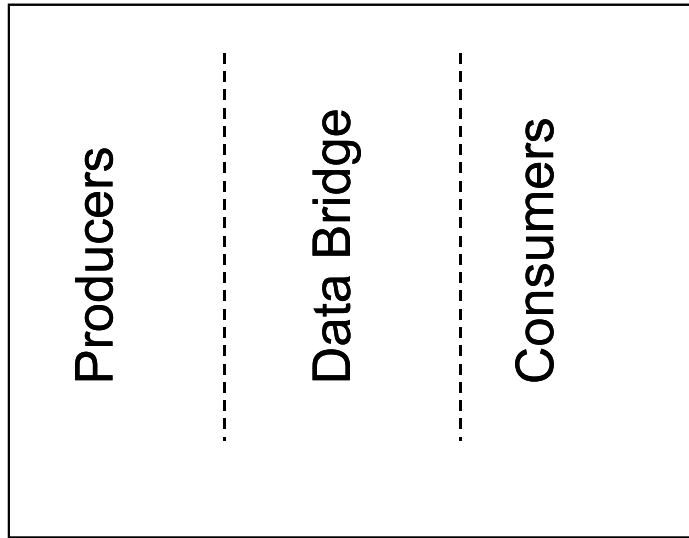
Also care must be taken while adding consumer processes/threads to handle the situation if no data is received by then from the producers. This could happen in an event if a data source mal functions or if the consumer thread got started sooner than the producer thread. Typically the producer processes/threads have higher priority and are made to start before the consumers. The

data bridge should be able to handle any excess data build-up that can potentially happen due to the aforementioned scenario.

### **3.5 Resulting Context**

Keeping the producers and the consumers decoupled from each other keeps the design flexible to handle future changes in data sources and/or business needs. This pattern when applied together with the “Multi Tiered Device Driver” makes way for a layered architecture in the application level code of the real time system. Please refer Figure 1.

*Figure 2: The Producer/Consumer or Publish/Subscribe pattern*



### **3.6 Related Patterns**

The “Multi-tiered” pattern discussed in Chapter 5 talks about a similar layered structured with the layers stacked vertically. The observer pattern [GHJV94] provides a mechanism to let the consumers know when a data that they are interested in has been produced and hence in some cases may be useful to implement the data bridge. However this pattern may need to be modified when no data being produced is a normal and accepted mode of operation by letting the consumer know the difference between no data being produced and something being wrong with the producer. It could do that by checking the health of the Producer during a large period of silence from it.

### **3.7 Known Uses**

Mark Grand talks about the Producer/Consumer design pattern in his book on Java programming for coordinating the asynchronous production and consumption of information objects [Grand02].

Stephen Morris talks about ‘Separation of concerns’ by implementing Producer-Consumer pattern. He calls the producer as ‘Publisher’ and the consumer as ‘Subscriber’ while suggesting the use of the Observer Pattern to implement the Data Bride [Morris05]

Rafael Stekolshchik on his website mentions the following know uses of the Producer-Consumer Pattern [Stekolshchik07]:

- Message-Driven Beans of the EJB 2.0 [Stearns01] allow a loose coupling between the Message Producer and the Message Consumer. Application example of this framework is the Publish/Subscribe Application [Almaer01], where Producer is a *Publisher* and Consumer is a *Subscriber*.
- Schmidt and Cranor in their a pattern Half-Sync/Half-Async have the asynchronous layer (Producer) pass messages to the synchronous layer (Consumer) through a message queue (Data Bridge) [VCK96, SSRB00]
- Java Message Service (*JMS*) programming model includes a *supplier* named MessageProducer and *consumer* named MessageConsumer. In an article on JMS and CORBA Notification Interworking [Trythall01], such a relation between *Notification Service* and *JMS* interfaces is used

## **4.0 Pattern: Multiplexed (MUX) ADC driver**

### **4.1 Context**

A real time data acquisition system may at times need to acquire data from an analog channel and convert the same to a digital format. Typically the hardware device that does this conversion is called the analog to digital converter or ADC. Often there may be several analog channels that need to be read and converted to a digital signal. In such cases these multiple analog channels are read through the same ADC device but in a predetermined fashion so that the system knows which channel's signal was just converted to digital format. This organization of multiple analog channels so that a single ADC device can periodically read them is called multiplexing. This pattern illustrates firmware implementation of data acquisition from an ADC device, which has several analog data channels, multiplexed through it.

There can be an entire pattern language developed just to handle various types of ADC data acquisitions that are possible. However for the purposes of this paper only a specific case of data acquisition from an ADC that samples one (selectable) channel at a time is being considered.

### **4.2 Problem**

Usually multiplexed ADC devices use interrupt driven I/O. The sequence in which data is read or written to the device has to be always in a specific order with only one data line to be read at a time. Hence the problem is to accomplish serial, periodic and interrupt-driven data acquisition over an ADC multiplexed channel. How do we convert a parallel stream of values into serially accessible separate data elements in a periodic fashion or how do we read those multiple channels using a serial interface?

### **4.3 Forces**

The reading of the multiplexed data channels from ADC has to be cyclic. After reading a particular channel the next data channel that needs to be read has to be set so it has enough time to stabilize before data is read from it the next time the ADC interrupt happens. The state of the read/write to the data channel has to be saved from one interrupt to another so that the interrupt handler knows which data channel it is reading.

### **4.4 Solution**

The solution is to setup a hardware based interrupts, one for periodically starting ADC data conversion ("Start ADC data conversion ") and the other for signaling that the conversion is done and consequently data is ready to be read ("ADC conversion done"). The first interrupt can typically be clubbed with the system timer interrupt so that every time there is a system timer tick there is new conversion started on the ADC. Of course this scheme will work only when the ADC conversion time is always less than time between two consecutive timer ticks on the system clock. If it is more then the ADC conversion can be started every other timer tick or a similar scheme like that.

Next, use a C/C++ switch statement in the interrupt handler for the "ADC conversion done" interrupt with some local static variables to preserve the state of the sequence in which the data channels have to be read. It is important to note that there is one state per channel that is to be

read. Figure 3 presents a flow chart of the MUX ADC driver. The index keeps track of the data channel to be read when the interrupt happens and is incremented by one before getting out of the interrupt handler to point to the next data channel that will be read. If the index value reaches the total number of channels (n) then it is reset so that the first channel can be read next and thus one cycle completes. Also a point to note is that after reading the current ADC data channel the next ADC channel that need to be read is set so that it gets some time to stabilize before it is read during the handling of the next ADC interrupt.

*Figure 3: ADC acquisition interrupt handler design pattern:*

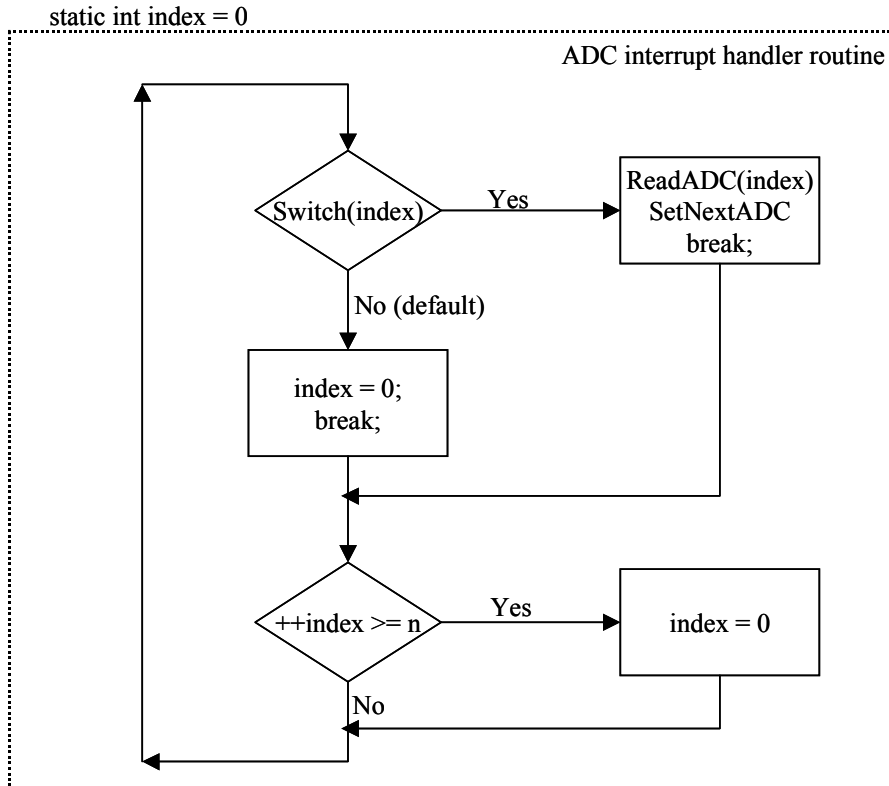


Figure 4 presents a sequence UML diagrams to explain the functioning of the pattern in more detail. It shows the sequence in which the interrupts happen and how they get handled. The interrupts are generated by the ADC resource when it's ready with the data to be read from it. This interrupt is handled by the handler, which implements the "MUX ADC driver" pattern. The sequence diagram shows two cycles in the interrupt sequence, which leads to data being read by the driver twice from the N channels that are multiplexed through the ADC.

Figure 4: ADC acquisition interrupt handler design pattern:

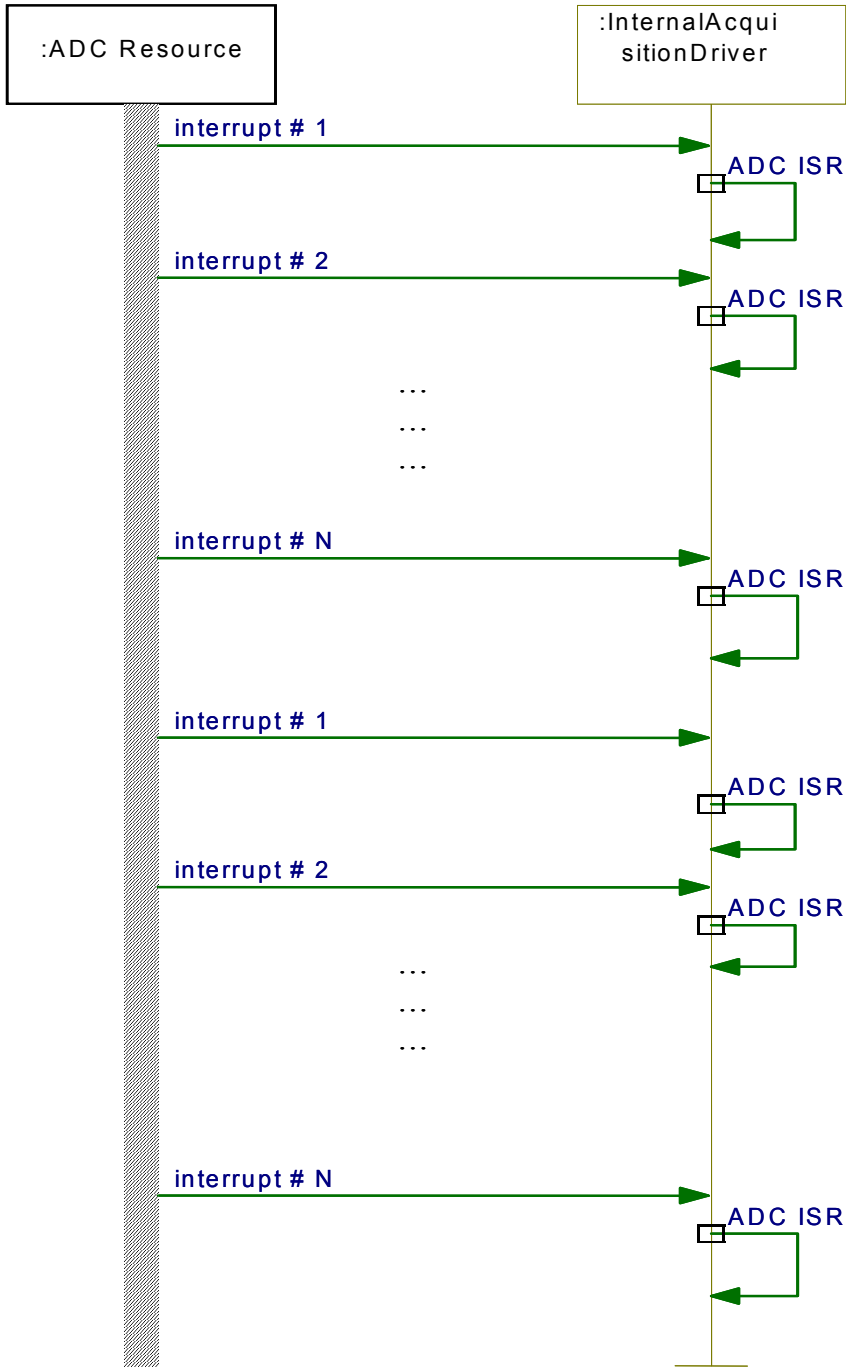
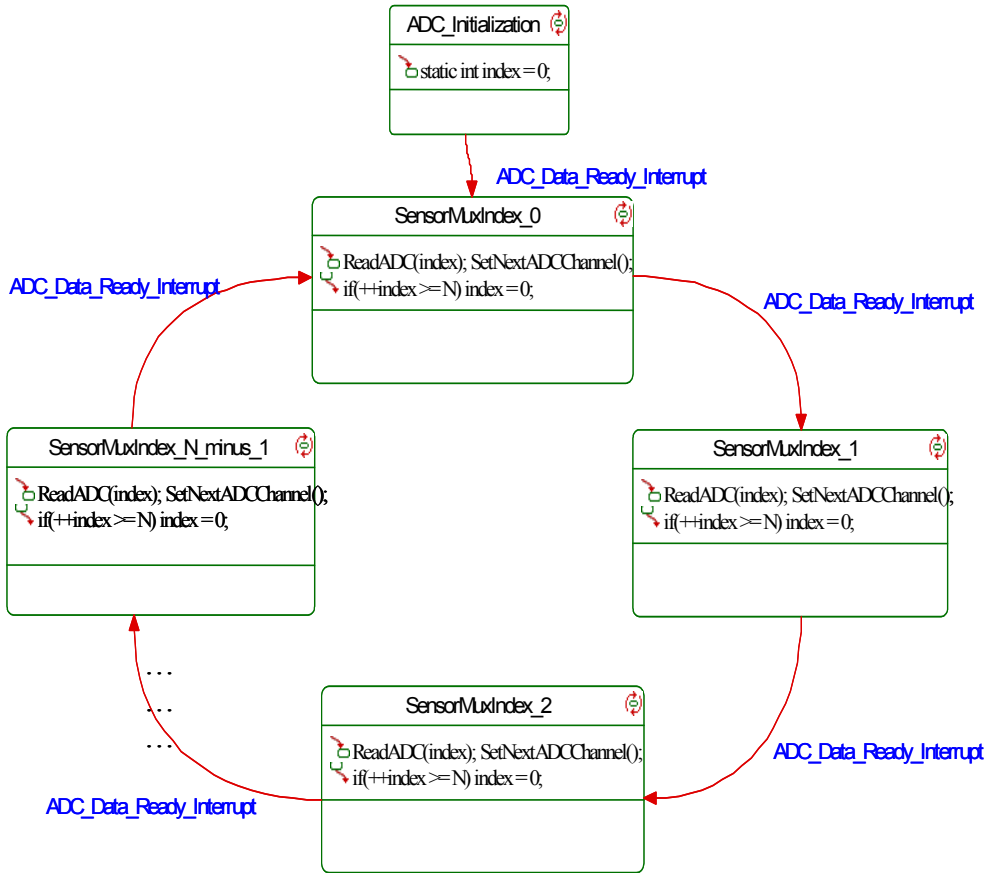


Figure 5 shows how the interrupt handler switched between different states depending on the “SensorMuxIndex” which is kept track by a local static variable in the ISR. Interrupt # N causes the ISR to be in “SensorMuxIndex\_Nminus1” state.

Figure 5: State Chart presenting the various states through which the MUX ADC driver cycles during data acquisition



The following code in Figure 6 presents a sample implementation of the pattern, where the total number of multiplexed ADC channels is 12. The handler for the timer interrupt that happens once every 10 msec has a dummy read call, which in turn starts the acquisition on the ADC. When the ADC finishes the data acquisition it fires an `ADC_DATA_READY` interrupt, which is then handled by a routine that implements the pattern. The initial ADC channel, `MUX_CHAN_0`, is set in a `ADCinit()` routine which called during the initialization of `MuxAdcDriver` class.

*Figure 6: Sample code showing implementation of the 'MUX ADC Driver' pattern implemented in the ADC\_DATA\_READY interrupt service routine*

```
//This happens outside inaterrupt handler during device
//initialization
static int SensorMuxIndex = 0;

//Interrupt handler implementing 'MUX ADC driver' pattern
void MuxAdcDriver::MuxAdcInterruptHandler(void)
{
    ...           ...           ...           ...
    ...           ...           ...           ...

    if(irq_source & ADC_INT_AVAIL) //ADC data available
    {
        switch(SensorMuxIndex)
        {
            case 0: //MUX_CHAN_0
                if(actel_stat & ADC_BUSY_PIN)
                    PreviousCHAN0 = Reg_ADC_DATA;
                //Set next MUX channel after clearing previous
                Reg_PORTF0 = ((Reg_PORTF0 & MUX_ADR_CLR) | MUX_CHAN_1);
                break;

            case 1: // MUX_CHAN_1
                if(actel_stat & ADC_BUSY_PIN)
                    PreviousCHAN1 = Reg_ADC_DATA;
                //Set next MUX channel after clearing previous
                Reg_PORTF0 = ((Reg_PORTF0 & MUX_ADR_CLR) | MUX_CHAN_2);
                break;

            ...           ...           ...           ...
            ...           ...           ...           ...
            ...           ...           ...           ...

            case 11: // MUX_CHAN_11
                if(actel_stat & ADC_BUSY_PIN)
                    PreviousCHAN11 = Reg_ADC_DATA;
                //Set next MUX channel after clearing previous
                Reg_PORTF0 = ((Reg_PORTF0 & MUX_ADR_CLR) | MUX_CHAN_0);
                break;

            default:
                SensorMuxIndex = 0;
                break;
        }
        if (++SensorMuxIndex >= 12)
            SensorMuxIndex = 0;
    }
}
```

#### **4.5 Resulting Context**

Data is read from the mux-ed channel in the desired sequence every time the concerned interrupt is handled. The switched statement and the local static index guarantees that the right ADC channel is read in the right sequence over and over again. This of course is going to be true if and only if the hardware device keeps functioning without a problem.

#### **4.6 Related Patterns**

There should be only one ADC driver per ADC device and this can be ensured by using the Singleton pattern [GHJV94]. Also the application level code does not need to know about the low level details along with the instantaneous data being collected by the ADC driver. Usually it uses some averaged or filtered value and this functionality can be encapsulated in an Adapter class [GHJV94], which provides an easy to use interface for accessing data from the ADC device.

#### **4.7 Known Uses**

MUX ADC Driver pattern has been widely used in Schlumberger's real time data acquisition firmware, which involved dealing with ADCs [SLB].

## **5.0 Pattern: Multi Tiered Device Driver**

### **5.1 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.

### **5.2 Problem**

A significant challenge in developing device drivers is to keep the design adaptable to future requirement changes and reusable. 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.

### **5.3 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 adaptability to future changes in requirements and code reuse 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.

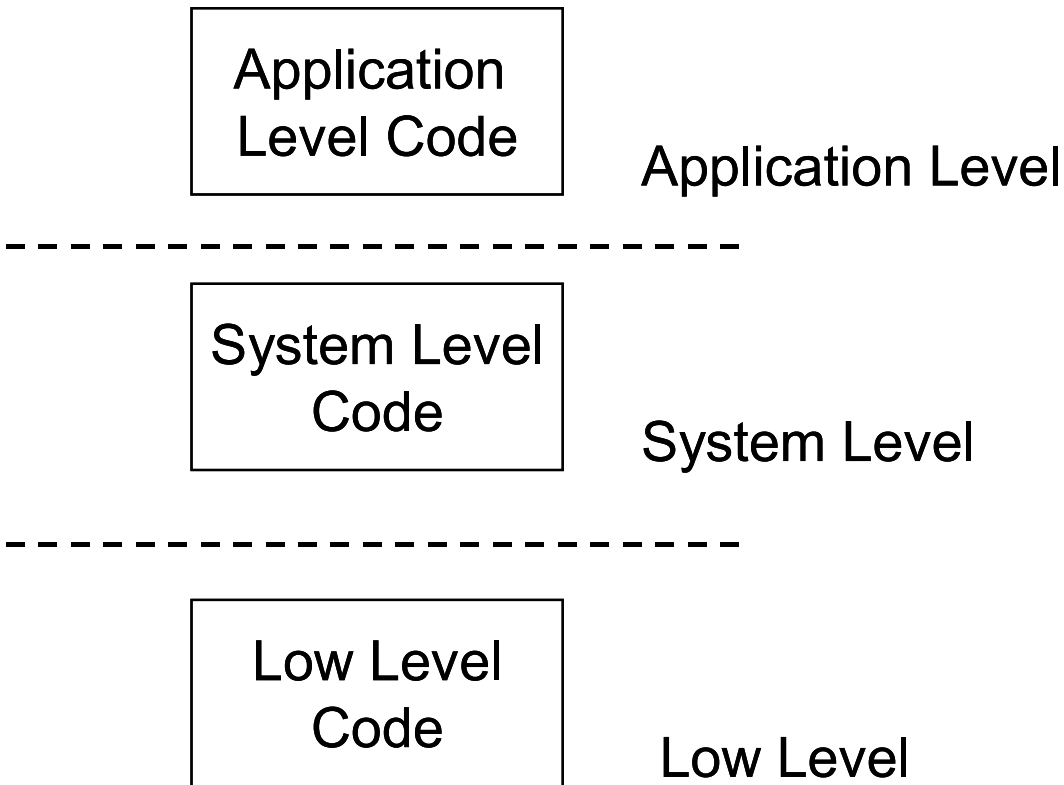
### **5.4 Solution**

Design a multi tiered architecture that divides the device driver code and the code that uses it 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 7.

### **5.5 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.

*Figure 7: Multi Tiered Architectural pattern for device driver design*



### **5.6 Related Patterns**

The device driver code uses the Singleton pattern [GHJV94] to guarantee 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. Buschmann et al. talk about the "Layers" pattern when discussing architectural patterns in their book [BMRSS96].

### **5.7 Known Uses**

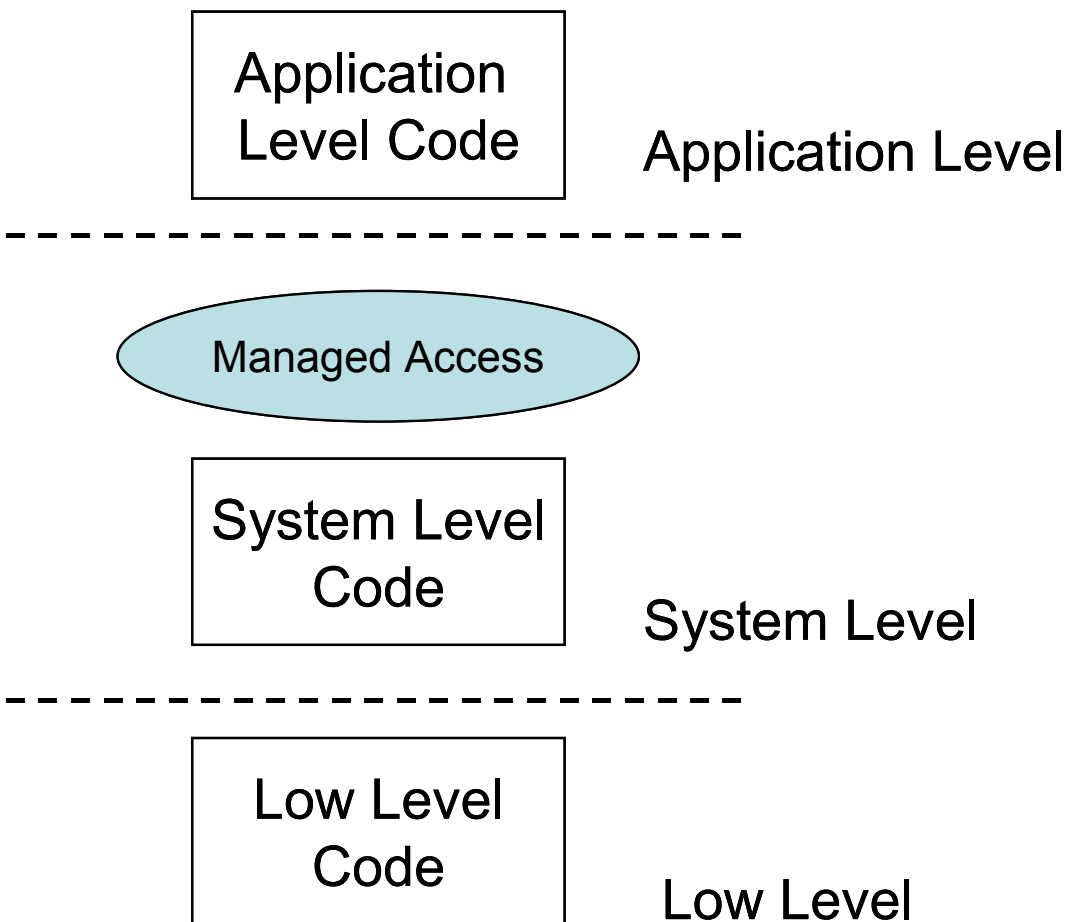
Barry Rubel [Rubel95] discusses the use of layered architecture in decomposing system requirements for mechanical control systems.

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

## 6.0 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/acknowledgment is not necessary or its simply too inefficient to wait for response/acknowledgment. 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 8: Multi Tiered Architectural pattern with Managed Access*



## **7.0 Pattern: Synchronous Managed Access**

### **7.1 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.

### **7.2 Problem**

Different application level objects may try to access the device at the same time and if not managed properly might think that the device is not functioning properly when they get erroneous or unexpected results. How can we implement managed access which involves waiting for response by application objects and no multiple requests to handle at the same time for the driver?

### **7.3 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.

### **7.4 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, mutexes, 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 9 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.

### **7.5 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.

### **7.6 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 from 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.

### **7.7 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 real time data acquisition 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].

## **8.0 Pattern: Asynchronous Managed Access**

### **8.1 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.

### **8.2 Problem**

Different application level objects may try to access the device at the same time and if not managed properly might think that the device is not functioning properly when they get erroneous or unexpected results. 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?

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

### **8.4 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 it 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 10 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 (system level code), as shown in Figure 11, to periodically poll the driver (low level code) 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 (low level code), as shown in Figure 12, informs the device adapter (system level code) 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 11 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 12 presents a typical sequence of events for this scenario.

### **8.5 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.

### **8.6 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.

### **8.7 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 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 9: Synchronous Managed Access for Input/Output

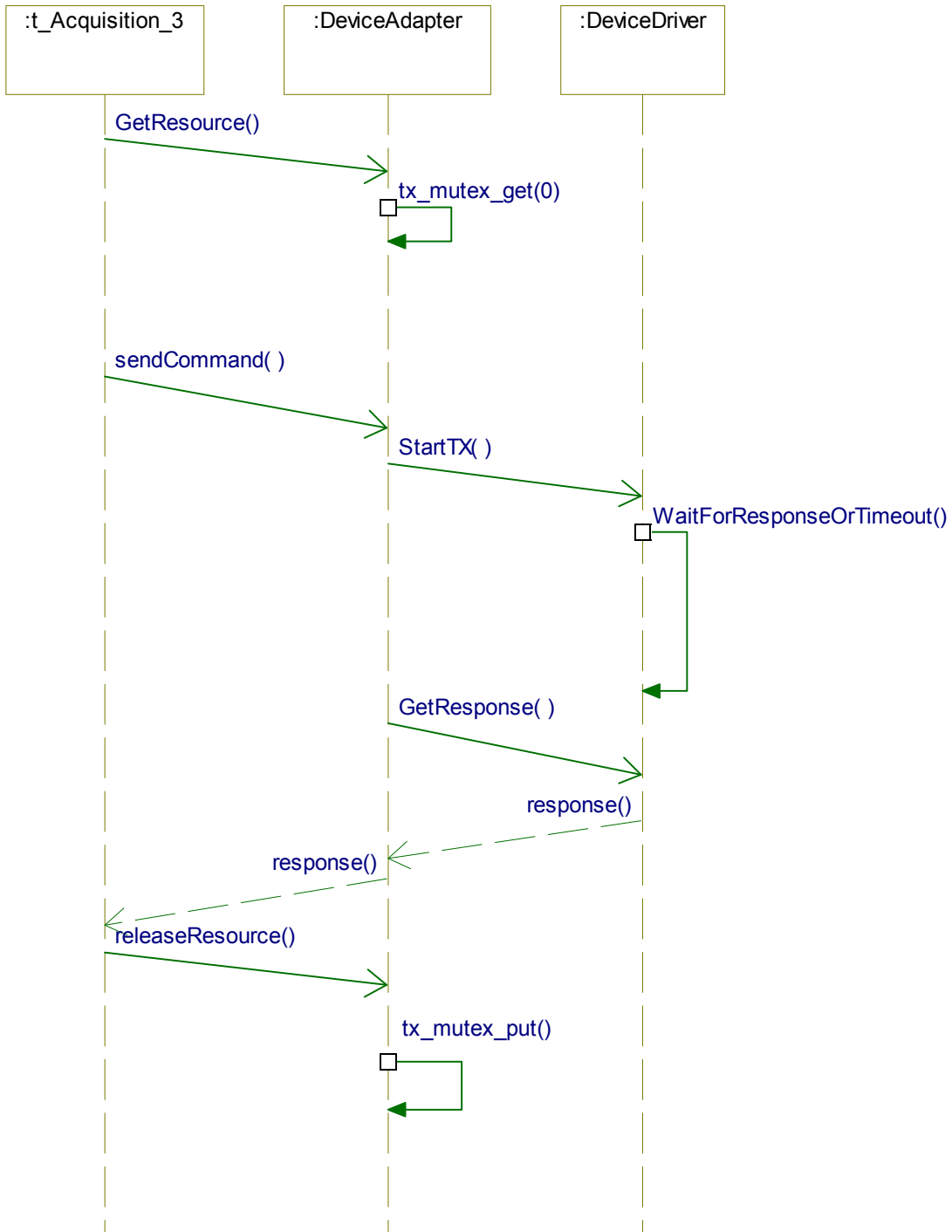


Figure 10: Asynchronous Managed Access for Output

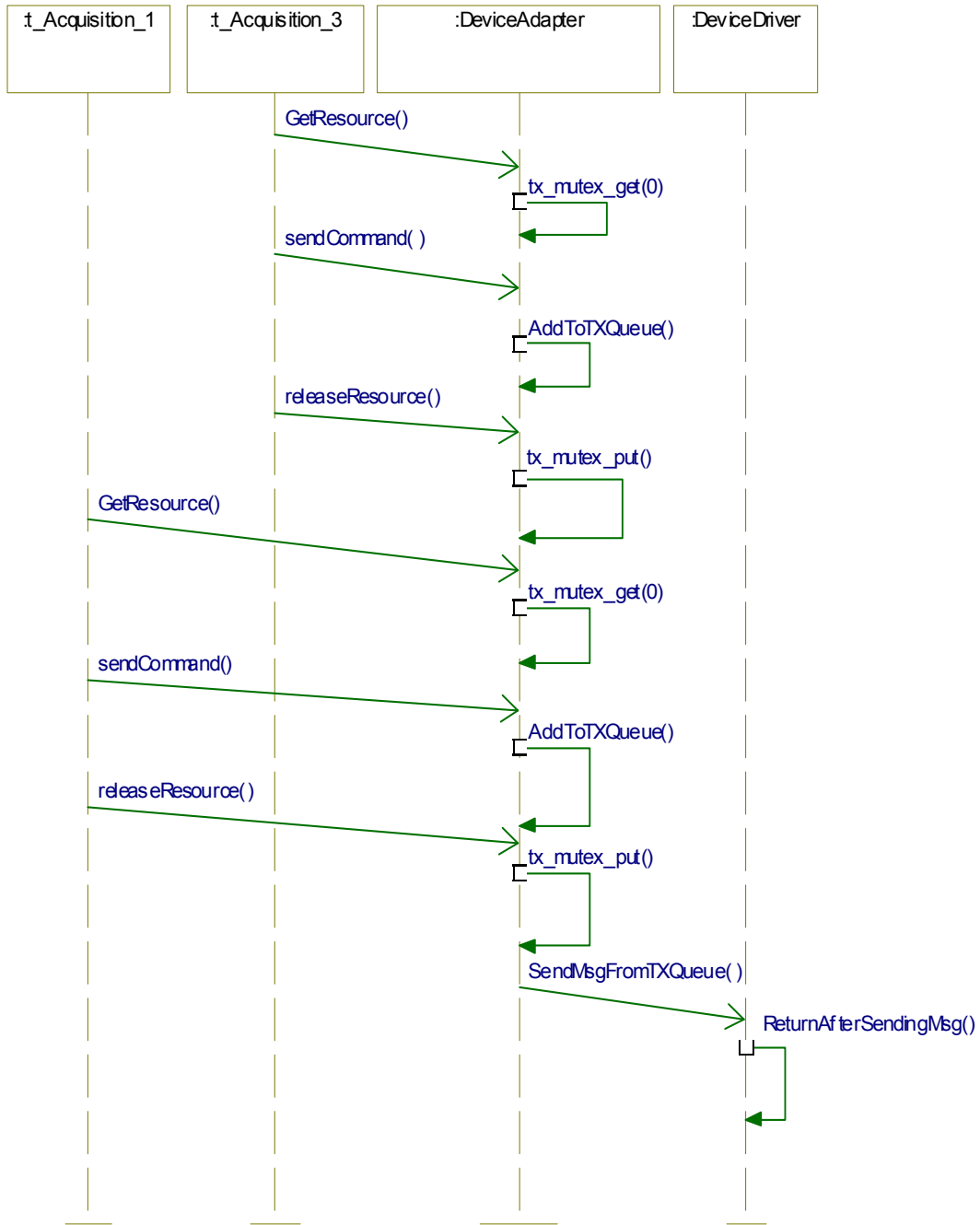


Figure 11: Asynchronous Managed Access for Input: Polling version

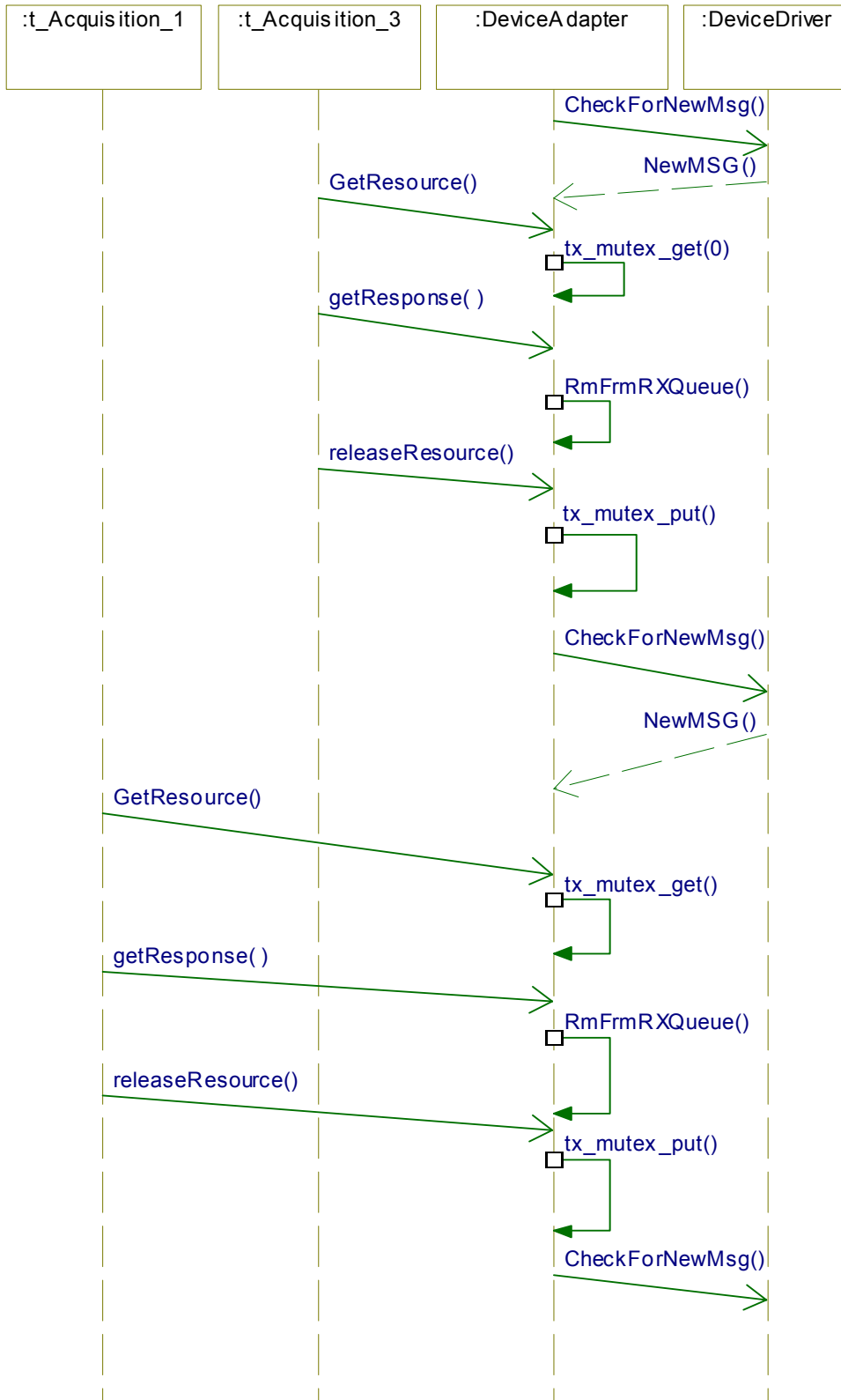
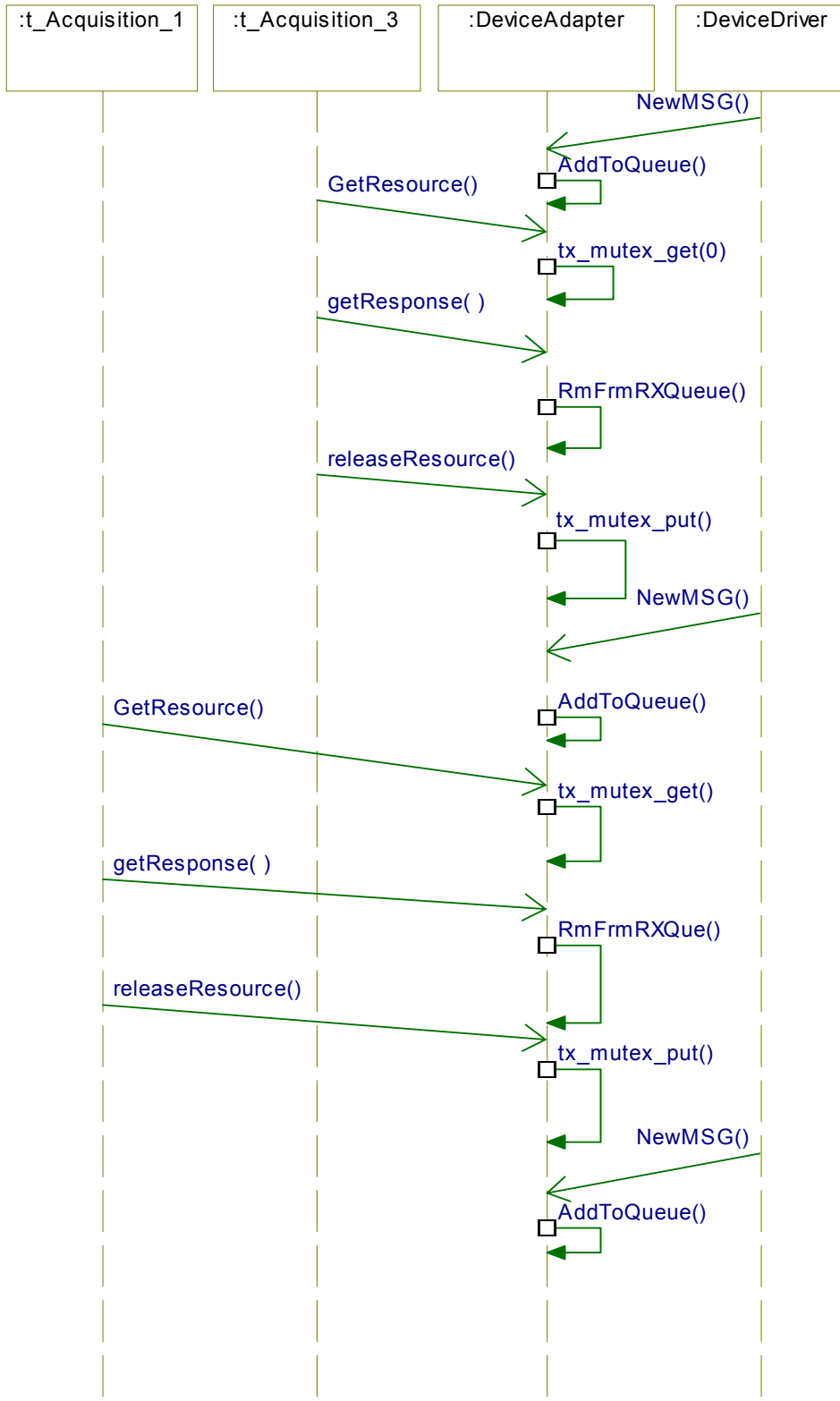


Figure 12: Asynchronous Managed Access for Input: Push version



## **9.0 Pattern: Friendship Zone**

### **9.1 Context**

Restricting access to the internal data buffers of the device drivers and other low level code is critical to prevent any other malicious code from accidentally corrupting them and consequently degrading the performance of the system. Encapsulating and having a separate abstraction layer for low-level code is the first step in this direction as shown in the “Multi Tier Device Driver” pattern. 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 especially in the low level code as presented in Chapter 5. This is a significant 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.

### **9.2 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.

### **9.3 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.

An approach is needed that allows objects in the low level code to maintain their data encapsulation but at the same time lets them allow greater access to their data to certain selected entities.

### **9.4 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. 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 9.3. This is because depending on the specific system requirements, proprietary hardware and the problem at hand; the relationships between the objects in the friendship zone can vary quite a bit. An example is presented in the author’s recent paper that was presented at PLoP 2006 [Bammi06]. In C, one could use global variables in the Friendship Zone for faster data access.

Some other things that can be considered to speedup things without breaking data encapsulation boundaries in the low level code are using in-line functions, no virtual member functions and using constant references in parameter passing so that copy constructor does not get called.

### 9.5 Resulting Context

The inefficiencies that can happen due to data encapsulation are addressed without having to compromise too much on data security. As a result in some cases it makes more sense to use both the “Multi Tier Device Driver” and the “Friendship Zone” patterns.

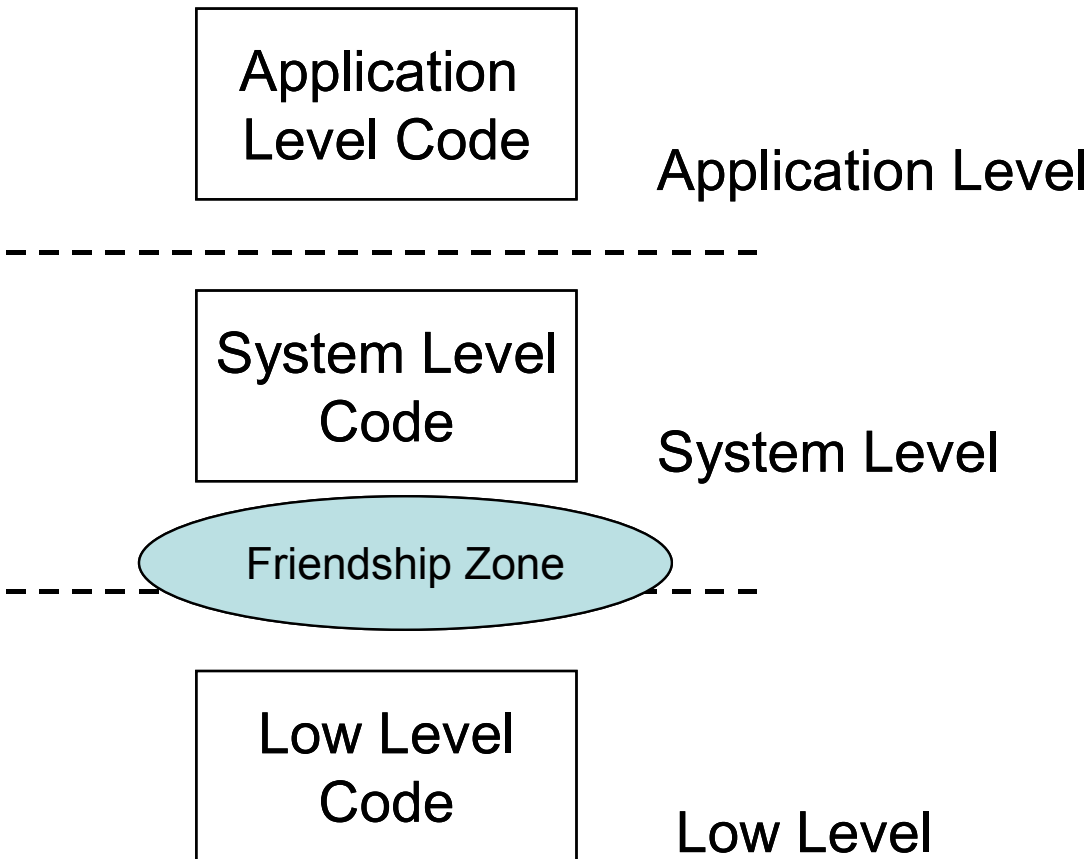
### 9.6 Related Patterns

Gamma et. al. in their 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].

### 9.7 Known Uses

In Schlumberger drivers for proprietary serial communication protocols in the real time data acquisition firmware implement this pattern [SLB].

*Figure 13: Multi Tiered Architectural pattern with Friendship Zone*



## 10.0 Pattern Thumbnails

Pattern	Intent
Producer/Consumer	Divide the system into “Data Producers”, “Data Consumers” and “Data Bridge”
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.
MUX ADC Driver	A common approach to sample Analog to Digital Converter (ADC) data from a multiplexed data acquisition channel.

## 11.0 Acknowledgements

The author would like to thank Dietmar Schuetz for shepherding this paper for EuroPlop 2007. The author would also like to express his gratitude for the help he received from James O. Coplien (shepherd), Lise Hvatum and the members of the writer’s workshop – “Intimacy Gradient” at PLoP 2006, for their valuable comments on the “Multi-Tiered Device Driver” and the “Friendship Zone” patterns.

## 12.0 References

1. [Almaer01] “EJB 2 Message-Driven Beans” at the O’Reilly on Java website, by Almaer, D. (May 2001), Link: [http://www.onjava.com/pub/a/onjava/2001/05/22/ejb\\_msg.html](http://www.onjava.com/pub/a/onjava/2001/05/22/ejb_msg.html) (accessed 12<sup>th</sup> January 2007)
2. [Bammi06] “Patterns for a Designing a Generic Device Driver for Interrupt Driven I/O”, Bammi S., presented at the Pattern Languages of Programming conference, Portland, Oregon, USA. (October 2006).
3. [BMRSS96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., “Pattern Oriented Software Architecture, Volume 1: A System of Patterns”, John Wiley & Sons; 1<sup>st</sup> edition, 1996.
4. [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.

5. [Ganssle01] Interrupt Latency, Ganssle, J. G. *Embedded Systems Programming*, VOL. 14 NO.12, October 2001. Link: <http://www.embedded.com/story/OEG20010918S0052> (accessed on 21<sup>st</sup> November 2006).
6. [GHJV94] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Boston, 1994.
7. [Grand02] *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, by Grand, M., Wiley 2nd Edition, Volume 1, Pg. 495 (September 17<sup>th</sup>, 2002).
8. [Horn93] Horn, C. "The Orbix Architecture," tech. rep., IONA Technologies, August 1993.
9. [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](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html) (accessed on 21<sup>st</sup> November 2006)
10. [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.
11. [Kalinsky06] Kalinsky, D., *Architectural Design of Device Drivers, Tutorial # ESC-505, Embedded Systems Conference 2006 San Jose – Silicon Valley*, D. Kalinsky Associates, 2006.
12. [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 21<sup>st</sup> November 2006)
13. [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.
14. [Morris05] "Publish and Subscribe using C++ and the Observer Pattern" by Morris, S. at the Pearson Education, Informit website (May 2005) Link: <http://www.informit.com/articles/article.asp?p=390393&seqNum=6&rl=1> (accessed 12<sup>th</sup> January, 2007)
15. [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 21<sup>st</sup> November 2006)
16. [Ritchie84] Ritchie, D. "A Stream Input–Output System," *AT&TBell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.
17. [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.
18. [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.
19. [SLB] Internal Schlumberger technical literature.
20. [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
21. [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<sup>st</sup> edition, 2000.

22. [Stearns01] “Migrating from EJB 1.1 to 2.0” on Sun Developer Network, by Sterans, B. (September 2001), Link: <http://java.sun.com/developer/technicalArticles/ebeans/ejbmigrate/> (accessed 12<sup>th</sup> January 2007).
23. [Stekolshchik07] The Producer-Consumer Applet, implemented and maintained by Stekolshchik. R. Link: <http://cities.lk.net/approco.html> (accessed 12<sup>th</sup> January, 2007)
24. [Trythall01] “JMS and CORBA Notification Internetworking” at the O’Reilly on Java website, by Trythall, S. (December 2001), Link: [http://www.onjava.com/pub/a/onjava/2001/12/12/jms\\_not.html](http://www.onjava.com/pub/a/onjava/2001/12/12/jms_not.html) (accessed 12<sup>th</sup> January 2007)
25. [VCK96] Vlissides, J., Coplien, J. and Kerth, N., eds. Pattern Languages of Program Design-2, Addison-Wesley, 1996.
26. [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.