Preface

This component is intended to provide dynamic allocation from a pre allocated array. So it is like malloc but with static allocation. Its need was felt during the keypad project. Each button required a separate timer to keep record of the states. So atleast 16 timers for a 4\*4 keypad. But no of timers should be as much as the concurrent presses supported by the component. So if it is set to have just 4 concurrent presses (through KConfig), then only 4 timers are enough. But those timers will then be shared by 16 buttons. And when a timer is used by a button it cannot be shared by the other. It is because the usage includes setting the alarm time and then waiting for it to fire, so only one button should have control of the timer object. So for this purpose a scheme is devised to create n timer objects at the start of application, then throughout the execution , the button objects allocate and deallocate from them. Concurrency adds complexity.

# Pool Allocator Design

## Introduction

In this chapter the details about the design will be documented

## Methodology

Since it is a single component, so it seems enough to start with directly defining the interface first, and then implementing it. The object struct will be kept private and only the pointer to the interface member will be provided at the time of creation. The allocation will be provided from a local array stored in the file while keeping the totol\_allocated count, thus static allocation

## Design

### Interface

The interface provides simple functions to allocate and deallocate from the pool  
  
struct pool\_alloc\_interface{

void\* (\*poolDrain)(struct pool\_alloc\_interface\* self);

    int (\*poolFill)(struct pool\_alloc\_interface\* self,void\*\* object);

};

The allocate function ‘poolDrain’ provides a void pointer from the list/queue and also removes it from the list  
The deallocate function accepts a void pointer to a void pointer, This way it not just adds the pointer to the memory but also assigns NULL to the provided pointer so that user no longer

### Implementation

Since the component will be accessed concurrently, so care must be taken. For this reason, Freertos xQueue is used because it supports MPMC , but with blocking. So it must be taken care of that the users must have a predictable time. The average time depends upon the total concurrent access, and thus for the button/timer case total concurrent buttons detectable.