

**Avtec Software Design Notes/Specification**

Title: AEF on Linux High Level Design Notes

Release: OutpostPlus

Author(s): Steve Yackey

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Revision Date | Revised by | Notes |
| .03 | 9/9/2017 | S. Yackey | Initial Version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 Introduction 2](#_Toc492567473)

[1.1 Problem Summary 2](#_Toc492567474)

[1.2 Proposed Solution Summary 2](#_Toc492567475)

[1.3 Refactoring 3](#_Toc492567476)

[1.4 Design Alternatives Considered 3](#_Toc492567477)

[1.4.1 Port it and Leave it alone 3](#_Toc492567478)

[1.4.2 Pimpl Idiom 3](#_Toc492567479)

[1.4.3 Factory Pattern(s) 3](#_Toc492567480)

[2 Design Overview 4](#_Toc492567481)

[2.1 Approach 4](#_Toc492567482)

[2.1.1 Original Class Composition Mechanism 4](#_Toc492567483)

[2.1.1.1 Possible Disadvantages 5](#_Toc492567484)

[2.1.2 Newer Approach 6](#_Toc492567485)

[2.1.2.1 Advantages (IMO) 8](#_Toc492567486)

[2.1.3 Porting Choices 9](#_Toc492567487)

[3 The Code 10](#_Toc492567488)

[3.1 Location 10](#_Toc492567489)

[3.2 So Just What Is It? 10](#_Toc492567490)

[3.3 Tests 10](#_Toc492567491)

[3.4 AEFApp 10](#_Toc492567492)

[3.4.1 taskLikeTestApp 10](#_Toc492567493)

[3.4.2 TimerTest 10](#_Toc492567494)

[3.4.3 pingPongTest 11](#_Toc492567495)

[3.4.4 taskAndQueueTask 11](#_Toc492567496)

[3.4.5 taskJustQueueTask 11](#_Toc492567497)

[4 TODOs 12](#_Toc492567498)

[4.1 Risks / Unknowns 12](#_Toc492567499)

[4.2 Compatibility 12](#_Toc492567500)

[5 Testing 12](#_Toc492567501)

# Introduction

## Problem Summary

The Avtec Embedded Framework (AEF) is utilized by Avtec’s embedded product Outpost. AEF provides base service abstractions that higher level software can build upon. Currently, AEF supports operation in a VxWorks environment.

OutpostPlus is intended to extend the capability of Outpost to a more robust platform. Additionally, other Avtec products MAY also reside on an OutpostPlus platform.

Currently, evaluation is underway to determine whether to a) continue utilizing VxWorks, albeit a more modern version b) move to a Linux based implementation.

* A VxWorks decision would mean an upgrade/enhancement.
* A Linux decision would mean a port.

This document describes an initial pass at a port to Linux.

The VxWorks vs. Linux decision remains outstanding. And which Linux distribution (exactly) remains outstanding.

## Proposed Solution Summary

This was a ‘straight port’ to a target of Ubuntu 17.04 – desktop (but that really does not matter).

The exact choice of Linux distribution is of low importance, as long as it supports gcc.

The initial port utilizes Netbeans 8.2 IDE.

The exact choice of IDE is of moderate importance. IDE capability effects developer productivity and ultimately quality. Like if it doesn’t have a decent hook into gdb for easy usage, you lose a lot; intelli-sense; that kind of stuff.

In addition to a port of functionality to a Linux platform, the code base was moved to a more object oriented friendly solution. The original code base is fine, but additional OO design changes could make it more easily used between two operational environments: VxWorks and Linux if that is ever to be the case.

In this regard, the AEF base code was changed to accommodate a standard inheritance model and a new ‘Linux’ flavor software tree was added.

The thought is that the ‘base’ would be utilized, unchanged, by a VxWorks flavor.

And finally, the application, Outpost, would be unchanged in its utilization of AEF. This was a major goal in this choice – but there are no real requirements in this regard.

## Refactoring

All of the AEF ‘base’ code was refactored, making each former concrete class an abstract base class.

The ‘Linux’ flavor code is new, so that is not strictly a refactor. The flavor(s) provide the concrete behavior.

The next refactoring will move common, concrete behavior(s) into the base classes. This should provide a clear view of what is ‘flavor’ specific for adapting VxWorks based flavors to this model. Note: there is absolutely no requirement, plan, need that I know of to ‘back-port’ any of this.

## Design Alternatives Considered

### Port it and Leave it alone

Just port the code and leave the ‘structure’ alone. The original code utilized a composition model utilizing pre-processor directives to provide a ‘flavor’. This is a perfectly fine approach. The only drawback was that there was platform specific code in the ‘non-flavor’ classes/methods. So figuring out exactly what needed to change and where was not as clean as it could have been.

### Pimpl Idiom

One attempt was to utilize the Pimpl idiom to hide the flavor specifics.

The drawback here was that this composition model, required too much developer knowledge of ‘who owned which’ method/member variable. With this form of composition, you need to navigate using the implementation pointer. If an item in the implementation class needed to access something in the parent, then you’d access that through a pointer to the parent object.

These types of references were replete in the code base, and it quickly became obvious that too much navigational knowledge was required.

### Factory Pattern(s)

There are many ways to implement this pattern. Generally speaking, a flavor would inherit from an interface and a factory would create a ‘generic’ object. That object would actually be of a given ‘flavor’ because the factory made it appropriately.

This is not all that far from the abstract base class method chosen.

The drawback was that I wanted to try and keep the application interface unchanged. That means constructors rather that a ‘make’ factory method.

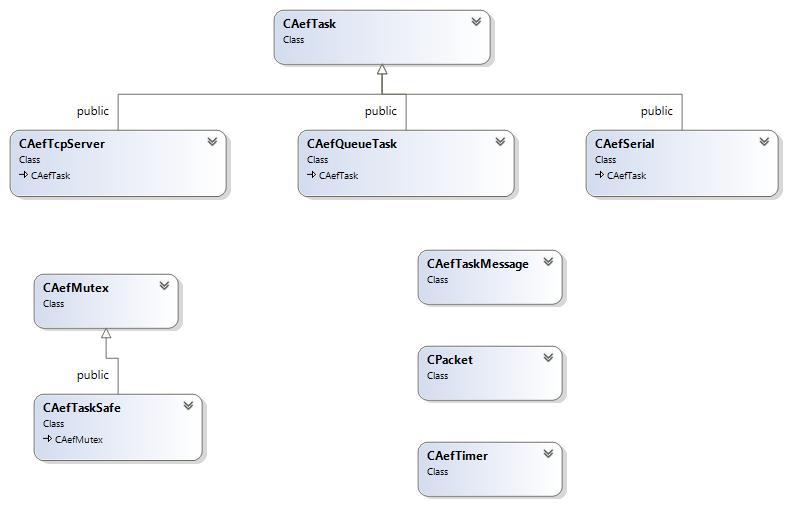
# Design Overview

## Approach

Note: Side-by-Side class diagram view is [here](http://pd-wiki.avtecinc.com/Engineering/Developer_Pages/Steve_Yackey/AEF_Class_Diagram_-_Original_vs_New) (unexciting, but it didn’t fit here).

### Original Class Composition Mechanism

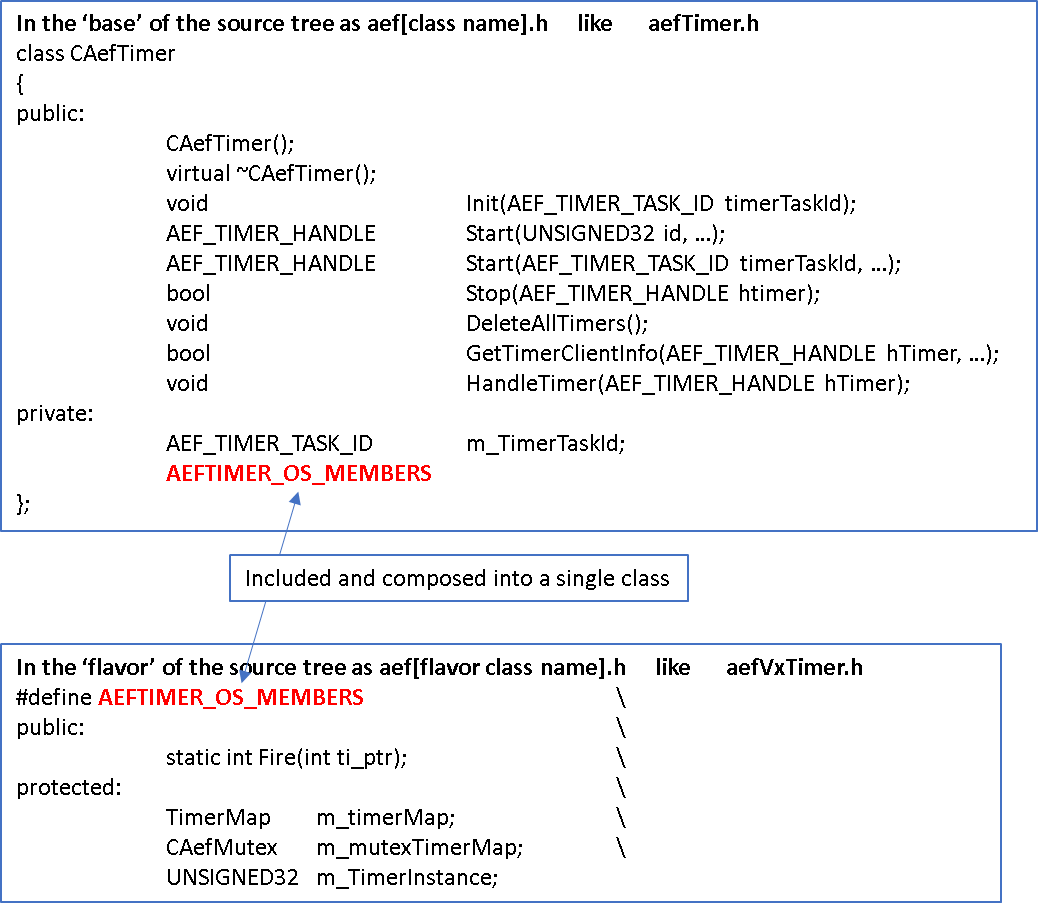
The original/existing class structure looks like this:



This is a very simple, effective modeling.

How exactly was this accomplished?

I’ll pick on CAefTimer – it is unremarkable, i.e., typical.



The ‘OS flavor’ was composed at pre-processor time.

There was a single cpp file that implemented a given class.

There is nothing wrong with this method.

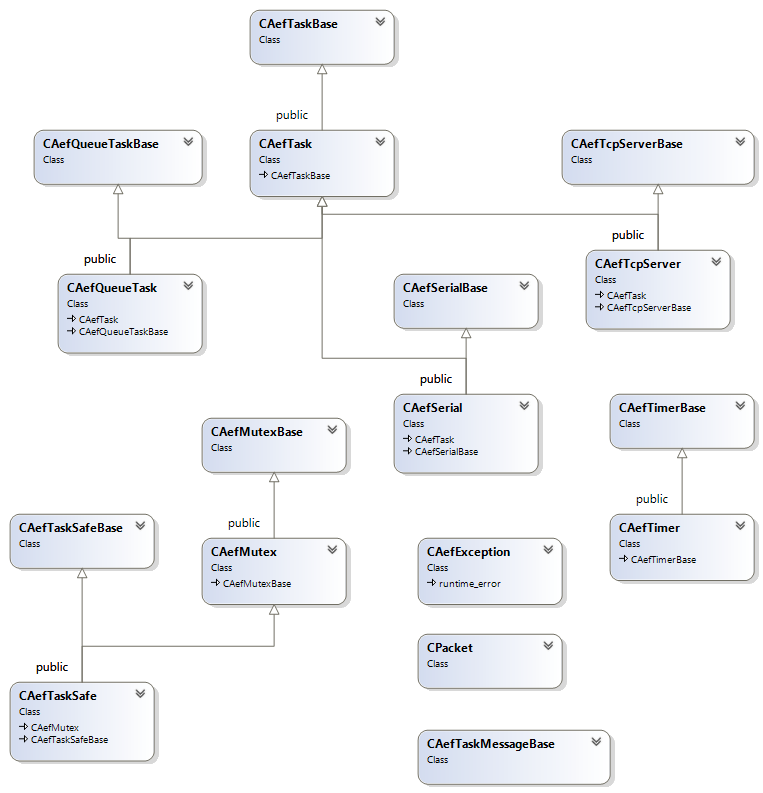
#### Possible Disadvantages

In some cases, however, there was OS specifics in the methods defined in the ‘base’ methods. But this has nothing to do with the include mechanism per se.

The ‘inheritance model’ could possibly be improved such that it is clearer what constitutes common code vs. OS specific code.

### Newer Approach

The new class structure looks like this:

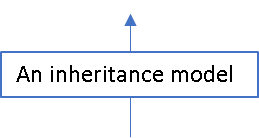


This is a more involved model. This seems effective in practice, but more scrutiny is needed in practice and in theory (by AFE SME’s) to be certain.

How exactly was this accomplished?

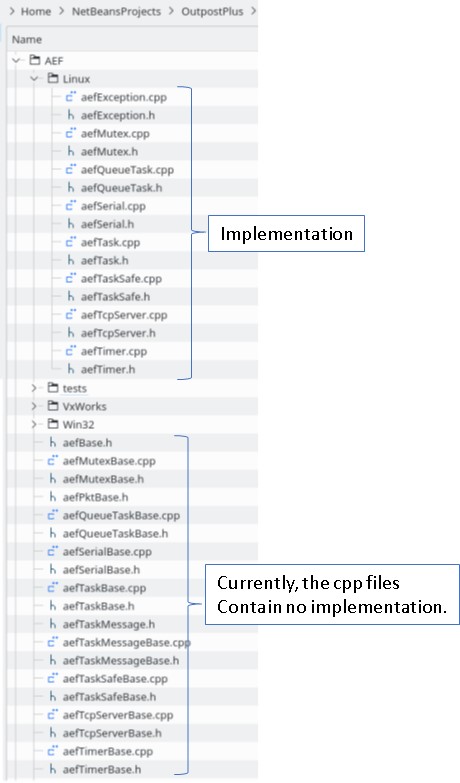
I’ll pick on CAefTimer – it is unremarkable, i.e., typical.







Kind of a summary look at the directory structure level.



#### Advantages (IMO)

As of this instant in time, the abstract base class has no code. However, a refactoring will move common code into the base class. This should provide a clear demarcation of OS specifics.

The derived class (continuing picking on the timer class) is known as CAefTimer. This should allow application users to operate unchanged on the various OS platforms.

This is from the AEF perspective. If there is OS specific code in an application unrelated to AEF, well that cannot be solved here.

And that is the gist of the new approach – for all AEF classes.

### Porting Choices

So, anything can be ported to anything. And it can actually be made to work. Changes in the class model are important, but so are the choices made to replace OS specific operations.

Note: I hope I didn’t leave anything out. Trust me, I made changes – even if I forgot (sad).

|  |  |  |
| --- | --- | --- |
| VxWorks | Linux | Notes |
|  |  |  |
| semLibs |  |  |
| semM | pthread\_mutex[[1]](#footnote-1) | Provides recursive, priority inversion etc |
| semB | Posix semaphores[[2]](#footnote-2) | Straight match |
| msgQLib |  |  |
| msgQSend et al | Posix message queues[[3]](#footnote-3) | “POSIX message queues provide a better designed interface than System V message queues; on the other hand POSIX message queues are less widely available (especially on older systems) than System V message queues.”  Older systems should not concern us. |
| wdLib |  |  |
| wdStart et al | Posix timer[[4]](#footnote-4) |  |
| taskLib |  |  |
| taskSpawn | Posix pthread\_create etc[[5]](#footnote-5) | Could have used the new std based stuff. But the Posix items seemed more convenient is terms of specifying attributes. |
| serial |  |  |
| ioctl | termios/tcsetattr[[6]](#footnote-6) | Standard |
| tcp |  |  |
| No real issues | Ordinary sockets |  |
| Ancillary items |  |  |
| ticks to/from MS  TicksFromTOD  … | Started with the basic value from  sysconf(\_SC\_CLK\_TCK)  and went from there |  |

For the most part, I chose the Posix solution. That seemed to provide a coherent set of functionalities, conveniently packaged.

# The Code

## Location

The code lives at:

<http://pd-svn/svn/software-engineering/Common/AEF/Branches/linux_work>

Code in the repository always works. Where works means:

* Builds
* Unit tests run
* Higher level tests run

## So Just What Is It?

I would characterize the state as initial prototype or maybe proof of porting concept.

It is certainly not complete, and has lots of comments and TODOs in the code. On the other hand, I didn’t do anything knowingly wrong or take any shortcuts.

So whatever it is, it’s my best shot to-date (continual refactor).

A snap-shot of an approach I would say.

## Tests

Included in the AEFLib project are **unit tests**.

These are always being worked on/added to, but what ever they are, they always work.

## AEFApp

This project utilizes AEFLib and performs ‘higher’ level tests.

### taskLikeTestApp

Uses tm\_req\_init(1).

This is taken from opTestAp in the Outpost\App\src tree and Runs the CTestAp.

The TcpServer is configured.

I connect from my old tcp server test program (on my windows box). And we send and receive some packets.

This does two important things:

* Utilizes the TcpServer class
* Demonstrates that application code runs unchanged, not noticing changes in the underlying AEFLib.

### TimerTest

Uses a class derived from CAefQueueTask.

Starts 6 timers, various durations, some periodic some one-shot.

The watchdog timer is running, also.

One timer, a 30 second, one-shot signals the test end.

Upon receipt of the ‘end of test’ timer message, a separate message is sent that signals that the timers stop.

This does two important things:

* Utilizes the timer class
* Demonstrates that application code runs unchanged, not noticing changes in the underlying AEFLib.

### pingPongTest

Is derived from CAefQueueTask and uses a pair of classes derived from CAefQueueTask.

So we have a pingPongTest and 2 pingPongWorkers.

The ‘test’ starts timers just like timer test and also starts the workers.

When the 30 second timer message is received, a stop message is sent to each worker.

During this 30 seconds, each worker responds to a 2 second timer. When the 2 second timer expires, a message is sent to the ‘other’ ping pong worker. Each prints message arrival information.

This continues until a stop message is received and Exit is called.

This does several important things:

* Utilizes the timer class
* Utilized message queues – not a stress test, but more than any other test
* Utilized the task/queue mapping
* Demonstrates that application code runs unchanged, not noticing changes in the underlying AEFLib.
* And was so unbelievably cool to see just a little of what this stuff could do I almost gave out a whoop !!!

### taskAndQueueTask

Of limited usefulness right now.

One of my first tests (I knew even less then – so it needs work).

Constructs a task providing the name of a QueueTask previously created.

The ‘run’ method of the task simply sends a message to the ‘parent’. When the message is received a print statement is emitted.

This does some important things:

* Utilizes the Task class ‘run’ – not earth shattering but …
* Demonstrates that application code runs unchanged, not noticing changes in the underlying AEFLib.

### taskJustQueueTask

Of limited usefulness right now.

Another of my very first tests.

Just sends a message to a QueueTask, the receipt of which is noted ala printf.

This does almost no important things (but it will grow into a giant):

* Utilizes QueueTask message traffic (it was a big deal then)
* Demonstrates that application code runs unchanged, not noticing changes in the underlying AEFLib.

# TODOs

What I think my priorities are.

* Add more unit tests
* Put the higher level tests into the projects CppUnit framework – so they’ll be easier to run
* Take a shot at brewing up a makefile that makes the Outpost code, i.e., morph the VxWorks build into a makefile
  + I am uncertain why I should not be able to produce a real live Outpost on my little Linux box.
    - Ignorance is bliss 😊
* Move code back into the base classes
  + For some reason, I think this will not that hard

## Risks / Unknowns

One risk is that someone might actually think something is done. Rather than something might be able to begin.

Unknowns – I don’t even know.

## Compatibility

I need to get a VxWorks workbench and see about compatibility.

# Testing

Yes !!

1. <https://linux.die.net/man/3/pthread_mutex_init> [↑](#footnote-ref-1)
2. <http://man7.org/linux/man-pages/man7/sem_overview.7.html> [↑](#footnote-ref-2)
3. <http://man7.org/linux/man-pages/man7/mq_overview.7.html> [↑](#footnote-ref-3)
4. <http://man7.org/linux/man-pages/man2/timer_create.2.html> [↑](#footnote-ref-4)
5. <http://man7.org/linux/man-pages/man3/pthread_create.3.html> [↑](#footnote-ref-5)
6. <http://man7.org/linux/man-pages/man3/termios.3.html> [↑](#footnote-ref-6)